Specialization of Concurrent Guarded Multi-Set Transformation Rules

Thom Frühwirth

Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. Program transformation and in particular partial evaluation are appealing techniques for declarative programs to improve not only their performance. This paper presents the first step towards developing program transformation techniques for a concurrent constraint programming language where guarded rules rewrite and augment multi-sets of atomic formulae, called Constraint Handling Rules (CHR).
We study the specialization of rules with regard to a given goal (query). We show the correctness of this program transformation: Adding and removing specialized rules in a program does not change the program's operational semantics. Furthermore termination and confluence of the program are shown to be preserved.

1 Introduction

Program transformation [PP96] is understood as a sequence of program text transformations that preserves semantic equivalence but at the same time improves the run-time, space-consumption or other aspects of the given program. *Partial evaluation* [MS97] is a popular instance of program transformation and of *program specialization*, which optimizes a given program for known values of the input.

Program transformation goes especially well with declarative (functional, logic, constraint) programming languages due to their clean semantics (avoidance of side-effects).

In the rule-based CHR language, we are interested in program specialization with regard to a given goal (query). We consider the rules that are applicable to the goal in any possible context (state of computation). We would like to *specialize these rules for the given goal*.

Our work is motivated by a renewed (and as we think, increasing) interest in program transformation and by the unique combination of features that the CHR language offers, in particular the multi-set programming style and the so-called propagation rules that add information without removing any. On one hand, these features mean that we have to adopt existing program transformation techniques for them or even come up with new ones, and on the other hand, there is hope that they make certain program transformations more straightforward.

We now discuss the appeal of program transformation and the special features of the CHR language in more detail.

Appeal of Program Transformation. Program transformation, and in particular rule specialization, have potential applications in the following areas:

- Using the specialized rules at run-time should *increase time and space efficiency.*
- In concurrent languages like CHR, we also can *eliminate communication channels, synchronization points and don't care nondeterminism* [EGM01].
- *Verification and model checking* can be done by program transformation [DP99,FPP01,RKRR04].
- *Agent can be specialized* to a specific context (Example in [EGM01]).
- *Constraint solving can be improved*, since particular classes of optimization problems like scheduling typically have a certain structure [Wal03].
- A complete set of specialized rules can be regarded as *conditional or qualified answer* for the user.
- *Agent communication can be improved* by exchanging conditional answers [PGGS98].

Constraint Handling Rules (CHR). In *constraint solving*, efficient special-purpose algorithms are employed to solve sub-problems involving distinguished relations referred to as constraints. CHR [Frü98] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints into simpler ones until they are solved.

In CHR, one distinguishes two kinds of rules: Simplification rules replace constraints by simpler constraints while preserving logical equivalence, e.g. $\texttt{X}{\geq}\texttt{Y} \wedge \texttt{Y}{\geq}\texttt{X} \Leftrightarrow \texttt{X=Y}$. Propagation rules add new constraints, which are logically redundant, but may cause further simplification, e.g. $\texttt{X}{\geq}\texttt{Y} \wedge \texttt{Y}{\geq}\texttt{Z} \Rightarrow \texttt{X}{\geq}\texttt{Z}$. The combination of propagation and multi-set transformation of logical formulae in a rule-based language that is concurrent, guarded and constraint-based make CHR a rather unique declarative programming language.

Typically, CHR programs are *well-behaved*, i.e. terminating and confluent. *Confluence* means that the result of a computation is independent from the order in which rules are applied to the constraints. Once termination has been established [Frü02], there is a decidable, sufficient and necessary test for confluence [Abd97,AFM99].

Related Work. Since CHR can be seen as an extension of concurrent constraint programming (CCP) [SR90] by multiple heads (multi-sets) and propagation rules, literature on program transformation for concurrent constraint and logic-based programming languages is relevant: [EGM01] deals with transformations of concurrent constraint logic programs, [FPP04] deals with constraint logic programs (CLP), and [UF88] deals with a guarded concurrent logic programming language called GHC.

Due to propagation rules and the multi-set character of CHR, the above results are not directly applicable. For example, multiple heads mean that unlike CCP, constraints are usually defined by several rules, and that unlike CCP and GHC, different constraints can be defined in one rule by their interaction. GHC

lacks built-in constraints and thus does not feature guard checking by logical implication. CLP among other things lacks concurrency. Moreover, these related works are not concerned directly with rule specialization, but with unfold/fold transformations.

Outline of the Paper. In Section 2, we define the CHR programming language. Section 3 introduces rule specialization. The next section shows correctness by considering specialized rules as redundant rules. The section also shows preservation of well-behavedness. Before we conclude, Section 5 gives some more examples.

2 The CHR Language

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [Frü98,FA03]. Readers familiar with CHR can skip this section (except for the introduction of the running example `max` maybe).

2.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in constraint symbols and CHR constraint symbols (user-defined symbols).

Built-in constraints are handled by a given, predefined constraint solver. We assume that these solvers are well-behaved (terminating and confluent). Built-in constraints include $=$, *true*, and *false*. The semantics of the built-in constraints is defined by a consistent first-order *constraint theory* CT. In particular, CT defines $=$ as the syntactic equality over finite terms.

CHR (user-defined) constraints are defined by a CHR program.

Definition 1. A *CHR program* is a finite set of rules. There are two kinds of rules:

A *simplification rule* is of the form

$$\mathit{Name} \mathbin{@} H \Leftrightarrow C \mid B.$$

A *propagation rule* is of the form

$$\mathit{Name} \mathbin{@} H \Rightarrow C \mid B,$$

where *Name* is an optional, unique identifier of a rule, the *head* H is a non-empty conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a conjunction of built-in and CHR constraints. A trivial guard "*true*" can be omitted together with "$\mid$".

A CHR symbol is *defined* in a CHR program if it occurs in the head of a rule in the program.

Example 1. Let $\leq$ and $<$ be built-in constraint symbols with the usual meaning. We define a CHR symbol `max`, where `max(X,Y,Z)` means that `Z` is the maximum of `X` and `Y`:

$$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}=\texttt{Y}.$$
$$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}=\texttt{X}.$$
$$\texttt{max(X,Y,Z)} \Rightarrow \texttt{X}\leq\texttt{Z} \wedge \texttt{Y}\leq\texttt{Z}.$$

The first rule states that `max(X,Y,Z)` is logically equivalent `Z`=`Y` if `X`$\leq$`Y`. Analogously for the second rule. The third rule states that `max(X,Y,Z)` unconditionally implies `X`$\leq$`Z` $\wedge$ `Y`$\leq$`Z`.

Note that `max` *will be our running example throughout this text.*

2.2 Operational Semantics of CHR

The operational semantics of CHR is given by a transition system.

Let P be a CHR program. We define the transition relation $\mapsto_P$ by introducing two computation steps (transitions), one for each kind of CHR rule (cf. Figure 1). Since the two computation steps (transitions) are structurally very similar, we first describe their common behavior and then explain the difference.

In the figure, all meta-variables stand for (possibly empty) conjunctions of constraints. C and D stand for built-in constraints only, H and H' for CHR constraints only.

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall\, (D \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G \wedge D) \mapsto_P^{\textbf{Simplify}} (G \wedge D \wedge B \wedge C \wedge H{=}H')$

Propagate

If	$(H \Rightarrow C \mid B)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall\, (D \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G \wedge D) \mapsto_P^{\textbf{Propagate}} (H' \wedge G \wedge D \wedge B \wedge C \wedge H{=}H')$

Fig. 1. Computation Steps of Constraint Handling Rules

A state is simply a goal, i.e. a conjunction of built-in and CHR constraints. Conjunctions are considered as multi-sets of conjuncts (conjuncts can be permuted). We will usually partition a state into subconjunctions of specific kinds of constraints. For example, any state can be written as $(H' \wedge G \wedge D)$, where H' contains only CHR constraints, D only built-in constraints, and G arbitrary constraints. Each of the subconjunctions may be empty (equivalent to *true*).

A (fresh variant of a) rule is *applicable to a state* $(H' \wedge G \wedge D)$ if H' matches its head H and its guard C hold when the built-in constraints D of the state

hold. A *fresh variant* of a rule is obtained by renaming its variables to fresh variables, $\bar{x}$.

Matching (one-sided unification) succeeds if H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. Matching is logically expressed by equating H' and H but existentially quantifying all variables from the rule, $\bar{x}$. This equation $H'{=}H$ is shorthand for pairwise equating the arguments of the constraints in H' and H, provided their constraint symbols are equal.

If an *applicable rule is applied*, the equation $H{=}H'$, its guard C and its body B are added to the resulting state. Any of the applicable rule can be applied (don't care non-determinism). A rule application cannot be undone (CHR is a committed-choice language without backtracking).

When a simplification rule is applied in the transition **Simplify**, the matching CHR constraints H' are removed from the state.

The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the resulting state. Trivial non-termination caused by applying the same propagation rule again and again is avoided by applying it at most once to the same constraints [Abd97].

A *computation* of a goal G in a program P is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto_P S_{i+1}$ beginning with the initial state $S_0 = G$ and ending in a final state or diverging. $\mapsto_P^*$ denotes the reflexive and transitive closure of $\mapsto_P$. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent (unsatisfiable). When it is clear from the context, we will drop the reference to the program P.

Example 2. Recall the program for `max` from Example 1. The first two rules are simplification rules, that replace `max(X,Y,Z)` by simpler constraints provided a guard holds. The third rule propagates constraints. Operationally, we add the body of the rule as redundant constraints, the `max` constraint is kept.

To the goal `max(1,2,M)` the first rule is applicable:

$$\texttt{max}(1,2,\texttt{M}) \mapsto^{\textbf{Simplify}} \texttt{M}{=}2.$$

To the goal `max(A,B,M)` $\wedge$ `A<B` the first rule is applicable:

$$\texttt{max(A,B,M)} \wedge \texttt{A<B} \mapsto^{\textbf{Simplify}} \texttt{M}{=}\texttt{B} \wedge \texttt{A<B}.$$

To the goal `max(A,A,M)` both simplification rules are applicable. In both cases the result is `M=A`.

$$\texttt{max}(\texttt{A},\texttt{A},\texttt{M}) \mapsto^{\textbf{Simplify}} \texttt{M}{=}\texttt{A}.$$

Redundancy from the propagation rule is useful, as the goal `max(A,3,3)` shows. Only the propagation rule is applicable, and then the first rule:

$$\texttt{max}(\texttt{A},3,3) \mapsto^{\textbf{Propagate}} \texttt{max(A,3,3)} \wedge \texttt{A}{\leq}3 \mapsto^{\textbf{Simplify}} \texttt{A}{\leq}3.$$

(The constraint `3=3` is simplified away by the built-in constraint solver.)

2.3 Well-Behavedness: Termination and Confluence

A CHR program is *well-behaved* if it is terminating and confluent.

Definition 2. A CHR program is called *terminating*, if there are no infinite computations.

For many existing CHR programs simple well-founded orderings are sufficient to prove termination [Frü02]. Problems arise with non-trivial interactions between simplification and propagation rules.

The confluence property of a program guarantees that any computation for a goal results in the same final state no matter which of the applicable rules are applied.

Definition 3. A CHR program is *confluent* if for all states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

The papers [Abd97,AFM99] give a decidable, sufficient and necessary condition for confluence for terminating CHR programs.

Example 3. The program for `max` from Example 1 is well-behaved. It is trivially terminating, since the bodies of the rules do not contain any CHR constraints. Thus confluence is decidable and can be shown to hold.

For example, to the state `max(X,Y,Z)` $\wedge$ `X=Y` all three rules are applicable, but in all cases, the final state is a built-in constraint logically equivalent to $\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z}$.

3 Rule Specialization

We are interested in any rule whose head could match (a part of) the given goal, taking into account any possible context. Therefore we consider all rules that have an overlap with the given goal. For an *overlap*, the head of the rule and the goal must have at least one CHR constraint in common. This is achieved by equating one or more constraints of the head and the goal.

We assume without loss of generality that rules (and goals) have disjoint sets of variables (if necessary, their variables have been renamed apart), unless otherwise noted.

In the following, meta-variables stand for (possibly empty) conjunctions of constraints. Unless otherwise noted, the letters C and D stand for built-in constraints, H for CHR constraints of the head of a rule, B for arbitrary constraints of the body of a rule, G for constraints in general.

We first specialize simplification rules.

Definition 4. Let G be a goal. Without loss of generality (w.l.o.g.), G can be written as

$$G_1 \wedge G_2 \wedge D,$$

where G_1 and G_2 are CHR constraints and D are built-in constraints.

Let R be a simplification rule

$$H_1 \wedge H_2 \Leftrightarrow C \mid B.$$

Then a *specialization of the simplification rule R with regard to the goal G* is the simplification rule

$$H_1 \wedge H_2 \wedge G_2 \Leftrightarrow H_1{=}G_1 \wedge C \wedge D \mid B \wedge G_2,$$

provided G_1 and H_1 are non-empty conjunctions and $CT \models \exists(H_1{=}G_1 \wedge C \wedge D)$.

$H_1{=}G_1$ defines the overlap of the goal with the head of the rule. (G_1 and H_1 are non-empty conjunctions, so that trivial overlaps are avoided.) G_2, the remainder of the goal G, occurs in both head and body of the specialized rule, since it will not be changed by the rule. The condition $CT \models \exists(H_1{=}G_1 \wedge C \wedge D)$ ensures that the specialized rule is not trivial. (With an unsatisfiable guard a rule is never applicable).

Example 4. Let G be the goal

$$\texttt{max}(\texttt{A},\texttt{B},\texttt{C}) \wedge \texttt{A} \geq \texttt{B}$$

Of course, this goal can be unfolded with the second rule of the program defining `max` (from Example 1), but for the sake of a simple example, let us specialize the first rule with it. Let R be the rule

$$\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \Leftrightarrow \texttt{X} \leq \texttt{Y} \mid \texttt{Z}{=}\texttt{Y}$$

We have a complete overlap with `max`, i.e.

$$G_1 = \texttt{max}(\texttt{A},\texttt{B},\texttt{C}), G_2 = \mathit{true}, D = (\texttt{A} \geq \texttt{B}),$$
$$H_1 = \texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}), H_2 = \mathit{true}, C = (\texttt{X} \leq \texttt{Y}), B = (\texttt{Z}{=}\texttt{Y}).$$

The resulting specialized rule is:

$$\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \wedge \mathit{true} \wedge \mathit{true} \Leftrightarrow \texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}){=}\texttt{max}(\texttt{A},\texttt{B},\texttt{C}) \wedge \texttt{X} \leq \texttt{Y} \wedge \texttt{A} \geq \texttt{B} \mid \texttt{Z}{=}\texttt{Y} \wedge \mathit{true}$$

After removal of redundant *true* constraints and after propagation and simplification of variable equalities and other built-in constraints, the above rule can be written as:

$$\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \Leftrightarrow \texttt{X}{=}\texttt{Y} \mid \texttt{Z}{=}\texttt{Y}.$$

The conditional answer that we get from this rule reads as:

Given G, if $\texttt{A}{=}\texttt{B}$ *then* $\texttt{C}{=}\texttt{B}$.

We now specialize propagation rules.

Definition 5. Let G be a goal of the form

$$G_1 \wedge G_2 \wedge D,$$

where G_1 and G_2 are CHR constraints and D are built-in constraints.

Let R be a propagation rule

$$H_1 \wedge H_2 \Rightarrow C \mid B.$$

Then a *specialization of the propagation rule R with regard to the goal G* is the propagation rule

$$H_1 \wedge H_2 \wedge G_2 \Rightarrow H_1{=}G_1 \wedge C \wedge D \mid B,$$

provided G_1 and H_1 are non-empty conjunctions and $CT \models \exists(H_1{=}G_1 \wedge C \wedge D)$.

In the propagation rule, we do not have to add the remainder G_2 of the goal to the body as in the case of a simplification rule, since it will not be removed from the head.

Example 5. Let G be the goal

$$\texttt{max}(\texttt{A},\texttt{B},\texttt{C}) \wedge \texttt{A} \geq \texttt{B}.$$

Let R be the propagation rule

$$\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \Rightarrow \texttt{X} \leq \texttt{Z} \wedge \texttt{Y} \leq \texttt{Z}.$$

The complete overlap with `max` is

$$G_1 = \texttt{max}(\texttt{A},\texttt{B},\texttt{C}), G_2 = \mathit{true}, D = (\texttt{A} \geq \texttt{B}),$$
$$H_1 = \texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}), H_2 = \mathit{true}, C = \mathit{true}, B = (\texttt{X} \leq \texttt{Z} \wedge \texttt{Y} \leq \texttt{Z}).$$

The specialized rule is

$$\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \wedge \mathit{true} \wedge \mathit{true} \Rightarrow \texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}){=}\texttt{max}(\texttt{A},\texttt{B},\texttt{C}) \wedge \mathit{true} \wedge \texttt{A} \geq \texttt{B} \mid$$
$$\texttt{X} \leq \texttt{Z} \wedge \texttt{Y} \leq \texttt{Z} \wedge \mathit{true}$$

After simplification of built-in constraints, the rule can be written as

$$\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \Rightarrow \texttt{Y} \leq \texttt{X} \mid \texttt{X} \leq \texttt{Z}.$$

In practice, we may introduce a new definition for the goal G, say `gmax`, and thus write the above rule as

$$\texttt{gmax}(\texttt{A},\texttt{B},\texttt{C}) \Rightarrow \texttt{A} \leq \texttt{C}.$$

More examples can be found in Section 5.

Remarks. If a goal is not specializable with any rule of the program, a programming error is likely. (The CHR constraints of the goal are either not defined or too specific.)

There are some interesting special cases of the above transformation: If we know that at run-time, the goal will not occur in any context with additional (CHR) constraints, we let H_2 be the empty conjunction. If in addition, G_2 is empty, we only specialize with rules whose heads overlap completely with the given goal.

A most general goal $G = c(X_1, \ldots, X_n)$, where X_i $(0 \leq i \leq n)$ are pairwise distinct variables, will return all rules that contain the constraint symbol c with arity n in their heads as a result of specialization.

4 Redundant Rules for Correctness

We show that the transformed rules are redundant in the program from which they derive. Hence they cannot change the operational semantics of the program. This result will establish correctness of the rule specialization transformation. We use a strict notion of correctness, where the observables are complete states (not only built-in constraints as usual in CC languages). We also show that specialized rules preserve termination and confluence (well-behavedness).

In this paper, we do not address the question whether original rules can be removed from the program once specialized rules are added. At the current state of research, we would like to refer to the papers [AF04] in which techniques to detect redundant rules in a program is described.

We start with a slightly more general definition of specialized rules than the ones derived in the previous section. Then we define redundant rules.

Definition 6. A rule R' is *special(ized)* in a CHR program P iff P contains another rule of the form

$$H \odot C \mid B \text{ where } \odot \in \{ \Leftrightarrow , \Rightarrow \}.$$

and R' is of the form

$$H \wedge G \odot C \wedge D \mid B \wedge G \text{ if } \odot = \Leftrightarrow ,$$

$$H \wedge G \odot C \wedge D \mid B \text{ if } \odot = \Rightarrow ,$$

provided the variables in the added goals G and D are either new or occur in H.

In [AF04] rule redundancy is defined in terms of finite computations.

Definition 7. A rule R is *redundant* in a CHR program P iff for all states S:

$$\text{If } S \mapsto_P^* S_1 \text{ then } S \mapsto_{P \setminus \{R\}}^* S_2,$$

where S_1 and S_2 are final states and S_1 and S_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

We need some statements about preservation of well-behavedness under addition and removal of redundant rules.

The addition of rules to a CHR program cannot inhibit computations.

Lemma 1. Given a CHR program P and a rule R. For all states S and S': If $S \mapsto_P^* S'$ then $S \mapsto_{P \cup \{R\}}^* S'$.

Proof. This is a direct consequence of the operational semantics of CHR. In a computation step, one may apply any of the applicable rules. So it suffices to ignore the newly added rule R to reproduce all computations of the original program without R.

The lemma also means that the removal of rules from a CHR program cannot introduce new computations.

From the above Lemma 1 the following two corollaries are immediate consequences.

Corollary 1. Removal of a redundant rule preserves termination and confluence of the program.

Proof. The claim holds since all computations are finite in a terminating program and since removal of a rule cannot introduce more computations.

Removal preserves confluence by definition of redundant rules, because a redundant rule could have only introduced computations that are also possible without it. □

Corollary 2. Addition of a redundant rule preserves confluence, but may destroy termination.

Proof. Addition of a redundant rule preserves confluence by definition, because a redundant rule only has finite computations that are also possible without it.

For termination, a counterexample suffices. Consider adding `p(X)` ⇔ `p(X)` to a program that defines `p`. Every finite computation with the new rule will be redundant, but there are obviously also infinite computations possible with the new rule. □

In order to arrive at our desired correctness result, we show that special rules are redundant rules. For the proof, we need the following three lemmata from [AF99].

Lemma 2. A computation can be repeated in any larger context, i.e. with states in which built-in and CHR constraints have been added.

$$\text{If } G \mapsto^* G' \text{ then } (G \wedge H) \mapsto^* (G' \wedge H).$$

Lemma 3. A computation can be repeated in a state where redundant built-in constraints have been removed. Let $CT \models \forall\, (D \rightarrow C)$.

$$\text{If } (H \wedge C \wedge D \wedge G) \mapsto^* S \text{ then } (H \wedge D \wedge G) \mapsto^* S.$$

Lemma 4. A computation can be repeated in a state where variables have been instantiated. Let H' and H be CHR constraints without common variables.

$$\text{If } (H \wedge H{=}H' \wedge C) \mapsto_P^* S \text{ then } (H' \wedge C[H{=}H']) \mapsto_P^* S,$$

where $C[H{=}H']$ denotes the substitution of the variables in C which also occur in H as prescribed by the syntactic equality $H{=}H'$.

We are now ready to prove that special rules are special redundant rules.

Theorem 1. Special rules are redundant rules.

Proof. By contradiction. We try to find a computation in a given CHR program P that is possible with the special rule R' but not possible without it (the program P still contains R). W.l.o.g. we consider single computation steps $S' \mapsto_{\{R'\}} S'_1$. We got to show that then $S' \mapsto_{\{R\}} S_1$ is always possible and S'_1 and S_1 are equivalent.

Consider the case where R' and R are simplification rules. Let R of the form

$$\mathbf{H} \Leftrightarrow \mathbf{C} \mid \mathbf{B}$$

Let R' be a special rule of R of the form

$$\mathbf{H} \wedge G \Leftrightarrow \mathbf{C} \wedge D \mid \mathbf{B} \wedge G$$

(Note that $\mathbf{H}$, $\mathbf{C}$ and $\mathbf{B}$ are identical in both rules.)

Consider any state S' with $S' \mapsto_{\{R'\}} S'_1$. Since R' is applicable, S' must be of the form

$$H' \wedge G' \wedge G'' \wedge D',$$

where $CT \models \forall(D' \rightarrow \mathbf{H}{=}H' \wedge G{=}G' \wedge \mathbf{C} \wedge D)$, and S'_1 must be of the form

$$\mathbf{B} \wedge G \wedge G'' \wedge \mathbf{H}{=}H' \wedge G{=}G' \wedge \mathbf{C} \wedge D \wedge D'.$$

But then a very similar computation step is possible with R, since $CT \models \forall(\mathbf{H}{=}H' \wedge G{=}G' \wedge \mathbf{C} \wedge D) \rightarrow \mathbf{H}{=}H' \wedge \mathbf{C})$, the applicability condition $CT \models \forall(D' \rightarrow \mathbf{H}{=}H' \wedge \mathbf{C})$ is fulfilled, and consequently S_1 is of the form

$$\mathbf{B} \wedge G' \wedge G'' \wedge \mathbf{H}{=}H' \wedge \mathbf{C} \wedge D'.$$

We now show that the two states S'_1 and S_1 are the equivalent up to renaming of local variables and equivalence of built-in constraints. More precisely, we are interested in operational equivalence of states: Given the program P, all computations with S'_1 as initial state are also possible with S_1 as initial state and vice versa.

Since S'_1 strictly contains S_1, we know by Lemma 2 that all computations with S_1 are also possible with S'_1.

We still have to show that S'_1 does not admit more computations than S_1. We transform S'_1 into S_1 while preserving logical and operational equivalence of states.

Since $CT \models \forall(D' \rightarrow D)$ as a consequence of the fulfilled rule applicability condition, we can remove D from state S_1' according to Lemma 3.

Finally, we apply Lemma 4 and compute $S_1'[G{=}G']$. The substitution affects the variables in G and their occurences in other subconjunctions of the state S_1' that stem from the rule. Clearly, $G[G{=}G'] = G'$. Also, H', G'' and D' remain unaffected, since they are subconjunctions from the goal that cannot have any variables in common with the rule from which G stems. Finally, $(\mathbf{H}{=}H')[G{=}G']$ can be left as $(\mathbf{H}{=}H')$ since the fulfilled applicability condition of R', $CT \models \forall(D' \rightarrow \mathbf{H}{=}H' \wedge G{=}G' \wedge \mathbf{C} \wedge D)$, implies that a variable common to H and G must be equated to the same term in both equations $\mathbf{H}{=}H'$ and $G{=}G'$. Since by definition of special rules, if G contains variables from the rule, they must also occur in $\mathbf{H}$, the subconjunction $\mathbf{C}$ is not affected either. So the overall result is the state:

$$\mathbf{B} \wedge G' \wedge G'' \wedge \mathbf{H}{=}H' \wedge \mathbf{C} \wedge D'.$$

We have successfully transformed S_1' into S_1. Hence there cannot exist a computation with R' that is not possible with R, i.e. the special rule R' is redundant in the program P that contains the rule R.

The proof for propagation rules is analogous.□

Corollary 3. The addition and removal of special rules to a program preserves its confluence.

Proof. Obvious, since special rules are redundant rules by Theorem 1 and Corollaries 1 and 2 for redundant rules.□

Theorem 2. The addition and removal of special rules to a program preserves its termination.

Proof. Since special rules are redundant rules by Theorem 1, their removal preserves termination by Corollary 1.

We show that the addition of special rules preserves termination by contradiction. In an infinite computation, the special rule must be applied infinitely often, since any sub-computation between the applications of the special rule must be finite, since the program without addition of the special rule is terminating.

The proof of Theorem 1 showed, that each computation step, where the special rules is applied, can be mimicked by exactly one computation steps without the special rule. But then the complete computation can be mimicked by applications of rules of the original program. Since the program was terminating, this computation cannot be infinite.□

5 More Examples

In this section, we use the concrete syntax of CHR implementations in Prolog instead of the abstract syntax presented so far. The reason for this is that we have transformed the following programs in that setting with a first implementation of rule specialization.

Recall the program for `max(X,Y,Z)` from Example 1.

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

Even though we did not adress unfolding of rules and simplification of built-in constraints in rules in this paper for space reasons, we will use these program manipulations in the following examples in a mild way in order to illustrate the usefulness of rule specialization.

Unfolding basically means to replace the body of a rule by the result of a computation starting with the guard and body of the rule. Note that in the case of propagation rules, we also add the head of the rule to the initial state of the computation (here the technical term "unfolding" turns into a misnomer). Since we assume well-behaved programs, unfolding will terminate and it suffices to consider any one computation because of confluence. *Built-in constraint simplification* basically replaces built-in constraints of the guard and body by simpler ones.

In the examples, we will derive all specialized rules for a given goal. However, we will not bother to derive specialized rules that are equivalent (up to reordering of head constraints and variable renaming) to other already derived specialized rules.

Example 6. Let the goal for specialization be:

```
max(X,Y,Z), max(Y,X,Z)
```

Specialization with the first conjunct of the goal, `max(X,Y,Z)` results in the specialized rules:

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

Unfolding of `max(Y,X,Z)` in each of the specialized rules:

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

Trivial simplification of built-in constraint in the rule bodies:

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

When specializing with the second conjunct `max(Y,X,Z)`, the same rules are derived (up to permutation of head constraints). These rules are obviously redundant.

Comparing the original and the specialized rules, we see that one of the `max` constraints in the goal is redundant, and, more generally, that `max` is commutative in its first two arguments. So an appropriate folding program transformation would allow us to derive the rule:

```
max(X,Y,Z), max(Y,X,Z) <=> max(X,Y,Z).
```

Example 7. The goal to specialize is now:

```
max(X,Y,Z), max(X,Y,U)
```

Specialization with first conjunct `max(X,Y,Z)` of the goal:

```
max(X,Y,Z), max(X,Y,U) <=> X=<Y | Z=Y, max(X,Y,U).
max(X,Y,Z), max(X,Y,U) <=> Y=<X | Z=X, max(X,Y,U).
max(X,Y,Z), max(X,Y,U) ==> X=<Z, Y=<Z.
```

Specialization with the other conjunct of the goal leads to the same rules (up to variable renaming). Unfolding of `max(X,Y,U)` in specialized rules:

```
max(X,Y,Z), max(X,Y,U) <=> X=<Y | Z=Y, U=Y.
max(X,Y,Z), max(X,Y,U) <=> Y=<X | Z=X, U=X.
max(X,Y,Z), max(X,Y,U) ==> X=<Z, Y=<Z, X=<U, Y=<U.
```

The built-in constraints in each simplification rules imply that `Z=U`. This reminds us that the third argument of `max` is functionally dependent on the first two arguments.

In the next example, we add a rule for functional dependency and specialize it with regard to the goal of Example 6. Because the goal and the head of the rule each have two constraints, there will be a more interesting overlap.

Example 8. The goal is:
`max(X,Y,Z), max(Y,X,Z).`
The functional dependency rule for `max` is:

```
max(X,Y,Z), max(X,Y,U) <=> max(X,Y,Z), Z=U.
```

Specialization with the functional dependency rule (again deriving the minimal number of rules):

```
max(A,B,C), max(A,B,D), max(B,A,C) <=> max(A,B,C), C=D,max(B,A,C).
max(A,A,C), max(A,A,C) <=> max(A,A,C), C=C.
```

If the folded rule of Example 6 is available, we can also unfold and simplify the first rule:

```
max(A,B,C), max(A,B,D), max(B,A,C) <=> C=D, max(A,B,C).
max(A,A,C), max(A,A,C) <=> A=C.
```

6 Conclusions

The current work is a first, small step into considering program transformation for the constraint handling rule (CHR) language. This line of research is motivated by two working hypothesis (as explained in the introduction):

– New applications of program transformation to problems such as verification, constraint solver optimization and agent specialization.
– The suitability of CHR as a declarative, concurrent constraint-based programming language with multi-headed rules for powerful program transformation techniques.

Here we have studied the specialization of rules with regard to a given goal. We have shown that the correctness of this program transformation: Adding and removing such specialized rules in a program does not change the program's operational semantics. Furthermore well-behavedness, i.e. termination and confluence, is preserved by these operations.

The additional examples in the previous section give some hints of what should be next:

– Unfolding and folding as well as rule simplifying program transformations for CHR.
– A methodology (strategies) how to employ these transformations to improve the performance of a program. In general, these strategies depend on the intended application of the programn transformation. A particular and basic question is to clarify which derived rules one should add and which original rules one should remove.

Finally, and not surprisingly, future work also concerns the practical aspects of improving the current preliminary ad-hoc implementation for rule specialization and applying it to larger examples.

References

[Abd97] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.

[AF99] Slim Abdennadher and Thom Frühwirth. Operational equivalence of CHR programs and constraints. In *Fifth International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713, pages 43–57. Springer, 1999.

[AF04] Slim Abdennadher and Thom Frühwirth. Integration and optimization of rule-based constraint solvers. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation - LOPSTR 2003, Revised Selected Papers*, LNCS. Springer, 2004.

[AFM99] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2), 1999.

[DP99] Giorgio Delzanno and Andreas Podelski. Model checking in CLP. *Lecture Notes in Computer Science*, 1579:223–239, 1999.

[EGM01] Sandro Etalle, Maurizio Gabbrielli, and Maria Chiara Meo. Transformations of CCP programs. *ACM Trans. Program. Lang. Syst.*, 23(3):304–395, 2001.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.

[FPP01] Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C.R. Ramakrishnan, and U. Ultes-Nitsche, editors, *ACM SIGPLAN International Workshop on Verification and Computational Logic*, pages 85–96, 2001.

[FPP04] Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Transformation rules for locally stratified constraint logic programs. 2004.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[Frü02] Thom Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.

[MS97] Torben Mogensen and Peter Sestoft. Partial evaluation. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. 1997.

[PGGS98] Josep Puyol-Gruart, Llus Godo, and Carles Sierra. Specialisation calculus and communication. *International Journal of Approximate Reasoning*, 18(1/2):107–130, 1998.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.*, 28(2):360–414, 1996.

[RKRR04] Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Trans. Program. Lang. Syst.*, 26(3):464–509, 2004.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1990.

[UF88] Kazunori Ueda and Koichi Furukawa. Transformation rules for GHC programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, pages 582–591, 1988.

[Wal03] Toby Walsh. Constraint patterns. In F. Rossi, editor, *9th International Conference on Principles and Practices of Constraint Programming (CP-2003)*, volume 2833, pages 53–64. Springer LNCS, 2003.