On Completion of Constraint Handling Rules

Slim Abdennadher and Thom Frühwirth

Computer Science Institute, University of Munich
Oettingenstr. 67, 80538 München, Germany
{Slim.Abdennadher, Thom.Fruehwirth}@informatik.uni-muenchen.de

Abstract. Constraint Handling Rules (CHR) is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers. An important property of any constraint solver is confluence: The result of a computation should be independent from the order in which constraints arrive and in which rules are applied. In previous work [1], a sufficient and necessary condition for the confluence of terminating CHR programs was given by adapting and extending results about conditional term rewriting systems. In this paper we investigate so-called completion methods that make a non-confluent CHR program confluent by adding new rules. As it turns out, completion can also exhibit inconsistency of a CHR program. Moreover, as shown in this paper, completion can be used to define new constraints in terms of already existing constraints and to derive constraint solvers for them.

1 Introduction

Constraint Handling Rules (CHR) is our proposal to allow more flexibility and application-oriented customization of constraint systems. CHR is a declarative language extension especially designed for writing user-defined constraints. CHR is essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. CHR defines both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints, which are logically redundant but may cause further simplification.

As a special-purpose language for constraints, CHR aims to fulfill the promise of user-defined constraints as described in [4]: "For the theoretician meta-theorems can be proved and analysis techniques invented once and for all; for the implementor different constructs (backward and forward chaining, suspension, compiler optimization, debugging) can be implemented once and for all; for the user only one set of ideas need to be understood, though with rich (albeit disciplined) variations (constraint systems)."

We have already shown in previous work [1] that analysis techniques are available for an important property of any constraint solver, namely confluence: The result of a computation should be independent from the order in which constraints arrive and in which rules are applied. For confluence of terminating CHR programs we were able to give a sufficient and necessary condition by adapting and extending work done in conditional term rewriting systems (TRS).

In this paper we investigate so-called completion methods as known from TRS [12]. Completion is the process of adding rules to a non-confluent set of rules until it becomes confluent. Once again, we have to adapt and extend the results from TRS to be applicable for CHR. As it turns out, our completion method for CHR can also exhibit inconsistency of the logical meaning of a CHR program.
A practical application of our completion method lies in software development. Completion can be used to define new constraints in terms of already existing ones and to derive constraint solvers for them. Furthermore, completion can be used as a method to provide generic answers given as a set of rules. In this way, completion helps the CHR programmer to extend, modify and specialize existing solvers instead of having to write them from scratch.
This paper is organized as follows. In Section 2 we define the CHR language and summarize previous confluence results. Section 3 presents our completion method for CHR, including a fair algorithm, a correctness theorem and a theorem relating completion and consistency. In Section 4 we give further examples for the use of our completion method. Finally, we conclude with a summary.

2 Preliminaries

In this section we give an overview of syntax and semantics as well as confluence results for CHR. More detailed presentations can be found in [1, 2]. We assume some familiarity with (concurrent) constraint (logic) programming [14, 11, 13].

2.1 Syntax of CHR

A *constraint* is a first order atom. We use two disjoint kinds of predicate symbols for two different classes of constraints: One kind for *built-in* constraints and one kind for *user-defined* constraints. Built-in constraints are those handled by a predefined constraint solver that already exists as a certified black-box solver. Typical built-in constraints include *true*, *false* and $=$. User-defined constraints are those defined by a CHR program.
A *CHR program* is a finite set of rules. There are two basic kinds of rules[1].
A *simplification rule* is of the form

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

A *propagation rule* is of the form

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

where *Rulename* is a unique identifier of a rule, the head H is a non-empty conjunction of user-defined constraints, the guard C is a conjunction of built-in constraints and the body B is a conjunction of built-in and user-defined constraints. Conjunctions of constraints as in the body are called *goals*. A guard "*true*" is usually omitted together with the vertical bar.

[1] Their syntax is inspired by concurrent logic programming languages like GHC.

2.2 Declarative Semantics of CHR

The logical meaning of a simplification rule is a logical equivalence provided the guard holds

$$\forall \bar{x}\ (C \rightarrow (H \leftrightarrow \exists \bar{y}\ B)).$$

The logical meaning of a propagation rule is an implication if the guard holds

$$\forall \bar{x}\ (C \rightarrow (H \rightarrow \exists \bar{y}\ B)),$$

where $\bar{x}$ is the list of variables occuring in H or in C and $\bar{y}$ are the variables occuring in B only.
The logical meaning $\mathcal{P}$ of a CHR program P is the conjunction of the logical meanings of its rules united with a (consistent) constraint theory CT that defines the built-in constraints. We require CT to define $=$ as syntactic equality.

2.3 Operational Semantics of CHR

The operational semantics of CHR is given by a transition system.
A *state* is a triple

$$<G, C_U, C_B>,$$

where G is a conjunction of user-defined and built-in constraints called *goal store*. C_U is a conjunction of user-defined constraints. C_B is a conjunction of built-in constraints. C_U and C_B are called *user-defined* and *built-in (constraint) stores*, respectively. An empty goal or user-defined store is represented by $\top$. An empty built-in store is represented by *true*.
Given a CHR program P we define the transition relation $\mapsto$ by introducing four kinds of computation steps (Figure 1). In the figure, all meta-variables stand for conjunctions of constraints. An equation $c(t_1, \ldots, t_n){=}d(s_1, \ldots, s_n)$ of two constraints stands for $t_1{=}s_1 \wedge \ldots \wedge t_n{=}s_n$ if c and d are the same predicate symbol and for *false* otherwise. An equation $(p_1 \wedge \ldots \wedge p_n){=}(q_1 \wedge \ldots \wedge q_m)$ stands for $p_1{=}q_1 \wedge \ldots \wedge p_n{=}q_n$ if $n = m$ and for *false* otherwise. Note that conjuncts can be permuted since conjunction is associative and commutative.[2]
In the **Solve** computation step, the built-in solver produces a new normalized constraint store C'_B that is logically equivalent (according to the constraint theory CT) to the conjunction of the new constraint C and the old constraint store C_B. **Introduce** transports a user-defined constraint H from the goal store into the user-defined constraint store. There it can be handled with other user-defined constraints by applying CHR.
To **Simplify** user-defined constraints H' means to remove them from the user-defined store and to add the body B of a fresh variant of a simplification rule $(R\ @\ H \Leftrightarrow C \mid B)$ to the goal store and the equation $H{=}H'$ and the guard C to the built-in store, provided H' matches the head H and the guard C is implied by the built-in constraint store C_B. Note that "matching" means that

[2] For technical reasons, we consider conjunctions of constraints not to be idempotent.

Solve

$$\frac{\begin{array}{c} C \text{ is a built-in constraint} \\ CT \models C_B \wedge C \leftrightarrow C'_B \end{array}}{<C \wedge G, C_U, C_B> \mapsto <G, C_U, C'_B>}$$

Introduce

$$\frac{H \text{ is a user-defined constraint}}{<H \wedge G, C_U, C_B> \mapsto <G, H \wedge C_U, C_B>}$$

Simplify

$(R \text{ @ } H \Leftrightarrow C \mid B)$ is a fresh variant of a rule in P with the variables $\bar{x}$

$$\frac{CT \models C_B \rightarrow \exists \bar{x} (H{=}H' \wedge C)}{<G, H' \wedge C_U, C_B> \mapsto <G \wedge B, C_U, C \wedge H{=}H' \wedge C_B>}$$

Propagate

$(R \text{ @ } H \Rightarrow C \mid B)$ is a fresh variant of a rule in P with the variables $\bar{x}$

$$\frac{CT \models C_B \rightarrow \exists \bar{x} (H{=}H' \wedge C)}{<G, H' \wedge C_U, C_B> \mapsto <G \wedge B, H' \wedge C_U, C \wedge H{=}H' \wedge C_B>}$$

Fig. 1. Computation Steps

it is only allowed to instantiate variables of H but not variables of H'. In the logical notation this is achieved by existentially quantifying only over the fresh variables $\bar{x}$ of the rule to be applied.
The **Propagate** transition is similar to the **Simplify** transition, but retains the user-defined constraints H' in the user-defined store. Trivial nontermination caused by applying the same propagation rule again and again is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses this issue can be found in [1].
An *initial state* for a goal G is of the form $<G, \top, \mathit{true}>$. A *final state* is either of the form $<G, C_U, \mathit{false}>$ (such a state is called *failed*) or of the form $<\top, C_U, C_B>$ with no computation step possible anymore and C_B not *false* (such a state is called *successful*).
A *computation* of a goal G is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$ beginning with the initial state for G and ending in a final state or diverging. $\mapsto^*$ denotes the reflexive and transitive closure of $\mapsto$.

Example 1. We define a user-defined constraint for a (partial) order $\leq$ that can handle variable arguments.

```
r1 @ X ≤ X ⇔ true.
r2 @ X ≤ Y ∧ Y ≤ X ⇔ X=Y.
r3 @ X ≤ Y ∧ Y ≤ Z ⇒ X ≤ Z.
r4 @ X ≤ Y ∧ X ≤ Y ⇔ X ≤ Y.
```

The CHR program implements reflexivity (`r1`), antisymmetry (`r2`), transitivity (`r3`) and idempotence (`r4`) in a straightforward way. The reflexivity rule `r1` states that $\texttt{X}\leq\texttt{X}$ is logically true. The antisymmetry rule `r2` means that if we find $\texttt{X}\leq\texttt{Y}$ as well as $\texttt{Y}\leq\texttt{X}$ in the current store, we can replace them by the logically equivalent `X=Y`. The transitivity rule `r3` propagates constraints. It states that the conjunction of $\texttt{X}\leq\texttt{Y}$ and $\texttt{Y}\leq\texttt{Z}$ implies $\texttt{X}\leq\texttt{Z}$. Operationally, we add the logical consequence $\texttt{X}\leq\texttt{Z}$ as a redundant constraint. The idempotence rule `r4` absorbs multiple occurrences of the same constraint.
In the following computation constraints which are considered in the current computation step are underlined:

$$
\begin{array}{ll}
 & <\underline{\texttt{A} \leq \texttt{B}} \wedge \underline{\texttt{C} \leq \texttt{A}} \wedge \underline{\texttt{B} \leq \texttt{C}}, \top, \texttt{true}> \\
\mapsto^*_{\textbf{Introduce}} & <\top, \underline{\texttt{A} \leq \texttt{B}} \wedge \underline{\texttt{C} \leq \texttt{A}} \wedge \texttt{B} \leq \texttt{C}, \texttt{true}> \\
\mapsto_{\textbf{Propagate}} & <\underline{\texttt{C} \leq \texttt{B}}, \texttt{A} \leq \texttt{B} \wedge \texttt{C} \leq \texttt{A} \wedge \texttt{B} \leq \texttt{C}, \texttt{true}> \\
\mapsto_{\textbf{Introduce}} & <\top, \texttt{A} \leq \texttt{B} \wedge \texttt{C} \leq \texttt{A} \wedge \underline{\texttt{B} \leq \texttt{C}} \wedge \underline{\texttt{C} \leq \texttt{B}}, \texttt{true}> \\
\mapsto_{\textbf{Simplify}} & <\underline{\texttt{B} = \texttt{C}}, \texttt{A} \leq \texttt{B} \wedge \texttt{C} \leq \texttt{A}, \texttt{true}> \\
\mapsto_{\textbf{Solve}} & <\top, \underline{\texttt{A} \leq \texttt{B}} \wedge \underline{\texttt{C} \leq \texttt{A}}, \texttt{B} = \texttt{C}> \\
\mapsto_{\textbf{Simplify}} & <\underline{\texttt{A} = \texttt{B}}, \top, \texttt{B} = \texttt{C}> \\
\mapsto_{\textbf{Solve}} & <\top, \top, \texttt{A} = \texttt{B} \wedge \texttt{B} = \texttt{C}>
\end{array}
$$

2.4 Confluence

The confluence property of a program guarantees that any computation starting from an arbitrary initial state, i.e. any possible order of rule applications, results in the same final state. Due to space limitations, we can just give an overview on confluence where some definitions are left informal. Detailed confluence results for CHR can be found in [1, 2, 3]. The papers adopt and extend the terminology and techniques of conditional TRS [8] about confluence. Our extensions enable handling of propagation rules, global context (the built-in constraint store) and local variables.
We require that states are normalized so that they can be compared syntactically in a meaningful way. Basically, we require that the built-in constraints are in a (unique) normal form, where all syntactical equalities are made explicit and are propagated to all components of the state. The normalization also has to make all failed states syntactically identical.

Definition 1. A CHR program is called *confluent* if for all states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then S_1 and S_2 are joinable. Two states S_1 and S_2 are called *joinable* if there exists a state T such that $S_1 \mapsto^* T$ and $S_2 \mapsto^* T$.

To analyze confluence of a given CHR program we cannot check joinability starting from any given ancestor state S, because in general there are infinitely many such states. However one can construct a finite number of "minimal" states where more than one rule is applicable (and thus more than one transition possible) based on the following observations: First, adding constraints to the components of the state cannot inhibit the application of a rule as long as the

built-in constraint store remains consistent (monotonicity property). Second, joinability can only be destroyed if one rule inhibits the application of another rule. Only the removal of constraints can affect the applicability of another rule, in case the removed constraint is needed by the other rule.
By monotonicity, we can restrict ourselves to ancestor states that consist of the head and guards of two rules. To possibly destroy joinability, at least one rule must be a simplification rule and the two rules must *overlap*, i.e. have at least one head atom in common in the ancestor state. This is achieved by equating head atoms in the state.

Definition 2. Given a simplification rule R_1 and an arbitrary (not necessarily different) rule R_2, whose variables have been renamed apart. Let G_i denote the guard ($i = 1, 2$). Let H_i^c and H_i be a partition of the head of the rule R_i into two conjunctions, where H_i^c is nonempty. Then a *critical ancestor state S of R_1 and R_2* is

$$<\top, H_1^c \wedge H_1 \wedge H_2, (H_1^c = H_2^c) \wedge G_1 \wedge G_2>,$$

provided $(H_1^c = H_2^c) \wedge G_1 \wedge G_2$ is consistent in CT.

The application of R_1 and R_2, respectively, to S leads to two states that form the so-called *critical pair*.

Definition 3. Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2 then the tuple (S_1, S_2) is the *critical pair* of R_1 and R_2. A critical pair (S_1, S_2) is *joinable*, if S_1 and S_2 are joinable.

Example 2. Consider the program for $\leq$ of Example 1. The following critical pair stems from the critical ancestor state[3] $<\top, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \wedge \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}>$ of `r2` and `r3`:

$$(S_1, S_2) := (<\mathtt{X} = \mathtt{Y}, \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}>, <\mathtt{X} \leq \mathtt{Z}, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{X}, \mathtt{true}>)$$

It is joinable. A computation beginning with S_1 proceeds as follows:

$$\begin{array}{ll} & <\underline{\mathtt{X} = \mathtt{Y}}, \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}> \\ \mapsto_{\mathbf{Solve}} & <\top, \mathtt{X} \leq \mathtt{Z}, \mathtt{X} = \mathtt{Y}> \end{array}$$

A computation beginning with S_2 results in the same final state:

$$\begin{array}{ll} & <\underline{\mathtt{X} \leq \mathtt{Z}}, \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{X}, \mathtt{true}> \\ \mapsto_{\mathbf{Introduce}} & <\top, \mathtt{X} \leq \mathtt{Z} \wedge \underline{\mathtt{X} \leq \mathtt{Y}} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \underline{\mathtt{Y} \leq \mathtt{X}}, \mathtt{true}> \\ \mapsto_{\mathbf{Simplify}} & <\underline{\mathtt{X} = \mathtt{Y}}, \mathtt{X} \leq \mathtt{Z} \wedge \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}> \\ \mapsto_{\mathbf{Solve}} & <\top, \underline{\mathtt{X} \leq \mathtt{Z}} \wedge \underline{\mathtt{X} \leq \mathtt{Z}}, \mathtt{X} = \mathtt{Y}> \\ \mapsto_{\mathbf{Simplify}} & <\top, \mathtt{X} \leq \mathtt{Z}, \mathtt{X} = \mathtt{Y}> \end{array}$$

Definition 4. A CHR program is called *terminating*, if there are no infinite computations.

[3] Variables from different rules already identified to have an overlap; for readability.

For most existing CHR programs it is straightforward to prove termination using simple well-founded orderings. Otherwise it seems impossible without relying on implementational details [10].
The following theorem from [1] gives a decidable, sufficient and necessary criterion for confluence of a terminating program:

Theorem 5. A terminating CHR program is confluent iff all its critical pairs are joinable.

3 Completion

The idea of completion as developed for term rewriting systems (TRS) is to derive a rule from a non-joinable critical pair that would allow a transition from one of the critical states into the other one, thus re-introducing confluence [12]. In analogy to completion algorithms for TRS [5], our algorithm for CHR maintains a set C of critical pairs and a set P of rules. These sets are manipulated by four inference rules (Figure 2). Terminology is taken from TRS. We write $(C,P) \longmapsto (C',P')$ to indicate that the pair (C',P') can be obtained from (C,P) by an application of an inference rule.

CP-Deduction:

$$\frac{\begin{array}{c}(C,P)\\ (S_1,S_2) \text{ is a critical pair of } P\end{array}}{(C \cup \{(S_1,S_2)\}, P)}$$

CP-Orientation:

$$\frac{\begin{array}{c}(C \cup \{(S_1,S_2)\}, P)\\ R = \mathit{orient}_{\gg}(S_1,S_2)\end{array}}{(C, P \cup R)}$$

CP-Deletion:

$$\frac{\begin{array}{c}(C \cup \{(S_1,S_2)\}, P)\\ S_1 \text{ and } S_2 \text{ are joinable}\end{array}}{(C,P)}$$

CP-Simplification:

$$\frac{\begin{array}{c}(C \cup \{(S_1,S_2)\}, P)\\ S_1 \mapsto S_1'\end{array}}{(C \cup \{(S_1',S_2)\}, P)}$$

$$\frac{\begin{array}{c}(C \cup \{(S_1,S_2)\}, P)\\ S_2 \mapsto S_2'\end{array}}{(C \cup \{(S_1,S_2')\}, P)}$$

Fig. 2. Inference rules of completion

The rule *CP-Deduction* permits to add critical pairs to C. *CP-Orientation* removes a critical pair from C and adds new rules to P, provided the critical pair can be oriented with respect to the termination ordering $\gg$. In contrast to completion methods for TRS, we need - as examplified below - more than one rule to make a critical pair joinable. With the inference rules *CP-Deletion* and *CP-Simplification*, C can be simplified. The rule *CP-Deletion* removes a joinable critical pair. The rule *CP-Simplification* replaces state in a critical pair by its successor state.
Different versions of completion differ in which critical pair they "orient" first and in how they keep track of critical pairs that still need to be processed. A version of completion is *fair* if it does not avoid processing any critical pair infinitely often. One simple fair version of completion is to use the following strategy:

1. Set $i := 0$ and begin with the set of the rules $P_0 := P$ and their non-joinable critical pairs C_0.
2. If $C_i = \emptyset$, stop successfully with $P' = P_i$.
3. Let C_i be $C \cup \{(S_1, S_2)\}$. Then $(C \cup \{(S_1, S_2)\}, P_i) \longmapsto^*_{CP-Simplification} (C \cup \{(T_1, T_2)\}, P_i)$, such that T_1 and T_2 are final states. If $R = \mathit{orient}_{\gg}(T_1, T_2)$, then $P_{i+1} := P_i \cup R$. Otherwise abort unsuccessfully.
4. Form all critical pairs between a rule of R and all rules of P_{i+1} by the inference rule *CP-Deduction*. To produce C_{i+1}, add these critical pairs to C_i and then remove all (in P_{i+1}) joinable critical pairs by the inference rule *CP-Deletion*.
5. Set $i := i + 1$ and go to 2.

With this strategy, we need to define $\mathit{orient}_{\gg}$ only for final states. For the case $C_{U1} \neq \top$ and $C_{U1} \gg C_{U2}$ (the case $C_{U2} \neq \top$ and $C_{U2} \gg C_{U1}$ is analogous) we define
$\mathit{orient}_{\gg}(<\top, C_{U1}, C_{B1}>, <\top, C_{U2}, C_{B2}>) :=$

$$\begin{cases} \{C_{U1} \Leftrightarrow C_{B1} \mid C_{U2} \wedge C_{B2},\ C_{U2} \Rightarrow C_{B2} \mid C_{B1}\} & \text{if } C_{U2} \neq \top \\ \{C_{U1} \Leftrightarrow C_{B1} \mid C_{B2}\} & \text{if } C_{U2} = \top \text{ and } CT \models C_{B1} \leftrightarrow C_{B2} \end{cases}$$

Note that propagation rules whose bodies consist only of *true* can be eliminated. One obvious difference to completion in TRS is that our completion for CHR derives *two* rules out of a critical pair in general. In example 4 we show why the additional propagation rule is necessary.

Example 3. Let P be a CHR program that represents a fragment of the Boolean constraint solver [9] defining the logical connectives `and` and `imp`. The constraint `and(X,Y,Z)` stands for `X` $\wedge$ `Y` $\leftrightarrow$ `Z` and `imp(X,Y)` for `X` $\rightarrow$ `Y`.[4]

```
and1 @ and(X,X,Z)  ⇔  X=Z.
and2 @ and(X,Y,X)  ⇔  imp(X,Y).
```

[4] In the solver, `imp` is used as an ordering relation which explains the binary notation in contrast to the ternary `and`.

```
and3 @ and(X,Y,Z) ∧ and(X,Y,Z1)  ⇔  and(X,Y,Z) ∧ Z=Z1.

imp1 @ imp(X,Y) ∧ imp(Y,X)  ⇔  X=Y.
```

We choose the termination ordering:

$C_1 \gg C_2$ iff C_2 is a conjunct of C_1 or C_1 is `and(X,Y,Z)` and C_2 is `imp(X,Y)`.

The completion procedure results in the following sequence; critical pairs which are considered in the current inference step are underlined.

$$
\begin{aligned}
P_0 = \quad & P \\
C_0 = \quad & \{\underline{(<\mathtt{imp}(\mathtt{X},\mathtt{X}),\top,\mathtt{true}>,<\mathtt{X}=\mathtt{X},\top,\mathtt{true}>)}, \\
& (<\mathtt{X}=\mathtt{Z},\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{X}),\mathtt{true}>,<\mathtt{imp}(\mathtt{X},\mathtt{Y}),\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{true}>), \\
& (<\mathtt{X}=\mathtt{Z},\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{true}>,<\mathtt{imp}(\mathtt{X},\mathtt{Y}),\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{true}>)\}
\end{aligned}
$$

$$
\begin{aligned}
P_1 = \quad & P \cup \{\mathtt{r1@imp}(\mathtt{X},\mathtt{X}) \Leftrightarrow \mathtt{true}\} \\
C_1 = \quad & \{\underline{(<\mathtt{X}=\mathtt{Z},\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{X}),\mathtt{true}>,<\mathtt{imp}(\mathtt{X},\mathtt{Y}),\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{true}>),} \\
& \underline{(<\mathtt{X}=\mathtt{Z},\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{true}>,<\mathtt{imp}(\mathtt{X},\mathtt{Y}),\mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{true}>)}\}
\end{aligned}
$$

$$
\begin{aligned}
P_2 = \quad & P_1 \cup \{\mathtt{r2@imp}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{and}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \Leftrightarrow \mathtt{imp}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{X}=\mathtt{Z}\} \\
C_2 = \quad & \{\underline{(<\mathtt{X}=\mathtt{X} \wedge \mathtt{imp}(\mathtt{X},\mathtt{Y}),\top,\mathtt{true}>,<\mathtt{imp}(\mathtt{X},\mathtt{Y}),\mathtt{imp}(\mathtt{X},\mathtt{Y}),\mathtt{true}>)}\}
\end{aligned}
$$

$$
\begin{aligned}
P_3 = \quad & P_2 \cup \{\mathtt{r3@imp}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{imp}(\mathtt{X},\mathtt{Y}) \Leftrightarrow \mathtt{imp}(\mathtt{X},\mathtt{Y})\} \\
C_3 = \quad & \emptyset
\end{aligned}
$$

Let c.p. stand for critical pair from now on. The first, underlined c.p. of C_0 comes from equating the heads of rules `and2` and `and1`. This c.p. becomes joinable by adding rule `r1` to P. The second c.p. of C_0 comes from equating the head of rule `and2` with the first head constraint of `and3`. It becomes joinable by adding rule `r2`. The third c.p. of C_0 comes from equating the head of `and2` with the second head constraint of `and3`. It also becomes joinable due to `r2`. A non-joinable c.p. is added in the third step, which comes from equating the head of `and2` and the second head constraint of `r2`. For the sake of simplicity we dropped all new propagation rules generated by *orient*, since they were trivial, i.e. their bodies consisted only of `true`.
The result of the completion procedure is $P' = P_3$:

```
% rules and1, and2, and3, imp1 together with

r1 @ imp(X,X)  ⇔  true.
r2 @ imp(X,Y) ∧ and(X,Y,Z)  ⇔  imp(X,Y) ∧ X=Z.
r3 @ imp(X,Y) ∧ imp(X,Y)  ⇔  imp(X,Y).
```

The new rules derived by completion reveal some interesting properties of `imp`, e.g. `r1` states that "`X` implies `X`" is always true. P' is terminating (see Theorem 9 for correctness) and all its critical pairs are joinable, therefore P' is confluent.

The next example shows that in general it is not sufficient to derive only simplification rules as in completion for TRS, in order to join a non-joinable critical pair.

Example 4. Let P be the following CHR program, where `p`, `q` and `r` are user-defined constraints and $\geq$, $\leq$ are built-in constraints.

$$
\begin{array}{l}
\mathtt{r1\ @\ p(X,Y)} \ \Leftrightarrow\ \mathtt{X} \geq \mathtt{Y} \wedge \mathtt{q(X,Y)}. \\
\mathtt{r2\ @\ p(X,Y)} \ \Leftrightarrow\ \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{r(X,Y)}.
\end{array}
$$

P is not confluent, since the c.p. stemming from `r1` and `r2`

$$(<\mathtt{q(X,Y)} \wedge \mathtt{X} \geq \mathtt{Y}, \top, \mathtt{true}>, <\mathtt{r(X,Y)} \wedge \mathtt{X} \leq \mathtt{Y}, \top, \mathtt{true}>)$$

is non-joinable. The corresponding final states are

$$<\top, \mathtt{q(X,Y)}, \mathtt{X} \geq \mathtt{Y}>, <\top, \mathtt{r(X,Y)}, \mathtt{X} \leq \mathtt{Y}>.$$

Let $\mathtt{r(X,Y)} \gg \mathtt{q(X,Y)}$. Then the completion procedure derives:

$$
\begin{array}{l}
\mathtt{r3\ @\ r(X,Y)} \ \Leftrightarrow\ \mathtt{X} \leq \mathtt{Y} \mid \mathtt{q(X,Y)} \wedge \mathtt{X} \geq \mathtt{Y}. \\
\mathtt{r4\ @\ q(X,Y)} \ \Rightarrow\ \mathtt{X} \geq \mathtt{Y} \mid \mathtt{X} \leq \mathtt{Y}.
\end{array}
$$

The following computations show that it is necessary to derive the propagation rule to P to join the c.p. above:

$$
\begin{array}{ll}
 & <\mathtt{r(X,Y)} \wedge \mathtt{X} \leq \mathtt{Y}, \top, \mathtt{true}> \\
\mapsto_{\mathbf{Solve}} & <\mathtt{r(X,Y)}, \top, \mathtt{X} \leq \mathtt{Y}> \\
\mapsto_{\mathbf{Introduce}} & <\top, \mathtt{r(X,Y)}, \mathtt{X} \leq \mathtt{Y}> \\
\mapsto_{\mathbf{Simplify}} & <\mathtt{q(X,Y)} \wedge \mathtt{X} \geq \mathtt{Y}, \top, \mathtt{X} \leq \mathtt{Y}> \\
\mapsto_{\mathbf{Solve}} & <\mathtt{q(X,Y)}, \top, \mathtt{X} = \mathtt{Y}> \\
\mapsto_{\mathbf{Introduce}} & <\top, \mathtt{q(X,Y)}, \mathtt{X} = \mathtt{Y}> \\
\mapsto_{\mathbf{Propagate}} & <\mathtt{X} \leq \mathtt{Y}, \mathtt{q(X,Y)}, \mathtt{X} = \mathtt{Y}> \\
\mapsto_{\mathbf{Solve}} & <\top, \mathtt{q(X,Y)}, \mathtt{X} = \mathtt{Y}>
\end{array}
$$

Without the application of the propagation rule the computation below would result in a different final state:

$$
\begin{array}{ll}
 & <\mathtt{q(X,Y)} \wedge \mathtt{X} \geq \mathtt{Y}, \top, \mathtt{true}> \\
\mapsto_{\mathbf{Solve}} & <\mathtt{q(X,Y)}, \top, \mathtt{X} \geq \mathtt{Y}> \\
\mapsto_{\mathbf{Introduce}} & <\top, \mathtt{q(X,Y)}, \mathtt{X} \geq \mathtt{Y}> \\
\mapsto_{\mathbf{Propagate}} & <\mathtt{X} \leq \mathtt{Y}, \mathtt{q(X,Y)}, \mathtt{X} \geq \mathtt{Y}> \\
\mapsto_{\mathbf{Solve}} & <\top, \mathtt{q(X,Y)}, \mathtt{X} = \mathtt{Y}>
\end{array}
$$

As is the case for TRS our completion procedure cannot be always successful. We distinguish three cases:

1. The algorithm stops successfully and returns a program P'.

2. The algorithm aborts unsuccessfully, if a critical pair cannot be transformed into rules for one of three reasons:
   - The program remains terminating if new rules are added but the termination ordering is too weak to detect this.
   - The program loses termination if new rules are added.
   - The critical pair consists exclusively of built-in constraints.
3. The algorithm does not terminate, because new rules produce new critical pairs, which require again new rules, and so on.

In the next section we show that when the algorithm stops successfully, the returned program P' is confluent and terminating.

3.1 Correctness of the Completion Algorithm

We now show that the completion procedure applied to a CHR program results in an equivalent program. For the proof to go through, every rule has to satisfy a *range-restriction* condition: Every variable in the body or the guard appears also in the head. In practice, in almost all solvers, rules with local variables (variables that occur on the right-hand side of a rule only) can be rewritten to be range-restricted. One introduces interpreted function symbols for the local variables and extends the equality theory in CT accordingly.
Some definitions are necessary before we go further.

Definition 6. Let P_1 and P_2 be CHR programs and let CT be the appropriate constraint theory. P_1 and P_2 are *equivalent*, if their logical meanings $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent:

$$CT \models \mathcal{P}_1 \leftrightarrow \mathcal{P}_2$$

Definition 7. Let S be a state $<Gs, C_U, C_B>$, which appears in a computation of G. The *logical meaning* of S is the formula

$$\exists \bar{x}\ Gs \wedge C_U \wedge C_B,$$

where $\bar{x}$ are the (local) variables appearing in S and not in G. A *computable constraint* of G is the logical meaning of a state which appears in a computation of G.

Lemma 8. Let P be a CHR program and G be a goal. Then for all computable constraints C_1 and C_2 of G the following holds:

$$P \cup CT \models \forall\ (C_1 \leftrightarrow C_2).$$

Proof. See [2].

Theorem 9. Let P be a range-restricted CHR program respecting a termination ordering $\gg$ and C be the set of the non-joinable critical pairs of P. If, for inputs $C_0 = C$, $P_0 = P$ and $\gg$, the completion procedure generates a successful derivation of the form $(C_0, P_0) \longmapsto \ldots \longmapsto (\emptyset, P')$, then P' is terminating with respect to $\gg$, confluent and equivalent to P.

Proof. Omitted from the final version for space reasons. See [2].

3.2 Consistency

Another property of completion is that it can exhibit inconsistency of the program to complete.

Definition 10. A constraint theory CT is called *complete*, if for every constraint c either $CT \models \forall c$ or $CT \models \forall \neg c$ holds.

Theorem 11. Let P be a CHR program and CT a complete theory. If the completion procedure aborts unsuccessfully, because the corresponding final states of a critical pair consist only of differing built-in constraints, then the logical meaning of P is inconsistent.

Proof. Let C_{B1}, C_{B2} be the built-in constraints of the final states. According to Lemma 8, the following holds

$$\mathcal{P} \cup CT \models \forall\, (\exists \bar{x}_1 C_{B1} \leftrightarrow \exists \bar{x}_2 C_{B2}),$$

where $\bar{x}_1, \bar{x}_2$ are the local variables of the final states.
We prove the claim by contradiction. Assume that $\mathcal{P}$ is consistent. Then $\mathcal{P} \cup CT$ is consistent. Therefore $CT \models \forall\, (\exists \bar{x}_1 C_{B1} \leftrightarrow \exists \bar{x}_2 C_{B2})$ holds, since CT is complete. Then according to the normalization function C_{B1} and C_{B2} have a unique form. This contradicts the prerequisite that the states are different. $\square$

Example 5. Let P be the following CHR program trying to implement the constraint `maximum(X,Y,Z)`, which holds, if `Z` is the maximum of `X` and `Y`, and where $\leq$ and $=$ are built-in constraints. Note that there is a typo in the body of the second rule, since `Y` should have been `Z`:

```
r1 @ maximum(X,Y,Z) ⇔ X ≤ Y | Z = Y.
r2 @ maximum(X,Y,Z) ⇔ Y ≤ X | Y = X.
```

The c.p.

$$(\texttt{<Z = Y}, \top, \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} \leq \texttt{X>}, \texttt{<Y = X}, \top, \texttt{X} \leq \texttt{Y} \wedge \texttt{Y} \leq \texttt{X>})$$

stemming from `r1` and `r2` is not joinable. The states of the c.p. consist only of built-in constraints. Thus the completion procedure aborts unsuccessfully.
The logical meaning of this CHR program is the theory

```
∀ X,Y,Z (X ≤ Y → (maximum(X,Y,Z) ↔ Z = Y))
∀ X,Y,Z (Y ≤ X → (maximum(X,Y,Z) ↔ Y = X))
```

together with an appropriate constraint theory describing $\leq$ as an order relation and $=$ as syntactic equality. The logical meaning $\mathcal{P}$ of this program is not a consistent theory. This can be exemplified by the atomic formula maximum$(1, 1, 0)$, which is logically equivalent to 0=1 (and therefore *false*) using the first formula. Using the second formula, however maximum$(1, 1, 0)$ is logically equivalent to 1=1 (and therefore *true*). This results in $\mathcal{P} \cup CT \models \mathit{false} \leftrightarrow \mathit{true}$.

4 More Uses of Completion

The first example shows that the completion method can be used - to some extent – to specialize constraints.

Example 6. We define the constraint $<$ as a special case of $\leq$. If we extend the CHR program for $\leq$ of Example 1 by the simplification rule

$\mathtt{r5}$ @ $\mathtt{X} \leq \mathtt{Y} \Leftrightarrow \mathtt{X} \neq \mathtt{Y} \mid \mathtt{X} < \mathtt{Y}.$

then the resulting program is not confluent anymore. With the termination ordering

$C_1 \gg C_2$ iff C_2 is a conjunct of C_1 or C_1 is $\mathtt{X} \leq \mathtt{Y}$ and C_2 is $\mathtt{X} < \mathtt{Y}$,

the completion procedure derives the following rules:

$\mathtt{r6}$ @ $\mathtt{X} < \mathtt{Y} \wedge \mathtt{Y} < \mathtt{X} \Leftrightarrow \mathtt{X} \neq \mathtt{Y} \mid \mathtt{false}.$
$\mathtt{r7}$ @ $\mathtt{X} < \mathtt{Y} \wedge \mathtt{X} < \mathtt{Y} \Leftrightarrow \mathtt{X} \neq \mathtt{Y} \mid \mathtt{X} < \mathtt{Y}.$

where `r6` comes from a c.p. of `r2` and `r5`,

$$(<\mathtt{X} = \mathtt{Y}, \top, \mathtt{X} \neq \mathtt{Y}>, <\mathtt{X} < \mathtt{Y}, \mathtt{Y} \leq \mathtt{X}, \mathtt{X} \neq \mathtt{Y}>).$$

`r7` comes from a c.p. of `r4` and `r5`,

$$(<\top, \mathtt{X} \leq \mathtt{Y}, \mathtt{X} \neq \mathtt{Y}>, <\mathtt{X} < \mathtt{Y}, \mathtt{Y} \leq \mathtt{X}, \mathtt{X} \neq \mathtt{Y}>).$$

`r6` obviously defines the antisymmetry of $<$ and `r7` idempotence. Irreflexivity of $<$ could not be derived, since the definition of $<$ by rule `r5` already presupposes that $\mathtt{X} \neq \mathtt{Y}$. However, completion can derive irreflexivity from rule `r1` and the rule

$\mathtt{r8}$ @ $\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} < \mathtt{X} \Leftrightarrow \mathtt{false}.$

since the resulting c.p.

$$(<\mathtt{X} < \mathtt{X}, \top, \mathtt{true}>, <\mathtt{false}, \top, \mathtt{true}>).$$

leads to the simplification rule

$\mathtt{r9}$ @ $\mathtt{X} < \mathtt{X} \Leftrightarrow \mathtt{false}.$

The next example shows how completion can be used as a method to provide generic answers, even if a constraint cannot further be simplified. This retains some of the power of logic languages like Prolog, where several answers can be given. Our approach is similar to the ones that related Prolog and TRS computation methods [7, 6].

Example 7. A CHR formulation of the classical Prolog predicate `member` as a user-defined constraint is ($\neq$ is built-in):

```
r1 @ member(X,[]) ⇔ false.
r2 @ member(X,[X|_]) ⇔ true.
r3 @ member(X,[H|T]) ⇔ X ≠ H | member(X,T).
```

Using CHR, the goal member(X,[1,2,3]) delays. However Prolog generates three solutions X=1, X=2 and X=3. If we add

```
r4 @ member(X,[1,2,3]) ⇔ answer(X).
```

to P, then the program is non-confluent. Our completion procedure derives:

```
a1 @ answer(1) ⇔ true.
a2 @ answer(2) ⇔ true.
a3 @ answer(3) ⇔ true.
a4 @ answer(X) ⇔ X ≠ 1 ∧ X ≠ 2 ∧ X ≠ 3 | false.
```

The rules a1,a2 and a3 correspond to the answers of the Prolog program, while the last rule a4 makes explicit the closed world assumption underlying Clark's completion semantics of Prolog.

Completion can also derive recursively defined constraints.

Example 8. Let P be the following CHR program

```
r1 @ append([],L,L) ⇔ true.
r2 @ append([X|L1],Y,[X|L2]) ⇔ append(L1,Y,L2).
```

defining the well-known ternary append predicate for lists as a simple constraint, which holds if its third argument is a concatenation of the first and the second argument. P is confluent since there are no critical pairs. When we add the rule

```
r3 @ append(L1,[],L3) ⇔ new(L1,L3).
```

confluence is lost. Completion derives a constraint solver for new:

```
r4 @ new([],[]) ⇔ true.           % joins c.p. of r1 and r3
r5 @ new([A|B],[A|C]) ⇔ new(B,C). % joins c.p. of r2 and r3
```

Our completion procedure has uncovered that append(L1,[],L3) holds exactly if L1 and L2 are the same list, as tested by the generated, recursive constraint new. Note that the rules represent in a finite way the infinitely many answers that would be generated by the corresponding program in Prolog.

5 Conclusions

We introduced a completion method for Constraint Handling Rules (CHR). Completion methods make a non-confluent CHR program confluent by adding new rules. We have shown that our proposed completion procedure is correct and can exhibit inconsistency of a CHR program. We also gave various examples to show that completion can be used as a method to provide generic answers and to define new constraints from existing ones and to derive constraint solvers for them. The latter helps the CHR programmer to extend, modify and specialize existing solvers instead of having to write them from scratch. We currently investigate using our completion method on larger real-life solvers for easing their modification and for giving more informative answers. An interesting direction for future work is to explore the relationship of completion to partial evaluation.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP'97*, LNCS 1330. Springer-Verlag, 1997.
2. S. Abdennadher. *Analyse von regelbasierten Constraintlösern (in German)*. PhD thesis, Computer Science Institute, LMU Munich, February 1998.
3. S. Abdennadher, T. Frühwirth, and H. Meuss. On confluence of constraint handling rules. In *2nd International Conference on Principles and Practice of Constraint Programming, CP'96*, LNCS 1118. Springer-Verlag, August 1996. Revised and extended version to appear in the Constraints Journal.
4. ACM. The constraint programming working group. Technical report, ACM-MIT SDRC Workshop, Report Outline, Draft, September 1996.
5. L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, LNCS 230. Springer-Verlag, July 1986.
6. M. P. Bonacina and J. Hsiang. On rewrite programs: Semantics and relationsship with PROLOG. *Journal of Logic Programming*, 14:155–180, 1992.
7. N. Dershowitz and N. A. Josephson. Logic programming by completion. In Sten-Åke Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, Uppsala, 1984.
8. N. Dershowitz, N. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems*, LNCS 308, 1988.
9. T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, March 1995.
10. T. Frühwirth. *A Declarative Language for Constraint Systems: Theory and Practice of Constraint Handling Rules.* Habilitation, Computer Science Institute, LMU Munich, 1998. Shortened version to appear in Journal of Logic Programming, Special Issue on Constraint Logic Programming, P. Stuckey and K. Marriot, editors.
11. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20, 1994.
12. D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*. Pergamon Press, 1970.
13. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.
14. V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, 1993.