Confluence Modulo Equivalence in Constraint Handling Rules

Henning Christiansen Maja H. Kirkeby*

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, Denmark
E-mail: henning@ruc.dk, majaht@ruc.dk

Abstract. Previous results on confluence for Constraint Handling Rules, CHR, are generalized to take into account user-defined state equivalence relations. This allows a much larger class of programs to enjoy the advantages of confluence, which include various optimization techniques and simplified correctness proofs. A new operational semantics for CHR is introduced that reduces notational overhead significantly and allows to consider confluence for programs with extra-logical and incomplete built-in predicates. Proofs of confluence are demonstrated for programs with redundant data representation, e.g., sets-as-lists, for dynamic programming algorithms with pruning as well as a Union-Find program, which are not covered by previous confluence notions for CHR.

1 Introduction

A rewrite system is confluent if all derivations from a common initial state end in the same final state. Confluence, like termination, is often a desirable property, and proof of confluence is a typical ingredient of a correctness proof. For a programming language based on rewriting such as Constraint Handling Rules, CHR, it ensures correctness of parallel implementations and application order optimizations.

Previous studies of confluence for CHR programs are based on Newman's lemma. This lemma concerns confluence defined in terms of alternative derivations ending in the exact same state, which excludes a large class of interesting CHR programs. However, the literature on confluence in general rewriting systems has, since the early 1970s, offered a more general notion of confluence modulo an equivalence relation; this defines that alternative derivations only need to end in states that are equivalent with respect to some equivalence relation (and not necessarily identical). In this paper, we show how confluence modulo equivalence can be applied in a CHR context, and we demonstrate interesting programs covered by this notion that are not confluent in any previous

* The second author's contribution has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

Presented at LOPSTR 2014,
Logic-Based Program Synthesis and Transformation,
Canterbury, UK, September 9-11, 2014.
Preliminary version; to be revised and published in a volume of
Lecture Notes in Artificial Intelligence, 2014

versions. The use of redundant data representations is one example of what becomes within reach of confluence, and programs that search for one best among multitudes of alternative solutions is another.

Example 1. The following CHR program, consisting of a single rule, collects a number of separate items into a (multi-) set represented as a list of items.

```
set(L), item(A) <=> set([A|L]).
```

This rule will apply repeatedly, replacing constraints matched by the left hand side by those indicated to the right. The query

```
?- item(a), item(b), set([]).
```

may lead to two different final states, $\{$`set([a,b])`$\}$ and $\{$`set([b,a])`$\}$, both representing the same set. This can be formalized by a state equivalence relation $\approx$ that implies $\{\texttt{set(}L\texttt{)}\} \approx \{\texttt{set(}L'\texttt{)}\}$, whenever L is a permutation of L'. The program is not confluent when identical end states are required, but it will be shown to be confluent modulo $\approx$.

Our generalization is based on a new operational semantics that permits extra-logical and incomplete predicates (e.g., Prolog's `var/2` and `is/2`), which is out of the scope of previous approaches. It also leads to a noticeable reduction of notational overhead due to a simpler structure of states.

Section 2 reviews previous work on confluence, in general and for CHR. Sections 3 and 4 give preliminaries and our operational semantics. Section 5 considers how to prove confluence modulo equivalence for CHR. Section 6 shows confluence modulo equivalence for a version in CHR of the Viterbi algorithm; it represents a wider class of dynamic programming algorithms with pruning, also outside the scope of earlier proposals. Section 7 shows confluence modulo equivalence for the Union-Find algorithm, which has become a standard test case for confluence in CHR; it is not confluent in any previously proposed way (except with construed side-conditions). Section 8 comments on related work in more details, and the final section provides a summary and a conclusion.

2 Background

A binary *relation* $\rightarrow$ on a set A is a subset of $A \times A$, where $x \rightarrow y$ denotes membership of $\rightarrow$. A *rewrite system* is a pair $\langle A, \rightarrow \rangle$; it is *terminating* if there is no infinite chain $a_0 \rightarrow a_1 \rightarrow \cdots$. The *reflexive transitive closure* of $\rightarrow$ is denoted $\overset{*}{\rightarrow}$. The *inverse relation* $\leftarrow$ is defined by $\{(y, x) \mid x \rightarrow y\}$. An *equivalence (relation)* $\approx$ is a binary relation on A that is reflexive, transitive and symmetric.

A rewrite system $\langle A, \rightarrow \rangle$ is *confluent* if and only if $y \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y' \Rightarrow \exists z.\ y \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y'$, and is *locally confluent* if and only if $y \leftarrow x \rightarrow y' \Rightarrow \exists z.\ y \overset{*}{\rightarrow} z \overset{*}{\leftarrow} z'$. In 1942, Newman showed his fundamental lemma [1]: *A terminating rewrite system is confluent if and only if it is locally confluent.* An elegant proof of Newman's lemma was provided by Huet [2] in 1980.

The more general notion of *confluence modulo equivalence* was introduced in 1972 by Aho et al. [3] in the context of the Church-Rosser property.

Definition 1 (Confluence modulo equivalence). *A relation* $\rightarrow$ *is confluent modulo an equivalence* $\approx$ *if and only if*

$$\forall x, y, x', y'.\quad x' \overset{*}{\leftarrow} x \approx y \overset{*}{\rightarrow} y' \qquad \Rightarrow \qquad \exists z, z'.\quad x' \overset{*}{\rightarrow} z \approx z' \overset{*}{\leftarrow} y'$$

This shown as a diagram in Fig. 1a. Sethi [4] showed in 1974 that confluence modulo equivalence for a bounded rewrite system is equivalent to the following properties, α and β; see also Fig. 1b.

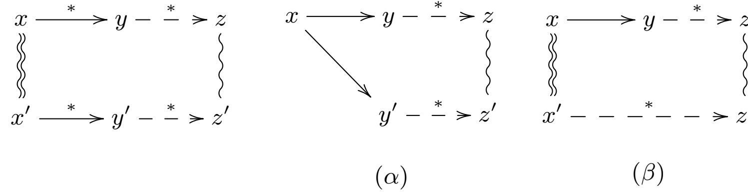
Definition 2 (α & β). *A relation* $\rightarrow$ *has the* α *property and the* β *property if and only if it satisfy the* α *condition and the* β *condition, respectively:*

$$\alpha: \qquad \forall x, y, y'.\quad y' \leftarrow x \rightarrow y \qquad \Longrightarrow \qquad \exists z, z'.\quad y' \overset{*}{\rightarrow} z \approx z' \overset{*}{\leftarrow} y$$
$$\beta: \qquad \forall x, x', y.\quad x' \approx x \rightarrow y \qquad \Longrightarrow \qquad \exists z, z'.\quad x' \overset{*}{\rightarrow} z \approx z' \overset{*}{\leftarrow} y$$

In 1980, Huet [2] generalized this result to any terminating system.

Definition 3 (Local confl. mod. equivalence). *A rewrite system is* locally confluent modulo an equivalence $\approx$ *if and only if it has the* α *and* β *properties.*

Theorem 1. *Let* $\rightarrow$ *be a terminating relation. For any equivalence* $\approx$*,* $\rightarrow$ *is confluent modulo* $\approx$ *if and only if* $\rightarrow$ *is locally confluent modulo* $\approx$*.*



(a) Confluence modulo $\approx$.

(b) Local Confluence modulo $\approx$.

Fig. 1: Diagrams for the fundamental notions. A dotted arrow (single wave line) indicates an inferred step (inferred equivalence).

The known results on confluence for CHR are based on Newman's lemma. Abdennadher *et al* [5] in 1996 seem to be the first to consider this, and they could show that confluence (without equivalence) for CHR is decidable and can be checked by examining a finite set of states formed by a combination of heads of rules. A refinement, called observational confluence was introduced in 2007 by Duck *et al* [6], in which only states that satisfy a given invariant are considered.

3 Preliminaries

We assume standard notions of first-order logic such as predicates, atoms and terms. For any expression E, $\mathit{vars}(E)$ refers to the set of variables that occurs

in E. A substitution is a mapping from a finite set of variables to terms, which also may be viewed as a set of first-order equations. For substitution σ and expression E, $E\sigma$ (or $E \cdot \sigma$) denotes the expression that arises when σ is applied to E; composition of two substitutions σ, τ is denoted $\sigma \circ \tau$. Special substitutions *failure*, *error* are assumed, the first one representing falsity and the second runtime errors.

Two disjoint sets of *(user) constraints* and *built-in* predicates are assumed. For the built-ins, we use a semantics that is more in line with implemented CHR systems than previous approaches, and which allows not only logical built-ins but also extra-logical devices such as Prolog's `var/1` and incomplete ones such as `is/2`. While [5,6,7] collect built-ins in a separate store and determine their satisfiability by a magic solver that mirrors a first-order semantics, we execute a built-in right away, which means that it serves as a test, possible giving rise to a substitution that is applied to form the subsequent state.

An evaluation procedure *Exe* for built-ins b is assumed, such that $\mathit{Exe}(b)$ is either a (possibly identity) substitution to a subset of $\mathit{vars}(b)$ or one of *failure*, *error*. It extends to sequences of built-ins as follows.

$$\mathit{Exe}((b_1, b_2)) = \begin{cases} \mathit{Exe}(b_1) & \text{when } \mathit{Exe}(b_1) \in \{\mathit{failure}, \mathit{error}\}, \\ \mathit{Exe}(b_2 \cdot \mathit{Exe}(b_1)) & \text{when otherwise } \mathit{Exe}(b_2 \cdot \mathit{Exe}(b_1)) \\ & \qquad \in \{\mathit{failure}, \mathit{error}\}, \\ \mathit{Exe}(b_1) \circ \mathit{Exe}(b_2 \cdot \mathit{Exe}(b_1)) & \text{otherwise} \end{cases}$$

A subset of built-in predicates are the *logical* ones, whose meaning is given by a first-order theory $\mathcal{B}$. For a logical atom b with $\mathit{Exe}(b) \neq \mathit{error}$, the following conditions must hold.

- Partial correctness: $\mathcal{B} \models \forall_{\mathit{vars}(b)}(b \leftrightarrow \exists_{\mathit{vars}(\mathit{Exe}(b))\setminus \mathit{vars}(b)} \mathit{Exe}(b))$.
- Instantiation monotonicity: $\mathit{Exe}(b \cdot \sigma)) \neq \mathit{error}$ for all substitutions σ.

A logical predicate p is *complete* whenever, for any p atom b that $\mathit{Exe}(b) \neq \mathit{error}$; later we define completeness with respect to a state invariant. Any built-in predicate which is not logical is called *extra-logical.* The following predicates are examples of built-ins; ϵ is the empty substitution.

1. $\mathit{Exe}(t\ \texttt{=}\ t') = \sigma$ where σ is a most general unifier of t and t'; if no such unifier exists, the result is *failure*.
2. $\mathit{Exe}(\texttt{true})$ is ϵ.
3. $\mathit{Exe}(\texttt{fail})$ is *failure*.
4. $\mathit{Exe}(t\ \texttt{is}\ t') = \mathit{Exe}(t\ \texttt{=}\ v)$ whenever t' is a ground term that can be interpreted as an arithmetic expression e with the value v; if no such e exists, the result is *error*.
5. $\mathit{Exe}(\texttt{var}(t))$ is ϵ if t is a variable and *failure* otherwise.
6. $\mathit{Exe}(\texttt{ground}(t))$ is ϵ when t is ground and *failure* otherwise.
7. $\mathit{Exe}(t\ \texttt{==}\ t')$ is ϵ when t and t' are identical and *failure* otherwise.
8. $\mathit{Exe}(t\ \texttt{\textbackslash=}\ t')$ is ϵ when t and t' are non-unifiable and *failure* otherwise.

The first three predicates are logical and complete; "`is`" is logical but not complete without an invariant that grounds its second arguments (considered later). The remaining ones are extra-logical.

The practice in previous semantics [5,6,7] of conjoining built-ins and testing them by satisfiability leads to ignorance of runtime errors and incompleteness.

To represent the propagation history, we introduce *indices:* An *indexed set* S is a set of items of the form $x{:}i$ where i belongs to some index set and each such i is unique in S. When clear from context, we may identify an indexed set S with its cleaned version $\{x \mid x{:}i \in S\}$. Similarly, the item x may identify the indexed version $x{:}i$. We extend this to any structure built from indexed items.

4 Constraint Handling Rules

We define an abstract syntax of CHR together with an operational semantics suitable for considering confluence. We use the *generalized simpagation* form as a common representation for the rules of CHR. Guards may unify variables that occur in rule bodies, but not variables that occur in the matched constraints. In accordance with the standard behaviour of implemented CHR systems, failure and runtime errors are treated the same way in the evaluation of a guard, but distinguished when occurring in a query or rule body, cf. definition 7, below.

Definition 4. *A* rule r *is of the form*

$$H_1 \setminus H_2 \texttt{ <=> } g \mid C,$$

where H_1 and H_2 are sequences of constraints, forming the head *of the rule, g is a* guard *being a sequence of built-ins, and C is a sequences of constraints and built-ins called the* body *of r. Any of H_1 and H_2, but not both, may be empty. A* program *is a finite set of rules.*

For any fresh variant of rule r with notation as above, an application instance *r'' is given as follows.*

1. *Let r' be a structure of the form*
   $H_1\tau \setminus H_2\tau \texttt{ <=> } C\tau\sigma$
   where τ is a substitution for the variables of H_1, H_2, $Exe(g\tau) = \sigma \notin \{failure, error\}$, and it holds that $(H_1 \setminus H_2)\tau = (H_1 \setminus H_2)\tau\sigma$,
2. *r'' is a copy of r' in which each atom in its head and body is given a unique index, where the indices used for the body are new and unused.*

The application record *for such an application instance is of the form*

$$r\,@\,i_1, \ldots, i_n$$

where $i_1, \ldots, i_n$ is the sequence of indices of H_1, H_2 in the order they occur.

Guards are removed from application instances as they are *a priori* satisfied.

A rule is a *simplification* when H_1 is empty, a *propagation* when H_2 is empty; in both cases, the backslash is left out, and for a propagation, the arrow symbol is written ==>. Any other rule is a *simpagation.* The following definition will become useful later on when we consider confluence.

Definition 5. *Consider two application instances* $r_i = (A_i \setminus B_i \mathrel{<=>} C_i)$, $i = 1, 2$. *We say that* r_1 *is* blocking r_2 *whenever* $B_1 \cap (A_2 \cup B_2) \neq \emptyset$.

For this to be the case, r_1 must be a simplification or simpagation. Intuitively, it means that if r_1 has been applied to a state, it is not possible subsequently to apply r_2. In the following definition of execution states for CHR, irrelevant details of the state representation are abstracted away using principles of [8]. To keep notation consistent with Section 2, we use letters such as x, y, etc. for states.

Definition 6. *A* state representation *is a pair* $\langle S, T \rangle$, *where*

- S *is a finite, indexed set of atoms called the* constraint store*,*
- T *is a set of application records called the* propagation history*.*

Two state representations S_1 *and* S_2 *are* isomorphic*, denoted* $S_1 \equiv S_2$ *whenever one can be derived from the other by a renaming of variables and a consistent replacement of indices (i.e., by a 1-1 mapping). When* Σ *is the set of all state representations, a* state *is an element of* $\Sigma/_{\equiv} \cup \{\mathit{failure}, \mathit{error}\}$*, i.e., an equivalence class in* Σ *induced by* $\equiv$ *or one of two special states; applying the failure (error) substitution to a state yields the failure (error) state. To indicate a given state, we may for simplicity mention one of its representations.*

A query q *is a conjunction of constraints, which is also identified with an initial state* $\langle q', \emptyset \rangle$ *where* q' *is an indexed version of* q.

To make statements about, say, two states x, y and an instance of a rule r, we may do so mentioning state representatives x', y' and application instance r' having recurring indices.

In contrast to [5,6,7], we have excluded global variables as they are easy to simulate: A query $q(X)$ is extended to $\mathit{global}('X', X), q(X)$, where $\mathit{global}/2$ is a new constraint predicate; $'X'$ is a constant that serves as a name of the variable. The value *val* for X is found in the final state in the unique constraint $\mathit{global}('X', \mathit{val})$. References [5,6,7] use a state component for constraints waiting to be processed, plus a separate derivation step to introduce them into the constraint store. This is unnecessary as the derivations made under either premises are basically the same. Our derivation relation is defined as follows; here and in the rest of this paper, $\uplus$ denotes union of disjoint sets.

Definition 7. *A* derivation step $\mapsto$ *from one state to another can be of two types: by rule* $\stackrel{r}{\mapsto}$ *or by built-in* $\stackrel{b}{\mapsto}$*, defined as follows.*

Apply: $\langle S \uplus H_1 \uplus H_2, T \rangle \stackrel{r}{\mapsto} \langle S \uplus H_1 \uplus C, T' \rangle$
whenever there is an application instance r *of the form* $H_1 \setminus H_2 \mathrel{<=>} C$ *with* applied$(r) \notin T$*, and* T' *is derived from* T *by 1) removing any application record having an index in* H_2 *and 2) adding* applied(r) *in case* r *is a propagation.*

Built-in: $\langle \{b\} \uplus S, T \rangle \stackrel{b}{\mapsto} \langle S, T \rangle \cdot \mathit{Exe}(b)$.

A state z is final *for query q, when $q \overset{*}{\mapsto} z$ and no step is possible from z.*

The removal of certain application records in Apply steps means to keep only those records that are essential for preventing repeated application of the same rule to the same constraints (identified by their indices).

As noticed by [6], an invariant can make more programs confluent as unusual states, that never appears in practice, are avoided. An invariant may also make it easier to characterize an equivalence relation for states.

Definition 8. *An* invariant *is a property $I(\cdot)$ which may or may not hold for a state, such that $I(x) \wedge x \mapsto y \Rightarrow I(y)$. An I-*state *is a state x with $I(x)$, and an I-*derivation *is one starting from an I-state. A program is I-*terminating *whenever all I-derivations are terminating. A set of* allowed queries *Q may be specified, giving rise to an invariant $reachable_Q(x) \Leftrightarrow \exists q \in Q \colon q \overset{*}{\mapsto} x$.*

A (state) equivalence *is an equivalence relation $\approx$ on the set of I-states.*

Theorem 1 applies specifically for CHR programs equipped with invariant I and equivalence relation $\approx$. When $\approx$ is identity, it coincides with a theorem of [6] for observable confluence. If, furthermore, $I \Leftrightarrow true$, we obtain the classical confluence results for CHR [9].

The following definition is useful when considering confluence for programs that use Prolog built-ins such as "`is/2`".

Definition 9. *A logical predicate p is* complete with respect to invariant I *whenever, for any p atom b in some I-state that $Exe(b) \neq error$.*

As promised earlier, "`is/2`" is complete with respect to an invariant that guarantees groundness of the second argument of any call to "`is/2`".

Example 2. Our semantics permits CHR programs that define constraints such as Prolog's `dif/2` constraint and a safer version of `is/2`.

```
dif(X,Y) <=> X==Y | fail.
dif(X,Y) <=> X\=Y | true.
X safer_is Y <=> ground(Y) | X is Y.
```

5 Proving Confluence Modulo Equivalence for CHR

We consider here ways to prove the local confluence properties α and β from which confluence modulo equivalence may follow, cf. Theorem 1. The corners in the following definition generalize the critical pairs of [5]. For ease of usage, we combine the common ancestor states with the pairs, thus the notion of corners corresponding to the "given parts" of diagrams for the α and β properties, cf. Fig. 1a. The definitions below assume a given I-terminating program with invariant I and state equivalence $\approx$. Two states x and x' are *joinable modulo* $\approx$ whenever there exist states z and z' such that $x \overset{*}{\mapsto} z \approx z' \overset{*}{\leftarrow\!\!\dashv} x'$.

Definition 10. *An* α-corner *consists of* I*-states* x, y *and* y' *with* $y \neq y'$ *and two derivation steps such that* $y \overset{\gamma}{\leftarrow\!\shortmid} x \overset{\delta}{\mapsto} y'$. *An* α*-corner is* joinable modulo $\approx$ *whenever* y *and* y' *are joinable modulo* $\approx$.

A β-corner *consists of* I*-states* x, x' *and* y *with* $x \neq x'$ *and a derivation step such that* $x' \approx x \overset{\gamma}{\mapsto} y$. *A* β*-corner is joinable modulo* $\approx$ *whenever* x' *and* y *are joinable modulo* $\approx$.

Some corners are critical, meaning that their satisfaction of the α or β property is not trivial.

Definition 11. *An* α*-corner* $y \overset{\gamma}{\leftarrow\!\shortmid} x \overset{\delta}{\mapsto} y'$ *or* β*-corner* $x' \approx x \overset{\gamma}{\mapsto} y$ *is* critical *if one of the following properties holds.*

α_1: *γ and δ are application instances where γ blocks δ (Def. 5).*
α_2: *γ is an application instance of a rule whose guard g contains an extra-logical built-in, and δ is a built-in with $vars(g) \cap vars(\delta) \neq \emptyset$.*
α_3: *γ and δ are built-ins with γ being extra-logical or not complete wrt. I, and $vars(\gamma) \cap vars(\delta) \neq \emptyset$.*
β: *there exists no state y' and single derivation step of x' such that $x' \overset{\delta}{\mapsto} y' \approx y$.*

Critical β-corners are motivated by the experience that often the δ step can be formed trivially by applying the same rule or built-in of γ in an analogous way to the state x'. By inspection and Theorem 1, we get the following.

Lemma 1. *Any non-critical corner is joinable modulo* $\approx$.

Theorem 2. *A terminating program is confluent modulo* $\approx$ *if and only if all its critical corners are joinable modulo* $\approx$.

Without invariant, equivalence and extra-logicals, the only critical corners are of type α_1; here [5] has shown that joinability of a finite set of minimal critical pairs is sufficient to ensure local confluence. In our case, this cannot be reduced to checking such minimal states, but the construction is useful as a way to group the cases that need to be considered. We adapt the definition of [5] as follows.

Definition 12. *An* α_1-critical pattern (with evaluated guards) *is of the form*

$$\langle S_1\sigma_1, \{R_1\}\rangle \overset{r_1}{\leftarrow\!\shortmid} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2\sigma_2, \{R_2\}\rangle$$

whenever there exist, for $k = 1, 2$ *indexed rules* $r_k = (A_k \setminus B_k \texttt{<=>}\ g_k \mid C_k)$, *and application record* $R_k = (r_i@i_1^k, \ldots, i_{nk}^k)$ *where* $i_1^k, \ldots, i_{nk}^k$ *is the list of indices in* A_k, B_k, *and* S, S_1 *and* S_2 *are determined in the following way.*

- *Let* $H_k = A_k \cup B_k$, $k = 1, 2$, *and split* B_1 *and* H_2 *into disjoint subsets by* $B_1 = B_1' \uplus B_1''$ *and* $H_2 = H_2' \uplus H_2''$, *where* B_1'' *and* H_2'' *must have the same number of elements* ≥ 1.
- *The set of indices used in* B_1'' *and* H_2'' *are assumed to be identical, and any other index unique, and* σ *is a most general unifier of* B_1'' *and a permutation of* H_2''.

- $S = A_1 \cup B_1 \cup A_2 \cup B_2$, $S_k = S \setminus B_k\sigma$, $k = 1,2$.
- $\sigma_k = Exe(g_k\sigma)$ *and* g_k *has no extra-logical built-ins,* $k = 1,2$.

An α_1-critical pattern (with delayed guards) *is of the form*

$$\langle S_1, \{R_1\}\rangle \overset{r_1}{\hookleftarrow} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2, \{R_2\}\rangle,$$

where all parts are defined as above, except in the last step, that one of g_k *contains either an extra-logical built-in or its evaluation leads to a logical built-in* b *with* $Exe(b) = error$*; the guards* $g_k\sigma$ *are recognized as the* unevaluated guards.

The constraints needed to produce the derivation steps that ensure joinability may not appear in the patterns of Definition 12, but are implied by the invariant. To cope with this, we proceed as follows.

Definition 13. *An* α_1*-critical corner* $y \overset{r_1}{\hookleftarrow} x \overset{r_2}{\mapsto} y'$ *is* covered by *an* α_1*-critical pattern with evaluated guards of the form* $\langle S_1, A_1\rangle \overset{r_1}{\hookleftarrow} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2, A_2\rangle$*, whenever there exist a set of indexed constraints* S^+*, a substitution* σ^+ *and a set of application records* A^+ *such that* $I(\langle S\sigma^+ \cup S^+, A^+\rangle)$ *holds and*

$$y = \langle S_1\sigma^+ \cup S^+, A_1 \cup A^+\rangle,\ x = \langle S\sigma^+ \cup S^+, A^+\rangle,\ y' = \langle S_2\sigma^+ \cup S^+, A_2 \cup A^+\rangle.$$

An α_1*-critical corner* $y \overset{r_1}{\hookleftarrow} x \overset{r_2}{\mapsto} y'$ *is* covered by *an* α_1*-critical pattern with delayed guards of the form* $\langle S_1, A_1\rangle \overset{r_1}{\hookleftarrow} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2, A_2\rangle$*, whenever there exist a set of indexed constraints* S^+*, a substitution* σ^+ *and a set of application records* A^+ *such that* $I(\langle S\sigma^+ \cup S^+, A^+\rangle)$ *holds and*

$$y = \langle (S_1\sigma^+ \cup S^+)Exe(g_1\sigma^+), A_1 \cup A^+\rangle,\ x = \langle S\sigma^+ \cup S^+, A^+\rangle,$$
$$y' = \langle (S_2\sigma^+ \cup S^+)Exe(g_2\sigma^+), A_2 \cup A^+\rangle;$$

g_1, g_2 *are the unevaluated guards of the pattern.*

Analogously to previous results on confluence of CHR, we can state the following.

Lemma 2. *For a given* I*-terminating program with invariant* I *and equivalence* $\approx$*, the set of critical* α_1*-patterns is finite, and any critical* α_1*-corner is covered by some critical* α_1*-pattern.*

Our work is currently intended for manual proofs and we do not assume any formal and decidable ways of specifying I and $\approx$, which would be needed for fully mechanizable proof methods. It is straightforward to define critical α_2- and α_3-patterns and formulate analogous versions of Lemma 2 (left out for reasons of space). These are not needed in our examples as the α_2 and α_3 properties will appear to be trivial. It is not possible to cover β-corners by patterns in a similarly syntactic way as two of the participating states are related only semantically by $\approx$. However, we can reuse the developments of [5] and joinability results derived by their methods, e.g., using automatic checkers for classical confluence [10].

Lemma 3. *If a critical α_1-pattern π (viewed as an α_1-corner) is joinable under I and the identity equivalence, any α_1-corner covered by π is joinable under I and $\approx$.*

Example 3 (example 1, ct'd). We formulate invariant and equivalence and prove confluence. The propagation history can be ignored as there are no propagations.

I: $I(x)$ holds if and only if $x = \{\texttt{set}(L)\} \cup \mathit{Items}$, where *Items* is a set of `item/1` constraints whose argument is a constant and L a list of constants.

$\approx$: $x \approx x'$ if and only if $x = \{\texttt{set}(L)\} \cup \mathit{Items}$ and $x' = \{\texttt{set}(L')\} \cup \mathit{Items}$ where *Items* is a set of `item/1` constraints and L is a permutation of L'.

There are no built-ins and thus no critical α_2- or α_3-patterns. There is only one critical α_1-pattern, namely

$$\{\texttt{set([}B\texttt{|}L\texttt{])}, \texttt{item(}A\texttt{)}\} \leftarrowtail \{\texttt{set(}L\texttt{)}, \texttt{item(}A\texttt{)}, \texttt{item(}B\texttt{)}\} \mapsto \{\texttt{set([}A\texttt{|}L\texttt{])}, \texttt{item(}B\texttt{)}\},$$

where L, A, and B are terms such that I holds for the indicate states. Joinability for any corner covered by this pattern is shown by applying the rule to the two "wing" states to form two states $\{\mathit{set}([\mathrm{B,A,}|\mathrm{L}])\} \approx \mathit{set}([\mathrm{A,B,}|\mathrm{L}])\}$.

To check the β property, we notice that any β-corner is of the form

$$\{\texttt{set(}L'\texttt{)}, \texttt{item(}A\texttt{)}\} \cup \mathit{Items} \approx \{\texttt{set(}L\texttt{)}, \texttt{item(}A\texttt{)}\} \cup \mathit{Items} \mapsto \{\texttt{set([}A\texttt{|}L\texttt{])}\} \cup \mathit{Items}$$

where L and L' are lists, one being a permutation of the other. Applying the rule to the "left wing" state leads to $\{\texttt{set([}A\texttt{|}L'\texttt{])}\} \cup \mathit{Items}$ which is equivalent (wrt. $\approx$) to the "right wing" state. As the program is clearly I-terminating, it follows that it is confluent modulo $\approx$.

6 Confluence of Viterbi Modulo Equivalence

Dynamic programming algorithms produce solutions to a problem by generating solutions to growing sub-problems, extending those solutions already found. The Viterbi algorithm [11] finds a most probable path of state transitions in a Hidden Markov Model (HMM) that produces a given emission sequence Ls, also called the *decoding* of Ls; see [12] for a background on HMMs. There may be exponentially many paths but an early pruning strategy ensures linear time. The algorithm has been studied in CHR by [13], starting from the following program.

```
:- chr_constraint path/4, trans/3, emit/3.

expand @ trans(Q,Q1,PT), emit(Q,L,PE), path([L|Ls],Q,P,PathRev) ==>
   P1 is P*PT*PE  |  path(Ls,Q1,P1,[Q1|PathRev]).

prune @ path(Ls,Q,P1,_) \ path(Ls,Q,P2,_) <=> P1 >= P2 | true.
```

The meaning of a constraint $\texttt{path(}Ls\texttt{,}q\texttt{,}p\texttt{,}R\texttt{)}$ is that Ls is a remaining emission sequence to be processed, q the current state of the HMM, and p the probability of a path R found for the already processed prefix of the emission sequence.

To simplify the program, a path is represented in reverse order. Constraint `trans(`q`,`q'`,`pt`)` indicates a transition from state q to q' with probability pt, and `emit(`q`,`ℓ`,`pe`)` a probability pe for emitting letter ℓ in state q.

Decoding of a sequence Ls is stated by the query "*HMM*, `path(`Ls`,q0,1,[])`" where *HMM* is an encoding of a particular HMM in terms of `trans` and `emit` constraints. Assuming *HMM* and Ls be fixed, the state invariant I is given as reachability from the indicated query. The program is I-terminating, as a new `path` constraint introduced by the expand rule has a first argument shorter than that of its predecessor. Depending on the application order, it may run in between linear and exponential time, and [13] proceeds by semantics preserving program transformations that lead to an optimal execution order.

The program is not confluent in the classical sense, i.e., without an equivalence, as the `prune` rule may need to select one out of two different and equally probable paths. A suitable state equivalence may be defined as follows.

Definition 14. *Let* $\langle \mathit{HMM} \cup \mathit{PATHS}_1, T\rangle \approx \langle \mathit{HMM} \cup \mathit{PATHS}_2, T\rangle$ *whenever: For any indexed constraint* $(i$: `path(`Ls`,`q`,`P`,`R_1`)`$) \in \mathit{PATHS}_1$ *there is another* $(i$: `path(`Ls`,`q`,`P`,`R_2`)`$) \in \mathit{PATHS}_2$ *and vice versa.*

The built-ins used in guards, `is/2` and `>=/2`, are logical and complete with respect to I so there are no α_2- or α_3-critical corners. For simplicity of notation, we ignore the propagation histories. There are three critical α_1 patterns to consider: *(i)* $y \overset{\texttt{prune}}{\leftarrowtail} x \overset{\texttt{prune}}{\mapsto} y'$, where x contains two `path` constraints that may differ only in their last arguments, and y and y' differ only in which of these constraints that are preserved; thus $y \approx y'$.
(ii) $y \overset{\texttt{prune}}{\leftarrowtail} x \overset{\texttt{expand}}{\mapsto} y'$ where $x = \{\pi_1, \pi_2, \tau, \eta\}$, π_i = `path(`L`,`q`,`P_i`,`R_i`)` for $i = 1, 2$, $P_1 \geq P_2$, and τ, η the `trans` and `emit` constraints used for the expansion step. Thus $y = \{\pi_1, \tau, \eta\}$ and $y' = \{\pi_1, \pi_2, \pi_2', \tau, \eta\}$ where π_2' is expanded from π_2. To show joinability, we show the stronger property of the existence of a state z with $y \overset{*}{\mapsto} z \overset{*}{\leftarrowtail} y'$. We select $z = \{\pi_1, \pi_1', \tau, \eta\}$, where π_1' is expanded from π_1.[1] The probability in π_1' is greater or equal to that of π_2', which means that a pruning of π_2' is possible when both are present. Joinability is shown as follows.

$$y \overset{\texttt{expand}}{\mapsto} z \overset{\texttt{prune}}{\leftarrowtail} \{\pi_1, \pi_1', \pi_2, \tau, \eta\} \overset{\texttt{prune}}{\leftarrowtail} \{\pi_1, \pi_1', \pi_2, \pi_2', \tau, \eta\} \overset{\texttt{expand}}{\leftarrowtail} y'$$

(iii) As case *ii* but with $P_2 \geq P_1$ and $y = \{\pi_2, \tau, \eta\}$; proof similar and omitted.

Thus all α-critical corners are joinable. There are no critical β corners, as whenever $x' \approx x \overset{r}{\mapsto} y$, the rule r can apply to x' with an analogous result, i.e., there exists a state y' such that $x' \overset{r}{\mapsto} y' \approx y$. This finishes the proof of confluence modulo $\approx$.

[1] It may be the case that π_1' was produced and pruned at an earlier stage, so the propagation history prevents the creation of π_1' anew. A detailed argument can show, that in this case, there will be another constraints π_1'' in the store similar to π_1' but with a $\geq$ probability, and π_1'' can be used for pruning π_2' and obtain the desired result in that way.

7 Confluence of Union-Find Modulo Equivalence

The Union-Find algorithm [14] maintains a collection of disjoint sets under union, with each set represented as a tree. It has been implemented in CHR by [15] who proved it nonconfluent using critical pairs [5]. We have adapted a version from [6], extending it with a new `token` constraint to be explained; let $UF_{\texttt{token}}$ refer to our program and UF_0 to the original without `token` constraints.

```
union    @ token, union(A,B) <=> find(A,X), find(B,Y), link(X,Y).
findNode @ A ~> B \ find(A,X) <=> find(B,X).
findRoot @ root(A) \ find(A,X) <=> A=X.
linkEq   @ link(A,A) <=> token.
link     @ root(A) \ link(A,B), root(B) <=> B ~> A, token.
```

The `~>` and `root` constraints, called *tree constraints*, represent a set of trees. A finite set T of ground tree constraints is *consistent* whenever: for any constant a in T, there is either one and only one $\texttt{root}(a) \in T$, or a is connected via a unique chain of `~>` constraints to some r with $\texttt{root}(r) \in T$. We define $sets(T)$ to be the set of sets represented by T, formally: the smallest equivalence relation over constants in T that contains the reflexive, the transitive closure of `~>`; $set(a,T)$ refers to the set in $sets(T)$ containing constant a.

Our *allowed queries* are ground and of the form $T \cup U \cup \{\texttt{token}\}$, where T is a consistent set of tree constraints, and U is a set of constraints $\texttt{union}(a_i,b_i)$, where a_i, b_i appear in T. The invariant I is defined by reachability from these queries. By induction, we can show the following properties of any I-state S.

- Either $S = T \cup U \cup \{\texttt{token}\}$, where T is a consistent set of tree constraints and U a set of `union` constraints whose arguments are in T, or
- $S = T \cup U \cup \{\texttt{link}(A_1,A_2)\} \cup F_1 \cup F_2$ where T, U are as in the previous case, and for $i = 1, 2$,
  - if A_i is a constant, $F_i = \emptyset$, otherwise
  - $F_i = \{\texttt{find}(a_i,A_i)\}$ or $F_i = \{(a_i = A_i)\}$ for some constant a_i.

As shown by [15], UF_0 is not confluent in the classical sense, which can be related to the following issues.

(i) When the detailed steps of two `union` operations are intertwined in an unfortunate way, the program may get stuck in a state where it cannot finish the operation as shown in the following derivation.
`root(a), root(b), root(c), union(a,b), union(b,c)` $\overset{*}{\mapsto}$
`root(a), root(b), root(c), link(a,b), link(b,c)` $\mapsto$
`b ~> a, root(a), root(c), link(b,c)`

(ii) Different execution orders of the `union` operations may lead to different data structures (representing the same sets). This is shown in the following derivations from a query $q_0 = \{\texttt{root(a)}, \texttt{root(b)}, \texttt{root(c)}, \texttt{union(a,b)}, \texttt{union(b,c)}\}$.
$q_0 \overset{*}{\mapsto}$ `root(a), root(c), b ~> a, union(b,c)` $\overset{*}{\mapsto}$ `root(a), b ~> a, c ~> a`
$q_0 \overset{*}{\mapsto}$ `root(a), root(b), c ~> b, union(a,b)` $\overset{*}{\mapsto}$ `root(b), b ~> a, c ~> b`

We proceed, now, to show that $UF_{\texttt{token}}$ is confluent modulo an equivalence $\approx$, defined as follows.

- $T \cup U \cup \{\texttt{token}\} \approx T' \cup U \cup \{\texttt{token}\}$ whenever $sets(T) = sets(T')$.
- $T \cup U \cup \{\texttt{link}(A_1,A_2)\} \cup F_1 \cup F_2 \approx T' \cup U \cup \{\texttt{link}(A'_1,A'_2)\} \cup F'_1 \cup F'_2$ whenever $sets(T) = sets(T')$ and for $i = 1, 2$, that
  - if A_i is a constant and (by I) $F_i = \emptyset$, then A'_i is a constant, $set(A_i, T) = set(A'_i, T')$ and $F'_i = \emptyset$
  - if A_i is a variable and $F_i = \{\texttt{find}(a_i, A_i)\}$ for some constant a_i, then $F'_i = \{\texttt{find}(a'_i, A'_i)\}$ and $set(a_i, T) = set(a'_i, T')$,
  - if A_i is a variable, $F_i = \{(a_i = A_i)\}$ for some constant a_i with $\texttt{root}(a_i) \in T$ then $F'_i = (a'_i = A'_i)\}$, $\texttt{root}(a'_i) \in T'$ and $set(a_i, T) = set(a'_i, T')$.

There are no critical α_2- and α_3-patterns. The α_1-patterns (critical pairs) of $UF_{\texttt{token}}$ are those of UF_0 and a new one, formed by an overlap of the `union` rule with itself. We reuse the analysis of [15] who identified all critical pairs for UF_0; by Lemma 3, we consider only those pairs, they identified as non-joinable.

They identified eight non-joinable critical pairs; the first one ("the unavoidable" pair) concerns issue *(ii)*. Its ancestor state $\{\texttt{find}(B, A), \texttt{root}(B), \texttt{root}(C), \texttt{link}(C, B)\}$, is excluded by I: any corner covered, B and C must be ground, thus also the `link` constraint, which according to I excludes a `find` constraint. This can be traced to the effect of our `token` constraint, that forces any `union` to complete its detailed steps, before a next `union` may be entered. However, issue *(ii)* pops up in the new pattern for $UF_{\texttt{token}}$, $y \leftarrowtail x \mapsto y'$ where:

$$
\begin{aligned}
x &= \{\texttt{token}, \texttt{union}(A, B), \texttt{union}(A', B')\}\\
y &= \{\texttt{find}(A, X), \texttt{find}(B, Y), \texttt{link}(X, Y), \texttt{union}(A', B')\}\\
y' &= \{\texttt{find}(A', X'), \texttt{find}(B', Y'), \texttt{link}(X', Y'), \texttt{union}(A, B)\}
\end{aligned}
$$

To show joinability of any corner covered by this pattern means to find z, z' such that $y \overset{*}{\mapsto} z \approx z' \overset{*}{\leftarrowtail} y'$. This can be done by, from y, first executing all remaining steps related to $\texttt{union}(A, B)$ and then the steps relating to $\texttt{union}(A', B')$ to reach a state $z = T \cup U \cup \{\texttt{token}\}$. In a similar way, we construct $z' = T' \cup U \cup \{\texttt{token}\}$, starting with the steps relating to $\texttt{union}(A', B')$ followed by those of $\texttt{union}(A, B)$. It can be proved by induction that $sets(T) = sets(T')$, thus $z \approx z'$.

Next, [15] identifies three critical pairs, that imply inconsistent tree constraints. The authors argue informally that these pairs will never occur for a query with consistent tree constraints. As noticed by [6], these arguments can be formalized using an invariant. The last four pairs of [15] relate to issue *(i)* above; [15] argues these to be avoidable when assuming procedural properties of implemented CHR systems (which may seem a bit unusual in a context concerned with confluence). In [6], those pairs are avoided by restricting allowed queries to include only a single `union` constraint; we can allow any number of those, but avoid the problem due to the control patterns imposed by the `token` constraints and formalized in our invariant I.

This finishes the argument that $UF_{\texttt{token}}$ satisfies the α property, and by inspection of the possible derivation steps one by one (for each rule and for the "=" constraint), it can be seen that there are no critical β corners. Thus $UF_{\texttt{token}}$ is locally confluent modulo $\approx$, and since tree consistency implies termination, it follows that $UF_{\texttt{token}}$ is confluent modulo $\approx$.

8 Discussion and detailed comments on related work

We already commented on the foundational work on confluence for CHR by [5], who, by the use of Newman's lemma, could devise a method to prove confluence by inspecting a finite number of critical pairs. This formed also the foundation of automatic confluence checkers [5,7,10] (with no invariant and no equivalence).

The addition of an invariant I in the specification of confluence problems for CHR was suggested by [6]. The authors considered a construction similar to our α_1-corners and critical α_1-patterns. They noted that critical α_1-patterns usually do not satisfy the invariant, so they based their approach on defining a collection of corners based on I-states as minimal extensions of such patterns. Local confluence, then, follows from joinability of this collection of minimally extended states. However, there are often infinitely many such minimally extended states; this happens even for a natural invariant such as groundness when infinitely many terms are possible, as is the case in Prolog based CHR versions. We can use this construction (in cases where it is finite!) to further cluster the space of our critical corners, but our examples worked quite well without this.

Of other work concerned with confluence for CHR, we may mention [16] who considered confluence for non-terminating CHR programs, used recently by [17] for a specific type inference problem. We may also refer to [18] that gives an overview of CHR related research until 2010, including on confluence.

9 Conclusion and future work

We have introduced confluence modulo equivalence for CHR, which allows a much larger class of programs to be characterized as confluent in a natural way, thus increasing the practical relevance of confluence for CHR.

We demonstrated the power of the framework by showing confluence modulo equivalence for programs that use a redundant data representation (the set-as-lists and Union-Find programs) and a dynamic programming algorithm (the Viterbi program); all these are out of scope of previous confluence notions for CHR. With the new operational semantics, we can also handle extra-logical and incomplete built-in predicates, and the notational improvements obtained by this semantics may also promote new research on confluence.

As a first steps towards semi- or fully automatic proof methods, it is important to notice that classical joinability of a critical pair – as can be decided by existing confluence checkers such as [10] – provide a sufficient condition for joinability modulo any equivalence.

Thus only classically non-joinable pairs – in our terminology α_1 patterns – need to be examined in more details involving the relevant equivalence; however, in some cases there may also be critical α_2, α_3 and β patterns that need to be considered. Machine supported proof methods for these parts need formal and decidable specifications of *(i)* invariant and *(ii)* equivalence, which may be supplied by additional CHR programs. For *(i)*, it may be done by checking for violations, and for *(ii)*, such a program may normalize states into a canonical form that can be compared in a straightforward way.

References

1. Newman, M.: On theories with a combinatorial definition of “equivalence”. Annals of Mathematics **43**(2) (1942) 223–243
2. Huet, G.P.: Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. J. ACM **27**(4) (1980) 797–821
3. Aho, A.V., Sethi, R., Ullman, J.D.: Code optimization and finite Church-Rosser systems. In Rustin, R., ed.: Design and Optimization of Compilers. Prentice-Hall (1972) 89–106
4. Sethi, R.: Testing for the Church-Rosser property. J. ACM **21**(4) (1974) 671–679
5. Abdennadher, S., Frühwirth, T.W., Meuss, H.: On confluence of Constraint Handling Rules. In Freuder, E.C., ed.: CP. Volume 1118 of Lecture Notes in Computer Science., Springer (1996) 1–15
6. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In Dahl, V., Niemelä, I., eds.: ICLP. Volume 4670 of Lecture Notes in Computer Science., Springer (2007) 224–239
7. Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: Proc. Logic Programming, 20th International Conference, ICLP 2004. Volume 3132 of Lecture Notes in Computer Science., Springer (2004) 90–104
8. Raiser, F., Betz, H., Frühwirth, T.W.: Equivalence of CHR states revisited. In Raiser, F., Sneyers, J., eds.: Proc. 6th International Workshop on Constraint Handling Rules. Report CW 555, Katholieke Universiteit Leuven, Belgium (2009) 33–48
9. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In Smolka, G., ed.: CP, Constraint Programming. Volume 1330 of Lecture Notes in Computer Science., Springer (1997) 252–266
10. Langbein, J., Raiser, F., Frühwirth, T.W.: A state equivalence and confluence checker for CHRs. In Weert, P.V., Koninck, L.D., eds.: Proceedings of the 7th International Workshop on Constraint Handling Rules. Report CW 588, Katholieke Universiteit Leuven, Belgium (2010) 1–8
11. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Transactions on Information Theory **13** (1967) 260–269
12. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press (1999)
13. Christiansen, H., Have, C.T., Lassen, O.T., Petit, M.: The Viterbi algorithm expressed in Constraint Handling Rules. In Van Weert, P., De Koninck, L., eds.: Proceedings of the 7th International Workshop on Constraint Handling Rules. Report CW 588, Katholieke Universiteit Leuven, Belgium (2010) 17–24
14. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2) (1984) 245–281
15. Schrijvers, T., Frühwirth, T.W.: Analysing the CHR implementation of union-find. In Wolf, A., Frühwirth, T.W., Meister, M., eds.: W(C)LP. Volume 2005-01 of Ulmer Informatik-Berichte., Universität Ulm, Germany (2005) 135–146
16. Haemmerlé, R.: Diagrammatic confluence for Constraint Handling Rules. TPLP **12**(4-5) (2012) 737–753
17. Duck, G.J., Haemmerlé, R., Sulzmann, M.: On termination, confluence and consistent CHR-based type inference. Accepted for ICLP 2014; to appear (2014)
18. Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint Handling Rules. TPLP **10**(1) (2010) 1–47