Coinductive Proofs over Streams as CHR Confluence Proofs*

Rémy Haemmerlé

Technical University of Madrid

Abstract. Coinduction is an important theoretical tool for defining and reasoning about unbounded data structures (such as streams, infinite trees, rational numbers ...), and infinite-behavior systems. Confluence is a fundamental property of Constraint Handling Rules (CHR) since, as in other rewriting formalisms, it guarantees that the computations are not dependent on rule application order, and also because it implies the logical consistency of the program's declarative view. In this paper, we illustrate how the confluence of CHR can be used to prove universal coinductive properties. In particular we give several examples of bisimulation proofs over streams.

1 Introduction

Induction and coinduction are contrasting terms for ways of describing and reasoning about a system. Whereas, the classical notion of inductive reasoning begins with some primitive properties (or definitions) and uses constructive operations on these to iteratively infer a whole set of conclusions, coinductive reasoning [4, 5, 20] starts from a set of conceivable properties (or definitions) and iteratively dismisses those that break the self-consistency of the whole set. Despite the fact that coinduction is less known than induction, it has started to receive attention in recent years in computer science. For instance, coinduction has been employed to define process equivalences in Concurrency Theory [17, 20], to study lazy evaluations in functional languages [9], or to deal with infinite data-structures and infinite computations in Logic Programming [21, 16].

Constraint Handling Rules (CHR) is a committed-choice constraint logic programming language, introduced by solvers. It has matured into a general-purpose concurrent programming language. Operationally, a CHR program consists of a set of guarded rules that rewrite multisets of constrained atoms. Declaratively, a CHR program can be viewed as a set of logical implications executed on a deduction principle.

* The research leading to these results has received funding from the Programme for Attracting Talent / young PHD of the MONTEGANCEDO Campus of International Excellence (PICD), the Madrid Regional Government under the CM project P2009/TIC/1465 (PROMETIDOS), and the Spanish Ministry of Science under the MEC project TIN-2008-05624 (DOVES).

Confluence is a basic property of rewriting systems. It refers to the fact that any two finite computations starting from a common state can be prolonged so as to eventually meet in a common state again. Confluence is an important property for any rule-based language, because it is desirable for computations to not be dependent on a particular rule application order. In the particular case of CHR, this property is even more desirable, as it guarantees the correctness of a program [3, 13]: any confluent program has a consistent logical reading.

In this paper, we illustrate how the proof of confluence of non-terminating CHR programs can be used to establish coinductive properties. In practice, we propose a simple encoding of streams[1] and bisimulation[2] as non-terminating CHR programs. Then, we explain how the confluence of the resulting programs provides an effective coinductive proof. Finally, we show how the confluence of these programs can be inferred by using a criterion we recently introduced [11]. The preliminary results presented in this paper are embedded in the more general goal of the research we started in [10]: Understanding relationships between coinduction and CHR.

The remainder of this paper is structured as follows: Sect. 2 gently introduces the notion of coinduction and stream coalgebra. In Sect. 3, we recall some preliminaries on CHR. Sect. 4 presents a criteria we recently introduced for proving confluence of non-terminating CHR programs. In Sect. 5, we present how to encode stream coalgebras in CHR. Then we show how using CHR confluence to infer bisimulation over such coalgebras, before concluding in Sect. 6.

2 Streams and Coinduction

In this section, we introduce the notion of stream, the canonical example of coinductive object. By using few bases from the theory of universal coalgebra, we explain how to define streams by coinduction. We conclude by showing how to prove equality of streams by coinduction.

This introduction is freely inspired from the one by Rutten [19]. It is deliberately kept short. The reader may refer to Rutten's works to get a more general picture of streams, coalgebras, and coinduction.

2.1 Streams

Let $\mathcal{A}$ be an arbitrary set. We define a *stream* over $\mathcal{A}$ as a function from natural numbers (the *positions*) to $\mathcal{A}$ (the *values*). Hence $\mathcal{A}^\omega$ is the set respecting the equation:

$$\mathcal{A}^\omega = (\mathbb{N} \to \mathcal{A})$$

For convenience, we may denote such a stream s by the informal notation:

$$s = [s(0), s(1), s(2), \dots]$$

[1] The canonical example of coinductive data-structure.
[2] The canonical example of coinductive property.

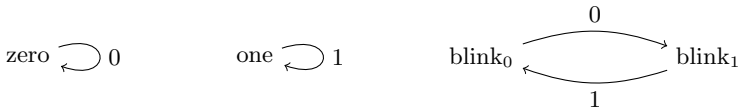

Fig. 1. LTS view of the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$.

Example 1. The stream containing only 0's (i.e. $[0, 0, 0, \dots]$) is the constant function $(x \mapsto 0)$. Similarly the stream containing only 1's (i.e. $[1, 1, 1, \dots]$) is the function $(x \mapsto 1)$. The streams alternating 0 and 1 (i.e. $[0, 1, 0, \dots]$ and $[1, 0, 1, \dots]$) are the respective functions $(x \mapsto (x \mod 2))$ and $(x \mapsto (x + 1 \mod 2))$.

Following analogy between finite lists and streams, we call *head* of a stream s the first value of s, i.e. $\mathrm{h}(s) = s(0)$ and call the *tail* of s the stream obtained by removing the head, i.e. $\mathrm{t}(s) = (x \mapsto s(x + 1))$.

Example 2. Consider the streams given in the previous example. The head and the tail of these streams respect the following equations:

$$\begin{array}{ll} \mathrm{h}([0,0,0,\dots]) = 0 & \mathrm{t}([0,0,0,\dots]) = [0,0,0,\dots] \\ \mathrm{h}([1,1,1,\dots]) = 1 & \mathrm{t}([1,1,1,\dots]) = [1,1,1,\dots] \\ \mathrm{h}([0,1,0,\dots]) = 0 & \mathrm{t}([0,1,0,\dots]) = [1,0,1,\dots] \\ \mathrm{h}([1,0,1,\dots]) = 1 & \mathrm{t}([1,0,1\dots]) = [0,1,0,\dots] \end{array}$$

2.2 Coalgebras

A *(stream) coalgebra* is a triple $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ consisting of a set $\mathcal{X}$ of *states* together with an *output function*: $\mathrm{h}_\mathcal{X} : \mathcal{X} \to \mathcal{A}$ and a *transition function* $\mathrm{t}_\mathcal{X} : \mathcal{X} \to \mathcal{X}$. In the following we may refer to these two functions as the *destructors*.

Example 3. The triple $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$ where $\mathcal{X}_3 = \{\mathrm{one}, \mathrm{zero}, \mathrm{blink}_0, \mathrm{blink}_1\}$ and $\mathrm{h}_3 : \mathcal{X}_3 \to \mathcal{A}$ and $\mathrm{t}_3 : \mathcal{X}_3 \to \mathcal{X}_3$ satisfying the equations below is a coalgebra.

$$\begin{array}{rr} \mathrm{h}_3(\mathrm{zero}) = 0 & \mathrm{t}_3(\mathrm{zero}) = \mathrm{zero} \\ \mathrm{h}_3(\mathrm{one}) = 1 & \mathrm{t}_3(\mathrm{one}) = \mathrm{one} \\ \mathrm{h}_3(\mathrm{blink}_0) = 0 & \mathrm{t}_3(\mathrm{blink}_0) = \mathrm{blink}_1 \\ \mathrm{h}_3(\mathrm{blink}_1) = 1 & \mathrm{t}_3(\mathrm{blink}_1) = \mathrm{blink}_0 \end{array}$$

Alternatively, a coalgebra can be viewed as a (possibly) infinite automaton or a Labelled Transition System (LTS) $(\mathcal{X}, \mathcal{A}, \to)$ verifying:

$$s \xrightarrow{h} t \qquad \text{if and only if} \qquad h = \mathrm{h}_\mathcal{X}(s) \;\&\; t = \mathrm{t}_\mathcal{X}(s)$$

Example 4. Figure 1 represents the coalgebra given in Example 3.

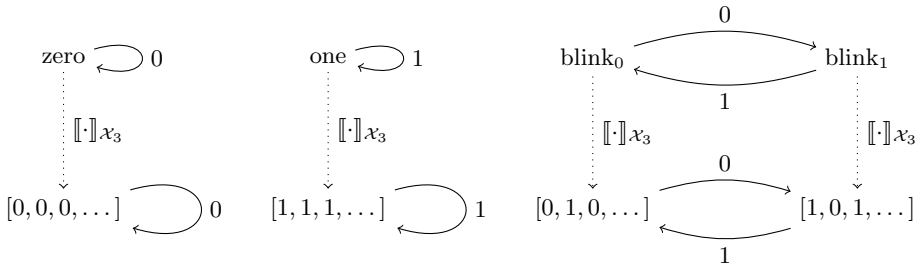

Fig. 2. LTS view of the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$.

Intuitively, within a coalgebra $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ a state $x \in \mathcal{X}$ "represents" a unique stream $s \in \mathcal{A}^\omega$, while the output and transition functions associate to x the head and (a state $x' \in \mathcal{X}$ representing) the tail of s. In order to formalize this intuition, we introduce now the notion of homomorphism and finality.

A *homomorphism* between two coalgebras $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ and $(\mathcal{Y}, \mathrm{h}_\mathcal{Y}, \mathrm{t}_\mathcal{Y})$ is a function $\phi : \mathcal{X} \to \mathcal{Y}$ that respects the destructors, i.e.:

$$\mathrm{h}_\mathcal{Y}(\phi(x)) = \mathrm{h}_\mathcal{X}(x) \qquad \text{and} \qquad \mathrm{t}_\mathcal{Y}(\phi(x)) = \phi(\mathrm{t}_\mathcal{X}(x))$$

The set of streams $\mathcal{A}^\omega$ can be viewed as the coalgebra $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$. This coalgebra has the following property:

Theorem 1 (Finality). *The coalgebra $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$ is* final *among the set of all coalgebras. That is, for any coalgebra $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ there exists a unique homomorphism ϕ from $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ to $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$.*

The finality of the set of streams gives us the formal basis to define what represents a state. Let $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ be a coalgebra. We say that a state $x \in \mathcal{X}$ represents the stream $[\![x]\!]_\mathcal{X} = \phi(x)$ where ϕ is a homomorphism from $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ to $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$. The finality of $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$ ensures us that this definition is meaningful, i.e. each state represents one and only one stream.

Example 5. Consider the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$ given in Example 3. Let $\phi : \mathcal{X}_3 \to \mathcal{A}^\omega$ be the function satisfying the equations:

$$\begin{aligned} \phi(\text{zero}) &= [0, 0, 0, \dots] & \phi(\text{blink}_0) &= [0, 1, 0, \dots] \\ \phi(\text{one}) &= [1, 1, 1, \dots] & \phi(\text{blink}_1) &= [1, 0, 1, \dots] \end{aligned}$$

One can easily verify that ϕ is a homomorphism. Hence the states zero, one, blink_0, and blink_1 respectively represent the streams $[0, 0, 0, \dots]$, $[1, 1, 1, \dots]$, $[0, 1, 0, \dots]$, and $[1, 0, 1, \dots]$. Figure 2 represents graphically this correspondence.

2.3 Proof by Coinduction

In order to prove that two streams s and s' are equal, it is necessary and sufficient to prove that

$$\text{for all } n \in \mathbb{N} \quad s(n) = s'(n)$$

An obvious method for establishing equality between streams s and s' consists of an induction on the natural number n (i.e. prove $s(0) = s'(0)$ and show that $s(n) = s'(n)$ implies $s(n+1) = s'(n+1)$). In this section, we present an alternative way, based on coinduction. This method is often more natural, especially for those streams defined using coalgebras.

A *bisimulation (relation)* between two coalgebras $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ and $(\mathcal{Y}, \mathrm{h}_\mathcal{Y}, \mathrm{t}_\mathcal{Y})$ is a relation $\mathcal{S} \subseteq \mathcal{X} \times \mathcal{Y}$ such that for all streams $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ if whenever $x \,\mathcal{S}\, y$ holds then both $\mathrm{h}_\mathcal{X}(x) = \mathrm{h}_\mathcal{Y}(y)$ and $\mathrm{t}_\mathcal{X}(x) \,\mathcal{S}\, \mathrm{t}_\mathcal{Y}(y)$. If there exists a bisimulation $\mathcal{S}$ with $x \,\mathcal{S}\, y$, we will write $x \sim y$, and say that x and y are *bisimilar*. Hence two states are bisimilar, if they have the same head and bisimilar tails. The Coinduction theory guarantees that two bisimilar states represent the same stream.

Theorem 2 (Coinduction). *Let $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ and $(\mathcal{Y}, \mathrm{h}_\mathcal{Y}, \mathrm{t}_\mathcal{Y})$ be two coalgebras. For all states $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, if $x \sim y$ then $[\![x]\!]_\mathcal{X} = [\![y]\!]_\mathcal{Y}$.*

This theorem gives us a proof principle: to prove that two streams represented by two states are equal, it is sufficient to exhibit a bisimulation that relates the states.

Example 6. Consider the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$ given in Example 3 together with the coalgebra $(\mathcal{Z}_6, \mathrm{h}_6, \mathrm{t}_6)$ with $\mathcal{Z}_6 = \{z, z'\}$ and satisfying the equations:

$$\mathrm{h}_6(z) = 0 \qquad \mathrm{t}_6(z) = z' \qquad \mathrm{h}_6(z') = 0 \qquad \mathrm{t}_6(z') = z$$

One may want to prove that zero and z represent the same stream. For this purpose assume the relation $\mathcal{S} = \{(\mathrm{zero}, z), (\mathrm{zero}, z')\}$. It is straightforward to demonstrate $\mathcal{S}$ is a bisimulation, i.e. $[\![\mathrm{zero}]\!] = [\![z]\!]$.

We finish the section about coalgebra and coinduction by some observations that may help readers non familiar with the concepts presented in this section.

The coinduction proof principle can be seen as a systematic way of strengthening the statement one is trying to prove. In the previous example instead of proving the identity $[\![\mathrm{zero}]\!]_{\mathcal{X}_3} = [\![z]\!]_{\mathcal{Z}_6}$, we extended the relation $\{(\mathrm{zero}, z)\}$ to a more general relation $\mathcal{S} = \{(\mathrm{zero}, z), (\mathrm{zero}, z')\}$ with $\mathrm{zero} \,\mathcal{S}\, z$.

It is also worth noting that a coalgebra can contain an arbitrary number of representatives for a given stream. For instance $[\![\mathrm{blink}_0]\!]_{\mathcal{X}_3}$ is represented by no state within $(\mathcal{Z}_6, \mathrm{h}_6, \mathrm{t}_6)$. Conversely, $[\![\mathrm{zero}]\!]_{\mathcal{X}_3}$ is represented twice in $(\mathcal{Z}_6, \mathrm{h}_6, \mathrm{t}_6)$. Indeed by Theorem 2, we have $[\![z]\!]_{\mathcal{Z}_6} = [\![\mathrm{zero}]\!]_{\mathcal{X}_3} = [\![z']\!]_{\mathcal{Z}_6}$.

3 Constraint Handling Rules

In this section, we recall briefly the syntax and the semantics of CHR. Frühwirth's book [7] can be referred to for a more general overview of the language.

3.1 Syntax

The formalization of CHR assumes a language of *(built-in) constraints* containing equality over some theory $\mathcal{C}$, and defines *(user-defined) atoms* using a different set of predicate symbols. In the following, $\mathcal{R}$ will denote an arbitrary set of identifiers. By a slight abuse of notation, we allow confusion of conjunctions and multiset unions, omit braces around multisets, and use the comma for multiset union. We use $\mathrm{fv}(\phi)$ to denote the set of free variables of a formula ϕ. The notation $\exists_{\text{-}\psi}\phi$ denotes the existential closure of ϕ with the exception of free variables of ψ.

A *(CHR) program* is a finite set of eponymous rules of the form:

$$(r \mathbin{@} \mathbb{K} \backslash \mathbb{H} \iff \mathbb{G} \mid \mathbb{B}; \mathbb{C})$$

where $\mathbb{K}$ (the *kept head*), $\mathbb{H}$ (the *removed head*), and $\mathbb{B}$ (the *user body*) are multisets of atoms, $\mathbb{G}$ (the *guard*) and $\mathbb{C}$ (the *built-in body*) are conjunctions of constraints and, $r \in \mathcal{R}$ (the *rule name*) is an identifier assumed unique in the program. Rules in which both heads are empty are prohibited. An empty guard $\top$ (resp. an empty kept head) can be omitted with the symbol $\mid$ (resp. with the symbol $\backslash$). The *local variables* of rule are the variables occurring in the guard and in the body but not in the head that is $\mathrm{lv}(r) = \mathrm{fv}(\mathbb{G}, \mathbb{B}, \mathbb{C}) \setminus \mathrm{fv}(\mathbb{K}, \mathbb{H})$. Rules are divided into two classes: *simplification rules*[3] if the removed head is non-empty and *propagation rules* otherwise. Propagation rules can be written using the alternative syntax:

$$(r \mathbin{@} \mathbb{K} \Longrightarrow \mathbb{G} \mid \mathbb{B}; \mathbb{C})$$

3.2 Operational semantics

In this section, we recall the equivalence-based operational semantics ω_e of Raiser et al. [18]. It is equivalent to the *very abstract* semantics ω_{va} of Frühwirth [6], which is the most general operational semantics of CHR. We prefer the former because it includes a rigorous notion of equivalence, which is an essential component of confluence analysis.

A *(CHR) state* is a tuple $\langle \mathbb{E}; \mathbb{C}; \bar{x} \rangle$, where $\mathbb{E}$ (the *user store*) is a multiset of atoms, $\mathbb{C}$ (the *built-in store*) is a conjunction of constraints, and $\bar{x}$ (the *global variables*) is a finite set of variables. Unsurprisingly, the *local variables* of a state are those variables of the state which are not global. When no confusion can occur, we will syntactically merge user and built-in stores. We may futhermore omit the global variables component when states have no local variables. In the following, we use Σ to denote the set of states. Following Raiser et al., we will always implicitly consider states modulo a structural equivalence. Formally, this *state equivalence* is the least equivalence relation $\equiv$ over states satisfying the following rules:

[3] Unlike standard presentations, our definition does not distinguish simplification rules form the so-called simpagation rules.

– $\langle \mathbb{E}; \mathbb{C}; \bar{x} \rangle \equiv \langle \mathbb{E}; \mathbb{D}; \bar{x} \rangle$ if $\mathcal{C} \vDash \exists_{\text{-}(\mathbb{E},\bar{x})}\mathbb{C} \leftrightarrow \exists_{\text{-}(\mathbb{E},\bar{x})}\mathbb{D}$
– $\langle \mathbb{E}; \bot; \bar{x} \rangle \equiv \langle \mathbb{F}; \bot; \bar{y} \rangle$
– $\langle \mathbb{E}, c; \mathbb{C}, c{=}d; \bar{x} \rangle \equiv \langle \mathbb{E}, d; \mathbb{C}, c{=}d; \bar{x} \rangle$
– $\langle \mathbb{E}; \mathbb{C}; \bar{x} \rangle \equiv \langle \mathbb{E}; \mathbb{C}; \{y\} \cup \bar{x} \rangle$ if $y \notin \mathrm{fv}(\mathbb{E}, \mathbb{C})$.

Once states are considered modulo equivalence, the operational semantics of CHR can be expressed by a single rule. Formally the operational semantics of a program $\mathcal{P}$ is given by the binary relation $\xrightarrow{\mathcal{P}} \mathcal{P}$ on states satisfying the rule:

$$\frac{(r \mathbin{@} \mathbb{K}\backslash\mathbb{H} \iff \mathbb{G}|\mathbb{B}; \mathbb{C}) \in \mathcal{P}\rho \quad \mathrm{lv}(r) \cap \mathrm{fv}(\mathbb{E}, \mathbb{D}, \bar{x}) = \emptyset}{\langle \mathbb{K}, \mathbb{H}, \mathbb{E}; \mathbb{G}, \mathbb{D}; \bar{x} \rangle \xrightarrow{\mathcal{P}} \langle \mathbb{K}, \mathbb{B}, \mathbb{E}; \mathbb{G}, \mathbb{C}, \mathbb{D}; \bar{x} \rangle}$$

where ρ is a renaming. If a program $\mathcal{P}$ contains a sole rule r, we may write $\xrightarrow{r}$ for $\xrightarrow{\{r\}}$. For any transition $\xrightarrow{P}$, the symbol $\xleftarrow{P}$ will denote its converse, $\xrightarrow{\mathcal{P}}{}^{\equiv}$ its reflexive closure, and $\xrightarrow{\mathcal{P}}{}^{*}$ its transitive-reflexive closure. We will use $\xrightarrow{\mathcal{P}} \cdot \xrightarrow{\mathcal{Q}}$ to denote the left-composition of all binary relations $\xrightarrow{\mathcal{P}}$ and $\xrightarrow{\mathcal{Q}}$.

We will say a program $\mathcal{P}$ is *terminating* if there is no infinite sequence of the form $e_0 \xrightarrow{\mathcal{P}} e_1 \xrightarrow{\mathcal{P}} e_2 \ldots$ Furthermore, we will say that $\mathcal{P}$ is *confluent* if for all states S, S_1, and S_2 satisfying $S \xrightarrow{\mathcal{P}}{}^{*} S_1$ and $S \xrightarrow{\mathcal{P}}{}^{*} S_2$, there exists a state S' such that $S_1 \xrightarrow{\mathcal{P}}{}^{*} S'$ and $S_2 \xrightarrow{\mathcal{P}}{}^{*} S'$.

3.3 Declarative semantics

Owing to its origins in the tradition of CLP, the CHR language features declarative semantics through direct interprestation in first-order logic. Formally, the *logical reading* of a rule of the form:

$$\mathbb{K}\backslash\mathbb{H} \iff \mathbb{G} \mid \mathbb{B}; \mathbb{C}$$

is the guarded equivalence:

$$\forall\big((\mathbb{K} \wedge \mathbb{G}) \rightarrow \big(\mathbb{H} \leftrightarrow \exists_{\text{-}(\mathbb{K},\mathbb{H})}(\mathbb{G} \wedge \mathbb{C} \wedge \mathbb{B})\big)\big)$$

The *logical reading* of a program $\mathcal{P}$ within a theory $\mathcal{C}$ is the conjunction of the logical readings of its rules with the constraint theory $\mathcal{C}$. It is denoted by $\mathcal{CP}$.

Operational semantics is sound and complete with respect to this declarative semantics [6, 3, 10]. Furthermore, we recently established that any confluent program $\mathcal{P}$ is correct and has a logical model expressible by a CLP program, called the CLP projection. The *(CLP) projection* of a CHR program $\mathcal{P}$ is a set $\pi(\mathcal{P})$ of CLP clauses defined as:

$$\pi(\mathcal{P}) = \{(a \leftarrow \mathbb{G}, \mathbb{C}, \mathbb{K}, \mathbb{B}) \mid (\mathbb{K}\backslash\mathbb{H} \iff \mathbb{G} \mid \mathbb{B}, \mathbb{C}) \in \mathcal{P} \text{ and } a \in \mathbb{H}\}$$

Theorem 3 ([13]). *Let $\mathcal{S}$ be an arbitrary model of the constraint theory $\mathcal{C}$. A confluent CHR program and its projection have the same least $\mathcal{S}$-model.*

It is worth noting that in the current state of knowledge Theorem 3 only holds when programs are considered with respect to the most general operational semantics for CHR, namely the very abstract semantics. In particular, the proofs of the theorem do not appear to be adaptable to more concrete semantics such as for instance Abdennadher's token-based semantics [1].

4 Diagrammatic Confluence for CHR

This section sums up some recent results about conlfuence of non-terminating CHR programs. More details can be found in [11].

4.1 Critical Peaks

In term rewriting systems, the basic techniques used to prove confluence consist of showing various confluence criteria on a finite set of special cases, called *critical pairs*. Critical pairs are generated by a superposition algorithm, in which one attempts to capture the most general way the left-hand sides of the two rules of the system may overlap. The notion of critical pairs has been successfully adapted to CHR by Abdennadher et al. [2]. Here, we introduce a slight extension of the notion.

Let us assume that r_1 and r_2 are CHR rules (form possible disctinct programs) renamed apart:

$$(r_1 \mathbin{@} \mathbb{K}_1 \backslash \mathbb{H}_1 \iff \mathbb{G}_1 \mid \mathbb{B}_1; \mathbb{C}_1) \in \mathcal{P}_1 \qquad (r_2 \mathbin{@} \mathbb{K}_2 \backslash \mathbb{H}_2 \iff \mathbb{G}_2 \mid \mathbb{B}_2; \mathbb{C}_2) \in \mathcal{P}_2$$

A *critical ancestor (state)* S_c for the rules r_1 and r_2 is a state of the form:

$$S_c = \langle \mathbb{H}_1^{\Delta}, \mathbb{H}_1^{\cap}, \mathbb{H}_2^{\Delta}; \mathbb{D}; \bar{x} \rangle$$

satisfying the following properties:

- $(\mathbb{K}_1, \mathbb{H}_1) = (\mathbb{H}_1^{\Delta}, \mathbb{H}_1^{\cap})$, $(\mathbb{K}_2, \mathbb{H}_2) = (\mathbb{H}_2^{\Delta}, \mathbb{H}_2^{\cap})$, $\mathbb{H}_1^{\cap} \neq \emptyset$, and $\mathbb{H}_2^{\cap} \neq \emptyset$;
- $\bar{x}_1 = \mathrm{fv}(\mathbb{K}_1, \mathbb{H}_1)$, $\bar{x_2} = \mathrm{fv}(\mathbb{K}_2, \mathbb{H}_2)$ and $\bar{x} = \bar{x}_1 \cup \bar{x}_2$;
- $\mathbb{D} = (\mathbb{H}_1^{\cap} = \mathbb{H}_2^{\cap}, \mathbb{G}_1, \mathbb{G}_2)$ and $\exists \mathbb{D}$ is $\mathcal{C}$-satisfiable;
- $\mathbb{H}_1^{\cap} \not\subseteq \mathbb{K}_1$ or $\mathbb{H}_2^{\cap} \not\subseteq \mathbb{K}_2$.

Then the following tuple is called a *critical peak* between r_1 and r_2 at S_c:

$$\langle \mathbb{K}_1, \mathbb{B}_1, \mathbb{H}_2^{\Delta}; \mathbb{D}, \mathbb{C}_1; \bar{x} \rangle \xleftarrow{r_1} \cdot \xrightarrow{r_2} \langle \mathbb{K}_2, \mathbb{B}_2, \mathbb{H}_1^{\Delta}; \mathbb{D}, \mathbb{C}_2; \bar{x} \rangle$$

4.2 Rule-decreasingness

In this section, we present the so-called *rule-decreasingness criterion*. This criterion derived from the decreasing diagrams technique [22] is a novel criterion on CHR critical pairs that generalizes both local confluence [3] and strong confluence [14] criteria.

Rule-decreasingness criterion assumes the set $\mathcal{R}$ of rule identifiers is defined as a disjoint union $\mathcal{R}_i \uplus \mathcal{R}_c$. For a given program $\mathcal{P}$, we denote by $\mathcal{P}^i$ (resp. $\mathcal{P}^c$) the set of rules form $\mathcal{P}$ built with $\mathcal{R}_i$ (resp. $\mathcal{R}_c$). We call $\mathcal{P}^i$ the *inductive* part of $\mathcal{P}$, because we will subsequently assume that $\mathcal{P}^i$ is terminating, while $\mathcal{P}^c$ will be called *coinductive*, as it will be typically non-terminating. A critical peak is *inductive* if it involves only inductive rules (i.e. a critical peak of $\mathcal{P}^i$), or

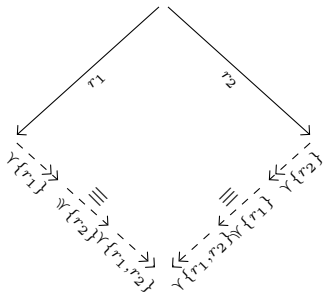

Fig. 3. Local decreasingness

coinductive if it involves at least one coinductive rule (i.e. a critical peak between $\mathcal{P}^c$ and $\mathcal{P}$).

In the rest of this paper, we will say that a preorder $\succcurlyeq$ is *wellfounded*, if the strict preorder $\succ$ associated with $\succcurlyeq$ (i.e. $\alpha \succ \beta$ iff $\alpha \succcurlyeq \beta$ but not $\beta \succcurlyeq \alpha$) is a terminating relation. A preorder $\succcurlyeq$ on rule identifiers is *admissible*, if any inductive rule identifier is strictly smaller than any coinductive one (i.e. for any $r_i \in \mathcal{R}_i$ and any $r_c \in \mathcal{R}_c$, $r_c \succ r_i$ holds).

A critical peak $S_1 \xleftarrow{r_1} \cdot \xrightarrow{r_2} S_2$ is decreasing with respect to a preorder $\succcurlyeq$ if the following property holds:

$$S_1 \xrightarrow{\Upsilon\{r_1\}}\!\!\!\!\!\rightarrow \cdot \xrightarrow{\curlyvee\{r_2\}}\!\!\!\!\!\rightarrow \equiv \cdot \xrightarrow{\Upsilon\{r_1, r_2\}}\!\!\!\!\!\rightarrow \cdot \xleftarrow{\Upsilon\{r_1, r_2\}}\!\!\!\!\!\leftarrow \cdot \xleftarrow{\curlyvee\{r_1\}}\!\!\!\!\!\leftarrow \equiv \cdot \xleftarrow{\Upsilon\{r_2\}}\!\!\!\!\!\leftarrow S_2 \qquad (\star)$$

where for any set K of rule identifiers, $\curlyvee K$ stands for $\{r \in \mathcal{R} \mid \exists r' \in K. r' \succcurlyeq r\}$ and ΥK for $\{r \in \mathcal{R} \mid \exists r' \in K. r' \succ r\}$. Property $(\star)$ is graphically represented in Figure 3. A program $\mathcal{P}$ is *rule-decreasing* with respect to an admissible preorder $\succcurlyeq$ if:

- the inductive part of $\mathcal{P}$ is terminating,
- any inductive critical peak of is joinable by $\xrightarrow{\mathcal{P}^i}\!\!\!\!\!\rightarrow \cdot \xleftarrow{\mathcal{P}^i}\!\!\!\!\!\leftarrow$, and
- any coinductive critical peaks is decreasing with respect to $\succcurlyeq$.

A program is *rule-decreasing* if it is rule-decreasing with respect to some admissible preorder.

Theorem 4 ([11]). *Rule-decreasing programs are confluent.*

To illustrate the use of the theorem, we recall now one example form [11].

Example 7 (Partial order constraint). Let $\mathcal{P}_7$ be the classic CHR introductory example, namely the constraint solver for partial order. This consists of the following four rules, which define the meaning of the *user-defined* symbol $\leq$

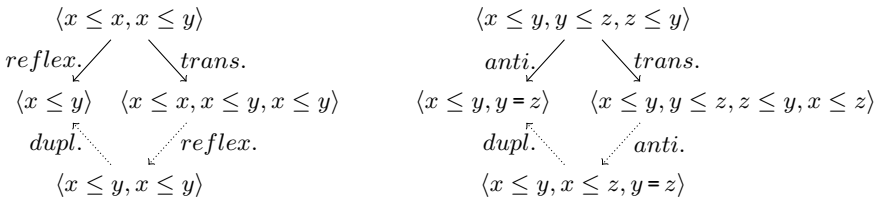

Fig. 4. Some rule-decreasing critical peaks for $\mathcal{P}_7$

using the built-in equality constraint $=$:

$$\begin{array}{lll} \textit{duplicate} & @ & x \leq y \setminus x \leq y \iff \top \\ \textit{reflexivity} & @ & x \leq x \iff \top \\ \textit{antisymmetry} & @ & x \leq y, y \leq x \iff x = y \\ \textit{transitivity} & @ & x \leq y, y \leq z \implies x \leq z \end{array}$$

Since $\mathcal{P}_7$ is trivially non-terminating (indeed, it uses propagation rules) one cannot apply local confluence criterion [3]. Nonetheless, confluence of $\mathcal{P}_7$ can be deduced using the full generality of Theorem 4. For this purpose, assume that all rules except *transitivity* are inductive and take any admissible preorder. Clearly the inductive part of $\mathcal{P}_7$ is terminating. Indeed the application of any one of the three first rules strictly reduces the number of atoms in a state. Then by a tedious but simple analysis, we prove that critical peaks of $\mathcal{P}_7$ can be joined while respecting the hypothesis of rule-decreasingness. In fact all critical peaks can be joined without using the *transitivity* rule. Some rule-decreasing diagrams involving the *transitivity* rule are given as examples in Figure 4.

5 Proving Stream Bisimulation Using CHR Confluence

In this section, we illustrate the power of CHR confluence and the rule-decreasingness criterion to prove coinductive properties.

5.1 Coalgebra in CHR

Following preliminary ideas for encoding coalgebras into CHR with the standard Herbrand constraint system [10], we use first-order terms as states, and define the destructors by means of a single user-defined atom $\mathrm{d}(s, h, t)$. Here the $\mathrm{d}(s, h, t)$ predicate must be understood as the function that returns for a given state s its head $h = \mathrm{h}(s)$ and its tail $t = \mathrm{t}(s)$. To enforce functionality of the destructor, we start our program $\mathcal{P}$ with the following simplification rule:

$$\mathrm{fun_d} \ @ \ \mathrm{d}(s, h_2, t_2) \setminus \mathrm{d}(s, h_1, t_1) \iff h_1 = h_2, t_1 = t_2$$

Now we use the terms zero, one, blink_0, and blink_1 as states for the respective streams containing only 0's, only 1's, alternations of 0 and 1, and alternations

of 1 and 0. Then we add to our program the following rules that specify the behaviour of the destructor on these states:

$$\begin{array}{lcll} \mathrm{d}_{\mathrm{zero}} & @ & \mathrm{d}(\mathrm{zero}, h, t) & \Longleftrightarrow\ h = 0, t = \mathrm{zero}. \\ \mathrm{d}_{\mathrm{one}} & @ & \mathrm{d}(\mathrm{one}, h, t) & \Longleftrightarrow\ h = 1, t = \mathrm{one}. \\ \mathrm{d}_{\mathrm{blink}_0} & @ & \mathrm{d}(\mathrm{blink}_0, h, t) & \Longleftrightarrow\ h = 0, t = \mathrm{blink}_1. \\ \mathrm{d}_{\mathrm{blink}_1} & @ & \mathrm{d}(\mathrm{blink}_1, h, t) & \Longleftrightarrow\ h = 1, t = \mathrm{blink}_0. \end{array}$$

We can go even further and define operators on streams. For this purpose, we use fresh function symbols as new operators and use recursive simplification rules to encode the behaviour of the destructors on these operators. For instance, we will encode the functions odd() and even(), which return the stream formed by the elements in the odd and even positions, respectively, and the function zip() which interlaces the elements from the two given streams. Hence, we add to $\mathcal{P}$ the rules:

$$\begin{array}{lcll} \mathrm{d}_{even} & @ & \mathrm{d}(\mathrm{even}(x), h, t) & \Longleftrightarrow\ \mathrm{d}(x, \_, t_1), \mathrm{d}(t_1, h, t_2), t = \mathrm{even}(t_2). \\ \mathrm{d}_{odd} & @ & \mathrm{d}(\mathrm{odd}(x), h, t) & \Longleftrightarrow\ \mathrm{d}(x, h, t_1), t = \mathrm{even}(t_1). \\ \mathrm{d}_{\mathrm{zip}} & @ & \mathrm{d}(\mathrm{zip}(x, y), h, t) & \Longleftrightarrow\ \mathrm{d}(x, h, t_1), t = \mathrm{zip}(y, t_1). \end{array}$$

It is not difficult to convince oneself that $\mathcal{P}$ is terminating. For this reason, we fix all the rules we have defined so far as inductive. Note that what is inductive here is solely the definition of the destructors. The encoded coalgebra still has a coinductive nature, in the sense that the destructor can be indefinitely called, in order to get all the elements composing a stream.

5.2 Proving simple Coinductive properties in CHR

The definition of bisimilation can translated into CHR by a single coinductive rule added to our program:

$$\sim\ @\ s_1 \sim s_2 \Longleftrightarrow \mathrm{d}(s_1, h, t_1), \mathrm{d}(s_2, h, t_2), t_1 \sim t_2$$

From a logical point of view, the declarative reading of this rule ensures that two states s_2 and s_1 are bisimilar if and only if there exists a model for the program $\mathcal{P}$ we have built so far, that contains the atom $s_1 \sim s_2$. Thanks to Theorem 3, we know that this is the case precisely if the program augmented with a "query" rule of the form $s_1 \sim s_2 \Longleftrightarrow \top$ is confluent. For instance, in order to prove that the stream $\mathrm{zip}(\mathrm{zero}, \mathrm{one})$ is equal to the stream blink_0, we can show that $\mathcal{P}$ together with the following rule is confluent:

$$\mathrm{q}_1\ @\ \mathrm{zip}(\mathrm{zero}, \mathrm{one}) \sim \mathrm{blink}_0 \Longleftrightarrow \top$$

Unfortunately this is not the case. Indeed, rules $\sim$ and q_1 yield a non-joinable peak:

$$\langle \mathrm{zip}(\mathrm{one}, \mathrm{zero}) \sim \mathrm{blink}_1 \rangle \xleftarrow{\mathcal{P}} \cdot \xleftarrow{\mathcal{P}}_{\sim} \langle \mathrm{zip}(\mathrm{zero}, \mathrm{one}) \sim \mathrm{blink}_0 \rangle \xrightarrow{\mathcal{P}}_{q_1} \langle \top \rangle$$

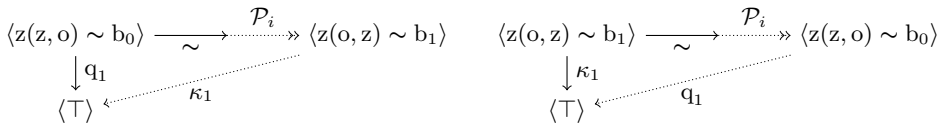

Fig. 5. Diagrammatic proofs for $\mathrm{z}(\mathrm{z},\mathrm{o}) \sim \mathrm{b}_0$

One idea for circumventing this problem is to "complete" the program, i.e. to add rules to it, in order to make it confluent. Indeed, if an interpretation is a model for the completed program, it is obviously a model for the original program. In the case of our example, we can just add a rule closing the previous peak:

$$\kappa_1 \mathbin{@} \mathrm{zip}(\mathrm{one}, \mathrm{zero}) \sim \mathrm{blink}_1 \iff \top$$

This time, the resulting program is confluent. We can use rule-decreasing criteria to prove this. For the proof passing through, we have to assume the rules q_1 and κ_1 are coinductive and strictly greater than the rule $\sim$. The proofs of the decreasingness of all the critical peaks involving at least one coinductive rule are graphically represented in Figure 5. For the sake of conciseness, symbols have been shortened to their initials in the figure.

5.3 Proving universal coinductive properties in CHR

In the previous section, we have proved a coinductive property for a particular stream. Here we are concerned with proving similar properties for arbitrary streams. The idea is to take benefit of the implicit universal quantification of rule-head variables to prove coinductive properties true with respect to arbitrary streams. However, we first have to formally define in our framework what a stream is.

As with bisimulation, being a stream is a coinductive property. It can be translated into CHR by the coinductive rule:

$$\mathbf{str} \mathbin{@} \mathbf{str}(x) \iff \mathrm{d}(x, \_, t), \mathbf{str}(t).$$

Basically, x is a stream if it can be deconstructed by d into a head and into a tail which is itself a stream. We now add to our program the following coinductive rules to specify which terms are actual streams:

$$\begin{array}{lcll} \mathbf{str}_{\mathrm{zero}} & @ & \mathbf{str}(\mathrm{zero}) & \iff \top \\ \mathbf{str}_{\mathrm{even}} & @ & \mathbf{str}(\mathrm{even}(x)) & \iff \mathbf{str}(x). \\ \mathbf{str}_{\mathrm{zip}} & @ & \mathbf{str}(\mathrm{zip}(x,y)) & \iff \mathbf{str}(x), \mathbf{str}(y) \end{array}$$

These rules state respectively that zero is a stream, $\mathrm{even}(x)$ is a stream if and only if x is a stream, and $\mathrm{zip}(y,z)$ is a stream if and only if both y and z are streams. From a typing point of view, rule **str** can be viewed as the definition of a (coinductive) type, and the rules $\mathbf{str}_x$ as type declarations. We do not

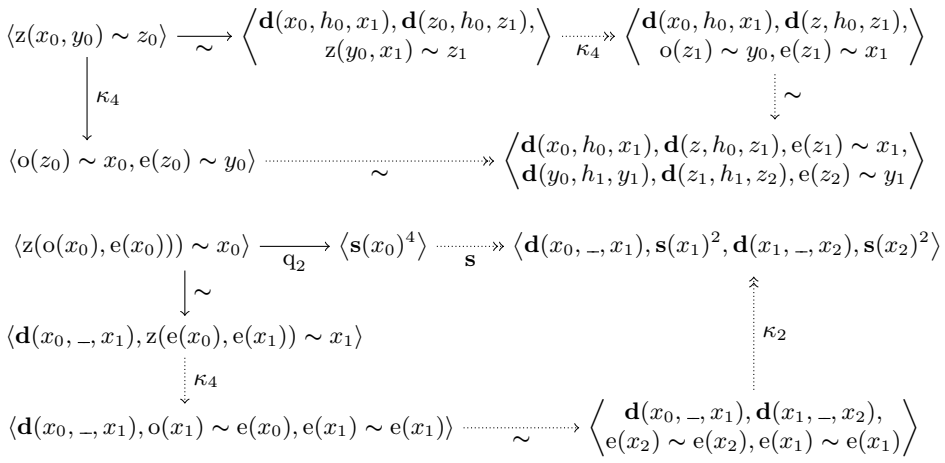

Fig. 6. Diagrammatic proofs for $\mathrm{zip}(\mathrm{odd}(x), \mathrm{even}(x)) \sim x$

explicitly present similar declarations for the streams one, blink_0, and blink_1, or for the operator odd. As we have done previously with bisimulation, we verify the coherency of these declarations by checking the confluence of the whole program. (We assume rules $\mathbf{str}_x$ are strictly greater than rule $\mathbf{str}$.)

Once streams have been properly defined and declared in our framework, we can try to prove that any stream x is bisimilar to $(\mathrm{zip}(\mathrm{odd}(x), \mathrm{even}(x)))$. For this purpose, we add the following coinductive rules to the current program:

$$\begin{array}{lll} q_2 \ @ \ \mathrm{zip}(\mathrm{odd}(x), \mathrm{even}(x)) \sim x & \Longleftrightarrow & \mathbf{str}(x)^4 \\ \kappa_2 \ @ \ x \sim x & \Longleftrightarrow & \mathbf{str}(x)^2 \\ \kappa_3 \ @ \ \mathrm{even}(\mathrm{zip}(x, y)) \sim y & \Longleftrightarrow & \mathbf{str}(x), \mathbf{str}(y) \\ \kappa_4 \ @ \ \mathrm{zip}(x, y) \sim z & \Longleftrightarrow & \mathrm{odd}(z) \sim x, \mathrm{even}(z) \sim y \end{array}$$

The first rule corresponds to the property we want to prove, while the other rules are there to complete the program, which would otherwise be non-confluent.

Once again, we are able to establish the confluence of the resulting program by using the rule-decreasingness criterion (assuming $q_2 \succ \kappa_4 \succ \kappa_3 \succ \kappa_2 \succ \mathbf{str}_x$). Note that, in order to simplify the proof, some atoms are duplicated. The exponents in the rules q_2 and κ_2 indicate how many times the atom $\mathbf{str}(x)$ is repeated. In practice, these repetitions are helpful because of the multiset nature of the user store. They are effective in closing a critical peak between κ_2 and κ_4 on the one hand, and a critical peak between q_2 and $\sim$ on the other hand. From a theoretical point of view, the repetition of atoms is not problematic, since an atom has a declarative meaning equivalent to the declarative meaning of several copies of it. Proofs of decreasingness for some relevant critical peaks of the program are graphically represented in Figure 6. Within the figure, states are implicitly normalized using the inductive part of the program.

When the program is proved confluent, the CLP projection provides us a logical model in the form of a CLP program. (In the practical case of our example, the projection is obtained by replacing the symbol $\Longleftrightarrow$ by $\leftarrow$ in all rules expcet for fun_d which can be safely ignored.) This finite representation is convenient since the model is in fact infinite: the domain of discourse of the model contains any stream obtained by composition of zero, one, blink_0, blink_1, even(), odd(), and zip().

From a model-theory point of view, we are not aware of any technique that is able to directly prove satisfiable formulas such as the ones presented in the last part of this section. In particular, the classical techniques, such as SMT solvers, inference-based theorem provers, or the analytic tableaux all seem inadequate for dealing with non-valid formulas mixing universal and existential quantifications, and which also have only infinite models. From a purely coinductive point of view, there do exist frameworks, such as circular coinductive rewriting [8], which are able to fully automatically prove bisimulations similar to the ones presented here. We argue nonetheless that our framework is more general, as coinductive properties are not hard-coded, but user-defined. Furthermore our work also has the advantage of shedding new light on the relationship between coinduction, confluence, and first-order satisfiability.

6 Conclusion

In this paper, we continued the study of relationships between CHR and coinduction started in [10]. Relying on universal coalgebra theory, we present a simple encoding of coalgebras and coinduction properties in CHR. Then, using the rule-decreasingness criterion we recently introduced, we realized effective coinductive proofs of bisimulations over streams. All the diagrammatic proofs sketched in the paper have been systematically verified by a prototype of a diagrammatic confluence checker [12] written in Ciao Prolog [15].

References

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the Int'l Conference on Principles and Practice of Constraint Programming (CP)*, volume 1330 of *LNCS*, pages 252–266, Berlin, Germany, 1997. Springer.

[2] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of Constraint Handling Rules. In *Proceedings of the Int'l Conference on Principles and Practice of Constraint Programming (CP)*, volume 1118 of *LNCS1*. Springer, 1996.

[3] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of Constraint Simplification Rules. *Constraints*, 4(2):133–165, 1999.

[4] P. Aczel. *Non-well-founded sets.* CSLI Publications, 1988.

[5] Jon Barwise and Larry Moss. *Vicious circles*. CSLI Publications, 1996.
[6] Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
[7] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
[8] Joseph A. Goguen, Kai Lin, and Grigore Rosu. Circular coinductive rewriting. In *Automated Software Engineering*, pages 123–132, 2000.
[9] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *In Proceedings of Glasgow Function Programming Workshop*, pages 78–95. Springer, 1994.
[10] R. Haemmerlé. (Co)-Inductive semantics for Constraint Handling Rules. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue*, 11(4–5):593–609, July 2011.
[11] R. Haemmerlé. Diagrammatic confluence for Cosntraint Handling Rules. *Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue*, 2012. To appear.
[12] R. Haemmerlé. `metaCHR` package, 2012. available online at `http://clip.dia.fi.upm.es/~remy/metaCHR/`.
[13] R. Haemmerlé, P. López, and M. Hermenegildo. CLP projection for Constraint Handling Rules. In *Proceedings of the 13th Int'l ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 137–148. ACM Press, July 2011.
[14] Rémy Haemmerlé and François Fages. Abstract critical pairs and confluence of arbitrary binary relations. In *Proceedings of the Int'l Conference on Rewriting Techniques and Applications (RTA)*, number 4533 in LNCS, pages 214–228, Berlin, Germany, 2007. Springer.
[15] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, 2012.
[16] Ekaterina Komendantskaya and John Power. Coalgebraic semantics for derivations in logic programming. *In Proceedings of the Int'l Conference on Algebra and Coalgebra in Computer Science (CALCO)*, volume 6859 of *LNCS*, pages 268–282. Springer, 2011.
[17] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[18] Frank Raiser, Hariolf Betz, and Thom Frühwirth. Equivalence of CHR states revisited. In *Proceedings of Int'l Workshop on Constraint Handling Rules*, Report CW 555, pages 34–48. Kath. Univ. Leuven, 2009.
[19] Jan J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.
[20] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
[21] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *ICLP*, volume 4079 of *LNCS*, pages 330–345, 2006.
[22] Vincent van Oostrom. Confluence by decreasing diagrams. *Theor. Comput. Sci.*, 126(2):259–280, 1994.