arXiv:1303.1994v2 [cs.LO] 11 Mar 2013

Automatic Equivalence Proofs for Non-deterministic Coalgebras

Marcello Bonsangue[a,d], Georgiana Caltais[b], Eugen-Ioan Goriac[b], Dorel Lucanu[c], Jan Rutten[d,e], Alexandra Silva[e,d,f]

[a]*LIACS - Leiden University, The Netherlands*
[b]*School of Computer Science - Reykjavik University, Iceland*
[c]*Faculty of Computer Science - Alexandru Ioan Cuza University, Romania*
[d]*Centrum Wiskunde & Informatica, The Netherlands*
[e]*Radboud University Nijmegen, The Netherlands*
[f]*HASLab / INESC TEC, Universidade do Minho, Braga, Portugal*

Abstract

A notion of generalized regular expressions for a large class of systems modeled as coalgebras, and an analogue of Kleene's theorem and Kleene algebra, were recently proposed by a subset of the authors of this paper. Examples of the systems covered include infinite streams, deterministic automata, Mealy machines and labelled transition systems. In this paper, we present a novel algorithm to decide whether two expressions are bisimilar or not. The procedure is implemented in the automatic theorem prover CIRC, by reducing coinduction to an entailment relation between an algebraic specification and an appropriate set of equations. We illustrate the generality of the tool with three examples: infinite streams of real numbers, Mealy machines and labelled transition systems.

1. Introduction

Regular expressions and finite deterministic automata (DFA's) constitute two of the most basic structures in computer science. Kleene's theorem [10] gives a fundamental correspondence between these two structures: each regular expression denotes a language that can be recognized by a DFA and, conversely, the language accepted by a DFA can be specified by a regular expression. Languages denoted by regular expressions are called regular. Two regular expressions are (language) equivalent if they denote the same regular language. Salomaa [21] presented a sound and complete axiomatization (later refined by Kozen in [11, 12]) for proving the equivalence of regular expressions.

The above programme was applied by Milner in [15] to process behaviours and labelled transition systems (LTS's). Milner introduced a set of expressions

Email addresses: `marcello@liacs.nl` (Marcello Bonsangue), `gcaltais10@ru.is` (Georgiana Caltais), `egoriac10@ru.is` (Eugen-Ioan Goriac), `dlucanu@info.uaic.ro` (Dorel Lucanu), `janr@cwi.nl` (Jan Rutten), `alexandra@cs.ru.nl` (Alexandra Silva)

Preprint submitted to Science of Computer Programming *February 6, 2014*

for finite LTS's and proved an analogue of Kleene's Theorem: each expression denotes the behaviour of a finite LTS and, conversely, the behaviour of a finite LTS can be specified by an expression (modulo bisimilarity). Milner also provided an axiomatization for his expressions, with the property that two expressions are provably equivalent if and only if they are bisimilar.

Coalgebras arose in the last decade as a suitable mathematical framework to study state-based systems, such as DFA's and LTS's. For a functor $\mathcal{G}\colon \mathbf{Set} \to \mathbf{Set}$, a $\mathcal{G}$-coalgebra or $\mathcal{G}$-system is a pair (S, g), consisting of a set S of states and a function $g\colon S \to \mathcal{G}(S)$ defining the "transitions" of the states. We call the functor $\mathcal{G}$ the type of the system. For instance, DFA's can be readily seen to correspond to coalgebras of the functor $\mathcal{G}(S) = 2 \times S^A$ and image-finite LTS's are obtained by $\mathcal{G}(S) = \mathcal{P}_\omega(S)^A$, where $\mathcal{P}_\omega$ is finite powerset.

For coalgebras of a large class of functors, a language of regular expressions, a corresponding generalization of Kleene's theorem, and a sound and complete axiomatization for the associated notion of behavioral equivalence were introduced in [23]. Both the language of expressions and their axiomatization were derived, in a modular fashion, from the functor defining the type of the system.

Algebra and related tools can be successfully used for reasoning on properties of systems. In this paper, we present a novel method for checking for bisimilarity of generalized regular expressions using the coinductive theorem prover CIRC [5, 17]. The main novelty of the method lies on the generality of the systems it can handle. CIRC is a metalanguage application implemented in Maude [4], and its target is to prove properties over infinite data structures. It has been successfully used for checking the equivalence of programs, and trace equivalence and strong bisimilarity of processes. The tool may be tested online and downloaded from `https://fmse.info.uaic.ro/tools/Circ/`.

Determining whether two expressions are equivalent is important in order to be able to compare behavioral specifications. In the presence of a sound and complete axiomatization one can determine equivalence using algebraic reasoning. A coalgebraic perspective on regular expressions has however provided a more operational/algorithmic way of checking equivalence: one constructs a bisimulation relation containing both expressions. The advantage of the bisimulation approach is that it enables automation since the steps of the construction are fairly mechanic and require almost no ingenuity.

We remark that in theory it has been shown that both problems are in PSPACE [13, 25], but in practice bisimulation checking tends to be easier. We illustrate this with an example, to give the reader the feeling of the more algorithmic nature of bisimulation. We want to stress however that we are not underestimating the value of an algebraic treatment of regular expressions: on the contrary, as we will show later, the axiomatization plays an important role in guaranteeing termination of the bisimulation construction and is therefore crucial for the main result of this article.

We show below a proof of the sliding rule: $a(ba)^* \equiv (ab)^*a$. The algebraic proof, using the rules and equations of Kleene algebra, needs to show the two

containments

$$a(ba)^* \leq (ab)^*a \qquad \text{and} \qquad (ab)^*a \leq a(ba)^*$$

and it requires some ingenuity in the choice of the equation applied in each step. We show the proof for the first inequality, the other would follow a similar proof pattern.

$$\begin{array}{lll} & a(ba)^* \leq (ab)^*a & \\ \Leftarrow & a + (ab)^*a(ba) \leq (ab)^*a & \text{right-star rule} \\ \Longleftrightarrow & (1 + (ab)^*ab)a \leq (ab)^*a & \text{associativity and distributivity} \\ \Longleftrightarrow & (ab)^*a \leq (ab)^*a & \text{right expansion rule: } 1 + r^*r = r^* \end{array}$$

For the coalgebraic proof, we build incrementally, and rather mechanically, a bisimulation relation containing the pair $(a(ba)^*, (ab)^*a)$. We start with the pair we want to prove equivalent and then we close the relation with respect to syntactic language derivatives, also known as *Brzozowski derivatives*. In the current example, the bisimulation relation would contain three pairs:

$$R = \{(a(ba)^*, (ab)^*a), ((ba)^*, b(ab)^*a + 1), (0, 0)\}$$

where 1 and 0 are, respectively, the regular expressions denoting the empty word and the empty language. In constructing this relation, no decisions were made, and hence the suitability of bisimulation construction as an automatic technique to prove equivalence of regular expressions.

The main contributions of this paper can be summarized as follows. We present a decision procedure to determine equivalence of generalized regular expressions, which specify behaviours of many types of transition systems, including Mealy machines, labelled transition systems and infinite streams. The valid expressions for each system are type-checked automatically in the tool. We illustrate the decision procedure we devised by applying it to several examples. As a vehicle of implementation, we choose CIRC, a coinductive theorem prover which has already been explored for the construction of bisimulations. To ease the implementation in CIRC, we present the algebraic specifications' counterpart of the coalgebraic framework of the generalized regular expressions mentioned above. This enables us to automatically derive algebraic specifications that model the language of expressions, and to define an appropriate equational entailment relation which mimics our decision procedure for checking behavioural equivalence of expressions. The implementation of both the algebraic specification and the entailment relation in CIRC allows for automatic reasoning on the equivalence of expressions.

The present paper is an extended version of the conference paper [2]. In comparison with the aforementioned paper we have extended the tool to deal with non-deterministic systems. More precisely, we have included the powerset function in the class of functors considered. Moreover, we have included all the proofs, more examples and additional explanations on the theory behind and implementation of the tool.

Organization of the paper. Section 2 recalls the basic definitions of the language associated to a non-deterministic functor. Section 3 describes the decision procedure to check equivalence of regular expressions. Section 4 formulates the aforementioned language as an algebraic specification, which paves the way to implement in CIRC the procedure to decide equivalence of expressions. The implementation of the decision procedure and its soundness are described in Section 5. In Section 6 we show, by means of several examples, how one can check bisimilarity, using CIRC. Section 7 contains concluding remarks and pointers for future work.

2. Regular Expressions for Non-deterministic Coalgebras

In this section, we briefly recall the basic definitions in [23].

Let **Set** denote the category of sets (represented by capital letters $X, Y, \ldots$) and functions (represented by lower case letters $f, g, \ldots$). We write Y^X for the family of functions from X to Y and $\mathcal{P}_\omega(X)$ for the collection of finite subsets of a set X. The product of two sets X, Y is written as $X \times Y$ and has the projections functions π_1 and π_2: $X \xleftarrow{\pi_1} X \times Y \xrightarrow{\pi_2} Y$. We define $X \oplus Y = X \uplus Y \uplus \{\bot, \top\}$ where $\uplus$ is the disjoint union of sets, with injections $X \xrightarrow{\kappa_1} X \uplus Y \xleftarrow{\kappa_2} Y$. Note that the set $X \oplus Y$ is different from the classical coproduct of X and Y (which we shall denote by $X + Y$), because of the two extra elements $\bot$ and $\top$. These extra elements are used to represent, respectively, underspecification and inconsistency in the specification of some systems.

For each of the operations defined above on sets, there are analogous ones on functions. Let $f\colon X \to Y$, $f_1\colon X \to Y$ and $f_2\colon Z \to W$. We define the following operations:

$$
\begin{array}{ll}
f_1 \times f_2\colon X \times Z \to Y \times W & f_1 \oplus f_2\colon X \oplus Z \to Y \oplus W \\
(f_1 \times f_2)(x, z) = \langle f_1(x), f_2(z)\rangle & (f_1 \oplus f_2)(c) = c,\ c \in \{\bot, \top\} \\
 & (f_1 \oplus f_2)(\kappa_i(x)) = \kappa_i(f_i(x)),\ i \in \overline{1,2} \\
f^A\colon X^A \to Y^A & \mathcal{P}_\omega(f)\colon \mathcal{P}_\omega(X) \to \mathcal{P}_\omega(Y) \\
f^A(g) = f \circ g & \mathcal{P}_\omega(f)(X_1) = \{y \in Y \mid f(x) = y, x \in X_1\}
\end{array}
$$

Remark 1. *For the sake of brevity, we use the notation $i \in \overline{1,n}$ as a shorthand for $i \in \{1, \ldots, n\}$.*

Note that in the definition above we are using the same symbols that we defined above for the operations on sets. It will always be clear from the context which operation is being used.

In our definition of non-deterministic functors we will use constant sets equipped with an information order. In particular, we will use join-semilattices. A (bounded) join-semilattice is a set B equipped with a binary operation $\vee_\mathsf{B}$ and a constant $\bot_\mathsf{B} \in \mathsf{B}$, such that $\vee_\mathsf{B}$ is commutative, associative and idempotent. The element $\bot_\mathsf{B}$ is neutral with respect to $\vee_\mathsf{B}$. As usual, $\vee_\mathsf{B}$ gives rise to a partial ordering $\leq_\mathsf{B}$ on the elements of B: $b_1 \leq_\mathsf{B} b_2 \Leftrightarrow b_1 \vee_\mathsf{B} b_2 = b_2$. Every set S can be mapped into a join-semilattice by taking B to be the set of all finite subsets of S with empty set as $\bot_\mathsf{B}$, and union as join.

Coalgebras. A coalgebra is a pair $(S, g\colon S \to \mathcal{G}(S))$, where S is a set of states and $\mathcal{G}\colon \mathbf{Set} \to \mathbf{Set}$ is a functor. The functor $\mathcal{G}$, together with the function g, determines the *transition structure* (or dynamics) of the $\mathcal{G}$-coalgebra [20].

A coalgebra (S, g) is *finite* if S is a finite set.

Definition 1 (Bisimulation). Let (S, f) and (T, g) be two $\mathcal{G}$-coalgebras. We call a relation $R \subseteq S \times T$ a *bisimulation* [9] iff

$$(s,t) \in R \Rightarrow (f(s), g(t)) \in \overline{\mathcal{G}}(R)$$

where $\overline{\mathcal{G}}(R)$ is defined as $\overline{\mathcal{G}}(R) = \{(\mathcal{G}(\pi_1)(x), \mathcal{G}(\pi_2)(x)) \mid x \in \mathcal{G}(R)\}$.

We write $s \sim_{\mathcal{G}} t$ whenever there exists a bisimulation relation containing (s, t) and we call $\sim_{\mathcal{G}}$ the bisimilarity relation. It is of interest to remark that the relation $\sim_{\mathcal{G}}$ is an equivalence relation. We shall drop the subscript $\mathcal{G}$ whenever the functor $\mathcal{G}$ is clear from the context. In the literature, one finds different definitions of bisimulation or behavioral equivalence [24]. For the class of functors we consider here the different notions coincide and therefore we will not discuss them.

Non-deterministic functors. They are functors $\mathcal{G}\colon \mathbf{Set} \to \mathbf{Set}$ built inductively from the identity, and constants, using $\times$, $\oplus$, $(-)^A$ and $\mathcal{P}_\omega$:

$$NDF \ni \mathcal{G} ::= \mathsf{Id} \mid \mathsf{B} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \times \mathcal{G} \mid \mathcal{G}^A \mid \mathcal{P}_\omega \mathcal{G} \tag{1}$$

where B is a finite join-semilattice and A is a finite set. Typical examples of non-deterministic functors include $\mathcal{S} = \mathsf{B} \times \mathsf{Id}$, $\mathcal{M} = (\mathsf{B} \times \mathsf{Id})^A$, $\mathcal{D} = 2 \times \mathsf{Id}^A$, $\mathcal{Q} = (1 \oplus \mathsf{Id})^A$, $\mathcal{N} = 2 \times \mathcal{P}_\omega(\mathsf{Id})^A$ and $\mathcal{L} = 1 \oplus \mathcal{P}_\omega(\mathsf{Id})^A$. These functors represent, respectively, the type of streams, Mealy, deterministic, partial deterministic automata, non-deterministic automata and labeled transition systems with explicit termination. $\mathcal{S}$-bisimulation is stream equality, whereas $\mathcal{D}$-bisimulation coincides with language equivalence.

Remark 2. *As stated in [23], the use of join-semilattices for constant functors and the sum $\oplus$ instead of the ordinary product enabled the use of underspecification and inconsistency (i.e., $\top$ and $\bot$, respectively) in the specification of systems, and moreover, has allowed the whole framework to be studied in the category* **Set**. *Even though underspecification and inconsistency can be captured by a semilattice structure, and the axiomatization provides the set of expressions with a join-semilattice structure (therefore allowing the work directly in the category of join-semilattices), remaining in the category* **Set** *was chosen for simplicity.*

Next, we give the definition of the ingredient relation, which relates a non-deterministic functor $\mathcal{G}$ with its *ingredients*, *i.e.*, the functors used in its inductive construction. We shall use this relation later for typing our expressions.

Definition 2. Let $\triangleleft \subseteq NDF \times NDF$ be the least reflexive and transitive relation on non-deterministic functors such that

$$\mathcal{G}_1 \triangleleft \mathcal{G}_1 \times \mathcal{G}_2, \quad \mathcal{G}_2 \triangleleft \mathcal{G}_1 \times \mathcal{G}_2, \quad \mathcal{G}_1 \triangleleft \mathcal{G}_1 \oplus \mathcal{G}_2, \quad \mathcal{G}_2 \triangleleft \mathcal{G}_1 \oplus \mathcal{G}_2, \quad \mathcal{G} \triangleleft \mathcal{G}^A, \quad \mathcal{G} \triangleleft \mathcal{P}_\omega \mathcal{G}.$$

Here and throughout this document we use $\mathcal{F} \triangleleft \mathcal{G}$ as a shorthand for $(\mathcal{F}, \mathcal{G}) \in \triangleleft$. If $\mathcal{F} \triangleleft \mathcal{G}$, then $\mathcal{F}$ is said to be an *ingredient* of $\mathcal{G}$. For example, 2, Id, Id^A and $\mathcal{D}$ itself are all the ingredients of the deterministic automata functor $\mathcal{D}$.

A language of regular expressions for non-deterministic coalgebras. We now associate a language of expressions $\mathsf{Exp}_\mathcal{G}$ with each non-deterministic functor $\mathcal{G}$.

Definition 3 (Expressions). Let A be a finite set, B a finite join-semilattice and X a set of fixed-point variables. The set Exp of all *expressions* is given by the following grammar, where $a \in A$, $b \in \mathsf{B}$ and $x \in X$:

$$\varepsilon \quad ::= \quad x \mid \varepsilon \oplus \varepsilon \mid \gamma \tag{2}$$

where γ is a *guarded expression* given by:

$$\gamma \quad ::= \quad \underline{\emptyset} \mid \gamma \oplus \gamma \mid \mu x.\gamma \mid b \mid l\langle \varepsilon \rangle \mid r\langle \varepsilon \rangle \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon) \mid \{\varepsilon\} \tag{3}$$

In the expression $\mu x.\gamma$, μ is a binder for all the free occurrences of x in γ. Variables that are not bound are free. A *closed expression* is an expression without free occurrences of fixed-point variables x. We denote the set of closed expressions by Exp^c.

The language of expressions for non-deterministic coalgebras is a generalization of the classical notion of regular expressions: $\underline{\emptyset}$, $\varepsilon_1 \oplus \varepsilon_2$ and $\mu x.\gamma$ play similar roles to the regular expressions denoting empty language, the union of languages and the Kleene star. Moreover, note that, not unexpectedly, in [23], $\oplus$ was axiomatized as an associative, commutative and idempotent operator, with $\underline{\emptyset}$ as a neutral element. The expressions $l\langle \varepsilon \rangle$, $r\langle \varepsilon \rangle$, $l[\varepsilon]$, $r[\varepsilon]$, $a(\varepsilon)$ and $\{\varepsilon\}$ specify the left and right hand-side of products and sums, function application and singleton sets, respectively. Next, we present a type assignment system for associating expressions to non-deterministic functors. This will allow us to associate with each functor $\mathcal{G}$ the expressions $\varepsilon \in \mathsf{Exp}^c$ that are valid specifications of $\mathcal{G}$-coalgebras.

Definition 4 (Type system). We now define a typing relation $\vdash \subseteq \mathsf{Exp} \times NDF \times NDF$ that will associate an expression ε with two non-deterministic functors $\mathcal{F}$ and $\mathcal{G}$, which are related by the ingredient relation ($\mathcal{F}$ is an ingredient of $\mathcal{G}$). We shall write $\vdash \varepsilon \colon \mathcal{F} \triangleleft \mathcal{G}$ for $(\varepsilon, \mathcal{F}, \mathcal{G}) \in \vdash$. The rules that define $\vdash$ are

the following:

$$
\frac{}{\vdash \underline{\emptyset} \colon \mathcal{F} \lhd \mathcal{G}} \qquad \frac{}{\vdash b \colon \mathsf{B} \lhd \mathcal{G}}\ (b \in \mathsf{B}) \qquad \frac{}{\vdash x \colon \mathcal{G} \lhd \mathcal{G}}\ (x \in X)
$$

$$
\frac{\vdash \varepsilon \colon \mathcal{G} \lhd \mathcal{G}}{\vdash \mu x.\varepsilon \colon \mathcal{G} \lhd \mathcal{G}} \qquad \frac{\vdash \varepsilon_1 \colon \mathcal{F} \lhd \mathcal{G} \quad \vdash \varepsilon_2 \colon \mathcal{F} \lhd \mathcal{G}}{\vdash \varepsilon_1 \oplus \varepsilon_2 \colon \mathcal{F} \lhd \mathcal{G}} \qquad \frac{\vdash \varepsilon \colon \mathcal{G} \lhd \mathcal{G}}{\vdash \varepsilon \colon \mathsf{Id} \lhd \mathcal{G}}
$$

$$
\frac{\vdash \varepsilon \colon \mathcal{F}_2 \lhd \mathcal{G}}{\vdash r[\varepsilon] \colon \mathcal{F}_1 \oplus \mathcal{F}_2 \lhd \mathcal{G}} \qquad \frac{\vdash \varepsilon \colon \mathcal{F} \lhd \mathcal{G}}{\vdash a(\varepsilon) \colon \mathcal{F}^A \lhd \mathcal{G}}\ (a \in A) \qquad \frac{\vdash \varepsilon \colon \mathcal{F}_1 \lhd \mathcal{G}}{\vdash l\langle\varepsilon\rangle \colon \mathcal{F}_1 \times \mathcal{F}_2 \lhd \mathcal{G}}
$$

$$
\frac{\vdash \varepsilon \colon \mathcal{F}_2 \lhd \mathcal{G}}{\vdash r\langle\varepsilon\rangle \colon \mathcal{F}_1 \times \mathcal{F}_2 \lhd \mathcal{G}} \qquad \frac{\vdash \varepsilon \colon \mathcal{F}_1 \lhd \mathcal{G}}{\vdash l[\varepsilon] \colon \mathcal{F}_1 \oplus \mathcal{F}_2 \lhd \mathcal{G}} \qquad \frac{\vdash \varepsilon \colon \mathcal{F}_1 \lhd \mathcal{G}}{\vdash \{\varepsilon\} \colon \mathcal{P}_\omega \mathcal{F}_1 \lhd \mathcal{G}}
$$

We can now formally define the set of $\mathcal{G}$-expressions: well-typed expressions associated with a non-deterministic functor $\mathcal{G}$.

Definition 5 ($\mathcal{G}$-expressions). Let $\mathcal{G}$ be a non-deterministic functor and $\mathcal{F}$ an ingredient of $\mathcal{G}$. We define $\mathsf{Exp}_{\mathcal{F} \lhd \mathcal{G}}$ by:

$$
\mathsf{Exp}_{\mathcal{F} \lhd \mathcal{G}} = \{\varepsilon \in \mathsf{Exp}^c \mid \vdash \varepsilon \colon \mathcal{F} \lhd \mathcal{G}\}\,.
$$

We define the set $\mathsf{Exp}_{\mathcal{G}}$ of well-typed $\mathcal{G}$-*expressions* by $\mathsf{Exp}_{\mathcal{G} \lhd \mathcal{G}}$.

In [23], it was proved that the set of $\mathcal{G}$-expressions for a given non-deterministic functor $\mathcal{G}$ has a coalgebraic structure:

$$
\delta_{\mathcal{G}} \colon \mathsf{Exp}_{\mathcal{G}} \to \mathcal{G}(\mathsf{Exp}_{\mathcal{G}})
$$

More precisely, in [23], which we refer to for the complete definition of $\delta_{\mathcal{G}}$, the authors defined a function $\delta_{\mathcal{F} \lhd \mathcal{G}} \colon \mathsf{Exp}_{\mathcal{F} \lhd \mathcal{G}} \to \mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$ and then set $\delta_{\mathcal{G}} = \delta_{\mathcal{G} \lhd \mathcal{G}}$.

The coalgebraic structure on the set of expressions enabled the proof of a Kleene like theorem.

Theorem 1 (Kleene's theorem for non-deterministic coalgebras). *Let $\mathcal{G}$ be a non-deterministic functor.*

1. *For any $\varepsilon \in \mathsf{Exp}_{\mathcal{G}}$, there exists a finite $\mathcal{G}$-coalgebra (S, g) and $s \in S$ such that $\varepsilon \sim s$.*
2. *For every finite $\mathcal{G}$-coalgebra (S, g) and $s \in S$ there exists an expression $\varepsilon_s \in \mathsf{Exp}_{\mathcal{G}}$ such that $\varepsilon_s \sim s$.*

In order to provide the reader with intuition over the notions presented above, we illustrate them with an example.

Example 1. *Let us instantiate the definition of $\mathcal{G}$-expressions to the functor of streams $\mathcal{S} = \mathsf{B} \times \mathsf{Id}$ (the ingredients of this functor are B, Id and $\mathcal{S}$ itself). Let X be a set of (recursion or) fixed-point variables. The set $\mathsf{Exp}_{\mathcal{S}}$ of* stream expressions *is given by the set of closed, guarded expressions generated by the following BNF grammar. For $x \in X$:*

$$
\begin{aligned}
\mathsf{Exp}_{\mathcal{S}} \ni \varepsilon &::= \underline{\emptyset} \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon \mid x \mid l\langle\tau\rangle \mid r\langle\varepsilon\rangle \\
\tau &::= \underline{\emptyset} \mid b \mid \tau \oplus \tau
\end{aligned} \tag{4}
$$

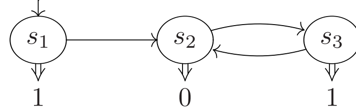
Intuitively, the expression $l\langle b\rangle$ is used to specify that the head of the stream is b, while $r\langle\varepsilon\rangle$ specifies a stream whose tail behaves as specified by ε. For the two element join-semilattice $\mathsf{B} = \{0,1\}$ (with $\bot_\mathsf{B} = 0$) examples of well-typed expressions include $\underline{\emptyset}$, $l\langle 1\rangle \oplus r\langle l\langle\underline{\emptyset}\rangle\rangle$ and $\mu x.r\langle x\rangle \oplus l\langle 1\rangle$. The expressions $l[1]$, $l\langle 1\rangle \oplus 1$ and $\mu x.1$ are examples of non well-typed expressions for $\mathcal{S}$, because the functor $\mathcal{S}$ does not involve $\oplus$, the subexpressions in the sum have different type, and recursion is not at the outermost level (1 has type $\mathsf{B} \lhd \mathcal{S}$), respectively.

By applying the definition in [23], the coalgebra structure on expressions $\delta_\mathcal{S}$ would be given by:

$$
\begin{array}{lcl}
\delta_\mathcal{S} \colon \mathsf{Exp}_\mathcal{S} \to \mathsf{B} \times \mathsf{Exp}_\mathcal{S} & & \\
\delta_\mathcal{S}(\underline{\emptyset}) & = & \langle \bot_\mathsf{B}, \underline{\emptyset}\rangle \\
\delta_\mathcal{S}(\varepsilon_1 \oplus \varepsilon_2) & = & \langle b_1 \vee b_2, \varepsilon_1' \oplus \varepsilon_2'\rangle \text{ where } \langle b_i, \varepsilon_i'\rangle = \delta_\mathcal{S}(\varepsilon_i),\ i \in \overline{1,2} \\
\delta_\mathcal{S}(\mu x.\varepsilon) & = & \delta_\mathcal{S}(\varepsilon[\mu x.\varepsilon/x]) \\
\delta_\mathcal{S}(l\langle\tau\rangle) & = & \langle \delta_{\mathsf{B}\lhd\mathcal{S}}(\tau), \underline{\emptyset}\rangle \\
\delta_\mathcal{S}(r\langle\varepsilon\rangle) & = & \langle \bot_\mathsf{B}, \varepsilon\rangle \\
\delta_{\mathsf{B}\lhd\mathcal{S}}(\underline{\emptyset}) & = & \bot_B \\
\delta_{\mathsf{B}\lhd\mathcal{S}}(b) & = & b \\
\delta_{\mathsf{B}\lhd\mathcal{S}}(\tau \oplus \tau') & = & \delta_{\mathsf{B}\lhd\mathcal{S}}(\tau) \vee \delta_{\mathsf{B}\lhd\mathcal{S}}(\tau')
\end{array}
$$

The proof of Kleene's theorem provides algorithms to go from expressions to streams and vice-versa. We illustrate it by means of examples.

Consider the following stream:



We draw the stream with an automata-like flavor. The transitions indicate the tail of the stream represented by a state and the output value the head. In a more traditional notation, the above automata represents the infinite stream $(1, 0, 1, 0, 1, 0, 1, \ldots)$.

To compute expressions ε_1, ε_2 and ε_3 equivalent to s_1, s_2 and s_3 we associate with each state s_i a variable x_i and get the equations:

$$\varepsilon_1 = \mu x_1.l\langle 1\rangle \oplus r\langle x_2\rangle \quad \varepsilon_2 = \mu x_2.l\langle 0\rangle \oplus r\langle x_3\rangle \quad \varepsilon_3 = \mu x_3.l\langle 1\rangle \oplus r\langle x_2\rangle$$

As our goal is to remove all the occurrences of free variables in our expressions, we proceed as follows. First we substitute x_2 by ε_2 in ε_1, and x_3 by ε_3 in ε_2, and obtain the following expressions:

$$\varepsilon_1 = \mu x_1.l\langle 1\rangle \oplus r\langle \varepsilon_2\rangle \quad \varepsilon_2 = \mu x_2.l\langle 0\rangle \oplus r\langle \varepsilon_3\rangle$$

Note that at this point ε_1 and ε_2 already denote closed expressions. Therefore, as a last step, we replace x_2 in ε_3 by ε_2 and get the following closed expressions:

$$\varepsilon_1 = \mu x_1.l\langle 1\rangle \oplus r\langle \varepsilon_2\rangle \quad \varepsilon_2 = \mu x_2.l\langle 0\rangle \oplus r\langle \varepsilon_3\rangle \quad \varepsilon_3 = \mu x_3.l\langle 1\rangle \oplus r\langle \mu x_2.l\langle 0\rangle \oplus r\langle x_3\rangle\rangle$$

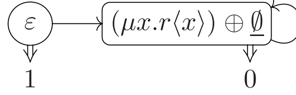
satisfying, by construction, $\varepsilon_1 \sim s_1$, $\varepsilon_2 \sim s_2$ and $\varepsilon_3 \sim s_3$.

For the converse construction, consider the expression $\varepsilon = (\mu x.r\langle x\rangle) \oplus l\langle 1\rangle$. We construct an automaton by repeatedly applying the coalgebra structure on expressions $\delta_{\mathcal{S}}$, modulo associativity, commutativity and idempotence (ACI) of $\oplus$ in order to guarantee finiteness.

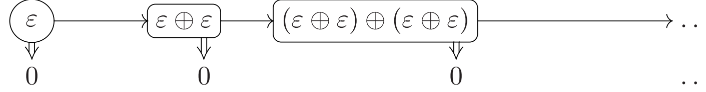
First, note that $\delta_{\mathcal{S}}(\mu x.r\langle x\rangle) = \delta_{\mathcal{S}}(r\langle \mu x.r\langle x\rangle\rangle) = \langle \perp_{\mathsf{B}}, \mu x.r\langle x\rangle\rangle$. Applying the definition of $\delta_{\mathcal{S}}$ above, we have:

$$\delta_{\mathcal{S}}(\varepsilon) = \langle 1, (\mu x.r\langle x\rangle) \oplus \underline{\emptyset}\rangle \text{ and } \delta_{\mathcal{S}}((\mu x.r\langle x\rangle) \oplus \underline{\emptyset}) = \langle 0, (\mu x.r\langle x\rangle) \oplus \underline{\emptyset}\rangle$$

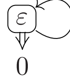
which leads to the following stream (automaton):



At this point, we want to remark that the direct application of $\delta_{\mathcal{S}}$, without ACI, might generate infinite automata. Take, for instance, the expression $\varepsilon = \mu x.r\langle x \oplus x\rangle$. Note that $\delta_{\mathcal{S}}(\mu x.r\langle x\oplus x\rangle) = \langle 0, \varepsilon\oplus\varepsilon\rangle$, $\delta_{\mathcal{S}}(\varepsilon\oplus\varepsilon) = \langle 0, (\varepsilon\oplus\varepsilon)\oplus(\varepsilon\oplus\varepsilon)\rangle$, and so on. This would generate the infinite automaton



instead of the intended, simple and very finite, automaton



In order to guarantee finiteness, one needs to identify the expressions modulo associativity, commutativity and idempotence (ACI), as we will discuss further in this paper. Moreover, the axiom $\varepsilon \oplus \underline{\emptyset} \equiv \underline{\emptyset}$ could also be used in order to obtain smaller automata, but it is not crucial for termination.

Throughout the paper, we will often use streams as a basic example to illustrate the definitions. It should be remarked that the framework is general enough to include more complex examples, such as deterministic automata, automata on guarded strings, Mealy machines and labelled transition systems. The latter two will be used as examples in Section 6.

3. A Decision Procedure for the Equivalence of Generalized Regular Expressions

In this section, we briefly describe the decision procedure to determine whether two expressions are equivalent or not.

The key observation is that point 1. of Theorem 1 above guarantees that each expression in the language for a given system can always be associated to

a *finite* coalgebra. Given two expressions ε_1 and ε_2 in the language $\mathsf{Exp}_{\mathcal{G}}$ of a given functor $\mathcal{G}$ we can decide whether they are equivalent by constructing a *finite* bisimulation between them. This is because the finite coalgebra generated from an expression contains precisely all states that one needs to construct the equivalence relation. Even though this might seem like a trivial observation, it has very concrete consequences: for (all well-typed) generalized regular expressions we can always either determine that they are bisimilar, and exhibit a proof in the form of a bisimulation, or conclude that they are not bisimilar and pinpoint the difference by showing why the bisimulation construction failed. Hence, we have a decision procedure for equivalence of generalized regular expressions.

We will give the reader a brief example on how the equivalence check works. Further examples, for different types of systems, including examples of non-equivalence, will appear in Section 6.

We will show that the stream expressions $\varepsilon_1 = \mu x.r\langle x\rangle \oplus l\langle 0\rangle$ and $\varepsilon_2 = r\langle \mu x.r\langle x\rangle \oplus l\langle 0\rangle\rangle \oplus l\langle 0\rangle$ are equivalent. In order to do that, we have to build a bisimulation relation $\mathcal{R}$ on expressions for the stream functor $\mathcal{S}$, defined above, such that $(\varepsilon_1, \varepsilon_2) \in \mathcal{R}$. We do this in the following way: we start by taking $\mathcal{R} = \{(\varepsilon_1, \varepsilon_2)\}$ and we check whether this is already a bisimulation, by applying $\delta_{\mathcal{S}}$ to each of the expressions and checking whether the expressions have the same output value and, moreover, that no new pairs of expressions (modulo associativity, commutativity and idempotence, for more details see page 25) appear when taking transitions. If new pairs of expressions appear we add them to $\mathcal{R}$ and repeat the process. Intuitively, for this particular example, the transition structure can be depicted as follows:

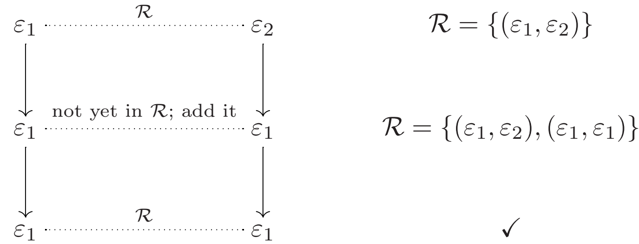


Figure 1: Bisimulation construction

Here, we omit the output values of the expressions, which are all 0. In the figure above, we use the notation $\varepsilon_1 \overset{\mathcal{R}}{\text{———}} \varepsilon_2$ to denote $(\varepsilon_1, \varepsilon_2) \in \mathcal{R}$. As illustrated in Figure 1, $\mathcal{R} = \{(\varepsilon_1, \varepsilon_2), (\varepsilon_2, \varepsilon_2)\}$ is closed under transitions and is therefore a bisimulation. Hence, ε_1 and ε_2 are bisimilar and specify the same infinite stream (concretely, the stream with only zeros).

4. An Algebraic View on the Coalgebra of Generalized Regular Expressions

Recall that our goal is to reason about equality of generalized regular expressions in a fully automated manner. As we showed in the introduction, obtaining this equality can be achieved in two distinct ways: either algebraically, reasoning with the axioms, or coalgebraically, by constructing a bisimulation relation. The latter, because of its algorithmic nature, is particularly suited for automation. Automatic constructions of bisimulations have been widely explored in CIRC and we will use this tool to implement our algorithm. This section contains material that enables us to soundly use CIRC. We want to stress however that the main result of the paper is the description of a *decision procedure* to determine whether two expressions are equivalent or not. This procedure in turn could be implemented in any other suitable tool or even as a standalone application. Choosing CIRC was natural for us, given the pre-existent work on bisimulation constructions. In Section 5, we show that the process of generating the $\mathcal{G}$-coalgebras associated to expressions by repeatedly applying $\delta_{\mathcal{G}}$ and normalizing the expressions obtained at each step is closely related to the proving mechanism already existent in CIRC.

In Section 2, we have introduced a (theoretical) framework which, given a functor $\mathcal{G}$, allows for the uniform derivation of 1) a language $\mathsf{Exp}_{\mathcal{G}}$ for specifying behaviors of $\mathcal{G}$-systems, and 2) a coalgebraic structure on $\mathsf{Exp}_{\mathcal{G}}$, which provides an operational semantics to the set of expressions. In this context, given that CIRC is based on algebraic specifications, we need two things in order to reach our final goal:

- extend and adapt the framework of Section 2 in order to enable the implementation of a tool which allows the automatic derivation of *algebraic specifications* that model 1) and 2) above, to deliver to CIRC;
- provide a decision procedure, implemented in CIRC based on an *equational entailment relation*, in order to check bisimilarity of expressions.

In the rest of the paper we will present the algebraic setting for reasoning on bisimilarity of generalized regular expressions. A brief overview on the parallel between the coalgebraic concepts in [23] and their algebraic correspondents introduced in this section is provided later, in Figure 2.

Algebraic specifications. An *algebraic specification* is a triple $\mathcal{E} = (S, \Sigma, E)$, where S is a set of *sorts*, Σ is a *S-sorted signature* and E is a set of *conditional equations* of the form $(\forall X)\, t = t' \texttt{ if } (\bigwedge_{i \in I} u_i = v_i)$, where t, t', u_i, and v_i ($i \in I$ – a set of indices for the conditions) are Σ-terms with variables in X. We say that the *sort of the equation* is s whenever $t, t' \in \mathcal{T}_{\Sigma,s}(X)$. Here, $\mathcal{T}_{\Sigma,s}(X)$ denotes the set of terms of sort s of the Σ-algebra freely generated by X. If $I = \{\}$ then the equation is *unconditional* and may be written as $(\forall X)\, t = t'$.

Let $\vdash$ be the *equational entailment (deduction) relation* defined as in [6]. For consistency reasons, we write $\mathcal{E} \vdash e$ whenever equation e is deducible from the

equations E in $\mathcal{E}$ by reflexivity, symmetry, transitivity, congruence or substitutivity (*i.e.*, whenever $E \vdash e$).

In this paper, the algebraic specifications of coalgebras of generalized regular expressions are built on top of definitions based on grammars in Backus-Naur form (BNF) such as (1) and (2). Therefore, in what follows, we introduce the general technique for transforming BNF notations into algebraic specifications.

From BNF grammars to algebraic specifications. The general rule used for translating definitions based on BNF grammars into algebraic specifications is as follows: each syntactical category and vocabulary is considered as a sort and each production is considered as a constructor operation or a subsort relation.

For instance, according to the grammar (1) of non-deterministic functors, we have a sort $\mathsf{SltName}$ – representing the vocabulary of join-semilattices B, a sort $\mathsf{AlphName}$ – for the vocabulary of the alphabets A, a sort $\mathsf{Functor}$ – associated to the syntactical category of the non-deterministic functors $\mathcal{G}$, a subsort relation $\mathsf{SltName} < \mathsf{Functor}$ representing the production $\mathcal{G} ::= \mathsf{B}$, and constructor operations for the other productions.

Generally, each production $A ::= \mathit{rhs}$ gives rise to a constructor $(\mathit{rhs}) \to (A)$, the direction of the arrow being reversed. For instance, for grammar (1), the production $\mathcal{G} ::= \mathsf{Id}$ is represented by a constant (nullary operation) $\mathsf{Id}\colon \to \mathsf{Functor}$, and the sum construction by the binary operation $_ \oplus _ : \mathsf{Functor}\ \mathsf{Functor} \to \mathsf{Functor}$.

Remark 3. *Note that the above mechanism for translating BNF grammars into algebraic specifications makes use of subsort relations for representing productions such as $\mathcal{G} ::= \mathsf{B}$. This is because CIRC works with order-sorted algebras, and we want to keep the algebraic specifications of non-deterministic functors as close as possible to their implementation in CIRC. However, order-sorted algebras can be reduced to many-sorted algebras [6], where a subsort relation $s < s'$ is modeled by an inclusion operation $c_{s,s'} : s \to s'$. This way, even if we use order-sorted algebras, we remain in the framework of circular coinduction.*

The algebraic specifications of coalgebras of generalized regular expressions are defined in a modular fashion, based on the specifications of:

- non-deterministic functors ($\mathcal{G}$);
- generalized regular expressions ($\varepsilon \in \mathsf{Exp}_{\mathcal{G}}$);
- "transition" functions ($\delta_{\mathcal{G}}$);
- "structured" expressions ($\sigma \in \mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$, for all $\mathcal{F}$ ingredients of $\mathcal{G}$).

Moreover, recall that for a non-deterministic functor $\mathcal{G}$, bisimilarity of $\mathcal{G}$-expressions is decided based on the relation lifting $\overline{\mathcal{G}}$ over "structured" expressions in $\mathcal{G}(\mathsf{Exp}_{\mathcal{G}})$ (Definition 1). Therefore, the deduction relation $\vdash$ has to be extended to allow a restricted contextual reasoning over "structured" expressions in $\mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$, for all ingredients $\mathcal{F}$ of $\mathcal{G}$.

The aforementioned algebraic specifications and the extension of $\vdash$ are modeled as follows.

The algebraic specification of a non-deterministic functor $\mathcal{G}$. It includes:

- the translation of the BNF grammar (1), as presented above;
- the specification of the functor ingredients, given by a sort $\mathsf{Ingredient}$ and a constructor $_\lhd_$: $\mathsf{Functor\ Functor} \to \mathsf{Ingredient}$ (according to Definition 2);
- the specification of each alphabet $A = \{a_1, \ldots, a_n\}$ occurring in the definition of $\mathcal{G}$: this consists of a subsort $A < \mathsf{Alph}$, a constant $a_i \colon \to A$ for $i \in \overline{1,n}$, and a distinguished constant A of sort $\mathsf{AlphName}$ used to refer the alphabet in the definition of the functor;
- the specification of each semilattice $\mathsf{B} = (\{b_1, \ldots, b_n\}, \vee, \bot_B)$ occurring in the definition of $\mathcal{G}$: this consists of a subsort $\mathsf{B} < \mathsf{Slt}$, a constant $b_i \colon \to \mathsf{B}$ for $i \in \overline{1,n}$, a distinguished constant B of sort $\mathsf{SltName}$ used to refer the corresponding semilattice in the definition of the functor, and the equations defining $\vee$ and $\bot_B$ (this should be one of b_i);
- an equation defining $\mathcal{G}$ (as a functor expression).

The algebraic specification of generalized regular expressions. It consists of:

- (according to the BNF grammar in Definition 3) a sort Exp representing expressions ε, $\mathsf{FixpVar}$ the sort for the vocabulary of the fixed-point variables, and Slt the sort for the elements of semilattices. Moreover, we consider constructor operations for all the productions. For example, the production $\varepsilon ::= \varepsilon \oplus \varepsilon$ is represented by an operation $_\oplus_ : \mathsf{Exp\ Exp} \to \mathsf{Exp}$, and $\varepsilon ::= \mu x.\gamma$ is represented by $\mu_._ : \mathsf{FixpVar\ Exp} \to \mathsf{Exp}$. (We chose not to provide any restriction to guarantee that γ is a guarded expression, at this stage in the definition of $\mu_._$. However, guards can be easily checked by pattern matching, according to the grammars in Definition 3);
- the specification of the substitution of a fixed-point variable with an expression, given by an operation $_[_/_] \colon \mathsf{Exp\ Exp\ FixpVar} \to \mathsf{Exp}$ and a set of equations – one for each constructor. For example, the equations associated to $\underline{\emptyset}$ and $\oplus$ are: $\underline{\emptyset}[\varepsilon/x] = \underline{\emptyset}$, and respectively, $(\varepsilon_1 \oplus \varepsilon_2)[\varepsilon/x] = (\varepsilon_1[\varepsilon/x]) \oplus (\varepsilon_2[\varepsilon/x])$, where $\varepsilon, \varepsilon_1, \varepsilon_2$ are $\mathcal{G}$-expressions and x is a fixed-point variable;
- the specification of the type-checking relation in Definition 4, given by an operation $_ : _ : \mathsf{Exp\ Ingredient} \to \mathsf{Bool}$ and an equation for each inference rule defining this relation. For example the rule

$$\frac{\vdash \varepsilon_1 \colon \mathcal{F} \lhd \mathcal{G} \qquad \vdash \varepsilon_2 \colon \mathcal{F} \lhd \mathcal{G}}{\vdash \varepsilon_1 \oplus \varepsilon_2 \colon \mathcal{F} \lhd \mathcal{G}}$$

is represented by the equation $\varepsilon_1 \oplus \varepsilon_2 \colon \mathcal{F} \triangleleft \mathcal{G} = \varepsilon_1 \colon \mathcal{F} \triangleleft \mathcal{G} \wedge \varepsilon_2 \colon \mathcal{F} \triangleleft \mathcal{G}$. The type-checking operator is used in order to verify whether the expressions checked for equivalence are well-typed (Definition 5). Moreover, note that for the consistency of notation, algebraically we write $\varepsilon \colon \mathcal{F} \triangleleft \mathcal{G}$ to represent expressions ε of type $\mathcal{F} \triangleleft \mathcal{G}$.

The algebraic specification of $\delta_{\mathcal{G}}$. It consists of:

- the specification of the coalgebra of $\mathcal{G}$-expressions $\delta_{\mathcal{G}}$ given by three operations $\delta_{_}(_) \colon \mathsf{Ingredient\ Exp} \to \mathsf{ExpStruct}$, $\mathit{Empty} \colon \mathsf{Ingredient} \to \mathsf{ExpStruct}$, and $\mathit{Plus}_{_}(_,_) \colon \mathsf{Ingredient\ ExpStruct\ ExpStruct} \to \mathsf{ExpStruct}$;
- a set of equations describing the definitions of these operations as in [23].

The algebraic specification of structured expressions. As mentioned above, the set of $\mathcal{G}$-expressions is provided with a coalgebraic structure given by the function $\delta_{\mathcal{G}} \colon \mathsf{Exp}_{\mathcal{G}} \to \mathcal{G}(\mathsf{Exp}_{\mathcal{G}})$, where $\mathcal{G}(\mathsf{Exp}_{\mathcal{G}})$ can be understood as the set of expressions with structure given by $\mathcal{G}$ (and its ingredients). The set of structured expressions is defined by the following grammar:

$$\sigma ::= \ \varepsilon \mid b \mid \langle \sigma, \sigma \rangle \mid k_1(\sigma) \mid k_2(\sigma) \mid \bot \mid \top \mid \lambda.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma) \mid \{\sigma\} \qquad (5)$$

where $\varepsilon \in \mathsf{Exp}_{\mathcal{G}}$ and $b \in \mathsf{B}$. The typing rules below give precise meaning to these expressions. Note that $\bot, \top$ are two expressions coming from $\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$, used to denote underspecification and overspecification, respectively.

The associated algebraic specification includes:

- a sort $\mathsf{ExpStruct}$ representing expressions σ (from $\mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$, with $\mathcal{F} \triangleleft \mathcal{G}$), and one operation for each production in the BNF grammar (5). Note that the construction $\lambda.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma)$ has as coalgebraic correspondent a function $f \in \mathcal{F}^A(\mathsf{Exp}_{\mathcal{G}})$, and is defined by cases as follows: $\lambda.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma)(a') = \mathit{if}$ $(a = a')$ $\mathit{then}\ \sigma\ \mathit{else}\ \mathit{Empty}_{\mathcal{F} \triangleleft \mathcal{G}}$;
- the extension of the type-checking relation to structured expressions, defined by:

$$\frac{\vdash b \colon \mathsf{B} \triangleleft \mathcal{G}}{\vdash b \in \mathsf{B}(\mathsf{Exp}\,\mathcal{G})} \qquad \frac{\vdash \varepsilon \colon \mathsf{Id} \triangleleft \mathcal{G}}{\vdash \varepsilon \in \mathsf{Id}(\mathsf{Exp}\,\mathcal{G})}$$

$$\frac{}{\vdash \bot \in \mathcal{F}_1 \oplus \mathcal{F}_2(\mathsf{Exp}\,\mathcal{G})} \qquad \frac{}{\vdash \top \in \mathcal{F}_1 \oplus \mathcal{F}_2(\mathsf{Exp}\,\mathcal{G})}$$

$$\frac{\vdash \sigma \in \mathcal{F}_i(\mathsf{Exp}\,\mathcal{G})}{\vdash k_i(\sigma) \in \mathcal{F}_1 \oplus \mathcal{F}_2(\mathsf{Exp}\,\mathcal{G})}\ i \in \overline{1,2} \qquad \frac{\vdash \sigma_1 \in \mathcal{F}_i(\mathsf{Exp}\,\mathcal{G}) \qquad \vdash \sigma_2 \in \mathcal{F}_i(\mathsf{Exp}\,\mathcal{G})}{\vdash \langle \sigma_1, \sigma_2 \rangle \in \mathcal{F}_1 \times \mathcal{F}_2(\mathsf{Exp}\,\mathcal{G})}$$

$$\frac{\vdash \sigma \in \mathcal{F}(\mathsf{Exp}\,\mathcal{G}),\ a \in A}{\vdash \lambda.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma) \in \mathcal{F}^A(\mathsf{Exp}\,\mathcal{G})} \qquad \frac{\vdash \sigma \in \mathcal{F}(\mathsf{Exp}\,\mathcal{G})}{\vdash \{\sigma\} \in \mathcal{P}_\omega\mathcal{F}(\mathsf{Exp}\,\mathcal{G})}$$

and specified by an operation $_ \in _(\mathsf{Exp}\ _) : \mathsf{ExpStruct\ Functor\ Functor} \to$

Bool (where we used a mix-fix notation) and an equation for each of the above inference rules. For example, the first rule has associated the equation $b \in \mathsf{B}(\mathsf{Exp}\,\mathcal{G}) = b\colon \mathsf{B} \triangleleft \mathcal{G}$. For consistency of notation, we write $\sigma \in \mathcal{F}(\mathsf{Exp}_\mathcal{G})$ to denote that σ is an element of $\mathcal{F}(\mathsf{Exp}_\mathcal{G})$.

Remark 4. *In terms of membership equational logic (MEL) [3], both $\mathcal{F} \triangleleft \mathcal{G}$ and $\mathcal{F}(\mathsf{Exp}\,\mathcal{G})$ can be thought of as being sorts and, for example, $\varepsilon\colon \mathcal{F} \triangleleft \mathcal{G}$ as a membership assertion. Even if MEL is an elegant theory, we prefer not to use it here because this implies the dynamic declaration of sorts and a set of assertions for such a sort. The above approach is generic and therefore more flexible.*

The equational entailment relation $\vdash_{NDF}$ for bisimilarity checking. As previously hinted in the beginning of this section, in order to algebraically reason on bisimilarity of $\mathcal{G}$-expressions in CIRC, one has to extend the deduction relation $\vdash$ to allow a restricted contextual reasoning on expressions in $\mathcal{F}(\mathsf{Exp}_\mathcal{G})$, for all ingredients $\mathcal{F}$ of a non-deterministic functor $\mathcal{G}$. We call the extended entailment $\vdash_{NDF}$.

The aforementioned restriction refers to inhibiting the use of congruence during equational reasoning, in order to guarantee the soundness of CIRC proofs. This is realized by means of a *freezing operator*, which intuitively behaves as a wrapper on the expressions checked for equivalence, by changing their sort to a fresh sort Frozen. This way, the hypotheses collected during a CIRC proof session cannot be used freely in contextual reasoning, hence preventing the derivation of untrue equations (as illustrated in Example 2).

We further show how the freezing mechanism is implemented in our algebraic setting, and define $\vdash_{NDF}$.

Let $\mathcal{E}$ be an algebraic specification. We extend $\mathcal{E}$ by adding the freezing operation $\boxed{-}\colon s \to \mathsf{Frozen}$ for each sort $s \in \Sigma$, where Frozen is a fresh sort. By $\boxed{t}$ we represent the *frozen* form of a Σ-term t, and by $\boxed{e}$ a *frozen equation* of the shape $(\forall X)\,\boxed{t} = \boxed{t'}$ if c. The entailment relation $\vdash$ is defined over frozen equations following the line in [17]; more details are provided in Section 5.

Recall from Section 2 that a relation $\mathcal{R} \subseteq \mathsf{Exp}_\mathcal{G} \times \mathsf{Exp}_\mathcal{G}$ is a bisimulation if and only if $(s,t) \in \mathcal{R} \Rightarrow (\delta_{\mathcal{G} \triangleleft \mathcal{G}}(s), \delta_{\mathcal{G} \triangleleft \mathcal{G}}(t)) \in \overline{\mathcal{G}}(\mathcal{R})$. Here, $\overline{\mathcal{G}}(\mathcal{R}) \subseteq \mathcal{G}(\mathsf{Exp}_\mathcal{G}) \times \mathcal{G}(\mathsf{Exp}_\mathcal{G})$ is the lifting of the relation $\mathcal{R} \subseteq \mathsf{Exp}_\mathcal{G} \times \mathsf{Exp}_\mathcal{G}$, defined as

$$\overline{\mathcal{G}}(\mathcal{R}) = \{(\mathcal{G}(\pi_1)(x), \mathcal{G}(\pi_2)(x)) \mid x \in \mathcal{G}(\mathcal{R})\}\ .$$

So, intuitively, reasoning on bisimilarity of two expressions $(\varepsilon, \varepsilon')$ in $\mathcal{R}$ reduces to checking whether the application of $\delta_\mathcal{G}$ maps them into $\overline{\mathcal{G}}(\mathcal{R})$.

Therefore, checking whether a pair (s^δ, t^δ) is in $\overline{\mathcal{G}}(\mathcal{R})$ consists in checking, for example for the case of $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2$, whether $(s_1^\delta, t_1^\delta) \in \overline{\mathcal{G}_1}(\mathcal{R})$ and $(s_2^\delta, t_2^\delta) \in \overline{\mathcal{G}_2}(\mathcal{R})$, where $s^\delta = \langle s_1^\delta, s_2^\delta \rangle$ and $t^\delta = \langle t_1^\delta, t_2^\delta \rangle$. In an algebraic setting, this would reduce to building an algebraic specification $\mathcal{E}$ and defining an entailment relation $\vdash_{NDF}$ such that one can infer $\mathcal{E} \vdash_{NDF} \boxed{\langle s_1^\delta, s_2^\delta \rangle} = \boxed{\langle t_1^\delta, t_2^\delta \rangle}$ (this is the algebraic correspondent we consider for $(\langle s_1^\delta, s_2^\delta \rangle, \langle t_1^\delta, t_2^\delta \rangle) \in \overline{\mathcal{G}}(\mathcal{R})$) by showing

$\mathcal{E} \vdash_{NDF} \boxed{s_1^\delta} = \boxed{t_1^\delta}$ (or $(s_1^\delta, t_1^\delta) \in \overline{\mathcal{G}_1}(\mathcal{R})$) and $\mathcal{E} \vdash_{NDF} \boxed{s_2^\delta} = \boxed{t_2^\delta}$ (or $(s_2^\delta, t_2^\delta) \in \overline{\mathcal{G}_2}(\mathcal{R})$). We hint that the aforementioned algebraic specification $\mathcal{E}$ consists of $\mathcal{E}_\mathcal{G}$ and a set of frozen equations (see Corollary 1).

The entailment relation $\vdash_{NDF}$ for reasoning on bisimilarity of $\mathcal{G}$-expressions is based on the definition of $\overline{\mathcal{G}}$.

Definition 6. The entailment relation $\vdash_{NDF}$ is the extension of $\vdash$ with the following inference rules, which allow a restricted contextual reasoning over the frozen equations of structured expressions:

$$\frac{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\sigma_1} = \boxed{\sigma_1'} \quad \mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\sigma_2} = \boxed{\sigma_2'}}{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\langle \sigma_1, \sigma_2 \rangle} = \boxed{\langle \sigma_1', \sigma_2' \rangle}} \tag{6}$$

$$\frac{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\sigma} = \boxed{\sigma'}}{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{k_i(\sigma)} = \boxed{k_i(\sigma')}} \; i \in \overline{1,2} \tag{7}$$

$$\frac{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{f(a)} = \boxed{g(a)},\ \textit{for all } a \in A}{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{f} = \boxed{g}} \tag{8}$$

$$\frac{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\sigma_{i_1}} = \boxed{\sigma_{j_1}'}, \ldots, \mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\sigma_{i_k}} = \boxed{\sigma_{j_k}'}}{\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\{\sigma_1, \ldots, \sigma_n\}} = \boxed{\{\sigma_1', \ldots, \sigma_m'\}}} \; \begin{array}{l} \{i_1, \ldots, i_k\} = \{1, \ldots, n\} \\ \{j_1, \ldots, j_k\} = \{1, \ldots, m\} \end{array} \tag{9}$$

Remark 5. *Note that the extension of the entailment relation $\vdash$ to $\vdash_{NDF}$ implies that $\mathcal{E}_\mathcal{G} \vdash e$ iff $\mathcal{E}_\mathcal{G} \vdash_{NDF} e$ holds, for any equation e of shape $\boxed{\varepsilon_1} = \boxed{\varepsilon_2}$ or $\varepsilon_1 = \varepsilon_2$, with $\varepsilon_1, \varepsilon_2$ non-structured expressions. Below, we will use the notation $\mathcal{E}_\mathcal{G} \vdash_{NDF} \mathcal{R}$, where $\mathcal{R}$ is a set of possibly frozen equations, to denote $\forall_{e \in \mathcal{R}} \cdot \mathcal{E}_\mathcal{G} \vdash_{NDF} e$.*

It is interesting to recall the relation lifting for the powerset functor which is encoded in the last rule of Definition 6. A pair (U, V) is in $\overline{\mathcal{P}_\omega \mathcal{G}}(\mathcal{R})$ if and only if for every $u \in U$ there exists a $v \in V$ such that (u, v) belongs to $\overline{\mathcal{G}}(\mathcal{R})$ and, conversely, for every $v \in V$, there exists a $u \in U$ such that (u, v) belongs to $\overline{\mathcal{G}}(\mathcal{R})$.

Remark 6. *As already hinted (and proved in Corollary 1), reasoning on bisimilarity of expressions in a binary relation $\mathcal{R} \subseteq \mathsf{Exp}_\mathcal{G} \times \mathsf{Exp}_\mathcal{G}$ reduces to showing that $\boxed{\delta_\mathcal{G}(s)} = \boxed{\delta_\mathcal{G}(t)}$ is a $\vdash_{NDF}$-consequence, for all $(s, t) \in \mathcal{R}$. The equational proof is performed in a "top-down" fashion, by reasoning on the subsequent equalities between the components of the corresponding structured expression $\delta_\mathcal{G}(s)$, $\delta_\mathcal{G}(t)$ in an inductive manner. This is realized by applying the inverted rules (6)–(9).*

Moreover, note that rule (9) is not invertible in the usual sense; rather any statement matching the form of the conclusion can only be proved by some instance of the rule.

We will further formalize the connection between the inductive definition of $\overline{\mathcal{G}}$ (on the coalgebraic side) and $\vdash_{NDF}$ (on the algebraic side) in Theorem 2, hence enabling the definition of bisimulations in algebraic terms, in Corollary 1.

Remark 7. *Equations in $\mathcal{E}_{\mathcal{G}}$ (built as previously described in this section) are used in the equational reasoning only for reducing terms of shape $\mathsf{op}(t_1, \ldots, t_n)$ according to the definition of the operation op. For the simplicity of the proofs of Theorem 2 and Corollary 1, whenever we write $\mathsf{op}(t_1, \ldots, t_n)$, we refer to the associated term reduced according to the definition of op.*

First we introduce some notation conventions. Let $\mathcal{G}$ be a non-deterministic functor and $\mathcal{R} \subseteq \mathsf{Exp}_{\mathcal{G}} \times \mathsf{Exp}_{\mathcal{G}}$. We write:

- $\mathcal{R}_{id}$ to denote the set $\mathcal{R} \cup \{(\varepsilon, \varepsilon) \mid \mathcal{E}_{\mathcal{G}} \vdash \varepsilon \colon \mathcal{G} \triangleleft \mathcal{G} = true\}$;
- $cl(\mathcal{R})$ for the closure of $\mathcal{R}$ under transitivity, symmetry and reflexivity;
- $\boxed{\mathcal{R}}$ to represent the set $\bigcup_{e \in \mathcal{R}} \{\boxed{e}\}$; (application of the freezing operator to all elements of $\mathcal{R}$)
- $\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon = \varepsilon')$ to represent the equation $\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon) = \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon')$;
- $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}}$ as a shorthand for $(S, \Sigma, E \cup \{\boxed{\varepsilon} = \boxed{\varepsilon'} \mid (\varepsilon, \varepsilon') \in \mathcal{R}\})$, where $\mathcal{E}_{\mathcal{G}} = (S, \Sigma, E)$;
- $(\sigma, \sigma') \in \overline{\mathcal{G}}(\mathcal{R})$ as a shorthand for: (σ, σ') is among the enumerated elements of a set S explicitly constructed as an enumeration of the finite set $\overline{\mathcal{G}}(\mathcal{R})$ (in the algebraic setting, $\overline{\mathcal{G}}(\mathcal{R})$ is a subset of $\mathcal{T}_{\Sigma,\mathsf{ExpStruct}} \times \mathcal{T}_{\Sigma,\mathsf{ExpStruct}}$ and $\mathcal{E}_{\mathcal{G}} \vdash \overline{\mathcal{G}}(\mathcal{R}) = S$).

Theorem 2. *Consider a non-deterministic functor $\mathcal{G}$. Let $\mathcal{F}$ be an ingredient of $\mathcal{G}$, $\mathcal{R}$ a binary relation on the set of $\mathcal{G}$-expressions, and $\sigma, \sigma' \in \mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$.*

a) If $\mathcal{G}$ is not a constant functor, then $(\sigma, \sigma') \in \overline{\mathcal{F}}(cl(\mathcal{R}_{id}))$ iff $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma} = \boxed{\sigma'}$;

b) If $\mathcal{G}$ is a constant functor B, then $(\sigma, \sigma') \in \overline{\mathsf{B}}(cl(\mathcal{R}_{id}))$ iff $\mathcal{E}_{\mathcal{G}} \vdash_{NDF} \boxed{\sigma} = \boxed{\sigma'}$.

In order to prove Theorem 2.a) we introduce the following lemma:

Lemma 1. *Consider $\mathcal{G}$ a non-deterministic functor and $\mathcal{R}$ a binary relation on the set of $\mathcal{G}$-expressions. If $(\varepsilon, \varepsilon') \in cl(\mathcal{R}_{id})$ then $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\varepsilon} = \boxed{\varepsilon'}$.*

Proof. The proof is trivial, as equality is reflexive, symmetric and transitive. □

We are now ready to prove Theorem 2.

Proof (Theorem 2).

- Proof of Theorem 2.a).

- " $\Rightarrow$ ". The proof is by induction on the structure of $\mathcal{F}$.
  Base case:
  - $\mathcal{F} = \mathsf{B}$. It follows that (σ, σ') is of shape (b, b) where $b \in \mathsf{B}$, therefore $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{b} = \boxed{b}$ holds by reflexivity.
  - $\mathcal{F} = \mathsf{Id}$. In this case $(\sigma, \sigma') \in cl(\mathcal{R}_{id}) = \overline{\mathsf{Id}}(cl(\mathcal{R}_{id}))$, so the result follows immediately by Lemma 1.

  Induction step:
  - $\mathcal{F} = \mathcal{F}_1 \times \mathcal{F}_2$. Obviously, $\sigma = \langle \sigma_1, \sigma_2 \rangle$ and $\sigma' = \langle \sigma'_1, \sigma'_2 \rangle$, where $(\sigma_1, \sigma'_1) \in \overline{\mathcal{F}_1}(cl(\mathcal{R}_{id}))$ and $(\sigma_2, \sigma'_2) \in \overline{\mathcal{F}_2}(cl(\mathcal{R}_{id}))$. Therefore, by the induction hypothesis, both $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma_1} = \boxed{\sigma'_1}$ and $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma_2} = \boxed{\sigma'_2}$ hold. Hence, according to the definition of $\vdash_{NDF}$ (see (6)), we conclude that $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\langle \sigma_1, \sigma_2 \rangle} = \boxed{\langle \sigma'_1, \sigma'_2 \rangle}$ holds.
  - The cases $\mathcal{F} = \mathcal{F}_1 \oplus \mathcal{F}_2$, $\mathcal{F} = \mathcal{F}_1^A$ and $\mathcal{F} = \mathcal{P}_\omega \mathcal{F}'$ are handled in a similar way.

- " $\Leftarrow$ ". We proceed also by induction on the structure of $\mathcal{F}$. Moreover, recall that the observations in Remark 7 hold (for each of the subsequent cases).
  Base case:
  - $\mathcal{F} = \mathsf{B}$. In this case (σ, σ') is of shape (b, b'), where b, b' are two elements of the semilattice B. Also, recall that $\mathcal{G} \neq \mathsf{B}$, therefore, the equations (of type $\mathcal{G} \lhd \mathcal{G} \neq \mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$) in $\mathcal{R}$ are not involved in the equational reasoning. We deduce that $\boxed{b} = \boxed{b'}$ is proved by reflexivity, hence $(b, b') = (b, b) \in \overline{\mathsf{B}}(cl(\mathcal{R}_{id}))$.
  - $\mathcal{F} = \mathsf{Id}$. Note that for this case, σ, σ' are expressions of the same type with the expressions in $\mathcal{R}$. We further identify two possibilities:
    - $\boxed{\sigma} = \boxed{\sigma'}$ is proved by reflexivity, therefore $(\sigma, \sigma') \in \{(\varepsilon, \varepsilon) \mid \varepsilon{:}\mathcal{G} \lhd \mathcal{G}\} \subseteq \mathcal{R}_{id} \subseteq cl(\mathcal{R}_{id}) = \overline{\mathsf{Id}}(cl(\mathcal{R}_{id}))$.
    - the equations in $\boxed{\mathcal{R}}$ are used in the equational reasoning $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma} = \boxed{\sigma'}$. In addition, the freezing operator inhibits contextual reasoning, therefore $\boxed{\sigma} = \boxed{\sigma'}$ is proved according to the equations in $\boxed{\mathcal{R}}$, based on the symmetry and transitivity of $\vdash_{NDF}$. In other words, $(\sigma, \sigma') \in cl(\mathcal{R}_{id}) = \overline{\mathsf{Id}}(cl(\mathcal{R}_{id}))$.

  Induction step:
  - $\mathcal{F} = \mathcal{F}_1 \times \mathcal{F}_2$. Obviously, due to their type, the equations in $\mathcal{R}$ are not involved in the equational reasoning. Also, recall that (*) holds. Therefore, $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\langle \sigma_1, \sigma_2 \rangle} = \boxed{\langle \sigma'_1, \sigma'_2 \rangle}$ is a consequence of the inverted rule (6). More explicitly, it follows that $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma_1} = \boxed{\sigma'_1}$ and $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma_2} =$

$\boxed{\sigma_2'}$ must hold. By the induction hypothesis, we deduce that $(\sigma_1, \sigma_1') \in \overline{\mathcal{F}_1}(cl(\mathcal{R}_{id}))$ and $(\sigma_2, \sigma_2') \in \overline{\mathcal{F}_2}(cl(\mathcal{R}_{id}))$. So by the definition of $\overline{\mathcal{F}_1 \times \mathcal{F}_2}$ we conclude that $(\langle \sigma_1, \sigma_2 \rangle, \langle \sigma_1', \sigma_2' \rangle) = (\sigma, \sigma') \in \overline{\mathcal{F}_1 \times \mathcal{F}_2}(\mathcal{R})$.

  * The cases $\mathcal{F} = \mathcal{F}_1 \oplus \mathcal{F}_2$, $\mathcal{F} = (\mathcal{F}_1)^A$ and $\mathcal{F} = \mathcal{P}_\omega \mathcal{F}'$ follow a similar reasoning.

- Proof of Theorem 2.*b*). It follows immediately by the definition of $\overline{\mathsf{B}}$ and Remark 7.

□

Corollary 1. *Let $\mathcal{G}$ be a non-deterministic functor and $\mathcal{R}$ a binary relation on the set of $\mathcal{G}$-expressions.*

a) If $\mathcal{G}$ is not a constant functor, then $cl(\mathcal{R}_{id})$ is a bisimulation iff $\mathcal{E}_\mathcal{G} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{R})}$;

b) If $\mathcal{G}$ is a constant functor B, then $cl(\mathcal{R}_{id})$ is a bisimulation iff $\mathcal{E}_\mathcal{G} \vdash_{NDF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{R})}$.

Proof.

- Proof of Corollary 1.*a*). We reason as follows:

$$
\begin{array}{lll}
 & cl(\mathcal{R}_{id}) \text{ is a bisimulation} & \\
\Leftrightarrow & (\forall (\varepsilon, \varepsilon') \in cl(\mathcal{R}_{id})).((\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon), \delta_{\mathcal{G} \triangleleft \mathcal{G}}(\varepsilon')) \in \overline{\mathcal{G}}(cl(\mathcal{R}_{id})) & \text{(Def. 1)} \\
\Leftrightarrow & \mathcal{E}_\mathcal{G} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(cl(\mathcal{R}_{id}))} & \text{(Thm. 2)} \\
\Leftrightarrow & \mathcal{E}_\mathcal{G} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(\mathcal{R})} & (cl(\mathcal{R}_{id}), \vdash_{NDF})
\end{array}
$$

- Proof of Corollary 1.*b*). It follows immediately by the definition of bisimulation relations and according to the observations in Remark 7.

□

In Figure 2 we briefly summarize the results of the current section, namely, the algebraic encoding of the coalgebraic setting presented in [23].

5. A Decision Procedure for Bisimilarity in CIRC

In this section, we describe how the coinductive theorem prover CIRC [14] can be used to implement the decision procedure for the bisimilarity of generalized regular expressions, which we discussed above.

coalgebraic	algebraic
$\vdash \varepsilon\colon \mathcal{F} \lhd \mathcal{G}$	$\mathcal{E}_{\mathcal{G}} \vdash \varepsilon\colon \mathcal{F} \lhd \mathcal{G} = true$
$\mathsf{Exp}_{\mathcal{F} \lhd \mathcal{G}}$	$\{\varepsilon \in \mathcal{T}_{\Sigma,\mathsf{Exp}} \mid \mathcal{E}_{\mathcal{G}} \vdash \varepsilon\colon \mathcal{F} \lhd \mathcal{G} = true\}$
$\mathsf{Exp}_{\mathcal{G}}$	$\{\varepsilon \in \mathcal{T}_{\Sigma,\mathsf{Exp}} \mid \mathcal{E}_{\mathcal{G}} \vdash \varepsilon\colon \mathcal{G} \lhd \mathcal{G} = true\}$
$\mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$	$\{\sigma \in \mathcal{T}_{\Sigma,\mathsf{ExpStruct}} \mid \mathcal{E}_{\mathcal{G}} \vdash \sigma \in \mathcal{F}(\mathsf{Exp}\,\mathcal{G}) = true\}$
$\delta_{\mathcal{F} \lhd \mathcal{G}}\colon \mathsf{Exp}_{\mathcal{F} \lhd \mathcal{G}} \to \mathcal{F}(\mathsf{Exp}_{\mathcal{G}})$	$\delta_{_}(_)\colon \mathsf{Ingredient\ Exp} \to \mathsf{ExpStruct}$
$(\sigma, \sigma') \in \overline{\mathcal{F}}(cl(\mathcal{R}_{id}))$	$\mathcal{E}_{\mathcal{G}} \vdash \sigma \in \mathcal{F}(\mathsf{Exp}\,\mathcal{G}) = true,$ $\mathcal{E}_{\mathcal{G}} \vdash \sigma' \in \mathcal{F}(\mathsf{Exp}\,\mathcal{G}) = true$ $\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\sigma} = \boxed{\sigma'}$ if $\mathcal{G} \neq \mathsf{B}$ or $\mathcal{E}_{\mathcal{G}} \vdash_{NDF} \boxed{\sigma} = \boxed{\sigma'}$ if $\mathcal{G} = \mathsf{B}$ (Thm. 2)
$cl(\mathcal{R}_{id})$ is a bisimulation	$\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{R}} \vdash_{NDF} \boxed{\delta_{\mathcal{G} \lhd \mathcal{G}}(\mathcal{R})}$ if $\mathcal{G} \neq \mathsf{B}$ or $\mathcal{E}_{\mathcal{G}} \vdash_{NDF} \boxed{\delta_{\mathcal{G} \lhd \mathcal{G}}(\mathcal{R})}$ if $\mathcal{G} = \mathsf{B}$ (Cor. 1)

Figure 2: non-deterministic functors - coalgebraic vs. algebraic approach

CIRC can be seen as an extension of Maude with behavioral features and its implementation is derived from that of Full-Maude. In order to use the prover, one needs to provide a specification (a CIRC theory) and a set of goals. A CIRC theory $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ consists of an algebraic specification (S, Σ, E), a set Δ of *derivatives*, and a set $\mathcal{I}$ of equational interpolants, which are expressions of the form $e \Rightarrow \{e_i \mid i \in I\}$ where e and e_i are equations. The intuition for this type of expressions is simple: e holds whenever for any i in I the equation e_i holds. In other words, to prove $E \vdash e$ one can chose to instead prove $E \vdash \{e_i \mid i \in I\}$. For the particular case of non-deterministic functors, we use equational interpolants to extend the initial entailment relation in a consistent way with rules (6)–(9). (For more information on equational interpolants see [7]). A derivative $\delta \in \Delta$ is a Σ-term containing a special variable $*{:}s$ (*i.e.*, a Σ-context), where s is the sort of the variable $*$. If e is an equation $t = t'$ with t and t' of sort s, then $\delta[e]$ is $\delta[t/*{:}s] = \delta[t'/*{:}s]$. We call this type of equation a *derivable equation*. The other equations are *non-derivable*. We write $\delta[\mathcal{R}]$ to represent $\{\delta[e] \mid e \in \mathcal{R}\}$, where $\mathcal{R}$ is a set of derivable equations, and $\Delta[e]$ for the set $\{\delta[e] \mid \delta \in \Delta$ appropriate for $e\}$.

Moreover, note that CIRC works with an extension of the entailment relation $\vdash$ over frozen equations (introduced in Section 4), with two more axioms, as in [17]:

$$E \cup \mathcal{R} \vdash \boxed{e} \ \textit{ iff } \ E \vdash e \tag{10}$$

$$E \cup \mathcal{R} \vdash \mathcal{G} \ \textit{ implies } \ E \cup \delta[\mathcal{R}] \vdash \delta[\mathcal{G}] \ \textit{for each } \delta \in \Delta \tag{11}$$

Above, E ranges over unfrozen equations, e over non-derivable unfrozen

equations, and $\mathcal{R}, \mathcal{G}$ over derivable frozen equations.

Remark 8. *Note that the new entailment* $\vdash_{NDF}$ *extended over frozen equations (in Definition 6) satisfies the assumptions (10) and (11).*

CIRC implements the coinductive proof system given in [17] using a set of reduction rules of the form $(\mathcal{B}, \mathcal{F}, \mathcal{G}) \Rightarrow (\mathcal{B}, \mathcal{F}', \mathcal{G}')$, where $\mathcal{B}$ represents a specification, $\mathcal{F}$ is the coinductive hypothesis (a set of frozen equations) and $\mathcal{G}$ is the current set of goals. The freezing operator is defined as described in Section 4. Here is a brief description of these rules:

[Done]: $(\mathcal{B}, \mathcal{F}, \{\}) \Rightarrow \cdot$
Whenever the set of goals is empty, the system terminates with success.

[Reduce]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ *if* $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$
If the current goal is a $\vdash$-consequence of $\mathcal{B} \cup \mathcal{F}$ then $\boxed{e}$ is removed from the set of goals.

[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\}, \mathcal{G} \cup \boxed{\Delta[e]})$ *if* $\mathcal{B} \cup \mathcal{F} \not\vdash \boxed{e}$
When the current goal e is derivable and it is not a $\vdash$-consequence, it is added to the hypothesis and its derivatives to the set of goals.

[Simplify]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{\theta(e)}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{\theta(e_i)} \mid i \in I\})$
if $e \Rightarrow \{e_i \mid i \in I\}$ is an equational interpolant from the specification and $\theta \colon X \to \mathcal{T}_\Sigma(Y)$ is a substitution.

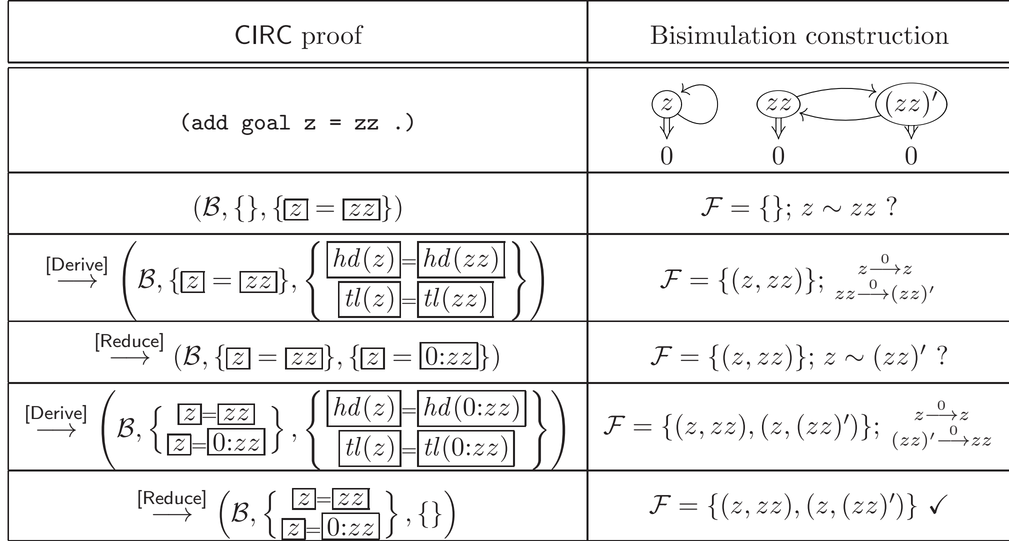
[Fail]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow$ *failure if* $\mathcal{B} \cup \mathcal{F} \not\vdash \boxed{e} \wedge e$ *is non-derivable*
This rule stops the reduction process with failure whenever the current goal e is non-derivable and is not a $\vdash$-consequence of $\mathcal{B} \cup \mathcal{F}$.

It is worth noting that there is a strong connection between a CIRC proof and the construction of a bisimulation relation. We illustrate this fact and the importance of the freezing operator with a simple example.

Example 2. *Consider the case of infinite streams. The set* B^ω *of infinite streams over a set* B *is the final coalgebra of the functor* $\mathcal{S} = \mathsf{B} \times \mathsf{Id}$*, with a coalgebra structure given by hd and tl, the functions that return the head and the tail of the stream, respectively. Our purpose is to prove that* $0^\infty = (00)^\infty$*. Let z and zz represent the stream on the left hand side and, respectively, on the right hand side. These streams are defined by the equations:* $hd(z) = 0, tl(z) = z, hd(zz) = 0, tl(zz) = 0{:}zz$*. Note that equations over* B *like* $hd(z) = 0$ *are not derivable and equations over streams like* $tl(z) = z$ *are derivable.*

In Fig. 3 we present the correlation between the CIRC proof and the construction of the bisimulation relation. Note how CIRC collects the elements of the bisimulation as frozen hypotheses.

Let us analyze what would happen if the freezing operator $\boxed{-}$ *were not used. Suppose the circular coinduction algorithm would add the equation* $z = zz$ *in its unfrozen form to the hypotheses. After applying the derivatives we obtain*



CIRC proof	Bisimulation construction
`(add goal z = zz .)`	$z \Downarrow 0$, $zz \Downarrow 0$, $(zz)' \Downarrow 0$
$(\mathcal{B}, \{\}, \{\boxed{z} = \boxed{zz}\})$	$\mathcal{F} = \{\};\ z \sim zz$?
$\xrightarrow{\text{[Derive]}} \left(\mathcal{B}, \{\boxed{z} = \boxed{zz}\}, \left\{\begin{array}{c}\boxed{hd(z)} = \boxed{hd(zz)} \\ \boxed{tl(z)} = \boxed{tl(zz)}\end{array}\right\}\right)$	$\mathcal{F} = \{(z, zz)\};\ \begin{array}{c} z \xrightarrow{0} z \\ zz \xrightarrow{0} (zz)' \end{array}$
$\xrightarrow{\text{[Reduce]}} (\mathcal{B}, \{\boxed{z} = \boxed{zz}\}, \{\boxed{z} = \boxed{0{:}zz}\})$	$\mathcal{F} = \{(z, zz)\};\ z \sim (zz)'$?
$\xrightarrow{\text{[Derive]}} \left(\mathcal{B}, \left\{\begin{array}{c}\boxed{z} = \boxed{zz} \\ \boxed{z} = \boxed{0{:}zz}\end{array}\right\}, \left\{\begin{array}{c}\boxed{hd(z)} = \boxed{hd(0{:}zz)} \\ \boxed{tl(z)} = \boxed{tl(0{:}zz)}\end{array}\right\}\right)$	$\mathcal{F} = \{(z, zz), (z, (zz)')\};\ \begin{array}{c} z \xrightarrow{0} z \\ (zz)' \xrightarrow{0} zz \end{array}$
$\xrightarrow{\text{[Reduce]}} \left(\mathcal{B}, \left\{\begin{array}{c}\boxed{z} = \boxed{zz} \\ \boxed{z} = \boxed{0{:}zz}\end{array}\right\}, \{\}\right)$	$\mathcal{F} = \{(z, zz), (z, (zz)')\}$ ✓

Figure 3: Parallel between a CIRC proof and the bisimulation construction

the goals $hd(z) = hd(zz), tl(z) = tl(zz)$. At this point, the prover could use the freshly added equation $z = zz$, and according to the congruence rule, both goals would be proven directly, though we would still be in the process of showing that the hypothesis holds. By following a similar reasoning, we could also prove that $0^\infty = 1^\infty$! In order to avoid these situations, the hypotheses are frozen, (i.e., their sort is changed from Stream *to* Frozen*) and this stops the application of the congruence rule, forcing the application of the derivatives according to their definition in the specification. Therefore, the use of the freezing operator is vital for the soundness of circular coinduction.*

Next, we focus on using CIRC for automatically reasoning on the equivalence of $\mathcal{G}$-expressions. As we will show, the implementation of both the algebraic specifications associated to non-deterministic functors and the equational entailment relation described in Section 4 is immediate. Given a non-deterministic functor $\mathcal{G}$, we define a CIRC theory $\mathcal{B}_\mathcal{G} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ as follows:

- (S, Σ, E) is $\mathcal{E}_\mathcal{G}$
- $\Delta = \{\delta_{\mathcal{G} \triangleleft \mathcal{G}}(*{:}\mathsf{Exp})\}$, so the only derivable equations are those of sort `Exp`. As we have already seen for the example of streams, equations of sort `Slt` must not be derivable. Since we have the subsort relation `Slt` $<$ `Exp`, we avoid the application of the derivative $\delta_{\mathcal{G} \triangleleft \mathcal{G}}(*{:}\mathsf{Exp})$ over equations of sort `Slt` by means of an interpolant (see below).
- $\mathcal{I}$ consists of the following equational interpolants , whose role is to replace

current proof obligations over non-trivial structures with simpler ones:

$$\langle \sigma_1, \sigma_2 \rangle = \langle \sigma'_1, \sigma'_2 \rangle \;\Rightarrow\; \{\sigma_1 = \sigma'_1,\; \sigma_2 = \sigma'_2\} \quad (12)$$
$$k_i(\sigma) = k_i(\sigma') \;\Rightarrow\; \{\sigma = \sigma'\} \quad (13)$$
$$f = g \;\Rightarrow\; \{f(a) = g(a) \mid a \in A\} \quad (14)$$
$$\cup_{i \in \overline{1,n}} \{\sigma_i\} = \cup_{j \in \overline{1,m}} \{\sigma'_j\} \;\Rightarrow\; \{\wedge_{i \in \overline{1,n}} (\vee_{j \in \overline{1,m}} \sigma_i = \sigma'_j) \;\; \wedge_{j \in \overline{1,m}} (\vee_{i \in \overline{1,n}} \sigma_i = \sigma'_j)\} \quad (15)$$

together with an equational interpolant

$$t = t' \;\Rightarrow\; \{t \simeq t' = \mathtt{true}\} \quad (16)$$

where $\simeq$ is the equality predicate equationally defined over the sort `Slt`. The last interpolant transforms the equations of sort `Slt` from derivable (because of the subsort relation $\mathtt{Slt} < \mathtt{Exp}$) into non-derivable and equivalent ones.

The interpolants (12–16) in $\mathcal{I}$ extend the entailment relation $\vdash_{NDF}$ (introduced in Definition 6) as follows:

$$\frac{E \vdash_{NDF} \{e_i \mid i \in I\}}{E \vdash_{NDF} e} \text{ if } e \Rightarrow \{e_i \mid i \in I\} \text{ in } \mathcal{I}$$

Theorem 3 (Soundness). *Let $\mathcal{G}$ be a non-deterministic functor, and $\mathcal{G}$ a binary relation on the set of $\mathcal{G}$-expressions.*
If $(\mathcal{B}_{\mathcal{G}}, \mathcal{F}_0 = \{\}, \mathcal{G}_0 = \boxed{\mathcal{G}}) \overset{*}{\Rightarrow} (\mathcal{B}_{\mathcal{G}}, \mathcal{F}_n, \mathcal{G}_n = \{\})$ using [Reduce], [Derive] and [Simplify], then $\mathcal{G} \subseteq \sim_{\mathcal{G}}$.

Proof. The idea of the proof is to find a bisimulation relation $\widetilde{\mathcal{F}}$ s.t. $\mathcal{G} \subseteq \widetilde{\mathcal{F}}$. First let $\mathcal{F}$ represent the set of hypotheses (or derived goals) collected during the proof session. We distinguish between two cases:

a) $\mathcal{G} = \mathsf{B}$. For this case, the set of expressions in $\mathcal{G}$ is given by the following grammar:

$$\varepsilon :: = \underline{\emptyset} \mid b \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon \,. \quad (17)$$

Note that the goals $\varepsilon = \varepsilon'$ in $\mathcal{G}$ are proven

1. either according to [Simplify], applied in the context of the equational interpolant (16). If this is the case, then $\varepsilon = \varepsilon'$ holds by reflexivity, therefore

$$\mathcal{B}_{\mathcal{G}} \vdash_{NDF} \boxed{\delta_{\mathsf{B} \triangleleft \mathsf{B}}(\varepsilon)} = \boxed{\delta_{\mathsf{B} \triangleleft \mathsf{B}}(\varepsilon')} \quad (18)$$

also holds;

2. or after the application of [Derive], case in which $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \boxed{\delta_{\mathsf{B}\lhd\mathsf{B}}(\varepsilon)} = \boxed{\delta_{\mathsf{B}\lhd\mathsf{B}}(\varepsilon')}$ holds. Moreover, note that $\delta_{\mathsf{B}\lhd\mathsf{B}}(\varepsilon)$ and $\delta_{\mathsf{B}\lhd\mathsf{B}}(\varepsilon')$ are reduced to b, respectively $b' \in \mathsf{B}$, according to (17) and the definition of $\delta_{\mathsf{B}\lhd\mathsf{B}}$. Consequently, the non-derivable (due to the subsort relation $\mathsf{B} < \mathtt{Slt}$) goal $\boxed{b} = \boxed{b'}$ holds by reflexivity, so the following is a sound statement:

$$\mathcal{B}_{\mathcal{G}} \vdash_{NDF} \boxed{\delta_{\mathsf{B}\lhd\mathsf{B}}(\varepsilon)} = \boxed{\delta_{\mathsf{B}\lhd\mathsf{B}}(\varepsilon')}. \tag{19}$$

Based on (18), (19) and Corollary 1.b), we conclude that $\widetilde{\mathcal{F}} = cl(\mathcal{G}_{id})$ is a bisimulation, hence $\mathcal{G} \subseteq cl(\mathcal{G}_{id}) \subseteq \sim_{\mathcal{G}}$.

b) $\mathcal{G} \neq \mathsf{B}$. Based on the reduction rules implemented in CIRC, it is quite easy to see that the initial set of goals $\mathcal{G}$ is a $\vdash_{NDF}$-consequence of $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}}$. In other words, $\mathcal{G} \subseteq cl(\mathcal{F}_{id})$. So, if we anticipate a bit, we should show that $\widetilde{\mathcal{F}} = cl(\mathcal{F}_{id})$ is a bisimulation, *i.e.*, according to Corollary 1, $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \boxed{\delta_{\mathcal{G}\lhd\mathcal{G}}(\mathcal{F})}$. This is achieved by proving that $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \mathcal{G}_i (i \in \overline{0,n})$ (note that $\boxed{\delta_{\mathcal{G}\lhd\mathcal{G}}(\mathcal{F})} \subseteq \bigcup_{i\in\overline{0,n}} \mathcal{G}_i$, according to [Derive]). The proof is by induction on j, where $n-j$ is the current proof step, and by case analysis on the CIRC reduction rules applied at each step.

We further provide a sketch of the proof.
The *base case* $j = n$ follows immediately, as $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \mathcal{G}_n = \emptyset$.
For the *induction step* we proceed as follows. Let $\boxed{e} \in \mathcal{G}_j$. If $\boxed{e} \in \mathcal{G}_{j+1}$ then $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \boxed{e}$ by the induction hypothesis. If $\boxed{e} \notin \mathcal{G}_{j+1}$ then, for example, if [Reduce] was applied then it holds that $\mathcal{B}_{\mathcal{G}} \cup \mathcal{F}_j \vdash_{NDF} \boxed{e}$. Recall that $\mathcal{F}_j \subseteq \boxed{\mathcal{F}}$, so $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \boxed{e}$ also holds. The result follows in a similar fashion for the application of [Derive] or [Simplify].

□

Remark 9. *The soundness of the proof system we describe in this paper does not follow directly from Theorem 3 in [17]. This is due to the fact that we do not have an experiment-based definition of bisimilarity. So, even though the mechanism we use for proving* $\mathcal{B}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \boxed{\delta_{\mathcal{G}\lhd\mathcal{G}}(\mathcal{F})}$ *(for the case* $\mathcal{G} \neq \mathsf{B}$*) is similar to the one described in [17], the current soundness proof is conceived in terms of bisimulations (and not experiments).*

Remark 10. *The entailment relation* $\vdash_{NDF}$ *that CIRC uses for checking the equivalence of generalized regular expressions is an instantiation of the parametric entailment relation* $\vdash$ *from the proof system in [17]. This approach allows CIRC to reason automatically on a large class of systems which can be modeled as non-deterministic coalgebras.*

As already stated, our final goal is to use CIRC as a decision procedure for the bisimilarity of generalized regular expressions. That is, whenever provided a set

of expressions, the prover stops with a yes/no answer w.r.t. their equivalence. In this context, an important aspect is that the sub-coalgebra generated by an expression $\varepsilon \in \mathsf{Exp}_{\mathcal{G}}$ by repeatedly applying $\delta_{\mathcal{G}}$ is, in general, infinite. Take for example the non-deterministic functor $\mathcal{S} = \mathsf{B} \times \mathsf{Id}$ associated to infinite streams, and consider the property $\mu x.\underline{\emptyset} \oplus r\langle x\rangle = \mu x.r\langle x\rangle$. In order to prove this, CIRC builds an infinite proof sequence by repeatedly applying $\delta_{\mathcal{S}}$ as follows:

$$\begin{array}{rcl} \delta_{\mathcal{S}}(\mu x.\underline{\emptyset} \oplus r\langle x\rangle) & = & \delta_{\mathcal{S}}(\mu x.r\langle x\rangle) \\ & \downarrow & \\ \langle 0, \underline{\emptyset} \oplus (\mu x.\underline{\emptyset} \oplus r\langle x\rangle)\rangle & = & \langle 0, \mu x.r\langle x\rangle\rangle \\ \delta_{\mathcal{S}}(\underline{\emptyset} \oplus (\mu x.\underline{\emptyset} \oplus r\langle x\rangle)) & = & \delta_{\mathcal{S}}(\mu x.r\langle x\rangle) \\ & \downarrow & \\ \langle 0, \underline{\emptyset} \oplus \underline{\emptyset} \oplus (\mu x.\underline{\emptyset} \oplus r\langle x\rangle)\rangle & = & \langle 0, \mu x.r\langle x\rangle\rangle \ [\ldots] \end{array}$$

In this case, the prover would never stop. We observed in Section 3 that Theorem 1 guarantees we can associate a finite coalgebra to a certain expression. In the proof of the aforementioned theorem, which is presented in [23], it is shown that the axioms for associativity, commutativity and idempotence (ACI) of $\oplus$ guarantee finiteness of the generated sub-coalgebra (note that these axioms have also been proven sound w.r.t. bisimulation). ACI properties can easily be specified in CIRC as the prover is an extension of Maude, which has a powerful matching modulo ACUI (ACI plus unity) capability. The idempotence is given by the equation $\varepsilon \oplus \varepsilon = \varepsilon$, and the commutativity and associativity are specified as attributes of $\oplus$. It is interesting to remark that for the powerset functor termination is guaranteed without the axioms, because the coalgebra structure on the expressions for the powerset functor already includes ACI (since $\mathcal{P}_\omega(\mathsf{Exp})$ is itself a join-semilattice).

Theorem 4. *Let $\mathcal{G}$ be a set of proof obligations over generalized regular expressions. CIRC can be used as a decision procedure for the equivalences in $\mathcal{G}$, that is, it can decide whenever a goal $(\varepsilon_1, \varepsilon_2) \in \mathcal{G}$ is a true or false equality.*

Proof. Note that as proven in [23], the ACI axioms for $\oplus$ guarantee that $\delta_{\mathcal{G}}$ is applied for a finite number of times in the generation of the sub-coalgebra associated to a $\mathcal{G}$-expression. Therefore, it straightforwardly follows that by implementing the ACI axioms in CIRC (as attributes of $\oplus$), the set of new goals obtained by applying $\delta_{\mathcal{G}}$ is finite. In these circumstances, whenever CIRC stops according to the reduction rule [Done], the initial proof obligations are bisimilar. On the other hand, whenever it terminates with [Fail], the goals are not bisimilar. □

6. A CIRC-based Tool

We have implemented a tool that, when provided with a functor $\mathcal{G}$, automatically generates a specification for CIRC which can then be used in order to automatically check whether two $\mathcal{G}$-expressions are bisimilar. The tool is implemented as a metalanguage application in Maude. It can be downloaded from

the address `http://goriac.info/tools/functorizer/`. In order to start the tool, one needs to launch Maude along with the extension Full-Maude and load the downloaded file using the command `in functorizer.maude .`

The general use case consists in providing the join-semilattices, the alphabets and the expressions. After these steps, the tool automatically checks if the provided expressions are guarded, closed and correctly typed. If this check succeeds, then it outputs a specification that can be further processed by CIRC. In the end, the prover outputs either the bisimulation, if the expressions are equivalent, or a negative answer, otherwise.

We present two case studies in order to emphasize the high degree of generality for the types of systems we can handle, and show how the tool is used.

Example 3. *We consider the case of Mealy machines, which are coalgebras for the functor* $(\mathsf{B} \times \mathsf{Id})^A$.

Formally, a Mealy machine is a pair (S, α) *consisting of a set* S *of states and a transition function* $\alpha\colon S \to (\mathsf{B} \times S)^A$, *which for each state* $s \in S$ *and input* $a \in A$ *associates an output value* b *and a next state* s'. *Typically, we write* $\alpha(s)(a) = (b, s') \Leftrightarrow \textcircled{s} \xrightarrow{a|b} \textcircled{s'}$.

In this example and in what follows we will consider for the output the two-value join-semilatice $\mathsf{B} = \{0, 1\}$ *(with* $\bot_{\mathsf{B}} = 0$*) and for the input alphabet* $A = \{a, b\}$. *The expressions for Mealy machines are given by the grammar:*

$$
\begin{array}{lll}
E & ::= & \underline{\emptyset} \mid x \mid E \oplus E \mid \mu x.E_2 \mid a(r\langle E\rangle) \mid b(r\langle E\rangle) \mid a(l\langle E_1\rangle) \mid b(l\langle E_1\rangle) \\
E_1 & ::= & \underline{\emptyset} \mid E_1 \oplus E_1 \mid 0 \mid 1 \\
E_2 & ::= & \underline{\emptyset} \mid E_2 \oplus E_2 \mid \mu x.E_2 \mid a(r\langle E\rangle) \mid b(r\langle E\rangle) \mid a(l\langle E_1\rangle) \mid b(l\langle E_1\rangle
\end{array}
$$

Intuitively, an expression of shape $a(l\langle E_1\rangle)$ *specifies a state that for an input* a *has an output value specified by* E_1. *For example, the expression* $a(l\langle 1\rangle)$ *specifies a state that for input* a *outputs* 1, *whereas in the case of* $a(l\langle\underline{\emptyset}\rangle)$ *the output is* 0. *An expression of shape* $a(r\langle E\rangle)$ *specifies a state that for a certain input* a *has a transition to a new state represented by* E. *For example, the expression* $\mu x.a(r\langle x\rangle)$ *states that for input* a, *the machine will perform a "a-loop" transition, whereas* $a(r\langle\underline{\emptyset}\rangle)$ *states that for input a there is a transition to the state denoted by* $\underline{\emptyset}$. *It is interesting to note that a state will only be fully specified in what concerns transitions and output (for a given input a if both* $a(l\langle E_1\rangle)$ *and* $a(r\langle E\rangle)$ *appear in the expression (combined by* $\oplus$*). In the case only transition (resp. output) are specified, the underspecification is solved by setting the target state (resp. output) to* $\underline{\emptyset}$ *(resp.* $\bot_B = 0$*).*

Next, to provide the reader with intuition, we will explain how one can reason on the bisimilarity of two simple expressions, by constructing bisimulation relations. Later on, we show how CIRC can be used in conjunction with our tool in order to act as a decision procedure when checking equivalence of two expressions, in a fully automated manner.

We will start with the expressions $\varepsilon_1 = \mu x.a(r\langle x\rangle)$ and $\varepsilon_2 = \underline{\emptyset}$.We have to build a bisimulation relation $\mathcal{R}$ on $\mathcal{G}$-expressions, such that $(\varepsilon_1, \varepsilon_2) \in \mathcal{R}$. We

do this in the following way: we start by taking $\mathcal{R} = \{(\varepsilon_1, \varepsilon_2)\}$ and we check whether this is already a bisimulation, by considering the output values and transitions and check whether no new expressions appear in this process. If new pairs of expressions appear we add them to $\mathcal{R}$ and repeat the process. Intuitively, this can be represented as follows:

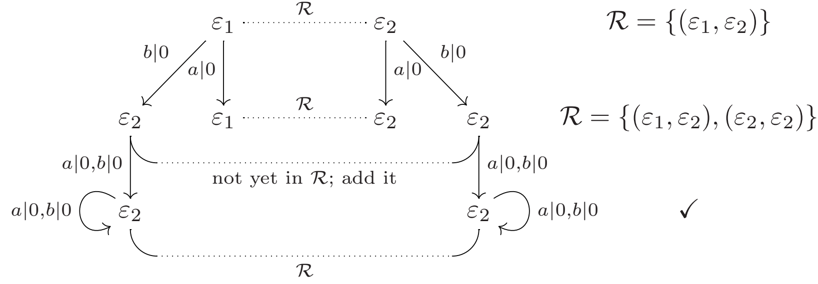
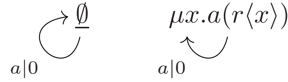


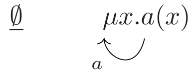
Figure 4: Bisimulation construction

In the figure above, and as before, we use the notation $\varepsilon_1 \overset{\mathcal{R}}{\text{——}} \varepsilon_2$ to denote $(\varepsilon_1, \varepsilon_2) \in \mathcal{R}$. As illustrated in Figure 4, $\mathcal{R} = \{(\varepsilon_1, \varepsilon_2), (\varepsilon_2, \varepsilon_2)\}$ is closed under transitions and is therefore a bisimulation. Hence, $\varepsilon_1 \sim_{\mathcal{G}} \varepsilon_2$.

The proved equality $\underline{\emptyset} = \mu x.a(r\langle x \rangle)$ might seem unexpected, if the reader is familiar with labelled transition systems. The equality is sound because these are expressions specifying behavior of a Mealy machine and, semantically, both denote the function that for every non-emtpy word outputs 0 (the semantics of Mealy machines is given by functions B^{A^+}, intuitively one can think of these expressions as both denoting the empty language). This is visible if one draws the automata corresponding to both expressions (say, for simplicity, the alphabet is $A = \{a\}$):



Note that (i) the $\underline{\emptyset}$ expression for Mealy machines is mapped with δ to a function that for input a gives $\langle 0, \underline{\emptyset} \rangle$, which represents a state with an a-loop to itself and output 0; (ii) the second expression specifies explicitly an a-loop to itself and it also has output 0, since no output value is explicitly defined. Now, also note that similar expressions for labelled transition systems (LTS), or coalgebras of the functor $\mathcal{P}_\omega(-)^A$, would not be bisimilar since one would have an a-transition and the other one not. This is because the $\underline{\emptyset}$ expression for LTS really denotes a deadlock state. In operational terms they would be converted to the systems



which now have an obvious difference in behavior.

By performing a similar reasoning as in the example above one can show that the expressions $\varepsilon_1 = \mu x.a(r\langle x\rangle) \oplus b(r\langle x\rangle)$ and $\varepsilon_2 = \mu x.a(r\langle x\rangle)$ are bisimilar, and the bisimulation relation is built as illustrated in Figure 5:

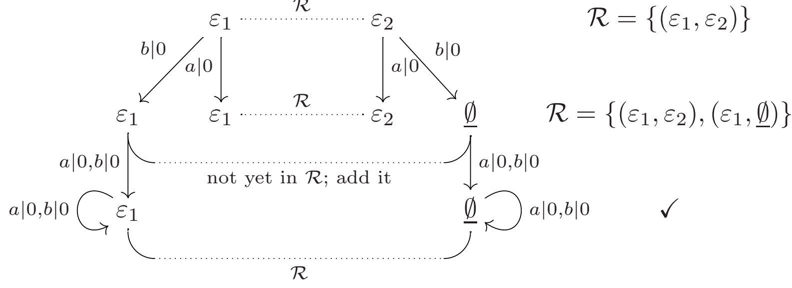


Figure 5: Bisimulation construction

Let us further consider the Mealy machine depicted in Figure 6, where all states are bisimilar.

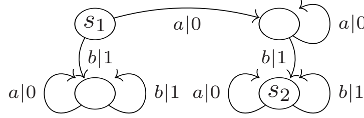


Figure 6: Mealy machine: $s_1 \sim s_2$

We show how to check the equivalence of two expression characterizing the states s_1 and s_2, in a fully automated manner, using CIRC. These expressions are $\varepsilon_1 = \mu x.b(l\langle 1\rangle) \oplus b(r\langle \varepsilon_2\rangle) \oplus a(\mu y.a(r\langle y\rangle) \oplus b(r\langle \varepsilon_2\rangle) \oplus b(l\langle 1\rangle))$ and $\varepsilon_2 = \mu x.b(l\langle 1\rangle) \oplus b(r\langle x\rangle) \oplus a(r\langle x\rangle)$, respectively.

In order to check bisimilarity of ε_1 and ε_2 we load the tool and define the semilattice $\mathsf{B} = \{0, 1\}$ and the alphabet $A = \{a, b\}$:

```
(jslt B is 0 1 bottom 0 .  0 v 0 = 0 .  0 v 1 = 1 .  1 v 1 = 1 .  endjslt)
(alph A is a b endalph)
```

We provide the functor $\mathcal{G}$ using the command `(functor (B x Id)^A .)`. The command `(set goal ... .)` specifies the goal we want to prove:

```
(set goal
 \mu X:FixpVar . b(l<1>) (+) a(l<0>) (+) b(r<X:FixpVar>) (+)
                 a(r<X:FixpVar>) =
 \mu X:FixpVar . b(l<1>) (+) b(<\mu X:FixpVar . b(l<1>) (+)
                 b(r<X:FixpVar>) (+) a(r<X:FixpVar>)>) (+)
                 a(\mu Y:FixpVar . a(r<Y:FixpVar>) (+)
                 b(<\mu X:FixpVar . b(l<1>) (+) a(l<0>) (+)
                 b(r<X:FixpVar>) (+) a(r<X:FixpVar>)>) (+) b(l<1>)) .)
```

In order to generate the CIRC specification we use the command `(generate`

`coalgebra .`). Next we need to load CIRC along with the resulting specification and start the proof engine using the command (`coinduction .`).

As already shown, behind the scenes, CIRC builds a bisimulation relation that includes the initial goal. The proof succeeds and the output consists of (a subset of) this bisimulation:

```
Proof succeeded.
  Number of derived goals: 2
  Number of proving steps performed: 50
  Maximum number of proving steps is set to: 256

Proved properties:
- phi (+) (\mu X . a(l<0>) (+) a(r<X>) (+) b(l<1>) (+) b(r<X>)) =
  phi (+) (\mu Y . a(r<Y>) (+) b(l<1>) (+)
  b(r<\mu X . a(l<0>) (+) a(r<X>) (+) b(l<1>)(+)b(r<X>)>))

- \mu X . a(l<0>) (+) a(r<X>) (+) b(l<1>) (+) b(r<X>) =
  \mu Z . a(r<\mu Y . a(r<Y>) (+) b(l<1>) (+)
          b(r<\mu X . a(l<0>) (+) a(r<X>) (+) b(l<1>) (+) b(r<X>)>)>) (+)
          b(l<1>) (+) b(r<\mu X . a(l<0>) (+) a(r<X>) (+)
          b(l<1>) (+) b(r<X>)>)
```

For the ease of understanding, here we printed a readable version of the proved properties. In Section 6.1, however, we show that internally each expression is brought to a canonical form by renaming the variables. Moreover, note that in our tool, $\underline{\emptyset}$ is represented by the constant `phi`. All the examples provided in the current section make use of this convention.

As previously mentioned, CIRC is also able to detect when two expressions are not equivalent. Take, for instance, the expressions $\mu x.a(l\langle 0\rangle) \oplus a(r\langle a(l\langle 1\rangle) \oplus a(r\langle x\rangle)\rangle)$ and $a(l\langle 0\rangle) \oplus a(r\langle a(r\langle \mu x.a(r\langle x\rangle) \oplus a(l\langle 0\rangle)\rangle) \oplus a(l\langle 1\rangle)\rangle)$, characterizing the states s_1 and s_3 from the Mealy machines in Fig. 7. After following some steps similar to the ones previously enumerated, the proof fails and the output message is `Visible goal [...]  failed during coinduction`.

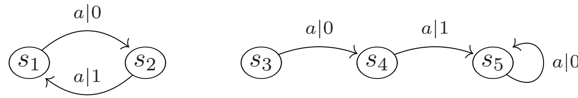


Figure 7: Mealy machines: $s_1 \not\sim s_3$

Example 4. *Let us show how one may check strong bisimilarity of two nondeterministic processes of a non-trivial CCS-like language with termination, deadlock, and divergence, as studied in [1]. A process is a guarded, closed term defined by the following grammar:*

$$P \quad ::= \quad \checkmark \mid \delta \mid \Omega \mid a.P \mid P + P \mid x \mid \mu x.P \tag{20}$$

where:

- $\checkmark$ *is the constant for successful termination,*
- δ *denotes deadlock,*
- Ω *is the divergent computation (*i.e., *the undefined process),*
- $a.P$ *is the process executing the action a and then continuing as the process P, for any action a from a given set A,*
- $P_1 + P_2$ *is the non-deterministic process behaving as either P_1 or P_2, and*
- $\mu x.P$ *is the recursive process $P[\mu x.P/x]$.*

In [23] is is shown that, up to strong bisimilarity, the above syntax of processes is equivalent to the canonical set of (guarded, closed) regular expressions derived for the functor $1 \oplus \mathcal{P}_\omega(\mathsf{Id})^A$,

$$\begin{array}{lcl} E & ::= & \emptyset \mid E \oplus E \mid x \mid \mu x.E \mid l[E_1] \mid r[E_2] \\ E_1 & ::= & \emptyset \mid E_1 \oplus E_1 \mid 1 \\ E_2 & ::= & \emptyset \mid E_2 \oplus E_2 \mid a(E_3) \\ E_3 & ::= & \emptyset \mid E_3 \oplus E_3 \mid \{E\} \end{array}$$

The translation map $(-)^\dagger$ from processes to expressions is defined by induction on the structure of the process:

$$\begin{array}{lcllcl} (\checkmark)^\dagger & = & l[1] & (a.P)^\dagger & = & r[a(\{P^\dagger\})] \\ (\delta)^\dagger & = & r[\emptyset] & (P_1+P_2)^\dagger & = & (P_1)^\dagger \oplus (P_2)^\dagger \\ (\Omega)^\dagger & = & \emptyset & (\mu x.P)^\dagger & = & \mu x.P^\dagger \\ x^\dagger & = & x\,. & & & \end{array}$$

Consider now two processes P and Q over the alphabet $A = \{a, b\}$:

$$\begin{array}{lcl} P & = & \mu x.(a.x + a.P_1 + b.b.\checkmark + b.(\delta + \Omega)) \\ Q & = & \mu z.(a.z + b.(\delta + b.\checkmark) + b.\delta) \end{array}$$

where $P_1 = \mu y.(a.(y + \delta) + b.\delta + b.(\delta + b.\checkmark) + \delta)$. Graphically, the two processes can be represented by the following labelled transition systems (for simplicity we omit annotating states with information regarding the satisfiability of successful termination, divergence, and deadlock):

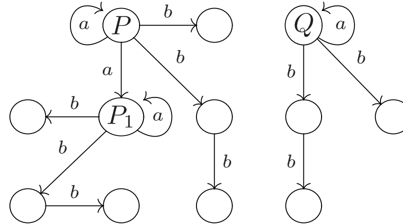


Figure 8: Nondeterministic processes: $Q \sim P$

We want to check if the process P is strongly bisimilar to the process Q. By using the above translation, process P is represented by the expression

$$\mu x.(r[a(\{\mu y.(r[a(\{y \oplus r[\emptyset]\})] \oplus r[b(\{r[\emptyset]\})] \oplus r[b(\{r[\emptyset] \oplus r[b(\{l[1]\})]\})] \oplus r[\emptyset])\})] \oplus r[a(\{x\})] \oplus r[b(\{r[b(\{l[1]\})]\})] \oplus r[b(\{r[\emptyset] \oplus \emptyset\})])$$

whereas process Q is represented by the expression

$$\mu z.(r[a(\{z\})] \oplus r[b(\{r[\emptyset] \oplus r[b(\{l[1]\})]\})] \oplus r[b(\{r[\emptyset]\})]).$$

In order to use the tool, one needs to specify the semilattice, the alphabet, the functor, and the goal in a manner similar to the one previously presented:

```
(jslt B is 1 bottom 1 .  1 v 1 = 1 .  endjslt)
(alph A is a b endalph)
(functor B + (P Id)^A .)

(set goal \mu X:FixpVar .
          r[ a( { X:FixpVar } ) ] (+)
          r[ a( { \mu Y:FixpVar .
                  r[ a( { Y:FixpVar (+) r[ phi ] } ) ] (+)
                  r[ b( { r[ phi ] } ) ] (+)
                  r[ b( { r[ phi ] (+) r[ b( { l[ 1 ] } ) ] } ) ] (+)
                  r[ phi ]
              } )
          ] (+)
          r[ b( { r[ b( { l[ 1 ] } ) ] } ) ] (+)
          r[ b( { r[ phi ] (+) phi } ) ]
          =
          \mu Z:FixpVar .
          r[ a( { Z:FixpVar } ) ] (+)
          r[ b( { r[ phi ] (+) r[ b( { l[ 1 ] } ) ] } ) ] (+)
          r[ b( { r[ phi ] } ) ]  .)
```

For the generated specification CIRC terminates and outputs a positive result:

```
Proof succeeded.
  Number of derived goals: 15
  Number of proving steps performed: 58
  Maximum number of proving steps is set to: 256

Proved properties:
- r[phi] (+) (\mu Y. r[phi] (+) r[a({r[phi] (+) Y})] (+) r[b({r[phi]})]
  (+) r[b({r[phi] (+) r[b({l[1]})]})])
  =
  \mu Z. r[a({Z})] (+) r[b({r[phi]})] (+) r[b({r[phi] (+) r[b({l[1]})]})]
- r[b({l[1]})] = r[phi] (+) r[b({l[1]})]
- \mu Y. r[phi] (+) r[a({r[phi] (+) Y})] (+) r[b({r[phi]})] (+)
  r[b({r[phi] (+) r[b({l[1]})]})]
  =
  \mu Z. r[a({Z})] (+) r[b({r[phi]})] (+) r[b({r[phi] (+) r[b({l[1]})]})]
- \mu X. r[a({X})] (+) r[a({\mu Y. r[phi] (+) r[a({r[phi] (+) Y})] (+)
  r[b({r[phi]})] (+) r[b({r[phi] (+) r[b({l[1]})]})]})] (+)
```

```
r[b({r[phi] + phi})] (+) r[b({r[b({l[1]})]})]
=
\mu Z. r[a({Z})] (+) r[b({r[phi]})] (+) r[b({r[phi] (+) r[b({l[1]})]})]
```

6.1. Implementation

In this section we present details on the implementation of the algebraic specification given in Section 4, based on the examples from Section 6.

In order to generate the algebraic specifications for CIRC when provided a functor and two expressions we used the Maude system [4]. We choose it for its suitability for performing equational and rewriting logic based computations, and because of its reflective properties allowing for the development of advanced metalanguage applications. As the technical aspects on how to work at the meta-level are beyond the scope of this paper, we refrain from presenting them and show, instead, what the generated specifications consist of.

Most of the algebraic specifications from Section 4 have a straightforward implementation in Maude. Consider, for instance, the case of Mealy machines presented in Example 3. The generated grammars for functors (1) and expressions (Definition 3) are coded as:

```
sort Functor .
sorts AlphName SltName .
subsort SltName < Functor .

op A : -> AlphName .
op B : -> SltName .
op G : -> Functor .
op Id : -> Functor .
op _+_ : Functor Functor -> Functor .
op _^_ : Functor AlphName -> Functor .
op _x_ : Functor Functor -> Functor .
```

```
sorts Exp ExpStruct Alph Slt .
subsort Exp < ExpStruct .
enum A is a b . enum B is 0 1 .
subsort A < Alph .
subsort B < Slt .

op _`(+`)_ : Exp Exp -> Exp .
op _`(_`) : Alph Exp -> Exp .
op \mu_._ : FixpVar Exp -> Exp .
ops l<_> r<_> : Exp -> Exp .
op phi : -> Exp .
```

```
eq G = (B x Id) ^ A .
```

Most of the syntactical constructs are Maude-specific: `sorts` and `subsort` declare the sorts we work with and, respectively, the relations between them; `op` declares operators; `eq` declares equations (the equation in our case defines the shape of the functor `G`). The only CIRC-specific construct, `enum`, is syntactic sugar for declaring enumerable sorts, *i.e.*, sorts that consist only of the specified constants. As a side note, if brackets (`(`, `[`, `{`) are used in the declaration of an operation, then they must be preceded by a backquote (`` ` ``).

As mentioned in Section 2, in order to guarantee the finiteness of our procedure, one needs to include the ACI axioms for `(+)`. Moreover, we have observed that the unity axiom for `(+)` plays an important role in decreasing the number of states generated by the repeated application of $\delta_{\mathcal{G}}$, therefore improving the overall time performance of the tool. For example, the number of rewritings CIRC performed in order to prove the bisimilarity of ε_1 and ε_2 in Figure 5 was halved when the unity axiom was used.

By turning on the axiomatization flag using the command `(axioms on .)`, the following code is generated:

```
op _`(+`)_ : Exp Exp -> Exp [assoc comm] .
eq E:Exp (+) E:Exp = E:Exp .
eq E:Exp (+) phi = E:Exp .
```

It is an obvious question why not to add other axioms to the tool, since the unity axiom has improved performance. At this stage we do not have studied in detail how much adding other axioms would help. It is in any case a trade-off on how many extra axioms one should include, which will get the automaton produced from an expression closer to the minimal automaton, and how much time the tool will take to reduce the expressions in each step modulo the axioms. For classical regular expressions, there is an interesting empirical study on this [16]. We leave it as future work to carry on a similar study for our expressions and axioms.

The process of substituting fixed-point variables has a natural implementation. We present the equations handling the basic expressions $\underline{\emptyset}$ and x, and the operation `(+)`:

```
op _`[_/_`] : Exp Exp FixpVar -> Exp .
eq phi [ E:Exp / X:FixpVar ] = phi .
ceq Y:FixpVar [ E:Exp / X:FixpVar ] = E:Exp if (X:FixpVar == Y:FixpVar) .
eq Y:FixpVar [ E:Exp / X:FixpVar ] = Y:FixpVar [owise] .
eq (E1:Exp (+) E2:Exp) [ E:Exp / X:FixpVar ] =
   (E1:Exp [E:Exp / X:FixpVar]) (+) (E2:Exp [E:Exp / X:FixpVar]) .
```

In order to avoid matching problems and to overpass the fact that in Maude one cannot handle an equation that has fresh variables in its right-hand-side (*i.e.*, they do not appear in the left-hand-side), we replace expression variables with parameterized constants: `op var :  Nat -> FixpVar .` The operation that obtains this canonical form has an inductive definition on the structure of the given expression and makes use of the substitution operation presented above. For this reason, the bisimulation CIRC builds contains parameterized constants instead of the user declared variables. The property proved in Example 4 is, therefore, written as:

```
\mu var(2) . r[a({var(2)})] (+) r[a({\mu var(1) . r[phi] (+)
r[a({r[phi] (+) var(1)})] (+) r[b({r[phi]})] (+) r[b({r[phi] (+)
r[b({l[1]})]})]})] (+) r[b({r[phi] (+) phi})] (+) r[b({r[b({l[1]})]})]
=
\mu var(1) . r[a({var(1)})] (+) r[b({r[phi]})] (+)
r[b({r[phi] (+) r[b({l[1]})]})]
```

The most important part of the algebraic specification consists of the equations defining the operations $\delta_{_}(_)$, $Plus_{_}(_,_)$, and *Empty*. Most of these equations are implemented as presented in [23]. The only difficulties we encountered were for the exponentiation case, as Maude does not handle higher-order functions. Without entering into details, as a workaround, we introduced a new sort `Function < ExpStruct` and an operation `\. : ExpoCase Alph Functor ExpStruct -> Function` in order to emulate function-passing. The first argument is used to memorize the origin where the exponentiation ingredient is encountered: δ, *Plus*, or *Empty*. Its purpose is purely technical – we use it in order to

avoid some internal matching problems. The other three parameters are those of the structured expression $\lambda.(a, \mathcal{F} \triangleleft \mathcal{G}, \sigma)$ presented in Section 4: a letter in the alphabet, an ingredient, and some other structured expression.

Another thing worth describing is the way we enable CIRC to prove equivalences when the powerset functor occurs. Namely, we present how interpolant (15) is implemented. Recall that we want to show that two sets of expressions are equivalent, which means that for each expression in the first set there must be an equivalent one in the second set and vice-versa.

In order to handle sets of structured expressions we introduce a new sort, `ExpStructSet` as a supersort for `ExpStruct`. We also consider the set separator `_,_ : ExpStructSet ExpStructSet -> ExpStructSet [assoc,comm]`, the empty set `emptyS : -> ExpStructSet`, and the set wrapping operation `{_} : ExpStructSet -> ExpStruct`. In order to mimic universal quantification over a set, we use a special constant referred to as token "`[/]`". In what follows, we consider two variables of sort `ExpStructSet`: `ES` and `ES'`, and two variables of sort `ExpStructSet`: `ESS` and `ESS'`. We now describe the process of finding the equivalence between two sets:

- whenever encountering two wrapped expression sets we add the universal quantification token to each of them in two distinct goals:

```
srl {ESS} = {ESS'} => {[/] ESS} = {ESS'} /\ {ESS} = {[/] ESS'} .
```

- iterate through the expressions on the left-hand-side (similarly for the other direction):

```
srl {[/] (ES , ESS)} = {ESS'} =>
    {[/] ES} = {ESS'} /\ {[/] ESS} = {ESS'} .
srl {ESS} = {[/] (ES' , ESS')} =>
    {ESS} = {[/] ES'} /\ {ESS} = {[/] ESS'} .
```

- when left with one expression on the left-hand-side, start iterating through the expressions on the right-hand-side until finding an equivalence (similarly for the other direction):

```
srl {[/] ES} = {ES' , ESS'} => ES = ES' \/ {[/] ES} = {ESS'} .
srl {ES , ESS} = {[/] ES'}  => ES = ES' \/ {ESS} = {[/] ES'} .
```

- if no equivalence has been found, transform the current goal into a visible failure:

```
srl {ESS} = emptyS => true = false .
srl emptyS = {ESS} => true = false .
```

Finally, the type checker for structured expressions has a straightforward implementation. Its code does not appear in the generated specification as it is only used when the tool receives the expressions as input. This prevents obtaining the specification and starting the prover in case invalid expressions are provided.

7. Discussion

One of the major contributions of this paper is that we provided a decision procedure for the bisimilarity of generalized regular expressions. In order to enable the implementation of the decision procedure, we have exploited an encoding of coalgebra into algebra, and we formalized the equivalence between the coalgebraic concepts associated to non-deterministic coalgebras [23] and their algebraic correspondents. This led to the definition of algebraic specifications ($\mathcal{E}_{\mathcal{G}}$) that model both the language and the coalgebraic structure of expressions. Moreover, we defined an equational deduction relation ($\vdash_{NDF}$), used on the algebraic side for reasoning on the bisimilarity of expressions.

The most important result of the parallel between the coalgebraic and algebraic approaches is given in Corollary 1, which formalizes the definition of the bisimulation relations in algebraic terms. Actually, this result is the key for proving the soundness of the decision procedure implemented in the automated prover CIRC [14]. As a coinductive prover, CIRC builds a relation $\mathcal{F}$ closed under the application of $\delta_{\mathcal{G}}$ with respect to $\vdash_{NDF}$ ($\mathcal{E}_{\mathcal{G}} \cup \boxed{\mathcal{F}} \vdash_{NDF} \boxed{\delta_{\mathcal{G}}(\mathcal{F})}$), hence automatically computing a bisimulation the initial proof obligations belong to.

The approach we present in this paper enables CIRC to perform reasoning based on bisimulations (instead of experiments [17]). This way, the prover is extended to checking bisimilarity in a large class of systems that can be modeled as non-deterministic coalgebras. Note that the constructions above are all automated – the (non-trivial) CIRC algebraic specification describing $\mathcal{E}_{\mathcal{G}}$, together with the interpolants implementing $\vdash_{NDF}$ are generated with the Maude tool presented in Section 6.

We now mention some of the existing coalgebraic based tools for proving bisimilarity and the main differences with the tool presented in this paper. CoCasl [8] and CCSL [19] are tools that can generate proof obligations for theorem provers from coalgebraic specifications. In [8] several tactics for interactive and automatic bisimulation building are implemented in Isabelle/HOL and are used to derive bisimilarities for translated specifications from CoCasl. The main difference between our tool and CoCasl or CCSL is that, given a functor, the tool derives a specification language for which equivalence is decidable (that is, it is automatic and not interactive). CIRC [5, 17], on top of which the current tool is built, is based on hidden logic [18] and uses a partial decision procedure for proving bisimilarities via implicit construction of bisimulations. Our tool can be seen as an extension of CIRC to a fully automatic theorem prover for the class of non-deterministic coalgebras. We stress the fact that the focus of this paper was on a language for which equivalence is decidable. Tools such as CoCasl, CCSL or CIRC have a more expressive language, where one can, for instance, specify streams which in our language could not be specified (intuitively, the streams we can specify in our language are eventually periodic). In those tools decidability of equivalence can however not be guaranteed.

There are several directions for future work.

Extending the class of systems to include quantitative coalgebras (such as weighted automata and Markov chains) will enlarge the scope of applicability of

the tool. The challenge in this extension arises from the fact that the definition of expressions for quantitative coalgebras involving the distribution monad is not as modular as for the other functors (for details see [22]). This is a consequence of the fact that the sum of two valid expressions might not be a valid expression anymore (since in distributions we require that the sum of probabilities add up to 1). Moreover, calculating bisimulation relations in the quantitative setting will encompass metric manipulation, which is currently not implemented in CIRC.

To improve usability, building a graphical interface for the tool is an obvious next step. The graphical interface should ideally allow the specification of expressions by means of systems of equations (which are then solved internally) or even by means of an automaton, which would then be translated to an expression using Kleene's theorem. We also would like to explore how adding more axioms than ACI to the prover (that is, each step of the bisimulation checking is performed modulo more equations) improves the performance. Our experience so far shows that by adding the axiom for the distribution of the $\underline{\emptyset}$ expression through the constructors, *i.e.* $\underline{\emptyset} \oplus \varepsilon = \varepsilon$, the prover works significantly faster.

We have not yet studied complexity bounds for the algorithms presented in this paper. We conjecture however that the bounds will be very similar to the already known for classical regular expressions [13, 25]. Further explorations in this direction are left as future work.

Acknowledgments. We would like to thank the referees for the many constructive comments, which greatly helped us to improve the paper. The authors are also grateful for useful comments from Luca Aceto, Filippo Bonchi, and Miguel Palomino Tarjuelo. The work of Georgiana Caltais and Eugen-Ioan Goriac has been partially supported by the project 'Meta-theory of Algebraic Process Theories' (nr. 100014021) of the Icelandic Research Fund. The work of Eugen-Ioan Goriac has also been partially supported by the project 'Extending and Axiomatizing Structural Operational Semantics: Theory and Tools' (nr. 110294-0061) of the Icelandic Research Fund. The work of Dorel Lucanu has been partially supported by the PNII grant DAK project Contract 161/15.06.2010, SMIS-CSNR 602-12516. The work of Alexandra Silva was partially funded by ERDF - European Regional Development Fund through the COMPETE Programme and by Fundao para a Cincia e a Tecnologia, Portugal within projects `FCOMP-01-0124-FEDER-020537` and `SFRH/BPD/71956/2010`.

References

References

[1] L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *J. ACM*, 39:147–187, January 1992.

[2] M. Bonsangue, G. Caltais, E.-I. Goriac, D. Lucanu, J. Rutten, and A. Silva. A decision procedure for bisimilarity of generalized regular expressions. In

Proceedings of the 13th Brazilian conference on Formal methods: foundations and applications, SBMF'10, pages 226–241, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, 2000.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All about Maude - a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, Berlin, Heidelberg, 2007.

[5] J. Goguen, K. Lin, and G. Rosu. Circular coinductive rewriting. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, pages 123–132, Washington, DC, USA, 2000. IEEE Computer Society.

[6] J. A. Goguen. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[7] E.-I. Goriac, D. Lucanu, and G. Roşu. Automating coinduction with case analysis. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 220–236, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] D. Hausmann, T. Mossakowski, and L. Schröder. Iterative Circular Coinduction for CoCasl in Isabelle/HOL. In M. Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2005.

[9] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[10] S. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–42, 1956.

[11] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *LICS*, pages 214–225. IEEE Computer Society, 1991.

[12] D. Kozen. Myhill-Nerode relations on automatic systems and the completeness of Kleene algebra. In A. Ferreira and H. Reichel, editors, *STACS*, volume 2010 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2001.

[13] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.

[14] D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu. CIRC: a behavioral verification tool based on circular coinduction. In *Proceedings of the 3rd international conference on Algebra and coalgebra in computer science*, CALCO'09, pages 433–442, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] R. Milner. A complete inference system for a class of regular behaviours. *J. Comput. System Sci.*, 28(3):439–466, 1984.

[16] S. Owens, J. H. Reppy, and A. Turon. Regular-expression derivatives reexamined. *J. Funct. Program.*, 19(2):173–190, 2009.

[17] G. Roşu and D. Lucanu. Circular coinduction: a proof theoretical foundation. In *Proceedings of the 3rd international conference on Algebra and coalgebra in computer science*, CALCO'09, pages 127–144, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] G. Rosu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.

[19] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. *J. UCS*, 7(2):175–193, 2001.

[20] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[21] A. Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[22] A. Silva, F. Bonchi, M. Bonsangue, and J. Rutten. Quantitative Kleene coalgebras. *Information and Computation*, 209(5):822–849, 2011.

[23] A. Silva, M. M. Bonsangue, and J. J. M. M. Rutten. Non-deterministic Kleene coalgebras. *Logical Methods in Computer Science*, 6(3), 2010.

[24] S. Staton. Relating coalgebraic notions of bisimulation: with applications to name-passing process calculi. In *Proceedings of the 3rd international conference on Algebra and coalgebra in computer science*, CALCO'09, pages 191–205, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] J. Worthington. Automatic proof generation in Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 4988 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.