Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Regular expressions for polynomial coalgebras

M.M. Bonsangue, J.J.M.M. Rutten, A.M. Silva

REPORT SEN-E0703 DECEMBER 2007

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Regular expressions for polynomial coalgebras

ABSTRACT
For polynomial set functors *G*, we introduce a language of expressions for describing elements of final *G*-coalgebra. We show that every state of a finite *G*-coalgebra corresponds to an expression in the language, in the sense that they both have the same semantics. Conversely, we give a compositional synthesis algorithm which transforms every expression into a finite *G*-coalgebra. The language of expressions is equipped with an equational system that is sound, complete and expressive with respect to *G*-bisimulation.

2000 Mathematics Subject Classification: 03B70
1998 ACM Computing Classification System: F.4.3, F.4.1
Keywords and Phrases: Coalgebra, Regular expressions, Logics, Synthesis

Regular expressions for polynomial coalgebras

M.M. Bonsangue
LIACS

Jan Rutten*
CWI and VUA

Alexandra Silva†
CWI

December 17, 2007

Abstract

For polynomial set functors G, we introduce a language of expressions for describing elements of final G-coalgebra. We show that every state of a finite G-coalgebra corresponds to an expression in the language, in the sense that they both have the same semantics. Conversely, we give a compositional synthesis algorithm which transforms every expression into a finite G-coalgebra. The language of expressions is equipped with an equational system that is sound, complete and expressive with respect to G-bisimulation.

1 Introduction

Regular expressions are a syntactic means to describe exactly the class of languages accepted by deterministic finite state automata, i.e. the class of regular languages. Ignoring the initial state for the moment, a deterministic automata is a set of states equipped with function determining for each state whether or not it is final, and assigning for each input symbol a next state.

Deterministic automata can be generalized to coalgebras for an endofunctor G on the category *Set*. A coalgebra is pair (S, g) consisting of a set of states S and a transition function $g : S \to GS$. Here the functor G determines the type of the dynamic system under consideration, and is the parameter of the theory of universal coalgebra [26]. The central concepts in this theory are homomorphism of coalgebras, bisimulation equivalence and final coalgebra.

For example, for a given finite set A of input symbols and taking $2 = \{0, 1\}$, finite deterministic automata correspond to coalgebras over a finite set for the functor $2 \times Id^A$. Coalgebra homomorphisms are the usual automata homomorphism, bisimulation equivalence is language equivalence and the final coalgebra is the set of all languages over A.

As usual, collections of coalgebras can be described by means of specification languages, i.e. an axiomatic description of a particular class of mathematical structures. In this paper we propose a simple but adequate and expressive language of regular expressions for the specification of coalgebras for polynomial endofunctors on *Set*. Here adequate means that the logical equivalence corresponds to the bisimulation equivalence on coalgebras, whereas expressive means that every finite coalgebra can be represented by a finite regular expression. Finally, simple means that the the set of regular expressions contains precisely what is needed to obtain this goal, and nothing more. The latter point is an important distinguishing factor in comparison with some already existing specification formalisms for coalgebra in the literature, discussed below.

Our approach is coalgebraic. The set of of regular expressions is parametrized by the type G of the dynamical system under consideration. We apply general insights from so-called coalgebraic modal logic (see e.g. [20, 2]) and define a language whose basic operations are derived directly from G. The equivalence induced by the regular expressions coincides with the notion of equivalence (G-bisimulation) induced by the functor G. Further, we give an equational system for reasoning about equivalence of regular expressions, that we prove sound and complete for the fixed point-free fragment.

*Partially supported by EU project IST-33826 CREDO (`http://credo.cwi.nl`).

†Partially supported by the Fundação para a Ciência e a Tecnologia, Portugal, under grant number SFRH/BD/27482/2006.

The main contributions of the paper can be summarized as follows:

(i) We present in Section 3a language of regular expressions for transition systems typed by polynomial functors, analogous to Kleene's regular expressions for finite deterministic automata. This language is equipped with a sound and complete (for the fixed-points free fragment) axiomatization in Section 3.3.

(ii) We show in Section 3.2 that, for every polynomial functor, every finite coalgebra can be specified by a regular expression.

(iii) Conversely, we present in Section 4 a synthesis algorithm that allows the construction of a finite coalgebra for every regular expression, whose behaviour is exactly characterized by that expression. We prove the termination of the algorithm in Section 4.4.

As running examples we use Moore automata [19], partial automata [14] and deterministic labelled infinite top-down binary tree automata [23]. Together, they cover the full spectrum of constructs for polynomial functors.

1.1 Related Work

Coalgebras of polynomial functors and the study of their (equational) logic has been investigated by many authors [8, 24, 10]. Our approach is similar in spirit to that of Goldblatt [8] in that we use the ingredients of a functor for typing expressions, and differs from the other two works because we do not need an explicit "next-state" operator, as we can deduce it from the type information.

In the last few years several proposals of specification languages for coalgebras appeared [20, 24, 10, 6, 1, 2, 18]. Apart from [18], the logics presented do not include fixpoint operators. Our language of regular expressions can be seen as an extension with fixed point operators of the coalgebraic logic of [1] and it is similar to a fragment of the logic presented in [18]. However, our goal is rather different: we want a *finitary* language that characterize exactly all *finite* coalgebras. Our language is minimal, as it contains only the operators necessary for this goal. This restriction does not decrease the expressiveness of the language with respect to bisimulation and it allows for a simple and direct algorithm for the synthesis of a finite coalgebra.

Automata synthesis is a popular and very active research area [21, 29, 16, 7, 9]. Most of the work done on synthesis has as main goal to find a proper and sufficiently expressive type of automata to encode a specific type of logic (such as LTL [29] or μ-calculus [16]). Recently, this automata theoretical approach has been generalized in [18] to coalgebras for endofunctors F over **Set**: the synthesis of a F-coalgebra from a formula passes by the construction of an alternating parity automata accepting F-coalgebra and then checking for non-emptiness. On this respect, our approach is a novel and fully coalgebraic synthesis method: the set of expressions is a coalgebra, and as such the synthesis of an expression is just a sub-coalgebra up to the application of some equations to guarantee finiteness.

Regular expressions have been originally introduced by Kleene [12] as a mathematical notation for describing languages recognized by deterministic finite automata. In [25], deterministic automata, the sets of formal languages and regular expressions are all presented as coalgebras of the functor $2 \times Id^A$ (where A is the alphabet, and 2 is the two elements set). It is then shown that the standard semantics of language acceptance of automata and the assignment of languages to regular expressions both arise as the unique homomorphism into the final coalgebra of formal languages. The coalgebra structure on the set of regular expressions is determined by their so-called *Brzozowski* derivatives [5]. In the present paper, the set of expressions for the functor $F(S) = 2 \times S^A$ differs from the classical definition in that we do not have Kleene star and full concatenation (sequential composition) but, instead, the least fixed point operator and action prefixing. Modulo that difference, the definition of a coalgebra structure on the set of expressions in both [25] and the present paper is essentially the same. All in all, one can therefore say that standard regular expressions and their treatment in [25] can be viewed as a special instance of the present approach.

Our equational axiomatization of regular expressions for coalgebra is implicational but finite. A finite implicational equational system for ordinary regular expressions has been developed by, e.g., Salomaa [27] and Kozen [15], as a semiring equipped with a least fixed point operator. Since we do not have full concatenation but only action prefix, our axiomatization is basically based on a idempotent commutative monoid (join semilattice) equipped with a unary function

for each element of the input alphabet that distributes over the monoid operation. As for ordinary regular expression, we have proof rules for a least fixed point operator.

The present paper can be seen as a generalization of [4], where a sound and complete specification language and a synthesis algorithm for Mealy machines is given. Mealy machines are coalgebras of the functor $(B \times Id)^A$, where A is a finite input alphabet and B a finite meet semilattice for the output alphabet.

2 Preliminaries

We give the basic definitions on polynomial functors, coalgebras and introduce the notions of simulation and bisimulation.

First we fix notation on sets and operations on them. Let **Set** be the category of sets and functions. Sets are denoted by capital letters $X, Y, \ldots$ and functions by lower case $f, g, \ldots$. The collection of functions from a set X to a set Y is denoted by Y^X. Given functions $f : X \to Y$ and $g : Y \to Z$ we write their composition as $g \circ f$. The product of two sets X, Y is written as $X \times Y$, with projection functions

$$X \xleftarrow{\pi_1} X \times Y \xrightarrow{\pi_2} Y$$

The set 1 is a singleton set typically written as $1 = \{*\}$. It can be regarded as the empty product. Coproducts (disjoint unions) of two set X and Y is denoted by $X + Y$, with injection functions

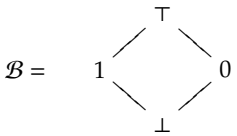
$$X \xrightarrow{\kappa_1} X + Y \xleftarrow{\kappa_2} Y$$

In our definition of polynomial functors we use constant sets equipped with an information order. In particular, we use join-semilattice. A (bounded) join-semilattice is a set B equipped with a binary operation $\vee_B$ and a constant $\bot_B \in B$, such that $\vee_B$ is commutative, associative and idempotent. The element $\bot_B$ is neutral w.r.t. $\vee_B$. As usual, $\vee_B$ gives rise to a partial ordering $\leq_B$ on the elements of B:

$$b_1 \leq_B b_2 \Leftrightarrow b_1 \vee_B b_2 = b_2$$

meaning that b_1 is more *concrete* than b_2. Every set S can be transformed into a join-semilattice by taking B to be the set of all finite subsets of S with union as join.

Most of our examples use the following four element join-semilattice:



Here, the $\top$ element is used for *abstracting* (under-specification) from any concrete data; the $\bot$ element denotes *inconsistency* (over-specification) of information; and the elements 0 and 1 are concrete output values.

We are now ready to define the class of polynomial functors. They are endofunctor of the category of sets, built inductively from the identity and constants, using products, coproducts and exponents.

Definition 1 The class PF of *polynomial functors* on **Set** is inductively defined by putting:

$$G ::= Id \mid B \mid G + G \mid G \times G \mid G^A$$

where B is a finite join-semilattice and A is a set. ☼

Typical examples of polynomial functors are $M = B \times Id^A$, $P = (1 + Id)^A$ and $T = B \times (Id \times Id)$. As we will see below, these represent, respectively, the type of *Moore automata*, *partial deterministic automata* and (infinite) *binary tree automata*. We will use these functors as our running examples.

Our definition of polynomial functors slightly differs from the one used in [24, 10] in the use of a join-semilattice as constant functor. This allow us to have a full coalgebraic treatment

of regular expressions. In fact, we will show that regular expressions (for this class of functors) are coalgebras. This is not a restriction since every set can be lifted to a join-semilattice.

Next, we give the definition of the ingredient relation, which relates a polynomial functor G with its *ingredients*,*i.e.* the functors used in its inductive construction. We use later this relation for typing our expressions.

Definition 2 (Ingredients) A polynomial functor F is said to be an *ingredient* of a polynomial functor G, denoted by $F \lhd G$, if they are part of the least reflexive and transitive relation on polynomial functors such that

$$\begin{gathered} G_1 \lhd G_1 \times G_2 \\ G_2 \lhd G_1 \times G_2 \\ G_1 \lhd G_1 + G_2 \\ G_2 \lhd G_1 + G_2 \\ G \lhd G^A \end{gathered}$$

☼

For example, B, Id, $B \times Id$, and M itself are all the ingredients of the Moore automata functor M.

Recall that for a functor G on **Set**, a G-coalgebra is a pair (S, f) consisting of a set of *states* S together with a function $f : S \to GS$. The functor G, together with the function f, determines the *transition structure* (or dynamics) of the G-coalgebra [26].

Let us now present three concrete examples of polynomial coalgebras – Moore automata, partial automata and binary tree automata:

(i) A Moore automaton (S, g) over a finite set A of input values and outputs in a finite join semilattice B consists of a set S of states together with a transition function

$$g : S \to B \times S^A$$

For a given state $s \in S$ the function f returns a pair consisting of an output value $b \in B$ and a function returning a successor state for each input $a \in A$.

(ii) A partial automaton (S, g) over a finite set A of input values consists of a set S of states together with a transition function

$$g : S \to (1 + S)^A$$

For a given state $s \in S$ and an input $a \in A$, the function f is either undefined (that is, $g(s)(a) \in 1$) or returns a successor state.

(iii) A binary tree automaton (S, g) with B-labelled nodes, consists of a set S of states together with a function

$$g : S \to B \times S \times S$$

For a given state $s \in S$, the function g returns the label of the state s and two successor states, representing the children of the node s.

Observe that Moore automata are M-coalgebras, partial automata are P-coalgebras and infinite binary trees are T-coalgebras (for M, P, and T the polynomial functors defined after Definiton 1). In the sequel, we use this observation to illustrate the concept we introduce for polynomial coalgebras.

Convention 3 We use the following conventions in representing the transition functions of the automata introduced above.

(i) Given a Moore automaton (S, g), we ypically write

$$g(s)(a) = \langle s{\downarrow},\ \lambda a.\, s_a \rangle$$

and call $s{\downarrow}$ the *(initial) output* and s_a the *next state on input* a. We also use the following convention for the representation of Moore automata:

$$g(s)(a) = \langle b, s' \rangle \iff s \xrightarrow{a|b} s'$$

Moore automata where A is the two element set $\{0,1\}$ and B is the meet-semilattice $\mathcal{B}$ are called *binary*, and they are *fully specified* if only 0 or 1 are used as output elements (and never $\bot$ or $\top$). In this case, we call a state final if its output is 1, and non-final otherwise.

(ii) Given a partial automaton (S,t), we write $t(s)(a)\Uparrow$, whenever $t(s)(a) = \kappa_1(*)$. If $t(s)(a) = \kappa_2(s')$ $(s' \in S)$, then we write $t(s)(a)\Downarrow$ and $t(s) = \lambda a.\, s_a$, graphically represented as:

$$t(s)(a) = s' \iff s \xrightarrow{\ a\ } s_a$$

In the graphical representation of partial automata we omit undefined transitions. Partial automata whose input alphabet is the set $A = \{0,1\}$ are called *binary*.

(iii) Given a binary tree automaton $(S, \langle i,l,r\rangle)$ we typically write

$$\langle i,l,r\rangle(s) = \langle s{\downarrow}, s_L, s_R\rangle$$

and call $s{\downarrow}$ the *(initial) output* of s and s_L and s_R the *left* and *right derivatives* of s, respectively. We also use the following convention for the representation of binary tree automata:

$$\langle i,l,r\rangle(s) = \langle b, s', s''\rangle \iff s{\downarrow}b \xrightarrow{\ l\ } s' \quad s{\downarrow}b \xrightarrow{\ r\ } s''$$

When no explicit reference to the transition structure $\langle i,l,r\rangle$ is needed, we often use S instead of $(S,\langle i,l,r\rangle)$.

To give few examples, let S be a two elements set $\{s_1, s_2\}$:

(i) The transition function of a fully specified binary Moore automaton over S, is determined by the following picture:

$$\mathcal{A}_1 = \quad \overset{0}{\circlearrowleft}\ s_1{\downarrow}0 \xrightarrow{\ 1\ } s_2{\downarrow}1 \ \overset{0,1}{\circlearrowright} \qquad . \tag{1}$$

(ii) The following picture defines the transition function of a binary partial automaton over S, in which state s_2 terminates for input 0.

$$\mathcal{A}_2 = \quad \overset{1}{\circlearrowleft}\ s_1 \xrightarrow{\ 0\ } s_2 \ \overset{1}{\circlearrowright} \tag{2}$$

(iii) Finally, for an example of a binary tree automata consider the following transition structure:

$$\mathcal{A}_3 = \quad \overset{l}{\circlearrowleft}\ s_1{\downarrow}1 \xrightarrow{\ r\ } s_2{\downarrow}0 \ \overset{l,r}{\circlearrowright} \tag{3}$$

It accept a tree labelled by 1 on the root node, and with the right and left subtrees all labelled by 0's and 1's, respectively.

Definition 4 (G-homomorphism) A *G-homomorphism* from a G-coalgebra (S,f) to a G-coalgebra (T,g) is a function $h\colon S \to T$ preserving the transition structure, *i.e.*, such that the following diagram commutes.

$$\begin{array}{ccc} S & \xrightarrow{\ h\ } & T \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ GS & \xrightarrow[\ Gh\ ]{} & GT \end{array} \qquad g \circ h = Gh \circ f$$

☼

This general notion of homomorphism instantiates for our running examples to the following definitions.

(i) A homomorphism from a Moore automaton (S, f) to a Moore automaton (T, g) is then a function $h\colon S \to T$ preserving initial outputs and next states:

$$h(s)\downarrow = s\downarrow \quad \text{and} \quad h(s_a) = h(s)_a$$

(ii) A homomorphism from a partial automaton (S, f) to a partial automaton (T, g) is then a function $h\colon S \to T$ preserving termination and next states:

1. $h(s_a) \Uparrow$ if and only if $h(s)_a \Uparrow$
2. If $h(s_a) \Downarrow$ then $h(s_a) = h(s)_a$.

(iii) A homomorphism from a binary tree automata S to a binary tree automata T is a function $h\colon S \to T$ preserving initial output and derivatives:

$$h(s)\downarrow = s\downarrow \quad \text{and} \quad h(\sigma_L) = h(\sigma)_L \quad \text{and} \quad h(\sigma_R) = h(\sigma)_R\,.$$

Next, we define the notion of finality, which play a key role in the soundness and completeness of our system of equations for regular expression.

Definition 5 (Final coalgebra) A G-coalgebra (S, f) is said to be *final* if for any other G-coalgebra (T, g) there exists a unique homomorphism h which makes the following diagram commute:

$$\begin{array}{ccc} T & \overset{h}{\dashrightarrow} & S \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle f} \\ GT & \xrightarrow[Gh]{} & GS \end{array}$$

☼

For every polynomial functor G there exists a final G-coalgebra (Ω_G, ω_G) [26]. In section 3.3 we give an explicit construction of the final coalgebra in terms of sets of regular expressions appropriately typed by polynomial functors.

Next we define the lifting of a relation R with respect to a polynomial functor G. This notion plays a crucial role in the definition of (bi)simulation.

Definition 6 Let G be a polynomial functor, and $R \subseteq S \times T$ a binary relation. We define $\overline{G}(R) \subseteq G(S) \times G(T)$ by induction on the structure of the functor G as follows:

$$\begin{array}{lcl}
\overline{Id}(R) & = & R \\
\overline{B}(R) & = & \{\langle b_1, b_2\rangle \mid b_2 \not\leq_B b_1\ ,\ b_1, b_2 \in B\} \\
\overline{G_1 \times G_2}(R) & = & \{\langle\langle u_1, v_1\rangle, \langle u_2, v_2\rangle\rangle \mid \langle u_1, u_2\rangle \in \overline{G_1}(R) \text{ and } \langle v_1, v_2\rangle \in \overline{G_2}(R)\} \\
\overline{G_1 + G_2}(R) & = & \{\langle \kappa_1(u_1), \kappa_1(u_2)\rangle \mid \langle u_1, u_2\rangle \in \overline{G_1}(R)\} \cup \\
 & & \{\langle \kappa_2(u_1), \kappa_2(u_2)\rangle \mid \langle u_1, u_2\rangle \in \overline{G_2}(R)\} \\
\overline{G^A}(R) & = & \{(f, g) \mid \forall_{a\in A}\langle f(a), g(a)\rangle \in \overline{G}(R)\}
\end{array}$$

☼

The definition of $\overline{G}(R)$ resembles the unique extension of a polynomial functor to a relator (also known as relation lifting). In fact, the only difference is in the clause for constant functors. In the typical definition of relation lifting [10, 18] we would have

$$\overline{B}(R) = \{\langle b, b\rangle \mid b \in B\}$$

In our case, the join-semilattice structure of B is transferred to the extension of the constant functor. The traditional definition of relation lifting is used for defining directly bisimulation instead of simulation. The latter is then defined by composition with an order relation [11]. In this paper, we consider simulation as main notion of equivalence and thus we have a different notion of relation lifting.

The notion of simulation is used for abstraction and refinement, while bisimulation plays an important role in the minimization of coalgebras.

Definition 7 ((Bi)simulation for G-coalgebras) Let (S, f) and (T, g) be two G-coalgebras. We call a relation $R \subseteq S \times T$ a *simulation* if for all $(s, t) \in S \times T$:

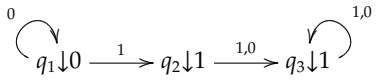
$$(s, t) \in R \Rightarrow (f(s), g(t)) \in \overline{G}(R)$$

We call R a *bisimulation* relation if both R and its (relational) inverse R^{-1} are simulations. ☼

We write $s \lesssim_G t$ (resp. $s \sim_G t$) whenever there exists a simulation relation (bisimulation relation) containing (s, t); and we call $\lesssim_G$ and $\sim_G$ the similarity and bisimilarity relations. We shall drop the subscript G whenever the functor G is clear from the context. By definition, we have $\lesssim \cap \lesssim^{-1} = \sim$.

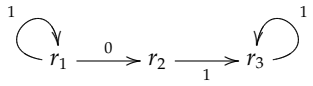
Let us now see some concrete examples of (bi)similar automata. In the last example the role of $\bot$ is also illustrated.

(i) Consider the following binary Moore automata:



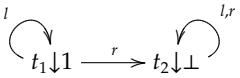
Observe that q_3 and q_2 are bisimilar, since $R = \{(q_2, q_3), (q_3, q_3)\}$ is a bisimulation. A minimal automaton is obtained by identifying all bisimilar states, yielding the automaton $\mathcal{A}_1$ presented in (1).

(ii) Consider the following partial automata



The states r_2 and r_3 are bisimilar, since $R = \{(r_2, r_3), (r_3, r_3)\}$ is a bisimulation. The automaton $\mathcal{A}_2$ presented in (2) is a minimal representation of this automaton.

(iii) Consider the following binary tree automaton



Note that this automaton simulates any fully-specified binary tree automaton substituting either 0 or 1 as output for the abstract $\bot$ value in the state t_2. For example, considering the tree automaton $\mathcal{A}_3$, presented in (3), we have $t_2 \lesssim s_2$ because $S = \{(t_2, s_2)\}$ is a simulation relation.

3 Regular expressions

In this section we wintroduce a language of regular expressions for coalgebras. We start by introducing an untyped language of expressions and then we single out the well-typed ones via an appropriate typing system, associating expressions to polynomial functors.

Definition 8 (Expressions) Let A be a finite set and let B be a finite join-semilattice. Furthermore, let X be a set of fixed point variables. The set of all *expressions* is given by the following grammar:

$$\varepsilon \quad ::= \quad \emptyset \mid x \mid \varepsilon \oplus \varepsilon \mid \mu x.\gamma \mid b \mid l(\varepsilon) \mid r(\varepsilon) \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon)$$

where γ is a *guarded expression* given by:

$$\gamma \quad ::= \quad \emptyset \mid \gamma \oplus \gamma \mid \mu x.\gamma \mid b \mid l(\varepsilon) \mid r(\varepsilon) \mid l[\varepsilon] \mid r[\varepsilon] \mid a(\varepsilon)$$

☼

A *closed expression* is an expression without free occurrences of fixed point variables x. We denote the set of guarded and closed expressions by Exp. Without loss of generality, we consider only guarded expressions. In fact, for every unguarded expression there exists a provably equivalent guarded expression (with respect to the equational system we will introduce in Section 3.3). This can be proven in a similar way as the equivalent result for μ-calculus formulas [17, Theorem 2.1].

Intuitively, expressions denotes elements of the final coalgebra. If we think of the latter as a collection of sets, then the expression $\emptyset$ denotes the empty set, while $\varepsilon_1 \oplus \varepsilon_2$ denotes the union of the sets denoted by ε_1 and ε_2. The expressions $l(\varepsilon)$, $r(\varepsilon)$, $l[\varepsilon]$, $r[\varepsilon]$ and $a(\varepsilon)$ refer to the left and right hand-side of products and sums, and function application, respectively. Finally, the expression $\mu x.\, \varepsilon$ denotes the least fixed-point. We shall soon illustrate, by means of examples, the role of these expressions.

Our language does not have any operator denoting intersection or complement (only includes the sum operator $\oplus$). This is a natural restriction, very much in the spirit of Kleene's regular expressions for deterministic finite automata. Interestingly, we will prove that this simple language is expressive enough to denote exactly all finite coalgebras.

Next, we present a typing assignment system for associating expressions to polynomial functors. The typing proceeds following the structure of the expressions and the ingredients of the functors.

Definition 9 (Typed expressions) We type *guarded* expressions ε using the ingredient relation, as follows:

$$\frac{F \neq F_1 + F_2}{\vdash \emptyset : F \lhd G} \qquad \frac{\vdash \varepsilon_1 : F \lhd G \quad \vdash \varepsilon_2 : F \lhd G}{\vdash \varepsilon_1 \oplus \varepsilon_2 : F \lhd G} \qquad \frac{}{\vdash x : F \lhd G} \qquad \frac{\vdash \varepsilon : F \lhd G}{\vdash \mu x.\varepsilon : F \lhd G}$$

$$\frac{}{\vdash b : B \lhd G} \qquad \frac{\vdash \varepsilon : G \lhd G}{\vdash \varepsilon : Id \lhd G} \qquad \frac{\vdash \varepsilon : F_1 \lhd G}{\vdash l(\varepsilon) : F_1 \times F_2 \lhd G} \qquad \frac{\vdash \varepsilon : F_2 \lhd G}{\vdash r(\varepsilon) : F_1 \times F_2 \lhd G}$$

$$\frac{\vdash \varepsilon : F_1 \lhd G}{\vdash l[\varepsilon] : F_1 + F_2 \lhd G} \qquad \frac{\vdash \varepsilon : F_2 \lhd G}{\vdash r[\varepsilon] : F_1 + F_2 \lhd G} \qquad \frac{\vdash \varepsilon : F \lhd G}{\vdash a(\varepsilon) : F^A \lhd G}$$

☼

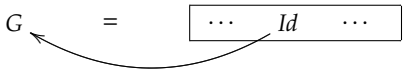
Note that the presented type system is decidable (expressions are of finite length and the system is recursive). Roughly, $\varepsilon : F \lhd G$ means that the set denoted by ε is an element (up to bisimulation) of $F(\Omega_G)$. Since in **Set** coproducts are disjoint unions, every element of $F_1 + F_2(\Omega_G)$ must belong to either $F_1(\Omega_G)$ or $F_2(\Omega_G)$, from which follows that any assignment of the expression $\emptyset$ to either of the two set is arbitrary. This explains our restriction in the first rule above.

Definition 10 (G-expressions) Let G be a polynomial functor and F an ingredient of G. We denote by $Exp_{F \lhd G}$ the following set:

$$Exp_{F \lhd G} = \{\varepsilon \in Exp \mid \vdash \varepsilon : F \lhd G\}\,.$$

The set Exp_G of *G-expressions* is $Exp_{G \lhd G}$. ☼

An important remark is that $Exp_{Id \lhd G} = Exp_G$. This is easily explained by the fact that for a polynomial functor G, if Id is one of the ingredients of G, then it is functioning as a pointer to the functor being defined:



If we instantiate the above definition for our running examples we obtain the following.

Definition 11 (Moore expressions) Let A be a set of input actions and let B be a finite join-semilattice of output actions. Furthermore, let X be a set of (recursion or) fixed point variables. The set Exp_M of *Moore expressions* is given by the following BNF syntax. For $a \in A$, $b \in B$, and $x \in X$:

$$\varepsilon ::= \emptyset \mid x \mid a(\varepsilon) \mid \downarrow b \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon$$

where occurrences of fixed point variables in ε are within the scope of an input action in A. ☼

Here, we have introduced some simplifications: $\downarrow b$ abbreviates $r(b)$ and $a(\varepsilon)$ is used instead of $l(a(\varepsilon))$.

Definition 12 (Partial automata expressions) Let A be a set of input actions and X be a set of (recursion or) fixed point variables. The set Exp_P of *partial expressions* is given by the following BNF syntax. For $a \in A$ and $x \in X$:

$$\varepsilon ::= \emptyset \mid x \mid a(\varepsilon) \mid a\uparrow \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon$$

where occurrences of fixed point variables in ε are within the scope of an input action in A. ☼

Again, for simplicity, $a\uparrow$ and $a(\varepsilon)$ abbreviate $a(l[*])$ and $a(r[\varepsilon])$.

Definition 13 (Tree expressions) Let A be a set of input actions and X be a set of (recursion or) fixed point variables. The set Exp_T of *tree expressions* is given by the following BNF syntax. For $b \in B$ and $x \in X$:

$$\varepsilon ::= \emptyset \mid x \mid l(\varepsilon) \mid r(\varepsilon) \mid b \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon$$

where occurrences of fixed point variables in ε are within the scope of either an l or a r label. ☼

Again, we have introduced some abbreviations: b represents $l(b)$, $l(\varepsilon)$ is in fact $r(l(\varepsilon))$ and $r(\varepsilon)$ is $r(r(\varepsilon))$). Note that the functor $T = B \times X \times X$ is isomorphic to the functor $B \times X^2$. Thus, by taking $A = 2 = \{l, r\}$ in the definition of Moore expressions we would obtain an equivalent grammar to the one of tree expressions.

3.1 Expressions are coalgebras

In this section we show that the set of G-expressions for a given polynomial functor G has a coalgebraic structure

$$\lambda_G : Exp_G \to G(Exp_G)\,.$$

We proceed by induction on the ingredients of G. More precisely we are going to define a function

$$\lambda_{F \lhd G} : Exp_{F \lhd G} \to F(Exp_G)$$

and then set $\lambda_G = \lambda_{G \lhd G}$. The definition of the above function requires the lifting of expressions in $Exp_{F \lhd G}$ to $F(Exp_G)$. In particular, we need a lift for $\emptyset \in Exp_{F \lhd G}$ and one for the operator $\oplus$ between expressions in $Exp_{F \lhd G}$.

Definition 14 For every polynomial functor F, let $F_\star = \{\star\} + F$.

(i) We define a constant $Empty_{F \lhd G} \in F_\star(Exp_{G \lhd G})$, as follows:

$$\begin{aligned}
&Empty_{Id \lhd G} = \begin{cases} \emptyset & G \neq G_1 + G_2 \\ \star & otherwise \end{cases} \\
&Empty_{B \lhd G} = \bot_B \\
&Empty_{F_1 \times F_2 \lhd G} = \langle Empty_{F_1 \lhd G}, Empty_{F_2 \lhd G} \rangle \\
&Empty_{F_1 + F_2 \lhd G} = \star \\
&Empty_{F^A \lhd G} = \lambda a.Empty_{F \lhd G}
\end{aligned}$$

(ii) We define $Plus_{F\lhd G} : F_\star(Exp_{F\lhd G}) \times F_\star(Exp_{F\lhd G}) \to F_\star(Exp_{F\lhd G})$ in the following way:

$$\begin{array}{lcl}
Plus_{F\lhd G}(\star, \varepsilon) & = & \varepsilon \\
Plus_{F\lhd G}(\varepsilon, \star) & = & \varepsilon \\
Plus_{Id\lhd G}(\varepsilon_1, \varepsilon_2) & = & \varepsilon_1 \oplus \varepsilon_2 \\
Plus_{B\lhd G}(b_1, b_2) & = & b_1 \vee_B b_2 \\
Plus_{F_1\times F_2\lhd G}(\langle \varepsilon_1, \varepsilon_2\rangle, \langle \varepsilon_3, \varepsilon_4\rangle) & = & \langle Plus_{F_1\lhd G}(\varepsilon_1, \varepsilon_3), Plus_{F_2\lhd G}(\varepsilon_2, \varepsilon_4)\rangle \\
Plus_{F_1+F_2\lhd G}(\kappa_i(\varepsilon_1), \kappa_i(\varepsilon_2)) & = & \kappa_i(Plus_{F_1\lhd G}(\varepsilon_1, \varepsilon_2)), \quad i \in \{1, 2\} \\
Plus_{F_1+F_2\lhd G}(\kappa_i(\varepsilon_1), \kappa_j(\varepsilon_2)) & = & \star \quad i, j \in \{1, 2\} \ \textit{and} \ i \neq j \\
Plus_{F^A\lhd G}(f, g) & = & \lambda a.\, Plus_{F\lhd G}(f(a), g(a))
\end{array}$$

☼

The symbol $\star$ is due to the partiality of our type system. In fact, $\emptyset \notin Exp_{G_1+G_2}$, which explain why $Empty_{Id\lhd G_1+G_2}$ and $Empty_{F_1+F_2\lhd G}$ is $\star$. Furthermore, if $F_1 + F_2$ is an ingredient of G then $Plus_{F\lhd G}$ may be undefined. This may happen when one of the expressions involved in the sum stems from F_1 and the other from F_2.

Now we have all we need to define $\lambda_{F\lhd G}$. Since it use the lifted operators defined above, $\lambda_{F\lhd G}$ has type $Exp_{F\lhd G} \to F_\star(Exp_G)$ rather than $Exp_{F\lhd G} \to F(Exp_G)$.

Definition 15 For every ingredient F of a polynomial functor G and expression $\varepsilon \in Exp_{F\lhd G}$, we define $\lambda_{F\lhd G}(\varepsilon)$ as follows:

$$\begin{array}{l}
\lambda_{F\lhd G}(\emptyset) = Empty_{F\lhd G} \\
\lambda_{F\lhd G}(\varepsilon_1 \oplus \varepsilon_2) = Plus_{F\lhd G}(\lambda_{F\lhd G}(\varepsilon_1), \lambda_{F\lhd G}(\varepsilon_2)) \\
\lambda_{F\lhd G}(\mu x.\varepsilon) = \lambda_{F\lhd G}(\varepsilon[\mu x.\varepsilon/x]) \\
\lambda_{Id\lhd G}(\varepsilon) = \varepsilon \\
\lambda_{B\lhd G}(b) = b \\
\lambda_{F_1\times F_2\lhd G}(l(\varepsilon)) = \langle \lambda_{F_1\lhd G}(\varepsilon), Empty_{F_2\lhd G}\rangle \\
\lambda_{F_1\times F_2\lhd G}(r(\varepsilon)) = \langle Empty_{F_1\lhd G}, \lambda_{F_2\lhd G}(\varepsilon)\rangle \\
\lambda_{F_1+F_2\lhd G}(l[\varepsilon]) = \kappa_1(\lambda_{F_1\lhd G}(\varepsilon)) \\
\lambda_{F_1+F_2\lhd G}(r[\varepsilon]) = \kappa_2(\lambda_{F_2\lhd G}(\varepsilon)) \\
\lambda_{F^A\lhd G}(a(\varepsilon)) = \lambda a'. \begin{cases} \lambda_{F\lhd G}(\varepsilon) & a = a' \\ Empty_{F\lhd G} & otherwise \end{cases}
\end{array}$$

Here, $\varepsilon[\mu x.\varepsilon/x]$ denotes syntactic substitution, replacing in ε every free occurrence of x. ☼

The above definition uses induction on the structure of F and the following complexity measure, which is based on the expression complexity and number of nested unguarded occurrences of μ-expressions:

$$\begin{array}{lcl}
N(\emptyset) & = & N(b) = N(a(\varepsilon)) = N(l(\varepsilon)) = N(r(\varepsilon)) = N(l[\varepsilon]) = N(r[\varepsilon]) = 0 \\
N(\varepsilon_1 \oplus \varepsilon_2) & = & max\{N(\varepsilon_1), N(\varepsilon_2)\} + 1 \\
N(\mu x.\varepsilon) & = & 1 + N(\varepsilon)
\end{array}$$

In order to see that the definition of $\lambda_{F\lhd G}$ is well-formed, note that in the case of $\mu x.\varepsilon$, we have:

$$N(\varepsilon) = N(\varepsilon[\mu x.\varepsilon/x])$$

This can easily be proved by (standard) induction on the syntactic structure of ε, since ε is guarded (in x).

The function $\lambda_{G\lhd G}$ above gives to G-expression a structure similar to that of a G-coalgebra. The only difference is that some expressions can be mapped to $\star$, meaning that the transition function is undefined.

Definition 16 (Regular expressions) For every ingredient F of a polynomial functor G, we say that an expression $\varepsilon \in Exp_{F\lhd G}$ is *regular* if $\lambda_{F\lhd G}(\varepsilon) \neq \star$. The set of regular expressions of $F \lhd G$ is denoted by $R_{F\lhd G}$ and we write R_G for $R_{G\lhd G}$. ☼

Note that if G does not have ingredients of the form $F_1 + F_2$ then every well-typed expression is regular, *i.e.*, $R_{F \triangleleft G} = Exp_{F \triangleleft G}$. Thus, all the Moore and tree expressions defined above are regular. For the case of the partial automata, $a(l[\emptyset] \oplus r[\emptyset])$ is an example of a non-regular expression whereas $a\uparrow \oplus b(\emptyset)$ is regular only if $A = \{a, b\}$, with $a \neq b$.

We have now shown that for each polynomial functor G, the set R_G of regular G-expressions together with λ_G is a coalgebra. It is important to remark that the (sub)coalgebra generated by an expression $\varepsilon \in R_G$ by repeatedly applying λ_G is, in general, infinite. Take for instance the Moore regular expression $\varepsilon_1 = \mu x.1(x \oplus \mu y.1(y))$ and observe that:

$$\begin{array}{lcl} \lambda_M(\varepsilon_1) & = & \langle \emptyset, \varepsilon_1 \oplus \mu y.1(y) \rangle \\ \lambda_M(\varepsilon_1 \oplus \mu y.1(y)) & = & \langle \emptyset, \varepsilon_1 \oplus \mu y.1(y) \oplus \mu y.1(y) \rangle \\ \lambda_M(\varepsilon_1 \oplus \mu y.1(y) \oplus \mu y.1(y)) & = & \langle \emptyset, \varepsilon_1 \oplus \mu y.1(y) \oplus \mu y.1(y) \oplus \mu y.1(y) \rangle \\ & \vdots & \end{array}$$

As one would expect, all these states are equivalent (using for example the equations we introduce in Section 3.3). However, the function λ_M does not make any state identification and thus yield an infinite coalgebra.

In the remainder of the paper we present a coalgebraic semantics for our regular expressions and a sound and complete (for the modal fragment) system of equations. Finally, we give an algorithm, which produces a *finite* coalgebra, based on a normalization of regular expressions which uses the equations of our system.

3.2 Satisfaction relation and semantics

Having a G-coalgebra structure on R_G has two advantages. First, it provides us, by finality, directly with a natural semantics because of the existence of a (unique) homomorphism:

$$\begin{array}{ccc} R_G & \xrightarrow{[\![\cdot]\!]} & \Omega_G \\ {\scriptstyle \lambda_G}\downarrow & & \downarrow{\scriptstyle \omega_G} \\ GR_G & \xrightarrow[G[\![\cdot]\!]]{} & G\Omega_G \end{array}$$

It assigns to every expression ε an element $[\![\varepsilon]\!]$ of the final coalgebra Ω_G.

The second advantage of the coalgebra structure on R_G is that it lets us use the notion of simulation between a G-coalgebra (S, g) and an expression $\varepsilon \in R_G$. This notion is closely related to the interpretation of regular expressions in terms of subsets of the states of a given coalgebra, which we present next.

Definition 17 For every G-coalgebra (S, g) and every valuation $\eta : Var \to \mathcal{P}(S)$, we define a function $\| \cdot \|^G_\eta : R_G \to \mathcal{P}S$. Whenever s belongs to $\|\varepsilon\|^G_\eta$, we say that s *satisfies* ε. The function $\| \cdot \|^G_\eta$ is defined as:

$$\|\varepsilon\|^G_\eta = \{s \mid g(s) \in \|\varepsilon\|^{G \triangleleft G}_\eta\},$$

where the function $\| \cdot \|^{F \triangleleft G}_\eta : R_{F \triangleleft G} \to \mathcal{P}(FS)$ is defined inductively on the structure of the expression as follows:

$$\begin{array}{lcl} \|\emptyset\|^{F \triangleleft G}_\eta & = & \emptyset \\ \|\varepsilon_1 \oplus \varepsilon_2\|^{F \triangleleft G}_\eta & = & \|\varepsilon_1\|^{F \triangleleft G}_\eta \cup \|\varepsilon_2\|^{F \triangleleft G}_\eta \\ \|x\|^{F \triangleleft G}_\eta & = & \eta(x) \\ \|\mu x.\varepsilon\|^{F \triangleleft G}_\eta & = & \bigcap \{T \subseteq S \mid \|\varepsilon\|^{F \triangleleft G}_{\eta[T/x]} \subseteq T\} \\ \|\varepsilon\|^{Id \triangleleft G}_\eta & = & \|\varepsilon\|^G_\eta \\ \|b\|^{B \triangleleft G}_\eta & = & \{b' \mid b \not\leq_B b'\} \\ \|l(\varepsilon)\|^{F_1 \times F_2 \triangleleft G}_\eta & = & \{s \mid \pi_1(s) \in \|\varepsilon\|^{F_1 \triangleleft G}_\eta\} \\ \|r(\varepsilon)\|^{F_1 \times F_2 \triangleleft G}_\eta & = & \{s \mid \pi_2(s) \in \|\varepsilon\|^{F_2 \triangleleft G}_\eta\} \\ \|l[\varepsilon]\|^{F_1 + F_2 \triangleleft G}_\eta & = & \{\kappa_1(s) \mid s \in \|\varepsilon\|^{F_1 \triangleleft G}_\eta\} \\ \|r[\varepsilon]\|^{F_1 + F_2 \triangleleft G}_\eta & = & \{\kappa_2(s) \mid s \in \|\varepsilon\|^{F_2 \triangleleft G}_\eta\} \\ \|a(\varepsilon)\|^{F^A \triangleleft G}_\eta & = & \{s \mid s(a) \in \|\varepsilon\|^{F \triangleleft G}_\eta\} \end{array}$$

Here, $\eta[T/x]$ denotes the valuation such that for every $v \in Var$, with $v \neq x$, returns $\eta(v)$ and for $v = x$ returns T. ☼

It can be shown, by a fairly straightforward and not very instructive proof, that if $\varnothing$ denotes the everywhere empty valuation, we have:

$$\varepsilon \lesssim s \quad \Leftrightarrow \quad s \in \|\varepsilon\|^G_\varnothing$$

for every $\varepsilon \in R_G$. Moreover, since every regular expression is closed we have that, for every valuation η, $\|\varepsilon\|^G_\eta = \|\varepsilon\|^G_\varnothing$. Therefore, we omit the subscript η whenever it is not needed.

The following theorem shows that our logic is sufficiently expressive to characterize bisimilarity. Moreover it shows that every state of finite coalgebra is bisimilar to a regular expression.

Theorem 18 Let G be a polynomial functor and (S, g) a G-coalgebra.

(1) For all states $s, s' \in S$, $s \sim s'$ if and only if $\forall \varepsilon \in R_G \,.\, \varepsilon \lesssim s \Leftrightarrow \varepsilon \lesssim s'$.

(2) If S is *finite* then there exists for any $s \in S$ an expression $\varepsilon_s \in R_G$ such that $\varepsilon_s \sim s$.

Proof. (1) Because $s \sim s'$ implies $s \lesssim s'$ and $s' \lesssim s$ we have, for any $\varepsilon \in R_G$,

$$\varepsilon \lesssim s \iff \varepsilon \lesssim s'$$

For the converse, note that the following relation

$$T_G \quad = \quad \left\{ \langle s, s' \rangle \in S \times S \mid \ \forall \varepsilon \in R_G \,.\, s \in \|\varepsilon\|^G \Rightarrow s' \in \|\varepsilon\|^G \right\}$$

and its inverse ${T_G}^{-1}$ are simulation relations on S. Thus T_G is a bisimulation.

To prove that T_G (resp. T_G^{-1}) is a simulation relation we have to first observe that defining

$$T_{F \triangleleft G} = \left\{ \langle s, s' \rangle \in FS \times FS \mid \ \forall \varepsilon \in R_{F \triangleleft G} \,.\, s \in \|\varepsilon\|^{F \triangleleft G} \Rightarrow s' \in \|\varepsilon\|^{F \triangleleft G} \right\}$$

we can prove by induction on the structure of F that

$$T_{F \triangleleft G} = \overline{F}(T_G)$$

Now, to prove that T_G is a simulation relation we must prove that for all $(s, s') \in T_G$, the pair $(g(s), g(s'))$ is an element of $T_{G \triangleleft G}$.

$$\begin{array}{lcl} g(s) \in \|\varepsilon\|^{G \triangleleft G} & \Leftrightarrow & s \in \|\varepsilon\|^G \\ & \Rightarrow & s' \in \|\varepsilon\|^G \\ & \Leftrightarrow & g(s') \in \|\varepsilon\|^{G \triangleleft G} \end{array}$$

(2) We want to construct for a given $s \in S$ an expression ε_s with $\varepsilon_s \sim s$. To this end, we associate with every state $s \in S$ a variable $x_s \in X$ and an expression $\varepsilon_s = \mu x_s . \varepsilon^G_s$ defined by induction on the structure of G as follows.

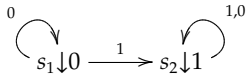
$$\begin{array}{l} \varepsilon^{Id}_s = x_{g(s)} \\ \varepsilon^{B}_s = g(s) \\ \varepsilon^{G_1 \times G_2}_s = l(\varepsilon^{G_1}_{\pi_1 \circ g(s)}) \oplus r(\varepsilon^{G_2}_{\pi_2 \circ g(s)}) \\ \varepsilon^{G_1 + G_2}_s = \begin{cases} l[\varepsilon^{G_1}_{g(s)}] & g(s) = \kappa_1(s') \\ r[\varepsilon^{G_2}_{g(s)}] & g(s) = \kappa_2(s') \end{cases} \\ \varepsilon^{G^A}_s = \bigoplus_{a \in A} a(\varepsilon^{G}_{g(s)(a)}) \end{array}$$

In the above definition we encounter expressions $\varepsilon^G_{s'}$ where $s' \in GS$. We define this expressions again by induction on the structure of G as follows:

$$\begin{array}{l} \varepsilon^{Id}_s = x_s \\ \varepsilon^{B}_b = b \\ \varepsilon^{G_1 \times G_2}_{\langle s, s' \rangle} = l(\varepsilon^{G_1}_s) \oplus r(\varepsilon^{G_2}_{s'}) \\ \varepsilon^{G_1 + G_2}_{\kappa_1(s)} = l[\varepsilon^{G_1}_s] \\ \varepsilon^{G_1 + G_2}_{\kappa_2(s)} = r[\varepsilon^{G_2}_s] \\ \varepsilon^{G^A}_f = \bigoplus_{a \in A} a(\varepsilon^{G}_{f(a)}) \end{array}$$

Syntactically replacing free occurrences of $x_{s'}$ by $\varepsilon_{s'}$ ensure that all ε_s is in R_G. By construction, $s \sim \varepsilon_s$. ☺

Let us illustrate the last construction above. Recall the Moore automaton $\mathcal{A}_1$ presented before:



We define $\varepsilon_1 = \mu x_1.\, \varepsilon$ and $\varepsilon_2 = \mu x_2.\, \varepsilon'$ by

$$\varepsilon = {\downarrow}0 \oplus 0(x_1) \oplus 1(x_2) \qquad \varepsilon' = {\downarrow}1 \oplus 1(x_2) \oplus 1(x_2)$$

Substituting x_2 by ε' in ε_1 then yields

$$\varepsilon_1 = \mu x_1.\, {\downarrow}0 \oplus 0(x_1) \oplus 1(\varepsilon_2) \qquad \varepsilon_2 = \mu x_2.\, {\downarrow}1 \oplus 1(x_2) \oplus 1(x_2)$$

By construction we have $s_1 \sim \varepsilon_1$ and $s_2 \sim \varepsilon_2$.

3.3 Equations

We now introduce an equational system between regular expressions of type $F \lhd G$. Again, we omit type annotation whenever we think there is no possibility of confusion.

For each polynomial functor G and its ingredient F, the first group of equations gives to the set of expressions $EXP_{F \lhd G}$ the structure of a bounded join-semilattice.

(*Idempotency*)	$\varepsilon \oplus \varepsilon = \varepsilon$
(*Commutativity*)	$\varepsilon_1 \oplus \varepsilon_2 = \varepsilon_2 \oplus \varepsilon_1$
(*Associativity*)	$\varepsilon_1 \oplus (\varepsilon_2 \oplus \varepsilon_3) = (\varepsilon_1 \oplus \varepsilon_2) \oplus \varepsilon_3$
(*Empty*)	$\emptyset \oplus \varepsilon = \varepsilon$

While the above equations are polymorphic, the next group of equations depends on the type of the expressions we are considering. The equations show the interactions between regular expressions and the join-semilattice structure.

$(B - \emptyset)$	$\emptyset = \bot_B$	$(B - \oplus)$	$b_1 \oplus b_2 = b_1 \vee_B b_2$
$(\times - \emptyset - L)$	$l(\emptyset) = \emptyset$	$(\times - \oplus - L)$	$l(\varepsilon_1 \oplus \varepsilon_2) = l(\varepsilon_1) \oplus l(\varepsilon_2)$
$(\times - \emptyset - R)$	$r(\emptyset) = \emptyset$	$(\times - \oplus - R)$	$r(\varepsilon_1 \oplus \varepsilon_2) = r(\varepsilon_1) \oplus r(\varepsilon_2)$
$(+ - \oplus - L)$	$l[\varepsilon_1 \oplus \varepsilon_2] = l[\varepsilon_1] \oplus l[\varepsilon_2]$	$(+ - \oplus - R)$	$r[\varepsilon_1 \oplus \varepsilon_2] = r[\varepsilon_1] \oplus r[\varepsilon_2]$
$(-^A - \emptyset)$	$a(\emptyset) = \emptyset$	$(-^A - \oplus)$	$a(\varepsilon_1 \oplus \varepsilon_2) = a(\varepsilon_1) \oplus a(\varepsilon_2)$

The fact that $\emptyset$ is not an expression for the functor $G_1 + G_2$ is reflected by the absence of equations for the distribution of $\emptyset$ over the $l[-]$ and $r[-]$ operators.

Finally, the last two rules $(\mu - i)$ and $(\mu - e)$ can be explained as stating that the term $\mu x.\gamma$ is the least prefixed point with respect to the order induced by join-semilattice structure, when viewing the open expression γ as a (monotone) map on expressions.

(*Fixed − point*)	$\gamma(\mu x.\gamma/x) = \gamma$
(*Park − induction*)	$\gamma(\varepsilon/x) \oplus \varepsilon = \varepsilon \Rightarrow \mu x.\gamma \oplus \varepsilon = \varepsilon$

The last equational implication could be eliminated in favour of an axiomatization that would use only equations, using Santocanale's method [28] at the price of extending our theory of regular expressions so that form also a a residuated semilattice, as for action algebras [22].

For two regular G-expressions $\varepsilon_1, \varepsilon_2 \in R_G$, we write $\vdash^G \varepsilon_1 = \varepsilon_2$ to indicate that the equality $\varepsilon_1 = \varepsilon_2$ is derivable from the above equations.

Theorem 19 (Soundness) The above equational system is sound, that is, for every polynomial functor G, if $\vdash^G \varepsilon_1 = \varepsilon_2$ then $\varepsilon_1 \sim_G \varepsilon_2$.

Proof. By an easy verification of the above identities using the definitions of $\|-\|^G$ and λ_G. ☺

Next we give, for each polynomial functor G a logical characterization of the final G-coalgebra. Let $\Im_{F\lhd G}$ be the set of all ideals of (the Lindenbaum algebra of) $R_{F\lhd G}$, where a *ideal* of a join-semilattice is a non-empty lower closed subset $\mathcal{I}$ such that if $a, b \in \mathcal{I}$ then also the join of a and b is in $\mathcal{I}$.

For every G-coalgebra (S, f), we can define a function $k_S{:}S \to \Im_G$ by setting

$$k_S(s) = \{\varepsilon \in R_G \mid s \notin \|\varepsilon\|^G\}$$

for every $s \in S$. Thus $k_S(s)$ is the set of all regular G-expressions that cannot simulate s. It is easy to verify that $k_S(s)$ is an ideal. Next we show that k_S is also a G-homomorphism. First of all we turn the set $\Im_G = \Im_{G\lhd G}$ into a G-coalgebra by the function $\zeta_G = \zeta_{G\lhd G}$ defined below.

Definition 20 For every polynomial functor G and ingredient F we define the function

$$\zeta_{F\lhd G} : \Im_{F\lhd G} \to F(\Im_G)$$

by induction on the ingredients of G as follows:

$$\begin{array}{lcll}
\zeta_{Id\lhd G}(\mathcal{I}) & = & \mathcal{I} & \\
\zeta_{B\lhd G}(\mathcal{I}) & = & \bigvee_B \mathcal{I} & \\
\pi_1(\zeta_{F_1\times F_2\lhd G}(\mathcal{F})) & = & \zeta_{F_1\lhd G}(\{\varepsilon | l(\varepsilon) \in \mathcal{I}\}) & \\
\pi_2(\zeta_{F_1\times F_2\lhd G}(\mathcal{I})) & = & \zeta_{F_2\lhd G}(\{\varepsilon | r(\varepsilon) \in \mathcal{I}\}) & \\
\zeta_{F_1+F_2\lhd G}(\mathcal{I}) & = & \begin{cases} \kappa_1(\zeta_{F_1\lhd G}(\{\varepsilon | l[\varepsilon] \in \mathcal{I}\})) & \text{if } l[\emptyset] \in \mathcal{I} \\ \kappa_2(\zeta_{F_2\lhd G}(\{\varepsilon | r[\varepsilon] \in \mathcal{I}\}) & \text{otherwise} \end{cases} & \\
\zeta_{F^A\lhd G}(\mathcal{I}) & = & \lambda a \in A.\zeta_{F\lhd G}(\{\varepsilon | a(\varepsilon) \in \mathcal{I}\}) &
\end{array}$$

☼

Note that in order $\zeta_{B\lhd G}$ to be well defined we assume B to be a *finite* join-semilattice. In case B is infinite, we would need B to be a complete join-semilattice. It is easy to see by induction of the ingredients of G that $\zeta_{F\lhd G}$ is injective. The only interesting case is when F is a join-semilattice B, which follows because ideals are lower closed sets.

Theorem 21 For every G-coalgebra (S, g) the function $k_S : S \to \Im_G$ is a G-homomorphism.

Proof. To begin with, we prove that, for every $x \in GS$, we have

$$\zeta_{F\lhd G}(\{\varepsilon \in R_{F\lhd G} \mid x \notin \|\varepsilon\|^{F\lhd G}\}) = (Fk_S)(x)\,. \tag{4}$$

We proceed by induction on the ingredients of G. We show only three cases. If B is an ingredient of G then

$$\begin{array}{lcll}
\zeta_{B\lhd G}(\{b' \mid b \notin \|b'\|^{B\lhd G}\} & = & \bigvee\{b' \mid b \notin \|b'\|^{B\lhd G}\} & \text{definition of } \zeta \\
& = & \bigvee\{b' \mid b' \leq_B b\} & \text{definition of } \|-\| \\
& = & b & \\
& = & (B\ k_S)(b) & \text{definition of } B.
\end{array}$$

More interesting is the other base case of the induction, when Id is an ingredient of G:

$$\begin{array}{lcll}
\zeta_{Id\lhd G}(\{\varepsilon \mid s \notin \|\varepsilon\|^{Id\lhd G}\}) & = & \{\varepsilon \mid s \notin \|\varepsilon\|^{Id\lhd G}\} & \text{definition of } \zeta \\
& = & \{\varepsilon \mid s \notin \|\varepsilon\|^G\} & \text{definition of } \|-\| \\
& = & k_S(s) & \text{definition of } k \\
& = & (Id\ k_S)(s) & \text{definition of } Id.
\end{array}$$

Finally we consider one case where induction is applied:

$$\begin{array}{lcll}
\zeta_{F^A\lhd G}(\{\varepsilon \mid h \notin \|\varepsilon)\|^{F^A\lhd G}\} & = & \lambda a \in A.\zeta_{F\lhd G}(\{\varepsilon \mid h \notin \|a(\varepsilon)\|^{F^A\lhd G}\}) & \text{definition of } \zeta \\
& = & \lambda a \in A.\zeta_{F\lhd G}(\{\varepsilon \mid h(a) \notin \|\varepsilon\|^{F\lhd G}\}) & \text{definition of } \|-\| \\
& = & \lambda a \in A.(F\ k_S)(h(a)) & \text{induction} \\
& = & (F^A k_S)(h) & \text{definition of } F^A.
\end{array}$$

We can now prove that k is a G-homomorphism by applying (4) above. In fact we have, for all $s \in S$,

$$\zeta_G(k_S(s)) = \zeta_{G\lhd G}(\{\varepsilon \mid s \notin \|\varepsilon\|^G\}) = \zeta_{G\lhd G}(\{\varepsilon \mid g(s) \notin \|\varepsilon\|^{G\lhd G}\}) = G(k_S)(g(s))$$

☺

As a consequence of the first item of Theorem 18 we have that $k_\Omega : \Omega_G \to \mathfrak{I}_G$ is injective. In fact, for all $s, t \in \Omega_G$:

$$\begin{array}{rcl} s = t & \Leftrightarrow & s \sim t \\ & \Leftrightarrow & \forall \varepsilon \in R_G.(s \in \|\varepsilon\|^G \Leftrightarrow t \in \|\varepsilon\|^G) \\ & \Leftrightarrow & k_{\Omega_G}(s) = k_{\Omega_G}(t)\,. \end{array}$$

It follows that k_{Ω_G} has a left inverse, that by finality of Ω_G, must be the unique G-homomorphism from $\mathfrak{I}_G$ to Ω_G.

3.4 Finality and Completeness of the Modal Fragment

In this subsection we prove completeness for the modal fragment of the set of regular expressions, that is, for all regular expressions which do not contain fixed point operators. We denote by $\mathfrak{I}^\sharp_{F\lhd G}$ the set of all ideals of the Lindenbaum algebra of the modal regular expressions in $R_{F\lhd G}$, and use $\mathfrak{I}^\sharp_G$ as a shorthand for $\mathfrak{I}^\sharp_{G\lhd G}$.

Proceeding in a similar way as in the previous section we have functions $\zeta^\sharp_{F\lhd G}{:}\mathfrak{I}^\sharp_{F\lhd G} \to F(\mathfrak{I}^\sharp_{G\lhd G})$ and $k^\sharp_S{:}S \to \mathfrak{I}^\sharp_G$ for every G-coalgebra (S,f). The former turns $\mathfrak{I}^\sharp_G$ into a G-coalgebra, while the latter is an homomorphism between G-coalgebras. Next we show that $\mathfrak{I}^\sharp_G$ is actually the final G-coalgebra.

Theorem 22 For every G-coalgebra (S,g) the homomorphism $k^\sharp_S{:}S \to \mathfrak{I}^\sharp_G$ is unique.

Proof. We prove a more general fact, namely that for every ingredient F of G, $f{:}S \to FS$ and $h{:}S \to \mathfrak{I}^\sharp_{F\lhd G}$ such that $\zeta^\sharp_{F\lhd G} \circ h = Fh \circ f$, it holds that, for every $s \in S$,

$$\varepsilon \in \zeta^\sharp_{F\lhd G}(h(s)) \quad \Leftrightarrow \quad f(s) \notin \|\varepsilon\|^{F\lhd G}\,. \tag{5}$$

We proceed by induction on the structure of ε. The cases for the expressions $\emptyset$ and $\varepsilon_1 \oplus E_2$ follow easily because $h(s)$ is an ideal and by the definition of $\|\varepsilon\|^{F\lhd G}$. We consider two other cases. If we take the expression $b \in B$, then F is the functor B. Thus we have

$$\begin{array}{rcll} b \in \zeta^\sharp_{B\lhd G}(h(s)) & \Leftrightarrow & b \leq_B f(s) & \text{commutativity, definition of } \zeta^\sharp_{B\lhd G} \\ & \Leftrightarrow & f(s) \notin \|b\|^{B\lhd G} & \text{definition of } \|-\|. \end{array}$$

In the case the expression is $a(\varepsilon)$, we first define the functions $f_a{:}S \to FS$ and $h_a{:}S \to \mathfrak{I}^\sharp_{F\lhd G}$ by setting

$$f_a(s) = f(s)(a) \;\text{ and }\; h_a(s) = \{\varepsilon \mid a(\varepsilon) \in h(s)\}$$

for every $s \in S$. Note that the following commutativity property holds $\zeta^\sharp_{F\lhd G}(h_a(s)) = F(h)(f_a(s))$. Further we have

$$\begin{array}{rcll} a(\varepsilon) \in h(s) & \Leftrightarrow & \varepsilon \in \zeta_{F^A\lhd G}(h(s))(a) & \text{definition of } \zeta_{F^A} \\ & \Leftrightarrow & \varepsilon \in F^A(h)(f(s))(a) & \text{commutativity} \\ & \Leftrightarrow & \varepsilon \in F(h)(f_a(s)) & \text{definitions of } F^A(g) \text{ and } f_a \\ & \Leftrightarrow & \varepsilon \in \zeta_{F\lhd G}(h_a(s)) & \text{commutativity} \\ & \Leftrightarrow & f_a(s) \notin \|\varepsilon\|^{F\lhd G} & \text{induction} \\ & \Leftrightarrow & f(s) \notin \|a(\varepsilon)\|^{F^A\lhd G} & \text{definitions of } \|-\| \text{ and } f_a. \end{array}$$

We use the above result to prove the statement of the theorem. Let $h{:}S \to \mathfrak{I}^\sharp_G$ be another homomorphism, and let $s \in S$. We have, by definition of $k^\sharp_S$ and $\|-\|$,

$$\varepsilon \in k^\sharp_S(s) \Leftrightarrow s \notin \|\varepsilon\|^G \Leftrightarrow g(s) \notin \|\varepsilon\|^{G\lhd G}\,.$$

By applying (5) once with h and another time with $k_S^\sharp$ we obtain

$$\zeta_G^\sharp(h(s)) = \zeta_G^\sharp(k_S(s))\,.$$

Because ζ is injective, it must be $h(s) = k_S(s)$. ☺

Note that because final coalgebras are unique up-to isomorphism, we have that for every polynomial functor G the homomorphism $k_{\Omega_G}{:}\Omega_G \to \Im_G^\sharp$ is an isomorphism. As a consequence of Theorem 18, this isomorphism is also an order isomorphism, where the order on Ω_G is simulation, whereas that on $\Im_G^\sharp$ is subset inclusion. The logical significance of this result is that our finitary language of expression with only a join operation and no fixed-points suffices to completely describe all coalgebra up to bisimilarity.

Theorem 22 together with the next lemma gives a logical interpretation of the final coalgebra: its elements correspond to complements of canonical models (in the logical sense) of regular expressions seen as logical formulas.

Lemma 23 For every polynomial functor G, modal regular expression $\varepsilon \in R_G$, and ideal $\mathcal{I} \in \Im_G^\sharp$ it holds that $\mathcal{I} \in \|\varepsilon\|^G$ if and only if $\varepsilon \notin \mathcal{I}$.

Proof. Since $\|\varepsilon\|^G = \{\mathcal{I} \mid \zeta_{G\lhd G}(\mathcal{I}) \in \|\varepsilon\|^{G\lhd G}\}$ it is enough to prove, for every ingredient F of G, that

$$\zeta_{F\lhd G}(\mathcal{I}) \in \|\varepsilon\|^{F\lhd F^A} \Leftrightarrow \varepsilon \notin \mathcal{I}\,.$$

We prove the above statement by induction on the structure of ε. Clearly, $\emptyset \in \mathcal{I}$ and $\mathcal{I} \notin \|E\|^{F\lhd G}$. More generally,

$$\begin{array}{lcll}
\zeta_{F\lhd G}(\mathcal{I}) \notin \|\varepsilon_1 \oplus \varepsilon_2\|^{F\lhd G} & \Leftrightarrow & \zeta_{F\lhd G}(\mathcal{I}) \notin \|\varepsilon_1\|^{F\lhd G} \text{ and } \zeta_{F\lhd G}(\mathcal{I}) \notin \|\varepsilon_2\|^{F\lhd G} & \\
& \Leftrightarrow & \varepsilon_1 \in \mathcal{I} \text{ and } \varepsilon_2 \in \mathcal{I} & \text{induction} \\
& \Leftrightarrow & \varepsilon_1 \oplus \varepsilon_2 \in \mathcal{I} & \mathcal{I} \text{ is an ideal.}
\end{array}$$

Next we consider the case when the expression is an element of B. By definition of $\|-\|$ and ζ_G we have

$$\zeta_{B\lhd G}(\mathcal{I}) \notin \|b\|^{B\lhd G} \Leftrightarrow b \leq \zeta_{B\lhd G}(\mathcal{I}) \Leftrightarrow b \leq \bigvee_B \mathcal{I} \Leftrightarrow b \in \mathcal{I}$$

We conclude with one more case, when the expression belongs to $R_{F^A\lhd G}$:

$$\begin{array}{lcll}
\zeta_{F^A\lhd G}(\mathcal{I}) \in \|a(\varepsilon)\|^{F^A\lhd G} & \Leftrightarrow & \zeta_{F^A\lhd G}(\mathcal{I})(a) \in \|\varepsilon\|^{F\lhd G} & \text{definition of } \|-\| \\
& \Leftrightarrow & \zeta_{F\lhd G}(\{\varepsilon' \mid a(\varepsilon') \in \mathcal{I}\}) \in \|\varepsilon\|^{F\lhd F^A} & \text{definition of } \zeta \\
& \Leftrightarrow & \varepsilon \in \{\varepsilon' \mid a(\varepsilon') \in \mathcal{I}\}) & \text{induction} \\
& \Leftrightarrow & a(\varepsilon) \in \mathcal{I}\,. &
\end{array}$$

☺

We can finally prove the completeness of the modal regular expressions.

Theorem 24 (Completeness) For every polynomial functor G and modal regular expressions $\varepsilon_1, \varepsilon_2 \in R_G$, for all G-coalgebra (S,f), if $\varepsilon_1 \sim_G \varepsilon_2$ then $\vdash \varepsilon_1 = \varepsilon_2$.

Proof. By Theorem 18 it is enough to find a state in a G-coalgebra (S,f) that is in $\|\varepsilon_1\|^G$ but not in $\|\varepsilon_2\|^G$. Without loss of generality, assume $\nvdash \varepsilon_1 \leq \varepsilon_2$. Define $\mathcal{I} = \{\varepsilon \mid \varepsilon \leq \varepsilon_1\}$. It is not very difficult to verify that $\mathcal{I}$ is an ideal, hence it is an element of the coalgebra $\Im_G^\sharp$. Clearly, $\varepsilon_2 \in \mathcal{I}$ but, by our assumption ε_1 is not. We can now conclude by applying Lemma 23. ☺

4 Synthesis

We now give a construction to prove the converse of the second item of Theorem 18, that is, we describe the synthesis process that produce a *finite* G-coalgebra from an arbitrary regular G-expression ε. The states of the resulting G-coalgebra are a finite subset of regular expressions expressions, including an expression ε' such that $\varepsilon \sim_G \varepsilon'$.

4.1 Formulae normalization

We have seen that the first group of axioms and rules of our proof system gives to the set of expressions the structure of a join-semilattice. In order to guarantee the termination of the synthesis process it is enough to identify expressions that are provably equivalent using only these axioms and rules. Since our axiomatization is sound we obtain expressions bisimilar to the original ones. For instance, it is easy to see that the Moore expressions

$$a(\emptyset) \oplus \downarrow b \oplus \emptyset \oplus \downarrow b \text{ and } a(\emptyset) \oplus \downarrow b$$

are equivalent using the equations (*Idenpotency*) and (*Empty*).

We thus work with normalized expressions in order to eliminate any syntactic redundancy present in the expression: in a sum, $\emptyset$ can be eliminated and, by idempotency, the sum of two syntactically equivalent expressions can be simplified. The function $norm_G : Exp_G \to Exp_G$ encodes this procedure. We define it by induction on the expression structure as follows:

$$\begin{array}{lcl}
norm_G(\emptyset) & = & \emptyset \\
norm_G(\varepsilon_1 \oplus \varepsilon_2) & = & plus(rem(flatten(norm_G(\varepsilon_1) \oplus norm_G(\varepsilon_2)))) \\
norm_G(\mu x.\varepsilon) & = & \mu x.(norm_G(\varepsilon))\,. \\
norm_B(b) & = & b \\
norm_{G_1 \times G_2}(l(\varepsilon)) & = & l(norm_{G_1}(\varepsilon)) \\
norm_{G_1 \times G_2}(r(\varepsilon)) & = & r(norm_{G_2}(\varepsilon)) \\
norm_{G_1 + G_2}(l[\varepsilon]) & = & l[norm_{G_1}(\varepsilon)] \\
norm_{G_1 + G_2}(r[\varepsilon]) & = & r[norm_{G_2}(\varepsilon)] \\
norm_{G^A}(a(\varepsilon)) & = & a(norm_G(\varepsilon))
\end{array}$$

Here, *plus* takes a list of expressions $[\varepsilon_1, \ldots, \varepsilon_n]$ and returns the expression $\varepsilon_1 \oplus \ldots \oplus \varepsilon_n$ (*plus* applied to the empty list yields $\emptyset$), *rem* removes duplicates in a list and *flatten* takes an expression ε and produces a list of expressions by omitting brackets and replacing $\oplus$-symbols by commas:

$$\begin{array}{lcl}
flatten(\varepsilon_1 \oplus \varepsilon_2) & = & flatten(\varepsilon_1) \cdot flatten(\varepsilon_2) \\
flatten(\emptyset) & = & [] \\
flatten(\varepsilon) & = & [\varepsilon],\ \varepsilon \in \{b, a(\varepsilon_1), l(\varepsilon_1), r(\varepsilon_1), l[\varepsilon_1], r[\varepsilon_1], \nu x.\varepsilon_1\}
\end{array}$$

In this definition, $\cdot$ denotes list concatenation and $[\varepsilon]$ the singleton list containing ε. Note that any occurrence of $\emptyset$ in a sum is eliminated because $flatten(\emptyset) = []$.

For example, the normalization of the two Moore expressions above results in the same expression – $a(\emptyset) \oplus \downarrow b$.

Note that $norm_G$ still distinguishes the expressions $\varepsilon_1 \oplus \varepsilon_2$ and $\varepsilon_2 \oplus \varepsilon_1$. To simplify the presentation of the normalization algorithm, we decided not to identify these expressions, since this does not influence termination. In the examples below this situation will never occur.

In the synthesis procedure we need a lifted version of the function $norm_G$. More precisely, for a given ingredient F of a polynomial functor G, we define a function $norm_{F \lhd G} : F(Exp_G) \to F(Exp_G)$ by induction on the structure of F as follows:

$$\begin{array}{l}
norm_{F \lhd G} : F(Exp_G) \to F(Exp_G) \\
norm_{Id \lhd G} = norm_G \\
norm_{B \lhd G}(b) = b \\
norm_{F_1 \times F_2 \lhd G}(\langle \varepsilon_1, \varepsilon_2 \rangle) = \langle norm_{F_1 \lhd G}(\varepsilon_1), norm_{F_2 \lhd G}(\varepsilon_2) \rangle \\
norm_{F_1 + F_2 \lhd G}(\kappa_1(\varepsilon_1)) = \kappa_1(norm_{F_1 \lhd G}(\varepsilon_1)) \\
norm_{F_1 + F_2 \lhd G}(\kappa_2(\varepsilon_1)) = \kappa_2(norm_{F_2 \lhd G}(\varepsilon_1)) \\
norm_{F^A \lhd G}(f) = \lambda a.\, norm_{F \lhd G}(f(a)))
\end{array}$$

Also here, whenever it is clear from the context, we drop the type subscript $F \lhd G$ and use *norm* instead $norm_{F \lhd G}$.

4.2 Synthesis procedure

We first describe what happens in a single step of the synthesis process and then we describe the synthesis process in terms of repeatedly applying such single steps.

The function δ_G, which does *one-step synthesis* for a single expression, takes an expression $\varepsilon \in Exp_G$ and produces a partial G-coalgebra. Below, δ_G is used in the function Δ_G, which synthesizes the total G-coalgebra.

We set $\delta_G = \delta_{G \lhd G}$, where the function $\delta_{F \lhd G}$ is defined, by induction on the structure of F and on complexity measure N defined in Section 3, as follows:

$$
\begin{array}{l}
\delta_{F \lhd G} : Exp_{F \lhd G} \to F_\star(Exp_G) \\
\delta_{F \lhd G}(\emptyset) = Empty_{F \lhd G} \\
\delta_{F \lhd G}(\varepsilon_1 \oplus \varepsilon_2) = norm(Plus_{F \lhd G}(\delta_{F \lhd G}(\varepsilon_1), \delta_{F \lhd G}(\varepsilon_2))) \\
\delta_{F \lhd G}(\mu x.\varepsilon) = norm(\delta_{F \lhd G}(\varepsilon[\mu x.\varepsilon / x])) \\
\delta_{Id \lhd G}(\varepsilon) = norm(\varepsilon) \\
\delta_{B \lhd G}(b) = b \\
\delta_{F_1 \times F_2 \lhd G}(l(\varepsilon)) = \langle \delta_{F_1 \lhd G}(\varepsilon), Empty_{F_2 \lhd G} \rangle \\
\delta_{F_1 \times F_2 \lhd G}(r(\varepsilon)) = \langle Empty_{F_1 \lhd G}, \delta_{F_2 \lhd G}(\varepsilon) \rangle \\
\delta_{F_1 + F_2 \lhd G}(l[\varepsilon]) = \kappa_1(\delta_{F_1 \lhd G}(\varepsilon)) \\
\delta_{F_1 + F_2 \lhd G}(r[\varepsilon]) = \kappa_2(\delta_{F_2 \lhd G}(\varepsilon)) \\
\delta_{F^A \lhd G}(a(\varepsilon)) = \lambda a'. \begin{cases} \delta_{F \lhd G}(\varepsilon) & a = a' \\ Empty_{F \lhd G} & otherwise \end{cases}
\end{array}
$$

Note that this function is very similar to the function $\lambda_{F \lhd G}$ presented in Section 3. In fact, the only difference is the normalization that is now being applied to the expressions so that $\delta_{F \lhd G}(\varepsilon) = norm_{F \lhd G}(\lambda_{F \lhd G}(\varepsilon))$. This guarantee that the coalgebra is finite. If we restrict the domain of $\delta_{F \lhd G}$ to regular expressions then the result has type $F(R_{F \lhd G})$.

To compute the entire coalgebra that satisfies ε, we need to apply δ_G to the new states generated at each step repeatedly until all states in the coalgebra have their transition structure fully defined.

We implement this procedure with the auxiliary function D_G. The arguments of this function are two sets of states: $sts \subseteq R_G$, the states that still need to be processed and $vis \subseteq R_G$, the states that already have been visited (synthesized). For each $\varepsilon \in sts$, D_G computes $\delta_G(\varepsilon)$ and produces an intermediate transition function (possibly partial) by taking the union of all those $\delta_G(\varepsilon)$. Then, it collects all new states appearing in this step and recursively computes the transition function for those.

$$
\begin{array}{rl}
D_G(sts, vis) = \begin{cases} \emptyset & sts = \emptyset \\ trans \cup D_G(newsts, vis') & otherwise \end{cases} \\
\quad \textbf{where} \quad trans = \{\langle \varepsilon, \delta_G(\varepsilon) \rangle \mid \varepsilon \in sts\} \\
\qquad\qquad sts' = collectStates_G(trans) \\
\qquad\qquad vis' = sts \cup vis \\
\qquad\qquad newsts = sts' \setminus vis'
\end{array}
$$

The function $collectStates_G$ is defined inductively in the structure of G as follows:

$$
\begin{array}{l}
collectStates_G : GS \to \mathcal{P}S \\
collectStates_{Id}(s) = \{s\} \\
collectStates_B(b) = \{\} \\
collectStates_{G_1 \times G_2}(\langle s_1, s_2 \rangle) = collectStates_{G_1}(s_1) \cup collectStates_{G_2}(s_2) \\
collectStates_{G_1 + G_2}(k_i(s)) = collectStates_{G_i}(s) \quad i \in \{1, 2\} \\
collectStates_{G^A}(f) = \bigcup_{a \in A} collectStates_G(f(a))
\end{array}
$$

The function Δ takes an expression $\varepsilon \in R_G$ and returns a G-coalgebra that satisfies ε:

$$
\Delta_G(\varepsilon) = (dom(g), g) \qquad \textbf{where} \quad g = D_G(\{norm_G(\varepsilon)\}, \emptyset)
$$

The function *dom* returns the domain of a finite function. The finiteness and termination of the synthesis algorithm is proved in Section 4.4.

4.3 Concrete synthesis: examples

In this section we give some concrete examples of the general synthesis algorithm presented in the previous section.

4.3.1 Moore automata

The one-step synthesis function δ_G presented above for a polynomial functor G instantiates for the Moore automata functor to

$$\delta_M : R_M \to B \times (R_M)^A$$

by defining δ_M as following. For $a \in A$ and $\varepsilon \in R_M$, we write $\delta_M(\varepsilon) = \langle \varepsilon\downarrow, \varepsilon_a \rangle$ and we define $\varepsilon\downarrow$ and ε_a by:

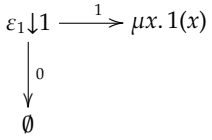
$$\begin{array}{ll}
\emptyset\downarrow = \bot_B & \emptyset_a = \emptyset \\
(\varepsilon_1 \oplus \varepsilon_2)\downarrow = \varepsilon_1\downarrow \vee_B \varepsilon_2\downarrow & (\varepsilon_1 \oplus \varepsilon_2)_a = norm((\varepsilon_1)_a \oplus (\varepsilon_2)_a) \\
(\mu x.\varepsilon)\downarrow = (\varepsilon[\mu x.\varepsilon/x])\downarrow & (\mu x.\varepsilon)_a = norm((\varepsilon[\mu x.\varepsilon/x])_a) \\
(\downarrow b)\downarrow = b & (\downarrow b)_a = \emptyset \\
(a(\varepsilon))\downarrow = \bot_B & (a(\varepsilon))_{a'} = \begin{cases} norm(\varepsilon) & a = a' \\ \emptyset & otherwise \end{cases}
\end{array}$$

Let us now present a few examples of Moore regular expressions and the correspondent synthesized Moore automata. To simplify the presentation, we consider binary Moore automata.

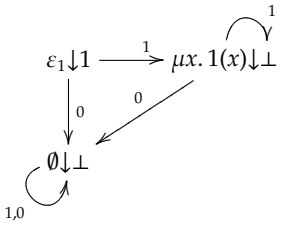
Let $\varepsilon_1 = 1\downarrow \oplus (\mu x.1(x))$. We can easily compute that $(\varepsilon_1)_0 = \emptyset$ and

$$\begin{array}{rcll}
(\varepsilon_1)\downarrow & = & (1\downarrow)\downarrow \vee (\mu x.1(x))\downarrow & \qquad (\varepsilon_1)_1 = norm((1\downarrow)_1 \oplus (\mu x.1(x))_1) \\
& = & 1 \vee \bot & \qquad = norm(\emptyset \oplus (1(\mu x.1(x)))_1) \\
& = & 1 & \qquad = \mu x.1(x)
\end{array}$$

So, $\delta_M(\varepsilon_1)$ is a (partial) finite function represented by the following diagram:



We now apply δ_M to the two new states, which yields the following Moore automata.



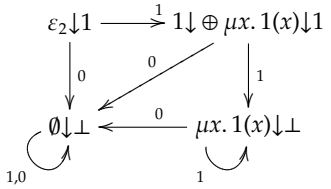
Note that the resulting Moore automata is not minimal. In this example, the states $\emptyset$ and $\mu x.1(x)$ are bisimilar and could be identified.

Let us see a few other examples of the synthesis process. The expressions presented below, similarly to ε_1, only has information for the input 1. Therefore, for the 0 input, the expression ε under consideration holds $(\varepsilon)_0 = \emptyset$.

Let us consider $\varepsilon_2 = 1(1\downarrow) \oplus \mu x.1(x)$. We have:

$$\begin{array}{rcll}
(\varepsilon_2)\downarrow & = & (1(1\downarrow))\downarrow \vee (\mu x.1(x))\downarrow & \qquad (\varepsilon_2)_1 = norm(1((1\downarrow))_1 \oplus (\mu x.1(x))_1) \\
& = & \bot \vee \bot & \qquad = norm(1\downarrow \oplus (1(\mu x.1(x)))_1) \\
& = & \bot & \qquad = 1\downarrow \oplus \mu x.1(x)
\end{array}$$

We now repeat the process for $1\downarrow \oplus \mu x.1(x)$, obtaining $(1\downarrow \oplus \mu x.1(x))\downarrow = 1$ and $(1\downarrow \oplus \mu x.1(x))_1 = \mu x.1(x)$. Finally, we calculate $(\mu x.1(x))\downarrow = \bot$ and $(\mu x.1(x))_1 = \mu x.1(x)$. The complete Moore automata is depicted in the following diagram:



Now, take $\varepsilon_3 = \mu x.\, 1(x \oplus \mu y.\, 1(y))$. We have $(\varepsilon_3)\downarrow = \bot$ and:

$$\begin{array}{lcl}(\varepsilon_3)_1 & = & norm((1(\varepsilon_3 \oplus \mu y.\, 1(y)))_1) \\ & = & \varepsilon_3 \oplus \mu y.\, 1(y)\end{array}$$

We proceed with the new state:

$$\begin{array}{lcl}(\varepsilon_3 \oplus \mu y.\, 1(y))_1 & = & norm((\varepsilon_3)_1 \oplus (\mu y.\, 1(y))_1) \\ & = & norm(\varepsilon_3 \oplus \mu y.\, 1(y) \oplus \mu y.\, 1(y)) \\ & = & \varepsilon_3 \oplus \mu y.\, 1(y)\end{array}$$

Note that if *norm* would not have been applied, the resulting state $\varepsilon_3 \oplus \mu y.\, 1(y) \oplus \mu y.\, 1(y)$ would be regarded as a new state, even though it is equivalent to $\varepsilon_3 \oplus \mu y.\, 1(y)$. Moreover, applying δ_M to this state (for input 1) would yield again an equivalent but (syntactically) different state, namely $\varepsilon_3 \oplus \mu y.\, 1(y) \oplus \mu y.\, 1(y) \oplus \mu y.\, 1(y)$. This illustrates that the function λ_G, defined in Section 3, generally produces an infinite machine, as was already pointed out in the same section. However, the identifications made by *norm* ensure the termination of the synthesis process, which we formally prove in Section 4.4.

4.3.2 Partial automata

For partial automata, the one-step synthesis function δ_G presented above instantiates to

$$\delta_P : R_P \to (1 + R_P)^A$$

by defining δ_P as following. For $a \in A$ and $\varepsilon \in R_M$, we define $\delta_P(\varepsilon)(a)$ by:

$$\begin{array}{lcl}\delta_P(\varepsilon_1 \oplus \varepsilon_2)(a) & = & \overline{\delta}_P(\varepsilon_1)(a) \mathbin{\overline{\oplus}} \overline{\delta}_P(\varepsilon_2)(a) \\ \delta_P(\mu x.\varepsilon)(a) & = & norm(\delta_P(\varepsilon[\mu x.\varepsilon/x])(a))\end{array}$$

where $\overline{\delta}_P$ and $\overline{\oplus}$ are defined as:

$$\begin{array}{lcl}\overline{\delta}_P & : & Exp_P \to (1 + Exp_P)^A \\ \overline{\delta}_P(\varepsilon_1 \oplus \varepsilon_2)(a) & = & \overline{\delta}_P(\varepsilon_1)(a) \mathbin{\overline{\oplus}} \overline{\delta}_P(\varepsilon_2)(a) \\ \overline{\delta}_P(\mu x.\varepsilon)(a) & = & norm(\delta_P(\varepsilon[\mu x.\varepsilon/x])(a)) \\ \overline{\delta}_P(a(\varepsilon))(a') & = & \begin{cases} \kappa_2(norm(\varepsilon)) & a = a' \\ \star & otherwise \end{cases} \\ \overline{\delta}_P(a\uparrow)(a') & = & \begin{cases} \kappa_1(*) & a = a' \\ \star & otherwise \end{cases}\end{array}$$

$$\begin{array}{lcllcl}\star & \overline{\oplus} & \varepsilon & = & norm(\varepsilon) & \\ \varepsilon & \overline{\oplus} & \star & = & norm(\varepsilon) & \\ \kappa_1(*) & \overline{\oplus} & \kappa_1(*) & = & \kappa_1(*) & \\ \kappa_2(\varepsilon_1) & \overline{\oplus} & \kappa_2(\varepsilon_2) & = & \kappa_2(norm(\varepsilon_1 \oplus \varepsilon_2)) & \\ \kappa_i(\varepsilon_1) & \overline{\oplus} & \kappa_j(\varepsilon_2) & = & \star & i \neq j\end{array}$$

Note that $\delta_P(\emptyset)$ is not defined, because $\emptyset$ is not a regular expression. The same happens, for instance, with the expression $1\uparrow \oplus 1(\varepsilon)$, which is also not regular, since it contains contradictory information about the behaviour of the automata for input 1. Moreover, any expression that has information missing about a certain input is not regular. One can easily see that if ε does not contain information about a certain input a, then $\overline{\delta}_P(\varepsilon) = \star$.

Let us see a few examples of the synthesis of regular expressions for binary partial automata. Consider the expression $\varepsilon_1 = 1\uparrow \oplus 0(1\uparrow \oplus 0\uparrow)$.

Note that $\delta_P(\varepsilon_1)(1) = \kappa_1(*)$, because $\delta_P(1\uparrow)(1) = \kappa_1(*)$ and $\delta_P(0(1\uparrow \oplus 0\uparrow))(1) = \star$. For input 0, $\delta_P(\varepsilon_1)(0) = 1\uparrow \oplus 0\uparrow$, since $\delta_P(1\uparrow)(0) = \star$ and $\delta_P(0(1\uparrow \oplus 0\uparrow))(0) = 1\uparrow \oplus 0\uparrow$.

So $\delta_P(\varepsilon_1)$ is represented by the following diagram:

$$\varepsilon_1 \xrightarrow{0} 1\uparrow \oplus 0\uparrow$$

Repeating the process for the new state $1\uparrow \oplus 0\uparrow$, we have that $\delta_P(1\uparrow \oplus 0\uparrow)(0) = \kappa_1(*)$ and $\delta_P(1\uparrow \oplus 0\uparrow)(1) = \kappa_1(*)$. Therefore, the above diagram represents the complete transition function.

Now, take the expression $\varepsilon_2 = 0(\mu x.\, 1\uparrow \oplus 0(x)) \oplus 1(\mu x.\, 0(x) \oplus 1(x))$.

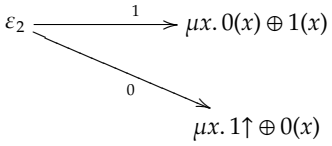
We have:

$$\begin{array}{lcl}\delta_P(\varepsilon_2)(0) & = & \delta_P(0(\mu x.\, 1\uparrow \oplus 0(x)))(0) \mathbin{\overline{\oplus}} \delta_P(1(\mu x.\, 0(x) \oplus 1(x)))(0) \\ & = & \kappa_2(norm(\mu x.\, 1\uparrow \oplus 0(x))) \mathbin{\overline{\oplus}} \star \\ & = & \kappa_2(\mu x.\, 1\uparrow \oplus 0(x))\end{array}$$

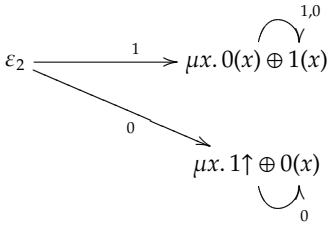
and

$$\begin{array}{rcl}\delta_P(\varepsilon_2)(1) & = & \delta_P(0(\mu x.\, 1\uparrow \oplus 0(x)))(1) \mathbin{\overline{\oplus}} \delta_P(1(\mu x.\, 0(x) \oplus 1(x)))(1) \\ & = & \star \mathbin{\overline{\oplus}} \kappa_2(norm(\mu x.\, 0(x) \oplus 1(x))) \\ & = & \kappa_2(\mu x.\, 0(x) \oplus 1(x))\end{array}$$

So, the (partial) transition function for ε_2 is depicted in the following diagram:



Computing δ_P for the two new states yield the following partial automaton:



4.4 Proof of termination

We finally prove that our synthesis algorithm deliver a finite coalgebra for every regular expression.

Definition 25 For any $\varepsilon \in Exp_{F\triangleleft G}$ we define the closure of ε as the smallest set $cl_{F\triangleleft G}(\varepsilon) \subseteq Exp_{F\triangleleft G}$ satisfying the following conditions

$$\begin{array}{l}\varepsilon \in cl_{F\triangleleft G}(\varepsilon) \\ \varepsilon_1 \oplus \varepsilon_2 \in cl_{F\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1, \varepsilon_2 \in cl_{F\triangleleft G}(\varepsilon) \\ \mu x.\, \varepsilon_1 \in cl_{F\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1[\mu x.\, \varepsilon_1/x] \in cl_{F\triangleleft G}(\varepsilon) \\ \varepsilon_1 \in cl_{Id\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1 \in cl_{G\triangleleft G}(\varepsilon) \\ l(\varepsilon_1) \in cl_{F_1\times F_2\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1 \in cl_{F_1\triangleleft G}(\varepsilon) \\ r(\varepsilon_1) \in cl_{F_1\times F_2\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1 \in cl_{F_2\triangleleft G}(\varepsilon) \\ l[\varepsilon_1] \in cl_{F_1+F_2\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1 \in cl_{F_1\triangleleft G}(\varepsilon) \\ r[\varepsilon_1] \in cl_{F_1+F_2\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1 \in cl_{F_2\triangleleft G}(\varepsilon) \\ a(\varepsilon_1) \in cl_{F^A\triangleleft G}(\varepsilon) \Rightarrow \varepsilon_1 \in cl_{F\triangleleft G}(\varepsilon)\end{array}$$

☼

One can prove that the set $cl_{F\triangleleft G}(\varepsilon)$ is finite for any expression $\varepsilon \in Exp_{F\triangleleft G}$ (using a similar argument as the one for the equivalent result for μ-calculus formulae [13]). Also note that if $\varepsilon_1 \in cl_{F\triangleleft G}(\varepsilon)$, then $cl_{F\triangleleft G}(\varepsilon_1) \subseteq cl_{F\triangleleft G}(\varepsilon)$.

Theorem 26 Let $\varepsilon \in R_{F\triangleleft G}$. Then, $\lambda_{F\triangleleft G}(\varepsilon) = Empty_{F\triangleleft G}$ or $\lambda_{F\triangleleft G}(\varepsilon) = Plus_{F\triangleleft G}(\varepsilon_1, \varepsilon_2)$, where $collectStates_F(\varepsilon_1), collectStates_F(\varepsilon_2) \subseteq cl(\varepsilon)$.

Proof. By induction on the structure of F and on the complexity measure N defined in Section 3. The result follows directly from the definition of $\lambda_{F\triangleleft G}$. Let us exemplify the case $\varepsilon = \varepsilon_1 \oplus \varepsilon_2$. The other cases use a similar argument.

$$\lambda_{F\triangleleft G}(\varepsilon_1 \oplus \varepsilon_2) = Plus_{F\triangleleft G}(\lambda_{F\triangleleft G}(\varepsilon_1), \lambda_{F\triangleleft G}(\varepsilon_2))$$

Now, applying the induction hypothesis we know that $\lambda_{F\triangleleft G}(\varepsilon_1)) = Plus_{F\triangleleft G}(\gamma_1, \gamma_2)$, where $collectStates_F(\gamma_1), collectStates_F(\gamma_2) \subseteq cl(\varepsilon_1)$. But, $cl(\varepsilon_1) \subseteq cl(\varepsilon)$ and thus:

$$\begin{aligned} collectStates_F(\lambda_{F\lhd G}(\varepsilon_1)) &= collectStates_F(Plus_{F\lhd G}(\gamma_1,\gamma_2)) \\ &= collectStates_F(\gamma_1) \cup collectStates_F(\gamma_2) \\ &\subseteq cl(\varepsilon) \end{aligned}$$

Similarly, because $cl(\varepsilon_2) \subseteq cl(\varepsilon)$, then $collectStates_F(\lambda_{F\lhd G}(\varepsilon_2)) \subseteq cl(\varepsilon)$.

☺

Before we present the next theorem we need to introduce a generalization of $Plus_{F\lhd G}$ that instead of taking two expressions as arguments takes a (finite) list of expressions. This operator – $Plus_{F\lhd G}[\varepsilon_1,\ldots,\varepsilon_k]$ – is defined in the obvious way using the binary $Plus_{F\lhd G}$, taking $Plus_{F\lhd G}[] = Empty_{F\lhd G}$.

Theorem 27 Let $\varepsilon \in R_{F\lhd G}$. Then, $\delta_{F\lhd G}(\varepsilon) = Plus_{F\lhd G}[\varepsilon_1,\ldots,\varepsilon_k]$, where $\varepsilon_k \in cl_{F\lhd G}(\varepsilon)$, $\varepsilon_i \neq \varepsilon_j$ $(i \neq j)$ and $k \leq |cl(\varepsilon)|$.

Proof. We need to prove that $norm_{F\lhd G}(\lambda_{F\lhd G}(\varepsilon)) = Plus_{F\lhd G}[\varepsilon_1,\ldots,\varepsilon_k]$, where $\varepsilon_k \in cl_{F\lhd G}(\varepsilon)$, $\varepsilon_i \neq \varepsilon_j$ $(i \neq j)$ and $k \leq |cl(\varepsilon)|$.

Using the previous theorem, we know that $\lambda_{F\lhd G}(\varepsilon) = Empty_{F\lhd G}$ or $\lambda_{F\lhd G}(\varepsilon) = Plus_{F\lhd G}(\varepsilon_1,\varepsilon_2)$, where $collectStates_F(\varepsilon_1), collectStates_F(\varepsilon_2) \subseteq cl(\varepsilon)$.

In the first case, the result follows directly. In the second case, we prove that $norm_{F\lhd G}(Plus_{F\lhd G}(\varepsilon_1,\varepsilon_2)) = Plus_{F\lhd G}[\varepsilon'_1,\ldots,\varepsilon'_k]$ by induction on the structure of F. We show the most interesting case, when $F = Id$.

$$\begin{aligned} norm_{Id\lhd G}(Plus_{Id\lhd G}(\varepsilon_1,\varepsilon_2)) &= norm_G(\varepsilon_1 \oplus \varepsilon_2) \\ &= plus(rem(flatten(norm_G(\varepsilon_1) \oplus norm_G(\varepsilon_2)))) \end{aligned}$$

Applying the induction hypothesis, we have that $norm_G(\varepsilon_1) = Plus_{Id\lhd G}[\gamma_1,\ldots,\gamma_k]$ and $norm_G(\varepsilon_2) = Plus_{Id\lhd G}[\gamma'_1,\ldots,\gamma'_k]$ with $\gamma_i \in cl(\varepsilon_1)$ and $\gamma'_i \in cl(\varepsilon_2)$. But $cl(\varepsilon_1) \subseteq cl(\varepsilon)$ and $cl(\varepsilon_2) \subseteq cl(\varepsilon)$ and therefore $\gamma_i \in cl(\varepsilon)$ and $\gamma'_i \in cl(\varepsilon)$. Applying *flatten* and *rem* to this expression we know that the argument list of *plus* will only have distinct elements of $cl(\varepsilon)$ and thus $norm_{Id\lhd G}(Plus_{Id\lhd G}(\varepsilon_1,\varepsilon_2)) = Plus_{F\lhd G}[\varepsilon'_1,\ldots,\varepsilon'_k]$, with $\varepsilon'_k \in cl(\varepsilon)$, $\varepsilon'_i \neq \varepsilon'_j$ $(i \neq j)$ and $k \leq |cl(\varepsilon)|$. ☺

Theorem 28 For a given expression $\varepsilon \in R_g$, $D(\{\varepsilon\},\emptyset)$ terminates.

Proof. We know that $\delta_{F\lhd G}(\varepsilon) = Plus_{F\lhd G}[\varepsilon_1,\ldots,\varepsilon_k]$, where $\varepsilon_k \in cl(\varepsilon)$ and $k \leq |cl(\varepsilon)|$. Because $cl(\varepsilon)$ is finite the number of possible combinations for the above expression if finite. That together with the fact that for all $\varepsilon_i \in cl_{F\lhd G}(\varepsilon)$, we have $cl_{F\lhd G}(\varepsilon_1) \subseteq cl_{F\lhd G}(\varepsilon)$ gives us an upper bound for the number of states that need to be processed. Since D guarantees that no state is processed twice, the set *newsts* will eventually be empty and, therefore, $D(\{\varepsilon\},\emptyset)$ terminates. ☺

This theorem (together with the fact that the function *dom* terminates) concludes the proof that Δ terminates and that our synthesis algorithm always return a finite coalgebra for a regular expression.

5 Concluding remarks

In this paper we introduced a language of regular expressions for denoting exactly all finite coalgebras over polynomial functors on **Set**. The semantics of an expression is given coalgebraically, in terms of the final coalgebra, and it is expressive with respect to bisimulation. Regular expressions come together with an equational system that is sound in general and complete for the fixed-point free fragment. Our main result is a generalization of Kleene's theorem for ordinary regular expression and deterministic finite automata: every state of a finite coalgebra is bisimilar to an expression in our language and, conversely, every regular expression is bisimilar to a state of a finite coalgebra synthesized from that expression.

In this paper we treated only coalgebras over polynomial functors. They correspond to simple deterministic systems. Considering also the powerset functor would allow for the modelling of non-deterministic systems. Currently, we are developing a corresponding language for regular expressions of such coalgebras.

Our axiomatization of regular expressions for coalgebras is finite (but implicational), and is inspired by that of the modal mu-calculus [13]. In this paper we have proved the completeness by constructing a canonical model for the fixed-point free fragment of the language. It is a topic of future research to establish if our axiomatization is complete for the full language of regular expressions.

The language of regular expressions for polynomial functors can be extended with a negation operator, as long as we consider constant functors to be Boolean algebras B instead of join semilattices. The language obtained would have both least and greatest fixed point operators and, as for the present set of regular expressions, can be given a coalgebraic structure. However, the resulting coalgebra would not be finite in general. We are currently investigating such an extension for model checking purpose.

Finally, our results are all based on polynomial functors on **Set**. Since our language is inspired by the approach to coalgebraic logic via duality [1, 2], it is tempting to extend our results to polynomial functors on other concrete categories, so to characterize, for example, regular expressions for name-passing processes using the logical approach of [3]. Moving to other categories rather than **Set** may also simplify our current treatment of regular expressions for the coproduct construct, by allowing coalgebras to be partial so that, contrary to approach in this paper, every well-typed expression would be regular.

Acknowledgements. We would like to thank Clemens Kupke, Helle Hvid Hansen and Yde Venema for valuable suggestions and discussions.

References

[1] M. M. Bonsangue and A. Kurz. Duality for logics of transition systems. In V. Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 2005.

[2] M. M. Bonsangue and A. Kurz. Presenting functors by operations and equations. In L. Aceto and A. Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *LNCS*, pages 172–186. Springer, 2006.

[3] M. M. Bonsangue and A. Kurz. Pi-calculus in logical form. In *LICS*, pages 303–312. IEEE Computer Society, 2007.

[4] M. M. Bonsangue, J. Rutten, and A. Silva. Coalgebraic logic and synthesis of mealy machines. In R. Amadio, editor, *FoSSaCS*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[5] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[6] C. Cîrstea and D. Pattinson. Modular construction of modal logics. In P. Gardner and N. Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2004.

[7] E. M. Clarke, S. M. German, Y. Lu, H. Veith, and D. Wang. Executable protocol specification in esl. In W. A. H. Jr. and S. D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.

[8] R. Goldblatt. Equational logic of polynomial coalgebras. In P. Balbiani, N.-Y. Suzuki, F. Wolter, and M. Zakharyaschev, editors, *Advances in Modal Logic 4*, pages 149–184. King's College Publications, 2002.

[9] H. H. Hansen, D. Costa, and J. J. M. M. Rutten. Synthesis of mealy machines using derivatives. *Electronic Notes in Theoretical Computer Science*, 164(1):27–45, 2006.

[10] B. Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *ITA*, 35(1):31–59, 2001.

[11] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. 2004. Draft.

[12] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata studies*, volume 34 of *Annals of mathematics studies*, pages 3–40. Princeton University Press, 1956.

[13] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[14] D. Kozen. Partial automata and finitely generated congruences: An extension of Nerode's theorem. Technical Report DAIME PB-400, 1992.

[15] D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.

[16] O. Kupferman and M. Y. Vardi. μ-calculus synthesis. In M. Nielsen and B. Rovan, editors, *MFCS'00*, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer, 2000.

[17] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[18] C. Kupke and Y. Venema. Coalgebraic automata theory: basic results. Technical Report SEN-E0701, CWI, The Netherlands, 2007.

[19] E. F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematical Studies*, 34:129–153, 1956.

[20] L. Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, 96, 1999.

[21] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL'89*, pages 179–190, 1989.

[22] V. R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI, European Workshop, JELIA '90, Amsterdam, The Netherlands, September 10-14, 1990, Proceedings*, volume 478 of *Lecture Notes in Computer Science*. Springer, 1990.

[23] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[24] M. Rößiger. Coalgebras and modal logic. *Electronic Notes in Theoretical Computer Science*, 33, 2000.

[25] J. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorgi and R. de Simone, editors, *Proceedings of CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218, 1998.

[26] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. Fundamental Study.

[27] A. Saloomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, pages 158–169, 1966.

[28] L. Santocanale. On the equational definition of the least prefixed point. *Theoretical Computer Science*, 1-3:341–370, 2003.

[29] S. Tini and A. Maggiolo-Schettini. Compositional synthesis of generalized mealy machines. *Fundamenta Informaticae*, 60(1-4):367–382, 2004.