


A Kleene Theorem for Polynomial Coalgebras

View metadata, citation and similar papers at core.ac.uk

brought to you by  CORE

provided by Universidade do Minho: RepositoriUM

² Centrum voor Wiskunde en Informatica (CWI)

³ Vrije Universiteit Amsterdam (VUA)

Abstract. For polynomial functors G , we show how to generalize the classical notion of regular expression to G -coalgebras. We introduce a language of expressions for describing elements of the final G -coalgebra and, analogously to Kleene's theorem, we show the correspondence between expressions and finite G -coalgebras.

1 Introduction

Regular expressions were first introduced by Kleene [8] to study the properties of neural networks. They are an algebraic description of languages, offering a declarative way of specifying the strings to be recognized and they define exactly the same class of languages accepted by deterministic (and non-deterministic) finite state automata: the regular languages. The correspondence between regular expressions and (non-)deterministic automata has been widely studied and a translation between these two different formalisms is presented in most books on automata and language theory [10,6].

Formally, a deterministic automaton consists of a set of states S equipped with a transition function $\delta : S \rightarrow 2 \times S^A$ determining for each state whether or not it is final and assigning to each input symbol a next state.

Deterministic automata can be generalized to coalgebras for an endofunctor G on the category **Set**. A coalgebra is a pair (S, g) consisting of a set of states S and a transition function $g : S \rightarrow GS$, where the functor G determines the type of the dynamic system under consideration and is the base of the theory of universal coalgebra [14]. The central concepts in this theory are homomorphism of coalgebras, bisimulation equivalence and final coalgebra. These can be seen, respectively, as generalizations of automata homomorphism, language equivalence and the set of all languages. In fact, in the case of deterministic automata, the functor G would be instantiated to $2 \times Id^A$ and the usual notions would be recovered. In particular, note that the final coalgebra for this functor is precisely the set 2^{A^*} of all languages over A [15].

Given the fact that coalgebras can be seen as generalizations of deterministic automata, it is natural to investigate whether there exists an appropriate

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

notion of regular expression in this setting. More precisely: is it possible to define a language of expressions that represents precisely the behaviour of finite G -coalgebras, for a given functor G ?

In this paper, we show how to define such a language for coalgebras of polynomial functors G (a functor is polynomial if it is built inductively from the identity and constant functors, using product, coproduct and exponential). We introduce a language of expressions for describing elements of the final G -coalgebra (Section 3). Analogously to Kleene's theorem, we show the correspondence between expressions and finite G -coalgebras. In particular, we show that every state of a finite G -coalgebra corresponds to an expression in the language (Section 4) and, conversely, we give a compositional synthesis algorithm which transforms every expression into a finite G -coalgebra (Section 5).

Related Work. Regular expressions have been originally introduced by Kleene [8] as a mathematical notation for describing languages recognized by deterministic finite automata. In [15], 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 element 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 expressions. The coalgebra structure on the set of regular expressions is determined by their so-called *Brzozowski* derivatives [4]. 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 [15] and the present paper is essentially the same. All in all, one can therefore say that standard regular expressions and their treatment in [15] can be viewed as a special instance of the present approach. This is also the case for the generalization of the results in [15] to automata on guarded strings [11]. Finally, the present paper extends the results in our FoSSaCS'08 paper [3], 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 is a finite meet semilattice for the output alphabet.

2 Preliminaries

We give the basic definitions on polynomial functors and coalgebras and introduce the notion of 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, \dots and functions by lower case f, g, \dots . The collection of functions from a set X to a set Y is denoted by Y^X . Given functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ 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 $1 = \{*\}$ and it can be regarded as the empty product. We define $X + Y$ as the set $X \uplus Y \uplus \{\perp, \top\}$, where \uplus is the disjoint union of sets, with injections $X \xrightarrow{\kappa_1} X \uplus Y \xleftarrow{\kappa_2} Y$. Note that the set $X + Y$ is different from the classical coproduct of X and Y , because of the two extra elements \perp and \top . These extra elements will later be used to represent, respectively, underspecification and inconsistency in the specification of some systems. The intuition behind the need of these extra elements will become clear when we present our language of expressions and concrete examples, in Section 5.3, of systems whose type involves $+$.

Polynomial Functors. In our definition of polynomial functors we will use constant sets equipped with an information order. In particular, we will use join-semilattices. A (bounded) join-semilattice is a set B equipped with a binary operation \vee_B and a constant $\perp_B \in B$, such that \vee_B is commutative, associative and idempotent. The element \perp_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 :

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

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.

We are now ready to define the class of polynomial functors. They are functors $G : \mathbf{Set} \rightarrow \mathbf{Set}$, built inductively from the identity and constants, using \times , $+$ and $(-)^A$. Formally, the class PF of *polynomial functors* on \mathbf{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 finite set.

Typical examples of polynomial functors are $D = 2 \times Id^A$ and $P = (1 + Id)^A$. These functors, which we shall later use as our running examples, represent, respectively, the type of *deterministic* and *partial deterministic* automata. It is worth noting that although when we mentioned the type of deterministic automata in the introduction, we did not made explicit that the set 2 was a join semilattice, which is in fact the case. Also the set of classical regular expressions has a join-semilattice structure, which provides also intuition for the differences in our definition of polynomial functors, when compared with [13,7], in the use of a join-semilattice as constant and in the definition of $+$. If we want to generalize regular expressions to polynomial functors then we must guarantee that they also have such structure, namely by imposing it in the constant and $+$ functors. For the \times and $(-)^A$ we do not need to add extra elements because the semilattice structure is compositionally inherited.

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 shall use this relation later for typing our expressions.

Let $\triangleleft \subseteq PF \times PF$ be the least reflexive and transitive relation such that

$$G_1 \triangleleft G_1 \times G_2, \quad G_2 \triangleleft G_1 \times G_2, \quad G_1 \triangleleft G_1 + G_2, \quad G_2 \triangleleft G_1 + G_2, \quad G \triangleleft G^A$$

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

Coalgebras. 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 $f : S \rightarrow GS$. The functor G , together with the function f , determines the *transition structure* (or dynamics) of the G -coalgebra [14]. Deterministic automata and partial automata are, respectively, coalgebras for the functors $D = 2 \times Id^A$ and $P = (1 + Id)^A$.

A G -homomorphism from a G -coalgebra (S, f) to a G -coalgebra (T, g) is a function $h : S \rightarrow T$ preserving the transition structure, *i.e.*, such that $g \circ h = Gh \circ f$. A G -coalgebra (Ω, ω) is said to be *final* if for any G -coalgebra (S, f) there exists a unique G -homomorphism $[\cdot] : S \rightarrow \Omega$. For every polynomial functor G there exists a final G -coalgebra (Ω_G, ω_G) [14]. For instance, as we already mentioned in the introduction, the final coalgebra for the functor D is the set of languages 2^{A^*} over A , together with a transition function $d : 2^{A^*} \rightarrow 2 \times (2^{A^*})^A$ defined as $d(\phi) = \langle \phi(\epsilon), \lambda a \lambda w. \phi(aw) \rangle$. Here ϵ denotes the empty sequence and aw denotes the word resulting from prefixing w with the letter a . The notion of finality will play a key role later in providing a semantics to expressions.

Let (S, f) and (T, g) be two G -coalgebras. We call a relation $R \subseteq S \times T$ a *bisimulation* [1] if there exists a map $e : R \rightarrow GR$ such that the projections π_1 and π_2 are coalgebra homomorphisms, *i.e.* the following diagram commutes.

$$\begin{array}{ccccc}
 S & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & T \\
 f \downarrow & & \exists e \downarrow & & \downarrow g \\
 GS & \xleftarrow{G\pi_2} & GR & \xrightarrow{G\pi_1} & GT
 \end{array}$$

We write $s \sim_G t$ whenever there exists a bisimulation relation containing (s, t) and we call \sim_G the bisimilarity relation. We shall drop the subscript G whenever the functor G is clear from the context. For G -coalgebras (S, f) and (T, g) and $s \in S, t \in T$, it holds that if $s \sim t$ then $[[s]] = [[t]]$.

3 A Language of Expressions for Polynomial Coalgebras

In this section we generalize the classical notion of regular expressions to polynomial 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.

Let A be a finite set, B a finite join-semilattice and X a set of fixpoint variables. The set of all *expressions* is given by the following grammar:

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

where γ is a *guarded expression* given by:

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

A *closed expression* is an expression without free occurrences of fixpoint variables x . We denote the set of guarded and closed expressions by *Exp*.

Intuitively, expressions denote elements of the final coalgebra. The expressions \emptyset , $\varepsilon_1 \oplus \varepsilon_2$ and $\mu x. \varepsilon$ will play a similar role to, respectively, the empty language, the union of languages and the Kleene star in classical regular expressions for deterministic automata. 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 (*i.e.*, represent projections and injections), and function application, respectively. We shall soon illustrate, by means of examples, the role of these expressions.

Our language does not have any operator denoting intersection or complement (it 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. 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. This will associate with each functor G the expressions $\varepsilon \in \text{Exp}$ that are valid specifications of G -coalgebras. The typing proceeds following the structure of the expressions and the ingredients of the functors.

We type expressions ε using the ingredient relation, as follows:

$$\begin{array}{c} \hline \vdash \emptyset : F \triangleleft G \\ \hline \vdash \varepsilon : G \triangleleft G \\ \hline \vdash \varepsilon : Id \triangleleft G \end{array} \quad \begin{array}{c} \hline \vdash b : B \triangleleft G \\ \hline \vdash \varepsilon_1 : F \triangleleft G \quad \vdash \varepsilon_2 : F \triangleleft G \\ \hline \vdash \varepsilon_1 \oplus \varepsilon_2 : F \triangleleft G \end{array} \quad \begin{array}{c} \hline \vdash x : G \triangleleft G \\ \hline \vdash \varepsilon : G \triangleleft G \\ \hline \vdash \mu x. \varepsilon : G \triangleleft G \end{array}$$

$$\begin{array}{c} \vdash \varepsilon : F_1 \triangleleft G \\ \hline \vdash l(\varepsilon) : F_1 \times F_2 \triangleleft G \end{array} \quad \begin{array}{c} \vdash \varepsilon : F_2 \triangleleft G \\ \hline \vdash r(\varepsilon) : F_1 \times F_2 \triangleleft G \end{array} \quad \begin{array}{c} \vdash \varepsilon : F \triangleleft G \\ \hline \vdash a(\varepsilon) : F^A \triangleleft G \end{array}$$

$$\begin{array}{c} \vdash \varepsilon : F_1 \triangleleft G \\ \hline \vdash l[\varepsilon] : F_1 + F_2 \triangleleft G \end{array} \quad \begin{array}{c} \vdash \varepsilon : F_2 \triangleleft G \\ \hline \vdash r[\varepsilon] : F_1 + F_2 \triangleleft G \end{array}$$

This type system is simple and most rules are self-explanatory. However, for full clarification some remarks should be made. (1) Intuitively, $\varepsilon : F \triangleleft G$ means that ε is an element (up to bisimulation) of $F(\Omega_G)$. (2) As expected, there is a rule for each expression construct. The extra rule involving $Id \triangleleft G$ reflects the isomorphism between the final coalgebra Ω_G and $G(\Omega_G)$. (3) Only fixpoints at the outermost level of the functor are allowed. This does not mean however that we disallow nested fixpoints. For instance, $\mu x. a(x \oplus \mu y. a(y))$ would be a well-typed expression for the functor D of deterministic automata, as it will become clear below, when we will present more examples of well-typed and non-well-typed expressions. (4) The presented type system is decidable (expressions are of finite length and the system is recursive).

We can now formally define the set of G -expressions: well-typed expressions associated with a polynomial functor G .

Definition 1 (G -expressions). *Let G be a polynomial functor and F an ingredient of G . We denote by $Exp_{F \triangleleft G}$ the following set:*

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

We define the set Exp_G of well-typed G -expressions by $Exp_{G \triangleleft G}$.

For the functor D , examples of well-typed expressions include $r(a(0))$, $l(1) \oplus r(a(l(0)))$ and $\mu x.r(a(x)) \oplus l(1)$. The expressions $l[1]$, $l(1) \oplus 1$ and $\mu x.1$ are examples of non well-typed expressions, because the functor D does not involve $+$, the subexpressions in the sum have different type, and recursion is not at the outermost level (1 has type $2 \triangleleft D$), respectively.

Let us instantiate the definition of expressions to the functors of deterministic automata $D = 2 \times Id^A$ and partial automata $P = (1 + Id)^A$.

Example 2 (Deterministic expressions). Let A be a finite set of input actions and let X be a set of (recursion or) fixpoint variables. The set of *deterministic expressions* is given by the following BNF syntax. For $a \in A$ and $x \in X$:

$$\varepsilon ::= \emptyset \mid x \mid r(a(\varepsilon)) \mid l(1) \mid l(0) \mid \varepsilon \oplus \varepsilon \mid \mu x.\varepsilon$$

where ε is closed and occurrences of fixpoint variables are within the scope of an input action.

It is easy to see that the closed (and guarded) expressions generated by the grammar presented above are exactly the elements of Exp_D . One can easily see that $l(1)$ and $l(0)$ are well-typed expressions for $D = 2 \times Id^A$ because both 1 and 0 are of type $2 \triangleleft D$. For the expression $r(a(\varepsilon))$ note that $a(\varepsilon)$ has type $Id^A \triangleleft D$ as long as ε has type $Id \triangleleft D$. And the crucial remark here is that, by definition of \vdash , $Exp_{Id \triangleleft G} = Exp_G$. Intuitively, this can be 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:



Therefore, ε has type $Id \triangleleft D$ if it is of type $D \triangleleft D$, or more precisely, if $\varepsilon \in Exp_D$, which explains why the grammar above is correct.

At this point, we should remark that the syntax of our expressions differs from the classical regular expressions in the use of μ and action prefixing $a(\varepsilon)$ instead of star and full concatenation. We shall prove later that these two syntactically different formalisms are equally expressive (Theorems 5 and 6).

Without additional explanation we present next the syntax for the expressions in Exp_P .

Example 3 (Partial automata expressions). Let A be a finite set of input actions and X be a set of (recursion or) fixpoint variables. The set 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 ε is closed and occurrences of fixpoint variables are within the scope of an input action. For simplicity, $a\uparrow$ and $a(\varepsilon)$ abbreviate $a(l[*])$ and $a(r[\varepsilon])$.

We have now defined a language of expressions which gives us an algebraic description of systems. In the remainder of the paper, we want to present a generalization of Kleene's theorem for polynomial coalgebras (Theorems 5 and 6). Recall that, for regular languages, the theorem states that a language is regular if and only if it is recognized by a finite automaton.

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 $\lambda_G : \text{Exp}_G \rightarrow G(\text{Exp}_G)$. We proceed by induction on the ingredients of G . More precisely we are going to define a function

$$\lambda_{F \triangleleft G} : \text{Exp}_{F \triangleleft G} \rightarrow F(\text{Exp}_G)$$

and then set $\lambda_G = \lambda_{G \triangleleft G}$. Our definition of the function $\lambda_{F \triangleleft G}$ will make use of the following.

- (i) We define a constant $\text{Empty}_{F \triangleleft G} \in F(\text{Exp}_G)$ by induction on the syntactic structure of F :

$$\begin{aligned} \text{Empty}_{\text{Id} \triangleleft G} &= \emptyset \\ \text{Empty}_{B \triangleleft G} &= \perp_B \\ \text{Empty}_{F_1 \times F_2 \triangleleft G} &= \langle \text{Empty}_{F_1 \triangleleft G}, \text{Empty}_{F_2 \triangleleft G} \rangle \\ \text{Empty}_{F_1 + F_2 \triangleleft G} &= \perp \\ \text{Empty}_{F^A \triangleleft G} &= \lambda a. \text{Empty}_{F \triangleleft G} \end{aligned}$$

- (ii) We define $\text{Plus}_{F \triangleleft G} : F(\text{Exp}_G) \times F(\text{Exp}_G) \rightarrow F(\text{Exp}_G)$ by induction on the syntactic structure of F :

$$\begin{aligned} \text{Plus}_{\text{Id} \triangleleft G}(\varepsilon_1, \varepsilon_2) &= \varepsilon_1 \oplus \varepsilon_2 \\ \text{Plus}_{B \triangleleft G}(b_1, b_2) &= b_1 \vee_B b_2 \\ \text{Plus}_{F_1 \times F_2 \triangleleft G}(\langle \varepsilon_1, \varepsilon_2 \rangle, \langle \varepsilon_3, \varepsilon_4 \rangle) &= \langle \text{Plus}_{F_1 \triangleleft G}(\varepsilon_1, \varepsilon_3), \text{Plus}_{F_2 \triangleleft G}(\varepsilon_2, \varepsilon_4) \rangle \\ \text{Plus}_{F_1 + F_2 \triangleleft G}(\kappa_i(\varepsilon_1), \kappa_i(\varepsilon_2)) &= \kappa_i(\text{Plus}_{F_i \triangleleft G}(\varepsilon_1, \varepsilon_2)), \quad i \in \{1, 2\} \\ \text{Plus}_{F_1 + F_2 \triangleleft G}(\kappa_i(\varepsilon_1), \kappa_j(\varepsilon_2)) &= \top \quad i, j \in \{1, 2\} \text{ and } i \neq j \\ \text{Plus}_{F_1 + F_2 \triangleleft G}(x, \top) &= \text{Plus}_{F_1 + F_2 \triangleleft G}(\top, x) = \top \\ \text{Plus}_{F_1 + F_2 \triangleleft G}(x, \perp) &= \text{Plus}_{F_1 + F_2 \triangleleft G}(\perp, x) = x \\ \text{Plus}_{F^A \triangleleft G}(f, g) &= \lambda a. \text{Plus}_{F \triangleleft G}(f(a), g(a)) \end{aligned}$$

Now we have all we need to define $\lambda_{F \triangleleft G}$. This function will be defined by double induction on the maximum number $N(\varepsilon)$ of nested unguarded occurrences of μ -expressions in ε and on the length of the proofs for typing expressions. We define $N(\varepsilon)$ as follows:

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

For every ingredient F of a polynomial functor G and expression $\varepsilon \in \text{Exp}_{F \triangleleft G}$, $\lambda_{F \triangleleft G}(\varepsilon)$ is defined as follows:

$$\begin{aligned} \lambda_{F \triangleleft G}(\emptyset) &= \text{Empty}_{F \triangleleft G} \\ \lambda_{F \triangleleft G}(\varepsilon_1 \oplus \varepsilon_2) &= \text{Plus}_{F \triangleleft G}(\lambda_{F \triangleleft G}(\varepsilon_1), \lambda_{F \triangleleft G}(\varepsilon_2)) \\ \lambda_{G \triangleleft G}(\mu x.\varepsilon) &= \lambda_{G \triangleleft G}(\varepsilon[\mu x.\varepsilon/x]) \\ \lambda_{Id \triangleleft G}(\varepsilon) &= \varepsilon \quad \text{for } G \neq Id \\ \lambda_{B \triangleleft G}(b) &= b \\ \lambda_{F_1 \times F_2 \triangleleft G}(l(\varepsilon)) &= \langle \lambda_{F_1 \triangleleft G}(\varepsilon), \text{Empty}_{F_2 \triangleleft G} \rangle \\ \lambda_{F_1 \times F_2 \triangleleft G}(r(\varepsilon)) &= \langle \text{Empty}_{F_1 \triangleleft G}, \lambda_{F_2 \triangleleft G}(\varepsilon) \rangle \\ \lambda_{F_1 + F_2 \triangleleft G}(l[\varepsilon]) &= \kappa_1(\lambda_{F_1 \triangleleft G}(\varepsilon)) \\ \lambda_{F_1 + F_2 \triangleleft G}(r[\varepsilon]) &= \kappa_2(\lambda_{F_2 \triangleleft G}(\varepsilon)) \\ \lambda_{F^A \triangleleft G}(a(\varepsilon)) &= \lambda a'. \begin{cases} \lambda_{F \triangleleft G}(\varepsilon) & a = a' \\ \text{Empty}_{F \triangleleft G} & \text{otherwise} \end{cases} \end{aligned}$$

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

In order to see that the definition of $\lambda_{F \triangleleft G}$ is well-formed, note the interplay between the two inductions: the length of the typing proof of the arguments in the recursive calls is strictly decreasing, except in the case of $\mu x.\varepsilon$; but, in this case we have that $N(\varepsilon) = N(\varepsilon[\mu x.\varepsilon/x])$, which can easily be proved by (standard) induction on the syntactic structure of ε , since ε is guarded (in x), and it guarantees that $N(\varepsilon[\mu x.\varepsilon/x]) < N(\mu x.\varepsilon)$. Also note that clause 4 of the above definition overlaps with clauses 1 and 2 (by taking $F = Id$). However, they give the same result and thus the definition is correct.

Definition 4. We can now define, for each polynomial functor G , a G -coalgebra

$$\lambda_G : \text{Exp}_G \rightarrow G(\text{Exp}_G)$$

by putting $\lambda_G = \lambda_{G \triangleleft G}$.

This means that we can define the subcoalgebra generated by an expression $\varepsilon \in \text{Exp}_G$, by repeatedly applying λ_G , which seems to be the correspondent of half of Kleene's theorem — the language represented by a given regular expression can be recognized by a finite state automaton. However, it is important to remark that the subcoalgebra generated by an expression $\varepsilon \in \text{Exp}_G$ by repeatedly applying λ_G is, in general, infinite. Take for instance the deterministic expression $\varepsilon_1 = \mu x. r(a(x \oplus \mu y. r(a(y))))$ and observe that:

$$\begin{aligned} \lambda_D(\varepsilon_1) &= \langle 0, \varepsilon_1 \oplus \mu y. r(a(y)) \rangle \\ \lambda_D(\varepsilon_1 \oplus \mu y. r(a(y))) &= \langle 0, \varepsilon_1 \oplus \mu y. r(a(y)) \oplus \mu y. r(a(y)) \rangle \\ &\vdots \end{aligned}$$

As one would expect, all the new states are bisimilar and can be identified. However, the function λ_D does not make any state identification and thus yields an infinite coalgebra.

The observation that the set of expressions has a coalgebra structure will be crucial for the proof of the generalized Kleene theorem, as will be shown in the next two sections.

4 Expressions Are Expressive

Having a G -coalgebra structure on Exp_G has two advantages. First, it provides us, by finality, directly with a natural semantics because of the existence of a (unique) homomorphism $\llbracket \cdot \rrbracket : Exp_G \rightarrow \Omega_G$, that assigns to every expression ε an element $\llbracket \varepsilon \rrbracket$ of the final coalgebra Ω_G .

The second advantage of the coalgebra structure on Exp_G is that it lets us use the notion of G -bisimulation to relate G -coalgebras (S, g) and expressions $\varepsilon \in Exp_G$. If one can construct a bisimulation relation between an expression ε and a state s of a given coalgebra, then the behaviour represented by ε is equal to the behaviour determined by the transition structure of the coalgebra applied to the state s . This is the analogue of computing the language $L(r)$ represented by a given regular expression r and the language $L(s)$ accepted by a state s of a finite state automaton and checking whether $L(r) = L(s)$.

The following theorem states that the every state in a finite G -coalgebra can be represented by an expression in our language. This generalizes *half* of Kleene's theorem: if a language is accepted by a finite automaton then it is regular. The generalization of the other *half* of the theorem (if a language is regular then it is accepted by a finite automaton) will be presented in Section 5.

Theorem 5. *Let G be a polynomial functor and (S, g) a G -coalgebra. If S is finite then there exists for any $s \in S$ an expression $\varepsilon_s \in Exp_G$ such that $\varepsilon_s \sim s$ (which implies $\llbracket \varepsilon_s \rrbracket = \llbracket s \rrbracket$).*

Proof. We construct, for a state $s \in S$, an expression $\varepsilon_s \sim s$. If $G = Id$, $\varepsilon_s = \emptyset$. Otherwise we proceed in the following way. Let $S = \{s_1, s_2, \dots, s_n\}$, where $s_1 = s$. We associate with each state s_i a variable $x_i \in X$ and an equation $\varepsilon_i = \mu x_i. \gamma_{g(s_i)}^G$, where $\gamma_{g(s_i)}^G$ is defined as follows. For $F \triangleleft G$ and $s' \in FS$, the expression $\gamma_{s'}^F \in Exp_{F \triangleleft G}$ is defined by induction on the structure of F :

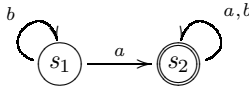
$$\begin{aligned} \gamma_s^{Id} &= x_s & \gamma_b^B &= b \\ \gamma_{(s, s')}^{F_1 \times F_2} &= l(\gamma_s^{F_1}) \oplus r(\varepsilon_{s'}^{F_2}) \\ \gamma_{\kappa_1(s)}^{F_1 + F_2} &= l[\gamma_s^{F_1}] & \gamma_{\kappa_2(s)}^{F_1 + F_2} &= r[\gamma_s^{F_2}] \\ \gamma_{\perp}^{F_1 + F_2} &= \emptyset & \gamma_{\top}^{F_1 + F_2} &= l[\emptyset] \oplus r[\emptyset] \\ \gamma_f^{F^A} &= \bigoplus_{a \in A} a(\gamma_{f(a)}^F) \end{aligned}$$

Note that the choice of $l[\emptyset] \oplus r[\emptyset]$ to represent inconsistency is arbitrary but *canonical*, in the sense that any other expression involving sum of $l[\varepsilon_1]$ and $r[\varepsilon_2]$ will be bisimilar.

Next, we eliminate all free occurrences of x_1, \dots, x_n from the system of equations $\varepsilon_1 = \mu x_1. \gamma_{g(s_1)}^G, \dots, \varepsilon_n = \mu x_n. \gamma_{g(s_n)}^G$ by first replacing x_n by ε_n in the equations for $\varepsilon_1, \dots, \varepsilon_{n-1}$. Next, we replace x_{n-1} by ε_{n-1} in the equations for $\varepsilon_1, \dots, \varepsilon_{n-2}$. Continuing in this way, we end up with an equation $\varepsilon_1 = \varepsilon$, where ε no longer contains any free variable. We then take $\varepsilon_s = \varepsilon$.

Moreover, $s \sim \varepsilon_s$, because the relation $R_G = \{ \langle \varepsilon_s, s \rangle \mid s \in S \}$ is a bisimulation (for every functor G). Due to space restrictions we omit the proof of this fact, which can be found in [2].

Let us illustrate the construction above by some examples. Consider the following deterministic automaton over a two letter alphabet $A = \{a, b\}$, whose transition function is depicted in the following picture (\odot represents that the state s is final):



Now define $\varepsilon_1 = \mu x_1. \varepsilon$ and $\varepsilon_2 = \mu x_2. \varepsilon'$ where

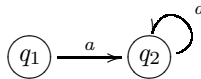
$$\varepsilon = l(0) \oplus r(b(x_1) \oplus a(x_2)) \quad \varepsilon' = l(1) \oplus r(a(x_2) \oplus b(x_2))$$

Substituting x_2 by ε_2 in ε_1 then yields

$$\varepsilon_1 = \mu x_1. l(0) \oplus r(b(x_1) \oplus a(\varepsilon_2)) \quad \varepsilon_2 = \mu x_2. l(1) \oplus r(a(x_2) \oplus b(x_2))$$

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

As another example, take the following partial automaton, also over a two letter alphabet $A = \{a, b\}$:



In the graphical representation of a partial automaton (S, p) we omit transitions for which $p(s)(a) = \kappa_1(*)$. In this case, this happens for both states for the input letter b .

We define $\varepsilon_1 = \mu x_1. \varepsilon$ and $\varepsilon_2 = \mu x_2. \varepsilon'$ where $\varepsilon = \varepsilon' = b\uparrow \oplus a(x_2)$. Substituting x_2 by ε_2 in ε_1 then yields

$$\varepsilon_1 = \mu x_1. b\uparrow \oplus a(\varepsilon_2) \quad \varepsilon_2 = \mu x_2. b\uparrow \oplus a(x_2)$$

Again we have $s_1 \sim \varepsilon_1$ and $s_2 \sim \varepsilon_2$.

5 Finite Systems for Expressions

We now give a construction to prove the converse of Theorem 5, that is, we describe a synthesis process that produces a *finite* G -coalgebra from an arbitrary regular G -expression ε . The states of the resulting G -coalgebra will consist of a finite subset of expressions, including an expression ε' such that $\varepsilon \sim_G \varepsilon'$.

5.1 Formula Normalization

We saw in Section 3.1 that the set of expressions has a coalgebra structure. We observed however that the subcoalgebra generated by an expression is in general infinite.

In order to guarantee the termination of the synthesis process we need to identify some expressions. In fact, as we will formally show later, it is enough to identify expressions that are provably equivalent using only the following axioms:

$$\begin{aligned}
 (\textit{Idempotency}) \quad & \varepsilon \oplus \varepsilon = \varepsilon \\
 (\textit{Commutativity}) \quad & \varepsilon_1 \oplus \varepsilon_2 = \varepsilon_2 \oplus \varepsilon_1 \\
 (\textit{Associativity}) \quad & \varepsilon_1 \oplus (\varepsilon_2 \oplus \varepsilon_3) = (\varepsilon_1 \oplus \varepsilon_2) \oplus \varepsilon_3 \\
 (\textit{Empty}) \quad & \emptyset \oplus \varepsilon = \varepsilon
 \end{aligned}$$

This group of axioms gives to the set of expressions the structure of a join-semilattice. One easily shows that if two expressions are provably equivalent using these axioms then they are bisimilar (soundness).

For instance, it is easy to see that the deterministic expressions

$$r(a(\emptyset)) \oplus l(1) \oplus \emptyset \oplus l(1) \text{ and } r(a(\emptyset)) \oplus l(1)$$

are equivalent using the equations (*Idempotency*) 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 $norm_G : Exp_G \rightarrow Exp_G$ encodes this procedure. We define it by induction on the expression structure as follows:

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

Here, the function *plus* takes a list of expressions $[\varepsilon_1, \dots, \varepsilon_n]$ and returns the expression $\varepsilon_1 \oplus \dots \oplus \varepsilon_n$ (*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:

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

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 deterministic expressions above results in the same expression: $r(a(\emptyset)) \oplus l(1)$.

Note that $norm_G$ only normalizes one level of the expression and 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 it does not influence termination. In the examples below, this situation will never occur.

5.2 Synthesis Procedure

Given an expression $\varepsilon \in Exp_G$ we will generate a finite G -coalgebra by applying repeatedly $\lambda_G : Exp_G \rightarrow Exp_G$ and normalizing the expressions obtained at each step. We will use the function Δ , which takes an expression $\varepsilon \in Exp_G$ and returns a G -coalgebra, and which is defined as follows:

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

Here, dom returns the domain of a finite function and D_G applies λ_G , starting with state $norm_G(\varepsilon)$, to the new states (after normalization) generated at each step, repeatedly, until all states in the coalgebra have their transition structure fully defined. The arguments of D_G are two sets of states: $sts \subseteq Exp_G$, the states that still need to be processed and $vis \subseteq Exp_G$, the states that already have been visited (synthesized). For each $\varepsilon \in sts$, D_G computes $\lambda_G(\varepsilon)$ and produces an intermediate transition function (possibly partial) by taking the union of all those $\lambda_G(\varepsilon)$. Then, it collects all new states appearing in this step, normalizing them, and recursively computes the transition function for those.

$$D_G(sts, vis) = \begin{cases} \emptyset & sts = \emptyset \\ trans \cup D_G(newsts, vis') & \text{otherwise} \end{cases}$$

where $trans = \{\langle \varepsilon, \lambda_G(\varepsilon) \rangle \mid \varepsilon \in sts\}$
 $sts' = collectStates_G(\pi_2(trans))$
 $vis' = sts \cup vis$
 $newsts = sts' \setminus vis'$

Here, $collectStates_G : GExp_G \rightarrow PExp_G$ is a function that collects and normalizes the G -expressions appearing in a *structured* state $\lambda_G(\varepsilon) \in GExp_G$. We can now formulate the converse of Theorem 5.

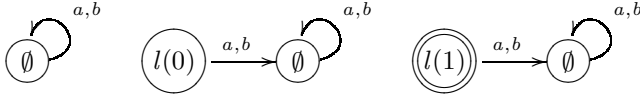
Theorem 6. *Let G be a polynomial functor. For every $\varepsilon \in Exp_G$, $\Delta_G(\varepsilon) = (S, g)$ is such that S is finite and there exists $s \in S$ with $\varepsilon \sim s$.*

Proof. First note that $\varepsilon \sim norm_G(\varepsilon)$ and $norm_G(\varepsilon) \in S$, by the definition of Δ_G and D_G . For space reasons, we omit the proof that S is finite, *i.e.* that $D_G(\{norm_G(\varepsilon)\}, \emptyset)$ terminates (details can be found in [2]).

5.3 Examples

In this subsection we will illustrate the synthesis algorithm presented above. For simplicity, we will consider deterministic and partial automata expressions over $A = \{a, b\}$.

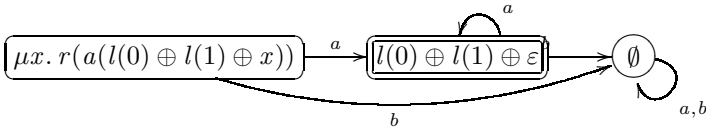
Let us start by showing the synthesised automata for the most simple deterministic expressions – \emptyset , $l(0)$ and $l(1)$.



It is interesting to make the parallel with the traditional regular expressions and remark that the first two automata recognize the empty language $\{\}$ and the last the language $\{\epsilon\}$ containing only the empty word.

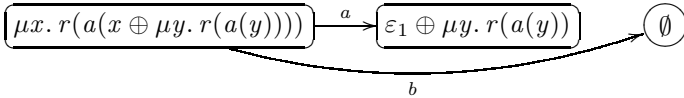
An important remark is that the automata generated are not minimal (for instance, the automata $l(0)$ and \emptyset are bisimilar). Our goal has been to generate a finite automaton from a regular expression. From this the minimal automaton can always be obtained by identifying bisimilar states.

For an example of an expression containing fixpoints, consider $\epsilon = \mu x. r(a(l(0) \oplus l(1) \oplus x))$. One can easily compute the synthesised automaton:

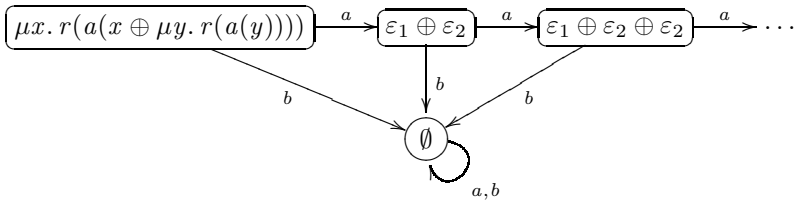


and observe that it recognizes the language aa^* . Here, the role of the join-semilattice structure is also visible: $l(0) \oplus l(1) \oplus \epsilon$ specifies that the current state is both final and non-final. Because $1 \vee 0 = 1$ the state is set to be final.

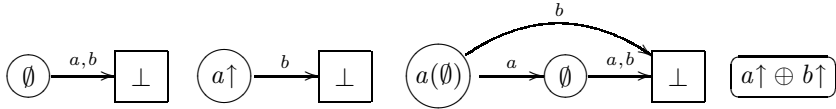
As a last example of deterministic expressions consider $\epsilon_1 = \mu x. r(a(x \oplus \mu y. r(a(y))))$. Applying λ_D to ϵ_1 one gets the following (partial) automaton:



Calculating $\lambda_D(\epsilon_1 \oplus \mu y. r(a(y)))(a)$ yields $\langle 0, \epsilon_1 \oplus \mu y. r(a(y)) \oplus \mu y. r(a(y)) \rangle$. When applying $collectStates_G$, the expression $\epsilon_1 \oplus \mu y. r(a(y)) \oplus \mu y. r(a(y))$ will be normalized to $\epsilon_1 \oplus \mu y. r(a(y))$, which is a state that already exists. Remark here the role of $norm$ in guaranteeing termination. Without normalization, one would get the following infinite coalgebra ($\epsilon_2 = \mu y. r(a(y))$):

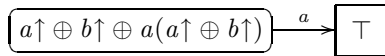


Let us now see a few examples of synthesis for partial automata expressions, where we will illustrate the role of \perp and \top . As before, let us first present the corresponding automata for simple expressions – \emptyset , $a\uparrow$, $a(\emptyset)$ and $a\uparrow \oplus b\uparrow$.



In the graphical representation of a partial automata (S, p) , whenever $g(s)(a) \in \{\perp, \top\}$ we represent a transition, but note that $\perp \notin S$ and $\top \notin S$ (thus, the square box) and have no defined transitions.

Here, one can now observe how \perp is used to encode underspecification, working as a kind of deadlock state. Note that in the first three expressions the behaviour for one or both of the inputs is missing, whereas in the last expression the specification is complete. The element \top is used to deal with inconsistent specifications. For instance, consider the expression $a\uparrow \oplus b\uparrow \oplus a(a\uparrow \oplus b\uparrow)$. All inputs are specified, but note that at the outermost level input a appears in two different sub-expressions – $a\uparrow$ and $a(a\uparrow \oplus b\uparrow)$ – specifying at the same time that input a leads to successful termination and that it leads to a state where $a\uparrow \oplus b\uparrow$ holds, which is contradictory, giving rise to the following automaton.



6 Conclusions

We have presented a generalization of Kleene’s theorem for polynomial coalgebras. More precisely, we have introduced a language of expressions for polynomial coalgebras and we have shown that they constitute a precise syntactic description of deterministic systems, in the sense that every expression in the language is bisimilar to a state of a finite coalgebra and vice-versa (Theorems 5 and 6).

The language of expressions presented in this paper can be seen as an alternative to the classical regular expressions for deterministic automata and to KAT expressions [11] and as a generalization of previous work of the authors on Mealy machines [3].

As was pointed out by the referees of the present paper, Theorem 5 is closely related to the well known fact that, for polynomial functors, an element in a finite subcoalgebra of the final coalgebra can be characterised as a “finite tree with loops”. This could in turn give rise to a different language of expressions $\varepsilon ::= x \mid \mu x. \varepsilon \mid \sigma(\varepsilon_1, \dots, \varepsilon_n)$, where σ is an n -ary operation symbol in the signature corresponding to a polynomial functor (e.g., if $GX = 1 + X + X + X^2$ then the signature has one constant, two unary and one binary operation symbol). This alternative approach might seem simpler than the one taken in this paper but does not provide an operator for combining specifications as our \oplus operator, and, more importantly, will not allow for an easy and modular axiomatization of bisimulation. Providing such a complete finite axiomatization generalizing the results presented in [9,5] is subject of our current research. This will provide a generalization of Kleene algebra to polynomial coalgebras.

Further, we would like to deal with non-deterministic systems (which amounts to include the powerset in our class of functors) and probabilistic systems.

In our language we have a fixpoint operator, $\mu x.\varepsilon$, and action prefixing, $a(\varepsilon)$, opposed to the use of star E^* and sequential composition E_1E_2 in classical regular expressions. We would like to study in more detail the precise relation between these two (equally expressive) syntactic formalisms. Ordinary regular expressions are closed under intersection and complement. We would like to study whether a similar result can be obtained for our language.

Coalgebraic modal logics (CML) [12] have been presented as a general theory for reasoning about transition systems. The connection between our language and CML is also subject of further study.

Acknowledgements. The authors are grateful to Dave Clarke, Helle Hansen, Clemens Kupke, Yde Venema and the anonymous referees for useful comments.

References

1. Aczel, P., Mendler, N.: A Final Coalgebra Theorem. In: Dybjer, P., Pitts, A.M., Pitt, D.H., Poigné, A., Rydeheard, D.E. (eds.) *Category Theory and Computer Science*. LNCS, vol. 389, pp. 357–365. Springer, Heidelberg (1989)
2. Bonsangue, M., Rutten, J., Silva, A.: Regular expressions for polynomial coalgebras. CWI Technical report E0703 (2007)
3. Bonsangue, M., Rutten, J., Silva, A.: Coalgebraic logic and synthesis of mealy machines. In: Amadio, R. (ed.) *FOSSACS 2008*. LNCS, vol. 4962, pp. 231–245. Springer, Heidelberg (2008)
4. Brzozowski, J.A.: Derivatives of regular expressions. *Journal of the ACM* 11(4), 481–494 (1964)
5. Ésik, Z.: Axiomatizing the equational theory of regular tree languages (extended abstract). In: Meinel, C., Morvan, M. (eds.) *STACS 1998*. LNCS, vol. 1373, pp. 455–465. Springer, Heidelberg (1998)
6. Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation*, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
7. Jacobs, B.: Many-sorted coalgebraic modal logic: a model-theoretic study. *ITA* 35(1), 31–59 (2001)
8. Kleene, S.: Representation of events in nerve nets and finite automata. *Automata Studies*, 3–42 (1956)
9. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: *Logic in Computer Science*, pp. 214–225 (1991)
10. Kozen, D.: *Automata and Computability*. Springer, New York (1997)
11. Kozen, D.: On the coalgebraic theory of Kleene algebra with tests. Technical Report, Computing and Information Science, Cornell University (March 2008), <http://hdl.handle.net/1813/10173>
12. Kurz, A.: Coalgebras and Their Logics. *SIGACT News* 37(2), 57–77 (2006)
13. Röbiger, M.: Coalgebras and modal logic. *ENTCS*, 33 (2000)
14. Rutten, J.: Universal coalgebra: a theory of systems. *TCS* 249(1), 3–80 (2000)
15. Rutten, J.: Automata and coinduction (an exercise in coalgebra). In: Sangiorgi, D., de Simone, R. (eds.) *CONCUR 1998*. LNCS, vol. 1466, pp. 194–218. Springer, Heidelberg (1998)