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)

Copyright © 2007, Stichting Centrum voor Wiskunde en Informatica
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 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.

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

Marcello Bonsangue Jan Rutten Alexandra Silva

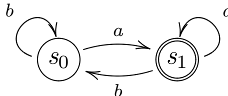
Abstract

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 [Kle56] 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 [Koz97, HMU06]. For instance, state s_0 of the automaton



accepts the same language as described by the expression $b^*a(a + bb^*a)^*$.

A deterministic automaton consists of a set of states S equipped with a transition function $\delta : S \to 2 \times S^A$ 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 $g : S \to GS$, where the functor G determines the type of the dynamic system under consideration and is the base of the theory of universal coalgebra [Rut00]. 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 [Rut98].

Given the fact that coalgebras can be seen as generalizations of deterministic automata, it is natural to investigate whether there exists an appropriate 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 will show how to generalize the notion of regular expression for G-transition systems. 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).

1.1 Related Work

Coalgebras of polynomial functors and the study of their specification languages has been investigated by many authors [Gol02, Röß00, Jac01]. Our approach is similar in spirit to that of Goldblatt [Gol02] 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 [Mos99, Röß00, Jac01, CP04, BK05, BK06, KV07]. Apart from [KV07], 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 [BK05] and it is similar to a fragment of the logic presented in [KV07]. 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 [PR89, TMS04, KV00, CGL^{+}00, HCR06]. 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 [TMS04] or μ-calculus [KV00]). Recently, this automata theoretical approach has been generalized in [KV07] 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 [Kle56] as a mathematical notation for describing languages recognized by deterministic finite automata. In [Rut98], 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 [Brz64]. 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 [Rut98] and the present paper is essentially the same. All in all, one can therefore say that standard regular expressions and their treatment in [Rut98] can be viewed as a special instance of the present approach. This is also the case for the generalization of the results in [Rut98] to automata on guarded strings [Koz08].

The present paper can be seen as a generalization of [BRS08], 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.

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

referees for pointing out some mistakes in an earlier version of this paper.

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 $X, Y, \ldots$ 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.

The coproduct, or disjoint union, of two sets X, Y is usually written as $X + Y$ with injections

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

and described pointwise as the set

$$X + Y = \{0\} \times X \cup \{1\} \times Y$$

We will use a extended notion of coproduct , which has two extra elements:

$$X \overline{+} Y = (X + Y) \cup \{\bot, \top\}$$

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 this extra elements will be clear when we present our language of expressions and concrete examples, in Section 5.3, of transition systems whose type involves coproduct. From this point onwards, we will only consider extended coproducts and, without any risk of confusion, we will refer to them simply as coproducts and denote them using $+$.

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 $\bot_B \in B$, 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:

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

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 $G : \mathbf{Set} \to \mathbf{Set}$, 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. $\Diamond$

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 automata* and *partial deterministic automata*.

Our definition of polynomial functors slightly differs from the one used in [Röß00, Jac01] in the use of a join-semilattice as constant functor. This small variation will play an important role in giving a full coalgebraic treatment of the language of expressions we shall later introduce. In fact, we will show that such a language (for this class of functors) is a coalgebra. Also note that this is not a real 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 shall use this relation later 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

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

◊

For example, 2, Id, $2 \times Id$, and D itself are all the ingredients of the deterministic automata functor D.

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 [Rut00]. Deterministic automata and partial automata are, respectively, coalgebras for the functors $D = 2 \times Id^A$ and $P = (1 + Id)^A$.

Definition 3 (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 notion of homomorphism instantiates in the case of the functors D and P mentioned above to the classical notion of automata homomorphism (morphisms preserving transitions and outputs).

Next, we define the notion of finality, which will play a key role later in providing a semantics to expressions.

Definition 4 (Final coalgebra) A G-coalgebra (S, f) is said to be *final* if for any G-coalgebra (T, g) there exists a unique homomorphism h which makes the following dia-

gram commute:

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

$\diamondsuit$

For every polynomial functor G there exists a final G-coalgebra (Ω_G, ω_G) [Rut00]. 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 $\pi : 2^{A^*} \to 2 \times (2^{A^*})^A$ defined as $\pi(\phi) = \langle \phi(\epsilon), \lambda a \lambda w.\phi(aw) \rangle$. Here ϵ denotes the empty sequence and aw denotes the word resulting from prefixing w with the letter a.

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 bisimulation.

Definition 5 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:

$$\begin{array}{lcl} \overline{Id}(R) & = & R \\ \overline{B}(R) & = & \{\langle b, b \rangle \mid b \in B\} \\ \overline{G_1 \times G_2}(R) & = & \{\langle \langle u_1, v_1 \rangle, \langle u_2, v_2 \rangle \rangle \mid \langle u_1, u_2 \rangle \in \overline{G_1}(R) \text{ and } \langle v_1, v_2 \rangle \in \overline{G_2}(R)\} \\ \overline{G_1 + G_2}(R) & = & \{\langle \kappa_1(u_1), \kappa_1(u_2) \rangle \mid \langle u_1, u_2 \rangle \in \overline{G_1}(R)\} \cup \\ & & \{\langle \kappa_2(u_1), \kappa_2(u_2) \rangle \mid \langle u_1, u_2 \rangle \in \overline{G_2}(R)\} \cup \{\langle \bot, \bot \rangle, \langle \top, \top \rangle\} \\ \overline{G^A}(R) & = & \{(f, g) \mid \forall_{a \in A} \langle f(a), g(a) \rangle \in \overline{G}(R)\} \end{array}$$

$\diamondsuit$

We can now define bisimulation, which will play an important role in providing semantics for our language of expressions.

Definition 6 (Bisimulation for G-coalgebras) Let (S, f) and (T, g) be two G-coalgebras. We call a relation $R \subseteq S \times T$ a *bisimulation* if for all $(s, t) \in S \times T$:

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

$\diamondsuit$

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.

Spelling out the definition of bisimulation for deterministic and partial automata one recovers the expected notions.

3 A language of expressions for polynomial coalgebras

In this section we introduce a language of expressions for coalgebras, generalizing the classical notion of regular expressions. 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 7 (Expressions) Let A be a finite set and let B be a finite join-semilattice. Furthermore, let X be 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:

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

$\Diamond$

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. 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 fixpoint and will play a similar role to the Kleene star in classical regular expressions for deterministic automata. 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 to each functor G the expressions $\varepsilon \in Exp$ that are valid specifications of G-coalgebras. The typing proceeds following the structure of the expressions and the ingredients of the functors.

Definition 8 (Typed expressions) We type *guarded* expressions ε using the ingredient relation, as follows:

$$\frac{}{\vdash \emptyset : F \lhd G} \qquad \frac{}{\vdash b : B \lhd G} \qquad \frac{}{\vdash x : G \lhd G}$$

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

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

$$\frac{\vdash \varepsilon : F_1 \lhd G}{\vdash l[\varepsilon] : F_1 + F_2 \lhd G} \qquad \frac{\vdash \varepsilon : F_2 \lhd G}{\vdash r[\varepsilon] : F_1 + F_2 \lhd G}$$

$\Diamond$

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 $F(\Omega_G)$. There is a rule for each expression construct. The extra rule involving $Id \lhd G$ reflects the isomorphism between the final coalgebra Ω_G and $G(\Omega_G)$. Remark that the type system allows only fixpoints at the outermost level of the functor. 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. Next we formally define the set of G-expressions: well-typed expressions associated with a polynomial functor G.

Definition 9 (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 (closed and guarded) well-typed G-*expressions* is $Exp_{G \lhd G}$. $\diamondsuit$

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 coproducts, the subexpressions in the sum have different type, and recursion is not at the outermost level (1 has type $2 \lhd D$), 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 10 (Deterministic expressions) Let A be a 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 $x \in X$:

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

where occurrences of fixpoint variables in ε are within the scope of an input action in A.

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 $2 \lhd D$. For the expression $r(a(\varepsilon))$ note that $a(\varepsilon)$ has type $Id^A \lhd D$ as long as ε has type $Id \lhd D$. And the crucial remark here is that, by definition of $\vdash$, $Exp_{Id \lhd G} = Exp_G$. Therefore, ε has type $Id \lhd D$ if it is of type $D \lhd D$, or more precisely, if $\varepsilon \in Exp_D$, which explains why the grammar above is correct.

Note that $Exp_{Id \lhd G} = Exp_G$ is a direct consequence of the isomorphism $\Omega_G \cong G(\Omega_G)$ mentioned above. 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:

$$G \quad = \quad \boxed{\cdots \quad Id \quad \cdots}$$

Below, we will simplify the notation for elements of Exp_D. Without any risk of confusion, 1 and 0 abbreviate, respectively, $l(1)$ and $l(0)$ and $a(\varepsilon)$ is used instead of $r(a(\varepsilon))$.

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

Example 11 (Partial automata expressions) Let A be a set of input actions and X be a set of (recursion or) fixpoint variables. The set *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 fixpoint variables in ε are within the scope of an input action in A.

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 15 and 16). 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

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

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 $\lambda_G = \lambda_{G \lhd G}$. Our definition of the function $\lambda_{F \lhd G}$ will make use of the following.

Definition 12 (i) We define a constant $Empty_{F \lhd G} \in F(Exp_G)$ by induction on the syntactic structure of F:

$$\begin{array}{l}
Empty_{Id \lhd G} = \emptyset \\
Empty_{B \lhd G} = \bot_B \\
Empty_{F_1 \times F_2 \lhd G} = \langle Empty_{F_1 \lhd G}, Empty_{F_2 \lhd G} \rangle \\
Empty_{F_1 + F_2 \lhd G} = \bot \\
Empty_{F^A \lhd G} = \lambda a. Empty_{F \lhd G}
\end{array}$$

(ii) We define $Plus_{F \lhd G} : F(Exp_G) \times F(Exp_G) \to F(Exp_G)$ by induction on the syntactic structure of F:

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

$\Diamond$

Now we have all we need to define $\lambda_{F \lhd G}$. 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 lenght of the proofs for typing expressions. We define $N(\varepsilon)$ as follows:

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

Definition 13 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:

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

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

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).

Definition 14 We can now define, for each polynomial functor G, a G-coalgebra

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

by defining $\lambda_G = \lambda_{G \lhd G}$. $\diamondsuit$

This means that we can define the subcoalgebra generated by an expression $\varepsilon \in Exp_G$, by repeatedly applying λ_G, which seems to be the correspondent of *half* of Kleene's theorem, which states that 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 $\varepsilon \in Exp_G$ by repeatedly applying λ_G is, in general, infinite. Take for instance the deterministic expression $\varepsilon_1 = \mu x.\, a(x \oplus \mu y.\, a(y))$ and observe that:

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

As one would expect, all these states are bisimilar. 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) homo-

morphism:

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

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

The second advantage of the coalgebra structure on 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 analogon of computing the language $L(r)$ represented by a given regular expression r and the language $L(s)$ accepted by a state s of finite state automaton and checking if $L(r) = L(s)$.

The following theorem states that the behaviour of every state in a finite G-coalgebra can be represented by an expression in our language. This generalizes *half* of Kleene's theorem for regular languages: 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 15 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 $\varepsilon_s \sim s$.

Proof. We associate with every state $s \in S$ a variable $x_s \in X$ and an expression $\varepsilon_s = \mu x_s.\, \varepsilon_s^G$ defined by induction on the structure of G as follows.

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

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.

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

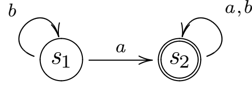
$$\begin{array}{l}
\varepsilon_s^{Id} = x_s \\
\varepsilon_b^{B} = b \\
\varepsilon_{\langle s, s'\rangle}^{G_1 \times G_2} = l(\varepsilon^{G_1}_{s}) \oplus r(\varepsilon^{G_2}_{s'}) \\
\varepsilon_{\kappa_1(s)}^{G_1 + G_2} = l[\varepsilon^{G_1}_{s}] \\
\varepsilon_{\kappa_2(s)}^{G_1 + G_2} = r[\varepsilon^{G_2}_{s}] \\
\varepsilon_{\bot}^{G_1 + G_2} = \emptyset \\
\varepsilon_{\top}^{G_1 + G_2} = l[\emptyset] \oplus r[\emptyset] \\
\varepsilon_f^{G^A} = \bigoplus_{a \in A} a(\varepsilon^G_{f(a)})
\end{array}$$

Syntactically replacing free occurrences of $x_{s'}$ in ε^G_s by $\varepsilon_{s'}$ ensures that all ε_s are in Exp_G. Moreover, $s \sim \varepsilon_s$, because, for every functor G, the relation

$$R_G = \{\langle \varepsilon_s, s\rangle \mid s \in S\}$$

is a bisimulation (for the proof see appendix A). □

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 ($(\!(s)\!)$ represents that the state s is final):



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

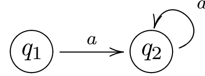
$$\varepsilon = 0 \oplus b(x_1) \oplus a(x_2) \qquad \varepsilon' = 1 \oplus a(x_2) \oplus b(x_2)$$

Substituting x_2 by ε_2 in ε_1 then yields

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

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

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

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 15, 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:

(*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*)	$\emptyset \oplus \varepsilon = \varepsilon$

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 expressions

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

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 $norm_G : Exp_G \to Exp_G$ encodes this procedure. We define it by induction on the expression structure as follows:

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

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:

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

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 deterministic expressions above results in the same expression – $a(\emptyset) \oplus \downarrow b$.

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 this 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 $\lambda_G : Exp_G \to Exp_G$ 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:

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

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.

$$D_G(sts, vis) = \begin{cases} \emptyset & sts = \emptyset \\ trans \cup D_G(newsts, vis') & otherwise \end{cases}$$
$$\begin{aligned} \textbf{where} \quad & trans = \{\langle \varepsilon, \lambda_G(\varepsilon)\rangle \mid \varepsilon \in sts\} \\ & sts' = collectStates_G(\pi_2(trans)) \\ & vis' = sts \cup vis \\ & newsts = sts' \setminus vis' \end{aligned}$$

Here, $collectStates_G = collectStates_{G \lhd G}$, where $collectStates_{F \lhd G}$ is defined by induction on the structure of F as follows:

$$\begin{aligned} & collectStates_{F \lhd G} : FExp_G \to \mathcal{P}Exp_G \\ & collectStates_{Id \lhd G}(s) = \{norm_G(s)\} \\ & collectStates_{B \lhd G}(b) = \{\} \\ & collectStates_{G_1 \times G_2 \lhd G}(\langle s_1, s_2\rangle) = collectStates_{G_1 \lhd G}(s_1) \cup collectStates_{G_2 \lhd G}(s_2) \\ & collectStates_{G_1 + G_2 \lhd G}(k_i(s)) = collectStates_{G_i \lhd G}(s) \qquad i \in \{1, 2\} \\ & collectStates_{G_1 + G_2 \lhd G}(\bot) = \{\} \\ & collectStates_{G_1 + G_2 \lhd G}(\top) = \{\} \\ & collectStates_{G_1^A \lhd G}(f) = \textstyle\bigcup_{a \in A} collectStates_{G_1 \lhd G}(f(a)) \end{aligned}$$

We can now formulate the converse of Theorem 15.

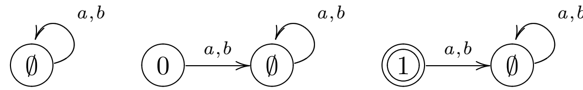
Theorem 16 Let G be a polynomial functor. For every $\varepsilon \in Exp_G$, $\Delta_G(\varepsilon) = (S, g)$ is such that S is finite and there exists $s \in S$ with $\varepsilon \sim s$.

Proof. First note that $\varepsilon \sim norm_G(\varepsilon)$ and $norm_G(\varepsilon) \in S$, by the definition of Δ_G and D_G. The proof that S is finite, *i.e.* that $D_G(\{norm_G(\varepsilon)\}, \emptyset)$ terminates, can be found in appendix B. □

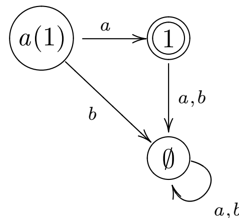
5.3 Examples

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

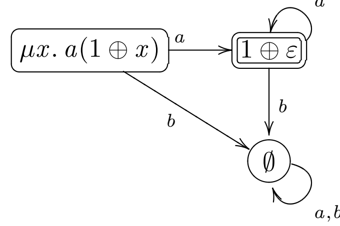
Let us start by showing the synthesised automata for the most simple deterministic expressions – $\emptyset$, 0 and 1.



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. The following automaton, generated from the expression $a(1)$, recognizes the language {a},



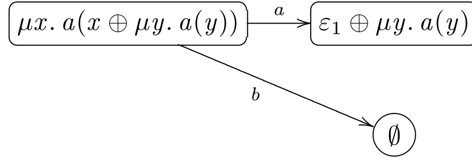
For an example of an expression containing fixpoints, consider $\varepsilon = \mu x.\, a(1 \oplus x)$. One can easily compute the synthesised automaton



and observe that it recognizes the language aa^*.

An important remark about these two last examples is that the automata generated are not minimal. 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.

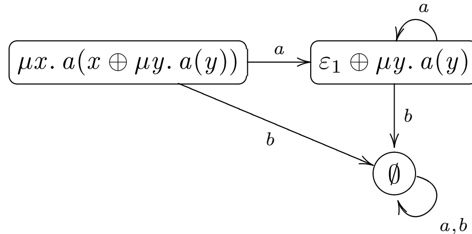
As a last example of deterministic expressions consider $\varepsilon_1 = \mu x.\, a(x \oplus \mu y.\, a(y))$. Applying λ_D to ε_1 one gets the following (partial) automaton:



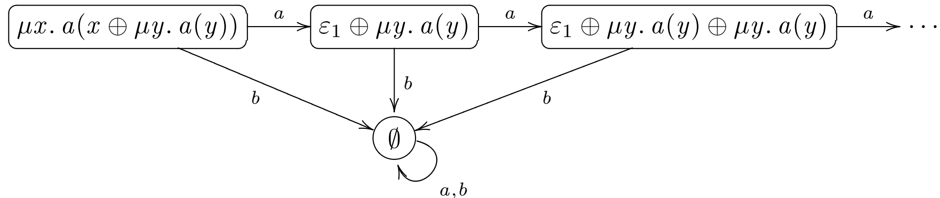
Calculating $\lambda_D(\varepsilon_1 \oplus \mu y.\, a(y))(a)$, we have:

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

When applying *collectStates*$_G$, the expression $\varepsilon_1 \oplus \mu y.\, a(y) \oplus \mu y.\, a(y)$ will be normalized to $\varepsilon_1 \oplus \mu y.\, a(y)$, which is a state that already exists. Remark here the role of *norm* in guaranteeing termination. As we saw in Section 3.1 only applying λ_D one would always generate syntactically different states which instead of the automata generated now:

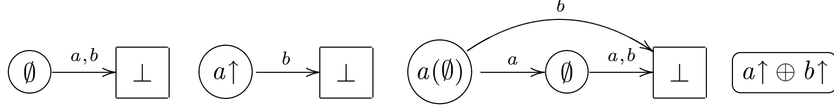


would lead to the following infinite coalgebra:



Let us now see a few examples of synthesis for partial automata expressions, where we will illustrate the role of $\bot$.

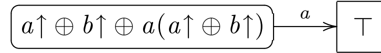
As before, let us first present the corresponding automata for simple expressions – $\emptyset$, $a\uparrow$, $a(\emptyset)$ and $a\uparrow \oplus b\uparrow$.



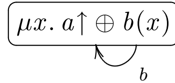
In the graphical representation of a partial automata (S, p), whenever $g(s)(a) \in \{\perp, \top\}$ we represent a transition, but note that $\perp \notin S$ and $\top \notin S$ (thus, the square box) and have no defined transitions.

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 on 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.



For an example with fixpoints take $\mu x.\, a\uparrow \oplus b(x)$, which generates the simple 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 (Theorem 15) and vice-versa (Theorem 16).

The work presented in this paper generalizes the classical Kleene theorem for deterministic automata, as well as previous work of the authors on Mealy machines [BRS08] and Kozen's recent results on coalgebraic theory of Kleene algebra with tests [Koz08].

Many questions remain to be answered. In particular one would like to be able to deal with non-deterministic systems (which amounts to include the powerset functor in our class of functors) and probabilistic systems.

Providing a complete finite axiomatization, generalizing results presented in [Koz91, É98] is also subject of current research. This will provide a generalization of Kleene algebra to polynomial coalgebras.

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.

Finally, the language of expressions introduced in this paper can be extended with a negation operator, as long as we consider constant functors to be Boolean algebras instead of join semilattices. The language obtained would have both least and greatest fixpoint operators and, as for the present set of 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 purposes.

References

[BK05] Marcello M. Bonsangue and Alexander Kurz. Duality for logics of transition systems. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 2005.

[BK06] Marcello M. Bonsangue and Alexander Kurz. Presenting functors by operations and equations. In Luca Aceto and Anna Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *LNCS*, pages 172–186. Springer, 2006.

[BRS08] Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. Coalgebraic logic and synthesis of Mealy machines. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2008.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[CGL+00] Edmund M. Clarke, Steven M. German, Yuan Lu, Helmut Veith, and Dong Wang. Executable protocol specification in esl. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.

[CP04] Corina Cîrstea and Dirk Pattinson. Modular construction of modal logics. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2004.

[É98] Zoltán Ésik. Axiomatizing the equational theory of regular tree languages (extended anstract). In *STACS '98: Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 455–465, London, UK, 1998. Springer-Verlag.

[Gol02] Robert Goldblatt. Equational logic of polynomial coalgebras. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Advances in Modal Logic 4*, pages 149–184. King's College Publications, 2002.

[HCR06] Helle Hvid Hansen, David Costa, and Jan J. M. M. Rutten. Synthesis of Mealy machines using derivatives. *Electronic Notes in Theoretical Computer Science*, 164(1):27–45, 2006.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Jac01] Bart Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *ITA*, 35(1):31–59, 2001.

[Kle56] Stephen Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–42, 1956.

[Koz91] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *Logic in Computer Science*, pages 214–225, 1991.

[Koz97] Dexter Kozen. *Automata and Computability.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[Koz08] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.

[KV00] Orna Kupferman and Moshe Y. Vardi. μ-calculus synthesis. In Mogens Nielsen and Branislav Rovan, editors, *MFCS'00*, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer, 2000.

[KV07] Clemens Kupke and Yde Venema. Coalgebraic automata theory: basic results. Technical Report SEN-E0701, CWI, The Netherlands, 2007.

[Mos99] Larry Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, 96, 1999.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL'89*, pages 179–190, 1989.

[Röß00] Martin Rößiger. Coalgebras and modal logic. *Electronic Notes in Theoretical Computer Science*, 33, 2000.

[Rut98] Jan J.M.M. 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.

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[TMS04] Simone Tini and Andrea Maggiolo-Schettini. Compositional synthesis of generalized Mealy machines. *Fundamenta Informaticae*, 60(1-4):367–382, 2004.

A Proof of Theorem 15 (cont.)

We prove, for all polynomial functors G and finite G-coalgebras (S, g), that the relation

$$R_G = \{\langle \varepsilon_s, s\rangle \mid s \in S\}$$

is a bisimulation. We have to show, for all $\langle \varepsilon_s, s\rangle \in R_G$, that $\langle \lambda_G(\varepsilon_1), g(s)\rangle \in \overline{G}(R_G)$.

The key observation is that the following relation

$$L_{F\lhd G} = \{\langle \lambda_{F\lhd G}(\varepsilon^F_{s'}[\varepsilon_q/x_q]), s'\rangle \mid s' \in FS\}$$

satisfies $L_{F\lhd G} = \overline{F}(R_G)$ and, for all $\langle \varepsilon_s, s\rangle \in R_G$, one has that $\langle \lambda_G(\varepsilon_s), g(s)\rangle \in L_G$ (as before, $L_G = L_{G\lhd G}$). Note that here $[\varepsilon_q/x_q]$ denotes the syntactic replacement of free ocurrences of x_q by ε_q, defined for all $q \in S$.

The proof that $L_{F\lhd G} = \overline{F}(R_G)$ follows easily by induction on the structure of F. Let us illustrate for the case $F = F_1 + F_2$.

$$L_{F_1+F_2\lhd G} = \{\langle \kappa_i(\lambda_{F_i\lhd G}(\varepsilon^{F_i}_{s'}[\varepsilon_q/x_q])), \kappa_i(s')\rangle \mid s' \in F_i(S)\} \cup \{\langle \bot, \bot\rangle, \langle \top, \top\rangle\} \qquad i \in 1, 2$$

Now, using the induction hypothesis, we know that $\langle \lambda_{F_i\lhd G}(\varepsilon^{F_i}_{s'}[\varepsilon_q/x_q]), s'\rangle \in \overline{F_i(R_G)}$ and therefore $L_{F_1+F_2} = \overline{F_1 + F_2}(R_G)$.

It remains to prove that for all $\langle \varepsilon_s, s\rangle \in R_G$, one has $\langle \lambda_G(\varepsilon_s), g(s)\rangle \in L_G$. In other words, we need to show that $\lambda_G(\varepsilon_s) = \lambda_G(\varepsilon^G_{g(s)}[\varepsilon_q/x_q])$. We prove it by induction on the structure of G. For $G = Id$, note that $\varepsilon_s = \mu x_s.\, \emptyset$ and we have:

$$\lambda_{Id}(\mu x_s.\, \emptyset) = \emptyset = \lambda_{Id}(\mu x_{g(s)}.\, \emptyset) = \lambda_{Id}(x_{g(s)}[\varepsilon_q/x_q]) = \lambda_{Id}(\varepsilon^{Id}_{g(s)}[\varepsilon_q/x_q])$$

Similarly, for $G = B$, $\varepsilon_s = \mu x_s.\, g(s)$ and

$$\lambda_B(\mu x_s.\, g(s)) = g(s) = g(s)[\varepsilon_q/x_q] = \lambda_B(g(s)[\varepsilon_q/x_q]) = \lambda_B(\varepsilon^B_{g(s)}[\varepsilon_q/x_q])$$

The remaining three cases are proven in a similar way. We will only show the proof for $G = G_1 + G_2$. If $g(s) = \bot$, then $\varepsilon_s = \emptyset$ and

$$\lambda_{G_1+G_2}(\mu x_s.\, \emptyset) = \lambda_{G_1+G_2}(\emptyset) = \lambda_{G_1+G_2}(\emptyset[\varepsilon_q/x_q]) = \lambda_{G_1+G_2}(\varepsilon^{G_1+G_2}_{\bot}[\varepsilon_q/x_q])$$

If $g(s) = \top$, then $\varepsilon_s = l[\emptyset] \oplus r[\emptyset]$ and

$$\lambda_{G_1+G_2}(\mu x_s.\, l[\emptyset] \oplus r[\emptyset]) = \lambda_{G_1+G_2}(l[\emptyset] \oplus r[\emptyset]) = \lambda_{G_1+G_2}(\varepsilon^{G_1+G_2}_{\top}[\varepsilon_q/x_q])$$

If $g(s) = k_1(s')$, then $\varepsilon_s = \mu x_s.\, l[\varepsilon^{G_1}_{s'}]$ and

$$\lambda_{G_1+G_2}(\mu x_s.\, l[\varepsilon^{G_1}_{s'}]) = \lambda_{G_1+G_2}(l[\varepsilon^{G_1}_{s'}][\varepsilon_s/x_s]) = \lambda_{G_1+G_2}(\varepsilon^{G_1+G_2}_{\kappa_1(s')}[\varepsilon_q/x_q])$$

Similarly for $g(s) = k_2(s')$.

B Proof of termination

We prove that our synthesis algorithm delivers a finite coalgebra for every regular expression, *i.e.* the function $D(\{\varepsilon\}, \emptyset)$ terminates (Theorem 23) . In order to prove this, we will show that all states generated during the synthesis process, starting with ε, are contained in a finite set (Theorem 22).

Definition 17 Given $\varepsilon \in \mathit{Exp}$, define the set $cl(\varepsilon)$ to be the smallest set satisfying:

$$\begin{array}{lcl} cl(\emptyset) & = & \{\emptyset\} \\ cl(\varepsilon_1 \oplus \varepsilon_2) & = & \{\varepsilon_1 \oplus \varepsilon_2\} \cup cl(\varepsilon_1) \cup cl(\varepsilon_2) \\ cl(\mu x.\, \varepsilon_1) & = & \{\mu x.\, \varepsilon_1\} \cup cl(\varepsilon_1[\mu x.\, \varepsilon_1/x]) \\ cl(l(\varepsilon_1)) & = & \{l(\varepsilon_1)\} \cup cl(\varepsilon_1) \\ cl(r(\varepsilon_1)) & = & \{r(\varepsilon_1)\} \cup cl(\varepsilon_1) \\ cl(l[\varepsilon_1]) & = & \{l[\varepsilon_1]\} \cup cl(\varepsilon_1) \\ cl(r[\varepsilon_1]) & = & \{r[\varepsilon_1]\} \cup cl(\varepsilon_1) \\ cl(a(\varepsilon_1)) & = & \{a(\varepsilon_1)\} \cup cl(\varepsilon_1) \end{array}$$

$\Diamond$

Note that $\varepsilon \in cl(\varepsilon)$. We have to prove two other properties about this set, which we will use later when proving termination.

Theorem 18 Let $\varepsilon, \varepsilon' \in \mathit{Exp}$. Then:

(i) $\varepsilon' \in cl(\varepsilon) \Rightarrow cl(\varepsilon') \subseteq cl(\varepsilon)$

(ii) Given an expression $\varepsilon \in \mathit{Exp}$, the set $cl(\varepsilon)$ is finite.

Proof. The theorem follows easily by double induction on the maximum number $N(\varepsilon)$ (as defined in Section 3.1) of nested unguarded occurrences of μ-expressions in ε and on the syntactic structure of ε. We treat a few selected cases.

For (i), we look at the cases when $N(\varepsilon) \leq k$ and $\varepsilon = \varepsilon_1 \oplus \varepsilon_2$ or $\varepsilon = \mu x.\, \varepsilon_1$. For the first case, suppose $\varepsilon' \in cl(\varepsilon_1 \oplus \varepsilon_2)$. Then, either $\varepsilon' = \varepsilon_1 \oplus \varepsilon_2$, and the result follows trivially, or $\varepsilon' \in cl(\varepsilon_i)$, $i \in \{1, 2\}$, and the result follows by induction and the fact that $cl(\varepsilon_i) \subseteq cl(\varepsilon_1 \oplus \varepsilon_2)$. For the second case, suppose that $\varepsilon' \in cl(\mu x.\, \varepsilon_1)$. Then, either $\varepsilon' = \mu x.\, \varepsilon_1$, and the result follows trivially, or $\varepsilon' \in cl(\varepsilon_1[\mu x.\, \varepsilon_1/x])$ and, by induction, $cl(\varepsilon') \subseteq cl(\varepsilon_1[\mu x.\, \varepsilon_1/x])$. Thus, $cl(\varepsilon') \subseteq cl(\mu x.\, \varepsilon_1)$.

For (ii), let us show the cases $N(\varepsilon) \leq k$ and $\varepsilon = a(\varepsilon_1)$ or $\varepsilon = \mu x.\, \varepsilon_1$. Both cases follow trivially by induction, since the induction hypothesis state that $cl(\varepsilon_1)$ and $cl(\varepsilon_1[\mu x.\, \varepsilon_1/x])$ are finite, respectively.

$\square$

Definition 19 Given a polynomial functor G and $\varepsilon \in \mathit{Exp}_G$, we define $cl_G(\varepsilon)$ by:

$$cl_G(\varepsilon) = \{\varepsilon' \in cl(\varepsilon) \mid \varepsilon' \in \mathit{Exp}_G\}$$

$\Diamond$

It is easy to see that $cl_G(\varepsilon)$ inherits the properties of $cl(\varepsilon)$: $\varepsilon \in cl_G(\varepsilon)$; if $\varepsilon' \in cl_G(\varepsilon)$ then $cl_G(\varepsilon') \subseteq cl_G(\varepsilon)$; and $cl_G(\varepsilon)$ is finite.

Theorem 20 Let $\varepsilon \in \mathit{Exp}_G$. Then, $\mathit{norm}_G(\varepsilon) = \varepsilon_1 \oplus \cdots \oplus \varepsilon_k$, with all ε_i distinct and $\varepsilon_i \in cl_G(\varepsilon)$.

Proof. By induction on the structure of ε. The result follows directly from the definition of $norm_G$. In fact, all the cases, apart from $\varepsilon_1 \oplus \varepsilon_2$ follow trivially because $\varepsilon \in cl_G(\varepsilon)$.

For $\varepsilon = \varepsilon_1 \oplus \varepsilon_2$, we have

$$norm_G(\varepsilon_1 \oplus \varepsilon_2) = plus(rem(flatten(norm_G(\varepsilon_1) \oplus norm_G(\varepsilon_2))))$$

Applying the induction hypothesis we see that $norm_G(\varepsilon_1) = \gamma_1 \oplus \cdots \oplus \gamma_k$, with all γ_i distinct and $\gamma_i \in cl_G(\varepsilon_1)$. Similar for ε_2. But $cl_G(\varepsilon_1) \subseteq cl_G(\varepsilon)$ and $cl_G(\varepsilon_2) \subseteq cl_G(\varepsilon)$. Therefore, applying *flatten* and *rem* to this expression we know that the argument list of *plus* will only have distinct elements of $cl_G(\varepsilon)$ and thus $norm_G(\varepsilon_1 \oplus \varepsilon_2) = \varepsilon'_1 \oplus \cdots \oplus \varepsilon'_k$, with all ε'_i distinct and $\varepsilon'_i \in cl(\varepsilon)$.

□

Theorem 21 Let $\varepsilon_1, \varepsilon_2 \in Exp_G$. Then:

$$collectStates_{F \lhd G}(Plus_{F \lhd G}(\varepsilon_1, \varepsilon_2)) \subseteq \{norm_G(\bigoplus_{\varepsilon \in X} \varepsilon) \mid X \in \mathcal{P}(csts_1 \cup csts_2)\}$$

where $csts_i = collectStates_{F \lhd G}(\varepsilon_i)$ $(i \in \{1, 2\})$.

Proof. The proof follows by induction on the structure of F. For $F = Id$, we have:

$$\begin{array}{rcl} collectStates_{Id \lhd G}(Plus_{Id \lhd G}(\varepsilon_1, \varepsilon_2)) & = & collectStates_{Id \lhd G}(\varepsilon_1 \oplus \varepsilon_2) \\ & = & \{norm_G(\varepsilon_1 \oplus \varepsilon_2)\} \\ & = & \{norm_G(norm_G(\varepsilon_1) \oplus norm_G(\varepsilon_2))\} \\ & \subseteq & \{norm_G(\bigoplus_{\varepsilon \in X} \varepsilon) \mid X \in \mathcal{P}(\{norm_G(\varepsilon_1), norm_G(\varepsilon_2)\})\} \end{array}$$

This result follows because $collectStates_{Id \lhd G}(\varepsilon_i) = \{norm_G(\varepsilon_i)\}$.

For $F = F_1 \times F_2$, we calculate:

$$\begin{array}{rl} & collectStates_{F_1 \times F_2 \lhd G}(Plus_{F_1 \times F_2 \lhd G}(\langle \varepsilon_1, \varepsilon_2 \rangle, \langle \varepsilon_3, \varepsilon_4 \rangle)) \\ = & collectStates_{F_1 \times F_2 \lhd G}(\langle Plus_{F_1 \lhd G}(\langle \varepsilon_1, \varepsilon_3 \rangle), Plus_{F_2 \lhd G}(\langle \varepsilon_2, \varepsilon_4 \rangle) \rangle) \\ = & collectStates_{F_1 \lhd G}(\langle Plus_{F_1 \lhd G}(\langle \varepsilon_1, \varepsilon_3 \rangle)) \cup collectStates_{F_2 \lhd G}(Plus_{F_2 \lhd G}(\langle \varepsilon_2, \varepsilon_4 \rangle) \rangle) \\ \subseteq & \{norm_G(\bigoplus_{\varepsilon \in X} \varepsilon) \mid X \in \mathcal{P}(csts_1 \cup csts_3)\} \cup \{norm_G(\bigoplus_{\varepsilon \in X} \varepsilon) \mid X \in \mathcal{P}(csts_2 \cup csts_4)\} \\ \subseteq & \{norm_G(\bigoplus_{\varepsilon \in X} \varepsilon) \mid X \in \mathcal{P}(csts_1 \cup csts_2 \cup csts_3 \cup csts_4)\} \end{array}$$

The result follows since

$$collectStates_{F_1 \times F_2 \lhd G}(\langle \varepsilon_1, \varepsilon_2 \rangle) \cup collectStates_{F_1 \times F_2 \lhd G}(\langle \varepsilon_3, \varepsilon_4 \rangle) = csts_1 \cup csts_2 \cup csts_3 \cup csts_4$$

The remaining cases are proven in a similar way. □

Theorem 22 Let $\varepsilon \in Exp_G$. Then, $collectStates_{F \lhd G}(\lambda_{F \lhd G}(\varepsilon)) \subseteq \{\varepsilon_1 \oplus \cdots \oplus \varepsilon_k \mid$ *with all* ε_i *distinct*$, \varepsilon_i \in cl_G(\varepsilon)\}$.

Proof. Follows easily by double induction on maximum number $N(\varepsilon)$ of nested unguarded occurrences of μ-expressions and on the length of proofs for typing expressions. We treat only a few cases. For $N(\varepsilon) = 0$ consider the case of $\varepsilon = \varepsilon_1 \oplus \varepsilon_2$.

$$\begin{array}{rll} & collectStates_{F \lhd G}(\lambda_{F \lhd G}(\varepsilon_1 \oplus \varepsilon_2)) & \\ = & collectStates_{F \lhd G}(Plus_{F \lhd G}(\lambda_{F \lhd G}(\varepsilon_1), \lambda_{F \lhd G}(\varepsilon_2))) & (\text{def. } \lambda_{F \lhd G}) \\ \subseteq & \{norm_G(\oplus_{\varepsilon' \in X} \varepsilon') \mid X \in \mathcal{P}(csts_1 \cup csts_2)\} & (\text{Theorem 21}) \\ \subseteq & \{\varepsilon''_1 \oplus \cdots \oplus \varepsilon''_k \mid \textit{ with all } \varepsilon''_i \textit{ distinct}, \varepsilon''_i \in cl_G(\varepsilon_1) \cup cl_G(\varepsilon_2)\} & (\text{ind. hyp. + Thm 20}) \\ \subseteq & \{\varepsilon''_1 \oplus \cdots \oplus \varepsilon''_k \mid \textit{ with all } \varepsilon''_i \textit{ distinct}, \varepsilon''_i \in cl_G(\varepsilon_1 \oplus \varepsilon_2))\} & (\text{def. } cl_G(\varepsilon_1 \oplus \varepsilon_2)) \end{array}$$

For $N(\varepsilon) \leq k$ consider the case of $\varepsilon = \mu x.\varepsilon_1$.

$$\begin{array}{lll}
& \mathit{collectStates}_{F\triangleleft G}(\lambda_{F\triangleleft G}(\mu x.\varepsilon_1)) & \\
= & \mathit{collectStates}_{F\triangleleft G}(\lambda_{F\triangleleft G}(\varepsilon_1[\mu x.\varepsilon_1/x])) & (\text{def. } \lambda_{F\triangleleft G}) \\
\subseteq & \{\varepsilon'_1 \oplus \cdots \varepsilon'_k \mid \mathit{with\ all\ } \varepsilon'_i \mathit{\ distinct}, \varepsilon'_i \in \mathit{cl}_G(\varepsilon_1[\mu x.\varepsilon_1/x])\} & (\text{ind. hyp.}) \\
\subseteq & \{\varepsilon'_1 \oplus \cdots \varepsilon'_k \mid \mathit{with\ all\ } \varepsilon'_i \mathit{\ distinct}, \varepsilon'_i \in \mathit{cl}_G(\mu x.\varepsilon_1))\} & (\text{def. } \mathit{cl}_G(\mu x.\varepsilon_1))
\end{array}$$

□

Theorem 23 For a given expression $\varepsilon \in \mathit{Exp}_G$, $D(\{\varepsilon\}, \emptyset)$ terminates.

Proof. From Theorem 22, we know that

$$\mathit{collectStates}_G(\lambda_G(\varepsilon)) \subseteq \{\varepsilon_1 \oplus \cdots \oplus \varepsilon_k \mid \mathit{with\ all\ } \varepsilon_i \mathit{\ distinct},\ \varepsilon_i \in \mathit{cl}_G(\varepsilon)\}$$

Because $\mathit{cl}_G(\varepsilon)$ is finite the number of possible combinations for the above expressions is finite. For all $\varepsilon_i \in \mathit{cl}_G(\varepsilon)$, we have $\mathit{cl}_G(\varepsilon_i) \subseteq \mathit{cl}_G(\varepsilon)$ (and thus $\mathit{cl}_G(\varepsilon_1 \oplus \cdots \oplus \varepsilon_k) \subseteq \mathit{cl}_G(\varepsilon)$). This gives us an upper bound for the number of states that need to be processed. Since by the definition of D no state is processed twice, the set *newsts* will eventually be empty and, therefore, $D(\{\varepsilon\}, \emptyset)$ terminates. □