Monadic Expressions and Their Derivatives

Samira Attou, Ludovic Mignot, Clément Miklarz, Florent Nicart

RAIRO-Theor. Inf. Appl. 58 (2024) 6
https://doi.org/10.1051/ita/2023014

RAIRO - Theoretical Informatics and Applications
www.rairo-ita.org

MONADIC EXPRESSIONS AND THEIR DERIVATIVES

Samira Attou[1], Ludovic Mignot[2,*], Clément Miklarz[2] and Florent Nicart[2]

Abstract. There are several well-known ways to compute derivatives of regular expressions due to Brzozowski, Antimirov or Lombardy and Sakarovitch. We propose another one which abstracts the underlying data structures (*e.g.* sets or linear combinations) using the notion of monad. As an example of this generalization advantage, we first introduce a new derivation technique based on the graded module monad and then show an application of this technique to generalize the parsing of expressions with capture groups and back references. We also extend operators defining expressions to any n-ary functions over value sets, such as classical operations (like negation or intersection for Boolean weights) or more exotic ones (like algebraic mean for rational weights). Moreover, we present how to compute a (non-necessarily finite) automaton from such an extended expression, using the Colcombet and Petrisan categorical definition of automata. These category theory concepts allow us to perform this construction in a unified way, whatever the underlying monad. Finally, to illustrate our work, we present a Haskell implementation of these notions using advanced techniques of functional programming, and we provide a web interface to manipulate concrete examples.

Mathematics Subject Classification. 68Q45, 68Q70.

Received January 27, 2023. Accepted November 23, 2023.

1. Introduction

Regular expressions are a classical way to represent associations between words and value sets. As an example, classical regular expressions denote sets of words and regular expressions with multiplicities denote formal series. From a regular expression, solving the membership test (determining whether a word belongs to the denoted language) or the weighting test (determining the weight of a word in the denoted formal series) can be solved, following Kleene theorems [1, 2] by computing a finite automaton, such as the position automaton [3**?** –5].

Another family of methods to solve these tests is the family of derivative computations, that does not require the construction of a whole automaton. The common point of these techniques is to transform the test for an arbitrary word into the test for the empty word, which can be easily solved in a purely syntactical way (*i.e.* by induction over the structure of expressions). Brzozowski [6] shows how to compute, from a regular expression E and a word w, a regular expression $d_w(E)$ denoting the set of words w' such that ww' belongs to the language denoted by E. Solving the membership test hence becomes the membership test for the empty word in the expression $d_w(E)$. Antimirov [7] modifies this method in order to produce sets of expressions instead

Keywords and phrases: derivatives of (weighted) regular expressions, monads, automata constructions, haskell implementation

[1] LIGM, Université Gustave Eiffel, 5 Boulevard Descartes — Champs s/ Marne 77454 Marne-la-Vallée Cedex 2, France.
[2] GR²IF, Université de Rouen Normandie, Avenue de l'Université, 76801 Saint-Étienne-du-Rouvray, France.

* Corresponding author: ludovic.mignot@univ-rouen.fr

of expressions, *i.e.* defines the partial derivatives $\partial_w(E)$ as a set of expressions the sum of which denotes the same language as $d_w(E)$. While the number of derivatives is exponential w.r.t. the length $|E|$ of E in the worst case[1], the partial derivatives produce at most a linear number of expressions w.r.t. $|E|$. Lombardy and Sakarovitch [8] extend these methods to expressions with multiplicities. Finally, Sulzmann and Lu [9] apply these derivation techniques to parse POSIX expressions.

It is well-known that these methods are based on a common operation, the quotient of languages. Furthermore, Antimirov's method can be interpreted as the derivation of regular expression with multiplicities in the Boolean semiring. However, the Brzozowski computation does not produce the same expressions (*i.e.* equality over the syntax trees) as the Antimirov one.

Main contributions: In this paper, which is an extended version of a contribution to NCMA 2022 [10], we present a unification of these computations by applying notions of category theory to the category of sets, and show how to compute categorical automata as defined in [11], by reinterpreting the work started in the "habilitation à diriger des recherches" of one of the authors [12]. We make use of classical monads to model well-known derivatives computations. Furthermore, we deal with *extended* expressions in a general way: in this paper, expressions can support extended operators like complement, intersection, but also any n-ary function (algebraic mean, extrema multiplications, *etc.*). The main difference with [12] is that we formally state the languages and series that the expressions denote in an inherent way w.r.t. the underlying monads.

More precisely, this paper presents:

- an extension of expressions to any n-ary function over the value set,
- a monadic generalization of expressions,
- a solution for the membership/weight test for these expressions,
- a computation of categorical derivative automata,
- a new monad that fits with the extension to n-ary functions,
- an illustration implemented in Haskell using advanced functional programming,
- an extension to capture groups and back references expressions.

Motivation: The unification of derivation techniques is a goal by itself. Moreover, the formal tools used to achieve this unification are also useful: Monads offer both theoretical and practical advantages. Indeed, from a theoretical point of view, these structures allow the abstraction of properties and focus on the principal mechanisms that allow solving the membership and weight problems. Besides, the introduction of exotic monads can also facilitate the study of finiteness of derivated terms. From a practical point of view, monads are easy to implement (even in some other languages than Haskell) and allow us to produce compact and safe code. Finally, we can easily combine different algebraic structures or add some technical functionalities (capture groups, logging, nondeterminism, *etc.*) thanks to notions like monad transformers [13] that we consider in this paper.

This paper is structured as follows. In Section 2, we gather some preliminary material, like algebraic structures or category theory notions. We also introduce some functions well-known to the Haskell community that can allow us to reduce the size of our equations. We then structurally define the expressions we deal with, the associated series and the weight test for the empty word in Section 3. In order to extend this test to any arbitrary word, we first state in Section 4 some properties required by the monads we consider. Once this so-called support is determined, we show in Section 5 how to compute the derivatives. The computation of derivative automata is explained in Section 6. A new monad and its associated derivatives computation is given in Section 7. An implementation is presented in Section 8. Finally, we show how to (alternatively to [9]) compute derivatives of capture group expressions in Section 9 and show that as far as the same operators are concerned, the derivative formulae are the same whatever the underlying monad is.

[1] as far as rules of associativity, commutativity and idempotence of the sum are considered, possibly infinite otherwise.

2. Preliminaries

For classical notions of category theory, the reader should refer to [14].

We denote by $S \to S'$ the set of functions from a set S to a set S'. The notation $\lambda x \to f(x)$ is the classical way to define an anonymous version of a function f. As an example, the function $\lambda x \to x + 1$ is the successor function.

A *monoid* is a set S endowed with an associative operation and a unit element. A *semiring* is a structure $(S, \times, +, 1, 0)$ such that $(S, \times, 1)$ is a monoid, $(S, +, 0)$ is a commutative monoid, $\times$ distributes over $+$ and 0 is an annihilator for $\times$. A *starred semiring* is a semiring with a unary function * such that

$$k^\star = 1 + k \times k^\star = 1 + k^\star \times k.$$

A $\mathbb{K}$-*series* over the free monoid $(\Sigma^*, \cdot, \varepsilon)$ associated with an alphabet Σ, for a semiring $\mathbb{K} = (K, \times, +, 1, 0)$, is a function from Σ^* to K. The set of $\mathbb{K}$-*series* can be endowed with the structure of semiring as follows:

$$1(w) = \begin{cases} 1 & \text{if } w = \varepsilon, \\ 0 & \text{otherwise,} \end{cases} \qquad 0(w) = 0,$$

$$(S_1 + S_2)(w) = S_1(w) + S_2(w), \qquad (S_1 \times S_2)(w) = \sum_{u \cdot v = w} S_1(u) \times S_2(v).$$

Furthermore, if $S_1(\varepsilon) = 0$ (*i.e.* S_1 is said to be *proper*), the *star* of S_1 is the series defined by

$$(S_1)^\star(\varepsilon) = 1, \qquad (S_1)^\star(w) = \sum_{n \leq |w|, w = u_1 \cdots u_n, u_j \neq \varepsilon} S_1(u_1) \times \cdots \times S_1(u_n).$$

Finally, any function f in $K^n \to K$ can be extended to combine n series into a new one as follows:

$$(f(S_1, \ldots, S_n))(w) = f(S_1(w), \ldots, S_n(w)). \tag{2.1}$$

A *functor* (more precisely, a functor over a subcategory of the category of sets) F associates with each set S a set $F(S)$ and with each function f in $S \to S'$ a function $F(f)$ from $F(S)$ to $F(S')$ such that

$$F(\mathrm{id}) = \mathrm{id}, \qquad F(f \circ g) = F(f) \circ F(g),$$

where id is the identity function and $\circ$ the classical function composition.

A *monad* (more precisely, a monad over a subcategory of the category of sets) M is a functor endowed with two (families of) functions

- `pure`, from a set S to the set $M(S)$,
- `bind`, sending any function f in $S \to M(S')$ to $M(S) \to M(S')$,

such that the three following conditions are satisfied:

$$\texttt{bind}(f)(\texttt{pure}(s)) = f(s), \quad \texttt{bind}(\texttt{pure}) = \mathrm{id},$$
$$\texttt{bind}(g)(\texttt{bind}(f)(m)) = \texttt{bind}(\lambda x \to \texttt{bind}(g)(f(x)))(m).$$

A monad can be viewed as an algebraic data structure allowing us to apply particular coherent transformations over the elements it might contain while preserving the properties of the data structure. Let us introduce now three monads that we will use in the following of this paper.

Example 2.1. The `Maybe` monad is a functor that allows us to extend a set with one element denoted by `Nothing`. It is an elegant way to model a partial function to a set S as a total function to the set $\texttt{Maybe}(S)$. More precisely, the `Maybe` monad associates:

- any set S with the set $\texttt{Maybe}(S) = \{\texttt{Just}(s) \mid s \in S\} \cup \{\texttt{Nothing}\}$, where `Just` and `Nothing` are two syntactic tokens allowing us to extend a set with one value;
- any function f with the function $\texttt{Maybe}(f)$ defined by

$$\texttt{Maybe}(f)(\texttt{Just}(s)) = \texttt{Just}(f(s)), \qquad \texttt{Maybe}(f)(\texttt{Nothing}) = \texttt{Nothing}$$

- is endowed with the functions `pure` and `bind` defined by:

$$\texttt{pure}(s) = \texttt{Just}(s), \qquad \begin{aligned} \texttt{bind}(f)(\texttt{Just}(s)) &= f(s), \\ \texttt{bind}(f)(\texttt{Nothing}) &= \texttt{Nothing}. \end{aligned}$$

Example 2.2. The `Set` monad is a functor that can be used to extend computations perform on a set S to computations over the subsets of S. It is classically considered to model non-deterministic computations and their compositions. More precisely, the `Set` monad associates:

- with any set S the set 2^S,
- with any function f the function $\texttt{Set}(f)$ defined by $\texttt{Set}(f)(R) = \bigcup_{r \in R} \{f(r)\}$,
- is endowed with the functions `pure` and `bind` defined by:

$$\texttt{pure}(s) = \{s\}, \qquad \texttt{bind}(f)(R) = \bigcup_{r \in R} f(r).$$

Example 2.3. The $\texttt{LinComb}(\mathbb{K})$ monad, for a given semiring $\mathbb{K}$, is a functor that allows us to easily extend computations perform on a set S to computations over the linear combinations over S. It can be used to model non-deterministic weighted computations and their compositions, as probabilistic measures. More precisely, the $\texttt{LinComb}(\mathbb{K})$ monad, for $\mathbb{K} = (K, \times, +, 1, 0)$, associates:

- with any set S the set of $\mathbb{K}$-linear combinations of elements of S, where a linear combination is a finite (formal, commutative) sum of couples (denoted by $\boxplus$) in $K \times S$ where $(k, s) \boxplus (k', s) = (k + k', s)$,
- with any function f the function $\texttt{LinComb}(\mathbb{K})(f)$ defined by

$$\texttt{LinComb}(\mathbb{K})(f)(R) = \bigboxplus_{(k,r) \in R} (k, f(r)),$$

- is endowed with the functions `pure` and `bind` defined by:

$$\texttt{pure}(s) = (1, s), \qquad \texttt{bind}(f)(R) = \bigboxplus_{(k,r) \in R} k \otimes f(r),$$

where $k \otimes R = \bigboxplus_{(k',r) \in R} (k \times k', r)$.

Sometimes, monad names can be very long (*e.g.*, $\texttt{LinComb}(\mathbb{K})$). For the sake of concision, we use the following operators from the Haskell community to abstract the ambient monad:

$$f \mathbin{<\$>} s = M(f)(s), \qquad m \gg\!= f = \texttt{bind}(f)(m).$$

If <$> can be used to lift unary functions to the monadic level, >>= and `pure` can be used to lift any n-ary function f in $S_1 \times \cdots \times S_n \to S$, defining a function $\texttt{lift}_n$ sending $S_1 \times \cdots \times S_n \to S$ to $M(S_1) \times \cdots \times M(S_n) \to M(S)$ as follows:

$$\begin{aligned}\texttt{lift}_n(f)(m_1, \ldots, m_n) = & m_1 >\!\!>\!= (\lambda s_1 \to \ldots \\ & m_n >\!\!>\!= (\lambda s_n \to \texttt{pure}(f(s_1, \ldots, s_n))) \ldots)\end{aligned}$$

Let us consider the set $\mathbb{1} = \{\top\}$ with only one element. The images of this set by some previously defined monads can be evaluated as value sets classically used to weight words in association with classical regular expressions. As an example, $\texttt{Maybe}(\mathbb{1})$ and $\texttt{Set}(\mathbb{1})$ are isomorphic to the Boolean set, and any set $\texttt{LinComb}(\mathbb{K})(\mathbb{1})$ can be converted into the underlying set of $\mathbb{K}$. This property allows us to extend coherently classical expressions to monadic expressions, where the type of the weights is therefore given by the ambient monad.

3. Monadic expressions

In this section, we extend the well-known notion of regular expressions to monadic expressions. This extension allows us to provide a unified version of classical algorithms to solve the membership/weight test, by generalizing derivatives computations using monads. More precisely, we show in this section how to compute the weight of the empty word by a recursive computation, unifying the classical computations of Brzozowski [6], Antimirov [7] and Lombardy and Sakarovitch [8].

As seen in the previous section, elements in $M(\mathbb{1})$ can be evaluated as classical value sets for some particular monads. Hence, we use these elements not only for the weights associated with words by expressions, but also for the elements that act over the denoted series, *e.g.*, scalar multiplication in classical weighted expressions over a semiring .

In the following, in addition to classical operators ($+$, $\cdot$ and *), we denote:

- the action of an element over a series by $\odot$,
- the application of a function named f by the same symbol f .

Definition 3.1. Let M be a monad. An *M-monadic expression* E over an alphabet Σ is inductively defined as follows:

$$\begin{array}{lll} E = a, & E = \varepsilon, & E = \emptyset, \\ E = E_1 + E_2, & E = E_1 \cdot E_2, & E = E_1^*, \\ E = \alpha \odot E_1, & E = E_1 \odot \alpha, & E = f\left(E_1, \ldots, E_n\right), \end{array}$$

where a is a symbol in Σ, $E_1, \ldots, E_n$ are n M-monadic expressions over Σ, α is an element of $M(\mathbb{1})$ and f is a function from $(M(\mathbb{1}))^n$ to $M(\mathbb{1})$.

We denote by $\mathrm{Exp}(\Sigma)$ the set of monadic expressions over an alphabet Σ.

Example 3.2. Notice that any n-ary function can be used in the generalization we exhibit. As an example of functions that can be used in our extension of classical operators, one can define the function $\texttt{ExtDist}(x_1, x_2, x_3) = \max(x_1, x_2, x_3) - \min(x_1, x_2, x_3)$ from $\mathbb{N}^3$ to $\mathbb{N}$.

Similarly to classical regular expressions, monadic expressions associate a weight with any word. Such a relation can be denoted *via* a formal series. However, before defining this notion, in order to simplify our study, we choose to only consider proper expressions. Let us first show how to characterize them by the computation of a nullability value.

Definition 3.3. Let M be a monad such that the structure $(M(\mathbb{1}), +, \times, ^\star, 1, 0)$ is a starred semiring. The *nullability value* of an M-monadic expression E over an alphabet Σ is the element $\texttt{Null}(E)$ of $M(\mathbb{1})$ inductively

defined as follows:

$$\begin{aligned}
\texttt{Null}(\varepsilon) &= 1, & \texttt{Null}(\emptyset) &= 0,\\
\texttt{Null}(a) &= 0, & \texttt{Null}(E_1 + E_2) &= \texttt{Null}(E_1) + \texttt{Null}(E_2),\\
\texttt{Null}(E_1 \cdot E_2) &= \texttt{Null}(E_1) \times \texttt{Null}(E_2), & \texttt{Null}(E_1^*) &= \texttt{Null}(E_1)^\star,\\
\texttt{Null}(\alpha \odot E_1) &= \alpha \times \texttt{Null}(E_1), & \texttt{Null}(E_1 \odot \alpha) &= \texttt{Null}(E_1) \times \alpha,\\
\multicolumn{4}{c}{\texttt{Null}(f(E_1, \ldots, E_n)) = f(\texttt{Null}(E_1), \ldots, \texttt{Null}(E_n)),}
\end{aligned}$$

where a is a symbol in Σ, $(E_1, \ldots, E_n)$ are n M-monadic expressions over Σ, α is an element of $M(\mathbb{1})$ and f is a function from $(M(\mathbb{1}))^n$ to $M(\mathbb{1})$.

When the considered semiring is not a starred one, the computation of the nullability value may diverge. Thus, in order to compute it, let us consider the Maybe monad to elegantly deal with such a partial function.

Definition 3.4. Let M be a monad such that the structure $(M(\mathbb{1}), +, \times, 1, 0)$ is a semiring. The *partial nullability value* of an M-monadic expression E over an alphabet Σ is the element $\texttt{PartNull}(E)$ of $\texttt{Maybe}(M(\mathbb{1}))$ defined as follows:

$$\begin{aligned}
\texttt{PartNull}(\varepsilon) = \texttt{Just}(1), \quad \texttt{PartNull}(\emptyset) &= \texttt{Just}(0), \quad \texttt{PartNull}(a) = \texttt{Just}(0),\\
\texttt{PartNull}(E_1 + E_2) &= \texttt{lift}_2(+)(\texttt{PartNull}(E_1), \texttt{PartNull}(E_2)),\\
\texttt{PartNull}(E_1 \cdot E_2) &= \texttt{lift}_2(\times)(\texttt{PartNull}(E_1), \texttt{PartNull}(E_2)),\\
\texttt{PartNull}(E_1^*) &= \begin{cases} \texttt{Just}(1) & \text{if } \texttt{PartNull}(E_1) = \texttt{Just}(0),\\ \texttt{Nothing} & \text{otherwise,} \end{cases}\\
\texttt{PartNull}(\alpha \odot E_1) &= (\lambda E \to \alpha \times E) \mathbin{<\$>} \texttt{PartNull}(E_1),\\
\texttt{PartNull}(E_1 \odot \alpha) &= (\lambda E \to E \times \alpha) \mathbin{<\$>} \texttt{PartNull}(E_1),\\
\texttt{PartNull}(f(E_1, \ldots, E_n)) &= \texttt{lift}_n(f)(\texttt{PartNull}(E_1), \ldots, \texttt{PartNull}(E_n)),
\end{aligned}$$

where a is a symbol in Σ, $(E_1, \ldots, E_n)$ are n M-monadic expressions over Σ, α is an element of $M(\mathbb{1})$ and f is a function from $(M(\mathbb{1}))^n$ to $M(\mathbb{1})$.

An expression E is *proper* if its partial nullability value is not $\texttt{Nothing}$, *i.e.* if it is a value $\texttt{Just}(v)$. In this case, v is its nullability value, denoted by $\texttt{Null}(E)$ (by abuse).

Definition 3.5. Let M be a monad such that the structure $(M(\mathbb{1}), +, \times, 1, 0)$ is a semiring, and E be a M-monadic proper expression over an alphabet Σ. The *series* $S(E)$ *associated* with E is inductively defined as follows:

$$S(\varepsilon)(w) = \begin{cases} 1 & \text{if } w = \varepsilon,\\ 0 & \text{otherwise,} \end{cases} \quad S(\emptyset)(w) = 0, \quad S(a)(w) = \begin{cases} 1 & \text{if } w = a,\\ 0 & \text{otherwise,} \end{cases}$$

$$\begin{aligned}
S(E_1 + E_2) = S(E_1) + S(E_2), \quad S(E_1 \cdot E_2) &= S(E_1) \times S(E_2), \quad S(E_1^*) = (S(E_1))^\star,\\
S(\alpha \odot E_1)(w) = \alpha \times S(E_1)(w), \quad S(E_1 \odot \alpha)(w) &= S(E_1)(w) \times \alpha,\\
S(f(E_1, \ldots, E_n)) &= f(S(E_1), \ldots, S(E_n)),
\end{aligned}$$

where a is a symbol in Σ, $(E_1, \ldots, E_n)$ are n M-monadic expressions over Σ, α is an element of $M(\mathbb{1})$ and f is a function from $(M(\mathbb{1}))^n$ to $M(\mathbb{1})$.

From now on, we consider the set $\mathrm{Exp}(\Sigma)$ of M-monadic expressions over Σ to be endowed with the structure of a semiring, and two expressions denoting the same series to be equal. The *weight associated with* a word w in Σ^* *by* E is the value $\mathtt{weight}_w(E) = S(E)(w)$. The nullability of a proper expression is the weight it associates with ε, following Definition 3.4 and Definition 3.5 by a trivial induction over the structure of expression setting w to ε in Definition 3.5 .

Proposition 3.6. *Let M be a monad such that the structure $(M(\mathbb{1}), +, \times, 1, 0)$ is a semiring. Let E be an M-monadic proper expression over Σ. Then:*

$$\mathtt{Null}(E) = \mathtt{weight}_\varepsilon(E).$$

The previous proposition implies that the weight of the empty word can be syntactically computed (*i.e.* inductively computed from a monadic expression). Now, let us show how to extend the computation of this weight from ε to any other word by defining the computation of derivatives for monadic expressions.

4. Monadic supports for expressions

A $\mathbb{K}$-*left-semimodule*, for a semiring $\mathbb{K} = (K, \times, +, 1, 0)$, is a commutative monoid $(S, \pm, \underline{0})$ endowed with a function $\triangleright$ from $K \times S$ to S such that:

$$(k \times k') \triangleright s = k \triangleright (k' \triangleright s), \quad (k + k') \triangleright s = k \triangleright s \pm k' \triangleright s,$$
$$k \triangleright (s \pm s') = k \triangleright s \pm k \triangleright s', \quad 1 \triangleright s = s, \quad 0 \triangleright s = k \triangleright \underline{0} = \underline{0}.$$

A $\mathbb{K}$-*right-semimodule* can be defined symmetrically.

An *operad* [15, 16] is a structure $(O, (\circ_j)_{j\in\mathbb{N}}, \mathrm{id})$ where O is a graded set (*i.e.* $O = \bigcup_{n\in\mathbb{N}} O_n$), id is an element of O_1, $\circ_j$ is a function defined for any three integers (i, j, k)[2] with $0 < j \leq k$ in $O_k \times O_i \to O_{k+i-1}$ such that for any elements $p_1 \in O_m$, $p_2 \in O_n$, $p_3 \in O_p$:

$$\forall j \text{ such that } 0 < j \leq m\text{: } \ \mathrm{id} \circ_1 p_1 = p_1 \circ_j \mathrm{id} = p_1,$$
$$\forall j, j' \text{ such that } 0 < j \leq m \text{ and } 0 < j' \leq n\text{: } \ p_1 \circ_j (p_2 \circ_{j'} p_3) = (p_1 \circ_j p_2) \circ_{j+j'-1} p_3,$$
$$\forall j, j' \text{ such that } 0 < j' \leq j \leq m\text{: } \ (p_1 \circ_j p_2) \circ_{j'} p_3 = (p_1 \circ_{j'} p_3) \circ_{j+p-1} p_2.$$

Combining these compositions $\circ_j$, one can define a composition $\circ$ sending $O_k \times O_{i_1} \times \cdots \times O_{i_k}$ to $O_{i_1+\cdots+i_k}$: for any element $(p, q_1, \ldots, q_k)$ in $O_k \times O^k$,

$$p \circ (q_1, \ldots, q_k) = (\cdots((p \circ_k q_k) \circ_{k-1} q_{k-1} \cdots) \cdots) \circ_1 q_1.$$

Conversely, the composition $\circ$ can define the compositions $\circ_j$ using the identity element: for any two elements (p, q) in $O_k \times O_i$, for any integer $0 < j \leq k$:

$$p \circ_j q = p \circ (\underbrace{\mathrm{id}, \ldots, \mathrm{id}}_{j-1 \text{ times}}, q, \underbrace{\mathrm{id}, \ldots, \mathrm{id}}_{k-j \text{ times}}).$$

As an example, the set of n-ary functions over a set, with the identity function as unit, forms an operad.

[2] every couple (i, k) unambiguously defines the domain and codomain of a function $\circ_j$

A *module over an operad* $(O, \circ, \mathrm{id})$ is a set S endowed with a function $\circledast$ from $O_n \times S^n$ to S such that

$$f \circledast (f_1 \circledast (s_{1,1}, \dots, s_{1,i_1}), \dots, f_n \circledast (s_{n,1}, \dots, s_{n,i_n})) \\ = (f \circ (f_1, \dots, f_n)) \circledast (s_{1,1}, \dots, s_{1,i_1}, \dots, s_{n,1}, \dots, s_{n,i_n}).$$

The extension of the computation of derivatives could be performed for any monad. Indeed, any monad could be used to define well-typed auxiliary functions that mimic the classical computations. However, some properties should be satisfied in order to compute weights equivalently to Definition 3.5. Therefore, in the following we consider a restricted kind of monads.

A *monadic support* is a structure $(M, +, \times, 1, 0, \pm, \underline{0}, \ltimes, \rhd, \lhd, \circledast)$ satisfying:

- M is a monad,
- $\mathbb{R} = (M(\mathbb{1}), +, \times, 1, 0)$ is a semiring,
- $\mathbb{M} = (M(\mathrm{Exp}(\Sigma)), \pm, \underline{0})$ is a monoid,
- $(\mathbb{M}, \ltimes)$ is a $\mathrm{Exp}(\Sigma)$-right-semimodule,
- $(\mathbb{M}, \rhd)$ is a $\mathbb{R}$-left-semimodule,
- $(\mathbb{M}, \lhd)$ is a $\mathbb{R}$-right-semimodule,
- $(M(\mathrm{Exp}(\Sigma)), \circledast)$ is a module for the operad of the functions over $M(\mathbb{1})$.

An *expressive support* is a monadic support $(M, +, \times, 1, 0, \pm, \underline{0}, \ltimes, \rhd, \lhd, \circledast)$ endowed with a function `toExp` from $M(\mathrm{Exp}(\Sigma))$ to $\mathrm{Exp}(\Sigma)$ satisfying the following conditions:

$$\mathtt{weight}_w(\mathtt{toExp}(m)) = m \gg\!= \mathtt{weight}_w \tag{4.1}$$
$$\mathtt{toExp}(m \ltimes F) = \mathtt{toExp}(m) \cdot F, \tag{4.2}$$
$$\mathtt{toExp}(m \pm m') = \mathtt{toExp}(m) + \mathtt{toExp}(m'), \tag{4.3}$$
$$\mathtt{toExp}(m \rhd x) = \mathtt{toExp}(m) \odot x, \tag{4.4}$$
$$\mathtt{toExp}(x \lhd m) = x \odot \mathtt{toExp}(m), \tag{4.5}$$
$$\mathtt{toExp}(f \circledast (m_1, \dots, m_n)) = f(\mathtt{toExp}(m_1), \dots, \mathtt{toExp}(m_n)). \tag{4.6}$$

Let us now illustrate this notion with three expressive supports that will allow us to model well-known derivatives computations.

Example 4.1 (The `Maybe` support)**.** A support can be defined for the `Maybe` monad, mimicking the Boolean semiring for the space of values, where `Nothing` is the `false` Boolean value and $\mathtt{Just}(\top)$ is the `true` Boolean value. The actions of these Boolean-like values are either to cancel a computation or to leave it unchanged, allowing us to easily chain computations such as Brzozowski derivatives. More formally, we set:

$$\mathtt{toExp}(\mathtt{Nothing}) = 0, \quad \mathtt{toExp}(\mathtt{Just}(E)) = E,$$
$$\mathtt{Nothing} + m = m, \qquad \mathtt{Nothing} \times m = \mathtt{Nothing},$$
$$m + \mathtt{Nothing} = m, \qquad m \times \mathtt{Nothing} = \mathtt{Nothing},$$
$$\mathtt{Just}(\top) + \mathtt{Just}(\top) = \mathtt{Just}(\top), \qquad \mathtt{Just}(\top) \times \mathtt{Just}(\top) = \mathtt{Just}(\top),$$
$$\mathtt{Nothing} \pm m = m, \quad m \pm \mathtt{Nothing} = m, \quad \mathtt{Just}(E) \pm \mathtt{Just}(E') = \mathtt{Just}(E + E'),$$
$$1 = \mathtt{Just}(\top), \quad 0 = \mathtt{Nothing}, \quad \underline{0} = \mathtt{Nothing},$$
$$m \ltimes F = (\lambda E \to E \cdot F) \mathbin{<\$>} m,$$
$$m \rhd m' = m \gg\!= (\lambda x \to m'), \quad m \lhd m' = m' \gg\!= (\lambda x \to m),$$
$$f \circledast (m_1, \dots, m_n) = \mathtt{pure}(f(\mathtt{toExp}(m_1), \dots, \mathtt{toExp}(m_n))).$$

Example 4.2 (The `Set` support)**.** A support can be defined for the `Set` monad, mimicking the Boolean semiring for the space of values, where $\emptyset$ is the `false` Boolean value, $\{\top\}$ is the `true` Boolean value, $+$ is the set union and $\times$ the intersection, mimicking the Boolean disjunction and conjunction. The actions of these Boolean-like values are either to cancel a computation or to leave it unchanged. However, unlike the `Maybe` monad, the `Set` monad allows us to non-determistically split a computation into several parts and compose associated computations, allowing us to easily chain Antimirov derivatives. More formally, we set:

$$\begin{gathered}
\texttt{toExp}(\{E_1, \dots, E_n\}) = E_1 + \cdots + E_n, \\
+ = \cup, \quad \times = \cap, \quad \pm = \cup, \quad 1 = \{\top\}, \quad 0 = \emptyset, \quad \underline{0} = \emptyset, \\
m \ltimes F = (\lambda E \to E \cdot F) \mathbin{<\$>} m, \\
m \triangleright m' = m \gg\!= (\lambda x \to m'), \quad m \triangleleft m' = m' \gg\!= (\lambda x \to m), \\
f \circledast (m_1, \dots, m_n) = \texttt{pure}(f(\texttt{toExp}(m_1), \dots, \texttt{toExp}(m_n))).
\end{gathered}$$

Example 4.3 (The `LinComb`($\mathbb{K}$) support)**.** A support can be defined for the `LinComb`($\mathbb{K}$) monad, mimicking the semiring $\mathbb{K}$ for the space of values, where any value $(k, \top)$ in `LinComb`($\mathbb{K}$)($\mathbb{1}$) can be seen as the scalar k of $\mathbb{K}$. The actions of these values allow us to weight the computations using linear combinations. More formally, we set:

$$\begin{gathered}
\texttt{toExp}((k_1, E_1) \boxplus \cdots \boxplus (k_n, E_n)) = k_1 \odot E_1 + \cdots + k_n \odot E_n, \\
+ = \boxplus, \quad (k, \top) \times (k', \top) = (k \times k', \top), \quad 1 = (1, \top), \quad 0 = (0, \top), \\
\pm = \boxplus, \quad \underline{0} = (0, \top), \\
m \ltimes F = (\lambda E \to E \cdot F) \mathbin{<\$>} m, \\
m \triangleright m' = m \gg\!= (\lambda x \to m'), \quad m \triangleleft k = (\lambda E \to E \odot k) \mathbin{<\$>} m, \\
f \circledast (m_1, \dots, m_n) = \texttt{pure}(f(\texttt{toExp}(m_1), \dots, \texttt{toExp}(m_n))).
\end{gathered}$$

5. Monadic derivatives

In the following, $(M, +, \times, 1, 0, \pm, \underline{0}, \ltimes, \triangleright, \triangleleft, \circledast, \texttt{toExp})$ is an expressive support.

Let us now show how to solve the membership/weight test for monadic expressions by unifying and generalizing the three already known methods using derivatives computations due to Brzozowski [6], Antimirov [7] and Lombardy and Sakarovitch [8].

As usual, the derivative of an expression w.r.t. a symbol is a purely syntactical version of the quotient operation that can be performed over the associated languages or series, where the quotient of a series S w.r.t. a symbol a associates with the word w the same weight as the series S associates with the word aw.

This computations holds inductively over the structure of expressions. As an example if S is a series associated with an expression $E_1 \cdot E_2$, then its derivative w.r.t. a symbol a, which basically filters the words that start with a and then removes this symbol keeping the suffixes, is computed from two parts:

1. we can filter the words that start with a in E_1 and remove this symbol keeping the suffixes, (which is a recursive call of the derivative computation over E_1) and then monadically catenate E_2 with the monadic derivatives (using the operation $\ltimes$ of the support),
2. or we can consider the weight of the empty word in E_1 (a.k.a. Null(E_1)), filter the words that start with a in E_2 and remove this symbol keeping the suffixes (which is a recursive call of the derivative computation over E_2), and combine the results with the support operation $\triangleright$,

and finally combining the two parts with the support sum $\pm$. Let us now exhibit the whole computation.

Definition 5.1. The *derivative* of an M-monadic expression E over Σ w.r.t. a symbol a in Σ is the element $d_a(E)$ in $M(\mathrm{Exp}(\Sigma))$ inductively defined as follows:

$$d_a(\varepsilon) = \underline{0}, \quad d_a(\emptyset) = \underline{0}, \quad d_a(b) = \begin{cases} \mathtt{pure}(\varepsilon) & \text{if } a = b, \\ \underline{0} & \text{otherwise,} \end{cases}$$
$$d_a(E_1 + E_2) = d_a(E_1) \pm d_a(E_2), \quad d_a(E_1^*) = d_a(E_1) \ltimes E_1^*,$$
$$d_a(E_1 \cdot E_2) = d_a(E_1) \ltimes E_2 \pm \mathtt{Null}(E_1) \triangleright d_a(E_2),$$
$$d_a(\alpha \odot E_1) = \alpha \triangleright d_a(E_1), \quad d_a(E_1 \odot \alpha) = d_a(E_1) \triangleleft \alpha,$$
$$d_a(f(E_1, \ldots, E_n)) = f \circledast (d_a(E_1), \ldots, d_a(E_n))$$

where b is a symbol in Σ, $(E_1, \ldots, E_n)$ are n M-monadic expressions over Σ, α is an element of $M(\mathbb{1})$ and f is a function from $(M(\mathbb{1}))^n$ to $M(\mathbb{1})$.

The link between derivatives and series can be stated as follows, which is an alternative description of the classical quotient.

Proposition 5.2. *Let E be an M-monadic expression over an alphabet Σ, a be a symbol in Σ and w be a word in Σ^*. Then:*

$$\mathtt{weight}_{aw}(E) = d_a(E) \gg\!= \mathtt{weight}_w.$$

Proof. Let us proceed by induction over the structure of E. All the classical cases (*i.e.* the function operator left aside) can be proved following the classical methods ([6–8]). Therefore, let us consider this last case.

$$\begin{aligned}
&d_a(f(E_1, \ldots, E_n)) \gg\!= \mathtt{weight}_w \\
&\quad = \mathtt{weight}_w(\mathtt{toExp}(d_a(f(E_1, \ldots, E_n)))) && \text{(Eq (4.1))} \\
&\quad = \mathtt{weight}_w(\mathtt{toExp}(f \circledast (d_a(E_1), \ldots, d_a(E_n))) && \text{(Def 5.1))} \\
&\quad = \mathtt{weight}_w(f(\mathtt{toExp}(d_a(E_1)), \ldots, \mathtt{toExp}(d_a(E_n)))) && \text{(Eq (4.6))} \\
&\quad = f(\mathtt{weight}_w(\mathtt{toExp}(d_a(E_1))), \ldots, \mathtt{weight}_w(\mathtt{toExp}(d_a(E_n)))) && \text{(Def 3.5, Eq (2.1))} \\
&\quad = f(d_a(E_1) \gg\!= \mathtt{weight}_w, \ldots, d_a(E_n) \gg\!= \mathtt{weight}_w) && \text{(Eq (4.1))} \\
&\quad = f(\mathtt{weight}_{aw}(E_1), \ldots, \mathtt{weight}_{aw}(E_n)) && \text{(Ind. hyp.)} \\
&\quad = \mathtt{weight}_{aw}(f(E_1, \ldots, E_n)) && \text{(Def 3.5, Eq (2.1))}
\end{aligned}$$

□

Let us define how to extend the derivative computation from symbols to words, using the monadic functions.

Definition 5.3. The *derivative* of an M-monadic expression E over Σ w.r.t. a word w in Σ^* is the element $d_w(E)$ in $M(\mathrm{Exp}(\Sigma))$ inductively defined as follows:

$$d_\varepsilon(E) = \mathtt{pure}(E), \qquad d_{a \cdot v}(E) = d_a(E) \gg\!= d_v,$$

where a is a symbol in Σ and v a word in Σ^*.

Finally, it can be easily shown, by induction over the length of the words, following Proposition 5.2, that the derivatives computation can be used to define a syntactical computation of the weight of a word associated with an expression.

Theorem 5.4. *Let E be an M-monadic expression over an alphabet Σ and w be a word in Σ^*. Then:*

$$\texttt{weight}_w(E) = d_w(E) \gg\!= \texttt{Null}.$$

Notice that, restraining monadic expressions to regular ones,

- the `Maybe` support leads to the classical derivatives [6],
- the `Set` support leads to the partial derivatives [7],
- the `LinComb` support leads to the derivatives with multiplicities [8].

Example 5.5. Let us consider the function `ExtDist` defined in Example 3.2 and the $\texttt{LinComb}(\mathbb{N})$-monadic expression $E = \texttt{ExtDist}(a^*b^* + b^*a^*, b^*a^*b^*, a^*b^*a^*)$. The respective weights of the words aaa and aab can be determined by repetitively compute the derivative w.r.t. the symbols a and b, and by finally computing the weight of the empty word with the function Null as follows:

$$\begin{aligned}
d_a(E) &= \texttt{ExtDist}(a^*b^* + a^*, a^*b^*, a^*b^*a^* + a^*)\\
d_{aa}(E) &= \texttt{ExtDist}(a^*b^* + a^*, a^*b^*, a^*b^*a^* + 2 \odot a^*)\\
d_{aaa}(E) &= \texttt{ExtDist}(a^*b^* + a^*, a^*b^*, a^*b^*a^* + 3 \odot a^*)\\
d_{aab}(E) &= \texttt{ExtDist}(b^*, b^*, b^*a^*)\\
\texttt{weight}_{aaa}(E) &= d_{aaa}(E) \gg\!= \texttt{Null}\\
&= \texttt{ExtDist}(1+1, 1, 1+3) = 4 - 1 = 3\\
\texttt{weight}_{aab}(E) &= d_{aab}(E) \gg\!= \texttt{Null} = \texttt{ExtDist}(1, 1, 1) = 0
\end{aligned}$$

In the next section, we show how to compute the derivative automaton associated with an expression.

6. Automata construction

Let us now show how to compute the derivatives automata classically associated with the derivatives computations. In order to abstract the ambient monads, let us promote the classical automaton definition to the categorical level [3], using a definition relatively close to the one of Colcombet and Petrisan [11].

A *category* $\mathcal{C}$ is defined by:

- a class $\mathrm{Obj}_{\mathcal{C}}$ of *objects*,
- for any two objects A and B, a set $\mathrm{Hom}_{\mathcal{C}}(A, B)$ of *morphisms*,
- for any three objects A, B and C, an associative *composition function* $\circ_{\mathcal{C}}$ in $\mathrm{Hom}_{\mathcal{C}}(B, C) \longrightarrow \mathrm{Hom}_{\mathcal{C}}(A, B) \longrightarrow \mathrm{Hom}_{\mathcal{C}}(A, C)$,
- for any object A, an *identity morphism* id_A in $\mathrm{Hom}_{\mathcal{C}}(A, A)$, such that for any morphisms f in $\mathrm{Hom}_{\mathcal{C}}(A, B)$ and g in $\mathrm{Hom}_{\mathcal{C}}(B, A)$, $f \circ_{\mathcal{C}} \mathrm{id}_A = f$ and $\mathrm{id}_A \circ_{\mathcal{C}} g = g$.

Given a category $\mathcal{C}$, a $\mathcal{C}$-automaton is a tuple $(\Sigma, I, Q, F, i, \delta, f)$ where

- Σ is a set of symbols (the alphabet),
- I is the initial object, in $\mathrm{Obj}(\mathcal{C})$,
- Q is the state object, in $\mathrm{Obj}(\mathcal{C})$,
- F is the final object, in $\mathrm{Obj}(\mathcal{C})$,
- i is the initial morphism, in $\mathrm{Hom}_{\mathcal{C}}(I, Q)$,
- δ is the transition function, in $\Sigma \longrightarrow \mathrm{Hom}_{\mathcal{C}}(Q, Q)$,
- f is the value morphism, in $\mathrm{Hom}_{\mathcal{C}}(Q, F)$.

[3] see [14] for an introduction to category theory.

The function δ can be extended as a monoid morphism from the free monoid $(\Sigma^*, \cdot, \varepsilon)$ to the morphism monoid $(\mathrm{Hom}_{\mathcal{C}}(Q,Q), \circ_{\mathcal{C}}, \mathrm{id}_Q)$, leading to the following weight definition.

The weight associated by a $\mathcal{C}$-automaton $A = (\Sigma, I, Q, F, i, \delta, f)$ with a word w in Σ^* is the morphism $\mathtt{weight}(w)$ in $\mathrm{Hom}_{\mathcal{C}}(I,F)$ defined by

$$\mathtt{weight}(w) = f \circ_{\mathcal{C}} \delta(w) \circ_{\mathcal{C}} i.$$

If the ambient category is the category of sets, and if $I = \mathbb{1}$, then the weight of a word is equivalently an element of F, following the isomorphism sending any element f of F over the function sending $\top$ to f. Consequently, a deterministic (complete) automaton is equivalently a Set-automaton with $\mathbb{1}$ as the initial object and $\mathbb{B}$ as the final object, since in this case:

- I (the initial object) is $\mathbb{1}$,
- Q (the state object) is any set,
- F (the final object) is the Boolean set $\mathbb{B}$,
- i (the initial morphism) is a function from $\mathbb{1}$ to Q, defining $i(\top)$ as the initial state of the DFA,
- δ (the transition function) is a function in $\Sigma \to Q \to Q$,
- f (the value morphism) is a function from Q to $\mathbb{B}$, defining the set of final states.

Given a monad M, the Kleisli composition [17] of two morphisms $f \in \mathrm{Hom}_{\mathcal{C}}(A,B)$ and $g \in \mathrm{Hom}_{\mathcal{C}}(B,C)$ is the morphism $(f \mathbin{>\!\!=\!\!>} g)(x) = f(x) \gg\!= g$ in $\mathrm{Hom}_{\mathcal{C}}(A,C)$. This composition defines a category, called the Kleisli category [17] $\mathcal{K}(M)$ of M, where:

- the objects are the sets,
- the morphisms between two sets A and B are the functions between A and $M(B)$,
- the identity is the function $\mathtt{pure}$.

Considering these categories:

- a deterministic automaton is equivalently a $\mathcal{K}(\mathtt{Maybe})$-automaton,
- a nondeterministic automaton is equivalently a $\mathcal{K}(\mathtt{Set})$-automaton,
- a weighted automaton over a semiring $\mathbb{K}$ is equivalently a $\mathcal{K}(\mathtt{LinComb}(\mathbb{K}))$-automaton,

all with $\mathbb{1}$ as both the initial object and the final object. As an example, in a $\mathcal{K}(\mathtt{Set})$-automaton, I (the initial object) is $\mathbb{1}$, Q (the state object) is any set, F (the final object) is $\mathbb{1}$, i (the initial morphism) is a function from $\mathbb{1}$ to $\mathtt{Set}(Q)$, defining $i(\top)$ as the set of the initial states of the NFA, δ (the transition function) is a function in $\Sigma \to Q \to \mathtt{Set}(Q)$, f (the value morphism) is a function from Q to $\mathtt{Set}(\mathbb{1})$, defining the set of final states (which are the states q of Q satisfying $f(q) = \{\top\}$).

Furthermore, for a given expression E, if $i = \mathtt{pure}(E)$, $\delta(a)(E') = \mathrm{d}_a(E')$ and $f = \mathtt{Null}$, we can compute the well-known derivative automata using the three previously defined supports, and the accessible part of these automata are finite ones as far as classical expressions are concerned [6–8].

More precisely, extended expressions can lead to infinite automata, as shown in the next example.

Example 6.1. Considering the computations of Example 5.5, it can be shown that

$$d_{a^n}(E) = \mathtt{ExtDist}(a^*b^* + a^*, a^*b^*, a^*b^*a^* + n \odot a^*).$$

Hence, there is not a finite number of derivated terms, that are the states in the classical derivative automaton. This infinite automaton is represented in Figure 1, where the final weights of the states are represented by double edges. The sink states are omitted.

In the following section, let us show how to model a new monad in order to solve this problem.

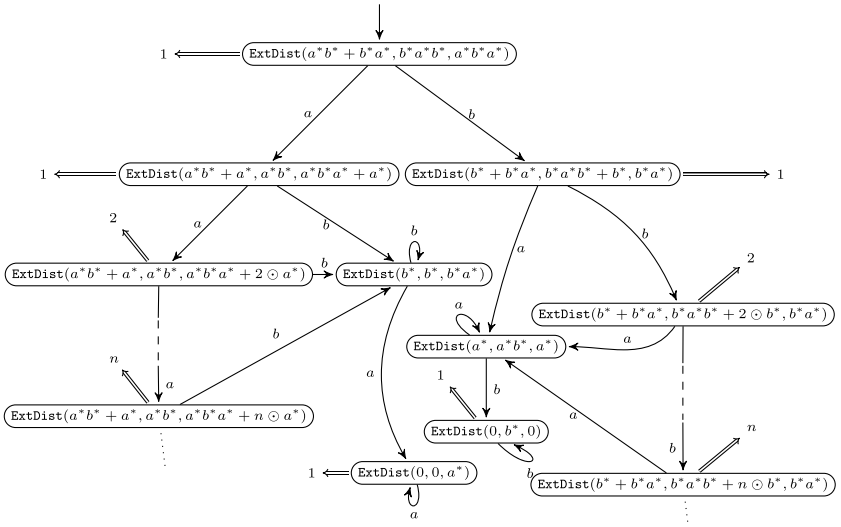

FIGURE 1. The (infinite) derivative weighted automaton associated with E.

7. The graded module monad

Let us consider an operad $\mathbb{O} = (O, \circ, \mathrm{id})$ and the *association* sending:

- any set S to $\bigcup_{n\in\mathbb{N}} O_n \times S^n$,
- any f in $S \to S'$ to the function g in $\bigcup_{n\in\mathbb{N}} O_n \times S^n \to \bigcup_{n\in\mathbb{N}} O_n \times S'^n$:

$$g(o, (s_1, \ldots, s_n)) = (o, (f(s_1), \ldots, f(s_n)))$$

It can be checked that this is a functor, denoted by $\texttt{GradMod}(\mathbb{O})$. Moreover, it forms a monad considering the two following functions:

$$\texttt{pure}(s) = (\mathrm{id}, s),$$
$$(o, (s_1, \ldots, s_n)) \gg\!= f = (o \circ (o_1, \ldots, o_n), (s_{1,1}, \ldots, s_{1,i_1}, \ldots, s_{n,1}, \ldots, s_{n,i_n}))$$

where $f(s_j) = (o_j, s_{j,1}, \ldots, s_{j,i_j})$. However, notice that $\texttt{GradMod}(\mathbb{O})(\mathbb{1})$ cannot be easily evaluated as a value space. Indeed, the values it contains are composed by a first component that is a (not necessarily 0-ary) n-ary function, and a second one that is a vector of n $\top$, the only element in $\mathbb{1}$. We would prefer to obtain a classical scalar value, which would be a value in O_0. Thus, let us compose it with another monad. As an example, let us consider a semiring $\mathbb{K} = (K, \times, +, 1, 0)$ and the operad $\mathbb{O}$ of the n-ary functions over K. Hence, let us define the functor[4] $\texttt{GradComb}(\mathbb{O}, \mathbb{K})$ that sends S to $\texttt{GradMod}(\mathbb{O})(\texttt{LinComb}(\mathbb{K})(S))$.

To show that this combination is a monad, let us first define a function α sending $\texttt{GradComb}(\mathbb{O}, \mathbb{K})(S)$ to $\texttt{GradMod}(\mathbb{O})(S)$. It can be easily done by converting a linear combination into an *operadic combination*, *i.e.* an element in $\texttt{GradMod}(\mathbb{O})(S)$, with the following function $\texttt{toOp}$:

$$\texttt{toOp}((k_1, s_1) \boxplus \cdots \boxplus (k_n, s_n)) = (\lambda(x_1, \ldots, x_n) \to k_1 \times x_1 + \cdots + k_n \times x_n, (s_1, \ldots, s_n)),$$
$$\alpha(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) = (o \circ (o_1, \ldots, o_n), (s_{1,1}, \ldots, s_{1,i_1}, \ldots, s_{n,1}, \ldots, s_{n,i_n}))$$

[4] it is folk knowledge that the composition of two functors is a functor.

where $(o_j, (s_{j,1}, \ldots, s_{j,i_j})) = \texttt{toOp}(\mathcal{L}_j)$.

Consequently, we can define the monadic functions as follows:

$$\texttt{pure}(s) = (\mathrm{id}, (1, s)),$$
$$(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) \gg\!= f = \alpha(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) \gg\!= f$$

where

- the left-hand side occurrence of $\gg\!=$ is the bind function of the $\texttt{GradComb}(\mathbb{O}, \mathbb{K})$, the function to be defined;
- the right-hand side occurrence of $\gg\!=$ is the bind function of the $\texttt{GradMod}(\mathbb{O})$ monad.

Let us finally define an expressive support for this monad:

$$\begin{aligned}
\texttt{toExp}(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) &= o(\texttt{toExp}(\mathcal{L}_1), \ldots, \texttt{toExp}(\mathcal{L}_n)),\\
(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) + (o', (\mathcal{L}'_1, \ldots, \mathcal{L}'_{n'})) &= (o + o', (\mathcal{L}_1, \ldots, \mathcal{L}_n, \mathcal{L}'_1, \ldots, \mathcal{L}'_{n'}))\\
(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) \times (o', (\mathcal{L}'_1, \ldots, \mathcal{L}'_{n'})) &= (o \times o', (\mathcal{L}_1, \ldots, \mathcal{L}_n, \mathcal{L}'_1, \ldots, \mathcal{L}'_{n'}))\\
\pm = +, \quad 1 = (\mathrm{id}, (1, \top)), \quad 0 = (\mathrm{id}, (0, \top)), \quad \underline{0} &= (\mathrm{id}, (0, \top)),\\
m \ltimes F &= \texttt{pure}(\texttt{toExp}(m) \cdot F),\\
(o, (\mathcal{M}_1, \ldots, \mathcal{M}_k)) \triangleright (o', (\mathcal{L}_1, \ldots, \mathcal{L}_n)) &= (o(\mathcal{M}_1, \ldots, \mathcal{M}_k) \times o', (\mathcal{L}_1, \ldots, \mathcal{L}_n)),\\
(o, (\mathcal{L}_1, \ldots, \mathcal{L}_n)) \triangleleft (o', (\mathcal{M}_1, \ldots, \mathcal{M}_k)) &= (o \times o'(\mathcal{M}_1, \ldots, \mathcal{M}_k), (\mathcal{L}_1, \ldots, \mathcal{L}_n))\\
f \circledast ((o_1, (\mathcal{L}_{1,1}, \ldots, \mathcal{L}_{1,i_1})), \ldots, (o_n, (\mathcal{L}_{n,1}, \ldots, \mathcal{L}_{n,i_n})))&\\
= (f \circ (o_1, \ldots, o_n), (\mathcal{L}_{1,1}, \ldots, \mathcal{L}_{1,i_1}, \ldots, \mathcal{L}_{n,1}, \ldots, \mathcal{L}_{n,i_n}))&\\
\text{where } (o + o')(x_1, \ldots, x_{n+n'}) &= o(x_1, \ldots, x_n) + o'(x_{n+1}, \ldots, x_{n+n'})\\
(o \times o')(x_1, \ldots, x_{n+n'}) &= o(x_1, \ldots, x_n) \times o'(x_{n+1}, \ldots, x_{n+n')}
\end{aligned}$$

This support can then be used to compute automata through monadic derivatives, as shown in the following example.

Example 7.1. Let us consider that two elements in $\texttt{GradComb}(\mathbb{O}, \mathbb{K})(\mathrm{Exp}(\Sigma))$ are equivalent, denoted by $\equiv$, if they have the same image by $\texttt{toExp}$. Let us consider the expression $E = \texttt{ExtDist}(a^*b^* + b^*a^*, b^*a^*b^*, a^*b^*a^*)$ of Example 5.5. The respective weights of the words aaa and aab can be determined by repetitively compute the derivative w.r.t. the symbols a and b, and by finally computing the weight of the empty word with the function Null as follows:

$$\begin{aligned}
d_a(E) &= \texttt{ExtDist} \circledast ((+, (a^*b^*, a^*)), (\mathrm{id}, a^*b^*), (+, (a^*b^*a^*, a^*)))\\
&= (\texttt{ExtDist} \circ (+, \mathrm{id}, +), (a^*b^*, a^*, a^*b^*, a^*b^*a^*, a^*))\\
d_{aa}(E) &= (\texttt{ExtDist} \circ (+, \mathrm{id}, + \circ (+, \mathrm{id})), (a^*b^*, a^*, a^*b^*, a^*b^*a^*, a^*, a^*))\\
&\equiv (\texttt{ExtDist} \circ (+, \mathrm{id}, + \circ (\mathrm{id}, 2\times)), (a^*b^*, a^*, a^*b^*, a^*b^*a^*, a^*))\\
d_{aaa}(E) &\equiv (\texttt{ExtDist} \circ (+, \mathrm{id}, + \circ (\mathrm{id}, 3\times)), (a^*b^*, a^*, a^*b^*, a^*b^*a^*, a^*))\\
d_{aab}(E) &\equiv (\texttt{ExtDist} \circ (+, \mathrm{id}, +), (b^*, \emptyset, b^*, b^*a^*, \emptyset))\\
&\equiv (\texttt{ExtDist}, (b^*, b^*, b^*a^*))\\
\texttt{weight}_{aaa}(E) &= d_{aaa}(E) \gg\!= \texttt{Null}\\
&= \texttt{ExtDist} \circ (+, \mathrm{id}, +)(1, 1, 1, 1, 3)\\
&= \texttt{ExtDist}(1 + 1, 1, 1 + 3) = 4 - 1 = 3\\
\texttt{weight}_{aab}(E) &= d_{aab}(E) \gg\!= \texttt{Null} = \texttt{ExtDist}(1, 1, 1) = 0
\end{aligned}$$

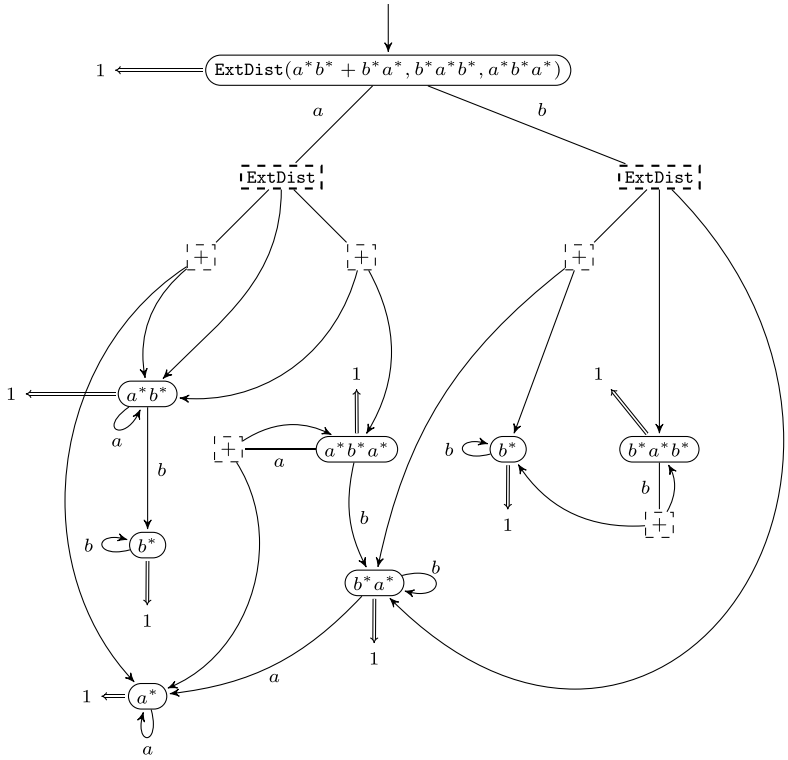

FIGURE 2. The Associated Derivative Automaton of $\texttt{ExtDist}(a^*b^* + b^*a^*, b^*a^*b^*, a^*b^*a^*)$.

Using this monad, the number of derivated terms, that is the number of states in the associated derivative automaton, is finite. Indeed, the computations are delayed in the transition structure during the evaluation instead of being syntactically represented in the expressions computed *via* derivatives. This automaton is represented in Figure 2. Notice that the dashed rectangle represent the functions that are composed during the traversal associated with a word. The final weights are represented by double edges. The sink states are omitted. The state b^* is duplicated to simplify the representation.

However, notice that not every monadic expression produces a finite set of derivated terms, as shown in the next example.

Example 7.2. Let us consider the expression E of Example 5.5 and the expression $F = E \cdot c^*$. It can be shown that

$$\begin{aligned} d_{a^n}(F) &= \texttt{toExp}(d_{a^n}(E)) \cdot c^* \\ &= \texttt{ExtDist}(a^*b^* + a^*, a^*b^*, a^*b^*a^* + n \odot a^*) \cdot c^*. \end{aligned}$$

Let us notice that each new application of the derivative operation w.r.t. the symbol a embeds a new expression, distinct from the previous ones. Consequently, the set of derivatives of F, *i.e.* the set $\{d_w(F) \mid w \in \Sigma^*\}$ is not finite, and so is not the derivative automaton of F.

The study of the necessary and sufficient conditions of monads that lead to a finite set of derivated terms is one of the next steps of our work. However, classical regular expressions still produce a finite set of derivated terms, since the performed computations remain the same as previous methods.

8. Haskell implementation

The membership/weight test has been implemented in Haskell, *via* a graphical interface where an expression E and a word w can be entered. Then the weight of the world w in E is computed for several monads:

- a Boolean *via* the supports associated with the monads `Maybe`, `Set`, `LinComb (Bool)` and `GradedModuleOfLinComb (Bool)`;
- an integer *via* the supports associated with the monads `LinComb (Int)` and `GradedModuleOfLinComb (Int)`;
- a double *via* the supports associated with the monads `LinComb (Double)` and `GradedModuleOfLinComb (Double)`.

Notice that no automaton computation is needed. The membership/weight test can be syntactically performed *via* the computation of derivatives, that necessarily halts as far as a finite word is concerned:

1. first, the derivatives of E w.r.t. w are inductively computed following Definition 5.1 and Definition 5.3;
2. then, the weight is computed following Theorem 5.4.

These notions are implemented using advanced functional programming elements:

- The notion of monad over a sub-category of sets is a typeclass using the *Constraint kind* to specify a sub-category;
- n-ary functions and their operadic structures are implemented using fixed length vectors, the size of which is determined at compilation using type level programming;
- The notion of graded module is implemented through an existential type to deal with unknown arities: Its monadic structure is based on an extension of heterogeneous lists, the *graded vectors*, typed w.r.t. the list of the arities of the elements it contains;
- The parser and some type level functions are based on dependently typed programming with singletons [18], allowing, for example, determining the type of the monads or the arity of the functions involved at run-time;
- An application is available on GitHub [19] illustrating the computations:
  - the backend uses servant to define an API;
  - the frontend is defined using Reflex, a functional reactive programming engine and cross compiled in JavaScript with GHCJS.

  As an example, the monadic expression of the previous examples can be entered in the web application as the following input:

```
ExtDist(a*.b*+b*.a*,b*.a*.b*,a*.b*.a*).
```

9. Capture groups

Standard POSIX regular expressions[20] and Perl compatible regular expressions[21] offer a mechanism to memorize a part of the matching input string and make reference to that part subsequently. Users of common tools such as `grep` or `sed` make use of this extensively. The pattern is captured by surrounding the corresponding sub-expression with parenthesis and reference to it can be made with $\backslash n$ where n is the number of the parenthesis group. For example:

```
> echo "user=Turing:Alan" | sed -r 's/.*=(.*):(.*)/Hi \2 \1 !/'
Hi Alan Turing !
```

We give here an equivalent definition along with derivation formulae and a monadic definition where the capture groups behave according to the POSIX specification. More precisely, when a capture group has been involved more than one time due to a stared sub-expression, the value of the corresponding variable corresponds to the last capture:

```
> echo "babbaabbaaab" |sed -r 's/(b(a*)b)*/X\2X/'
XaaaX
```

We also allow the nesting of back-references in the matching part of the expression:

```
> echo "xxaaabaayy" | grep -E "(a*)b\1"
xxaaabaayy
```

9.1. Syntax of expressions with capture groups

A *capture-group expression* E over a symbol alphabet Σ and a variable alphabet Γ (or Σ, Γ-expression for short) is inductively defined as

$$\begin{array}{lll} E = a, & E = \varepsilon, & E = \emptyset, \\ E = F + G, & E = F \cdot G, & E = F^*, \\ E = (F)_x, & & E = x, \end{array}$$

where F and G are two Σ, Γ-expressions, a is a symbol in Σ, u is in Σ^* and x is a variable in Γ. In the POSIX syntax, capture groups are implicitly mapped with variables respectively with the order of the opening parenthesis of a pair. Here, each capture group is associated explicitly to a variable by indexing the closing parenthesis with the name of this variable.

Example 9.1. Considering the last `sed` example, we will write the matching expression this way :

$$E = ((a^*)_x bx)^*.$$

9.2. Contextual expressions and their contextual languages

In order to define the contextual language and the derivation of capture-group expressions, we need to extend the syntax of the expressions in order to attach to any capture group the current part of the input string captured during an execution.

A *contextual capture-group expression* E over a symbol alphabet Σ and a variable alphabet Γ (or Σ, Γ-expression for short) is inductively defined as

$$\begin{array}{lll} E = a, & E = \varepsilon, & E = \emptyset, \\ E = F + G, & E = F \cdot G, & E = F^*, \\ E = (F)_x^u, & & E = x, \end{array}$$

where F and G are two Σ, Γ-expressions, a is a symbol in Σ, u is in Σ^* and x is a variable in Γ.

Notice that a Σ, Γ-expression is equivalent to a contextual capture-group expression where $u = \varepsilon$ for every occurrence of capture group.

In the following, we consider that a *context* is a function from Γ to Maybe(Σ^*), modelling the possibility that a variable was initialized (or not) during the parsing. The set of contexts is denoted by $\mathrm{Ctxt}(\Gamma, \Sigma)$.

Using these notions of contexts, let us now explain the semantics of contextual capture-group expressions. While parsing, a context is built to memorize the different associations words / variables . Therefore, a (contextual) language associated with an expression is a set of couples built from a language and the context that was used to compute it.

The classic atomic cases (a symbol, the empty word or the empty set) are easy to define, preserving the context. Another one is the case of a variable x: the context is applied here to compute the associated word (if it exists) and is preserved.

The recursive cases are interpreted as follows :

- The contextual language of a sum of two expressions is the union of their contextual languages, computed independently.
- The contextual language of a catenation of two expressions F and G is computed in three steps. First, the contextual language of F is computed. Secondly, for each couple (L, ctxt) of this contextual language, the function ctxt is considered as the new context to compute the contextual language of G, leading to new couples (L', ctxt'). Finally, for each of these combinations, a couple $(L \cdot L', \mathrm{ctxt}')$ is added to form the resulting contextual language.
- The contextual language of a starred expression is, classically, the infinite union of iterated catenations.
- The contextual language of a captured expression $(F)_x^u$ is computed in two steps. First, the contextual language of F is computed. Then, for each couple (L, ctxt) of it, a word w is chosen in L and the context ctxt must be updated coherently.

More formally, the *contextual language of* a Σ, Γ-expression E associated with a context ctxt in $\mathrm{Ctxt}(\Gamma, \Sigma)$ is the subset $\mathsf{L}^{\mathrm{ctxt}}(E)$ of $2^{\Sigma^*} \times \mathrm{Ctxt}(\Gamma, \Sigma)$ inductively defined as follows:

$$
\begin{aligned}
\mathsf{L}^{\mathrm{ctxt}}(a) &= \{(\{a\}, \mathrm{ctxt})\},\\
\mathsf{L}^{\mathrm{ctxt}}(\varepsilon) &= \{(\{\varepsilon\}, \mathrm{ctxt})\},\\
\mathsf{L}^{\mathrm{ctxt}}(\emptyset) &= \emptyset,\\
\mathsf{L}^{\mathrm{ctxt}}(x) &= \begin{cases} \emptyset & \text{if } \mathrm{ctxt}(x) = \mathrm{Nothing},\\ \{(\{w\}, \mathrm{ctxt})\} & \text{otherwise if } \mathrm{ctxt}(x) = \mathrm{Just}(w), \end{cases}\\
\mathsf{L}^{\mathrm{ctxt}}(F + G) &= \mathsf{L}^{\mathrm{ctxt}}(F) \cup \mathsf{L}^{\mathrm{ctxt}}(G),\\
\mathsf{L}^{\mathrm{ctxt}}(F \cdot G) &= \bigcup_{\substack{(L_1, \mathrm{ctxt}_1) \in \mathsf{L}^{\mathrm{ctxt}}(F),\\ (L_2, \mathrm{ctxt}_2) \in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(L_1 \cdot L_2, \mathrm{ctxt}_2)\},\\
\mathsf{L}^{\mathrm{ctxt}}(F^*) &= \bigcup_{n \in \mathbb{N}} (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{n}},\\
\mathsf{L}^{\mathrm{ctxt}}((F)_x^u) &= \bigcup_{\substack{(L_1, \mathrm{ctxt}_1) \in \mathsf{L}^{\mathrm{ctxt}}(F),\\ w \in L_1}} \{(\{w\}, [\mathrm{ctxt}_1]_{x \leftarrow uw})\},
\end{aligned}
$$

where F and G are two Σ, Γ-expressions, a is a symbol in Σ, x is a variable in Γ, u is in Σ^*, $\mathcal{L}^{\underline{n}}$ is defined, for any set $\mathcal{L}$ of couples (language, context) by

$$
\mathcal{L}^{\underline{n}} = \begin{cases} \displaystyle\bigcup_{(L, \mathrm{ctxt}) \in \mathcal{L}} \{(\{\varepsilon\}, \mathrm{ctxt})\} & \text{if } n = 0,\\ \displaystyle\bigcup_{\substack{(L_1, \mathrm{ctxt}_1) \in \mathcal{L},\\ (L_2, \mathrm{ctxt}_2) \in \mathcal{L}^{\underline{n-1}}}} \{(L_1 \cdot L_2, \mathrm{ctxt}_2)\} & \text{otherwise,} \end{cases}
$$

and $[\mathrm{ctxt}]_{x \leftarrow w}$ is the context defined by

$$
[\mathrm{ctxt}]_{x \leftarrow w}(y) = \begin{cases} \mathtt{Just}(w) & \text{if } x = y,\\ \mathrm{ctxt}(y) & \text{otherwise.} \end{cases}
$$

The *contextual language* of an expression E is the set of couples obtained from an uninitialised context, where nothing is associated with any variable, that is the set

$$\mathsf{L}^{\lambda\_\to\mathrm{Nothing}}(E).$$

Finally, the *language denoted* by an expression E is the set of words obtained by forgetting the contexts, that is the set

$$\bigcup_{(L,\_)\in\mathsf{L}^{\lambda\_\to\mathrm{Nothing}}(E)} L.$$

Example 9.2. Let us consider the three following expressions over the symbol alphabet $\{a, b, c\}$ and the variable alphabet $\{x\}$:

$$E_1 = ((a^*)_x bx)^*, \qquad E_2 = cx, \qquad E = E_1 \cdot E_2.$$

The language denoted by E_2 is empty, since it is computed from the empty context, where nothing is associated with x. However, parsing E_1 allows us to compute contexts that define word values to associate with x. Let us thus show how is defined the contextual language of E_1:

- the contextual language of $(a^*)_x$ is the set

$$\bigcup_{n\in\mathbb{N}} \{(\{a^n\}, \lambda x \to \mathrm{Just}(a^n))\}$$

  where each word a^n is recorded in a context;
- the contextual language of $(a^*)_x bx$ is the set

$$\bigcup_{n\in\mathbb{N}} \{(\{a^n b a^n\}, \lambda x \to \mathrm{Just}(a^n))\}$$

  where each word a^n is recorded in a context applied to evaluate the variable x;
- the contextual language of E_1 is the union of the two following sets S_1 and S_2:

$$S_1 = \{(\{\varepsilon\}, \lambda x \to \mathrm{Nothing})\}$$
$$S_2 = \{(\{a^n b a^n \mid n \in \mathbb{N}\}^* \cdot \{a^m b a^m\}, \lambda x \to \mathrm{Just}(a^m)) \mid m \in \mathbb{N}\}$$

  where each iteration of the outermost star produces a new record for the variable x in the context; however, notice that only the last one is recorded at the end of the process.

Finally, the language of E is obtained by considering the contexts obtained from the parsing of E_1 to evaluate the occurrence of x in E_2, leading to the set

$$\bigcup_{m\in\mathbb{N}} (\{a^n b a^n \mid n \in \mathbb{N}\}^* \cdot \{a^m b a^m c a^m\}).$$

Obviously, some classical equations still hold with these computations:

Lemma 9.3. *Let E, F and G be three Σ,Γ-expressions and* ctxt *be a context in* $\mathrm{Ctxt}(\Gamma,\Sigma)$. *The two following equations hold:*

$$\mathsf{L}^{\mathrm{ctxt}}(E\cdot(F+G)) = \mathsf{L}^{\mathrm{ctxt}}(E\cdot F + E\cdot G)$$
$$\mathsf{L}^{\mathrm{ctxt}}(F^*) = \mathsf{L}^{\mathrm{ctxt}}(\varepsilon + F\cdot F^*)$$

Proof. Let us proceed by equality sequences:

$$\begin{aligned}
\mathsf{L}^{\mathrm{ctxt}}(E\cdot(F+G)) &= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in\mathsf{L}^{\mathrm{ctxt}}(E),\\ (L_2,\mathrm{ctxt}_2)\in\mathsf{L}^{\mathrm{ctxt}_1}(F+G)}} \{(L_1\cdot L_2,\mathrm{ctxt}_2)\} && \text{(By definition of the language of a catenation)}\\
&= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in\mathsf{L}^{\mathrm{ctxt}}(E),\\ (L_2,\mathrm{ctxt}_2)\in\mathsf{L}^{\mathrm{ctxt}_1}(F)\cup\mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(L_1\cdot L_2,\mathrm{ctxt}_2)\} && \text{(By definition of the language of a sum)}\\
&= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in\mathsf{L}^{\mathrm{ctxt}}(E),\\ (L_2,\mathrm{ctxt}_2)\in\mathsf{L}^{\mathrm{ctxt}_1}(F)}} \{(L_1\cdot L_2,\mathrm{ctxt}_2)\}\\
&\quad\cup \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in\mathsf{L}^{\mathrm{ctxt}}(E),\\ (L_2,\mathrm{ctxt}_2)\in\mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(L_1\cdot L_2,\mathrm{ctxt}_2)\} && \text{(By definition of the union)}\\
&= \mathsf{L}^{\mathrm{ctxt}}(E\cdot F)\cup\mathsf{L}^{\mathrm{ctxt}}(E\cdot G) && \text{(By definition of the language of a catenation)}\\
&= \mathsf{L}^{\mathrm{ctxt}}(E\cdot F + E\cdot G) && \text{(By definition of the language of a sum)}
\end{aligned}$$

$$\begin{aligned}
\mathsf{L}^{\mathrm{ctxt}}(F^*) &= \bigcup_{n\in\mathbb{N}} (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{n}} && \text{(By definition of the language of a star)}\\
&= (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{0}} \cup \bigcup_{n\in\mathbb{N},n\geq 1} (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{n}} && \text{(By definition of } \mathbb{N})\\
&= (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{0}} \cup \bigcup_{n\in\mathbb{N}} \mathsf{L}^{\mathrm{ctxt}}(F)\cdot(\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{n}} && \text{(By definition of } \underline{n})\\
&= (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{0}} \cup \mathsf{L}^{\mathrm{ctxt}}(F)\cdot \bigcup_{n\in\mathbb{N}} (\mathsf{L}^{\mathrm{ctxt}}(F))^{\underline{n}} && \text{(By factorization of } \mathsf{L}^{\mathrm{ctxt}}(F), \text{ from previous point)}\\
&= L^{\mathrm{ctxt}}(\varepsilon + F\cdot F^*) && \text{(By definition of the language of the expression } \varepsilon + F\cdot F^*)
\end{aligned}$$

□

In order to solve the membership test for the contextual capture-group expressions, let us extend the classical derivation method. But first, let us show how to extend the nullability predicate, needed at the end of the process.

9.3. Nullability computation

The nullability predicate allows us to determine whether the empty word belongs to the language denoted by an expression. As far as capture groups are concerned, a context has to be computed. Therefore, the nullability predicate can be represented as a set of contexts the application of which produces a language that contains the empty word.

As we have seen, the nullability depends on the current context. Given an expression and a context ctxt, the nullability predicate is a set in $2^{\mathrm{Ctxt}(\Gamma,\Sigma)}$, computed as follows:

$$\begin{aligned}
\mathrm{Null}^{\mathrm{ctxt}}(\varepsilon) &= \{\mathrm{ctxt}\}\\
\mathrm{Null}^{\mathrm{ctxt}}(\emptyset) &= \emptyset\\
\mathrm{Null}^{\mathrm{ctxt}}(a) &= \emptyset\\
\mathrm{Null}^{\mathrm{ctxt}}(x) &= \begin{cases}\{\mathrm{ctxt}\} & \text{if } \mathrm{ctxt}(x) = \mathtt{Just}(\varepsilon)\\ \emptyset & \text{otherwise.}\end{cases}\\
\mathrm{Null}^{\mathrm{ctxt}}(E+F) &= \mathrm{Null}^{\mathrm{ctxt}}(E) \cup \mathrm{Null}^{\mathrm{ctxt}}(F)\\
\mathrm{Null}^{\mathrm{ctxt}}(E\cdot F) &= \bigcup_{\substack{\mathrm{ctxt}'\in\mathrm{Null}^{\mathrm{ctxt}}(F),\\ \mathrm{ctxt}''\in\mathrm{Null}^{\mathrm{ctxt}'}(G)}} \{\mathrm{ctxt}''\}\\
\mathrm{Null}^{\mathrm{ctxt}}(E^*) &= \{\mathrm{ctxt}\}\\
\mathrm{Null}^{\mathrm{ctxt}}((E)_x^u) &= \bigcup_{\mathrm{ctxt}'\in\mathrm{Null}^{\mathrm{ctxt}}(F)} \{[\mathrm{ctxt}']_{x\leftarrow u}\}
\end{aligned}$$

where E and F are two Σ,Γ-expressions, a is a symbol in Σ, x is a variable in Γ and u is in Σ^*.

Example 9.4. Let us consider the three expressions of Example 9.2:

$$E_1 = ((a^*)_x bx)^*, \qquad E_2 = cx, \qquad E = E_1 \cdot E_2.$$

For any context ctxt,

$$\mathrm{Null}^{\mathrm{ctxt}}(E_1) = \{\mathrm{ctxt}\}, \qquad \mathrm{Null}^{\mathrm{ctxt}}(E_2) = \emptyset, \qquad \mathrm{Null}^{\mathrm{ctxt}}(E) = \emptyset.$$

Indeed,

- the contextual language of the expression E_1 always contains the empty word, since it is a starred expression, and the context is preserved;
- the contextual languages of the expressions E_2 and E never contains the empty word, since E_2 starts with the symbol c and since E is a catenation of E_1 and E_2, implying the presence of the symbol c.

The nullability predicate allows us to determine whether there exists a couple in the contextual language of an expression such that its first component contains the empty word.

Proposition 9.5. *Let E be a Σ,Γ-expression and* ctxt *be a context in* $\mathrm{Ctxt}(\Gamma,\Sigma)$. *Then the two following conditions are equivalent:*

- $\mathrm{Null}^{\mathrm{ctxt}}(E) \neq \emptyset$,

- $\exists (L, \_) \in \mathsf{L}^{\mathrm{ctxt}}(E) \mid \varepsilon \in L$.

Proof. By induction over the structure of E:

- If $E = a \in \Sigma$ or $E = \emptyset$, the property holds since $\mathrm{Null}^{\mathrm{ctxt}}(E)$ is empty and since there is no couple (L, ctxt') in $\mathsf{L}^{\mathrm{ctxt}}(E)$ with ε in L.
- If $E = \varepsilon$, the following two conditions hold,

$$\mathrm{Null}^{\mathrm{ctxt}}(E) = \{\mathrm{ctxt}\}, \qquad \mathsf{L}^{\mathrm{ctxt}}(E) = \{(\{\varepsilon\}, \mathrm{ctxt})\},$$

  satisfying the stated condition.
- If $E = F + G$, the following two conditions hold:

$$\mathrm{Null}^{\mathrm{ctxt}}(F + G) = \mathrm{Null}^{\mathrm{ctxt}}(F) \cup \mathrm{Null}^{\mathrm{ctxt}}(G), \qquad \mathsf{L}^{\mathrm{ctxt}}(F + G) = \mathsf{L}^{\mathrm{ctxt}}(F) \cup \mathsf{L}^{\mathrm{ctxt}}(G).$$

  Since, by induction hypothesis, the following two conditions hold

$$\mathrm{Null}^{\mathrm{ctxt}}(F) \neq \emptyset \Leftrightarrow \exists (L, \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(F) \mid \varepsilon \in L,$$
$$\mathrm{Null}^{\mathrm{ctxt}}(G) \neq \emptyset \Leftrightarrow \exists (L, \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(G) \mid \varepsilon \in L,$$

  the proposition holds.
- If $E = F \cdot G$, the two following conditions hold:

$$\mathrm{Null}^{\mathrm{ctxt}}(F \cdot G) = \bigcup_{\substack{\mathrm{ctxt}' \in \mathrm{Null}^{\mathrm{ctxt}}(F), \\ \mathrm{ctxt}'' \in \mathrm{Null}^{\mathrm{ctxt}'}(G),}} \{\mathrm{ctxt}''\},$$
$$\mathsf{L}^{\mathrm{ctxt}}(F \cdot G) = \bigcup_{\substack{(L, \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(F), \\ (L', \mathrm{ctxt}'') \in \mathsf{L}^{\mathrm{ctxt}'}(G),}} \{(L \cdot L', \mathrm{ctxt}'')\}.$$

  Since, by induction hypothesis, the two following conditions hold,

$$\mathrm{Null}^{\mathrm{ctxt}}(F) \neq \emptyset \Leftrightarrow \exists (L, \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(F) \mid \varepsilon \in L,$$
$$\mathrm{Null}^{\mathrm{ctxt}'}(G) \neq \emptyset \Leftrightarrow \exists (L, \mathrm{ctxt}'') \in \mathsf{L}^{\mathrm{ctxt}'}(G) \mid \varepsilon \in L,$$

  the proposition holds.
- If $E = F^*$, since the two following conditions hold

$$\mathrm{Null}^{\mathrm{ctxt}}(F^*) = \{\mathrm{ctxt}\}, \qquad \mathsf{L}^{\mathrm{ctxt}}(F)^{\underline{0}} = \{(\{\varepsilon\}, \mathrm{ctxt})\} \in \mathsf{L}^{\mathrm{ctxt}}(F^*),$$

  the stated condition holds.
- If $E = (F)_x^u$, both following conditions hold:

$$\mathrm{Null}^{\mathrm{ctxt}}((F)_x^u) = \bigcup_{\mathrm{ctxt}' \in \mathrm{Null}^{\mathrm{ctxt}}(F)} \{[\mathrm{ctxt}']_{x \leftarrow u}\},$$
$$\mathsf{L}^{\mathrm{ctxt}}((F)_x^u) = \bigcup_{\substack{(L, \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(F), \\ w \in L}} \{(\{w\}, [\mathrm{ctxt}']_{x \leftarrow uw})\}.$$

Then, following induction hypothesis,

$$\mathrm{Null}^{\mathrm{ctxt}}(F) \neq \emptyset \Leftrightarrow \exists (L, \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(F) \mid \varepsilon \in L,$$

the stated condition holds.

- If $E = x$, both following conditions hold:

$$\mathrm{Null}^{\mathrm{ctxt}}(x) = \begin{cases} \{\mathrm{ctxt}\} & \text{if } \mathrm{ctxt}(x) = \mathtt{Just}(\varepsilon) \\ \emptyset & \text{otherwise,} \end{cases}$$

$$\mathsf{L}^{\mathrm{ctxt}}(x) = \begin{cases} \emptyset & \text{if } \mathrm{ctxt}(x) = \mathrm{Nothing}, \\ \{(\{w\}, \mathrm{ctxt})\} & \text{otherwise if } \mathrm{ctxt}(x) = \mathrm{Just}(w). \end{cases}$$

Therefore, the proposition holds.

□

9.4. Derivation formulae

Similarly to the nullability predicate, the derivation computation builds the context while parsing the expression. Therefore, the derivative of an expression with respect to a context is a set of couples (expression, context), inductively computed as follows, for any Σ, Γ-expression and for any context ctxt in $\mathrm{Ctxt}(\Gamma, \Sigma)$:

$$\begin{aligned}
\mathrm{d}_a^{\mathrm{ctxt}}(\varepsilon) &= \emptyset \\
\mathrm{d}_a^{\mathrm{ctxt}}(\emptyset) &= \emptyset \\
\mathrm{d}_a^{\mathrm{ctxt}}(b) &= \begin{cases} \emptyset & \text{if } a \neq b, \\ \{(\varepsilon, \mathrm{ctxt})\} & \text{otherwise,} \end{cases} \\
\mathrm{d}_a^{\mathrm{ctxt}}(x) &= \begin{cases} \mathrm{d}_a^{\mathrm{ctxt}}(w) & \text{if } \mathrm{ctxt}(x) = \mathtt{Just}(w) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathrm{d}_a^{\mathrm{ctxt}}(F + G) &= \mathrm{d}_a^{\mathrm{ctxt}}(F) \cup \mathrm{d}_a^{\mathrm{ctxt}}(G) \\
\mathrm{d}_a^{\mathrm{ctxt}}(F \cdot G) &= \bigcup_{(\mathrm{ctxt}', F') \in \mathrm{d}_a^{\mathrm{ctxt}}(F)} \{(F' \cdot G, \mathrm{ctxt}')\} \cup \bigcup_{\mathrm{ctxt}' \in \mathrm{Null}^{\mathrm{ctxt}}(F)} \mathrm{d}_a^{\mathrm{ctxt}'}(G) \\
\mathrm{d}_a^{\mathrm{ctxt}}(F^*) &= \bigcup_{(\mathrm{ctxt}', F') \in \mathrm{d}_a^{\mathrm{ctxt}}(F)} \{(F' \cdot F^*, \mathrm{ctxt}')\} \\
\mathrm{d}_a^{\mathrm{ctxt}}((F)_x^u) &= \bigcup_{(\mathrm{ctxt}', F') \in \mathrm{d}_a^{\mathrm{ctxt}}(F)} \{((F')_x^{u \cdot a}, \mathrm{ctxt}')\}
\end{aligned}$$

where F and G are two Σ, Γ-expressions, a is a symbol in Σ, x is a variable in Γ and u is in Σ^*.

Example 9.6. Let us consider the three expressions of Example 9.2:

$$E_1 = ((a^*)_x bx)^*, \qquad E_2 = cx, \qquad E = E_1 \cdot E_2.$$

Then, for any context ctxt, the derivatives of E w.r.t. the symbols a, b and c are the following ones:

$$\begin{aligned}
\mathrm{d}_a^{\mathrm{ctxt}}(E) &= \{((a^*)_x^a bx((a^*)_x bx)^* cx, \mathrm{ctxt})\}, \\
\mathrm{d}_b^{\mathrm{ctxt}}(E) &= \{(x((a^*)_x bx)^* cx, [\mathrm{ctxt}]_{x \leftarrow \varepsilon})\},
\end{aligned}$$

$$\mathrm{d}_c^{\mathrm{ctxt}}(E) = \{(x, \mathrm{ctxt})\}.$$

Indeed,

- the computation of the derivative of E w.r.t. a does not modify the context but (syntactically) memorizes that the symbol a was read once during the parsing (producing the subexpression $(a^*)_x^a$);
- the computation of the derivative of E w.r.t. b forces the context to associate x with ε; indeed, the capture group associated with x, $(a^*)_x$, needs to be evaluated to ε in order the symbol b to be the first symbol read during the parsing;
- the computation of the derivative of E w.r.t. a does not modify the context since the expression E_1 is a starred expression, which contextual language contains ε for any context, allowing c to be the first symbol read during the parsing.

The derivation of an expression allows us to syntactically express the computation of the quotient of the language components in contextual languages, where the quotient $w^{-1}(L)$ is the set $\{w' \mid ww' \in L\}$.

Proposition 9.7. *Let E be a Σ, Γ-expression,* ctxt *be a context in* $\mathrm{Ctxt}(\Gamma, \Sigma)$ *and a be a symbol in Σ. Then:*

$$\bigcup_{(E', \mathrm{ctxt}') \in d_a^{\mathrm{ctxt}}(E)} \mathsf{L}^{\mathrm{ctxt}'}(E') = \bigcup_{(L', \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(E)} \{(a^{-1}(L'), \mathrm{ctxt}')\}$$

Proof. By induction over the structure of E, assimilating $\emptyset$ and $\{(\emptyset, \mathrm{ctxt})\}$ for any context ctxt.

- If $E = \varepsilon$ or $E = \emptyset$, the property vacuously holds.
- If $E = b \in \Sigma$,

$$\begin{aligned}
&\bigcup_{(E', \mathrm{ctxt}') \in d_a^{\mathrm{ctxt}}(b)} \mathsf{L}^{\mathrm{ctxt}'}(E') \\
&= \begin{cases} \emptyset & \text{if } b \neq a, \\ \{(\{\varepsilon\}, \mathrm{ctxt})\} & \text{otherwise,} \end{cases} && \text{(By definition of the derivative of } b \text{)} \\
&= \{(a^{-1}(\{b\}), \mathrm{ctxt})\} && \text{(By definition of } a^{-1}\text{)} \\
&= \bigcup_{(L', \mathrm{ctxt}') \in \mathsf{L}^{\mathrm{ctxt}}(b)} \{(a^{-1}(L'), \mathrm{ctxt}')\} && \text{(By definition of the contextual language of } b \text{).}
\end{aligned}$$

- If $E = F + G$,

$$\begin{aligned}
&\bigcup_{(E', \mathrm{ctxt}') \in d_a^{\mathrm{ctxt}}(F+G)} \mathsf{L}^{\mathrm{ctxt}'}(E') \\
&= \bigcup_{(E', \mathrm{ctxt}') \in d_a^{\mathrm{ctxt}}(F) \cup d_a^{\mathrm{ctxt}}(G)} \mathsf{L}^{\mathrm{ctxt}'}(E') && \text{(By definition of the derivative of a sum)} \\
&= \bigcup_{(E', \mathrm{ctxt}') \in d_a^{\mathrm{ctxt}}(F)} \mathsf{L}^{\mathrm{ctxt}'}(E') \\
&\quad \cup \bigcup_{(E', \mathrm{ctxt}') \in d_a^{\mathrm{ctxt}}(G)} \mathsf{L}^{\mathrm{ctxt}'}(E') && \text{(By definition of the union)}
\end{aligned}$$

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(F)} \{(a^{-1}(L'),\mathrm{ctxt}')\}$$

$$\cup \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(G)} \{(a^{-1}(L'),\mathrm{ctxt}')\} \qquad \text{(Following the induction hypothesis)}$$

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(F)\cup \mathsf{L}^{\mathrm{ctxt}}(G)} \{(a^{-1}(L'),\mathrm{ctxt}')\} \qquad \text{(By definition of the union)}$$

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(F+G)} \{(a^{-1}(L'),\mathrm{ctxt}')\} \qquad \text{(By definition of the language of a sum).}$$

- If $E = F \cdot G$,

$$\bigcup_{(E',\mathrm{ctxt}')\in d_a^{\mathrm{ctxt}}(F\cdot G)} \mathsf{L}^{\mathrm{ctxt}'}(E')$$

$$= \bigcup_{(\mathrm{ctxt}',F')\in \mathrm{d}_a^{\mathrm{ctxt}}(F)} \mathsf{L}^{\mathrm{ctxt}'}(F'\cdot G)$$

$$\cup \bigcup_{\substack{\mathrm{ctxt}'\in \mathrm{Null}^{\mathrm{ctxt}}(F),\\ (G',\mathrm{ctxt}'')\in \mathrm{d}_a^{\mathrm{ctxt}'}(G)}} \mathsf{L}^{\mathrm{ctxt}''}(G') \qquad \text{(By definition of the derivative of a catenation)}$$

$$= \bigcup_{\substack{(\mathrm{ctxt}',F')\in \mathrm{d}_a^{\mathrm{ctxt}}(F),\\ (L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F'),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(L_1\cdot L_2,\mathrm{ctxt}_2)\}$$

$$\cup \bigcup_{\substack{\mathrm{ctxt}'\in \mathrm{Null}^{\mathrm{ctxt}}(F),\\ (G',\mathrm{ctxt}'')\in \mathrm{d}_a^{\mathrm{ctxt}'}(G)}} \mathsf{L}^{\mathrm{ctxt}''}(G') \qquad \text{(By definition of the language of a catenation)}$$

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(a^{-1}(L_1)\cdot L_2,\mathrm{ctxt}_2)\}$$

$$\cup \bigcup_{\substack{\mathrm{ctxt}_1\in \mathrm{Null}^{\mathrm{ctxt}}(F),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(a^{-1}(L_2),\mathrm{ctxt}_2)\} \qquad \text{(Following the induction hypothesis}$$

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(a^{-1}(L_1)\cdot L_2,\mathrm{ctxt}_2)\}$$

$$\cup \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ \varepsilon\in L_1,\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(a^{-1}(L_2),\mathrm{ctxt}_2)\} \qquad \text{(According to Proposition 9.5)}$$

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(G)}} \{(a^{-1}(L_1\cdot L_2),\mathrm{ctxt}_2)\} \qquad \text{(By definition of } a^{-1} \text{ for a a catenation)}$$

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(F\cdot G)} \{(a^{-1}(L'),\mathrm{ctxt}')\}$$ (By definition of the catenation of languages).

- If $E = F^*$,

$$\bigcup_{(E',\mathrm{ctxt}')\in d_a^{\mathrm{ctxt}}(F^*)} \mathsf{L}^{\mathrm{ctxt}'}(E')$$

$$= \bigcup_{(\mathrm{ctxt}',F')\in \mathrm{d}_a^{\mathrm{ctxt}}(F)} \mathsf{L}^{\mathrm{ctxt}'}(F'\cdot F^*)$$ (By definition of the derivative of a star)

$$= \bigcup_{\substack{(\mathrm{ctxt}',F')\in \mathrm{d}_a^{\mathrm{ctxt}}(F),\\ (L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F'),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(F^*)}} \{(L_1\cdot L_2,\mathrm{ctxt}_2)\}$$ (By definition of the language of a catenation)

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(F^*)}} \{(a^{-1}(L_1)\cdot L_2,\mathrm{ctxt}_2)\}$$ (Following the induction hypothesis)

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ (L_2,\mathrm{ctxt}_2)\in \mathsf{L}^{\mathrm{ctxt}_1}(F^*)}} \{(a^{-1}(L_1\cdot L_2),\mathrm{ctxt}_2)\}$$ (By definition of a^{-1})

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(F\cdot F^*)} \{(a^{-1}(L'),\mathrm{ctxt}')\}$$ (By definition of the language of a catenation)

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(\varepsilon+F\cdot F^*)} \{(a^{-1}(L'),\mathrm{ctxt}')\}$$ (By definition of of a^{-1})

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(F^*)} \{(a^{-1}(L'),\mathrm{ctxt}')\}$$ (From Lemma 9.3)

- If $E = (F)_x^u$,

$$\bigcup_{(E',\mathrm{ctxt}')\in d_a^{\mathrm{ctxt}}((F)_x^u)} \mathsf{L}^{\mathrm{ctxt}'}(E')$$

$$= \bigcup_{(\mathrm{ctxt}',F')\in \mathrm{d}_a^{\mathrm{ctxt}}(F)} \mathsf{L}^{\mathrm{ctxt}'}((F')_x^{u\cdot a})$$ (By definition of the derivative of a capture group)

$$= \bigcup_{\substack{(\mathrm{ctxt}',F')\in \mathrm{d}_a^{\mathrm{ctxt}}(F)\\ (L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}'}(F'),\\ w\in L_1}} \{(\{w\},[\mathrm{ctxt}_1]_{x\leftarrow uaw})\}$$ (By definition of the language of a capture group)

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ w\in a^{-1}(L_1)}} \{(\{w\},[\mathrm{ctxt}_1]_{x\leftarrow uaw})\}$$ (Following the induction hypothesis)

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ aw\in L_1}} \{(\{w\}, [\mathrm{ctxt}_1]_{x\leftarrow uaw})\} \qquad \text{(By definition of } a^{-1})$$

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ aw\in L_1}} \{(a^{-1}(\{aw\}), [\mathrm{ctxt}_1]_{x\leftarrow uaw})\} \qquad \text{(By definition of } a^{-1})$$

$$= \bigcup_{\substack{(L_1,\mathrm{ctxt}_1)\in \mathsf{L}^{\mathrm{ctxt}}(F),\\ w\in L_1}} \{(a^{-1}(\{w\}), [\mathrm{ctxt}_1]_{x\leftarrow uw})\} \qquad \text{(By definition of } a^{-1})$$

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}((F)^u_x)} \{(a^{-1}(L'), \mathrm{ctxt}')\} \qquad \text{(By definition of } a^{-1})$$

- If $E = x$,

$$\bigcup_{(E',\mathrm{ctxt}')\in d_a^{\mathrm{ctxt}}(x)} \mathsf{L}^{\mathrm{ctxt}'}(E')$$

$$= \begin{cases} \displaystyle\bigcup_{(E',\mathrm{ctxt}')\in d_a^{\mathrm{ctxt}}(w)} \mathsf{L}^{\mathrm{ctxt}'}(E') & \text{if } \mathrm{ctxt}(x) = \mathrm{Just}(w), \\ \emptyset & \text{otherwise,} \end{cases} \qquad \begin{array}{c}\text{(By definition of}\\ \text{the derivative of}\\ \text{a variable } x)\end{array}$$

$$= \begin{cases} \displaystyle\bigcup_{(w,\mathrm{ctxt})\in d_a^{\mathrm{ctxt}}(aw)} \mathsf{L}^{\mathrm{ctxt}}(w) & \text{if } \mathrm{ctxt}(x) = \mathrm{Just}(aw), \\ \emptyset & \text{otherwise,} \end{cases} \qquad \begin{array}{c}\text{(Substituting } w\\ \text{by } aw)\end{array}$$

$$= \begin{cases} \{(\{w\}, \mathrm{ctxt})\} & \text{if } \mathrm{ctxt}(x) = \mathrm{Just}(aw), \\ \emptyset & \text{otherwise,} \end{cases} \qquad \begin{array}{c}\text{(By definition of}\\ \text{the language of}\\ \text{aword)}\end{array}$$

$$= \begin{cases} \{(a^{-1}(\{aw\}), \mathrm{ctxt})\} & \text{if } \mathrm{ctxt}(x) = \mathrm{Just}(aw), \\ \emptyset & \text{otherwise,} \end{cases} \qquad \begin{array}{c}\text{(By definition}\\ \text{of } a^{-1})\end{array}$$

$$= \begin{cases} \{(a^{-1}(\{w\}), \mathrm{ctxt})\} & \text{if } \mathrm{ctxt}(x) = \mathrm{Just}(w), \\ \emptyset & \text{otherwise,} \end{cases} \qquad \begin{array}{c}\text{(Substituting } aw\\ \text{by } w)\end{array}$$

$$= \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(x)} \{(a^{-1}(L'), \mathrm{ctxt}')\} \qquad \begin{array}{c}\text{(By definition of}\\ \text{the language of}\\ \text{a variable)}\end{array}$$

□

The derivation w.r.t. a word is, as usual, an iterated application of the derivation w.r.t. a symbol, recursively defined as follows, for any Σ, Γ-expression E, for any context ctxt in $\mathrm{Ctxt}(\Gamma, \Sigma)$, for any symbol a in Σ and for any word v in Σ^*:

$$d_\varepsilon^{\mathrm{ctxt}}(E) = \{(E, \mathrm{ctxt})\}, \qquad d_{a\cdot v}^{\mathrm{ctxt}}(E) = \bigcup_{(E',\mathrm{ctxt}')\in d_a^{\mathrm{ctxt}}(E)} d_v^{\mathrm{ctxt}'}(E').$$

Example 9.8. Let us consider the three expressions of Example 9.6:

$$E_1 = ((a^*)_x bx)^*, \qquad E_2 = cx, \qquad E = E_1 \cdot E_2.$$

Then, for any context ctxt,

$$\begin{aligned}
\mathrm{d}^{\mathrm{ctxt}}_{ab}(E) &= \mathrm{d}^{\mathrm{ctxt}}_{b}((a^*)^a_x bx((a^*)_x bx)^* cx) \\
&= \{(x((a^*)_x bx)^* cx, [\mathrm{ctxt}]_{x\leftarrow a})\} \\
\mathrm{d}^{\mathrm{ctxt}}_{aba}(E) &= \mathrm{d}^{[\mathrm{ctxt}]_{x\leftarrow a}}_{a}(x((a^*)_x bx)^* cx) \\
&= \{(((a^*)_x bx)^* cx, [\mathrm{ctxt}]_{x\leftarrow a})\} \\
\mathrm{d}^{\mathrm{ctxt}}_{abac}(E) &= \mathrm{d}^{[\mathrm{ctxt}]_{x\leftarrow a}}_{c}(((a^*)_x bx)^* cx) \\
&= \{(x, [\mathrm{ctxt}]_{x\leftarrow a})\} \\
\mathrm{d}^{\mathrm{ctxt}}_{abaca}(E) &= \mathrm{d}^{[\mathrm{ctxt}]_{x\leftarrow a}}_{a}(x) \\
&= \{(\varepsilon, [\mathrm{ctxt}]_{x\leftarrow a})\}
\end{aligned}$$

Indeed, like it was the case in the previous example, these derivatives computations preserve the context, except for the case of the derivative w.r.t. ab. In this case, the nullability of the subexpression $(a^*)^a_x$ implies the contextual association of x with a, updating the initial context.

Such an operation allows us to syntactically compute the quotient.

Proposition 9.9. *Let E be a Σ,Γ-expression,* ctxt *be a context in* $\mathrm{Ctxt}(\Gamma,\Sigma)$ *and w be a word in Σ^*. Then:*

$$\bigcup_{(E',\mathrm{ctxt}')\in d^{\mathrm{ctxt}}_w(E)} \mathsf{L}^{\mathrm{ctxt}'}(E') = \bigcup_{(L',\mathrm{ctxt}')\in \mathsf{L}^{\mathrm{ctxt}}(E)} \{(w^{-1}(L'), \mathrm{ctxt}')\}$$

Proof. By a direct induction over the structure of words. □

Finally, the membership test of a word w can be performed as usual by first computing the derivation w.r.t. w, and then by determining the existence of a nullable derivative, as a direct corollary of Proposition 9.5 and Proposition 9.9.

Theorem 9.10. *Let E be a Σ,Γ-expression,* ctxt *be a context in* $\mathrm{Ctxt}(\Gamma,\Sigma)$ *and w be a word in Σ^*. Then the two following conditions are equivalent:*

- $\exists (L, \_) \in \mathsf{L}^{\mathrm{ctxt}}(E) \mid w \in L$,
- $\exists (E', \mathrm{ctxt}') \in d^{\mathrm{ctxt}}_w(E) \mid \mathrm{Null}^{\mathrm{ctxt}'}(E') \neq \emptyset$.

We have shown how to compute the derivatives and solve the membership test in a classical way. Let us show how to embed the context computation in a convenient monad, in order to generalize the definitions to other structure than sets.

9.5. The StateT Monad transformer

It is well known that the composition of two functors is a functor. However, the composition of two monads is not necessarily a monad (*e.g.* the composition of the `Set` monad with itself [22]). However, ones can consider particular combinations of these objects. Among those, well-known patterns are the monad transformers like the StateT Monad Transformer [13]. This combination allows us to mimick the use of global variables in a functional way. In our setting, it allows us to embed the context computation in an elegant way.

Let S be a set and M be a monad. We denote by $\texttt{StateT}(S, M)$ the following mapping:

$$\texttt{StateT}(S, M)(A) = S \to M(A \times S).$$

In other terms, $\texttt{StateT}(S, M)(A)$ is the set of functions from S to the monadic structure $M(A \times S)$ based on couples in the cartesian product $(A \times S)$.

The mapping $\texttt{StateT}(S, M)$ can be used to define a functor: for any function f from a set A to a set B:

$$\texttt{StateT}(S, M)(f)(\text{state})(s) = M(\lambda(a, s) \to (f(a), s))(\text{state}(s)).$$

It can also be used to define a monad: for any function f from a set A to the set $\texttt{StateT}(S, M)(B)$:

$$\begin{aligned} \texttt{pure}(a) &= \lambda s \to \texttt{pure}(a, s) \\ \texttt{bind}(f)(\text{state})(s) &= \text{state}(s) \gg\!= \lambda(a, s') \to f(a)(s') \end{aligned}$$

9.6. Monadic definitions

The previous definitions associated with capture-group expressions can be equivalently restated using the StateT monad transformer specialised with the `Set` monad.

Let us first consider the following claims where $M = \texttt{StateT}(\text{Ctxt}(\Gamma, \Sigma), \texttt{Set})$, allowing us to bring closer M and the previous notion of monadic support:

- $\mathbb{R} = (M(\mathbb{1}), +, \times, 1, 0)$ is a semiring by setting:

$$\begin{aligned} f_1 + f_2 &= \lambda s \to f_1(s) \cup f_2(s), & f_1 \times f_2 &= f_1 \gg\!= \lambda\_ \to f_2, \\ 1 &= \lambda s \to \{(\top, s)\} = \text{pure}(\top), & 0 &= \lambda s \to \emptyset, \end{aligned}$$

- $\mathbb{M} = (M(\text{Exp}(\Sigma)), \pm, \underline{0})$ is a monoid by setting:

$$\pm = +, \qquad \underline{0} = 0,$$

- $(\mathbb{M}, \ltimes)$ is a $\text{Exp}(\Sigma)$-right-semimodule by setting:

$$f \ltimes F = \lambda s \to \bigcup_{(E, \text{ctxt}) \in f(s)} \{(E \cdot F, \text{ctxt})\},$$

- $(\mathbb{M}, \triangleright)$ is a $\mathbb{R}$-left-semimodule by setting:

$$f_1 \triangleright f_2 = f_1 \gg\!= \lambda\_ \to f_2.$$

Then, the nullable predicate formulae can be equivalently restated as an element in $\texttt{StateT}(\text{Ctxt}(\Gamma, \Sigma), \texttt{Set})(\mathbb{1})$, which is equal by definition to $\text{Ctxt}(\Gamma, \Sigma) \to \texttt{Set}(\mathbb{1} \times \text{Ctxt}(\Gamma, \Sigma))$, isomorphic to $\text{Ctxt}(\Gamma, \Sigma) \to \texttt{Set}(\text{Ctxt}(\Gamma, \Sigma))$. It can inductively be computed as follows:

$$\begin{aligned} \text{Null}(\varepsilon) &= 1 & \text{Null}(\emptyset) &= 0 \\ \text{Null}(a) &= 0 & \text{Null}(E + F) &= \text{Null}(E) + \text{Null}(F) \\ \text{Null}(E \cdot F) &= \text{Null}(E) \times \text{Null}(F) & \text{Null}(E^*) &= 1 \end{aligned}$$

$$\mathrm{Null}(x)(\mathrm{ctxt}) = \begin{cases} \mathrm{pure}((\top, \mathrm{ctxt})) & \text{if } \mathrm{ctxt}(x) = \mathtt{Just}(\varepsilon), \\ \emptyset & \text{otherwise,} \end{cases}$$
$$\mathrm{Null}((E)_x^u)(\mathrm{ctxt}) = \mathrm{Set}(\lambda(\top, \mathrm{ctxt}') \to (\top, [\mathrm{ctxt}']_{x \leftarrow u}))(\mathrm{Null}(F)(\mathrm{ctxt})),$$

where E and F are two Σ, Γ-expressions, a is a symbol in Σ, x is a variable in Γ and u is in Σ^*. Notice that these formulae are the same as the ones in Definition 3.3 as far as classical operators are concerned, and that these formulae can be easily generalized to other convenient monads than Set. Moreover, the derivative of an expression is an element in $\mathtt{StateT}(\mathrm{Ctxt}(\Gamma, \Sigma), \mathrm{Set})(\mathrm{Exp}(\Sigma, \Gamma))$:

$$\mathrm{d}_a(\varepsilon) = \underline{0} \qquad \mathrm{d}_a(\emptyset) = \underline{0}$$
$$\mathrm{d}_a(b) = \begin{cases} \underline{0} & \text{if } a \neq b, \\ \mathtt{pure}(\varepsilon) & \text{otherwise,} \end{cases} \qquad \mathrm{d}_a(E + F) = \mathrm{d}_a(E) \pm \mathrm{d}_a(F)$$
$$\mathrm{d}_a(E \cdot F) = \mathrm{d}_a(E) \ltimes F + \mathrm{Null}(E) \triangleright \mathrm{d}_a(F) \qquad \mathrm{d}_a(E^*) = \mathrm{d}_a(E) \ltimes E^*$$

$$\mathrm{d}_a((E)_x^u) = \mathtt{StateT}(\mathrm{Ctxt}(\Gamma, \Sigma), \mathrm{Set})(\lambda F \to (F)_x^{ua})(\mathrm{d}_a(E))$$
$$\mathrm{d}_a(x)(\mathrm{ctxt}) = \begin{cases} \mathrm{pure}((w, \mathrm{ctxt})) & \text{if } \mathrm{ctxt}(x) = \mathtt{Just}(aw), \\ \emptyset & \text{otherwise,} \end{cases}$$

where E and F are two Σ, Γ-expressions, a is a symbol in Σ, x is a variable in Γ and u is in Σ^*. Once again, notice that these formulae are the same that the ones in Definition 5.1 as far as classical operators are concerned, and that these formulae can be easily generalized to other convenient monads than Set.

Finally, the derivation w.r.t. a word is monadically defined as in previous sections:

$$d_\varepsilon(E) = \mathtt{pure}(E), \qquad d_{av}(E) = d_a(E) \gg\!= d_v,$$

and the membership test of a word w can be equivalently rewritten as follows:

$$(d_w(E) \gg\!= \mathrm{Null})(\lambda\_ \to \mathrm{Nothing}) \neq \emptyset.$$

10. Conclusion and perspectives

In this paper, we achieved the first step of our plan to unify the derivative computation over word expressions. Monads are indeed useful tools to abstract the underlying computation structures and thus may allow us to consider some other functionalities, such as capture groups via the well-known StateT monad transformer [13]. We aim to study the conditions satisfying by monads that lead to finite set of derivated terms, and to extend this method to tree expressions using enriched categories. Finally, we plan to extend monadic derivation to other underlying monads for capture groups, linear combinations for example.

References

[1] S. Kleene, Representation of events in nerve nets and finite automata. Automata studies. *Ann. Math. Stud.* **34** (1956) 3–41.
[2] M.P. Schützenberger, On the definition of a family of automata. *Inf. Control.* **4** (1961) 245–270.
[3] G. Berry and R. Sethi, From regular expressions to deterministic automata. *Theoret. Comput. Sci.* **48** (1986) 117–126.
[4] P. Caron and M. Flouret, From glushkov wfas to k-expressions. *Fundam. Informaticae* **109** (2011) 1–25.

[5] J.-M. Champarnaud, É. Laugerotte, F. Ouardi and D. Ziadi, From regular weighted expressions to finite automata. *Int. J. Found. Comput. Sci.* **15** (2004) 687–700.
[6] J.A. Brzozowski, Derivatives of regular expressions. *J. ACM* **11** (1964) 481–494.
[7] V.M. Antimirov, Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.* **155** (1996) 291–319.
[8] S. Lombardy and J. Sakarovitch, Derivatives of rational expressions with multiplicity. *Theor. Comput. Sci.* **332** (2005) 141–177.
[9] M. Sulzmann and K. Zhuo Ming Lu, POSIX regular expression parsing with derivatives, in FLOPS. Vol. 8475 of *Lecture Notes in Computer Science*. Springer (2014) 203–220.
[10] S. Attou, L. Mignot, C. Miklarz and F. Nicart, Monadic expressions and their derivatives, in NCMA. Vol. 367 of *EPTCS* (2022) 49–64.
[11] T. Colcombet and D. Petrisan, Automata and minimization. *SIGLOG News* **4** (2017) 4–27.
[12] L. Mignot, *Une proposition d'implantation des structures d'automates, d'expressions et de leurs algorithmes associés utilisant les catégories enrichies (in French)*. Habilitation à diriger des recherches, Université de Rouen normandie, Décembre 2020. 212 pages.
[13] M.P. Jones, Functional programming with overloading and higher-order Polymorphism, in Adv. Func. Prog.. Vol. 925 of *LNCS*. Springer (1995) 97–136.
[14] S. Mac Lane, Categories for the Working Mathematician, Vol. 5. Springer Science & Business Media (2013).
[15] J.-L. Loday and B. Vallette, Algebraic Operads, Vol. 346. Springer Science & Business Media (2012).
[16] J.P. May, The Geometry of Iterated Loop Spaces. Vol. 271 of *Lecture Notes in Mathematics*. Springer, Berlin, New York (1972).
[17] H. Kleisli, Every standard construction is induced by a pair of adjoint functors. *Proc. Am. Math. Soc.* **16** (1956) 544–546.
[18] R.A. Eisenberg and S. Weirich, Dependently typed programming with singletons, in Haskell. ACM (2012) 117–130.
[19] L. Mignot, Monadic derivatives. https://github.com/LudovicMignot/MonadicDerivatives, 2022.
[20] IEEE Std 1003.1 – posix regular expressions. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.
[21] Pcre – perl compatible regular expressions. https://www.pcre.org.
[22] B. Klin and J. Salamanca, Iterated covariant powerset is not a monad. *Electr. Notes Theor. Comput. Sci.* **341** (2018) 261–276.