Monadic Expressions and their Derivatives

Samira Attou
LITIS,
Université de Rouen Normandie,
Avenue de l'Université,
76801 Saint-Étienne-du-Rouvray, France
samira.attou@univ-rouen.fr

Ludovic Mignot
GR^{2}IF,
Université de Rouen Normandie,
Avenue de l'Université,
76801 Saint-Étienne-du-Rouvray, France
ludovic.mignot@univ-rouen.fr

Clément Miklarz
GR^{2}IF,
Université de Rouen Normandie,
Avenue de l'Université,
76801 Saint-Étienne-du-Rouvray, France
clement.miklarz1@univ-rouen.fr

Florent Nicart
GR^{2}IF,
Université de Rouen Normandie,
Avenue de l'Université,
76801 Saint-Étienne-du-Rouvray, France
florent.nicart@univ-rouen.fr

We propose another interpretation of well-known derivatives computations from regular expressions, due to Brzozowski, Antimirov or Lombardy and Sakarovitch, in order to abstract the underlying data structures (*e.g.* sets or linear combinations) using the notion of monad. As an example of this generalization advantage, we introduce a new derivation technique based on the graded module monad.

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.

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 [10, 17] by computing a finite automaton, such as the position automaton [8, 2, 4, 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 [3] shows how to

H. Bordihn, G. Horváth, G. Vaszil (Eds.): 12th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2022)
EPTCS 367, 2022, pp. 49–64, doi:10.4204/EPTCS.367.4

© S.Attou et al.
This work is licensed under the
Creative Commons Attribution License.

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 [1] modifies this method in order to produce sets of expressions instead 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 $d_w(E)$. If 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. $|E|$. Finally, Lombardy and Sakarovitch [12] extends these methods to expressions with multiplicities.

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, 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 [6], by reinterpreting the work started in [15]. 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 [15] 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.

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 [9].

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

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

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. Finally, our implementation is presented in Section 8.

2 Preliminaries

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 an equivalent notation for a function f.

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 $\mathbb{K}$ where $\mathbb{K} = (K,\times,+,1,0)$, 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, for any function f in $K^n \to K$, we set:

$$(f(S_1,\ldots,S_n))(w) = f(S_1(w),\ldots,S_n(w)). \tag{1}$$

A *functor*[2] 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*[3] M is a functor endowed with two (families of) functions

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

such that the three following conditions are satisfied:

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

Example 1. *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*
$$\texttt{Maybe}(f)(\texttt{Just}(s)) = \texttt{Just}(f(s)), \qquad \texttt{Maybe}(f)(\texttt{Nothing}) = \texttt{Nothing}$$

[2] More precisely, a functor over a subcategory of the category of sets.
[3] More precisely, a monad over a subcategory of the category of sets.

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

To compact equations, we use the following operators for any monad M:

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

If $<\$>$ can be used to lift unary functions to the monadic level, $\gg\!=$ 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:

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

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 in a coherent way classical expressions to monadic expressions, where the type of the weights is therefore given by the ambient monad.

3 Monadic Expressions

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.

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 by itself.

Definition 1. *Let M be a monad. An M-*monadic expression *E over an alphabet Σ is inductively defined as follows:*

$$E = a, \quad E = \varepsilon, \quad E = \emptyset,$$
$$E = E_1 + E_2, \quad E = E_1 \cdot E_2, \quad E = E_1^*,$$
$$E = \alpha \odot E_1, \quad E = E_1 \odot \alpha, \quad E = f(E_1, \ldots, E_n),$$

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 4. *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 2. *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:*

$$\texttt{Null}(\varepsilon) = 1, \quad \texttt{Null}(\emptyset) = 0,$$
$$\texttt{Null}(a) = 0, \quad \texttt{Null}(E_1 + E_2) = \texttt{Null}(E_1) + \texttt{Null}(E_2),$$
$$\texttt{Null}(E_1 \cdot E_2) = \texttt{Null}(E_1) \times \texttt{Null}(E_2), \quad \texttt{Null}(E_1^*) = \texttt{Null}(E_1)^\star,$$
$$\texttt{Null}(\alpha \odot E_1) = \alpha \times \texttt{Null}(E_1), \quad \texttt{Null}(E_1 \odot \alpha) = \texttt{Null}(E_1) \times \alpha,$$
$$\texttt{Null}(f(E_1, \ldots, E_n)) = f(\texttt{Null}(E_1), \ldots, \texttt{Null}(E_n)),$$

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, we restrict the nullability value computation to expressions where a starred subexpression admits a *null nullability value*. In order to compute it, let us consider the Maybe monad, allowing us to elegantly deal with such a partial function.

Definition 3. *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:*

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

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 `Nothing`, therefore if it is a value $\texttt{Just}(v)$; in this case, v is its nullability value, denoted by $\texttt{Null}(E)$ (by abuse).

Definition 4. *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:*

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

$$S(E_1+E_2)=S(E_1)+S(E_2),\quad S(E_1\cdot E_2)=S(E_1)\times S(E_2),\quad S(E_1^*)=(S(E_1))^\star,$$
$$S(\alpha\odot E_1)(w)=\alpha\times S(E_1)(w),\quad S(E_1\odot\alpha)(w)=S(E_1)(w)\times\alpha,$$
$$S(f(E_1,\ldots,E_n))=f(S(E_1),\ldots,S(E_n)),$$

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 $\texttt{weight}_w(E)=S(E)(w)$. The nullability of a proper expression is the weight it associates with ε, following Definition 3 and Definition 4.

Proposition 1. *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:*

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

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 this computation 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* [11, 13] 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)[4] with $0<j\leq k$ in $O_k\times O_i\to O_{k+i-1}$ such that for any elements $p_1\in O_m$, $p_2\in O_n$, $p_3\in O_p$:

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

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 $O_k\times O^k$,

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

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

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.

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

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

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 4. Therefore, in the following we consider a restricted kind of monads.

A *monadic support* is a structure $(M, +, \times, 1, 0, \pm, \underline{0}, \ltimes, \triangleright, \triangleleft, \circledast)$ 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,
- $(\mathbb{M}, \triangleright)$ is a $\mathbb{R}$-left-semimodule,
- $(\mathbb{M}, \triangleleft)$ 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 $(M, +, \times, 1, 0, \pm, \underline{0}, \ltimes, \triangleright, \triangleleft, \circledast)$ endowed with a function `toExp` from $M(\mathrm{Exp}(\Sigma))$ to $\mathrm{Exp}(\Sigma)$ satisfying the following conditions:

$$\mathtt{weight}_w(\mathtt{toExp}(m)) = m \gg\!= \mathtt{weight}_w \tag{2}$$

$$\mathtt{toExp}(m \ltimes F) = \mathtt{toExp}(m) \cdot F, \tag{3}$$

$$\mathtt{toExp}(m \pm m') = \mathtt{toExp}(m) + \mathtt{toExp}(m'), \tag{4}$$

$$\mathtt{toExp}(m \triangleright x) = \mathtt{toExp}(m) \odot x, \tag{5}$$

$$\mathtt{toExp}(x \triangleleft m) = x \odot \mathtt{toExp}(m), \tag{6}$$

$$\mathtt{toExp}(f \circledast (m_1, \ldots, m_n)) = f(\mathtt{toExp}(m_1), \ldots, \mathtt{toExp}(m_n)). \tag{7}$$

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

Example 5 (The `Maybe` support)**.**

$$\mathtt{toExp}(\mathtt{Nothing}) = 0, \quad \mathtt{toExp}(\mathtt{Just}(E)) = E,$$

$$\mathtt{Nothing} + m = m, \qquad \mathtt{Nothing} \times m = \mathtt{Nothing},$$

$$m + \mathtt{Nothing} = m, \qquad m \times \mathtt{Nothing} = \mathtt{Nothing},$$

$$\mathtt{Just}(\top) + \mathtt{Just}(\top) = \mathtt{Just}(\top), \qquad \mathtt{Just}(\top) \times \mathtt{Just}(\top) = \mathtt{Just}(\top),$$

$$\mathtt{Nothing} \pm m = m, \quad m \pm \mathtt{Nothing} = m, \quad \mathtt{Just}(E) \pm \mathtt{Just}(E') = \mathtt{Just}(E + E'),$$

$$1 = \mathtt{Just}(\top), \quad 0 = \mathtt{Nothing}, \quad \underline{0} = \mathtt{Nothing},$$

$$m \ltimes F = (\lambda E \to E \cdot F) \mathbin{<\$>} m,$$

$$m \triangleright m' = m \gg\!= (\lambda x \to m'), \quad m \triangleleft m' = m' \gg\!= (\lambda x \to m),$$

$$f \circledast (m_1, \ldots, m_n) = \mathtt{pure}(f(\mathtt{toExp}(m_1), \ldots, \mathtt{toExp}(m_n))).$$

Example 6 (The `Set` support)**.**

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

Example 7 (The $\texttt{LinComb}(\mathbb{K})$ support)**.**

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

5 Monadic Derivatives

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

Definition 5. *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:*

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

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 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:*

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

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 ([1, 3, 12]). Therefore, let us consider

this last case.

$$\begin{aligned}
d_a(f(E_1,\ldots,E_n)) \gg= \mathtt{weight}_w \\
&= \mathtt{weight}_w(\mathtt{toExp}(d_a(f(E_1,\ldots,E_n)))) && \text{(Eq (2))}\\
&= \mathtt{weight}_w(\mathtt{toExp}(f \circledast (d_a(E_1),\ldots,d_a(E_n))) && \text{(Def 5))}\\
&= \mathtt{weight}_w(f(\mathtt{toExp}(d_a(E_1)),\ldots,\mathtt{toExp}(d_a(E_n)))) && \text{(Eq (7))}\\
&= f(\mathtt{weight}_w(\mathtt{toExp}(d_a(E_1))),\ldots,\mathtt{weight}_w(\mathtt{toExp}(d_a(E_n)))) && \text{(Def 4, Eq (1))}\\
&= f(d_a(E_1) \gg= \mathtt{weight}_w,\ldots,d_a(E_n) \gg= \mathtt{weight}_w) && \text{(Eq (2))}\\
&= f(\mathtt{weight}_{aw}(E_1),\ldots,\mathtt{weight}_{aw}(E_n)) && \text{(Ind. hyp.)}\\
&= \mathtt{weight}_{aw}(f(E_1,\ldots,E_n)) && \text{(Def 4, Eq (1))}
\end{aligned}$$

□ □

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

Definition 6. *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 2, that the derivatives computation can be used to define a syntactical computation of the weight of a word associated with an expression.

Theorem 1. *Let E be an M-monadic expression over an alphabet Σ and w be a word in Σ^*. Then:*

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

Notice that, restraining monadic expressions to regular ones,

- the `Maybe` support leads to the classical derivatives [3],
- the `Set` support leads to the partial derivatives [1],
- the `LinComb` support leads to the derivatives with multiplicities [12].

Example 8. *Let us consider the function* `ExtDist` *defined in Example 4 and the* $\mathtt{LinComb}(\mathbb{N})$*-monadic expression* $E = \mathtt{ExtDist}(a^*b^* + b^*a^*, b^*a^*b^*, a^*b^*a^*)$.

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

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

6 Automata Construction

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

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

Given a category $\mathscr{C}$, a $\mathscr{C}$-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 $\text{Obj}(\mathscr{C})$,
- Q is the state object, in $\text{Obj}(\mathscr{C})$,
- F is the final object, in $\text{Obj}(\mathscr{C})$,
- i is the initial morphism, in $\text{Hom}_{\mathscr{C}}(I,Q)$,
- δ is the transition function, in $\Sigma \longrightarrow \text{Hom}_{\mathscr{C}}(Q,Q)$,
- f is the value morphism, in $\text{Hom}_{\mathscr{C}}(Q,F)$.

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

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

$$\texttt{weight}(w) = f \circ_{\mathscr{C}} \delta(w) \circ_{\mathscr{C}} i.$$

If the ambient category is the category of sets, and if $I = \mathbb{1}$, the weight of a word is equivalently an element of 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.

Given a monad M, the Kleisli composition of two morphisms $f \in \text{Hom}_{\mathscr{C}}(A,B)$ and $g \in \text{Hom}_{\mathscr{C}}(B,C)$ is the morphism $(f \mathrel{>\!=\!>} g)(x) = f(x) \mathrel{>\!>\!=} g$ in $\text{Hom}_{\mathscr{C}}(A,C)$. This composition defines a category, called the Kleisli category $\mathscr{K}(M)$ 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 `pure`.

Considering these categories:

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

all with $\mathbb{1}$ as both the initial object and the final object.

Furthermore, for a given expression E, if $i = \texttt{pure}(E)$, $\delta(a)(E') = \text{d}_a(E')$ and $f = \texttt{Null}$, 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 [3, 1, 12].

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

Example 9. *Considering the computations of Example 8, it can be shown that*

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

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.

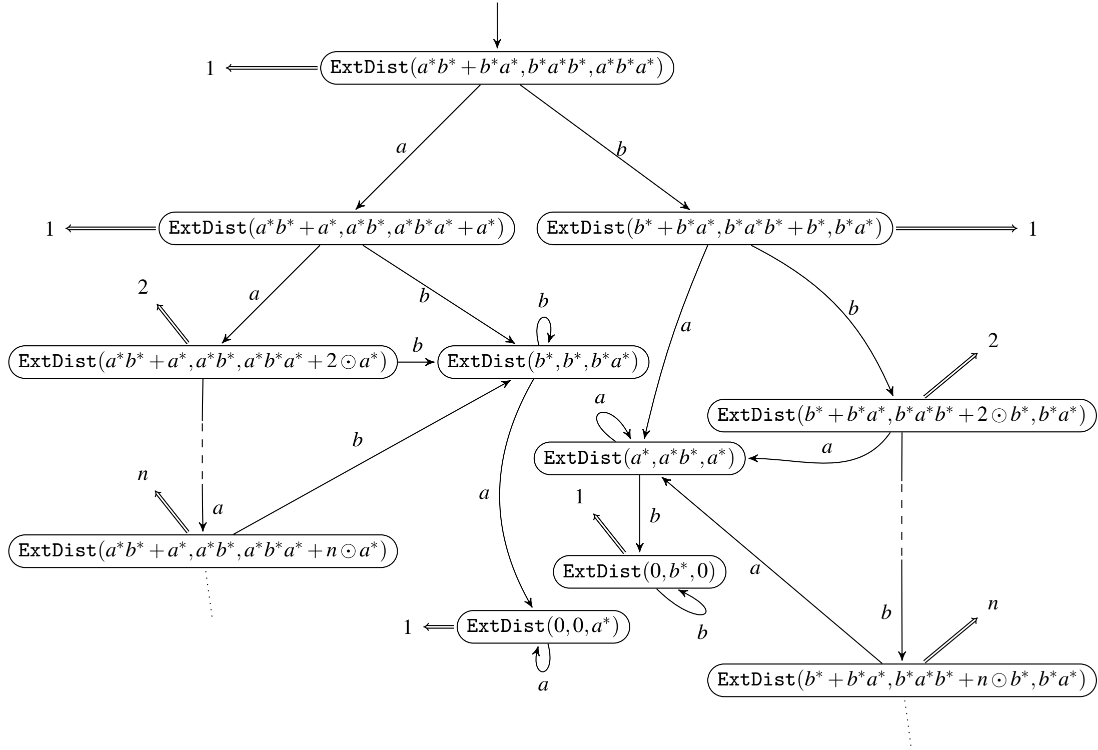

Figure 1: The (infinite) derivative weighted automaton associated with E.

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

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. 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[5] $\texttt{GradComb}(\mathbb{O},\mathbb{K})$ 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 $\texttt{toOp}$:

$$\begin{aligned}
&\texttt{toOp}((k_1,s_1) \boxplus \cdots \boxplus (k_n,s_n))\\
&\qquad = (\lambda(x_1,\dots,x_n) \to k_1 \times x_1 + \cdots + k_n \times x_n, (s_1,\dots,s_n)),\\
&\alpha(o,(\mathscr{L}_1,\dots,\mathscr{L}_n)) = (o \circ (o_1,\dots,o_n),(s_{1,1},\dots,s_{1,i_1},\dots,s_{n,1},\dots,s_{n,i_n}))
\end{aligned}$$

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

Consequently, we can define the monadic functions as follows:

$$\begin{aligned}
\texttt{pure}(s) &= (\mathrm{id},(1,s)),\\
(o,(\mathscr{L}_1,\dots,\mathscr{L}_n)) \gg\!= f &= \alpha(o,(\mathscr{L}_1,\dots,\mathscr{L}_n)) \gg\!= f
\end{aligned}$$

where the second occurrence of $\gg\!=$ is the monadic function associated with the monad $\texttt{GradMod}(\mathbb{O})$.

Let us finally define an expressive support for this monad:

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

Example 10. *Let us consider that two elements in* $\texttt{GradComb}(\mathbb{O},\mathbb{K})(\mathrm{Exp}(\Sigma))$ *are equal 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*

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

Example 8.

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

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 absorbed in the transition structure. 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.

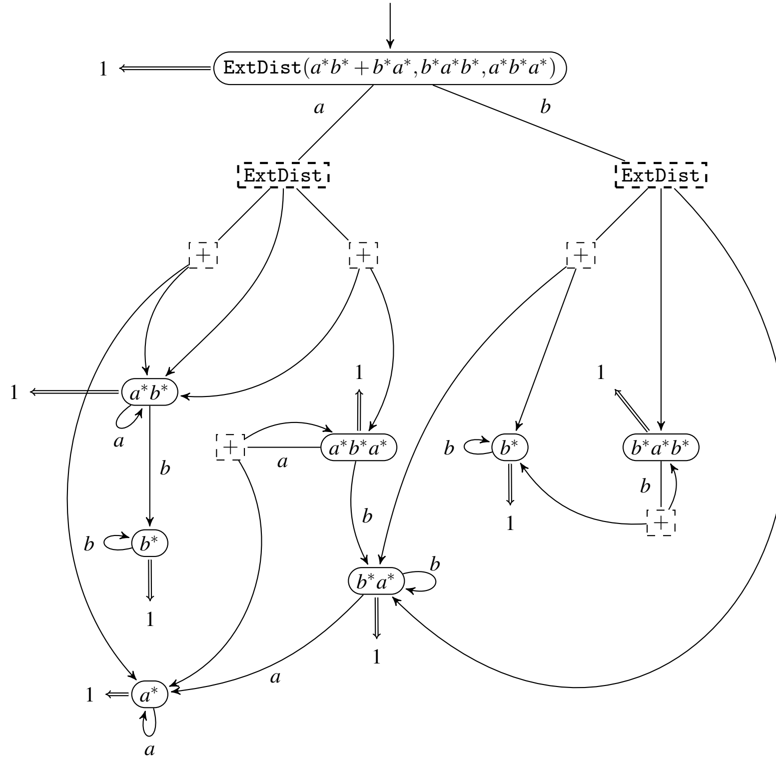

Figure 2: The Associated Derivative Automaton of ExtDist$(a^*b^* + b^*a^*, b^*a^*b^*, a^*b^*a^*)$.

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

Example 11. *Let us consider the expression E of Example 8 and the expression $F = E \cdot c^*$. It can be shown that*

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

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.

8 Haskell Implementation

The notions described in this paper have been implemented in Haskell, as follows:

- 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 [7], allowing, for example, determining the type of the monads or the arity of the functions involved at run-time;
- An application is available here [14, 16] illustrating the computations:
  - the backend uses servant to define an API hosted by Heroku;
  - 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 input `ExtDist(a*.b*+b*.a*,b*.a*.b*,a*.b*.a*)`.

9 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 [9], that we plan to study in a future work. We also 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.

References

[1] Valentin M. Antimirov (1996): *Partial Derivatives of Regular Expressions and Finite Automaton Constructions*. Theor. Comput. Sci. 155(2), pp. 291–319, doi:10.1016/0304-3975(95)00182-4.

[2] Gerard Berry & Ravi Sethi (1986): *From regular expressions to deterministic automata*. Theoretical computer science 48, pp. 117–126, doi:10.1016/0304-3975(86)90088-5.

[3] Janusz A. Brzozowski (1964): *Derivatives of Regular Expressions*. J. ACM 11(4), pp. 481–494, doi:10.1145/321239.321249.

[4] Pascal Caron & Marianne Flouret (2011): *From Glushkov WFAs to K-Expressions*. Fundam. Informaticae 109(1), pp. 1–25, doi:10.3233/FI-2011-427.

[5] Jean-Marc Champarnaud, Éric Laugerotte, Faissal Ouardi & Djelloul Ziadi (2004): *From Regular Weighted Expressions To Finite Automata*. Int. J. Found. Comput. Sci. 15(5), pp. 687–700, doi:10.1142/S0129054104002698.

[6] Thomas Colcombet & Daniela Petrisan (2017): *Automata and minimization*. SIGLOG News 4(2), pp. 4–27, doi:10.1145/3090064.3090066.

[7] Richard A. Eisenberg & Stephanie Weirich (2012): *Dependently typed programming with singletons*. In: Haskell, ACM, pp. 117–130, doi:10.1145/2364506.2364522.

[8] Victor Mikhaylovich Glushkov (1961): *The abstract theory of automata*. Russian Mathematical Surveys 16(5), p. 1, doi:10.1070/rm1961v016n05abeh004112.

[9] Mark P. Jones (1995): *Functional Programming with Overloading and Higher-Order Polymorphism*. In: Adv. Func. Prog., LNCS 925, Springer, pp. 97–136, doi:10.1007/3-540-59451-5_4.

[10] S. Kleene (1956): *Representation of events in nerve nets and finite automata*. Automata Studies Ann. Math. Studies 34, pp. 3–41, doi:10.1515/9781400882618-002. Princeton U. Press.

[11] Jean-Louis Loday & Bruno Vallette (2012): *Algebraic operads*. 346, Springer Science & Business Media, doi:10.1007/978-3-642-30362-3_5.

[12] Sylvain Lombardy & Jacques Sakarovitch (2005): *Derivatives of rational expressions with multiplicity*. Theor. Comput. Sci. 332(1-3), pp. 141–177, doi:10.1016/j.tcs.2004.10.016.

[13] J Peter May (2006): *The geometry of iterated loop spaces*. 271, Springer, doi:10.1007/BFb0067491.

[14] Ludovic Mignot: *Application: Monadic derivatives*. http://ludovicmignot.free.fr/programmes/monDer/index.html. Accessed: 2022-05-26.

[15] Ludovic Mignot (2020): *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, doi:10.48550/arXiv.2012.10641. Available at https://arxiv.org/abs/2012.10641. 212 pages.

[16] Ludovic Mignot (2022): *Monadic derivatives*. https://github.com/LudovicMignot/MonadicDerivatives.

[17] Marcel Paul Schützenberger (1961): *On the definition of a family of automata*. Inf. Control. 4(2-3), pp. 245–270, doi:10.1016/S0019-9958(61)80020-X.