Another Look at Derivatives of Regular Expressions, with an Experimental Evaluation

Baudouin Le Charlier
Université catholique de Louvain, Belgium
baudouin.lecharlier@uclouvain.be

September 22, 2023

Abstract

In this paper, I first discuss and clarify the notion of *derivative of a regular expression* introduced by Brzozowski in [5]. Next, I define a notion of *normalized expression*, which allows me to present an efficient algorithm for computing the *syntactic derivatives* of an expression as well as a *set of equations* relating the syntactic derivatives to their direct derivatives. Such a set of equations defines a deterministic finite automata (DFA) for recognizing the language defined by the normalized expression. Normalized expressions are implemented in a global data structure, called *the background*, which maintains equivalence classes of expressions denoting the same regular language, as well as equations relating expressions to their derivatives. This makes it possible to efficiently reduce the set of equations computed by the basic algorithm. I also propose an experimental evaluation of the whole machinery, showing its practical usefulness, and a comparison with other derivation algorithms, including Brzozowski's. Additionally, the appendix details five significant examples and provides some proofs omitted from the main document.

1 Introduction

The notion of derivative of a regular expression was introduced by Brzozowski in [5], as a conceptual tool to build a deterministic finite automaton (DFA) recognizing the regular language defined by the expression. However, the proposed method suffers from a fundamental ambiguity: The fact that the derivation of regular languages applies to sets of chain, not to regular expressions. Let S be a set of chains. The derivative of S with respect to a chain w is the set $\{u \mid \exists s \in S : w.u = s\}$.[1] This notion applies to any set of chains, not to regular languages only. Thus, Brzozowski provides explicit rules to calculate the derivatives of regular languages based on regular expressions defining them. This is similar to giving derivation formulas for mathematical functions. The method is useful because a regular language has a finite number of derivatives that can be viewed as the nodes of a DFA recognizing the language. This is proven in [5] by induction on the length of regular expressions. Nevertheless, the set of regular expressions $D_w(E)$ to be computed by the method is infinite because the set of all chains w is infinite: So, a strict application of the derivation rules gives raise to longer and longer formulas. Fortunately, it is possible to simplify the regular expressions with rules involving the operator + and the symbols 0 and 1, in such a way that the set of expressions $D_w(E)$ (for all w) becomes finite, making the process terminate. But the proof of this fact (Theorem 5.2, in [5], page 493) is sketchy.

[1] I freely use notations defined later on, in Section 2.

Some years later, Conway proposes a differential calculus of events (i.e. regular languages) ([8], Chapter 5). His derivation rules are the same as Brzozowski's but the rules apply to events, not to expressions.[2] He proves that the sets of all derivatives of an event is finite, giving an upper bound on their number much smaller than Brzozowski's but he does not make an attempt to propose a terminating algorithm to compute them (as a set of regular expressions). This is apparently because he thinks that one can rely on "expertise" to compute the derivatives, simplify them, and see that they are equal (as events). For instance, on top of page 43, he proposes an example (see Appendix E), and he writes : "In this case, as in many others, the process gives the minimal machine directly to anyone skilled in input differentiation". In my opinion, it is for a similar reason that Brzozowski was satisfied with a rough proof of his Theorem 5.2: In practice, he could rely on expertise to simplify expressions and detect their equivalence.

Twenty-five years later, Antimirov makes an important new contribution in [3]. He introduces the notion of partial derivative of a regular expression, a purely syntactic notion, this time. He proves that the set of partial derivatives of an expression is finite and, in fact, quite small. Every derivative of the regular set denoted by the expression can be syntactically represented by an expression of the form $P_1 + ... + P_n$ $(n \geq 0)$, where the P_i are partial derivatives. This can be used to design a variant of Brzozowski's method.

In this paper, I introduce a notion of syntactic derivative strongly related to Antimirov's work. However, there are several important differences:

1. I propose to work with a notion of *normalized expression*. Using normalized expressions in Brzozowski's algorithm ensures termination automatically.

2. I define a notion of partial derivative of a normalized expression with a slightly different approach than Antimirov. I prove that they are finite in number in a different way, by showing that they form a subset of an obviously finite set of expressions. I precisely define the set of syntactic derivatives of a normalized expression, which are all of the form $P_1 + ... + P_n$ $(n \geq 0)$, where the P_i are partial derivatives, as in Antimirov's proposal.

3. I propose an efficient algorithm to compute all the syntactic derivatives of an expression. This is different from Antimirov's work, which concentrates on computing the partial derivatives, to obtain a nondeterministic finite automata (NFA). My algorithm can be modified very simply to get such a NFA but this version is not the fundamental algorithm.

This paper also contains the following contributions:

1. The DFAs computed by my new algorithm (or by Brozozowski's) in fact are sets of equations of the form $E = o_E + \ldots + x.E_x + \ldots$, where 0_E is either 0 or 1, the x's are letters, and E and the E_x's are regular expressions (the notation is borrowed from [8], page 41). I use a global data structure, called *the background*, inside which normalized expressions are uniquely represented and gathered in equivalence classes. Equations are built with expressions that are the representatives of their equivalence classes. This allows me to merge equations sharing the same left part (E) or the same right part ($o_E + \ldots + x.E_x + \ldots$), thereby reducing the size of the DFAs using these equations. More information about the background can be found in [6]. But, here, I concentrate on how it can be used to improve the fundamental algorithm.

[2] In fact, Brzozowski does not make a clear distinction between regular languages and regular expressions, writing equalities such as $P = Q$ where P denotes a set of chains and Q an expression.

2. I give an experimental evaluation of my algorithm on several sets of randomy generated expressions. The evaluation also involves a comparison with a version of Brzozowski's algorithm working on normalized expressions, and with a modified version computing partial derivatives. I report on the execution times and the size of the sets of equations. I also propose and evaluate two optimizations of the fundamental algorithm. Experiments on examples from the literature are also presented in the appendix.

The paper is organized as follows: Section 2 defines the notion of *normalized expression* as well as the operations applicable to them. Section 3 defines the *syntactic derivatives* of a normalized expression and proves that they correctly represent the derivatives of the regular set denoted by the expression. An algorithm to compute the *direct* derivatives (i.e. the derivatives with respect to a single letter x) is presented with an elegant correctness proof. Then, the notion of *partial derivative* is introduced as a tool to prove that the set of all syntactic derivatives of an expression is finite.[3] It is proved that the set of partial derivatives is finite by a new method (different from Antimirov's method). Then, the algorithm to compute all syntactic derivatives of a given normalized expression is described and proved correct. Section 4 defines the notion called the *background* and its operations. It is shown how the algorithm computing all syntactic derivatives can be modified to take advantage of it. Section 5 provides the experimental evaluation. Section 6 contains the conclusion with some final reflections on previous related work. Some proofs omitted in the main paper are given in the appendix, which also contains five extensive examples illustrating the notions defined in the paper.

2 Normalized regular expressions

We assume a finite set *Letter* of *letters*. In practice, in this paper, only lowercase letters: a, b, ..., z are used. (For the examples, mainly a and b.) A chain of letters, denoted by t, u or w, is a sequence $x_1x_2 \ldots x_n$ where $x_1, x_2, \ldots, x_n$ are letters (n is the length of the chain). The empty chain ($n = 0$) is denoted by 1. We identify every letter x with the chain of length 1 containing x only. The set of all chains is denoted by $Letter^*$. The concatenation of two chains u and w, denoted by $u.w$ or, more simply, by $u\,w$ is the sequence of $y_1y_2 \ldots y_mx_1x_2 \ldots x_n$ where $u = y_1y_2 \ldots y_m$ and $w = x_1x_2 \ldots x_n$. Let S, S_1, and S_2 be sets of chains. The concatenation of S_1 and S_2, denoted by $S_1.S_2$, is the set of chains $\{w_1.w_2 \mid w_1 \in S_1 \;\&\; w_2 \in S_2\}$. The iteration of S, denoted by S^* is the set of chains $\{w_1 \,.\, w_2 \,.\, \ldots \,.w_n \mid w_1\,,\, w_2\,,\, \ldots\,, w_n \in S\ (n \geq 0)\}$.[4]

Definition 1. Expressions, normalized expressions, extended expressions

1. A regular expression E is either 0, or 1, or a letter, or a *union*, of the form $E_1 + E_2$, or a *concatenation*, of the form $E_1.E_2$,[5] or an *iteration*, of the form E_1^*, where E_1 and E_2 are (simpler) regular expressions. A regular expression denotes a set of chains of letters in the "obvious way" (see, e.g. [1, 8, 11]). This set is denoted by $\mathcal{L}\,E$ and is called a regular set.

2. A *normalized* regular expression is either 0, or 1, or a letter, or a *union*, of the form $E_1 + E_2 + \ldots + E_n$, where $n \geq 2$ and E_1, E_2, ..., E_n are syntactically different normalized expressions that are not unions, nor equal to 0, or a *concatenation*, of the form $E_1\,E_2$, where E_1 and E_2

[3] Remember that the remarkable proof of Conway does not apply to *syntactic* derivatives.

[4] The definitions above are given for the sake of completeness but I take for granted that the reader already knows what they are about.

[5] In practice, we often omit the operator . , writing simply $E_1\ E_2$.

Figure 1: Set of chains denoted by a normalized expression ($\mathcal{L}\, E$)

$$
\begin{array}{rcl}
\mathcal{L}\, 0 & = & \{\} \\
\mathcal{L}\, 1 & = & \{1\} \\
\mathcal{L}\, x & = & \{x\} \\
\mathcal{L}\, (E_1 + \ldots + E_n) & = & \mathcal{L}\, E_1 \cup \ldots \cup \mathcal{L}\, E_n \quad (n \geq 2) \\
\mathcal{L}(E_1 \,.\, E_2) & = & (\mathcal{L}\, E_1) \,.\, (\mathcal{L}\, E_2) \\
\mathcal{L}\, (E^*) & = & (\mathcal{L}\, E)^*
\end{array}
$$

are not equal to 0 or 1, and E_1 is not a concatenation, or an *iteration*, of the form E^*, where E is not equal to 0 or 1, and is not an iteration.

We sometimes write (meta-)expressions such as "$E_1 + E_2 + \ldots + E_n$, where $n \geq 0$". This means that the intended expression either is 0 (if $n = 0$), or is not a union, nor 0 (if $n = 1$), or is a union (if $n \geq 2$).

We assume a total ordering on the set of normalized expressions.[6] Using this ordering, we impose the additional constraint that, in a union $E_1 + E_2 + \ldots + E_n$, the sequence E_1, E_2, ..., E_n is stricly sorted, in ascending order.

A normalized regular expression represents a set of chains according to the rules given in Figure 1. This set is denoted by $\mathcal{L}\, E$ and obviously is a regular set. The notations used in Figure 1 assume that the symbols in the rules fulfill the conditions made precise in this paragraph. The same assumption must be made in other figures that appears later on.

3. An *extended* regular expression is of the form $(E_1\, \omega\, E_2)$, where E_1 and E_2 are either regular or extended regular expressions, and ω is a set operator such as $\cap$, $\setminus$, $\triangle$. They denote a set of chains of letters in the obvious way.

4. A *normalized extended* regular expression is of the form $(E_1 \setminus E_2)$, where E_1 and E_2 are either normalized or extended normalized expressions. [7]

5. The *size* or *length* of an expression, denoted by size(E), is the number of occurrences of symbols it is made of. A formal definition by induction on the syntactic structure of expressions is immediate.

(End of definition)

In this paper, I only consider normalized regular expressions. Classical expressions are useful mainly to allow the user to write expressions in the usual way, when providing them as input to the "system". They are automatically normalized before being processed by other algorithms. Notice that when writing normalized expression, I drop some parentheses, leaving to the reader the burden of parsing

[6] For simplicity, this ordering is fixed by the implementation (see Sections 4 and 5). It is different at each "run" of the program, depending on which expressions were first created. It will become clear later that his choice is the most efficient for implementing the basic operations on normalized expressions.

[7] For simplicity, extended expressions are not considered in this paper. They are taken into account in [6].

the expression properly. As a typical example, the expression

$$\text{b(a + b(1 + a + b*b))((a + b)a*)*}$$

has to be parsed as $E_1\,(E_2\,E_3)$ where $E_1 = \text{b}$, $E_2 = \text{a + b(1 + a + b*b)}$, and $E_3 = \text{((a + b)a*)*}$.

My choice of using normalized regular expressions should be contrasted with the choice of other authors, such as [3, 5], who check equivalence of regular expressions with respect to a congruence relation taking into account some of Kleene's classical axioms (see, e.g. [8], page 25). With normalized expressions, testing such equivalence boils down to checking syntactic equality.

Classical regular expressions are normalized thanks to three operations: union, concat, and iter. They take one or two normalized expressions, as argument(s), and they return a normalized expression denoting the union, the concatenation or the iteration of the language(s) denoted by their argument(s). For the sake of beauty, I usually write these operations as the infix operators $\oplus$ (union) and $\odot$ (concat), and the postfix operator $\star$ (iter). The operations are defined below (Definition 2). Since 0, 1, and the letters already are normalized, normalizing classical expressions can be recursively done, by induction on the structure of classical expressions. It is important to note that, in these definitions, the symbol =, used between normalized expressions, denotes strict syntactic equality.

Definition 2. Operations on normalized regular expressions

Let E, E_1, and E_2 be normalized expressions.

- The operation union

  1. $\text{union}(0, E) = \text{union}(E, 0) = E$
  2. Assume that E_1 and E_2 are different from 0.
     For $i = 1, 2$, let $S_i = \{E_{i\,1}, \ldots, E_{i\,n_i}\}$ where $E_i = E_{i\,1} + \ldots + E_{i\,n_i}$, if E_i is a union, and let $S_i = \{E_i\}$, otherwise. Let $F_1, \ldots, F_m$ be the strictly ordered sequence of normalized expressions such that $S_1 \cup S_2 = \{F_1, \ldots, F_m\}$. Then, by definition, $\text{union}(E_1, E_2)$ is the normalized expression $F_1 + \ldots + F_m$, if $m \geq 2$. Note that we can have $m = 1$. In that case, the result simply is F_1, which is not a union by definition of normalized expressions.

  It is clear, from this definition, that the operation union is associative and commutative, so that we can freely write $E_1 \oplus E_2 \ldots \oplus E_n$ without paying attention to the position of the expressions E_i in the whole expression. (Remember that union is an operation on normalized expressions, *not on sets of chains*; therefore, it was not obvious *a priori* that the two properties hold.) The operation also is idempotent: $E \oplus E = E$.

- The operation concat

  1. $\text{concat}(0, E) = \text{concat}(E, 0) = 0$
  2. $\text{concat}(1, E) = \text{concat}(E, 1) = E$
  3. Assume that E_1 and E_2 are different from 0 and 1.
     If E_1 is not a concatenation, then $\text{concat}(E_1, E_2)$ is the concatenation $E_1\,.\,E_2$. Otherwise, E_1 can be written as $F_1\,.\,F_2$, where F_1 is not a concatenation. In that case, $\text{concat}(E_1, E_2) = F_1\,.\,G$ where $G = \text{concat}(F_2, E_2)$.

The operation concat is associative so that we can write:

$$E_1 \odot E_2 \odot E_3 \;=\; (E_1 \odot E_2) \odot E_3 \;=\; E_1 \odot (E_2 \odot E_3).$$

As for the operation union this property is not completely obvious *a priori* since the result of the operation is a normalized expression, not a set of chains.

- The operation iter

  1. iter(0) = iter(1) = 1
  2. If E is an iteration, iter(E) $= E$.
  3. If E is not an iteration and is different from 0 and 1, iter(E) $= E^*$.

(End of definition)

3 Computing syntactic derivatives

In this section, I explain how *syntactic* derivatives of normalized expressions are computed. My method is more efficient than Brzozowski's method [5] as will be demonstrated in Section 5. It is also completely automatic to the contrary of the classical characterization of derivatives given in [8], which requires to use additional simplification rules, and is basically intended to be applied "by hand". It is very related to the method proposed in [3] but it deals with derivatives, from the start, not with partial derivatives. Partial derivatives are introduced later as a tool to prove that the set of derivatives of an expression is finite. My proof that the set of partial derivatives of an expression is finite is different and more intuitive, in my opinion, than the proof in [3], although Antimirov uses standard regular expressions, not normalized ones.

This section is structured as follows: In Subsection 3.1, the method is motivated by showing how the direct derivatives of a normalized expression can be computed, "by hand", just by (left) unfolding some sub-expressions of the form E^* to $1 + E\,E^*$, and normalizing the resulting expression. In Subsection 3.2, I give a precise definition of (my notion of) syntactic derivative. In Subsection 3.3, I propose an algorithm that efficiently computes all *direct* syntactic derivatives at once, and I prove its correctness. In Subsection 3.4, I define two new notions: *partial derivatives* and *left unfolded suffixes*, which are both special kinds of normalized expressions. I use the notion of left unfolded suffix to prove that the set of all partial derivatives of a given expression E is finite. I also show that every syntactic derivative of a normalized expression is equal to the union $E_1 \oplus \ldots \oplus E_n$ of a set of partial derivatives (i.e. $\{E_1\,,\,\ldots\,,\,E_n\}$), which proves that the set of all syntactic derivatives of a normalized expression is finite. Finally, in Subsection 3.5, I give a simple algorithm to compute all the syntactic derivatives of a normalized expression. This algorithm makes use of the algorithm described in Subsection 3.3.

3.1 Computing syntactic derivatives by left unfolding

I show on an example (from [8]) how the direct derivatives of a normalized expression can be computed using simple manipulations of regular expressions, notably by unfolding E^* to $1 + E\,E^*$. Let E be the regular expression (ab*a + ba*b)*(1 + ab* + ba*).

We can write the following equalities:[8]

$$\begin{aligned} E &= (\mathrm{ab^*a} + \mathrm{ba^*b})^*(1 + \mathrm{ab^*} + \mathrm{ba^*}) \\ &= 1 + \mathrm{ab^*} + \mathrm{ba^*} + (\mathrm{ab^*a} + \mathrm{ba^*b})(\mathrm{ab^*a} + \mathrm{ba^*b})^*(1 + \mathrm{ab^*} + \mathrm{ba^*}) \\ &= 1 + \mathrm{ab^*} + \mathrm{ba^*} + (\mathrm{ab^*a} + \mathrm{ba^*b})\, E \\ &= 1 + \mathrm{ab^*} + \mathrm{ba^*} + \mathrm{ab^*a}\ E + \mathrm{ba^*b}\, E \\ &= 1 + \mathrm{a}(\mathrm{b^*} + \mathrm{b^*a}\, E) + \mathrm{b}(\mathrm{a^*} + \mathrm{a^*b}\, E) \end{aligned}$$

The last equality shows that $D_a\, E = \mathrm{b^*} + \mathrm{b^*a}\, E$ and $D_b\, E = \mathrm{a^*} + \mathrm{a^*b}\, E$. This exactly is the way syntactic derivates are computed by the algorithm I am about to describe, in the following subsections. Note that both derivatives are computed at the same time. This should be contrasted with a computation of the derivatives based on the classical formulas given in [8]. Using them $D_a\, E$ and $D_b\, E$ are computed separately, implying redundant work. Even worse, derivatives are computed for *all* sub-expressions of E.

3.2 Syntactic derivatives

We define the notion of syntactic derivative in Figure 2. In this definition, x and y stand for letters and w for a chain of letters ($w \in \mathit{Letter}^*$). The expressions used in the left-hand sides of equalities (e.g. $E_1 + \ldots + E_n$) are supposed to fulfill the conditions imposed to them in Definition 2. A derivative of the form $D_x\, E$ is called a *direct* derivative of E. In the definition of $D_1\, E$, the symbol 1 denotes the empty chain of letters.

The definition uses a new binary operator, denoted by $\otimes$. This operator, which I call *right distributed concatenation*, simplifies the definition and it has interesting properties that simplify the correctness proof of the derivation algorithm of Section 3.3.

Definition 3. Right distributed concatenation

Let E be a normalized expression. As made precise before, it can be written as $E_1 + E_2 + \ldots + E_n$ ($n \geq 0$), where none of the expressions $E_1\,, E_2\,, , \ldots\,, E_n$ are unions. Let F be a normalized expression. By definition, $E \otimes F$ denotes the normalized expression

$$(E_1 \odot F) \ \oplus\ (E_2 \odot F) \ \oplus\ \ldots\ \oplus\ (E_n \odot F)$$

Let G, G_1, $\ldots$, G_m be other normalized expressions ($m \geq 0$). The operation $\otimes$ enjoys the following two properties

$$(E \otimes F) \otimes G \ = \ E \otimes (F \odot G)$$

$$(G_1 \oplus \ldots \oplus G_m) \otimes F \ = \ (G_1 \otimes F) \oplus \ldots \oplus (G_m \otimes F)$$

(End of definition)

It may seem counter-intuitive to use the operation $\otimes$ instead of $\odot$ in the definitions of Figure 2, but this is needed to ensure that the syntactic derivatives of a normalized expression are unions of partial derivatives of this expression as stated in Theorem 2, which, in turn, is needed to prove that normalized expressions have finitely many syntactic derivatives.

[8] Notice that these are not syntactic equalities between regular expressions, but equalities of the languages they denote.

Figure 2: Syntactic derivatives

$D_x\,0$	$=$	0	
$D_x\,1$	$=$	0	
$D_x\,y$	$=$	1	if $y = x$
	$=$	0	otherwise
$D_x\,(E_1 + \ldots + E_n)$	$=$	$D_x\,E_1 \oplus \ldots \oplus D_x\,E_n$	$(n \geq 2)$
$D_x\,(E_1\,.\,E_2)$	$=$	$D_x\,E_1 \otimes E_2$	if $1 \notin \mathcal{L}(E_1)$
	$=$	$D_x\,E_1 \otimes E_2 \oplus D_x\,E_2$	otherwise
$D_x\,E^*$	$=$	$D_x\,E \otimes E^*$	
$D_1\,E$	$=$	E	
$D_{w\,x}\,E$	$=$	$D_x\,(D_w\,E)$	

Theorem 1 *Correctness of syntactic derivatives Let E be a normalized regular expression and w a chain of letters. The following equality holds:*

$$\mathcal{L}(\,D_w\,E) = D_w\,(\mathcal{L}\,E)$$

Proof

Notice first that the symbol D_w is overloaded in the statement of Theorem 1. In the right part of the equality, it denotes the derivative in the "semantic" sense, i.e. $\{u \in \mathit{Letter}^* \mid w.u \in \mathcal{L}\,E\}$, while in the left part, it denotes the syntactic derivative, defined in Figure 2.

We first prove the result when w is a single letter x. The proof uses an induction on the structure of E and the facts that $\mathcal{L}(E_1 \oplus E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$ and $\mathcal{L}(E_1 \odot E_2) = \mathcal{L}(E_1 \otimes E_2) = \mathcal{L}(E_1).\mathcal{L}(E_2)$. The induction is easy using the definition of $\mathcal{L}E$ in Figure 1 and the definition of semantic derivatives.

The proof for arbitray chains w is an easy recurrence on the length of w, using the definitions of the derivatives and the result for a single letter x. (End of proof)

3.3 An algorithm to compute direct derivatives

Now I am about to present an efficient algorithm to compute the direct derivatives of a normalized expression E. The syntactic derivatives $D_x\,E$ are computed all at once, for all letters x. Moreover, they are computed little by little by unfolding the expression E until some part of a syntactic derivative is reached. These parts are precisely defined in Subsection 3.4 but we do not need this precise definition in this subsection. (We will need it later to prove that the set of all syntactic derivatives of E is finite.) Since the algorithm computes the derivatives little by little and simultaneously, we need a programming object to which the parts can be added at the right time. I call it an array of expressions. It is a mutable object of the same kind as arrays in most programming languages. It also is the mutable counterpart of a right part $o + \ldots + x.E_x + \ldots$ of an equation.

Definition 4. Arrays of expressions

An array of expressions, denoted by $tabD$, is a mutable function that maps 0 to 0 or 1, denoted by $tabD\,[0]$, and every letter x to a normalized expression, denoted by $tabD\,[x]$. The denoted values $tabD\,[0]$ and $tabD\,[x]$ can be modified by writing, for instance, $tabD\,[x] := E$, where E is a normalized expression. We also write $tabD\,[x] \;\oplus\!=\; E$, as a shorthand for $tabD\,[x] := tabD\,[x] \oplus E$. To express the same operation, we sometimes simply say that we *add* E to $tabD\,[x]$. An array of expressions must be *created* before it is used by an algorithm. All its elements are initialized to 0.

(End of definition)

The algorithm to compute the direct derivatives of an expression E returns an array of expressions $tabD$ such that $tabD\,[0] = o$, where $o = 1$ if $1 \in \mathcal{L}E$ and $o = 0$ otherwise, and $tabD[x] = D_x\,E$ for every letter x. Let us call it the top level algorithm.

The top level algorithm uses an auxiliary algorithm, which is recursive and more general. We denote a call to this auxiliary algorithm by a statement of the form

$$o := \mathit{deriveByUnfold}\,(tabD\,, E, F)$$

The effect of such a statement is twofold:

- It gives the value 1 to o if $1 \in \mathcal{L}\,E$ and the value 0, otherwise.
- It executes $tabD\,[x] \;\oplus\!=\; D_x\,E \;\otimes\; F$ for every letter x.

Since $0 \oplus (D_x\,E \;\otimes\; 1) \;=\; D_x\,E$, it is clear that, in order to fulfill its specification, the top level algorithm just has to create a new array $tabD$ and to execute:

$$tabD\,[0] := \mathit{deriveByUnfold}\,(tabD\,, E, 1)$$

The auxiliary algorithm is presented in Figure 3. Let us prove its correctness.[9]

Correctness proof of the algorithm *deriveByUnfold*

The proof proceeds by induction on the syntactic structure of E. Let us consider the different cases.

1. If E is equal to 0 or 1, $D_x\,E = 0 = 0 \otimes F = D_x\,E \otimes F$, for every letter x. The fact that the algorithm does not change any element of $tabD$ is thus correct. Since $1 \in \mathcal{L}(1)$ and $1 \notin \mathcal{L}(0)$, the returned value is correct, also.

2. If E is a letter x, $D_x\,E = 1$. Thus, $D_x\,E \otimes F = 1 \otimes F = F$. The fact that the algorithm add F to $tabD\,[x]$ without changing the values of $tabD\,[y]$ for $y \neq x$ is thus correct. Moreover, returning 0 is correct since $1 \notin \mathcal{L}(x)$.

3. if E is a union $E_1 + \ldots + E_n$, we can assume by induction hypothesis (i.e. the algorithm is correct for every E_i) that, for every letter x, the algorithm successively adds to $tabD[x]$ all the values $D_x\,E_i \otimes F$ $(1 \leq i \leq n)$. Thus, it performs the action $tabD[x] := tabD[x] \oplus (D_x\,E_1 \otimes F) \oplus \ldots \oplus (D_x\,E_n \otimes F)$, for every letter x. Since $(D_x\,E_1 \otimes F) \oplus \ldots \oplus (D_x\,E_n \otimes F) = (D_x\,E_1 \oplus \ldots \oplus D_x\,E_n) \otimes F \;=\; D_x\,E \otimes F$, the algorithm modifies the elements $tabD\,[x]$ as required. Moreover, the for loop gives the value 1 to o only if $1 \in \mathcal{L}\,E_i$, for at least one i, i.e. only if $1 \in \mathcal{L}\,E$. Thus, the returned value is correct.

[9] Nowadays, most people seem to believe that proving the correctness of an algorithm is just nonsense. In my view, however, the "proof" below is the key to understand how the algorithm works, based on the mathematical properties of the involved operations : $\oplus$, $\odot$, and $\otimes$.

Figure 3: The algorithm *deriveByUnfold*

1. If E is equal to 0 or 1, do
   return E;

2. If E is a letter x, do
   $tabD\,[x]\ \oplus\!= F$;
   return 0;

3. if E is a union $E_1 + \ldots + E_n$, do
   $o := 0$;
   for $i := 1$ to n, do
   $o \oplus\!= \mathit{deriveByUnfold}\,(tabD\,, E_i, F)$;
   return o;

4. if E is a concatenation $E_1.E_2$, do
   $o := \mathit{deriveByUnfold}\,(tabD\,, E_1, E_2 \odot F)$;
   if $o = 1$, do
   $o := \mathit{deriveByUnfold}\,(tabD\,, E_2, F)$;
   return o;

5. if E is an iteration E_1^*, do
   $o := \mathit{deriveByUnfold}\,(tabD\,, E_1, E \odot F)$;
   return 1;

4. if E is a concatenation $E_1.E_2$, we can assume that the first call to *deriveByUnfold* add to $tabD\,[x]$ the value $D_x\,E_1 \otimes (E_2 \odot F) = (D_x\,E_1 \otimes E_2) \otimes F$, for every letter x. This is correct if $1 \notin \mathcal{L}\,E_1$. Otherwise, the second call to *deriveByUnfold* also is executed, which adds to every $tabD[x]$ the value $D_x\,E_2 \otimes F$. So, all in all, the value $(D_x\,E1 \otimes E_2) \otimes F \ \oplus\ D_x\,E_2 \otimes F = (D_x\,E_1 \otimes E_2 \ \oplus D_x\,E_2) \otimes F = D_x\,E \otimes F$ is added to every $tabD[x]$ as required. To conclude on this case, one can see that the algorithm return 1 only if $1 \in \mathcal{L}\,E_1$ and $1 \in \mathcal{L}\,E_2$, i.e., only if $1 \in \mathcal{L}\,E$, and 0 otherwise, as required.

5. Finally, if E is an iteration E_1^*, the call to *deriveByUnfold* add the value $D_x\,E_1 \otimes (E \odot F) = (D_x\,E_1 \otimes E) \otimes F = D_x\,E \otimes F$ to every $tabD\,[x]$, as required. The algorithm also returns the value 1, which is correct since E is an iteration.

(End of correctness proof)

Now, let us discuss about the "programming style" and the efficiency of the algorithm presented in Figure 3. It uses a programming technique, first introduced in functional programming (see e.g. [9]), called *accumulators*. In functional programming, accumulators are "additional" parameters added to a function definition to improve its efficiency by reducing the use of recursion and/or replacing some operations by other, more efficient, ones. The idea is that the algorithm just has to traverse a data structure (for instance, a list) while collecting values "on the fly" in the accumulator. The algorithm *deriveByUnfold* makes a beautiful use of this idea. In fact, it uses *two* accumulators,

which play different roles: The first accumulator is *tabD* into which the "parts" of the syntactic derivatives successively are added; the second accumulator is F into which suffixes of these "parts" gradually are extended until a complete one is ready to be put into *tabD*. It is very illuminating to compare Figure 3 and Figure 2: In fact, the top part of Figure 2 can be seen (and can be used) as a functional program to compute $D_x\,E$. As already asserted in Subsection 3.1, such a program as two defects: First, the derivatives are recomputed separately for every letter x; second, and more importantly, the derivatives of all sub-expressions of E are computed. To the contrary, the algorithm *deriveByUnfold* does not compute these derivatives. As amazing as it may seem, their definition is *only used in the reasoning* of the correctness proof. Note also that the operation $\otimes$, used in Figure 2, is not used by the algorithm *deriveByUnfold* (but it is used by the correctness proof). To argue further on the efficiency of the algorithm *deriveByUnfold*, we should take into account the efficiency of the basic operations $\oplus$ and $\odot$, which will be done in Section 5.

3.4 Partial derivatives and their relation to syntactic derivatives

Now we turn to proving that the set of syntactic derivatives of a normalized expression E is finite. This is a necessary condition to be able to compute all these derivatives using the algorithm of Subsection 3.3, and, consequently, a set of equations of the form $F = tabD_{\,F}$ giving rise to a DFA for $\mathcal{L}\,E$. We proceed in three steps.

1. We define the (syntactic) notion of *partial derivatives*. Partial derivatives are a precise characterization of the "parts" of syntactic derivatives, computed by the algorithm of Subsection 3.3.

2. We prove that every syntactic derivative $D_w\,E$ ($w \in \mathit{Letter}^*$) of a normalized expression is of the form $E_1 + \ldots + E_n$ ($n \geq 0$), where $E_1, \ldots, E_n$ are partial derivatives of E.

3. We prove that the set of all partial derivatives of E is finite by showing that it is a subset of a finite set of normalized expression $\mathit{Lufs}\,(E)$. It happens that $\sharp\ \mathit{Lufs}\,(E) \leq \mathrm{size}(E) + 1$. Hence, the number of partial derivatives is quite small. In fact, it is less or equal to $(\mathrm{size}(E) + 1)/2$.

3.4.1 Partial derivatives

The notion of partial derivatives of an expression is defined in Figure 4. The set $\mathcal{D}_x\,E$ is first defined for a single letter x, then $\mathcal{D}_w\,E$ is defined by induction on the length of w ($w \in \mathit{Letter}^*$). Note that $\mathcal{D}_w\,E$ denotes a *set* of partial derivatives. Partial derivatives are normalized expressions, not sets. The definition uses an extension of the operation $\odot$ where the first argument is a set of normalized expression. By definition, one has: $\{E_1, \ldots, E_n\} \odot E = \{E_1 \odot E, \ldots, E_n \odot E\}$. Theorem 2 precisely relates syntactic derivatives to partial derivatives.

Theorem 2 *Let E and w be a normalized expression and a chain of letters. One has:*

$$D_w\,E = E_1 \oplus \ldots \oplus E_n \quad \textit{where} \quad \mathcal{D}_w\,E = \{E_1, \ldots, E_n\}\ \ (n \geq 0)$$

Moreover, none of the partial derivatives $E_1, \ldots, E_n$ are unions (in the sense of Definition 2).

Proof

Figure 4: (Sets of) Partial derivatives

$$
\begin{array}{rcll}
\mathcal{D}_x\, 0 & = & \{\,\} & \\
\mathcal{D}_x\, 1 & = & \{\,\} & \\
\mathcal{D}_x\, y & = & \{1\} & \text{if } \ y = x \\
 & = & \{\,\} & \text{otherwise} \\
\mathcal{D}_x\, (E_1 \,+\, \ldots \,+\, E_n) & = & \mathcal{D}_x\, E_1 \,\cup\, \ldots \,\cup\, \mathcal{D}_x\, E_n & (n \geq 2) \\
\mathcal{D}_x\, (E_1 \,.\, E_2) & = & \mathcal{D}_x\, E_1 \,\odot\, E_2 & \text{if } \ 1 \notin \mathcal{L}(E_1) \\
 & = & \mathcal{D}_x\, E_1 \,\odot\, E_2 \,\cup\, \mathcal{D}_x\, E_2 & \text{otherwise} \\
\mathcal{D}_x\, E^* & = & \mathcal{D}_x\, E \,\odot\, E^* & \\
\hline
\mathcal{D}_1\, E & = & \{E_1\,, \ldots\,, E_n\} & \text{where} \\
 & & & E = E_1 \,+\, \ldots \,+\, E_n \ (n \geq 0) \\
\mathcal{D}_{w\,x}\, E & = & \displaystyle\bigcup_{F \,\in\, \mathcal{D}_w\, E} \mathcal{D}_x\, F & \\
\end{array}
$$

We first prove the result when the chain w is a single letter x. The proof is rather straightforward by induction on the syntactic structure of normalized expressions. The only delicate point is to verify that $D_x\, E_1 \otimes E_2 = F_1 \oplus \ldots \oplus F_m$ where $\mathcal{D}_x\, E_1 \odot E_2 = \{F_1, \ldots, F_m\}$, for some normalized expressions $F_1, \ldots, F_m$ $(m \geq 0)$. Let $F'_1, \ldots, F'_m$ $(m \geq 0)$ be expressions such that $\mathcal{D}_x\, E_1 = \{F'_1, \ldots, F'_m\}$. Clearly, $\mathcal{D}_x\, E_1 \odot E_2 = \{F'_1 \odot E_2, \ldots, F'_m \odot E_2\}$. On the other hand, by induction hypothesis, we have that $D_x\, E_1 = F'_1 \oplus \ldots \oplus F'_m$. Therefore, $D_x\, E_1 \otimes E_2 = F'_1 \odot E_2 \oplus \ldots \oplus F'_m \odot E_2$, which proves the result (with $F_i = F'_i \odot E_2$ $(1 \leq i \leq m)$).

It is also necessary to prove the result for arbitrary chains w. Once again this is an easy induction of the length of the chains. (End of proof)

3.4.2 The set of all syntactic derivatives of a normalized expression is finite.

Since syntactic derivatives of an expression are unions of partial derivatives of the same expression, it is sufficient to prove that the set of all partial derivatives of such an expression is finite. To do so, we define the set of normalized expressions $\mathit{Lufs}\,(E)$, which is finite and contains all the partial derivatives of E. This set is presented in Figure 5. It is defined by induction on the syntactic structure of E.

Theorem 3 *The set $\mathit{Lufs}\,(E)$ is finite. More specifically, the number $\sharp\, \mathit{Lufs}\,(E)$ of its elements is less or equal to $\mathit{size}(E) + 1$.*

Proof

If $E = 0$, the result is obvious. Now, we assume that $E \neq 0$. It is sufficient to prove that $\sharp(\mathit{Lufs}\,(E) \setminus \{1\}) \leq \mathrm{size}(E)$. We prove it by induction on the structure of E. It is obvious if E is 1, or a letter. Otherwise, the result is easily proven by induction. For instance, if E is a union $E_1 + \ldots + E_n$,

Figure 5: Left unfolded suffixes

$$
\begin{array}{rcl}
\mathit{Lufs}\,(0) & = & \{\,\} \\
\mathit{Lufs}\,(1) & = & \{1\} \\
\mathit{Lufs}\,(x) & = & \{x,\,1\} \\
\mathit{Lufs}\,(E_1 + \ldots + E_n) & = & \{E_1 + \ldots + E_n\} \\
 & & \cup\ \mathit{Lufs}\,(E_1) \cup \ldots \cup \mathit{Lufs}\,(E_n) \quad (n \geq 2) \\
\mathit{Lufs}\,(E_1 \,.\, E_2) & = & \mathit{Lufs}\,(E_1) \odot E_2 \ \cup\ \mathit{Lufs}\,(E_2) \\
\mathit{Lufs}\,(E^*) & = & \mathit{Lufs}\,(E) \odot E^* \ \cup\ \{1\}
\end{array}
$$

$$
\begin{array}{rcll}
\sharp(\mathit{Lufs}\,(E) \setminus \{1\}) & \leq & 1 + \sharp(\mathit{Lufs}\,(E_1) \setminus \{1\}) + \ldots + \sharp(\mathit{Lufs}\,(E_n) \setminus \{1\}) & (n \geq 2) \\
 & \leq & 1 + \mathrm{size}(E_1) + \ldots + \mathrm{size}(E_n) & \\
 & = & 1 + \mathrm{size}(E_1 + \ldots + E_n) - (n-1) & (n-1 \geq 1) \\
 & \leq & \mathrm{size}(E_1 + \ldots + E_n) & \\
 & = & \mathrm{size}(E) &
\end{array}
$$

(End of proof)

To prove that the set of all partial derivatives of a normalized expression E is included in $\mathit{Lufs}\,(E)$, and thus finite, we need four lemmas.

Lemma 1 *Let E be a normalized regular expression different from 0. Then,*

$$1 \ \in\ \mathit{Lufs}\,(E)$$

Lemma 2 *Let E be a normalized regular expression different from 0. Then,*

$$E \ \in\ \mathit{Lufs}\,(E)$$

Lemma 3 *Let E_1 and E_2 be normalized regular expressions different from 0 and 1. Then,*

$$\mathit{Lufs}\,(E_1 \odot E_2) \ = \ \mathit{Lufs}\,(E_1) \odot E_2 \ \cup\ \mathit{Lufs}\,(E_2)$$

Lemma 4 *For any normalized expressions E and F, the following implication holds*

$$F \in \mathit{Lufs}\,(E) \quad \rightarrow \quad \mathit{Lufs}\,(F) \subseteq \mathit{Lufs}\,(E)$$

The proof of Lemma 4 is given below. The proofs of the first three lemmas are given in Appendix A.

Proof

The result is immediate if $F = E$. Now, we assume that $F \neq E$. The proof uses an induction on the syntactic structure of E.

1. If E is equal to 0, 1 or is a letter, the result is clear from the definition of $Lufs(E)$.

2. If E is a union $E_1 + \ldots + E_n$ ($n \geq 2$), it is true that $F \in Lufs(E_i)$ for at least one i ($1 \leq i \leq n$). By induction hypothesis, $Lufs(F) \subseteq Lufs(E_i) \subseteq Lufs(E)$.

3. If E is a concatenation $E_1.E_2$, the proof is similar as above if $F \in Lufs(E_2)$. Otherwise, we have that $F \in Lufs(E_1) \odot E_2$. Thus, there exists a normalized expression $F' \in Lufs(E_1)$ such that $F = F' \odot E_2$. By induction hypothesis, we can write that $Lufs(F') \subseteq Lufs(E_1)$. By Lemma 3, we thus have:

$$\begin{array}{rcl} Lufs(F) & = & Lufs(F' \odot E_2) \\ & = & Lufs(F') \odot E_2 \cup Lufs(E_2) \\ & \subseteq & Lufs(E_1) \odot E_2 \cup Lufs(E_2) \\ & = & Lufs(E) \end{array}$$

4. If E is an iteration, the proof is similar to the case $F \in Lufs(E_1) \odot E_2$ in point 3.

(End of proof)

Theorem 4 *Let E and w be a normalized expression and a chain of letters. It is the case that*

$$\mathcal{D}_w E \quad \subseteq \quad Lufs(E)$$

Proof

As usual, we first prove the result when w is a single letter x. The proof is an easy induction on the syntactic structure of E. No comments are really needed.

The proof for an arbitrary chain w requires an induction on the length of w.

1. If $w = 1$, one has $\mathcal{D}_w E = \{E_1, \ldots, E_n\}$ where $E = E_1 + \ldots + E_n$ and $n \geq 0$. If $n = 0$, the result is obvious since $\mathcal{D}_w E$ is empty. If $n = 1$, one has $\mathcal{D}_w E = \{E\}$ and the result is a consequence of Lemma 2. If $n \geq 2$, we have $\mathcal{D}_w E = \{E_1, \ldots, E_n\} \subseteq (Lufs(E_1) \cup \ldots \cup Lufs(E_n)) \subseteq Lufs(E)$, by induction on the structure of E.

2. If $w = w'x$, where w' is a chain and x is a letter, by definition

$$\mathcal{D}_{w'x} E \quad = \quad \bigcup_{F \in \mathcal{D}_{w'} E} \mathcal{D}_x F$$

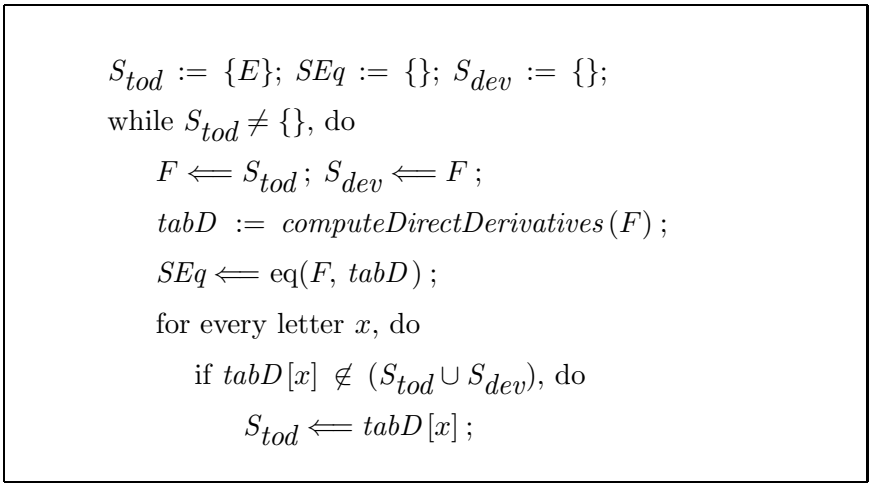
By induction hypothesis, $\mathcal{D}_{w'} E \subseteq Lufs(E)$. Thus, for every $F \in \mathcal{D}_{w'} E$, one has: $Lufs(F) \subseteq Lufs(E)$, by Lemma 4. Therefore,

$$\mathcal{D}_{w'x} E \quad = \quad \bigcup_{F \in \mathcal{D}_{w'} E} \mathcal{D}_x F \quad \subseteq \quad \bigcup_{F \in \mathcal{D}_{w'} E} Lufs(F) \quad \subseteq \quad \bigcup_{F \in \mathcal{D}_{w'} E} Lufs(E) \quad \subseteq \quad Lufs(E)$$

(End of proof)

Corollary 1 *Let us say that a partial derivative of E is* proper *if it belongs to a set $\mathcal{D}_w E$ where w is not empty. The set of all proper partial derivatives of a normalized expression E is finite. In fact, their number is less or equal to $(size(E) + 1)/2$.*

Figure 6: The algorithm that computes all syntactic derivatives (*computeDerivatives*).



Proof

The basic result is a consequence of Theorems 3 and 4. One can easily prove, by induction on E, that $Lufs(E)$ contains at most $(\mathrm{size}(E)+1)/2$ normalized expressions of the form $x \odot F$ where x is a letter. (F can be equal to 1.) An element of $Lufs(E)$ that is a proper partial derivative necessarily is such an F. Hence, the more precise result.

(End of proof)

Examples of derivatives, partial derivatives, and left unfolded suffixes are provided in Appendix D.

3.5 An algorithm to compute all syntactic derivatives

An algorithm that computes all syntactic derivatives of a normalized expression E is presented in Figure 6. It also builds the set of all equations $F = tabD_F$ where F is a syntactic derivative of E and $tabD_F$ is an array of expressions such that $tabD[0] = 1$ if $1 \in \mathcal{L}F$ and 0, otherwise, and $tabD[x] = D_x F$ for every letter x. This algorithm is called *computeDerivatives*. It makes use of the (top level) algorithm *computeDirectDerivatives*, described at Section 3.3.

To understand the algorithm *computeDerivatives*, some notations must be made precise. The symbols S_{tod} and S_{dev} denote two (mutable) sets of normalized expressions. S_{tod} is the set of normalized expressions still "to be developed", i.e. their direct derivatives are not computed, yet. S_{dev} is the set of normalized expressions already "developed", i.e. their direct derivatives have been computed and belong to $S_{tod} \cup S_{dev}$. The symbol SEq is the name of the set of equations already computed (their left part belongs to S_{dev}). An equation is represented by an "object" of the form $\mathrm{eq}(F, tabD)$. If S is a mutable set of "elements" and X, a variable for an element, the statement $X \Longleftarrow S$ removes an arbitrarily chosen element from S and it assigns it to X. Symmetrically, the statement $S \Longleftarrow X$ adds the element X to S. The correctness of the algorithm *computeDerivatives* is proven in Appendix B.

4 A background for reasoning about normalized expressions

Now I present a global data structure, called *the background*. All algorithms described previously, and later too, are working in the context of a (current) background. I first describe what the background contains in Section 4.1. Then, I explain operations that can be used to extend and refine the background in Section 4.2. In this section, the background is described at an abstract level. Relevant information about its practical implementation is given in Section 5 (see also [6]).

4.1 The background

Definition 5.
The background is a mutable object consisting of the following parts.

- A set of normalized expressions

  This set is downwards closed: If an expression belongs to the background, all its sub-expressions belong to the background as well.

- A partition of the set of expressions

  The partition is such that two expressions E_1, E_2 belonging to the same equivalence class denote the same regular language (i.e. $\mathcal{L}(E_1) = \mathcal{L}(E_2)$). (But the converse is not necessary true.) Moreover, there is a function that maps every expression E (belonging to the background) to an expression $\mathrm{rep}(E)$ belonging to the same equivalence class. The function rep is such that

  1. $\mathrm{size}(\mathrm{rep}(E)) \leq \mathrm{size}(E)$
  2. $\mathrm{rep}(E_1) = \mathrm{rep}(E_2)$ as soon as E_1, E_2 belong to the same equivalence class.

  The expression $\mathrm{rep}(E)$ is called *the representative of* E in its equivalence class.

- A set of equations relating some expressions to their derivatives

  This set contains equations of the form $E = tabD$ where E is the representative of an equivalence class of the background and $tabD$ is an array of expressions belonging to the background (see Definition 4, in Section 3.3). Such an equation can be written as $E = o + a.E_a + \ldots + x.E_x + \ldots$. The following conditions must hold.

  1. For every letter x, one has: $E_x = \mathrm{rep}(E_x)$.
  2. $\mathcal{L}(E) = \mathcal{L}(o) \cup a.\mathcal{L}(E_a) \cup \ldots \cup x.\mathcal{L}(E_x) \cup \ldots$

  (Of course, I write $x.\mathcal{L}(E_x)$ to mean $\{x\}.\mathcal{L}(E_x)$.) The latter condition above can be roughly re-expressed by saying that the E_x are derivatives of E with respect to x. However, they are not derivatives of regular languages (obviously) nor syntactic derivatives of E in the sense of Section 3, strictly speaking, in general. However, these are arguably the best possible equations, since they use the shortest expressions in both their left and right-hand sides.

  It is *not* required that an equation exists for every equivalence class. But it is often the case that the set of equations of the background is *complete*, i.e., all expressions E_x occurring in the right part of an equation occur as the left part of an equation. This ensures that every representative of an equivalence class determines a DFA for the regular language denoted by the expressions in the class.

(End of definition)

The definition of the background does not prevent that two different equations $E_1 = tabD_1$ and $E_2 = tabD_2$ are such that $E_1 = E_2$ or $tabD_1 = tabD_2$. We say that such equations *overlap*.[10] If it is the case, it means that at least two equivalence classes contains expressions that denote the same regular language. Thus, the background can be improved by merging those equivalence classes. If no such equations exist we say that the background is *reduced*. Notice however that a reduced background may nevertheless contain different equivalence classes whose expressions denote the same regular language.

4.2 Operations on the background

4.2.1 Adding new expressions

The set of expressions of the background can normally only be extended, with new expressions. These expressions are automatically added when one of the operations union, concat, iter is executed by an algorithm and does return an expression not belonging to the background beforehand. In that case, the new expression is put in a new equivalence class containing this expression only. The set of equations is not modified. If the returned expression already was belonging to the framework, no change takes place.

4.2.2 Merging two equivalence classes

Let E_1 and E_2 be the representatives of two different equivalence classes. Executing the operation merge(E_1, E_2) is valid only if $\mathcal{L}(E_1) = \mathcal{L}(E_2)$ and size$(E_1) \leq$ size(E_2). It changes the background as follows.

- The two equivalence classes are merged into a single one. Other classes are left unchanged.
- The expression E_1 becomes the representative of the new equivalence class. The representatives of other classes are not changed.
- The expression E_2 is replaced by E_1 in every equation $E = tabD$ present in the background. Thus, E is replaced by E_1 if $E = E_2$, initially. Also, for every letter x, $tabD[x]$ becomes equal to E_1 if $tabD[x] = E_2$, initially.

It can be checked that a valid application of the operation merge maintains all properties required for the background, in Definition 5. However, it may happen that the modified background contains equations that overlap, even if it was previously reduced.
The efficiency of the operation merge can also be questioned. In fact, the background can be implemented in such a way that the complexity of the operation is $O(nl \times \text{nocc}(E_2))$ where nl is the number of letters and nocc(E_2) is the number of occurrences of E_2 in all equations. This is quite efficient if nl is small.

4.2.3 Reducing the background

It is clear that a reduced background provides better information than a non reduced one. Thus, we need an operation reduce(S), where S is a set of pairs of expressions corresponding to pairs of equivalence classes to be merged. For a simple reduction of the background, S should be empty.

[10] They must be distinct.

Figure 7: Reducing the background (reduce(S))

```
while S ≠ {} or at least two equations overlap, do
    while S ≠ {}, do
        ⟨F1, F2⟩ ⟸ S;
        ⟨E1, E2⟩ := trim(F1, F2);
        if E1 ≠ E2, do
            merge(E1, E2);
    if at least two equations overlap, do
        let Ei = tabDi (i = 1, 2) be such equations;
        if E1 ≠ E2, do
            S ⟸ ⟨E1, E2⟩;
        elsefor every letter x, do
            if tabD1[x] ≠ tabD2[x], do
                S ⟸ ⟨tabD1[x], tabD2[x]⟩;
```

An algorithm for reduce(S) is depicted in Figure 7. The algorithm uses an auxiliary operation trim(F_1, F_2), where F_1 and F_2 are normalized expressions. It returns a pair of expressions $\langle E_1, E_2\rangle$ such that $\{E_1, E_2\} = \{\text{rep}(F_1), \text{rep}(F_2)\}$ and $\text{size}(E_1) \leq \text{size}(E_2)$.

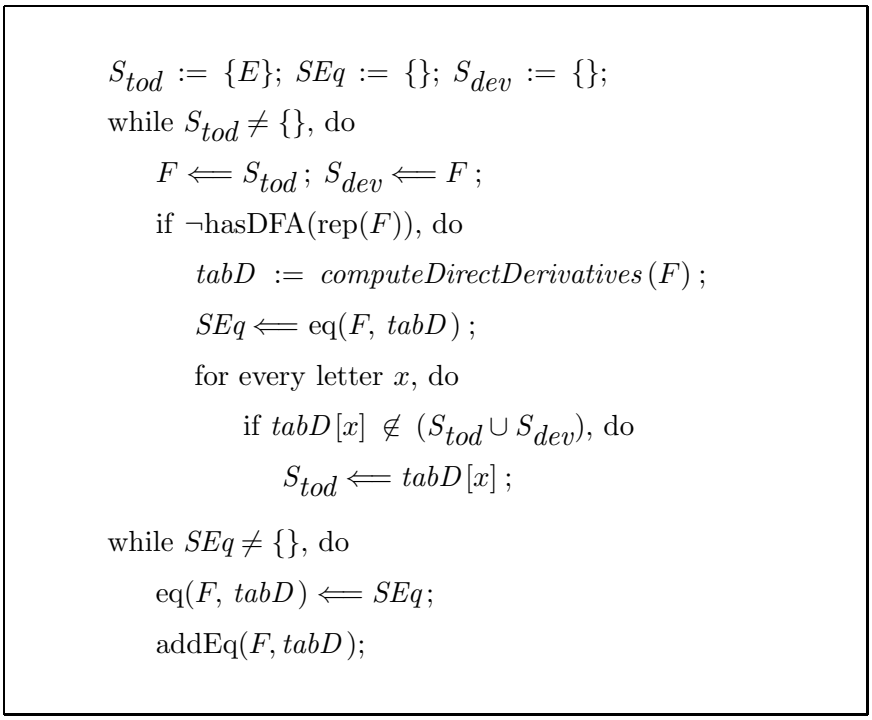
The algorithm reduce(S) maintains all the properties required for the background since all pairs $\langle E_1, E_2\rangle$ put in S are such that $\mathcal{L}(E_1) = \mathcal{L}(E_2)$. The set of expressions of the background is left unchanged but equivalence classes can be merged and the number of equations can decrease. Only termination is not completely obvious. A termination proof is given in Appendix C.
The implementation of the backgound is such that it is possible to decide in constant time whether two overlapping equations exist, and to choose two of them, as well. A complete description of the implementation of the background is out of the scope of this paper but it is similar to the data structure presented in [7].

4.2.4 Merging equivalence classes further

As said before, it may happen that two different equivalence classes (of expressions in the background) denote the same regular language, *even if* the background is reduced. It is possible to detect this fact when these classes are associated to equations belonging to the same complete set of equations.

Let *SEq* be a complete subset of the equations in the background (possibly all of them). The set *SEq* can be seen as a DFA to which Moore's algorithm [10] can be applied in order to build the equivalence classes of the states that accept the same language (see also [1], Algorithm 2.2, page 126). Then, we can apply the algorithm merge to all pairs $\langle E_1, E_2\rangle$ of distinct expressions belonging to the same equivalence class. Afterwards, we can execute reduce({}) to obtain a reduced

Figure 8: Adding equations to the background to get a DFA for E.



background in which all equivalence classes corresponding to equations $E_i = tabD_i$ $(i = 1, 2)$ in SEq have been merged whenever $\mathcal{L}(E_1) = \mathcal{L}(E_2)$.[11]

A version of such an algorithm is available as an algorithm of the background. It can be called either as minimize(), which considers all equations in the background, or as minimize(S), where S is a set of expressions that are representatives of their classes. In the latter case, the algorithm applies to the set of equations that are reachable from S. In either case, the set of equations to consider must be complete and the complexity of the operation is $O(nl \times neq^2)$ where nl is the number of letters used by expressions and neq is the number of equations.

4.3 Improving the computation of derivatives with the background

Although the algorithms described in Section 3 remain entirely correct in the presence of the background, part of the work they are supposed to do can be avoided sometimes because equations already exist for some derivatives or their representatives. In some cases, there is nothing to do at all: It is the case when the representative of an expression to derive is the left part of an equation in the background, and when this equation belongs to complete set of equations.

From now on, we assume that the background provides an operation hasDFA(E) that returns *true* only if the background contains an equation of the form $E = tabD$ belonging to a complete set of equations. In many case, this set of equations is more compact that the set of equations that would be computed by the algorithm *computeDerivatives* of Section 3 because the background is reduced.

[11] Alternatively, we can apply reduce($\{\langle E_1, E_2\rangle\}$) for all such pairs.

Note however that the algorithm *computeDerivatives* is the primary provider of equations to the background. As can be shown in Figure 6, it computes a complete set of equations *SEq*. However, it is not necessarily the case that all expressions occurring in these equations are representatives of their equivalence class, as required by the definition of the background. Thus, the equations must be modified to contain representatives only, before being added to the background. This is achieved by the operation addEq($E, tabD$), which adds to the background the equation $E' = tabD'$, where $E' = \text{rep}(E)$ and $tabD'[x] = \text{rep}(tabD[x])$ for every letter x. (Also, $tabD'[0] = tabD[0]$.) After execution of the operation addEq($E, tabD$), any call hasDFA(rep(E)) returns *true*.[12]

A modified (optimized) version of the algorithm *computeDerivatives* is presented in Figure 8. It only computes equations that are really needed to get a DFA for E. I do not give a detailed correctness proof for the modified algorithm but I explain the main ideas of such a proof.

- The first while loop of the algorithm puts in *SEq* a set of objects eq($F, tabD$) such that $tabD[x] = D_x F$ and $tabD[x] \in S_{dev}$, for every letter x. Moreover, the set of possible values for F is $\{F \mid F \in S_{dev} \ \& \ \neg\text{hasDFA}(\text{rep}(F))\}$.
- The second while loop adds to the background a set of equations that fulfills the conditions imposed to equations in Definition 5.
- After execution of the second while loop, all equations added to the background are part of a complete set of equations because all expressions in the right part of the equations newly added to the backgroud, either are equal to the left part of one of these new equations, or were already the left part of an equation of the backgound before the algorithm was started.
- When the algorithm terminates, it contains an equation of which the expression rep(E) is the left part, either because such an equation already was present beforehand or because the algorithm created it.

As a complementary note, remark that some equations of the background may overlap after execution of the improved algorithm *computeDerivatives*(E). This is because several distinct syntactic derivatives computed by the first while loop may have the same representatives in the background. Therefore, it is worthwhile to add a call reduce({}), as a final statement to the algorithm. Alternatively, a call reduce({}) can be added as a final statement in the operation addEq($E, tabD$).

5 Experimental evaluation

In this section, an experimental evaluation of the algorithms described in Sections 3 and 4 is proposed. The objective is to give a precise idea of their practical usefulness. To make the discussion more interesting and convincing, a comparison with some variants of the algorithms is given, as well as a comparison with an implementation of Brzozowski's method and variants of this method similar to the variants of my own method.

5.1 Important aspects of the implementation of the background

The results soon presented are meaningful mostly in the context of a specific implementation of the background data structure presented in Section 4. Therefore, information about this implementation is provided now but it is limited to some aspects essential for understanding the results. More

[12] This is correct because no such call is executed before the second while loop is completed.

Table 1: Statistics on computing derivatives and partial derivatives

ℓ	$\sharp CD$	$\sharp CDA$	$\sharp Red$	$\sharp Min$	t_{deriv}	t_{red}	t_{min}	$\sharp CP$	$\sharp CPA$	$\sharp Red$	t_{deriv}	t_{red}
10	1.76	8.94	3.82	3.82	$66.2\,\mu$	$8.42\,\mu$	$17.3\,\mu$	3.39	8.19	1.12	$90.2\,\mu$	$3.92\,\mu$
20	3.60	32.9	5.09	5.03	$96.1\,\mu$	$17\,\mu$	$23.6\,\mu$	5.16	21.3	2.45	$118\,\mu$	$8.34\,\mu$
40	8.34	135	8.35	7.88	$252\,\mu$	$41.3\,\mu$	$39.7\,\mu$	8.98	53	5.73	$153\,\mu$	$10.2\,\mu$
80	17.2	646	13.2	10.3	$283\,\mu$	$66.3\,\mu$	$76\,\mu$	15.5	146	11.2	$200\,\mu$	$16.3\,\mu$
160	45.2	$3.97\,K$	30.8	18.6	$675\,\mu$	$132\,\mu$	$133\,\mu$	29.8	355	23.6	$341\,\mu$	$53.5\,\mu$
320	160	$34.7\,K$	105	41	$4.12\,m$	$359\,\mu$	$488\,\mu$	57.3	922	48.4	$543\,\mu$	$44.3\,\mu$
640	592	$361\,K$	344	36	$24.3\,m$	$1.32\,m$	$1.06\,m$	112	$2.16\,K$	96	$1.05\,m$	$96.6\,\mu$
1280	$5.85\,K$	$8.04\,M$	$3.51\,K$	37.4	$521\,m$	$11.8\,m$	$9.33\,m$	224	$4.70\,K$	196	$1.35\,m$	$76.4\,\mu$
2560	$53.1\,K$	$166\,M$	$34\,K$	28.2	$14.4\,s$	$109\,m$	$62.1\,m$	443	$10.1\,K$	388	$3.66\,m$	$129\,\mu$

information is given in [6]. The main idea of the implementation is to attach a unique identifier to every normalized expression represented in the backgound. The number of identifiers is chosen and fixed at each start of the system. The identifiers are attributed dynamically. Thus, expressions are given different identifiers at different runs of the system. Moreover, expressions are normalized differently in different runs because sub-expressions of unions are sorted on their identifiers (which is as efficient as possible). Since the number of identifiers is limited, a kind of specialized garbage collector has been implemented to reuse identifiers no longer needed for the current task at hand. The system is implemented in java (version `1.8.0_131`, on a MacBook Pro (Retina, 13-inch, Early 2015)) but the implementation only uses integers (mainly `int`), arrays of integers, and arrays of arrays of integers, so any java version can be used. The main benefit of using integers to identify expressions obviously is that syntactic equality of normalized expressions is checked in $O(1)$, and, in fact, as efficiently as possible. The complexity of the operation $E_1 \oplus E_2$ is $O(n_1 + n_2)$ where $E_i = E_{i_1} + \ldots E_{i_{n_i}}$ $(n_i \geq 0)$ $(i = 1, 2)$. This is quite small, in general, and much smaller than the size of the resulting expression. A similar complexity result holds for the operation $\odot$. Nevertheless, in some situations, a repeated use of the operation $\oplus$ can lead to a quadratic complexity where a $n \log n$ complexity could be possible. This issue is taken into account later on, when the experimental efficiency of the algorithms is discussed.

5.2 Experimental efficiency of the algorithm *computeDerivatives*

Now measures of the experimental efficiency of Algorithm *computeDerivatives* are presented and discussed (see Figures 3 and 6 in Section 3, and Figure 8 in Section 4). Afterwards, some variants of the "fundamental" algorithm are motivated and evaluated as well, with a comparison to the original one. Nine sets of regular expressions have been generated "randomly" by an algorithm giving the same probability of choice to every expression (in a given "universe" of possible expressions). The expressions use two letters only, and have various fixed lengths from 10 to 2560. Each set contains 100 expressions of equal lengths. Each run of the program works as follows: An empty background is created with a range of $5,000,000$ identifiers. Then, every expression is read in turn and it is normalized. Afterwards, its derivatives are computed by the algorithm of Figure 8. Of course, the equations related to the derivatives are computed as well. The set of equations is reduced by reducing the background (see Figure 7). Reduction takes place every time an equation is added to the background. Finally, a minimization algorithm is applied to the reduced set of equations.

Statistics about the fundamental algorithm are given in the eight first columns of Figure 1. The

first column ℓ gives the size of the expressions of the data set. The second column $\sharp CD$ is the number of calls to the (top level) algorithm *computeDirectDerivatives*, on the average; it is also the number of equations added to the background (before reduction). The third column ($\sharp CDA$) gives the number of calls to the recursive algorithm of Figure 3. The columns $\sharp Red$ and $\sharp Min$ are the average numbers of equations for an expression after reduction and minimization, respectively. The columns t_{deriv}, t_{red}, and t_{min} present the average execution times for the computation of the derivatives, the reduction of equations, and the minimization of equations, respectively. (Times are given in seconds (s), milliseconds(m), or microseconds (μ)). The reduction algorithm is applied during the computation of the derivatives as well as during the minimization of equations. Thus t_{red} is part of both t_{deriv} and t_{min}. The following observations can be done: the execution times and the number of derivatives remain quite small until $\ell = 640$. It is only from $\ell = 1280$ that the exponential character of the algorithm becomes clear. One can also see that t_{red}, and t_{min} are negligible with respect to t_{deriv}. For small values of ℓ, minimal or almost minimal sets of equations are obtained without using minimization. The fact that $\sharp CD < \sharp Red$ until $\ell = 40$ is because the derivation algorithm detects quite often that the current expression already has a set of equations, so no call to *computeDirectDerivatives* is executed (see Figure 8). It can also be noted that reduction of the set of equations reduces this set by about 40%, which reduces the minimization time.

5.3 An algorithm that computes partial derivatives

The algorithm to compute derivatives can easily be modified into an algorithm to compute partial derivatives only (look at Figure 6). First, if the expression to be considered is a union $E_1 + \ldots + E_n$ $n \geq 2$, we initialize S_{tod} with

$$S_{tod} := \{E_1, \ldots, E_n\};$$

Second, after computing a new array of derivatives (for a partial derivative in S_{tod}), we decompose its components $tabD[x]$ that are unions into their sub-expressions, which are then put into S_{tod} if they are not already in $(S_{tod} \cup S_{dev})$. This algorithm computes an equation for every partial derivative. This set of equations can be viewed as a non deterministic finite automaton, similar to the automaton proposed in [3]. However, it can be smaller because the expression first is normalized and because reduction is applied to the set of equations. The last columns of Table 1 provides statistics for this algorithm. The column $\sharp CP$ is the number of calls to the derivation algorithm, on the average, while $\sharp CPA$ is the number of calls to the auxiliary recursive algorithm. The columns $\sharp Red$ is the average number of equations after reduction. The last two columns t_{deriv} and t_{red} are the average derivation and reduction times, respectively. There is no minimization time here since minimization applies to deterministic automata only.[13] The following comments can be made. The average number of calls $\sharp CP$ grows strictly linearly. (We can write $\sharp CP \approx 0.175\,\ell$.) It is almost the same for the derivation time. (This is much better than the complexity of the algorithm proposed in [3].) A detailed analysis of the number of equations reveals that they are always close to the average $\sharp CP$. Comparing this algorithm with the algorithm computing (full) derivatives, we see that both the execution times t_{deriv} and the number of calls $\sharp CP$ of the former are negligible with respect to those of the later, when ℓ grows. However, it is not the case for small values of ℓ because the modified algorithm can make more work at every iteration (i.e. decomposing unions into sub-expressions). We also see that the number of equations is reduced by about 15%.

[13] Attempting to design a minimization algorithm for non deterministic automata would be of little use here.

Table 2: Optimizing the computation of derivatives

ℓ	$\sharp CD$	$\sharp CDA$	t_d	ex	gc	$\sharp CD_0$	$\sharp CDA_0$	$\sharp D_0$	t_{d_0}	ex_0	gc_0	t_{d_1}	ex_1	gc_1
10	1.76	8.94	$66.2\,\mu$	0	0	1.15	4.55	1.75	$69\,\mu$	0	0	$102\,\mu$	0	0
20	3.60	32.9	$96.1\,\mu$	0	0	2.74	17.3	3.67	$135\,\mu$	0	0	$221\,\mu$	0	0
40	8.34	135	$252\,\mu$	0	0	6.15	48.1	8.61	$200\,\mu$	0	0	$231\,\mu$	0	0
80	17.2	646	$283\,\mu$	0	0	12.5	142	18.2	$283\,\mu$	0	0	$345\,\mu$	0	0
160	45.2	$3.97\,K$	$675\,\mu$	0	0	26.6	350	47.2	$649\,\mu$	0	0	$664\,\mu$	0	0
320	160	$34.7\,K$	$4.12\,m$	0	0	54.4	916	165	$1.79\,m$	0	0	$1.59\,m$	0	0
640	592	$361\,K$	$24.3\,m$	0	0	109	$2.16\,K$	595	$11.7\,m$	0	0	$8.53\,m$	0	0
1280	$5.85\,K$	$8.04\,M$	$521\,m$	0	0	220	$4.69\,K$	$5.84\,K$	$222\,m$	0	0	$122\,m$	0	0
2560	$53.1\,K$	$166\,M$	$14.4\,s$	20	17	441	$10.1\,K$	$71.4\,K$	$10.2\,s$	12	10	$4.87\,s$	4	2

5.4 Two optimizations of the fundamental algorithm

The discussion above suggests two ways of improving the computation of derivatives. For partial derivatives, there is basically no need for additional improvements but the figures in Table 1 suggest that most (i.e. not only a few) derivatives are strict unions of partial derivatives. Having a look at Figure 3, we understand that the third case of the algorithm is executed very often with $F = 1$, so that the derivatives of partial derivatives are recomputed very often. Thus, it can be worthwhile to memoize the calls to the algorithm for partial derivatives. Another remark is that the operation $tabD\,[x] \oplus\!= F$ has to be executed for all sub-expressions F of all derivatives of the expression at hand. This entails a quadratic complexity on the number of expressions F to be added to $tabD\,[x]$. We can do better by unfolding the top level call to the algorithm *deriveByUnfold*: for computing the direct derivatives of a derivative $E_1 + \ldots + E_n$ ($n \geq 2$), where the E_i are (memoized) partial derivatives, we simply merge the sub-expressions of every array $tabD_i[x]$ to obtain the new derivatives $tabD\,[x]$. This can be done in $O((m_1 + \ldots + m_n) \log n)$ instead of $O((m_1 + \ldots + m_n)^2)$ (where we assume that $tabD_i[x]$ has m_i sub-expressions). Using this method, we can also wait to create only one expression when the complete merging of the derivatives is completed. This entails more economical use of new identifiers and less use of the garbage collector. Finally, instead of simply memoizing the derivation of partial derivatives, we can first compute all partial derivatives at once, using the algorithm previously described. Then, the computation of derivatives can be done by only merging lists of identifiers, without any new call to the algorithm *computeDirectDerivatives*. Statistics on these optimizations are presented in Table 2. The columns $\sharp CD$, $\sharp CDA$ and t_d are repetitions of the columns in Table 2 (t_d stands for t_{derive}). A time-out of 36 seconds has been imposed to all algorithms (for each single expression). The column *ex* indicates how many times the time-out was exceeded for the fundamental algorithm *computeDirectDerivatives*. The column *gc* gives the number of garbage collector calls. The next six columns reports on the algorithm that memoizes the calls to *deriveByUnfold*, with no further optimization. The meaning of the columns is easily deduced by looking at the previous columns, except $\sharp D_0$ which gives the number of equations created by the optimized algorithm (without reduction). The last three columns reports on the algorithm that first computes all partial derivatives and computes the (full) derivatives afterwards by merging arrays of partial derivatives. We observe that the last (most optimized) version of the algorithm is approximatively four times faster than the fundamental algorithm for large expressions ($\ell \geq 1280$) and two times faster than the memoizing algorithm. The timings are similar for $\ell = 160$ but for small expressions the fundamental algorithm is faster. For "very large" expressions ($\ell = 2560$) no

algorithm is able to complete its work for all expressions but the most optimized one does a much better job than the two others and, it fact, it is able to succeed on all examples if we relax the time-out to 100 seconds. It must also be noticed that the garbage collectour is called only two times because much less normalized expressions are created as explained above.

5.5 Implementation of Brzozowski's algorithm, using the background

A reader could be asking whether the algorithms presented in this paper constitute a significant improvement to Brzozowki's algorithm presented in [5]. As a piece of the answer, I report here on an implementation of Brzozowki's algorithm with normalized expressions and the background. It is not necessary to use the "full power" of the background to realize such an implementation and a simpler use of normalized expressions could be more efficient but, for simplicity, exactly the same version of the background is used below.

But, what does it mean exactly to implement Brzozowski's method with normalized expressions. It is very simple: we only have to replace each use of the operation $\otimes$ by the use of $\odot$ in Figure 2 where my own notion of syntactic derivative is defined. As a matter of fact, this simple change makes the definition coincide with the definition of Brzozowski and with the rules given by Conway in [8], except for the fact that normalized expressions are used. Then, we can adapt the algorithms of Figures 3, 6, and 8 to get the desired implementation. The only delicate point is the modification of the algorithm *deriveByUnfold* (see Figure 3). It is no longer possible to use the accumulator F. The correctness proof would fail in the case of a union. It is necessary to use a more recursive algorithm. Also the use of a single accumulator *tabD* in all recursive calls must be replaced by the creation of new arrays of expressions in the case of a concatenation and of an iteration. The modified algorithm is thus less efficient. Another issue to solve is the termination problem: Are there finitely many syntactic derivatives? We could simply argue that it is so because normalized expressions embodies all properties of expressions used by Brzozowski in its method to ensure termination. But a better justification is to look at Conway's proof that an event has finitely many derivatives (see [8], Theorem 3, page 43). Although normalized expressions are not events, the proof can be reformulated by replacing events with normalized expressions. The only delicate point is that the reasoning with infinite unions of events must be replaced by something more technical. As a corollary, we get that the number of syntactic derivatives obtained by this method is less or equal to $2^{\mathrm{size}(E)+1}$, which is much better than the bounds given by Brzozowski. It is my belief that this way of implementing Brzozowski's method is as efficient as possible except for the quadratic behaviour entailed by the use of the operations $\oplus$ and $\odot$ in some situations.

Statistics about this implementation of Brzozowski method are given in Tables 3 and 4. The tables have the same structure than Tables 1 and 2 and should be understood by refeering to those two. Some explanations are necessary however. The basic algorithm, whose statistics are given in the first columns of Table 3 is obtained simply and solely by modifying the algorithm *deriveByUnfold*, as explained in the previous paragraph. To complete the comparison, we need a notion of "partial derivative" corresponding to Brzozowski's method. Remembering that derivatives are union of partial derivatives in our framework, we can *define* these new partial derivatives as the direct sub-expressions of Brzozowski's derivatives that are unions. Then all algorithms presented in Subsections 5.3 and 5.4 can be modified straightforwardly just by replacing *deriveByUnfold* by its modified version. Note that this is a bit artificial since it is not possible to give a definition of these partial derivatives similar to the definition of Figure 4 in order to directly prove that there are a finite number of them. To the contrary, we must rely on the fact that (full) derivatives are finite in number to prove that the partial primitives are. Comparing the two sets of tables shows that

Table 3: Statistics on computing derivatives à la Brzozowski

ℓ	$\sharp CD$	$\sharp CDA$	$\sharp D$	$\sharp Red$	t_{deriv}	t_{red}	$\sharp CP$	$\sharp CPA$	$\sharp Red$	t_{deriv}	t_{red}
10	1.76	9.14	2.06	3.82	$102\,\mu$	$42.8\,\mu$	3.41	8.98	1.08	$92.4\,\mu$	$5.11\,\mu$
20	3.84	41.8	3.84	5.12	$119\,\mu$	$23.3\,\mu$	5.31	26.3	2.60	$114\,\mu$	$8.31\,\mu$
40	10	249	10	9.19	$316\,\mu$	$89.1\,\mu$	9.90	91.8	6.27	$250\,\mu$	$16.4\,\mu$
80	35.2	$4.74\,K$	35.2	20.4	$1.30\,m$	$218\,\mu$	24.7	$1.18\,K$	17.2	$566\,\mu$	$81.6\,\mu$
160	$4.17\,K$	$7.86\,M$	$4.17\,K$	$2.24\,K$	$419\,m$	$8.59\,m$	263	$124\,K$	172	$10.7\,m$	$266\,\mu$
320	$38.1\,K$	$205\,M$	$9.05\,K$	$29.9\,K$	$8.48\,s$	$32.3\,m$	$30.1\,K$	$116\,M$	$25.6\,K$	$5.20\,s$	$11.1\,m$

Table 4: Statistics on computing derivatives à la Brzozowski (continued)

ℓ	$\sharp CD$	$\sharp CDA$	t_d	ex	gc	$\sharp CD_0$	$\sharp CDA_0$	$\sharp D_0$	t_{d_0}	ex_0	gc_0	t_{d_1}	ex_1	gc_1
10	1.76	9.14	$102\,\mu$	0	0	1.15	5.11	2.06	$70.2\,\mu$	0	0	$101\,\mu$	0	0
20	3.84	41.8	$119\,\mu$	0	0	2.90	22.1	3.88	$164\,\mu$	0	0	$177\,\mu$	0	0
40	10	249	$316\,\mu$	0	0	7.28	86.9	10.2	$198\,\mu$	0	0	$235\,\mu$	0	0
80	35.2	$4.74\,K$	$1.30\,m$	0	0	22	$1.18\,K$	36.2	$580\,\mu$	0	0	$733\,\mu$	0	0
160	$4.17\,K$	$7.86\,M$	$419\,m$	0	0	260	$124\,K$	$3.17\,K$	$28.4\,m$	0	0	$23.7\,m$	0	0
320	$38.1\,K$	$205\,M$	$8.48\,s$	20	1	$19\,K$	$92.8\,M$	$19.7\,K$	$5.10\,s$	12	6	$5.46\,s$	13	5

our method is much more efficient than Brzozowski's method. (That is: More efficient than our implementation of Brzozowski's method.) We can at least make the following observations. First, none of the algorithms analyzed in Tables 3 and 4 are able to deal with expressions of size greater than 320. Second, the number of "partial derivatives" grows very quickly (exponentially), while it grows linearly in Table 4. We see that the timings for this algorithm are not significantly better than the timings for computing full derivatives (except for $\ell = 160$). Therefore the timings for the "optimized" versions in Table 4 are not much better either.

In conclusion, I would say that the method of computing DFA based on syntactic derivatives, described in this paper, is significantly better than Brzozowski's method. It seems that the whole secret is just to replace . by $\otimes$ instead of $\odot$ in the derivation rules proposed by Brzozowski and Conway.

5.6 Other experiments

In this Section 5 "randomly generated" expressions have been used to get general information on the behaviour of the proposed algorithms. Other examples of expressions and classes of expressions are experimentally studied in the appendices. Appendix E shows on an example from [8] why Brzozowski's algorithm is less effective than the algorithm based on partial derivatives. Appendix F shows another example from [8] were minimization is unavoidable to get the best result. Appendices G and H shows how the algorithms behave on two classes of expressions from [4] and [12] for which the DFA size is exponential on the size of the expressions.

6 Conclusion

The contribution of this paper can be summarized as follows: 1) The notion of derivative of a regular expression, introduced in [5], is clarified and a purely syntactic definition of derivative is proposed. 2) Algorithms to compute derivatives of regular expressions are proposed and evaluated experimentally.

As for the first point, the notion of normalized regular expression is first intoduced, as well as operations working on normalized expressions. Operations on normalized expressions are uniquely defined rather than modulo an equivalence relation. This makes it possible to define derivation as a *function* mapping normalized expressions to normalized expressions. This is not the case in [5] nor in [8]: In both proposals, "expertise" is needed to compute the set of all derivatives of an expression, by detecting the equivalence of differently calculated derivatives. A strictly syntactic notion of partial derivative is proposed in [3] but the notion is not "extended" to a syntactic notion of (full) derivative: In a footnote, at page 292, this author writes that derivatives of an expression do not enjoy the property of being finite in number. Thus, he prefers to stick to Brzozowski's derivative notion, without attempting to replace it with a new single syntactic notion. Instead, he works with sets of partial derivatives. My proposal, in this paper, is more direct, allowing me to construct a straight algorithm computing the syntactic derivatives of a normalized expression. Looking back at Figure 2, the reader can observe that the definition of syntactic derivative does not require a notion of partial derivative beforehand. However, this notion is useful to prove that the set of syntactic derivatives of a normalized expression is finite. In my opinion, my proof of this fact, based on "left unfolded suffixes" is more natural than the presentation in [3] as it relates partial derivatives to the way derivation can be done by hand (with "expertise").

Concerning the second point, I introduce a programming framework, called "the background", into which the algorithms for computing derivatives can be efficiently implemented. A complete description of its implementation is out of the scope of this paper, but the aspects useful to understand why the algorithms can be efficiently implemented in this framework are made explicit. (The reader may also have a look at [7, 6].) Experiments with the algorithms on sets of expressions of various sizes, from very short to relatively large ones, give a good idea of the practical applicability of using derivatives of regular expressions to build DFAs, since the implementation proves effective. Variants of the fundamental algorithms are also proposed and evaluated. In particular, an algorithm to compute partial derivatives is shown very efficient. It can be viewed as an optimal implementation of the algorithm proposed in [3]. Another variant arguably is an implementation of Brzozowski's method, benefiting from the use of normalized expressions. It turns out to be much less applicable in general than the proposal in this paper.

References

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling: Compiling*. Prentice-Hall series in automatic computation. Prentice-Hall, 1972.

[2] Valentin M. Antimirov. Rewriting regular inequalities (extended abstract). In Horst Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium, FCT '95, Dresden, Germany, August 22-25, 1995, Proceedings*, volume 965 of *Lecture Notes in Computer Science*, pages 116–125. Springer, 1995.

[3] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[4] Valentin M. Antimirov and Peter D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143:195–209, 1994.

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

[6] Baudouin Le Charlier. A program that simplifies regular expressions (tool paper). *CoRR*, abs/2307.06436, 2023.

[7] Baudouin Le Charlier and Mêton Mêton Atindehou. A data structure to handle large sets of equal terms. In James H. Davenport and Fadoua Ghourabi, editors, *7th International Symposium on Symbolic Computation in Software Science, SCSS 2016, Tokyo, Japan, March 28-31, 2016*, volume 39 of *EPiC Series in Computing*, pages 81–94. EasyChair, 2016.

[8] J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall mathematics series. Dover Publications, Incorporated, 2012.

[9] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[10] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[11] Alley Stoughton. *Formal Language Theory: Integrating Experimentation and Proof*. https://alleystoughton.us/forlan/book.pdf, 2003–2022. Open source book.

[12] Sheng Yu, Qingyu Zhuang, and Kai Salomaa. The state complexities of some basic operations on regular languages. *Theor. Comput. Sci.*, 125(2):315–328, 1994.

A Proofs of Lemmas 1, 2, 3

Proof of Lemma 1

The proof proceeds by induction on the syntactic structure of E. The result is immediate if E is equal to 1, is a letter, or is an iteration. In the case of a union, $1 \in \mathit{Lufs}(E_i)$ (for $i = 1, \ldots, n$), by induction hypothesis. Thus, $1 \in \mathit{Lufs}(E)$. Finally, in the case of a concatenation, $1 \in E_2$, by induction hypothesis. Thus, $1 \in \mathit{Lufs}(E)$, again.

(End of proof)

Proof of Lemma 2

Again, the proof proceeds by induction on the syntactic structure of E. The result is immediate if E is equal to 1, is a letter, or is a union. In the case of a concatenation, $E_1 \in \mathit{Lufs}(E_1)$, by induction hypothesis. Moreover, E_1 is not a concatenation, by hypothesis. Thus, $E_1 \odot E_2 = E_1 \,.\, E_2 = E$. Hence, $E \in \mathit{Lufs}(E_1) \odot E_2 \subseteq \mathit{Lufs}(E)$. Finally, if E is an iteration, say F^*, one has: $1 \in \mathit{Lufs}(F)$, by Lemma 1. Consequently, $E = 1 \odot E \in \mathit{Lufs}(F) \odot E \subseteq \mathit{Lufs}(E)$.

(End of proof)

Proof of Lemma 3

The proof is immediate if E_1 is not a concatenation, by definition of *Lufs* (see Figure 5). If E_1 is a concatenation, say $F_1 \,.\, F_2$, one has:

$$\begin{aligned}
\mathit{Lufs}(E_1 \odot E_2) &= \mathit{Lufs}((F_1 \,.\, F_2) \odot E_2) \\
&= \mathit{Lufs}(F_1 \,.\, (F_2 \odot E_2)) \\
&= \mathit{Lufs}(F_1) \odot (F_2 \odot E_2) \cup \mathit{Lufs}(F_2 \odot E_2) \\
&= (\mathit{Lufs}(F_1) \odot F_2) \odot E_2 \cup \mathit{Lufs}(F_2) \odot E_2 \cup \mathit{Lufs}(E_2) \\
&= (\mathit{Lufs}(F_1) \odot F_2 \cup \mathit{Lufs}(F_2)) \odot E_2 \cup \mathit{Lufs}(E_2) \\
&= \mathit{Lufs}(F1 \,.\, F_2) \odot E_2 \cup \mathit{Lufs}(E_2) \\
&= \mathit{Lufs}(E_1) \odot E_2 \cup \mathit{Lufs}(E_2)
\end{aligned}$$

The fourth equality uses an induction on E_1. The other equalities simply use the associativity of $\cup$ and $\odot$, and the definitions.

(End of proof)

B Correctness proof of the algorithm *computeDerivatives*.[14]

The correctness proof uses the following loop invariant:[15]

1. $S_{tod} \cap S_{dev} = \{\}$
2. $E \in (S_{tod} \cup S_{dev}) \subseteq D_* E$

[14] The algorithm *computeDerivatives* follows a very common scheme. Thus, I readily agree that proving its correctness probably is superfluous. Nonetheless, I prefer to provide a proof for the sake of beauty and for ensuring the completeness of this report, as well.

[15] Remember that a loop invariant is an assertion that holds every time the condition in the while loop is about to be evaluated.

3. $\forall F' \in S_{dev} : \forall x \in Letter : D_x\, F' \in S_{tod} \cup S_{dev}$

(I use $D_*\, E$ to denote the set of all derivatives of E, i.e. $D_*\, E = \{D_w\, E \mid w \in Letter^*\}$.) Let us check that the algorithm fulfills the loop invariant.

1. The invariant clearly holds the first time the condition $S_{tod} \neq \{\}$ is about to be evaluated, since we have $S_{tod} = \{E\}$ and $S_{dev} = \{\}$, at this point.
2. Every execution of the body of the while loop maintains the invariant:
   (a) The condition $S_{tod} \cap S_{dev} = \{\}$ is maintained since the only change to S_{tod} and S_{dev} is the fact that F is moved from S_{tod} to S_{dev}.
   (b) The second condition of the invariant also is maintained since the set $S_{tod} \cup S_{dev}$ is just extended with the direct derivatives of F not already present in it. Since F is a derivative of E, direct derivatives of F also are derivatives of E.
   (c) The third condition of the invariant is maintained because
      i. it holds before the execution of the loop,
      ii. the set S_{dev} is extended, with only one element: F,
      iii. the set $S_{tod} \cup S_{dev}$ is extended, with the direct derivatives of F not already in it, only.
3. Let us check that one has $S_{dev} = D_*\, E$ when the algorithm terminates (if it ever does).
   (a) $S_{dev} \subseteq D_*\, E$, by the condition 2. of the invariant and the fact that $S_{tod} = \{\}$.
   (b) $D_*\, E \subseteq S_{dev}$ because $E \in S_{dev}$ and because S_{dev} contains the direct derivatives of all its elements. Thus, by induction on the length of w, S_{dev} contains all derivatives $D_w\, E$ ($w \in Letter^*$).[16]
4. The algorithm eventually terminates because the number of elements in S_{dev} is increased by 1 at each iteration (a consequence of the condition 1. of the invariant) and because it is ensured by the condition 2. of the invariant that this number is bounded by the number of elements in $D_*\, E$, which is finite as proven in Section 3.4.2.

Finally, it is clear that the algorithm gathers exactly all equations $F = tabD_F$ such that $F \in D_*\, E$, in the set SEq, by executing the statement $SEq \Longleftarrow \mathrm{eq}(F,\ tabD)$ at each iteration.

C Termination proof for reduce(S)

To prove that the algorithm reduce(S) terminates, we observe that the number of equivalence classes in the background necessarily decreases by at least one unit, after *two* successive executions of the body of the main loop. Indeed, let ex1 and ex2 be two such executions. If the number of equivalence classes decreases during ex1, the result is proven. Otherwise, necessarily, the call merge(E_1, E_2) in the inner while loop cannot be executed during ex1. Since the algorithm does not terminate after ex1, at least two equations overlap, so that at least a pair $\langle E_1, E_2 \rangle$ of expressions belonging to different equivalence classes are put in S by ex1. Thus, the equivalence classes of these expressions are merged by a call merge(E_1, E_2), during ex2.

In conclusion, the execution of reduce(S) terminates because the number of equivalence classes cannot decrease indefinitely.

[16] It is possible to modify the condition 3. of the invariant to avoid the need of an induction proof here. But this "improved" condition 3. is more difficult to check.

D Example 1

Let E be the following (randomly generated) normalized regular expression:

$$((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^* + (\mathrm{a} + \mathrm{b}(1 + \mathrm{b})\mathrm{b})\mathrm{aa}(1 + \mathrm{a})$$

D.1 Syntactic derivatives of E

Note that expressions of the form E_w below stand for $D_w\, E$.

E	=	$((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^* + (\mathrm{a} + \mathrm{b}(1 + \mathrm{b})\mathrm{b})\mathrm{aa}(1 + \mathrm{a})$
E_a	=	$\mathrm{aa}(1 + \mathrm{a}) + \mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$
E_b	=	$\mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^* + (1 + \mathrm{b})\mathrm{baa}(1 + \mathrm{a})$
E_{bb}	=	$\mathrm{aa}(1 + \mathrm{a}) + \mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^* + \mathrm{baa}(1 + \mathrm{a})$
E_{aa}	=	$\mathrm{a}(1 + \mathrm{a}) + \mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$
E_{aaa}	=	$1 + \mathrm{a} + \mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$
E_{aaaa}	=	$1 + \mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$
E_{ab}	=	$\mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$

D.2 Equations relating the syntactic derivatives of E

It can be checked by hand that the syntactic derivatives of E are related by the equations below.

E	=	$1 + aE_a + bE_b$	E_{aa}	=	$1 + aE_{aaa} + bE_{ab}$
E_a	=	$1 + aE_{aa} + bE_{ab}$	E_{aaa}	=	$1 + aE_{aaaa} + bE_{ab}$
E_b	=	$1 + aE_{aa} + bE_{bb}$	E_{aaaa}	=	$1 + aE_{ab} + bE_{ab}$
E_{bb}	=	$1 + aE_{aa} + bE_{ba}$	E_{ab}	=	$1 + aE_{ab} + bE_{ab}$

These equations are reduced as follows. Since the right parts of the equations for E_{aaaa} and E_{ab} are identical, the two equations are put into the same equivalence class, of which E_{ab} is the representative, and E_{aaaa} is replaced by E_{ab} in the equation for E_{aaa}. Thus, the right part of this equation becomes identical to the one for E_{ab}. Continuing so, we see that all derivatives are put into a single equivalence class of which E_{ab} is the representative since it is the shortest of all the derivatives. All the equations are replaced by a single one: $E_{ab} = 1 + aE_{ab} + bE_{ab}$. So, at this point, E is simplified to E_{ab}, i.e. $\mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$. Interestingly, minimization of the set of equations is not needed in this case. Reducing the background is enough.

D.3 Partial derivatives of E

One can see that the partial derivatives of E are:

P_1	=	$((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$	P_4	=	$\mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$	P_7	=	$\mathrm{a}(1 + \mathrm{a})$
P_2	=	$(\mathrm{a} + \mathrm{b}(1 + \mathrm{b})\mathrm{b})\mathrm{aa}(1 + \mathrm{a})$	P_5	=	$(1 + \mathrm{b})\mathrm{baa}(1 + \mathrm{a})$	P_8	=	a
P_3	=	$\mathrm{aa}(1 + \mathrm{a})$	P_6	=	$\mathrm{baa}(1 + \mathrm{a})$	P_9	=	1

Therefore, one can write:

$E = P_1 + P_2$	$E_{bb} = P_3 + P_4 + P_6$	$E_{aaaa} = P_9 + P_4$
$E_a = P_3 + P_4$	$E_{aa} = P_7 + P_4$	$E_{ab} = P_4$
$E_b = P_4 + P_5$	$E_{aaa} = P_9 + P_8 + P_4$	

Observe that there are 9 partial derivatives but only 8 syntactic derivatives, although the number of syntactic derivatives can be exponentially greater than the number of partial derivatives.

D.4 Left unfolded suffixes of E

The set $Lufs(E)$ of the left unfolded suffixes of E is depicted below:

1	aaa(1 + a)	(1 + b)baa(1 + a)
a	baa(1 + a)	(a + b)a*((a + b)a*)*
1 + a	a*((a + b)a*)*	b(1 + b)baa(1 + a)
a(1 + a)	bbaa(1 + a)	(a + b(1 + b)b)aa(1 + a)
((a + b)a*)*	aa*((a + b)a*)*	((a + b)a*)* + (a + b(1 + b)b)aa(1 + a)
aa(1 + a)	ba*((a + b)a*)*	

It can be checked that

1. all partial derivatives P_1 to P_9 belongs to $Lufs(E)$;
2. all proper partial derivatives P_i are such that $x \odot P_i$ belongs to $Lufs(E)$ ($x \in Letter$);
3. $\sharp Lufs(E) = 17 < 26 = \text{size}(E) + 1$;
4. the number of proper partial derivatives (7) is less than $13 = (\text{size}(E) + 1)/2$.

E Example 2

Let us consider now the expression F = a*(aab + bb*a + bb)* from [8], page 43. The set of equations computed by the algorithm *computeDerivatives* (see Figure 6) is depicted below. In the right column the actual value of each syntactic derivative is displayed. The letter E stands for the sub-expression (aab + bb*a + bb) , as is done in [8].

F	=	1	+	$a.F_a$	+	$b.F_b$	a*E*
F_a	=	1	+	$a.F_{aa}$	+	$b.F_b$	a*E* + ab$E$*
F_b	=			$a.F_{ba}$	+	$b.F_{bb}$	bE* + b*aE*
F_{aa}	=	1	+	$a.F_{aa}$	+	$b.F_{aab}$	a*E* + bE* + ab$E$*
F_{ba}	=	1	+	$a.F_{baa}$	+	$b.F_b$	E*
F_{bb}	=	1	+	$a.F_{bba}$	+	$b.F_b$	E* + b*aE*
F_{aab}	=	1	+	$a.F_{bba}$	+	$b.F_{aab}$	E* + bE* + b*aE*
F_{baa}	=			$a.F_{baaa}$			ab$E$*
F_{bba}	=	1	+	$a.F_{bbaa}$	+	$b.F_b$	E* + ab$E$*
F_{baaa}	=					$b.F_{ba}$	bE*
F_{bbaa}	=			$a.F_{baaa}$	+	$b.F_{ba}$	bE* + ab$E$*

The equations above are fully equivalent to the (minimal) machine shown in Fig.5.2 of [8], page 42. As for the derivatives, they also are the same, except that Conway factorizes E in every expression. For instance, a$^*E^*$ + bE^* + abE^* becomes (a* + b + ab)E^*. It is a good idea to perform these factorizations to present the final result elegantly but it is not the best way to compute the syntactic derivatives (one after one). To argue this point further, let us now depict the derivatives and the equations computed by (my version of) Brzozowski's algorithm. See below.

F	=	1	+	$a.F_a$	+	$b.F_b$	a$^*E^*$
F_a	=	1	+	$a.F_{aa}$	+	$b.F_b$	a$^*E^*$ + abE^*
F_b	=			$a.F_{ba}$	+	$b.F_{bb}$	(b + b*a)E^*
F_{aa}	=	1	+	$a.F_{aa}$	+	$b.F_{aab}$	a$^*E^*$ + bE^* + abE^*
F_{ba}	=	1	+	$a.F_{baa}$	+	$b.F_b$	E^*
F_{bb}	=	1	+	$a.F_{bba}$	+	$b.F_{bbb}$	(1 + b*a)E^*
F_{aab}	=	1	+	$a.F_{bba}$	+	$b.F_{aabb}$	E^* + (b + b*a)E^*
F_{baa}	=			$a.F_{baaa}$			abE^*
F_{bba}	=	1	+	$a.F_{bbaa}$	+	$b.F_b$	E^* + abE^*
F_{bbb}	=			$a.F_{ba}$	+	$b.F_{bbbb}$	(b + b*a)E^* + b*aE^*
F_{aabb}	=	1	+	$a.F_{bba}$	+	$b.F_{aabbb}$	(b + b*a)E^* + (1 + b*a)E^*
F_{baaa}	=					$b.F_{ba}$	bE^*
F_{bbaa}	=			$a.F_{baaa}$	+	$b.F_{ba}$	bE^* + abE^*
F_{bbbb}	=	1	+	$a.F_{bba}$	+	$b.F_{bbb}$	(1 + b*a)E^* + b*aE^*
F_{aabbb}	=	1	+	$a.F_{bba}$	+	$b.F_{aabbb}$	(b + b*a)E^* + (1 + b*a)E^* + b*aE^*

We see that the number of equations has increased from 12 to 16. It is no longer minimal. Some derivatives are redundant. And it is because E is factorized in some sub-expressions, due to the use of the operation $\odot$ (instead of $\otimes$) in the definition of derivatives. By distributing E in all expressions and removing duplicates, we immediately see that the expressions F_{aabb} and F_{aabbb} are equivalent to F_{aab}, while F_{bbb} is equivalent to F_b, and F_{bbbb} is equivalent to F_{bb}. Fortunately, these equivalences can be detected by the operation reduce of the background without simplifying explicitly expressions: The equations for F_{aabb} and F_{aabbb} have identical right parts, so, F_{aabbb} is replaced by F_{aabb} in every equations. As a result, the equations for F_{aabb} and F_{aab} now have the same right parts. Thus, F_{aabb} is replaced by F_{aab} everywhere. Similarly, F_{bbbb} is first replaced by F_{bbb}, implying that F_{bbb} is replaced by F_b, afterwards. We are left with the following 12 equations.

F	=	1	+	$a.F_a$	+	$b.F_b$	a$^*E^*$
F_a	=	1	+	$a.F_{aa}$	+	$b.F_b$	a$^*E^*$ + abE^*
F_b	=			$a.F_{ba}$	+	$b.F_{bb}$	(b + b*a)E^*
F_{aa}	=	1	+	$a.F_{aa}$	+	$b.F_{aab}$	a$^*E^*$ + bE^* + abE^*
F_{ba}	=	1	+	$a.F_{baa}$	+	$b.F_b$	E^*
F_{bb}	=	1	+	$a.F_{bba}$	+	$b.F_b$	(1 + b*a)E^*
F_{aab}	=	1	+	$a.F_{bba}$	+	$b.F_{aab}$	E^* + (b + b*a)E^*
F_{baa}	=			$a.F_{baaa}$			abE^*
F_{bba}	=	1	+	$a.F_{bbaa}$	+	$b.F_b$	E^* + abE^*
F_{baaa}	=					$b.F_{ba}$	bE^*
F_{bbaa}	=			$a.F_{baaa}$	+	$b.F_{ba}$	bE^* + abE^*

This set of equations is identical to the set of equations obtained previously but the syntactic derivatives are not all the same: E^* is factorized sometimes (but not everywhere).

F Example 3

Let E be the following expression, also borrowed from [8]: (ab*a + ba*b)*(1 + ab* + ba*). The derivatives of E and the corresponding equations are given below.

E	=	1	+	$a.E_a$	+	$b.E_b$	E
E_a	=	1	+	$a.E$	+	$b.E_a$	b* + b*aE
E_b	=	1	+	$a.E_b$	+	$b.E$	a* + a*bE

For this example, Brzozowski's method gives exactly the same result than the method proposed in this paper. One see immediately that there is no opportunity to reduce the set of equations. However, the set of equations is not minimal. Clearly, it defines a DFA in which all states are accepting. So, minimization reduces the set of equation to a single one:

E	=	1	+	$a.E$	+	$b.E$	E

The expression E is proposed by Conway as a difficult example to simplify (i.e. to prove equal to (a + b)*), using the classical Kleene's axioms (see [8], page 25). Minimization seems to be needed mainly for such difficult expressions.

G Example 4

We consider the expressions E of the form (a + b)*a(a + b)$^{n-1}$ where $n \geq 1$, from [4]. We depict their derivatives and their equations, for $n = 3$, hereunder.

E	=			$a.E_a$	+	$b.E$	E
E_a	=			$a.E_{aa}$	+	$b.E_{ab}$	$(a + b)^3 + E$
E_{aa}	=			$a.E_{aaa}$	+	$b.E_{aab}$	$(a + b)^2 + (a + b)^3 + E$
E_{ab}	=			$a.E_{aba}$	+	$b.E_{abb}$	$(a + b)^2 + E$
E_{aaa}	=			$a.E_{aaaa}$	+	$b.E_{aaab}$	$a + b + (a + b)^2 + (a + b)^3 + E$
E_{aab}	=			$a.E_{aaba}$	+	$b.E_{aabb}$	$a + b + (a + b)^2 + E$
E_{aba}	=			$a.E_{abaa}$	+	$b.E_{abab}$	$a + b + (a + b)^3 + E$
E_{abb}	=			$a.E_{abba}$	+	$b.E_{abbb}$	$a + b + E$
E_{aaaa}	=	1	+	$a.E_{aaaa}$	+	$b.E_{aaab}$	$1 + a + b + (a + b)^2 + (a + b)^3 + E$
E_{aaab}	=	1	+	$a.E_{aaba}$	+	$b.E_{aabb}$	$1 + a + b + (a + b)^2 + E$
E_{aaba}	=	1	+	$a.E_{abaa}$	+	$b.E_{abab}$	$1 + a + b + (a + b)^3 + E$
E_{aabb}	=	1	+	$a.E_{abba}$	+	$b.E_{abbb}$	$1 + a + b + E$
E_{abaa}	=	1	+	$a.E_{aaa}$	+	$b.E_{aab}$	$1 + (a + b)^2 + (a + b)^3 + E$
E_{abab}	=	1	+	$a.E_{aba}$	+	$b.E_{abb}$	$1 + (a + b)^2 + E$
E_{abba}	=	1	+	$a.E_{aa}$	+	$b.E_{ab}$	$1 + (a + b)^3 + E$
E_{abbb}	=	1	+	$a.E_a$	+	$b.E$	$1 + E$

As a matter of fact, all algorithms for computing derivatives discussed in Section 5 produces the same derivatives and the same equations. The set of equations is minimal. There is no need to reduce

the background nor to minimize the set of equations (the latter should be expensive). The execution times are similar for all versions of the algorithm. This is because computing the direct derivatives of any partial derivative is done in constant time. For instance, in the case of an expression (a + b)^i $(i > 1)$, only 4 calls to *deriveByUnfold* are executed and every call executes a fixed number of operations all executed in constant time. Thus, memoizing the calls to *deriveByUnfold* brings no improvement. Full derivatives are unions and are executed in $O(n^2)$. One can see, on the example above, that they are computed exactly two times. This leads to an overal complexity of $O(n^2\,2^n)$ for computing all derivatives and equations. Computing the direct derivatives of a union can be done in $O(n \log n)$ but it does not bring a significant improvement since only small values of n can be used, as the space complexity of the result clearly is $O(n\,2^n)$.

H Example 5

In this section, we consider the expressions (a + b)*b(ab*)$^{n-2}$((ab*)$^{n-1}$)* , where $n \geq 2$. This example, cited in [2], comes from [12]. Below, we show the derivatives and equations produced by our algorithms, for $n = 4$. For readability, we note A = b*(ab*ab*ab*)*, B = b*a A, C = a B, D = b* C, E = (a + b)*b C = (a + b)*bab*ab*(ab*ab*ab*)*.

All algorithms compute the same set of 2^n syntactic derivatives and equations. More precisely, for $n = 4$, we get the following 16 entries:

E	=			$a.E$	+	$b.E_b$		E
E_b	=			$a.E_{ba}$	+	$b.E_b$		$C + E$
E_{ba}	=			$a.E_{baa}$	+	$b.E_{bab}$		$B + E$
E_{baa}	=	1	+	$a.E_{baaa}$	+	$b.E_{baab}$		$A + E$
E_{bab}	=			$a.E_{baba}$	+	$b.E_{bab}$		$B + C + E$
E_{baaa}	=			$a.E_{ba}$	+	$b.E_{baaab}$		$E + D$
E_{baab}	=	1	+	$a.E_{baaba}$	+	$b.E_{baab}$		$A + C + E$
E_{baba}	=	1	+	$a.E_{babaa}$	+	$b.E_{babab}$		$A + B + E$
E_{baaab}	=			$a.E_{ba}$	+	$b.E_{baaab}$		$C + E + D$
E_{baaba}	=			$a.E_{baba}$	+	$b.E_{baabab}$		$B + E + D$
E_{babaa}	=	1	+	$a.E_{baaba}$	+	$b.E_{babaab}$		$A + E + D$
E_{babab}	=	1	+	$a.E_{bababa}$	+	$b.E_{babab}$		$A + B + C + E$
E_{baabab}	=			$a.E_{baba}$	+	$b.E_{baabab}$		$B + C + E + D$
E_{babaab}	=	1	+	$a.E_{baaba}$	+	$b.E_{babaab}$		$A + C + E + D$
E_{bababa}	=	1	+	$a.E_{bababa}$	+	$b.E_{bababab}$		$A + B + E + D$
$E_{bababab}$	=	1	+	$a.E_{bababa}$	+	$b.E_{bababab}$		$A + B + C + E + D$

Four pairs of equations overlap. They correspond to the following pairs of derivatives: $\langle E+D\ ,\ C+E+D\rangle$, $\langle B+E+D\ ,\ B+C+E+D\rangle$, $\langle A+E+D\ ,\ A+C+E+D\rangle$, $\langle A+B+E+D\ ,\ A+B+C+E+D\rangle$. This is easily explained by the fact that the expressions D and $C + D$ exactly have the same direct derivatives, implying that adding C in the computation of the direct derivatives of $E+D$, $B+E+D$, $A + E + D$, $A + B + E + D$ does not change the result (remember the definitions of the operations $\oplus$ and $\oplus$=). Experiments with other values of n show that it is always the case that 2^{n-2} pairs of equations overlap. Thus, reducing the background reduces the set of equations by 25%. For $n = 4$, we obtain the following equations and derivatives:

E	$=$			$a.E$	$+$	$b.E_b$	E
E_b	$=$			$a.E_{ba}$	$+$	$b.E_b$	$C + E$
E_{ba}	$=$			$a.E_{baa}$	$+$	$b.E_{bab}$	$B + E$
E_{baa}	$=$	1	$+$	$a.E_{baaa}$	$+$	$b.E_{baab}$	$A + E$
E_{bab}	$=$			$a.E_{baba}$	$+$	$b.E_{bab}$	$B + C + E$
E_{baaa}	$=$			$a.E_{ba}$	$+$	$b.E_{baaa}$	$E+ D$
E_{baab}	$=$	1	$+$	$a.E_{baaba}$	$+$	$b.E_{baab}$	$A + C + E$
E_{baba}	$=$	1	$+$	$a.E_{babaa}$	$+$	$b.E_{babab}$	$A + B + E$
E_{baaba}	$=$			$a.E_{baba}$	$+$	$b.E_{baaba}$	$B + E + D$
E_{babaa}	$=$	1	$+$	$a.E_{baaba}$	$+$	$b.E_{babaa}$	$A + E + D$
E_{babab}	$=$	1	$+$	$a.E_{bababa}$	$+$	$b.E_{babab}$	$A + B + C + E$
E_{bababa}	$=$	1	$+$	$a.E_{bababa}$	$+$	$b.E_{bababa}$	$A + B + E + D$

This is not the whole story since minimizing the new set of equations reduces its size by a further 4 (2^{n-2}, in general). This cannot be detected by means of overlapping, but looking at the new equations we can deduce that the following pairs of derivatives are equivalent: $\langle C + E\ ,\ E + D\rangle$, $\langle B+C+E\ ,\ B+E+D\rangle$, $\langle A+C+E\ ,\ A+E+D\rangle$, $\langle A+B+C+E\ ,\ A+B+E+D\rangle$. This can be done by "solving" the equations corresponding to the derivatives, using Salomaa's rule (see e.g. [8], Theorem 4, page 107), and concluding that both expressions are equivalent to a third one.[17] For instance,[18]

- $C + E = E_b = a.E_{ba} + b.E_b = b^*a.E_{ba}$
- $E + D = E_{baaa} = a.E_{ba} + b.E_{baaa} = b^*a.E_{ba}$

Finally, we get the following set of 8 equations and derivatives (2^{n-1} in general, see [12]).

E	$=$			$a.E$	$+$	$b.E_b$	E
E_b	$=$			$a.E_{ba}$	$+$	$b.E_b$	$C + E$
E_{ba}	$=$			$a.E_{baa}$	$+$	$b.E_{bab}$	$B + E$
E_{baa}	$=$	1	$+$	$a.E_b$	$+$	$b.E_{baab}$	$A + E$
E_{bab}	$=$			$a.E_{baba}$	$+$	$b.E_{bab}$	$B + C + E$
E_{baab}	$=$	1	$+$	$a.E_{bab}$	$+$	$b.E_{baab}$	$A + C + E$
E_{baba}	$=$	1	$+$	$a.E_{baab}$	$+$	$b.E_{babab}$	$A + B + E$
E_{babab}	$=$	1	$+$	$a.E_{babab}$	$+$	$b.E_{babab}$	$A + B + C + E$

It is also interesting to compare the efficiency of the different algorithms, on this example. This is done below. The algorithms have been applied to the expressions for $n = 2$ to $n = 20$. The meaning of the other columns are the same as in Tables 1 and 2, except t_{Br}, which is the derivation time for Brzozowski's algorithm, and t_{Sal}, which gives the minimization time for an algorithm that systematically applies Salomaa's rule to all relevant equations obtained after background reduction. Such an algorithm can be efficiently implemented using existing data structures provided by the

[17] This kind of reasoning is a bit outside the scope of the current paper. But this could be an avenue for future research on extending the algorithm of Figure 7.

[18] The symbol = is used with three different meanings here. But all of them imply equivalence of the denoted regular languages.

background, in $O(nl^2 \times neq)$, where nl and neq are the number of letters used by the expressions and the number of equations, respectively. In this particular case, the time complexity thus is $O(2^n)$. We can observe significant differences in the computation times of the different algorithms, although those differences are less dramatic than those of Section 5. They are clearer for "large" values of n. The timings for Brzozowski algorithm are significantly greater than those of the standard algorithm of this paper: 25% larger for $n = 20$. The memoizing version is better than the standard one: 20% faster for $n = 20$. Finally, the most optimized version, which computes the partial derivatives at once and efficiently merges arrays of identifiers still is 27% faster (for $n = 20$, again). Since all versions of the algorithms compute the same syntactic derivatives, the differences of efficiency stem solely from algorithmic aspects: The algorithm *deriveByUnfold* of the standard method is replaced by a more recursive algorithm that creates much more arrays of derivatives and makes heavier use of the operation $\odot$, in Brzozowski's version. The memoizing version computes the derivatives of the partial derivatives much less often ($\sharp P$) than the standard version. Finally, the most optimized version computes unions of arrays of identifiers more efficiently than the (simple) memoizing version, and it computes no intermediate expressions (see Subsection 5.4). We can also see that the timings t_{Sal} reported for the ad hoc minimizing algorithm are much better than the timings t_{min} reported for the general minimizing algorithm. The latter has a less good theoretical complexity. We moreover notice that the reduction of the background is more efficient than the two minimizing algorithms. Its complexity is $O(nl \times neq')$, where neq' is the number of pairs of overlapping equations (in this case: 2^{n-2}).

n	$\sharp CD$	$\sharp CDA$	t_d	t_{Br}	$\sharp P$	$\sharp PA$	t_{d0}	t_{d1}	t_{red}	t_{min}	t_{Sal}
2	4	49	455 μ	529 μ	3	16	513 μ	567 μ	62.2 μ	98.6 μ	83.8 μ
3	8	115	280 μ	329 μ	4	20	223 μ	350 μ	31.3 μ	68.2 μ	83.8 μ
4	16	271	395 μ	392 μ	5	25	357 μ	372 μ	47.1 μ	150 μ	114 μ
5	32	623	540 μ	536 μ	6	30	534 μ	591 μ	81.4 μ	288 μ	192 μ
6	64	1.40 K	802 μ	776 μ	7	35	799 μ	758 μ	121 μ	380 μ	302 μ
7	128	3.13 K	1.10 m	1.54 m	8	40	1.01 m	1.09 m	148 μ	601 μ	433 μ
8	256	6.91 K	1.75 m	1.77 m	9	45	1.28 m	1.93 m	281 μ	730 μ	602 μ
9	512	15.1 K	1.88 m	1.40 m	10	50	3.08 m	2.53 m	342 μ	1.17 m	869 μ
10	1.02 K	32.7 K	3.30 m	2.85 m	11	55	2.91 m	3.99 m	303 μ	1.91 m	1.20 m
11	2.04 K	70.6 K	5.78 m	6.38 m	12	60	5.69 m	7.89 m	514 μ	3.59 m	2.18 m
12	4.09 K	151 K	11.3 m	13.4 m	13	65	8.05 m	7.49 m	979 μ	8.44 m	4.54 m
13	8.19 K	323 K	24.1 m	39.2 m	14	70	18.7 m	16.3 m	2.31 m	13.5 m	10.1 m
14	16.3 K	688 K	64.6 m	96.8 m	15	75	49.3 m	34.7 m	2.64 m	30.4 m	15.8 m
15	32.7 K	1.45 M	98.7 m	125 m	16	80	85 m	78.8 m	4.78 m	56.3 m	36.4 m
16	65.5 K	3.08 M	216 m	269 m	17	85	181 m	115 m	22 m	151 m	50.7 m
17	131 K	6.48 M	418 m	539 m	18	90	326 m	225 m	22.5 m	278 m	96.7 m
18	262 K	13.6 M	885 m	1.29 s	19	95	686 m	632 m	45.3 m	629 m	189 m
19	524 K	28.5 M	2.10 s	2.62 s	20	100	1.70 s	1.28 s	96.3 m	1.52 s	396 m
20	1.04 M	59.7 M	4.31 s	5.75 s	21	105	3.47 s	2.55 s	206 m	3.63 s	858 m