Making Descriptions Of Regular Languages Communicate To Simplify Regular Expressions

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

July 27, 2023

Abstract

To be filled in…

Chapter 1

Introduction

1.1 Motivation

As stated in the title, this paper reports on the results of my long time research devoted to the problem of simplifying regular expressions. This problem is difficult and even intractable in full generality for (worst-case) complexity reasons: For instance, the apparently much simpler problem of deciding whether a regular expression represents all possible chains is PSPACE-complete (Dexter Kozen, personal communication, June 12, 2020). Nevertheless, when reasonable, it may be interesting to simplify a regular expressions in order to "grasp" its meaning (i.e., the regular language it denotes). As an example, consider the expression c* + c*a(c*a + b)*c*. It is equivalent to the simpler expression (c + ab*)* , which is more understandable. Simplifying the former into the latter can be done using the classical axioms of Kleene's theory [10] but it requires some "expertise". My goal was to design an algorithm able to compete with experts to perform such simplifications.

To my knowledge, few authors have proposed algorithms to simplify regular expressions. A notable exception is Alley Stoughton in [16]. She proposes simplification rules as well as two simplification algorithms. However, each step of her method transforms an expression into a shorter one, which makes the method unable to perform the simplification above: To simplify an expression by means of a sequence of equalities, it is generally necessary first to show it equal to a larger one, which can then be more easily simplified [7, 8, 9].

This paper is *informal* to be more easily understandable than papers written in the usual academic style, using formal definitions and theorems but lacking intuition. I hope that this way of presenting my results will make sense for the selected readers to whom I plan to send this document. I expect to get substantial feedback from them, which will help me to write a more formal and detailed paper, in a more classical style (if I can). In that sense, this report is *preliminary.*

1.2 Overview

Now, I am willing to give a broad view of what I have achieved. Regular languages can be represented ("specified") in several equivalent ways (see, e.g. [1]): Regular expressions, deterministic and non deterministic finite automata, right-linear grammars (among others). Efficient algorithms exist to translate a representation into another. Different representations can be better than others for different goals, but, to my knowledge, until now, no author has suggested to work on regular languages by constantly maintaining several representations of the languages under consideration. Doing so is the key idea of the method I propose. Such an idea is rarely exploited in the literature but a notable exception is the representation of polyhedrons (see, e.g. [4]). My proposal is very similar to what is done for polyhedrons: While dealing with regular expressions (to simplify them, let's say), I maintain deterministic automata for their respective languages. This allows me to maintain equivalent classes of regular expressions and to simplify expressions by choosing shortest expressions in the classes. (The same idea was used in [7, 8, 9] to simplify expressions in the context of an equational theory.)

It is often asserted that using deterministic automata (in fact, minimal deterministic automata) to decide the equality of two regular languages is computationally too costly (e.g., [2, 16]). There are (at least) two reasons why this assertion is not always true in the context of simplifying regular expressions: Firstly, simplifying sub-expressions of an expression before computing an automaton for it can drastically reduce the size of the expression, the automaton of which can then be much smaller than an automaton for the original expression. (Of course, the minimal deterministic automaton is the same for both expressions, but it is normally obtained by minimizing two non minimal automata, of possibly very different sizes.) Secondly, my method maintain a bank of data structures that are frequently reused at a negligible cost. This is similar to what is done for BDDs in [5].

The rest of this paper is organized as follows. In Section 2, I explain different ways/approaches that can be used to simplify regular expressions. None is best in all situations I have tried. Therefore, which method(s) to use is a parameter of my implemented system. Section 3 describes the data structures and the algorithms that I propose, at a "high level", i.e., without emphasis on the fact that they can be efficiently implemented. This important aspect is dealt with in Section 3.5.5. As an interlude between these two sections, a number of motivating simplification examples is presented in Section 3.5.5. They involve the simplification of "large" randomly generated expressions, as well as the simplification of smaller "difficult" ones found in [2, 3, 10, 11]. As usual, the paper ends with Sections 3.5.5 and 3.5.5 on related and future work.

Chapter 2

A framework to represent regular languages and its use for simplifying regular expressions

2.1 The framework

In what follows, I use a unified framework to simultaneously represent sets of regular expressions together with their derivatives (see, e.g., [2, 3, 6, 10]), and the deterministic finite automata accepting the languages denoted by the expressions. Equivalent regular expressions are gathered in equivalence classes and a smallest expression of minimal size is chosen as a representative in each class. Each class possibly is associated with an equation of the form

$$E = o + a\,E_a + b\,E_b + \dots,$$

where E is the representative of the class, o stands for 0 or 1 and E_x is the derivative of E with respect to x. (Above I am using the notations from [10].)

However, it is necessary to elaborate a bit about the notion of a derivative and to precisely explain how I use the term derivative in this paper. It is my feeling that the notion of derivative is somewhat ambiguously used in the litterature (e.g., in [10]) because it is not always clear whether it applies to a regular language or to a regular expression. In the former case, the definition is clear[1] but it cannot be directly used to build automata since a regular language is an (often infinite) set of chains, not a formal object. In the latter case, there is a problem since many regular expressions are candidate to be *the* derivative of an expression. The recursive definition of derivatives given in [10] correctly refers to regular languages (called "events") but when applying it to build deterministic finite automata, the author uses regular expressions plus "expertise" to represent the regular languages (see [10], exercise 7, page 121). Hence, strictly speaking, the derivative of a regular expression is not well defined (as an expression).A way to uniquely define derivatives of a regular expression as regular expressions was proposed in [2]. It requires to use a (syntactically) restricted form of

[1]The derivative of a set of chains E with respect to a chain w is the set of all chains u such that $w\,u$ belongs to E.

regular expressions, as well as operations that maintains the restricted form when combining expressions. This ensures that a deterministic automaton can be mechanically computed for any regular expression without resorting to any "expertise". In my work, I use a similar restricted form of regular expressions[2] (see Section 3) to compute the set of all (syntactic) derivatives of an expression, which also provides an equation for every derivative. Notice that any derivative belongs to an equivalence class of expressions. Therefore, the equations must be modified by replacing every derivative by the representative of its equivalence class. It is that representative that I call a derivative of the expression. As a consequence, the actual value of a derivative depends on the "context" defined by the current set of expressions (represented in the "system") and their equivalence classes.

When expressions are simplified, the representatives of classes may change since they must remain smallest in the class. Consequently, equations have to be modified again, to ensure that equations only use representatives. Let us call E, the left part, and $o + a\,E_a + b\,E_b + \ldots$, the right part of an equation as above. Whenever the right parts of two equations become equal, the equivalence classes of their left part must be merged. (I say that the classes are *unified.*) Conversely, when two classes are unified, the classes of their derivatives must be unified in turn. The whole process is guaranteed to terminate because the number of classes is decreased by one at each unification. Equations can be used to build deterministic automata [6, 10]. In fact, in what follows, I do not make any difference between deterministic automata and sets of equations, except for the fact that a set of equations that represents an automaton must be *complete* in the sense that each representative used in the right part of an equation must appear as the left part of an equation (possibly the same).[3] If E is the left part of an equation, the (smallest) complete set of equations determined by this equation defines a deterministic automaton that is a recognizer for the regular language denoted by E.

Unfortunately, unifying classes does not guarantee that distinct classes represent different regular languages. We can further refine a complete set of equations by computing the equivalence classes of the "nodes" of the corresponding automata (see, e.g., [1]). (Two nodes are equivalent is they are the starting nodes of two automata accepting the same language.) Equivalent "nodes" in fact are representatives of equivalence classes of regular expressions that we further unify, as explained above, to get a minimal set of equations, all left parts of which denote different regular languages.

2.2 An overview of the algorithms

To give a first understanding of my simplification methods, I now show how they work on a single *running* example. Consider the following simple regular expression:

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

[2]I call them *normalized* regular expressions.

[3]Representing automata by such sets of equations may appear to be "inefficient". But Section 3.5.5 will show that my low-level implementation of equations makes it as efficient and, in fact, technically equivalent to the straightforward implementation of deterministic automata.

This expression is a "randomly chosen" one.[4] Its size is equal to 51 since I define the size of an expression as the number of its symbols, excluding parentheses (concatenation is noted by juxtaposition but it counts for one symbol). It is argued in [16] that this simple size notion is not elaborated enough to compare regular expressions. In fact, it is rather straightforward to use the more complicated notion of [16] in the context of my work (at the price of losing some efficiency) but, in practice, my simpler notion almost always gives the same results (see Section 3.5.5).

2.2.1 The core algorithm

The most fundamental algorithm of my work proceeds as follows.

1. The sub-expressions of the main expression are simplified first.
2. The main expression is *pre-simplified* by replacing its direct sub-expressions by their simplified version.
3. The *syntactic* derivatives of the pre-simplified expression are computed, giving rise to a set of equations equivalent to a DFA (deterministic finite automata) accepting the language denoted by the expression. This set of equations is added to the set of all previously existing equations, which is then reduced to ensure that no two equations have the same left part or the same right part.
4. The set of all equations is minimized to keep only one equation for every represented regular language. For each set of equivalent regular expressions found by the algorithm, a smallest expression is kept as representative. This is the way (other) expressions are simplified.

Let us see how the algorithm works on the running example. We will see that the initial expression is simplified to the following expression (of size 22):

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

However, at the final step of the algorithm, this result is just obtained by pre-simplification of the initial expression. A more interesting step is the simplification of the sub-expression:

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

Its pre-simplification does not change it. Thus its syntactic derivatives are computed. There are eight such derivatives (see Appendix A.1.1). It happens that all of them denote the same regular language. Every such derivative gives rise to an equation that is added to the current set of equations. Then, the set of all equations is reduced to ensure that no two equations have the same left part or the same right part. This results to the fact that all derivatives are

[4] I have writen an algorithm to choose an expression of a given length, which uses a given number of letters. Every expression has the same probablility of being chosen. In this case, the number of letters is two and the length is 80, but it is reduced to 51, just by normalizing the expression.

found equivalent and are put in the same equivalence class. As consequence, the expression E is simplified to

$$((a + b)a^*)^*$$

Notice that this "simplest" expression is not part of the list of derivatives (see again A.1.1). The additional simplification comes from the fact that the two expressions ((a + b)a*)* and a*((a + b)a*)* were already part of the same equivalence class *before* the simplification of E.

It is also interesting to show how is simplified the sub-expression

$$F = (1 + a + b + b^*)(((a + b)a^*)^* + (a + b(1 + b)b)aa(1 + a))$$

Note that $F = (1 + a + b + b^*)E$. Thus, F is first pre-simplified to

$$F' = (1 + a + b + b^*)((a + b)a^*)^*$$

The expression F' has four syntactic derivatives including ((a + b)a*)* . This time, however, the reduction of the four equations (inside the set of all equations) does not result in a single equation. There are two equations left:

$$\begin{aligned} F' &= 1 + aF'_a + bF' \\ F'_a &= 1 + aF'_a + bF'_a \end{aligned}$$

These two equations define a deterministic automaton with only two states. The classical algorithm to determine equivalent nodes in a deterministic automaton is applied to it, showing that F' and F'_a denote the same language. Hence, the equivalence classes of F' and F'_a are unified. That is: The expression F is ultimately simplified to

$$((a + b)a^*)^*$$

2.2.2 Adding simplification rules

The core algorithm can be enhanced with simplification rules that remove redundant sub-expressions or redundant occurrences of the star symbol (*). I use simplification rules inspired by [16] but there are a number of differences: a) some rules from [16] are useless since they are subsumed by the core algorithm, b) the rules must be re-written to apply to normalized expressions (notably the fact that unions are sorted lists of expressions), and, much more importantly, c) I use an exact algorithm to compute inclusion of the languages denoted by the expressions. It is argued in [16] that such an algorithm is computationally too costly but, due to the compact representation of the set of equations and the fact that the derivatives of an expression are computed only once, the cost remains acceptable in many cases.

In the case of our running example, adding simplification rules, we obtain the following result (of size 15):

$$b(a + b(a + b^*))(a + b)^*$$

Let us show how sub-expressions of

$$b(a + b(1 + a + b^*b))((a + b)a^*)^*$$

are further simplified. To simplify ((a + b)a*)*, the algorithm uses the following rule ([16]):[5]

$$E \geq 1 \;\;\&\;\; E \leq F^* \quad\rightarrow\quad (F\,E)^* = F^*$$

The rule is instantiated with E = a* and F = a + b. To prove that $E \leq F^*$, the algorithm makes sure that $E + F^* = F^*$, i.e., that a* + (a + b)* = (a + b)*. Only the derivatives and the equations for a* + (a + b)* must be computed since the equations for (a + b)* already exist. Only one new equation is created. Putting together the equations for a* + (a + b)* and (a + b)* and reducing them shows that the expressions are equivalent. Note also that checking the condition $E \geq 1$ is immediate, using the equation for E. The proof that (1 + a + b*b) simplifies to a + b* is in two steps. First, b*b is replaced by b* inside the union, because the union contains 1. Afterwards, a minimal covering of (1 + a + b*) is computed, which is (a + b*).

Notice that we apply the simplification rules after the pre-simplification of the expression, and before computing the derivatives of the pre-simplified expression. The idea is that simplifying the expression further may reduce the cost of computing the derivatives. However, it is often the case that the derivatives of the expression have already been computed by the simplification rules due to the computation of the relation ≤ between some expressions.

2.2.3 Solving equations

As soon as a reduced set of equations has been computed for a sub-expression E, we can use it to compute a (possibly) new expression equivalent to E, which can be shorter than E, sometimes. The algorithm I use to do so is related to a well-known algorithm to solve a set of so-called regular expression equations (see, e.g., [1]). This algorithm makes extensive use of Salomaa's rule (see, e.g., [10]):

$$E \not\geq 1 \;\;\&\;\; X = E\,X + F \quad\rightarrow\quad X = E^*F$$

My own algorithm is a top-down one, starting at the top equation of the set of equations to solve. Let see how it works on the running example: Consider the sub-expression

$$X = (\mathrm{a} + \mathrm{b}(\mathrm{a} + \mathrm{b}^*))(\mathrm{a} + \mathrm{b})^*$$

which is part of the simplified expression of Section 2.2.2. After computing the derivatives and reducing the equations, we are left with only two equations:

$$\begin{array}{rcl} X & = & a\,X_a + b\,X_a \\ X_a & = & 1 + a\,X_a + b\,X_a \end{array}$$

We can see, from the equations, that $X_a = 1 + X$. Thus, we can write:

$$X \;=\; a(1+X) + b(1+X) \;=\; (\mathrm{a} + \mathrm{b})X + (\mathrm{a} + \mathrm{b})$$

[5]Obviously, in this rule (and elsewhere), I make a (very) ambiguous use of the equality symbol =, either to denote syntactic equality or equivalence (equality of the denoted languages). Using different symbols would be straightforward, of course, but a bit ugly, in my view. So, I hope and I guess that the reader will readily find the right interpretation in each and every case.

Finally, applying Salomaa's rule, we get:

$$X = (\mathrm{a} + \mathrm{b})^*(\mathrm{a} + \mathrm{b})$$

Therefore, adding the algorithm to solve equations, the running example is simplified to the following expression (of size 10):

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

The reader can see on the example above that my algorithm to solve equations works a bit differently from what would seem obvious. The reason is that it attemps to delay the application of Salomaa's rule to the top equation in order to get an expression of the form $E^* F$ with E and F as short as possible. In the example under consideration, however, the obvious way to solve the equations first gives $X_a = (\mathrm{a} + \mathrm{b})^*$ by application of Salomaa's rule to the second equation. Then, the first equation gives:

$$X \; = \; aX_a + bX_a \; = \; (\mathrm{a} + \mathrm{b})X_a \; = \; (\mathrm{a} + \mathrm{b})(\mathrm{a} + \mathrm{b})^*$$

This solution is as satisfactory as the first one, in this case. (But it is nevertheless different.)

An important fact about solving equations is that the actual values of the expressions used by the equations are not used at all by the solving process.[6] This allows us to use the solving of equations to apply set operations on regular expressions. Let us consider, for example, the expression

$$((\mathrm{a} + \mathrm{b})^* \;\backslash\; \mathrm{a}^*)$$

which denotes the set difference between the languages denoted by $(\mathrm{a} + \mathrm{b})^*$ and a^* . "Simplifying" this *extended* regular expression with my algorithm, enabled with the solving of equations, gives the following result:

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

which describes, in a rather compact way, all non empty chains containing at least one occurrence of b, i.e., what is expected. To get this behaviour, I have first modified the size notion for extended regular expressions, by giving a very big weight (say 2^{31}) to the symbol \. Hence, any classical expression is normally smaller that an equivalent extended one, so that eliminating the \ symbol boils down to simplify the expression to a non extended one.

Now the "simplification" of $((\mathrm{a} + \mathrm{b})^* \;\backslash\; \mathrm{a}^*)$ works as follows: We use the fact ([10]) that any derivative of a difference $E \;\backslash\, F$ is equal to the difference between the corresponding derivatives of E and F. This allows us to compute all the derivatives of $((\mathrm{a} + \mathrm{b})^* \;\backslash\; \mathrm{a}^*)$ and their corresponding equations. Note that some expressions used by the equations are extended ones but, as noted above, we do not need them to solve the equations. In fact, only two equations are computed for $E \;=\; ((\mathrm{a} + \mathrm{b})^* \;\backslash\; \mathrm{a}^*)$:

$$\begin{aligned} E \;&=\; a\,E \;+\; b\,E_b \\ E_b \;&=\; 1 \;+\; a\,E_b \;+\; b\,E_b \end{aligned}$$

Solving the equations first gives $E_b \;=\; (\mathrm{a} + \mathrm{b})^*$, by Salomaa's rule. Then, one has: $E \;=\; a\,E \;+\; b\,(\mathrm{a} + \mathrm{b})^*$, which implies that $E \;=\; \mathrm{a}^*\,b\,(\mathrm{a} + \mathrm{b})^*$, by Salomaa's rule, again.

[6]Note, however, that the fact that we already know a value for the left part of every equation may allow us to cut down the solving process in some cases.

Chapter 3

Abstract data structures and algorithms

In this chapter, I present the data structures and the algorithms constituting my framework at a "high level". Clearly, the motivation is to make things more understandable than a direct description of how they are implemented. It is not always the case, however, that such a high level description allow the reader to understand whether the presented techniques are efficient, or not. For this reason, this chapter needs to be complemented by Chapter 3.5.5, devoted to explain the "key" choices of the implementation. (In fact, some notions and some algorithm are best described by their implementation, sometimes.)

The main data structure that is used in this work is the notion of *normalized regular expression*, which relieves us from using basic properties of the operators $+$ and $.$ to check the equivalence of some expressions. (See [2, 6] for a discussion of this issue.) Normalized regular expressions can be implemented efficiently in such a way that they are uniquely represented by an identifier, which is an integer. Thus checking syntactic equality boils down to check the equality of two integers. (See Chapter 3.5.5.)

I define normalized regular expressions in Section 3.1, as well as a number of operations to manipulate (construct) them. In Section 3.2, I define the syntactic derivatives of a normalized regular expression, and I provide efficient algorithms to compute them, which allows us to build a set of equations relating the derivatives to their own derivatives, i.e., a DFA for the original expression.

3.1 Normalized regular expressions

We assume a finite set *Letter* of *letters*. In practice, in this paper, I only use lowercase letters: a , b , . . . , z . A chain of letters, often denoted by 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 $S^\star$

is the set of chains $\{w_1 . w_2 . \ldots . w_n \mid w_1 , w_2 , \ldots , w_n \in S \ (n \geq 0)\}$.[1]

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 $E_1 \ .E_2$,[2] or a *star*, 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, 10, 16]). This set is denoted by $\mathcal{L}\,E$ and it 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 $E_1, E_2, \ldots, E_n$ 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 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.[3] Using this ordering, we impose the additional constraint that, in a union $E_1 + E_2 + \ldots + E_n$, the sequence $E_1, E_2, \ldots, E_n$ is stricly sorted.

   A normalized regular expression represents a set of chains according to the rules given in Figure 3.1. This set is denoted by $\mathcal{L}\,E$ and obviously is a regular set. The notations used in Figure 3.1 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 (e.g., Figure 3.2).

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.

(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

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

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

[3] For simplicity, this ordering is fixed by the implementation (see Section 3.5.5). It is different at each "run" of the program, typically depending on the current expression to simplify. This choice also is the most efficient for implementing the basic operations on normalized expressions.

Figure 3.1: Set of chains denoted by a normalized expression ($\mathcal{L}\,E$)

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

other algorithms. Notice that when writing normalized expression, I drop some parentheses, leaving to the reader the burden of parsing the expression properly. As a typical example, the expression

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

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

My choice of using normalized regular expressions should be contrasted with the choice of other authors, such as [2, 6], who check equivalence of regular expressions with respect to a congruence relation taking into account some of Kleene's classical axioms (see, e.g. [10]). With normalized expressions, testing such equivalence boils down to check syntactic equality.

Classical regular expressions are normalized thanks to three operations: union, concat, and iter. They take one or two normalized expressions, as argument, and they return a normalized expression denoting the union or the concatenation or the iteration of the language(s) denoted by their argument(s). For the sake of beauty, I preferably 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.

Definition 2. Operations on normalized regular expressions

Let E, E_1, and E_2 be normalized expressions.

- The operation union
  1. $\mathrm{union}(0, E) = \mathrm{union}(E, 0) = E$
  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 $F_1, \ldots, F_m$, the strictly ordered sequence of normalized expressions such that $S_1 \cup S_2 = \{F_1, \ldots, F_m\}$. Then, by

definition, 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. concat$(0, E) =$ concat$(E, 0) = 0$
  2. concat$(1, E) =$ 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 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, concat$(E_1, E_2) = F_1 \,.\, G$ where $G =$ 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)

3.2 Computing syntactic derivatives

In this section, I explain in details how I compute *syntactic* derivatives of normalized expressions. My method is much more efficient than a direct application of the classical characterization of derivatives as given in [10], which requires to use additional simplification rules, and is basically intended to be applied “by hand”. It is very closed to the method proposed in [2] but my presentation is somewhat different and more intuitive, in my opinion, altough Antimorov uses standard regular expressions, not normalized ones. Computing syntactic derivatives provides equations relating expressions to their derivatives (as first sketched in

Section 2). These equations can be further refined during the simplification process, in the sense that the expressions occurring in their left and right parts can be replaced by other, normally smaller, ones, but I do not take these questions into account here, concentrating only on the correct computation of the syntactic derivatives. Thus, this section is interesting in its own, independently of its use for simplifying regular expressions. It gives an efficient method for computing a DFA from a regular expression.

This section is structured as follows: In Subsection 3.2.1, I motivate my method 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 3.2.2, I give a precise definition of (my notion of) syntactic derivative. In Subsection 3.2.3, I propose an algorithm that efficiently computes all direct syntactic derivatives at once, and I prove its correctness. In Subsection 3.2.4, I define two new notions: *partial derivatives* and *left unfolded suffixes*, which are both special kinds of normalized expressions.[4] 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 3.2.5, I give a simple algorithm to compute all the syntactic derivatives of a normalized expression. Unsurprisingly, this algorithm makes use of the algorithm described in Subsection 3.2.3.

3.2.1 Computing syntactic derivatives by left unfolding

I show on an example (from [10]) how the direct derivatives of a normalized expression can be computed using simple properties of regular expressions, notably the identity $E^* = 1 + E\,E^*$. Let E be the regular expression $(\mathrm{ab^*a} + \mathrm{ba^*b})^*(1 + \mathrm{ab^*} + \mathrm{ba^*})$.
We can write the following equalities:[5]

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

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 $D_a\,E$ and $D_b\,E$ are computed by the algorithm I am about to describe, in the next subsection. 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 [10]. 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. This last issue prevents me to present[6] the complete computation of the derivatives by this method.

[4](My) partial derivatives are "strongly" related to the partial derivatives of [2] but they are normalized while the latter are not.

[5]Of course, these are not syntactic equalities between regular expressions, but equalities of the languages they denote.

[6]even in an appendix

3.2.2 Syntactic derivatives

We define the notion of syntactic derivative in Figure 3.2. In this definition, x and y stands for letters and w for a chain of letters ($w \in \mathit{Letter}^*$). The expressions used in the left part of equalities (e.g., $E_1 + \ldots + E_n$) are supposed to fulfill the conditions imposed to the same ones in Definition 3.1. 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 some interesting properties that moreover simplifies the correctness proof of the derivation algorithm of Section 3.2.3.

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 be still another normalized expression. It is easy to check that the operation $\otimes$ enjoys the property that

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

(End of definition)

It may seem counter-intuitive to use the operation $\otimes$ instead of $\odot$, sometimes, but this is needed in the definition of Figure 3.2 to ensure that the syntactic derivatives of a normalized expression actually are unions of partial derivatives of the same expression as stated in Theorem 2, which also is needed to prove that normalized expressions have finitely many syntactic derivatives.

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., $\{u \in \mathit{Letter}^* | w.u \in \mathcal{L}\, E\}$, while in the left part, it denotes the syntactic derivative, defined in Figure 3.2.

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 properties of derivatives stated in Section A.2 or, directly, 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)

Figure 3.2: Syntactic derivatives

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

3.2.3 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 3.2.4 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 $o + aE_a + bE_b + ...$ 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 algorithm.

Figure 3.3: 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
        $o \oplus= deriveByUnfold(tabD, E_i, F)$;
     return $o$;

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

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

The top algorithm uses an auxiliary algorithm which is recursive and more general. We denote a call to this auxiliary algorithm by a statement such as

$$o := deriveByUnfold(tabD, E, F)$$

The effect of such a statement is twofold:[7]

- 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 algorithm just has to create a new array *tabD* and to execute:

$$tabD[0] := deriveByUnfold(tabD, E, 1)$$

The auxiliary algorithm is presented in Figure 3.3.

[7] It does "nothing else".

I think that this algorithm is very efficient, which I discuss below. But I first give a correctness proof for it. 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 $\otimes$.

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 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.

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 $D_x\,E_1 \otimes (E_2 \odot F) = (D_x\,E1 \otimes E_2) \otimes F$, 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 $(D_x\,E1 \otimes E_2) \otimes F \;\oplus\; D_x\,E_2 \otimes F = (D_x\,E1 \otimes E_2 \oplus D_x\,E_2) \otimes F = D_x\,E \otimes F$ 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 3.3. It uses a programming technique, first introduced in functional programming (see e.g., [12]), 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 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 3.3 and Figure 3.2: In fact, the top part of Figure 3.2 can be seen (and can be used) as a functional program to compute $D_x\,E$. As already asserted in Subsection 3.2.1, 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 3.2, is not needed for the algorithm *deriveByUnfold* (but it is needed for the correctness proof). To argue further on the efficiency of the algorithm *deriveByUnfold*, we should take into account that of the basic operations $\oplus$ and $\odot$, which is to be done in Section 3.5.5.

3.2.4 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 3.2.3, and, consequently, a set of equations of the form $F = \mathit{tabD}_F$ which gives rise to a DFA for E (and also for its syntactic derivatives). 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 3.2.3.

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 $(\mathrm{size}(E) + 1)/2$.

3.2.4.1 Partial derivatives

The notion of partial derivatives of an expression is defined in Figure 3.4. 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.

Figure 3.4: (Sets of) Partial derivatives

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

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

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

Note that, due to the properties of the operation $\oplus$, it is not necessary to assume that $i \neq j \rightarrow E_i \neq E_j$ in the enumeration $E_1, \ldots, E_n$.

Moreover, none of the partial derivatives $E_1, \ldots, E_n$ are unions (in the sense of Definition 3.1).

Proof

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)

Figure 3.5: Left unfolded suffixes

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

3.2.4.2 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 3.5. It is defined by induction on the syntactic structure of E. Intuitively, it contains all expressions that are taken into account when computing syntactic derivatives of E: For every call $o := \mathit{deriveByUnfold}(\mathit{tabD}, E', F)$ executed when computing derivatives of E (see Figure 3.3), the expression $E' \odot F$ belongs to $\mathit{Lufs}(E)$.

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 $E \neq 0$. The proof uses Lemma 1, which is proven afterwards. Lemma 1 says that $1 \in \mathit{Lufs}(E)$. Thus, it is equivalent 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 equal to $E_1 + \ldots + E_n$,

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

(End of proof)

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

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

Proof

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)

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

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

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

Proof

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)

Lemma 3 *Let E_1 and E_2 be normalized regular expressions. Then,*

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

Proof

The proof is straightforward if E_1 or E_2 are equal to 0 or 1. It is also immediate if E_1 is not a concatenation, by definition of *Lufs* (see Figure 3.5). If E_1 is a concatenation, say $F_1 \,.\, F_2$, one has:

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

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

(End of proof)

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

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

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 $\mathit{Lufs}\,(E)$.
2. if E is a union $E_1 + \ldots + E_n$ $(n \geq 2)$, it is true that $F \in \mathit{Lufs}\,(E_i)$ for at least one i $(1 \leq i \leq n)$. By induction hypothesis, $\mathit{Lufs}\,(F) \subseteq \mathit{Lufs}\,(E_i) \subseteq \mathit{Lufs}\,(E)$.
3. If E is a concatenation $E_1.E_2$, the proof is similar as above if $F \in \mathit{Lufs}\,(E_2)$. Otherwise, we have that $F \in \mathit{Lufs}\,(E_1) \odot E_2$. Thus, there exists a normalized expression $F' \in \mathit{Lufs}\,(E_1)$ such that $F = F' \odot E_2$. By induction hypothesis, we can write that $\mathit{Lufs}\,(F') \subseteq \mathit{Lufs}\,(E_1)$. By Lemma 3, we thus have:
$$\begin{array}{rcl} \mathit{Lufs}\,(F) & = & \mathit{Lufs}\,(F' \odot E_2) \\ & = & \mathit{Lufs}\,(F') \odot E_2 \cup \mathit{Lufs}\,(E_2) \\ & \subseteq & \mathit{Lufs}\,(E_1) \odot E_2 \cup \mathit{Lufs}\,(E_2) \\ & = & \mathit{Lufs}\,(E) \end{array}$$
4. If E is an iteration, the proof is similar to the case $F \in \mathit{Lufs}\,(E_1) \odot E_2$ in the preceding one.

(End of proof)

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

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

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 $\mathcal{D}_w\,E = \{E_1, \ldots, E_n\} \subseteq (\mathit{Lufs}\,(E_1) \cup \ldots \cup \mathit{Lufs}\,(E_n)) \subseteq \mathit{Lufs}\,(E)$, by induction on the structure of E.

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

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

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

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

(End of proof)

Corollary 1 *The set of all partial derivatives of a normalized expression E is finite. In fact, their number is less or equal to $(\mathit{size}(E) + 1)/2$.*

Proof

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

(End of proof)

3.2.5 An algorithm to compute all syntactic derivatives

An algorithm that computes all syntactic derivatives of a normalized expression E is presented in Figure 3.6. It also builds the set of all equations $F = \mathit{tabD}_F$ where F is a syntactic derivative of E and tabD_F is an array of expressions such that $\mathit{tabD}\,[0] = 1$ if $1 \in \mathcal{L}F$ and 0, otherwise, and $\mathit{tabD}\,[x] = D_x\,F$ for every letter x. This algorithm is called *computeDerivatives*. It makes use of the (top) algorithm *computeDirectDerivatives*, described at Section 3.2.3.

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 $\text{eq}(F, \mathit{tabD})$. 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.

Now we are ready to give a correctness proof of the algorithm *computeDerivatives*.[8]

Correctness proof of the algorithm *computeDerivatives*

[8]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.

Figure 3.6: The algorithm that computes all syntactic derivatives (*computeDerivatives*).

```
$S_{tod}$ := $\{E\}$; $SEq$ := $\{\}$; $S_{dev}$ := $\{\}$;
while $S_{tod} \neq \{\}$, do
    $F \Longleftarrow S_{tod}$ ; $S_{dev} \Longleftarrow F$ ;
    $tabD$ := $computeDirectDerivatives(F)$ ;
    $SEq \Longleftarrow$ eq($F$, $tabD$) ;
    for every letter $x$, do
        if $tabD[x] \notin (S_{tod} \cup S_{dev})$, do
            $S_{tod} \Longleftarrow tabD[x]$ ;
```

The correctness proof uses the following loop invariant:[9]

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

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

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 $D_w E$ ($w \in \mathit{Letter}^*$).[10]

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 3.2.4.2.

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

(End of correctness proof)

3.3 A background for reasoning about normalized expressions

Now we arrive at the point where the core aspects of this work are about to be explained precisely and completely. Although our main goal is the simplification of regular expressions, this cannot be tackled before defining and creating an environment of "things" that provides useful information about normalized expressions and the regular languages they denote. I call this environment, *the background.* All algorithms that are to be developped from now on are working in the context of a (current) background. I first describe what the background contains in Section 3.3.1. Then, I explain operations that can be used to extend and refine the background in Section 3.3.2.

3.3.1 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 subexpressions belong to the background as well.

[10]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.

- 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. $\mathcal{L}(E_1) = \mathcal{L}(E_2)$). 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. This function 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 $E = \mathit{tabD}$ 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 3.2.3). 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 sets of expressions (obviously) nor syntactic derivatives of E in the sense of Section 3.2, strictly speaking, in general.

  It is *not* required that an equation exists for every equivalence class. But it is required that the set of equations of the background is *complete*, i.e., all expressions E_x occurring in the right part of an equation must 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 $E_1 = \mathit{tabD}_1$ and $E_2 = \mathit{tabD}_2$ are such that $E_1 = E_2$ or $\mathit{tabD}_1 = \mathit{tabD}_2$. We say that such equations *overlap.*[11] If it is the case, it means that at least two equivalence classes contains expressions that denote the same regular language. Thus, the background could 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 the expressions of which denote the same regular language.

[11]They must be distinct.

3.3.2 Operations on the background

3.3.2.1 Adding new expressions

The set of expressions of the background normally can 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.

3.3.2.2 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 to 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 $E = \mathit{tabD}$ present in the background. Thus, E is replaced by E_1 if $E = E_2$, initially. Also, for every letter x, $\mathit{tabD}[x]$ becomes equal to E_1 if $\mathit{tabD}[x] = E_2$, 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 $O(\mathit{nl} \times \text{nocc}(E_2))$ where nl is the number of letters and nocc(E_2) is the number of occurrences of E_2 in all equations (see Chapter 3.5.5). This is quite efficient if nl is small.

3.3.2.3 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. An algorithm for reduce(S) is depicted in Figure 3.7. 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 size$(E_1) \leq$ 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 below.

Figure 3.7: Reducing the background (reduce(S))

```
while $S \neq \{\}$ or at least two equations overlap, do
    while $S \neq \{\}$, do
        $\langle F_1, F_2 \rangle \Longleftarrow S$;
        $\langle E_1, E_2 \rangle :=$ trim$(F_1, F_2)$;
        if $E_1 \neq E_2$, do
            merge$(E_1, E_2)$;
    if at least two equations overlap, do
        let $E_i = tabD_i$ $(i = 1, 2)$ be such equations;
        if $E_1 \neq E_2$, do
            $S \Longleftarrow \langle E_1, E_2 \rangle$;
        elsefor every letter $x$, do
            if $tabD_1[x] \neq tabD_2[x]$, do
                $S \Longleftarrow \langle tabD_1[x], tabD_2[x] \rangle$;
```

Termination proof of the algorithm 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.

(End of termination proof)

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. See Chapter 3.5.5

3.3.2.4 Merging equivalence classes denoting the same language

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 both classes are associated to an equation (and thus to a DFA, since the set of equation of the background is complete).

Let *SEq* be a complete subset of the equations in the background (possibly all of them). The set *SEq* can be seen as the set of states of a DFA to which Moore's algorithm [15] can be applied in order to build the equivalence classes of the states that accept the same language (see also [1]). We can apply the algorithm merge to all pairs $\langle E_1, E_2 \rangle$ of distinct expressions belonging to some of these equivalence classes. Afterwards, we can execute reduce($\{\}$) to obtain a reduced background in which all equivalence classes corresponding to equations $E_i = \mathit{tabD}_i$ $(i = 1, 2)$ in *SEq* have been merged whenever $\mathcal{L}(E_1) = \mathcal{L}(E_2)$.[12]

I have developped my own version of such an algorithm. 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. The first version of the operation minimize() is more powerful than the latter but it can be too costly to use it repeatedly (in a simplification algorithm). So, later on, we often prefer to use the second version.

3.3.3 Operations taking advantage of the background

3.3.3.1 Computation of derivatives

Although the algorithms described in Section 3.2 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 happens when the representative of an expression to derive is the left part of an equation in the background.

From now on, we assume that the background provides an operation hasDFA(E) that returns *true* if the background contains an equation of the form $E = \mathit{tabD}$, and *false*, otherwise. If it returns *true*, the background contains a complete set of equations equivalent to a DFA for the expression E and, in many case, this set of equations is more compact that the set of equations that should be computed by the algorithm *computeDerivatives* of Section 3.2.

Note however that the algorithm *computeDerivatives* is the primary provider of equations to the background. As can be shown in Figure 3.6, 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. We assume that the operation addEq(E, tabD) add to the background the equation $E' = \mathit{tabD}'$, where $E' = \mathrm{rep}(E)$ and $\mathit{tabD}'[x] = \mathrm{rep}(\mathit{tabD}[x])$ for every letter x. (Also, $\mathit{tabD}'[0] = \mathit{tabD}[0]$.) It is assumed that no such equation belongs to the background beforehand.

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

Figure 3.8: Adding equations to the background to get a DFA for E.

```
$S_{tod}$ := $\{E\}$; $SEq$ := $\{\}$; $S_{dev}$ := $\{\}$;
while $S_{tod} \neq \{\}$, do
    $F \Longleftarrow S_{tod}$ ; $S_{dev} \Longleftarrow F$ ;
    if $\neg$hasDFA(rep($F$)), do
        $tabD$ := $computeDirectDerivatives(F)$ ;
        $SEq \Longleftarrow$ eq($F$, $tabD$) ;
        for every letter $x$, do
            if $tabD[x] \notin (S_{tod} \cup S_{dev})$, do
                $S_{tod} \Longleftarrow tabD[x]$ ;
while $SEq \neq \{\}$, do
    eq($F$, $tabD$) $\Longleftarrow SEq$;
    addEq($F$, $tabD$);
```

A modified (optimized) version of the algorithm *computeDerivatives* is presented in Figure 3.8. 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 $\{F \mid F \in S_{dev}$ & $\neg$hasDFA(rep(F))$\}$.
- The second while loop add to the background a set of equations that fulfill the conditions imposed to equations in Definition 5 except completeness.
- After execution of the second while loop the set all equations in the background is complete 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 they 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 distinct representatives in the background. Therefore, it is worthwhile to add a call reduce($\{\}$), as a final statement to the algorithm. I have omitted it, in Figure 3.8, to simplify the discussion, but, from now on, we always use the complete version, which ensures that the final background is reduced.

3.3.3.2 Deciding equivalence of two normalized expressions

An algorithm to decide equivalence of two normalized expressions is presented in Figure 3.9. It takes advantage of the information provided by the background, as explained hereunder.

1. If the two expressions have the same representative, they denote the same language. Hence, the first if statement.
2. Otherwise, the algorithm computes a complete set of equations for both E_1 and E_2 (in the case they are not in the background beforehand).
3. If $\mathcal{L}(E_1) = \mathcal{L}(E_2)$, it may be the case that the representatives of E_1 and E_2 are now equal, being the left part of the same equation in the background. Hence, the second if statement.
4. If the representatives are not the same, it may nevertheless be the case that $\mathcal{L}(E_1) = \mathcal{L}(E_2)$. To decide it, the call minimize($\{\mathrm{rep}(E_1), \mathrm{rep}(E_2)\}$) is executed.
5. After this call, one has $\mathcal{L}(E_1) = \mathcal{L}(E_2)$ if and only if $\mathrm{rep}(E_1) = \mathrm{rep}(E_2)$. Hence, the last statement.

Figure 3.9: Deciding equivalence of two normalized expressions (eq(E_1, E_2))

```
if $\mathrm{rep}(E_1) = \mathrm{rep}(E_2)$, do
    return *true*;

*computeDerivatives*$(E_1)$;
*computeDerivatives*$(E_2)$;

if $\mathrm{rep}(E_1) = \mathrm{rep}(E_2)$, do
    return *true*;

minimize($\{\mathrm{rep}(E_1), \mathrm{rep}(E_2)\}$);

return $\mathrm{rep}(E_1) = \mathrm{rep}(E_2)$ ;
```

Deciding equivalence of two normalized expressions may be costly, particularly when they are not equivalent. However, even in this case, executing the call eq(E_1, E_2) improves the information in the background, which can benefit to other algorithms executed later on.

3.3.3.3 Checking whether $E_1 \leq E_2$

Remember that the relation $\leq$ between regular expressions is defined by

$$E_1 \leq E_2 \quad \longleftrightarrow \quad \text{the expressions } E_1 \oplus E_2 \text{ and } E_2 \text{ denote the same language,}$$

or, alternatively, by

$$E_1 \leq E_2 \quad \longleftrightarrow \quad \mathcal{L}(E_1) \subseteq \mathcal{L}(E_2).$$

The most straightforward way to check this relation is to execute eq($E_1 \oplus E_2, E_2$), which we abbreviate by infEq(E_1, E_2).

3.4 Solving equations

A main idea of the work presented in this report, is that simplifying an expression E amounts to find (in the background) a simpler expression to which it is equivalent. Thus, the more expressions in the background, the best chance we have to get a good simplification. In this section, I propose an algorithm, or, more exactly, a family of algorithms that allow us to add "hopefully interesting" equations in the background. As a motivating example, consider the expression E = (ab*a + ba*b)*(1 + ab* + ba*), discussed in Section 3.2.1. It has two direct syntactic derivatives A = a* + a*b E and B = b* + b*a E (see Section 3.2.1). Computing all syntactic derivatives gives rise to three equations only:

$$E \;=\; 1 + a\,B + b\,A \qquad A \;=\; 1 + a\,A + b\,E \qquad B \;=\; 1 + a\,E + b\,B$$

Minimizing the equations, by executing minimize($\{E\}$), reduces the set of equations to only one: $E \;=\; 1 + a\,E + b\,E$. However, the shortest expression of the equivalence class of E remains E itself, since it is shorter than A and B. A better result is obtained by applying Salomaa's rule (see Section 2.2.3) to the modified equation $E \;=\; 1 + (a + b)\,E$. We get that E is equivalent to (a + b)*. Notice that the same result can be obtained more simply, by executing minimize(), in the case where the expression F = (a + b)* is already present in the background with the equation $F \;=\; 1 + a\,F + b\,F$.

Let E be a normalized expression that is the left part of an equation of the background. This expression determines a complete set of equations, which can be seen as a DFA for the regular language $\mathcal{L}(E)$. The left part of these equations "are" the derivatives of E, in the sense made precise in Definition 5. So, below, I call them *the derivatives* of E. I am going to present (several variants of) an algorithm to "solve" these equations in order to produce another expressions E', equivalent to E, and possibly simpler.

Algorithms to solve equations involving regular expressions have been proposed before (see, e.g., [1]). The goal of such algorithms is a bit different from mine, however.

- They use *variables* standing for unknown languages, to be determined by the algorithm. In our case, the equations involve normalized expressions, not variables. That is: The equations already are solved at the start. Nevertheless, my algorithm (normally) does not make use of the values of the expressions, it only use the relations between the expressions defined by the equations. Thus, in fact, it uses the expressions as if they were variables.
- We are not interested to solve all equations but only to find a solution for the first one, of which E is the left part, hoping that it would be simpler that E itself.

3.4.1 The basic algorithm

The algorithm makes use of an auxiliary (and more general) algorithm *solveAux*, which takes two arguments:

1. A sequence of distinct expressions $S = \langle E_1, \ldots, E_m \rangle$ $(m \geq 0)$;
2. An expression F.

It is required that all expressions $E_1, \ldots, E_m$, and F are derivatives of E.
The auxiliary algorithm returns a sequence of expressions $R = \langle A_0, A_1, \ldots, A_m \rangle$ such that

$$F \;=\; A_0 + A_1.E_1 + \ldots + A_m.E_m.$$

It must be made clear that the equation $F \;=\; A_0 + A_1.E_1 + \ldots + A_m.E_m.$ is not a syntactic equality between two regular expressions. It has to be interpreted as an equality between their denoted languages, i.e.,

$$\mathcal{L}(F) \;=\; \mathcal{L}(A_0) \;\cup\; \mathcal{L}(A_1).\mathcal{L}(E_1) \;\cup \ldots \cup\; \mathcal{L}(A_m).\mathcal{L}(E_m)$$

Since the correct notation is heavy, the simplified one is always used in the rest of this section.

Obviously, the top algorithm only has to execute $R := \mathit{solveAux}(\langle\rangle, E)$ to get a solution, as first and only element of R. The auxiliary algorithm works as follows.

1. If F is syntactically equal to one of the expressions $E_1, \ldots, E_m$, let i such that $F = E_i$. The algorithm returns the sequence $\langle A_0, A_1, \ldots, A_m \rangle$ such that $A_0 = 0$, $A_j = 0$ if $j \neq i$, and $A_i = 1$.
2. Otherwise, the background contains an equation $F \;=\; o_F + \ldots + x\, F_x + \ldots$. For all letters x such that $F_x \neq 0$,[13] the algorithm recursively executes $R_x := \mathit{solveAux}(S', F_x)$, where $S' = \langle E_1, \ldots, E_m, F \rangle$. By the specification of *solveAux*, for every letter x, we can write: $F_x = A_0^x + A_1^x.E_1 + \ldots + A_m^x.E_m + A_{m+1}^x.F$ where $R_x = \langle A_0^x, A_1^x, \ldots, A_m^x, A_{m+1}^x \rangle$. Let $B_i \;=\; \ldots + x\,.A_i^x + \ldots$ $(1 \leq i \leq m+1)$. Replacing F_x by the right part of the equation above, and performing some calculations, we get:

$$F = o_F + B_1.E_1 + \ldots + B_m.E_m + B_{m+1}.F$$

[13] This is not really needed but it is more "efficient".

The definition of B_{m+1} clearly shows that $1 \not\leq B_{m+1}$. Therefore, we are allowed to apply Salomaa's rule to the equation above, which gives:

$$\begin{array}{rcl} F & = & B^*_{m+1}.(o_F + B_1.E_1 + \ldots + B_m.E_m) \\ & = & (B^*_{m+1}.o_F) + (B^*_{m+1}.B_1).E_1 + \ldots + (B^*_{m+1}.B_m).E_m \end{array}$$

It is thus correct to return

$$R = \langle A_0, A_1, \ldots, A_m \rangle$$

where $A_0 = B^*_{m+1}.o_F$ and $A_i = B^*_{m+1}.B_i$ $(1 \leq i \leq m)$.

The above reasoning is not completely satisfactory, from a logical point of view, because some equalities, such as $F_x = A^x_0 + A^x_1.E_1 + \ldots + A^x_m.E_m + A^x_{m+1}.F$ are equalities of the languages denoted by the expressions, not syntactic equalities, while other equalities such as $B_i = \ldots + x\,.A^x_i + \ldots$ or $A_i = B^*_{m+1}.B_i$ must be treated as syntactic equalities to produce the result of the algorithm as a syntactic object. Since we work with normalized regular expressions, we must, in the case of syntactic equalities, use the operators $\oplus$, $\odot$, and $\star$ instead of $+$, $.$, and $*$. A cleaner version of the algorithm is thus given in Figure 3.10. I leave it to the reader to prove the correctness of this version by adapting the previous reasoning.

Figure 3.10: Solving equations ($solveAux(S, F)$)

```
1. Let $S = \langle E_1, \ldots, E_m \rangle$.

2. If $\exists i : 1 \leq i \leq m : F = E_i$, let $i$ such that $1 \leq i \leq m$ and $F = E_i$.
   return $\langle A_0, A_1, \ldots, A_m \rangle$ such that $A_0 = 0$, $A_j = 0$ if $j \neq i$, and $A_i = 1$.

3. Let $S' = \langle E_1, \ldots, E_m, F \rangle$.

4. Retrieve the equation $F = tabD$, from the background.

5. For all letters $x$ such that $tabD[x] \neq 0$,do
      $R_x := solveAux(S', tabD[x])$;
      Let $R_x = \langle A^x_0, A^x_1, \ldots, A^x_m, A^x_{m+1} \rangle$;

6. For $i = 1$ to $m + 1$, do
      $B_i := 0$;
      For all letters $x$ such that $tabD[x] \neq 0$,do
         $B_i \oplus= x \odot A^x_i$;

7. $C := B_{m+1}\star$;

8. return $R = \langle A_0, A_1, \ldots, A_m \rangle$ where
      $A_0 = C \odot tabD[0]$ and $A_i = C \odot B_i$ $(1 \leq i \leq m)$.
```

A few additional remarks are helpful to understand the algorithm. Basically, the algorithm recursively computes the new expression, equivalent to E, from new values related to its derivatives. The termination of the algorithm is ensured by the fact that all expressions in $S = \langle E_1, \ldots, E_m \rangle$ are distinct and are derivatives of E. Thus, the size of S is bounded by the number of derivatives. There is a special case when F belongs to S. In that case, Salomaa's rule is used by the algorithm when it returns inside the enclosing call $solveAux(\langle E_1, \ldots, E_{i-1}\rangle, F)$ $(i < m)$. Formally, in Figure 3.10, the rule systematically is applied but in the case where $B_{m+1} \in \{0, 1\}$, one has $C = B_{m+1}\star = 1$, so that its application is not really needed and can be avoided as an "optimization".

Example 1. Solving equations related to $E =$ c* + c*a(c*a + b)*c*

Computing the syntactic derivatives of E gives only two derivatives, and thus only two equations:

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

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

Let us see how the algorithm works for E. The initial call to *solveAux* is $solveAux(\langle\rangle, E)$. Since the equation for E is $E = 1 + \mathrm{a}\,F + \mathrm{c}\,E$, two recursive calls are executed:

- $solveAux(\langle E\rangle, F)$
  The equation for F is $F = 1 + \mathrm{a}\,F + \mathrm{b}\,F + \mathrm{c}E$. Only two recursive calls are needed.

  - $solveAux(\langle E, F\rangle, F)$
    Since F belongs to the list $\langle E, F\rangle$, the list $\langle 0, 0, 1\rangle$ is returned.
  - $solveAux(\langle E, F\rangle, E)$
    Similarly, the list $\langle 0, 1, 0\rangle$ is returned.

  Combining the results, we get $F = 1 + \mathrm{E} + (\mathrm{a + b})F$. Applying Salomaa's rule, we have $F = \mathrm{(a + b)^*(1 + E)}$. Thus, the list $\langle \mathrm{(a + b)^*, (a + b)^*}\rangle$ is returned.

- $solveAux(\langle E\rangle, E)$
  Since E belongs to the list $\langle E\rangle$, the list $\langle 0, 1\rangle$ is returned.

Now, we are back to the top call. Combining the results of the recursive calls, we get: $E = \mathrm{1 + a(a + b)^* + (a(a + b)^* + c)}E$, which gives $E = \mathrm{(a(a + b)^* + c)^*(1 + a(a + b)^*)}$, by Salomaa's rule.

This result can be simplified a follows:

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

We see that solving equations does not always produce simplest expressions. Simplification rules can help to improve the results. Nevertheless, I am about to present an "improved version" of the algorithm, which directly computes the best result, for this example, at least. (End of example)

3.4.2 Improving the basic algorithm

The recursive calls of the algorithm *solveAux* defines a tree and, in general, the size of the expression returned by the top algorithm is more than proportional to the size of this tree because the expression $B_{m+1}\star$ is distributed to the left of all expressions B_i $(0 \le i \le m)$ at the end of each recursive call. It is sometimes possible to reduce the size of this tree by "subtracting" some of the expressions E_i in S from F. The "subtraction" takes place at the level of the (direct) derivatives of E_i and F. So, the number of (direct) recursive calls executed during the call $solveAux(S, F)$ can be reduced. In some cases, no recursive calls are needed at all.

To implement this idea, we use two auxiliary operations: $\mathrm{infEqD}(tabD_1, tabD_2)$ and $\mathrm{subtract}(tabD_1, tabD_2)$, where $tabD_1$ and $tabD_2$ are arrays of expressions (in practice, the right parts of two equations $E_i = tabD_i$ from the background $(i = 1, 2)$).

The first operation returns *true* if for, every x that is a letter or equal to 0, one has $tabD_1[x] = 0$ or $tabD_1[x] = tabD_2[x]$; it returns *false*, sinon. The second operation returns a new array of expressions $tabD$ such that $tabD[x] = 0$ if $tabD_1[x] = tabD_2[x]$ and $tabD[x] = tabD_2[x]$, otherwise. When the first operation returns *true*, it is the case that $\mathcal{L}(E_1) \subseteq \mathcal{L}(E_2)$, but there is more: all derivatives of E_1 that are different from 0, also are derivatives of E_2. In that case, we can use the operation $\mathrm{subtract}(tabD_1, tabD_2)$ to get an array containing only the other derivatives of E_2 (as elements different from 0). We can use those two operations to avoid some recursive calls in the algorithm $solveAux(S, F)$. A variant (an "improvement") of $solveAux(S, F)$ is shown in Figure 3.11. Since most of the work is done at the level of the arrays of derivatives, the list $S = \langle E_1, \ldots, E_m\rangle$ is replaced by $S = \langle tabD_1, \ldots, tabD_m\rangle$ where all equations $E_i = tabD_i$ $(1 \le i \le m)$ belongs to the background. This simplifies the algorithm a bit.

This new version of the algorithm is one possibility among many others. The best way to proceed remains an open problem to me. An other interesting open problem would be to decide whether a shortest expression can be computed by a suitable version of the algorithm. Instead of formally proving the correctness of the algorithm in Figure 3.11, I propose to show how it works on the expression from Example 1.

Figure 3.11: Improving the algorithm $solveAux(S, F)$

```
1. Let $S = \langle tabD_1, \ldots, tabD_m \rangle$.

2. Retrieve the equation $F = tabD_F$ from the background.

3. $i := 1$; found := $false$;
   while $(i \neq m + 1)$ and $\neg$ found, do
       if infEqD($tabD_i, tabD_F$), do
          found := $true$ ;
       else, do
          $i := i + 1$ ;

4. if $i \neq m + 1$, do
       $tabD$ := subtract($tabD_i, tabD_F$);
   else, do
       $tabD := tabD_F$;

5. Let $S' = \langle E_1, \ldots, E_m, tabD_F \rangle$.

6. For all letters $x$ such that $tabD[x] \neq 0$, do
       $R_x := solveAux(S', tabD[x])$;
       Let $R_x = \langle A_0^x, A_1^x, \ldots, A_m^x, A_{m+1}^x \rangle$;

7. For $j := 1$ to $m + 1$, do
       $B_j := 0$;
       For all letters $x$ such that $tabD[x] \neq 0$, do
          $B_j \oplus= x \odot A_j^x$;


8. if $i \neq (m + 1)$, do
          $B_i := B_i + 1$;

9. $C := B_{m+1}\star$;

10. return $R = \langle A_0, A_1, \ldots, A_m \rangle$ where
       $A_0 = C \odot tabD[0]$ and $A_i = C \odot B_i$ $(1 \leq i \leq m)$.
```

Example 2. Another way to solve equations related to $E = \mathbf{c^* + c^*a(c^*a + b)^*c^*}$

Remember that the expression E determines only two equations:

$$E \;=\; 1 + \mathrm{a}\,F + \mathrm{c}\,E \;\; (= \mathit{tabD}_E) \qquad\qquad F \;=\; 1 + \mathrm{a}\,F + \mathrm{b}\,F + \mathrm{c}E \;\; (= \mathit{tabD}_F)$$

The improved algorithm works as follows, for E. The initial call to *solveAux* is $\mathit{solveAux}(\langle\rangle, E)$. Since the equation for E is $E \;=\; 1 + \mathrm{a}\,F + \mathrm{c}\,E$, two recursive calls are executed:

- $\mathit{solveAux}(\langle \mathit{tabD}_E\rangle, F)$
  The equation for F is $F \;=\; 1 + \mathrm{a}\,F + \mathrm{b}\,F + \mathrm{c}E$. We see that $\mathrm{infEqD}(\mathit{tabD}_E, \mathit{tabD}_F)$ holds. Thus, step 3 gives the value 1 to i. Afterwards, step 4 creates an array of derivatives *tabD* such that $\mathit{tabD}[b] = F$ and $\mathit{tabD}[x] = 0$ if $x \neq b$. Therefore, only one recursive call is executed in step 6:

  - $\mathit{solveAux}(\langle \mathit{tabD}_E, \mathit{tabD}_F\rangle, F)$
    Since tabD_F belongs to the list $\langle \mathit{tabD}_E, \mathit{tabD}_F\rangle$, the list $\langle 0, 0, 1\rangle$ is returned.

  In step 7, we get $B_0 = 0$, $B_1 = 0$, and $B_2 = b$. In step 8, B_1 is set to 1. Moreover, C is set to b* in step 9. Finally, $\langle 0, \mathrm{b}^*\rangle$ is returned.

- $\mathit{solveAux}(\langle \mathit{tabD}_E\rangle, E)$
  Since tabD_E belongs to the list $\langle \mathit{tabD}_E\rangle$, the list $\langle 0, 1\rangle$ is returned.

Now, we are back to the top call. Combining the results of the recursive calls, we get: $B_1 = \mathrm{ab}^* + \mathrm{c}$, and thus, $C = (\mathrm{ab}^* + \mathrm{c})^*$. The final result therefore is:

$$A_0 = C \odot \mathit{tabD}_E[0] = (\mathrm{ab}^* + \mathrm{c})^* \odot 1 = (\mathrm{ab}^* + \mathrm{c})^*$$

This time the simplest result is directly obtained. No further simplification is necessary.

(End of example)

3.5 Local simplification

In this section, I present algorithms that apply simplification rules to normalized regular expressions. The simplification rules are adapted or inspired by the work presented in [16]. They are better used in the context of a global simplification algorithm that is presented later on, i.e, they are more useful and less costly when they are applied to expressions the sub-expressions of which already are simplified.

This section is organised as follows. I first question the usefulness of local simplification algorithms in the context of a global algorithm using the methods previously presented: computation of derivatives, reduction of equations in the background, minimization, and solving of equations. This is necessary because, on the one hand, these methods alone may already provide substantial simplifications and because, on the other hand, they are used as sub-algorithms by the local simplification algorithms. Therefore, there is a risk of redundancy if we add local simplification rules to the global simplification process. This is discussed in Subsection 3.5.1. Afterwards, in Subsection 3.5.2, I explain some important differences between the work presented in [16] and mine. Finally, I describe the local simplification algorithms in three subsections dedicated to unions, concatenations, and iterations, respectively.

3.5.1 Is local simplification really needed, and when?

As an example, let us first consider the normalized expression

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

(See Appendix A.1.) As explained in details in the appendix, it is simplified to $\mathrm{a}^*((\mathrm{a} + \mathrm{b})\mathrm{a}^*)^*$ just by computing its derivatives and reducing the background. Additionally, solving the unique equation resulting from this process, it is simplified to $(\mathrm{a} + \mathrm{b})^*$, i.e., as much as possible. It means that no new algorithms are needed for this example. The methods presented in the previous sections are powerful enough, and it is so for many other examples. But there are cases where additional simplifications give still better results.

As another example, consider the expression

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

Computing the derivatives and reducing the background provides no simplification at all; although, clearly, the expression can be simplified to $\mathrm{b} + \mathrm{a}^*$. If we solve the resulting equations (after minimization or not), we may get:

$$1 + \mathrm{aa}^* + \mathrm{b},$$

which is not completely satisfactory. In this case, solving equations is not the best approach. A better approach consists of removing sub-expressions that are subsumed by others. In this case, 1, a, and aa are subsumed by a^*.

Unfortunately (and predictably), removing subsumed sub-expressions from a union is not always powerful enough, as shown in the next, slightly different example:

$$1 + \mathrm{a} + \mathrm{aa} + \mathrm{b} + \mathrm{aaaa}^*.$$

Here, no sub-expression is subsumed by another. A possible way to get the best simplification is to group the four sub-expressions 1, a, aa, and aaaa^*: computing the derivatives of $1 + \mathrm{a} + \mathrm{aa} + \mathrm{aaaa}^*$ and reducing the background gives the expression a^*, which can be combined with b to provide the result.

Now, let us consider this expression:

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

It cannot be simplified by any previously suggested method. A way that works consists of computing a "minimal covering" of the expression: the "sub-union" a* + b(ba*)* denotes the same regular language, a fact that can be checked by the operation eq of Figure 3.9.

The few examples discussed above suggest that the simplification algorithms I am about to present largely are based on the operations eq and infEq of Subsection 3.3.3.2. But these operations themselves essentially consist of computing derivatives, reducing the background, and minimizing sets of equations. So we see that, in any case, these methods remain at the heart of the simplification process (in my approach).

3.5.2 Main differences with Stoughton's work

- In [16], Alley Stoughton proposes *reduction rules* that apply to so-called *weakly simplified* regular expressions. Weakly simplified expressions are very similar to the notion of normalized expressions used in this report. The main difference is that the total ordering used to sort sub-expressions, in a union $E_1 + \ldots + E_n$, is uniquely defined, once for all, by induction on the structure of expressions. This may seem nicer but it is less efficient than my choice of sorting sub-expressions on the integer values of their identifiers.

- The reduction rules of [16] apply to binary (or unary) expressions. Therefore, a weakly simplified expression must be transformed by so called *structural rules* that perform permutations of sub-expressions in unions and add parentheses in unions and concatenations, in all possible ways. In my approach, only one single expression is considered. This complicates the writing of some algorithms but is more efficient again, in general.

- The so-called *local* algorithm of [16] builds a strictly decreasing, necessarily finite, sequence of weakly simplified expressions. This is a top-down approach, which makes it too costly to consider large expressions. In my approach, sub-expressions are simplified first, which, in many cases, leads to a pre-simplified expression much shorter than the initial one, *before reductions rules are applied.* Therefore, larger expressions can be handled.

- In [16], the ordering used to compare expressions is very elaborate and it captures a notion of simplicity appealing to the author. I use the straightforward notion of size (number of symbols) to compare expressions but it would be possible to use Stoughton's measure, albeit at a cost.

- In my approach, more than one currently most simplified expression is considered at each stage of the process. The background keeps many regular expressions distributed in equivalence classes. It is often the case that an expression is *indirectly* simplified by creating another equivalent expression not shorter but belonging to another equivalence class, the representative of which is shorter.

- The reduction rules in [16] use preconditions involving the inclusion of (the language denoted by) a sub-expression into another. However, the author does not use an exact algorithm to check inclusion, arguing that it would be too costly. She proposes a more efficient, approached algorithm. In my case, I use the exact algorithm of Section 3.3.3.3. This appears to be reasonable because the equations present in the background often allow us to check inclusion efficiently. Moreover, when the check requires more work, it adds new information into the background. Thus, this extra work is not lost for the future, contrary to the work done to (conservatively) check inclusion in [16].

Figure 3.12: Simplifying binary union (sunion(E_1, E_2))

```
if rep($E_1$) = rep($E_1 \oplus E_2$), do
    return rep($E_1$);
if rep($E_2$) = rep($E_1 \oplus E_2$), do
    return rep($E_2$);

computeDerivatives($E_1$);
computeDerivatives($E_2$);
computeDerivatives($E_1 \oplus E_2$);

if rep($E_1$) = rep($E_1 \oplus E_2$), do
    return rep($E_1$);
if rep($E_2$) = rep($E_1 \oplus E_2$), do
    return rep($E_2$);

minimize({rep($E_1$), rep($E_2$), rep($E_1 \oplus E_2$)});

return rep($E_1 \oplus E_2$) ;
```

3.5.3 Simplifying a union

Recall that a union is a normalized expression E of the form $E_1 + \ldots + E_n$ ($n \geq 2$) where none of the expressions $E_1, \ldots, E_n$ are unions. I propose two algorithms to simplify unions. They are complementary and can also be combined into a single and more powerful one. Minimizing a union is very difficult in general, so the algorithms presented here can be viewed as "guidelines" for the design of more ambitious algorithms using "expertise" and methods related to "artificial intelligence".

3.5.3.1 Removing subsumed sub-expressions

The first algorithm I am about to present basically removes from the union E (see above) some expressions E_i that are subsumed by (possibly the union of) some others. The principle

Figure 3.13: An algorithm to simplify an union ($\mathit{simplifyUnion}(E)$)

```
Let $E = E_1 + \ldots + E_n$ $(n \geq 2)$;
$F := E_1$;
for $i := 2$ to $n$, do
    let $F = F_1 + \ldots + F_m$ $(m \geq 1)$;
    $G := E_i$;
    for $j := 1$ to $m$, do
        $G :=$ sunion$(F_j, G)$;
    $F := G$;
reduce$(\{\langle F, E \rangle\})$;
```

of the algorithm is as follows: We consider each expression E_i, in turn. If E_i is subsumed by $E_1 + \ldots + E_{i-1}$, we remove it from E, otherwise, we compare E_i with each expression E_j already seen (and not removed); if E_j is subsumed by E_i, we remove it from E, as well. This method ensures that no remaining expression is subsumed by another but it is nevertheless possible that some remaining expression E_j is subsumed by the union of some other expressions. This can be overcome by a more costly algorithm but I do not elaborate on this now. Instead, I propose a different improvement to the previous method. I introduce a new operation called sunion(E_1, E_2) that checks if one of the two expressions E_1, E_2 subsumes the other but also does a little bit more: when possible, it computes a simplified version of $E_1 \oplus E_2$. Remember that our method to decide whether one of the two expressions E_1, E_2 subsumes the other, consists of computing, reducing, and possibly minimizing the derivatives of $E_1 \oplus E_2$. In every case where a derivative of $E_1 \oplus E_2$ is equivalent to this expression and has a representative shorter than $E_1 \oplus E_2$, a simplified version is obtained. The algorithm of sunion is presented in Figure 3.12.

The algorithm to simplify a union is depicted in Figure 3.13. This algorithm simplifies the expression E in all cases where at least one expression E_j is subsumed by an expression of the form $E_1 + \ldots + E_{j-1} + E_i$, with $1 \leq j < i \leq n$, or when an expression E_i is subsumed by $E_1 + \ldots + E_{i-1}$. In particular, an expression shorter than E is produced (and unified to E) whenever any sub-expression E_i is subsumed by another one. However, the shorter expression not necessarily is of the form $E_{i_1} + \ldots + E_{i_m}$ with $1 \leq i_1 < \ldots < i_m \leq n$ because some additional simplifications may replace several sub-expressions by a single, shorter, one (see Subsection 3.5.1, for some examples).

Figure 3.14: Computing a minimal covering of a union (*minimalCover*(E))

```
Let $E = E_1 + \ldots + E_n$ $(n \geq 2)$;
$F := 0$;
for $i := n$ downto 2, do
    $tabE[i] := F$;
    $F \oplus= E_i$;
$tabE[1] := F$;
$G := minimalCoverAux(0, 0, \mathrm{rep}(E), E, tabE)$;
$reduce(\{\langle G, E\rangle\})$;
```

3.5.3.2 Computing a minimal covering

The second algorithm I propose to simplify a union computes a minimal covering $E_{i_1}+\ldots+E_{i_m}$ of E $(1 \leq i_1 < \ldots < i_m \leq n)$. Precisely, this covering minimizes the value of $\mathrm{size}(E_{i_1} + \ldots + E_{i_m})$. No attempt is made to combine several expressions E_i into an expression shorter than their union. Therefore, it is best to apply this algorithm to the result of the previous one. The algorithm, called *minimalCover*(E), is depicted in Figure 3.14. Most of the work is done by an auxiliary, recursive, algorithm, presented in Figure 3.15.

The effect of this auxiliary algorithm can be specified as follows.

- The parameters $i, F, B, tabE$ are assumed to satisfy the following conditions at each call of the algorithm:

  1. $0 \leq i \leq n$
  2. The expression F is of the form $E_{i_1} + \ldots + E_{i_m}$ with $1 \leq i_1 < \ldots < i_m \leq i$.
  3. $\mathcal{L}(F \oplus (E_{i+1} + \ldots + E_n)) = \mathcal{L}(E)$
  4. $\mathcal{L}(B) = \mathcal{L}(E)$
  5. $\mathrm{size}(F) < \mathrm{size}(B)$
  6. $tabE[j] = E_{j+1} + \ldots + E_n$, for all j such that $1 \leq j \leq n$.

- The algorithm returns a shortest expression among B and all expressions of the form $F \oplus (E_{j_1} + \ldots + E_{j_p})$ such that

  1. $i + 1 \leq j_1 < \ldots < j_p \leq n$
  2. $\mathcal{L}(F \oplus (E_{j_1} + \ldots + E_{j_p})) = \mathcal{L}(E)$

Figure 3.15: The auxiliary algorithm *minimalCoverAux*$(i, F, B, E, tabE)$

```
Let $E = E_1 + \ldots + E_n$ $(n \geq 2)$;
if eq$(F, E)$, do
    return $F$;
noBetterSolution := *false*;
for $j := i + 1$ to $n$ until noBetterSolution, do
    $G := F \oplus E_j$;
    if size$(G) <$ size$(B)$,
        $B :=$ *minimalCoverAux*$(j, G, B, E, tabE)$;
    noBetterSolution := $\neg$eq$(F \oplus tabE[j], E)$;
return $B$;
```

Since the auxiliary algorithm is first called with $i = 0$, $F = 0$, and $B =$ rep(E), it comes that the returned expression G is shortest among rep(E) and all covering of E by a union of its sub-expressions. The top algorithm previously computes an array of expressions *tabE* that respects the condition 6. of the specification of *minimalCoverAux*. This array is used to prevent the recursive algorithm from calling itself with values of F that cannot be a prefix of a covering, i.e., contradicting the condition 3. of the specification.
One can observe that the statement $G := F \oplus E_j$; can be replaced by $G :=$ sunion(F, E_j);, in the algorithm *minimalCoverAux*. This may improve the final result of the algorithm but it does not relieve us from executing *simplifyUnion*(E) beforehand because the condition size$(G) <$ size(B) would fail for expressions G that could be shortened later on, in the recursive call to *minimalCoverAux*. On the other hand, omitting the test if size$(G) <$ size(B) may increase the computational cost of the algorithm much; it also requires to complicate the rest of the algorithm.

3.5.3.3 Other simplifications

Other simplifications can be applied to an union, preferably after applying the algorithms presented in the two preceding sections.

- We can factorize sub-expressions E_i having a common left factor or a common right factor.
- We can add various specific simplification rules in the operation sunion(E_1, E_2). For instance, if $1 \in \mathcal{L}(E_1)$ and E_2 is of the form $F(GF)^*G$, the result $E_1 \oplus (FG)^*$ can be returned. Other rules inspired by [16] can be added.

Figure 3.16: The algorithm *simplifyConcat*(E)

```
Let $E = E_1.E_2$.
if $\neg$hasOne($E_1$), do
    return $E$;
if $E_2$ is of the form $F_{2,1}^* \odot F_{2,2}$, do
    if infEq($E_1, F_{2,1}^*$), do
        return $E_2$;
if $E_1$ is of the form $F_1^*$, do
    let $E_2 = F_{2,1} \odot F_{2,2}$;
    if infEq($F_{2,1}, E_1$), do
        return $simplifyConcat(E_1 \odot F_{2,2})$;
return $E$;
```

3.5.4 Simplifying a concatenation

I propose a simplification algorithm for concatenation, based on two symmetric reduction rules from [16]. Let E_1, E_2 be two normalized regular expressions.

If $1 \in \mathcal{L}(E_1)$, $\mathcal{L}(E_1) \subseteq \mathcal{L}(E_2)$, and E_2 is an iteration, we have:

$$\mathcal{L}(E_1 \odot E_2) = \mathcal{L}(E_2.E_1) = \mathcal{L}(E_2)$$

The algorithm *simplifyConcat*(E) is presented in Figure 3.16. Let $E = E_1.E_2$. As a pre-condition, we may assume that both expressions E_1 and E_2 already are simplified, because the general principle of our simplification method ensures that the sub-expressions of an expression are simplified first. There are two cases where the reduction rule can be applied:

1. If E_2 is of the form $F_{2,1}^* \odot F_{2,2}$ (where $F_{2,2}$ possibly is equal to 1), we can apply the rule, if $1 \in \mathcal{L}(E_1)$ and $\mathcal{L}(E_1) \subseteq \mathcal{L}(F_{2,1}^*)$. Then, E simplifies to $F_{2,1}^* \odot F_{2,2} = E_2$.

2. If E_1 is of the form F_1^*, and E_2 is of the form $F_{2,1} \odot F_{2,2}$(where $F_{2,2}$ possibly is equal to 1), we can apply the rule, if $1 \in \mathcal{L}(F_{2,1})$ and $\mathcal{L}(F_{2,1}) \subseteq \mathcal{L}(E_1)$. Then, E simplifies to $E_1 \odot F_{2,2}$. However, it is not necessarily the case that $E_1 \odot F_{2,2}$ is simplified. Thus, we recursively apply the algorithm to $E_1 \odot F_{2,2}$ (only if $E_1 \odot F_{2,2}$ actually is a concatenation). The recursive call terminates because $E_1 \odot F_{2,2}$ is shorter than E.

To see how this method works, let us consider the following “large” concatenation:

(1+a)(1 + bb)(a + b)*(1 + ab)a*(1 + b)b*(1 + a)

The sub-expressions are simplified first. Thus the whole work is done as follows, from right to left, since the parentheses are implicitely placed as follows:

(1+a)((1 + bb)((a + b)*((1 + ab)(a*((1 + b)(b*(1 + a)))))))

1. b*(1 + a) already is simplified.
2. (1 + b)b*(1 + a) simplifies to b*(1 + a) .
3. a*b*(1 + a) already is simplified.
4. (1 + ab)a*b*(1 + a) already is simplified.
5. (a + b)*(1 + ab)a*b*(1 + a) simplifies to (a + b)*a*b*(1 + a) , which successively recursively simplifies to
   (a) (a + b)*b*(1 + a) ,
   (b) (a + b)*(1 + a) ,
   (c) (a + b)* .
6. (1 + bb)(a + b)* simplifies to (a + b)* .
7. (1+a)(a + b)* simplifies to (a + b)* .

It is interesting to note that most or all of these simplifications can be done by simply using some of the “core” algorithms: computing derivatives, reducing the background, minimizing equations, and solving equations. Only using the computation of derivatives and reduction of the background , the global simplification process simplifies the expression to

(a + b)*(1 + ab)a*b*(1 + a)

since (1 + b)b*(1 + a) simplifies to b*(1 + a) because, after reduction, both expressions have the same set of derivatives and because b*(1 + a) is a direct sub-expression of (1 + b)b*(1 + a). For the same reason, (1 + bb)(a + b)* simplifies to (a + b)* and (1+a)(a + b)* simplifies to (a + b)* . If we also use complete minimization of the equations or solving of the equations, we get the best result (a + b)*. In the first case, it is because (a + b)* is equivalent to (a + b)*(1 + ab)a*b*(1 + a) and is a sub-expression of it. In the latter case, it is because solving the (minimized) equations of (a + b)*(1 + ab)a*b*(1 + a) returns (a + b)*. Nevertheless, using the simplification algorithm slightly is more efficient.

One can also observe that the algorithm *simplifyConcat*(E) is similar to *minimalCover*(E) in that it computes a kind of minimal covering of E. We can complement it with another algorithm able to simplify E in another way, using, for instance, the Kleene axioms

$$(x^*y)^*x^* \;=\; (x+y)^* \;=\; y^*(xy^*)^*$$

I do not provide an explicit version of this algorithm here[14] but it is already important to note that in order to be useful these axioms must sometimes be used to produce longer expressions that can be better reduced later, as in the following example:

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

3.5.5 Simplifying an iteration

Simplification of an iteration $E = F^*$ should best be called *simplification under star* because it mainly amounts to simplify the (sub-)expression F with more powerful simplification rules than those that are applicable to F regardless of the fact that it occurs "under" the star symbol. Eight such simplification rules under star are given in [16]. I propose two algorithms that apply these rules in a way more systematic than what is proposed in [16]. Moreover, I show that it is possible to strengthen the rules[15]. A benefit of this "stronger" method is that it provides a clear recursive algorithm to "flatten" the expression F into an expression $F_1 + \ldots + F_n$ ($n \geq 1$) the size of which *at most is* the size of F, and is even much shorter, sometimes. Finally, this union possibly can be further simplified by removing some unnecessary sub-expressions and/or factoring some of them. (Of course, this clearly is an optimistic presentation. Obviously, there are cases where no simplifications are applicable.)

The algorithm to flatten F "under star" is depicted in Figure 3.18. It uses an auxiliary operation *goodSplit*(F_1, F_2, E), which, roughly speaking, determines whether a concatenation $F_1 \odot F_2$ can be replaced by the expression $F_1 \oplus F_2$ inside an expression $\ldots + F_1 \odot F_2 + \ldots$ such that $(\ldots + F_1 \odot F_2 + \ldots)^* = E$. The operation *goodSplit* is presented in Figure 3.17.

Figure 3.17: The operation *goodSplit*(F_1, F_2, E)

```
If hasOne($F_1$), do
  if hasOne($F_2$), do
    return *true* ;
  else, do
    return infEq($F_1, E$) ;
else, do
  if hasOne($F_2$), do
    return infEq($F_2, E$) ;
  else, do
    return infEq($F_1, E$) & infEq($F_2, E$) ;
```

[14]maybe later on...

[15]at an additional computation cost

Figure 3.18: The algorithm *flattenAux*(P, F, E)

```
1. If $F = 1$, do
      return $P$ ;

2. If $F$ is a letter $x$, do
      return $P \oplus x$ ;

3. If $F$ is an iteration $(F')^*$, do
      return $\mathit{flattenAux}(P, F', E)$ ;

4. If $F$ is a union $F_1 + \ldots + F_n$ $(n \geq 2)$, do
      $i := 0$ ;
      while $i \neq n$, do
         $i := i + 1$ ;
         $P := \mathit{flattenAux}(P, F_i, E)$ ;
      return $P$ ;

5. If $F$ is a concatenation $F_1.F_2$, do
      if $\mathit{goodSplit}(F_1, F_2, E)$, do
         return $\mathit{flattenAux}(\mathit{flattenAux}(P, F_1, E), F_2, E)$ ;
      while $F_2$ is a concatenation $F'_1.F'_2$, do
         $F_1 := F_1 \odot F'_1$ ; $F_2 := F'_2$ ;
         if $\mathit{goodSplit}(F_1, F_2, E)$, do
            return $\mathit{flattenAux}(\mathit{flattenAux}(P, F_1, E), F_2, E)$ ;
      return $P \oplus F$ ;
```

Technically, the specification of the operation *goodSplit*(F_1, F_2, E) is the following: Assuming that F_1 and F_2 are different from 0 and 1, and that $\mathrm{infEq}(F_1 \odot F_2, E)$ holds, it returns *true* if and only if one has $\mathrm{infEq}(F_1, E)$ and $\mathrm{infEq}(F_2, E)$.

Note that, in order to flatten F under star, the exact statement to write is

$$F := \mathit{flattenAux}(0, F, F^*) ;$$

The reason is that the algorithm *flattenAux*(P, F, E) uses an accumulator P, which is more efficient than writing a direct recursive algorithm *flatten*(F, E), with only two arguments.

Before proving the correctness of the algorithm *flattenAux*(P, F, E), it is interesting to show how it works on a significant example. Let us consider the following iteration

$$E = (\mathrm{b}^*((\mathrm{a} + \mathrm{b})^*\mathrm{a}(\mathrm{a}(\mathrm{b}^* + \mathrm{a}^*))^*)^*)^*$$

Thus, in this case, we have: $F = \mathrm{b}^*((\mathrm{a} + \mathrm{b})^*\mathrm{a}(\mathrm{a}(\mathrm{b}^* + \mathrm{a}^*))^*)^*$.

1. The expression F is a concatenation, wich first is split into two expressions b* and ((a + b)*a(a(b* + a*))*)* because the operation hasOne returns *true* for both.

2. The recursive call on b* returns b, passing by Step 3 and Step 2.

3. Then, the algorithm is applied to b, ((a + b)*a(a(b* + a*))*)*, and E. The expression ((a + b)*a(a(b* + a*))*)* is an iteration. So Step 3 is applied, which leads to a call on the arguments b, (a + b)*a(a(b* + a*))*, and E.

4. This time (a + b)*a(a(b* + a*))* is a concatenation, which is split into (a + b)* and a(a(b* + a*))* because hasOne((a + b)*) and infEq(a(a(b* + a*))*, E) return *true*.

5. The call on b, (a + b)*, and E passes by Step 3, Step 4, and Step 2 (two times). Thus it returns b $\oplus$ a $\oplus$ b, i.e., a + b.

6. Afterwards, there is a call on the arguments a + b, a(a(b* + a*))*, and E.

7. The concatenation a(a(b* + a*))* is split into a and (a(b* + a*))* because infEq(a, E) and hasOne((a(b* + a*))*) both return *true*.

8. The call on a + b, a, and E executes Step 2 and returns (a + b) $\oplus$ a $=$ a + b.

9. The call on a + b, (a(b* + a*))*, and E successively passes by Step 3, Step 5 (where a(b* + a*) is split into a and b* + a*), Step 2, Step 4, and, two times, Step 3 followed by Step 2. It returns (a + b) $\oplus$ a $\oplus$ b $\oplus$ a $=$ a + b.

In conclusion, the expression F is simplified to a + b "under star", which means that E is simplified to (a + b)*. Two interesting remarks can be added here.

- It can be proved that *any* iteration E, equivalent to (a + b)*, can actually be simplified to (a + b)* by the method explained above. Of course, this also holds for an expression using any number of letters.

- It is possible to use a lighter version of the operation *goodSplit*, which only uses the operation hasOne. This corresponds to two rules proposed in [16]. On the preceding example, it results to a much less good simplification:

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

  Nevertheless, the algorithms proposed in [16], produce the expected result (a + b)*, using some other simplification rules (Alley Stoughton, personal communication, October 6, 2022).

Now, I can give a correctness proof of the algorithm of Figure 3.18. It does not give very strong information on the simplifications that are made but, at least, it guarantees that the result is not longer than the input, and that it denotes the same regular language.

Thus the specification of *flattenAux* (P, F, E) is as follows: Assume that F is different from 0, E is an iteration, and there exists an expression S[16] such that $\mathcal{L}((P \oplus F \oplus S)^\star) = \mathcal{L}(E)$, then the algorithm returns an expression P' such that $\mathcal{L}((P' \oplus S)^*) = \mathcal{L}(E)$ and $\text{size}(P') \leq \text{size}(P) + \text{size}(F) + 1$. (The expression P' never is an iteration.)

The correctness proof of the algorithm is made simpler by first proving some properties of normalized regular expressions.

Theorem 5 *Let Q be a normalized expression. The following properties hold.*

1. $\mathcal{L}\,(Q \oplus 1)^\star = \mathcal{L}\,(Q)^\star$
2. *Let F^*, an iteration. Then,* $\mathcal{L}\,(Q \oplus F^*)^\star \;=\; \mathcal{L}\,(Q \oplus F)^\star$.
3. *Let F, F_1, F_2, three expressions, such that $F = F_1 \odot F_2$ and $\mathcal{L}(F_i) \subseteq \mathcal{L}(Q \oplus F)^\star$ ($i = 1, 2$). One has:* $\mathcal{L}\,(Q \oplus F)^\star \;=\; \mathcal{L}\,(Q \oplus F_1 \oplus F_2)^\star$.

Proof of Theorem 5

1. $\mathcal{L}\,(Q)^\star$ is the set of all chains of letters that can be built by concatenating an arbitrary number of chains belonging to $\mathcal{L}(Q)$. Adding occurrences of the empty chain anywhere in such a concatenation does not change its value. Thus, $\mathcal{L}\,(Q \oplus 1)^\star = \mathcal{L}\,(Q)^\star$.
2. Any chain of $\mathcal{L}\,(Q \oplus F^*)^\star$ is the concatenation of a number of chains belonging to $\mathcal{L}\,(Q)$ or $\mathcal{L}\,(F^*)$. But chains belonging to $\mathcal{L}\,(F^*)$ are concatenations of chains from $\mathcal{L}\,(F)$. Thus, $\mathcal{L}\,(Q \oplus F^*)^\star \;=\; \mathcal{L}\,(Q \oplus F)^\star$.
3. Firstly, it is clear that $\mathcal{L}\,(Q \oplus F)^\star \;\subseteq\; \mathcal{L}\,(Q \oplus F_1 \oplus F_2)^\star$, because any chain of $\mathcal{L}(F)$ is the concatenation of a chain of $\mathcal{L}(F_1)$ with a chain of $\mathcal{L}(F_2)$. Secondly, because $\mathcal{L}(F_i) \subseteq \mathcal{L}(Q \oplus F)^\star$ ($i = 1, 2$), any chain of $\mathcal{L}(F_1 \oplus F_2)$ is the concatenation of chains from $\mathcal{L}(Q \oplus F)^\star$. Hence, $\mathcal{L}\,(Q \oplus F_1 \oplus F_2)^\star \subseteq \mathcal{L}\,(Q \oplus F)^\star$.

(End of proof)

Note

To prove the correctness of the algorithm *flattenAux* (P, F, E), we need to assume that $\text{size}(0) = -1$. This is needed to prove that $\text{size}(0) + \text{size}(F) + 1 \;\leq\; \text{size}(F)$.

(End of note)

Correctness proof of the algorithm *flattenAux* (P, F, E)

The proof is by induction on the structure of F. Let us consider the possible cases for F.

1. If $F = 1$, only Step 1 is executed and P is returned. The result holds with $P' = P$.

[16] The specification holds for any recursive call of the algorithm. The "existential" expression S stands for a union of the expressions waiting to be taken into account in the outer calls of the algorithm. And, in fact, it is possible to give a non recursive (iterative) version of the algorithm, where S becomes a local variable.

2. If F is a letter x, Step 2 is executed and $P \oplus x$ is returned. The result hold with $P' = P \oplus x$. Notice that the size of P' can be lower than $\text{size}(P) + \text{size}(F) + 1$.

3. If F is an iteration $(F')^*$, the algorithm is recursively called with F' instead of F. By the second property of Theorem 5, $\mathcal{L}((P \oplus F' \oplus S)^\star) = \mathcal{L}((P \oplus F \oplus S)^\star) = \mathcal{L}(E)$. Thus, by induction hypothesis, the algorithm returns P' such that $\mathcal{L}((P' \oplus S)^*) = \mathcal{L}(E)$ and $\text{size}(P') \leq \text{size}(P) + \text{size}(F') + 1 < \text{size}(P) + \text{size}(F) + 1$.

4. If F is a union $F_1 + \ldots + F_n$ $(n \geq 2)$, let P_0 be the initial value of P.

   We prove by induction on i_1 that for all value of $i_1 \in \{0, \ldots, n\}$, the condition $i \neq n$ in the while loop of Step 4, is evaluated with $i = i_1$, $\mathcal{L}(P \oplus F_{i_1+1} \oplus \ldots \oplus F_n \oplus S)^\star = \mathcal{L}(E)$, and $\text{size}(P) <= \text{size}(P_0) + \text{size}(F_1 \oplus \ldots \oplus F_{i_1}) + 1$. One easily checks that the assertion holds for $i_1 = 0$, due to the precondition of the algorithm. Now consider i_1 such that $0 < i_1 \leq n$. We can assume that the result holds for $i_1 - 1$. Let us call P_{i_1-1}, the current value of P. The algorithm is recursively called on P_{i_1-1}, F_{i_1}, and E. By induction on i_1, one has : $\mathcal{L}(P_{i_1-1} \oplus F_{i_1} \oplus \ldots \oplus F_n \oplus S)^\star = \mathcal{L}(E)$. The precondition holds, using $F_{i_1+1} \oplus \ldots \oplus F_n \oplus S$ as the current "existential" value of S. Thus, by induction of the parameter F, the recursive call returns a value of P such that $\mathcal{L}(P \oplus F_{i_1+1} \oplus \ldots \oplus F_n \oplus S)^\star = \mathcal{L}(E)$ and $\text{size}(P) \leq \text{size}(P_{i_1-1}) + \text{size}(F_{i_1}) + 1 \leq (\text{size}(P_0) + \text{size}(F_1 \oplus \ldots \oplus F_{i_1-1}) + 1) + \text{size}(F_{i_1}) + 1$ $= \text{size}(P_0) + \text{size}(F_1 \oplus \ldots \oplus F_{i_1}) + 1$.

   In the particular case where $i_1 = n$, the condition of the while loop is false, so that the algorithm returns a value of P such that $\mathcal{L}(P \oplus S) = \mathcal{L}(E)$ and $\text{size}(P) \leq \text{size}(P_0) + \text{size}(F) + 1$, as expected.

5. If F is a concatenation, Step 5 is executed. The algorithm of Step 5 attemps to decompose F into F_1 and F_2 such that $\mathcal{L}(F_i) \subseteq \mathcal{L}(E)$ $(i = 1, 2)$ (of course, $F = F_1 \odot F_2$).

   If such a decomposition is found, a recursive call *flattenAux*(P, F_1, E) first is executed. Because of Property 3 of Theorem 5, one has: $\mathcal{L}(E) = \mathcal{L}(P \oplus F \oplus S)^\star = \mathcal{L}(P \oplus F_1 \oplus (F_2 \oplus S))^\star$. Thus, the precondition of *flattenAux* is fulfilled, with $F_2 \oplus S$ as "existential" value of S. Therefore, we may assert that the recursive call returns a value P_1 such that $\mathcal{L}(P_1 \oplus (F_2 \oplus S))^* = \mathcal{L}(E)$ and $\text{size}(P_1) \leq \text{size}(P_0) + \text{size}(F_1) + 1$. The outer recursive call is then executed applying *flattenAux* to P_1, F_2, E. The precondition of *flattenAux* is respected, because $\mathcal{L}(P_1 \oplus F_2 \oplus S)^* = \mathcal{L}(E)$. Thus, Step 5 terminates and returns a value P_2 such that $\mathcal{L}(P_2 \oplus S)^\star = \mathcal{L}(E)$ and $\text{size}(P_2) \leq \text{size}(P_1) + \text{size}(F_2) + 1 \leq (\text{size}(P_0) + \text{size}(F_1) + 1) + \text{size}(F_2) + 1 \leq \text{size}(P_0) + \text{size}(F) + 1$.

   If no such decomposition is found, the value $P \oplus F$ is returned, which trivially respects the postcondition of the specification.

(End of correctness proof)

Corollary 2 *Now we can prove the correctness of our simplification method "under star". Remember that, in order to simplify F^*, we first execute:*

$$P := \mathit{flattenAux}(0, F, F^*);$$

Afterwards, we simply return $P^\star$. *Clearly, the precondition of flattenAux is fulfilled with* $P = 0$, $E = F^*$, *and the existential value of* S *equal to* 0. *Thus, the returned value of* P *is such that* $\mathcal{L}(P^\star) = \mathcal{L}(F^*)$ *and* $\mathit{size}(P) \leq \mathit{size}(0) + \mathit{size}(F) + 1 = (-1) + \mathit{size}(F) + 1 = \mathit{size}(F)$.

Bibliography

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

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

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[5] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In Richard C. Smith, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 40–45. IEEE Computer Society Press, 1990.

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

[7] Baudouin Le Charlier. Experimental evaluation of a method to simplify expressions. *CoRR*, abs/2003.06203, 2020.

[8] Baudouin Le Charlier and Mêton Mêton Atindehou. A method to simplify expressions: Intuition and preliminary experimental results. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL@LPAR 2015, 11th International Workshop on the Implementation of Logics, Suva, Fiji, November 23, 2015*, volume 40 of *EPiC Series in Computing*, pages 37–51. EasyChair, 2015.

[9] 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.

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

[11] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.

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

[13] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994.

[14] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972*, pages 125–129. IEEE Computer Society, 1972.

[15] 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.

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

[17] Alley Stoughton. The Forlan Project. https://alleystoughton.us/forlan/, 2003–2022. Open source software.

Appendix A

Syntactic derivatives

A.1 An example

Let E be the following normalized regular expression:

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

A.1.1 Syntactic derivatives of E

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

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

A.1.2 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.

$$\begin{array}{lcl|lcl}
E & = & 1 + aE_a + bE_b & E_{aa} & = & 1 + aE_{aaa} + bE_{ab} \\
E_a & = & 1 + aE_{aa} + bE_{ab} & E_{aaa} & = & 1 + aE_{aaaa} + bE_{ab} \\
E_b & = & 1 + aE_{aa} + bE_{bb} & E_{aaaa} & = & 1 + aE_{ab} + bE_{ab} \\
E_{bb} & = & 1 + aE_{aa} + bE_{ba} & E_{ab} & = & 1 + aE_{ab} + bE_{ab}
\end{array}$$

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., a*((a + b)a*)* . Moreover, if we solve the remaining equation, as explained in Section 3.4, E_{ab} ultimately is simplified to (a + b)* (and so is E). Interestingly, minimization of the set of equations is not needed in this case. Reducing the background is enough.

A.1.3 Partial derivatives of E

One can see that the partial derivatives of E are:

P_1 = ((a + b)a*)*	P_4 = a*((a + b)a*)*	P_7 = a(1 + a)
P_2 = (a + b(1 + b)b)aa(1 + a)	P_5 = (1 + b)baa(1 + a)	P_8 = a
P_3 = aa(1 + a)	P_6 = baa(1 + 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, in theory.

A.2 On derivatives of sets of chains

In this section, I elaborate a bit on the notion of derivatives of sets of chains of letters and I discuss their characterization by Conway in [10], as opposed to mine in Section 3.2.2.

Let S be a *set of chains of letters*, but not necessarily a regular set. Let w a chain of letters. By definition, the derivative of S with respect to w, is the set of chains $\{u \in \mathit{Letter}^* | w.u \in S\}$. We denote this set by $D_w\, S$.

Derivatives of sets of chains enjoy a number of properties that are summarized in Figure A.1. These properties can easily be proven from the definition above. Furthermore, they can be used to prove Theorem 1 of Section 3.2.2. It is also possible to prove the theorem directly from the definition above but proving the properties and the theorem separately is clearer (in my opinion).

Now I discuss the "characterization" of derivatives of regular sets proposed by Conway in [10]. A similar characterization is given for normalized regular expressions in Figure A.2. I want to make the following comments about these rules.

Figure A.1: Derivatives of sets of chains

$$\begin{array}{|ccll|}
\hline
D_x\,\{\} & = & \{\} & \\
D_x\,\{1\} & = & \{\} & \\
D_x\,\{y\} & = & \{1\} & \text{if } \ y = x \\
 & = & \{\} & \text{otherwise} \\
D_x\,(S_1 \,\cup\, \ldots \,\cup\, S_n) & = & D_x\,S_1 \,\cup\, \ldots \,\cup\, D_x\,S_n & (n \geq 2) \\
D_x\,(S_1\,.\,S_2) & = & D_x\,S_1\,.\,S_2 & \text{if } \ 1 \notin S_1 \\
 & = & D_x\,S_1\,.\,S_2 \;\cup\; D_x\,E_2 & \text{otherwise} \\
D_x\,S^\star & = & D_x\,S\,.\,S^\star & \\
\hline
D_1\,S & = & S & \\
D_{w\,x}\,S & = & D_x\,(D_w\,S) & \\
\hline
\end{array}$$

Figure A.2: Derivatives of normalized expressions adapted from [10]

$$\begin{array}{|ccll|}
\hline
o(E) & = & 1 & \text{if } \ 1 \in \mathcal{L}(E_1) \\
 & = & 0 & \text{otherwise} \\
\hline
D_x\,0 & = & 0 & \\
D_x\,1 & = & 0 & \\
D_x\,y & = & 1 & \text{if } \ y = x \\
 & = & 0 & \text{otherwise} \\
D_x\,(E_1 \,+\, \ldots \,+\, E_n) & = & D_x\,E_1 \,\oplus\, \ldots \,\oplus\, D_x\,E_n & (n \geq 2) \\
D_x\,(E_1\,.\,E_2) & = & D_x\,E_1 \,\odot\, E_2 \,\oplus\, o(E_1) \odot D_x\,E_2 & \\
D_x\,E^* & = & D_x\,E \,\odot\, E^* & \\
\hline
D_1\,E & = & E & \\
D_{w\,x}\,E & = & D_x\,(D_w\,E) & \\
\hline
\end{array}$$

- Using these rules as a basis for proving the correctness of my own definition of syntactic derivatives (see Figure 3.2 in Section 3.2.2) is less convenient than using the properties of derivatives of sets of chains given in Figure A.1. It needs some technical manipulations of normalized expressions that are less natural than reasoning about the sets denoted by the normalized expressions.

- Conway does not make a clear distinction between the regular expressions and the sets of chains they denote. It uses the same symbols (typically, E) for both. The rules are about regular sets but they *assume* that these regular sets are explicitely written as regular expressions. It is completely similar to the classical derivation rules given for functions of a real variable. To be able to apply the rules to functions the latter have to be represented by formulas that can be unified to the left parts of the rules (e.g., a scheme for polynomials).

- As a consequence, the rules are not totally sufficient for practical applications. They must be completed by simplification rules. In the case of functions of a real variable, it is not at all practical in general to compute the second and third derivative of a function without simplifying the previous one beforehand. The same applies to the derivatives of regular expressions/languages. For instance, when computing the expression $D_x\, E_1 \odot E_2 \;\oplus\; o(E_1) \odot D_x\, E_2$, it is essential to simplify the expression to $D_x\, E_1 \odot E_2$ in the case that $o(E_1) = 0$. One can see that the rules are supposed to be applied by hand, simplifying expressions "on the fly". In my work, however, this is not necessary. Normalization of the expressions is sufficient.