Deciding Kleene Algebra Terms Equivalence in Coq

Nelma Moreira[a], David Pereira[b], Simão Melo de Sousa[c]

[a] *CMUP & DCC-FC – University of Porto, Rua do Campo Alegre 1021, 4169-007, Porto, Portugal*
[b] *CISTER Research Centre – ISEP/IPP, Rua Dr. António Bernardino de Almeida 431, 4200-072 Porto, Portugal*
[c] *LIACC & DI – University of Beira Interior, Rua Marquês d´Avila e Bolama, 6201-001, Covilhã, Portugal*

Abstract

This paper presents a mechanically verified implementation of an algorithm for deciding the equivalence of Kleene algebra terms within the Coq proof assistant. The algorithm decides equivalence of two given regular expressions through an iterated process of testing the equivalence of their partial derivatives and does not require the construction of the corresponding automata. Recent theoretical and experimental research provides evidence that this method is, on average, more efficient than the classical methods based in automata. We present some performance tests, comparisons with similar approaches, and also introduce a generalization of the algorithm to decide the equivalence of terms of Kleene algebra with tests. The motivation for the work presented in this paper is that of using the libraries developed as trusted frameworks for carrying out certified program verification.

Keywords: proof assistants, regular expressions, Kleene algebra with tests, program verification

1. Introduction

Formal languages are one of the pillars of Computer Science. Amongst the several computational models of formal languages, that of *regular expression* is one of the most widely known and used. The notion of regular expressions has its origins in the seminal work of Kleene, where the author introduced them as a specification language for *deterministic finite automata* (DFA) [**?**]. Nowadays, regular expressions find applications in a wide variety of areas due to their capability of expressing patterns in a succinct and comprehensive way. They abound in technologies deriving from the *World Wide Web*, in text processors, in structured languages such as XML, and are a core element of programming languages like Perl [**?**] and Esterel [**?**]. More recently, regular expressions have been successfully applied in the runtime verification of programs [**? ?**].

In the past years, much attention has been given to the mechanization of *Kleene algebra* (KA) – the algebra of regular expressions – within proof assistants. Formally, a KA is an idempotent semiring together with the Kleene star operator $\cdot^\star$, that is characterized axiomatically. J.-C. Filliâtre [**?**] provided a first formalisation of the Kleene theorem for regular languages [**?**] within the Coq proof assistant [**?**]. Höfner and Struth [**?**] investigated the automated reasoning in variants of Kleene algebras with Prover9 and Mace4 [**?**]. Pereira and Moreira [**?**] implemented in Coq an abstract specification of *Kleene algebra with tests* (KAT) [**?**] and the proofs that propositional Hoare logic deduction rules are theorems of KAT. An obvious follow up of that work was to implement a certified procedure for deciding equivalence of KA terms, *i.e.*, regular expressions. A first step was the proof of the correctness of the partial derivative automaton construction from a regular expression [**?**]. In this paper we describe the mechanization of a decision procedure based on partial derivatives that was proposed by Almeida *et al.* [**?**], and that is a functional variant of the rewrite system introduced by Antimirov and Mosses in [**?**]. This procedure decides regular expression equivalence through an iterated process of testing the equivalence of their partial derivatives.

Similar approaches based on the computation of a bisimulation between the two regular expressions were used recently. In 1971, Hopcroft and Karp [**?**] presented an almost linear algorithm for equivalence of two DFA. By transforming regular expressions into equivalent DFAs, Hopcroft and Karp's method can be used for regular expressions

Email addresses: nam@ncc.up.pt (Nelma Moreira), dmrpe@isep.ipp.pt (David Pereira), desousa@di.ubi.pt (Simão Melo de Sousa)

equivalence. A comparison of that method with the method here proposed is discussed by Almeida *et. al.* [**?** **?**]. There it is conjectured that a direct method should perform better on average, and that is corroborated by theoretical studies based on analytic combinatorics [**?**]. Hopcroft and Karp's method was used by Braibant and Pous [**?**] to formally verify Kozen's proof of the completeness of Kleene algebra [**?**] in Coq.

Independently of the work presented here, Coquand and Siles [**?**] mechanically verified an algorithm for deciding regular expression equivalence based on Brzozowski's derivatives [**?**] and an inductive definition of finite sets called Kuratowski-finite sets. Based on the same notion of derivative, Krauss and Nipkow [**?**] provide an elegant and concise formalisation of Rutten's co-algebraic approach of regular expression equivalence [**?**] in the Isabelle proof assistant [**?**], but they do not address the termination of the decision procedure. Komendantsky provides a novel functional construction of the *partial derivative automaton* [**?**], and also made contributions [**?**] to the mechanization of concepts related to the Mirkin's construction [**?**] of that automata. More recently, Andrea Asperti formalized a decision procedure for the equivalence of *pointed regular expressions* [**?**], that is both compact and efficient.

Besides avoiding the need for building DFAs, our use of partial derivatives also avoids the necessary normalisation of regular expressions modulo *ACI* (*i.e.*, the normalization modulo associativity, idempotence and commutativity of the union of regular expressions) in order to ensure the finiteness of Brzozowski's derivatives. Like in other approaches [**?**], our method also includes a refutation step that improves the detection of inequivalent regular expressions.

Although the algorithm we have chosen to verify seems straightforward, the process of its mechanical verification in a theorem prover based in a type theory raises several issues which are quite different from a usual implementation in standard programming languages. The Coq proof assistant allows users to specify and implement programs, and also to prove that the implemented programs are compliant with their specification. In this sense, the first task is the effort of formalizing the underlying algebraic theory. Afterwards, and in order to encode the decision procedure, we have to provide a formal proof of its termination since our procedure is a general recursive one, whereas Coq's type system accepts only provable terminating functions. Finally, a formal proof must be provided in order to ensure that the functional behavior of the implemented procedure is correct wrt. regular expression equivalence. Moreover, the encoding effort must be conducted with care in order to obtain a solution that is able to compute inside Coq, or extracted and compiled as an OCaml development, both with reasonable performances.

1.1. Paper organization

This paper is organized as follows. In Section 2 we provide a concise introduction to the Coq proof assistant. In Section 3 we review some of the concepts of formal languages that we need to formalise in order to implement the decision procedure; in Section 4 we describe the formalisation of the decision procedure, its proofs of correctness and completeness, and comment on the procedure's computational efficiency; in Section **??** we describe the generalization of the decision procedure to decide KAT terms equivalence, and show how this procedure is useful in program verification; finally, in Section **??** we present our conclusions about the work presented in this paper, and point to future research directions. The work presented here is an extended version of the work previously presented in [**?** **?**], and the corresponding development in Coq is available at [**?**].

2. An Overview of the Coq Proof Assistant

The Coq proof assistant [**?**] is an implementation of Paulin-Mohring's *Calculus of Inductive Constructions* (CIC) [**?**]. The CIC is a rich typed λ-calculus that features polymorphism, dependent types, and that extends Coquand and Huet's *Calculus of Constructions* (CC) [**?**] with very expressive (co-)inductive types.

The CIC is built upon the *Curry-Howard Isomorphism* (CHI) *programs-as-proofs* principle [**?**], where a typing relation $t : A$ is interpreted either as a term t that has the type A, or as t being a proof of the proposition A. Hence, the CIC is simultaneously a functional programming language with a very expressive type system and a higher-order logic, and so, users can define specifications of programs, and also build proofs concerning those specifications.

In the CIC there exists no distinction between terms and types. Therefore, all types also have their own type, called a *sort*, and each sort belongs to the well-formed set $\mathcal{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}$, where $\mathsf{Type}(i)$ is the type of smaller sorts $\mathsf{Type}(j)$ with $j < i$, including the sorts Prop and Set which ensure a strict separation between *logical types* and *informative types*: the former is the type of propositions and proofs, whereas the latter accommodates

data types and functions defined over those data types. An immediate effect of the non-existing distinction between types and terms in CIC is that computations occur both in programs and in proofs. A fundamental feature of Coq's underlying type system is the support for *dependent product types* $\Pi x : A.B$ which extends functional types $A \to B$ in the sense that the type of $\Pi x : A.B$ is the type of functions that map each instance of x of type A to a type of B where x may occur in it. If x does not occur in B then the dependent product corresponds to the function type $A \to B$.

Inductive definitions are a key ingredient of Coq. Inductive types are introduced by a collection of *constructors*, each with its own arity. A term of an inductive type is a composition of such constructors and if T is the type under consideration, then its constructors are functions whose final type is T, or an application of T to arguments. Using *pattern matching*, we can implement recursive functions by deconstructing the given term and producing new terms for each constructor. For instance, it is straightforward to define Peano natural numbers and a function `plus` that implements addition on these numbers:

```
Inductive nat : Set :=
| 0 : nat
| S : nat → nat.

Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (p + m) end
where "n + m" := (plus n m).
```

The definition of `plus` is accepted by Coq's type-checker because it exhaustively pattern-matches over all the constructors of `nat`, and because the recursive calls are performed on terms that are *structurally smaller* than the recursive argument. This is a strong requirement of CIC that forces all functions to be terminating.

We can define inductive types that are more complex that `nat`, namely, inductive types that depend on values. A classic example is the family of vectors of length $n \in \mathbb{N}$, whose elements have a type `A`:

```
Inductive vect (A : Type) : nat → Type :=
| vnil : vect A 0
| vcons : ∀ n : nat, A → vect A n → vect A (S n)
```

Given the definition of `vect`, we can define the concatenation of vectors, as follows:

```
Fixpoint app(n:nat)(l1:vect A n)(n':nat)(l2:vect A n'){struct l1} : vect (n+n') :=
 match l1 in (vect _ m') return (vect A (m' + n')) with
  | vnil ⇒ l2
  | vcons n0 v l1' ⇒vcons A (n0 + n') v (app n0 l1' n' l2)
 end.
```

Note that there is a difference between the pattern-matching construction used in the definition of `plus` and the one used to implement `app`: in the latter, the returning type depends on the sizes of the vectors given as arguments; therefore, the extended `match` construction in `app` has to bind the dependent argument m' to ensure that the final return type is a vector whose size is $n + n'$.

In Coq's environment, the primitive way to construct a proof is to explicitly build CIC terms. However, proofs can be built more conveniently, in an interactive and backward fashion through the usage of high-level commands called *tactics*. The CIC terms built by tactics are always verified by Coq's type checker, which ensures that possible errors in the tactics do not interfere with the soundness of the proof construction process.

We finish our brief introduction to Coq addressing the development of non structurally recursive functions. Above we have seen pattern matching over (dependent) inductive types, and whose decreasing criteria is structural recursion. However, this approach is not always possible and the way to deal with this problem is via an encoding of the original formulation into an equivalent function that is structurally recursive. There are several techniques available to address the development of non-structurally decreasing functions in Coq, which are described in detail in [**?**]; here we will consider the method for defining *well-founded recursive functions*.

A given binary relation $\mathcal{R}$ over a set S is said to be *well-founded* if for all elements $x \in S$, there exists no infinite sequence $(x, x_0, x_1, x_2, \dots)$ of elements of S such that $(x_{i+1}, x_i) \in \mathcal{R}$, for all $i \in \mathbb{N}$. Well-founded relations are available in Coq through the definition of the inductive predicate `Acc` and the predicate `well_founded` :

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
| Acc_intro : ( ∀ y : A, R y x →Acc A R y) →Acc A R x
```

Since the type `Acc` is inductively defined, we can use it as the structurally recursive argument in the definition of a function. Thankfully, Coq provides a high-level command named `Function` [**?**] that eases the burden of manually constructing a recursive function over `Acc` predicates. The command `Function` allows users to explicitly state that the target function is going to be defined over a proof that asserts that the underlying recursive measure is well-founded.

For further information about the details of the Coq proof assistant, we point the reader to the works of Bertot and Casterán [**?**], of Chlipala [**?**], and of Pierce *et. al.* [**?**].

3. Preliminaries of Formal Languages

In this section we introduce some classic concepts of formal languages that we will need in the work we are about to describe. These concepts can be found in the introductory chapters of classical textbooks such as the one by Hopcroft and Ullman [**?**] or the one by Kozen [**?**]. The encoding in Coq of the several definitions that we are about to introduce can be seen in [**?**].

3.1. Alphabets, Words and Languages

An *alphabet* Σ is a non-empty finite set of objects usually called *symbols* (or *letters*). A *word* (or *string*) over an alphabet Σ is a finite sequence of symbols from Σ. A *language* is any finite or infinite set of words over an alphabet Σ. Given an alphabet Σ, the set of all words over Σ, denoted by $\Sigma^\star$, is inductively defined as follows: the empty word ϵ is an element of $\Sigma^\star$ and, if $w \in \Sigma^\star$ and $a \in \Sigma$, then aw is also a member of $\Sigma^\star$. The constant languages are the *empty language*, the language containing only ϵ, and the language containing only a symbol $a \in \Sigma$. The operations over languages include the usual Boolean set operations (union, intersection, and complement), plus *concatenation*, *power* and *Kleene star*. The concatenation of two languages L_1 and L_2 is defined by $L_1 L_2 = \{wu \,|\, w \in L_1 \wedge u \in L_2\}$. The *power* of a language L, denoted by L^n, with $\in \mathbb{N}$, is inductively defined by $L^0 = \{\epsilon\}$, and $L^{n+1} = LL^n$, for $n \in \mathbb{N}$. The *Kleene star* of a language L is the union of all the finite powers of L, that is,

$$L^\star = \bigcup_{i \geq 0} L^i. \tag{1}$$

We denote language equality by $L_1 = L_2$. Finally, we introduce the concept of the *left-quotient* of a language L with respect to a word $w \in \Sigma^\star$, which is defined as $\mathcal{D}_w(L) = \{v \,|\, wv \in L\}$. In particular, if $w = a$, with $a \in \Sigma$, we say that $\mathcal{D}_a(L)$ is the left-quotient of L with respect to the symbol a.

3.2. Regular Expressions

Regular expressions are inductively defined over an alphabet Σ, as follows: the constants 0 and 1 are regular expressions; all the symbols $a \in \Sigma$ are regular expressions; if α and β are regular expressions, then their *union* $\alpha + \beta$ and their *concatenation* $\alpha\beta$ are regular expressions as well; finally, if α is a regular expression, then so is its *Kleene star* $\alpha^\star$. The syntactic equality of two regular expressions α and β is denoted by $\alpha \equiv \beta$. The set of all regular expressions over an alphabet Σ is the set RE_Σ. The *length* of a regular expression α is the total number of constants, symbols and operators of α; the *alphabetic length* of a regular expression α is the total number of occurrences of symbols of Σ in α. The previous two measures are denoted by $|\alpha|$ and by $|\alpha|_\Sigma$, respectively.

Regular expressions *denote* regular languages. The language of a regular expression α, denoted $\mathcal{L}(\alpha)$, is inductively defined in the expected way: the languages of the constants 0 and 1 are, respectively, the sets $\emptyset$ and $\{\epsilon\}$; the language of the regular expression a, with $a \in \Sigma$, is the set $\{a\}$; if α and β are regular expressions, then the languages denoted by the expressions $\alpha + \beta$, $\alpha\beta$, and $\alpha^\star$ are, respectively, the languages $\mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, $\mathcal{L}(\alpha)\mathcal{L}(\beta)$, and $\mathcal{L}(\alpha)^\star$. The language of a finite set of regular expressions S is defined by

$$\mathcal{L}(S) = \bigcup_{\alpha_i \in S} \mathcal{L}(\alpha_i).$$

Two regular expressions α and β are said to be equivalent if they denote the same language, and we write $\alpha \sim \beta$ whenever that is the case[1]. Naturally, two sets of regular expressions S_1 and S_2 are equivalent if $\mathcal{L}(S_1) = \mathcal{L}(S_2)$, and

[1] As the reader will notice, we overload the notation "$\sim$" whenever equivalence by means of language equality is considered.

we write $S_1 \sim S_2$. Given a set of regular expressions $S = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ we define

$$\sum S = \alpha_1 + \alpha_2 + \ldots + \alpha_n,$$

whose language is

$$\mathcal{L}\Big(\sum S\Big) = \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2) \cup \cdots \cup \mathcal{L}(\alpha_n).$$

We say that a regular expression α is *nullable* if $\epsilon \in \mathcal{L}(\alpha)$ and *non-nullable* otherwise. Moreover, we consider the Boolean function $\varepsilon(\cdot)$ such that the $\varepsilon(\alpha) = \texttt{true}$ if and only if $\epsilon \in \mathcal{L}(\alpha)$ holds. Nullability extends to sets of regular expressions in a straightforward way: a set S is nullable if $\varepsilon(\alpha)$ evaluates positively, that is, if $\varepsilon(\alpha) = \texttt{true}$ for at least one $\alpha \in S$. We denote the nullability of a set of regular expressions S by $\varepsilon(S)$. Two sets of regular expressions S_1 and S_2 are *equi-nullable* if $\varepsilon(S_1) = \varepsilon(S_2)$. We also consider the *right-concatenation* $S \odot \alpha$ of a regular expression α with a set of regular expressions S, which is defined as follows: $S \odot \alpha = \emptyset$ if $\alpha \equiv 0$, $S \odot \alpha = S$ if $\alpha \equiv 1$, and $S \odot \alpha = \{\beta\alpha \,|\, \beta \in S\}$ otherwise. We usually omit the operator $\odot$ and write $S\,\alpha$ instead.

3.3. Derivatives of Regular Expressions

The notion of *derivative* of a regular expression α was introduced by Brzozowski in the 1960's [**?**], and was motivated by the construction of sequential circuits directly from regular expressions extended with intersection and complement. In the same decade, Mirkin introduced the notion of *prebase* and *base* of a regular expression as a method to construct *non-deterministic finite automata* (NFA) that recognise the corresponding languages [**?**]. Mirkin's definition is a generalisation of Brzozowski's derivatives for NFA and was independently re-discovered almost thirty years later by Antimirov [**?**], who coined it as the *partial derivatives* of a regular expression.

Let α be a regular expression and let $a \in \Sigma$. The *set* $\partial_a(\alpha)$ *of partial derivatives* of the regular expression α with respect to a is inductively defined as follows:

$$\begin{array}{lcl@{\qquad}lcl}
\partial_a(0) & = & \emptyset & \partial_a(\alpha+\beta) & = & \partial_a(\alpha) \cup \partial_a(\beta) \\
\partial_a(1) & = & \emptyset & \partial_a(\alpha\beta) & = & \begin{cases} \partial_a(\alpha)\beta \cup \partial_a(\beta) & \text{if } \varepsilon(\alpha) = \texttt{true}, \\ \partial_a(\alpha)\beta & \text{otherwise.} \end{cases} \\
\partial_a(b) & = & \begin{cases} \{\varepsilon\} & \text{if } a \equiv b, \\ \emptyset & \text{otherwise.} \end{cases} & \partial_a(\alpha^\star) & = & \partial_a(\alpha)\alpha^\star
\end{array}$$

The operation of partial derivation naturally extends to a set of regular expressions S as follows:

$$\partial_a(S) = \bigcup_{\alpha \in S} \partial_a(\alpha).$$

The language of the set of partial derivatives $\partial_a(\alpha)$ is the left-quotient of $\mathcal{L}(\alpha)$, *i.e.*, $\mathcal{L}(\partial_a(\alpha)) = \mathcal{D}_a(\mathcal{L}(\alpha))$. The set of partial derivatives is extended to words in the following way: given a regular expression α and a word $w \in \Sigma^\star$, the partial derivative $\partial_w(\alpha)$ of α with respect to w is defined inductively by $\partial_\varepsilon(\alpha) = \{\alpha\}$, and $\partial_{wa}(\alpha) = \partial_a(\partial_w(\alpha))$. We can use partial derivatives and nullability of regular expressions to determine if a word $w \in \Sigma^\star$ is a member of some language $\mathcal{L}(\alpha)$. For that, it is enough to check the value computed by $\varepsilon(\partial_w(\alpha))$: if $\varepsilon(\partial_w(\alpha)) = \texttt{true}$ then we have $w \in \mathcal{L}(\alpha)$; otherwise, $w \notin \mathcal{L}(\alpha)$ holds.

Example 1. *The word derivative of the regular expression $ab^\star$ with respect to abb is given by the following computation:*

$$\begin{aligned}
\partial_{abb}(\alpha) &= \partial_b(\partial_b(\partial_a(ab^\star))) \\
&= \partial_b(\partial_b(\partial_a(a)b^\star)) \\
&= \partial_b(\partial_b(\{b^\star\})) \\
&= \partial_b(\partial_b(b)b^\star) \\
&= \partial_b(\{b^\star\}) \\
&= \{b^\star\}.
\end{aligned}$$

From the nullability of the resulting set of regular expression $\{b^\star\}$, we easily conclude that $abb \in \mathcal{L}(\alpha)$ since $\varepsilon(b^\star) =$ `true`.

Finally, we present the *set of partial derivatives* of a given regular expression α, which is defined by

$$PD(\alpha) = \bigcup_{w \in \Sigma^\star} (\partial_w(\alpha)).$$

Antimirov proved in [**?**] that given a regular expression α, the set $PD(\alpha)$ is always finite and its cardinality has an upper bound of $|\alpha|_\Sigma + 1$. Champarnaud and Ziadi [**?**] introduced an elegant recursive function for calculating the *support* of a given regular expression α, and from which it is easy to calculate $PD(\alpha)$. The function, denoted by $\pi(\alpha)$, is recursively defined as follows:

$$\begin{array}{lcllcl} \pi(0) & = & \emptyset & \pi(\alpha + \beta) & = & \pi(\alpha) \cup \pi(\beta) \\ \pi(1) & = & \emptyset & \pi(\alpha\beta) & = & \pi(\alpha)\beta \cup \pi(\beta) \\ \pi(a) & = & \{\varepsilon\} & \pi(\alpha^\star) & = & \pi(\alpha)\alpha^\star \end{array}$$

Champarnaud and Ziadi proved that $PD(\alpha) = \{\alpha\} \cup \pi(\alpha)$ holds for all regular expressions α, and once again we conclude that $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$.

4. A Procedure for Regular Expressions Equivalence

In this section we present the decision procedure EQUIVP for deciding regular expression equivalence, and describe its implementation in Coq. The base concepts for this mechanization were already presented in the previous sections. The procedure EQUIVP follows along the lines of the work of Almeida *et. al.* [**?**], and has its origins in the rewrite system proposed by Antimirov and Mosses [**?**] to decide regular expression equivalence using Brzozowski's derivatives.

4.1. Partial Derivatives and Regular Expression Equivalence

Given a regular expression α, it holds that

$$\alpha \sim \varepsilon(\alpha) \cup \bigcup_{a \in \Sigma} a(\sum \partial_a(\alpha)). \tag{2}$$

We overload the notation $\varepsilon(\alpha)$ in the sense that in the current context $\varepsilon(\alpha) = \{\varepsilon\}$ if α is nullable, and $\varepsilon(\alpha) = \emptyset$ otherwise. Following the equivalence (2), checking if $\alpha \sim \beta$ is tantamount to checking the equivalence

$$\varepsilon(\alpha) \cup \bigcup_{a \in \Sigma} a(\sum \partial_a(\alpha)) \sim \varepsilon(\beta) \cup \bigcup_{a \in \Sigma} a(\sum \partial_a(\beta)).$$

This will be an essential ingredient for the decision method because deciding if $\alpha \sim \beta$ resumes to checking if $\varepsilon(\alpha) = \varepsilon(\beta)$ and if $\partial_a(\alpha) \sim \partial_a(\beta)$, for each $a \in \Sigma$. Moreover, since partial derivatives are finite, and since testing if a word $w \in \Sigma^\star$ belongs to $\mathcal{L}(\alpha)$ is equivalent to checking syntactically that $\varepsilon(\partial_w(\alpha)) = \texttt{true}$, we obtain the following equivalence:

$$(\forall w \in \Sigma^\star, \varepsilon(\partial_w(\alpha)) = \varepsilon(\partial_w(\beta))) \leftrightarrow \alpha \sim \beta. \tag{3}$$

In the opposite situation, we can prove that α and β are not equivalent by showing that

$$\varepsilon(\partial_w(\alpha)) \neq \varepsilon(\partial_w(\beta)) \rightarrow \alpha \nsim \beta, \tag{4}$$

for $w \in \Sigma^\star$. Equation (3) can be seen as an iterative process of testing regular expression equivalence by testing the equivalence of their derivatives. Equation (4) can be seen as the point where we find a counterexample of two derivatives during the same iterative process. In the next section we will describe a decision procedure that constructs a *bisimulation* that leads to Equation (3), or that finds a counterexample like in (4) which proves that such a bisimulation cannot exist.

4.2. The Procedure EQUIVP

Recall from the previous section that a proof of the equivalence of regular expressions can be obtained by an iterated process of checking the equivalence of their partial derivatives. Such an iterated process is given in Algorithm 1 presented below. Given two regular expressions α and β the procedure EQUIVP corresponds to the iterated process of deciding the equivalence of their derivatives, in the way noted in Equation (3). The procedure works over pairs of sets of regular expressions (S_α, S_β) such that $S_\alpha = \partial_w(\alpha)$ and $S_\beta = \partial_w(\beta)$, for some word $w \in \Sigma^\star$. From now on we will refer to these pairs of sets of partial derivatives simply by derivatives.

Algorithm 1 The procedure EQUIVP.

Require: $S = \{(\{\alpha\}, \{\beta\})\}$, $H = \emptyset$
Ensure: true or false

```
1:  procedure EQUIVP(S, H)
2:      while S ≠ ∅ do
3:          (S_α, S_β) ← POP(S)
4:          if ε(S_α) ≠ ε(S_β) then
5:              return false
6:          end if
7:          H ← H ∪ {(S_α, S_β)}
8:          for a ∈ Σ do
9:              (S'_α, S'_β) ← ∂_a(S_α, S_β)
10:             if (S'_α, S'_β) ∉ H then
11:                 S ← S ∪ {(S'_α, S'_β)}
12:             end if
13:         end for
14:     end while
15: return true
16: end procedure
```

EQUIVP requires two arguments: a set H that serves as an accumulator for the derivatives (S_α, S_β) already processed; and a set S that serves as a working set that gathers new derivatives (S'_α, S'_β) yet to be processed. The set H ensures the termination of EQUIVP due to the finiteness of the set of partial derivatives. The set S has no influence in the termination argument. When EQUIVP terminates, then it must do so in one of two possible configurations: either the set H contains all the derivatives of α and β and all of them are equi-nullable; or a counterexample (S_α, S_β) such that $\varepsilon(S_\alpha) \neq \varepsilon(S_\beta)$ was found. By Equation (3), we conclude that we have $\alpha \sim \beta$ in the first case, whereas in the second case we must conclude that $\alpha \nsim \beta$. Below, we give an example that shows how EQUIVP handles regular expression equivalence.

Example 2. *Suppose we want to check that $\alpha = (ab)^\star a$ and $\beta = a(ba)^\star$ are equivalent. Considering that s_0 corresponds to the pair $(\{(ab)^\star a\}, \{a(ba)^\star\})$, we must show that*

$$\text{EQUIVP}(\{s_0\}, \emptyset) = \texttt{true}.$$

The computation of EQUIVP *for these particular α and β involves the construction of the new derivatives $s_1 = (\{1, b(ab)^\star a\}, \{(ba)^\star\})$ and $s_2 = (\emptyset, \emptyset)$. We can trace the computation by the following table*

i	S_i	H_i	drvs.
0	$\{s_0\}$	$\emptyset$	$\partial_a(s_0) = s_1, \partial_b(s_0) = s_2$
1	$\{s_1, s_2\}$	$\{s_0\}$	$\partial_a(s_1) = s_2, \partial_b(s_1) = s_0$
2	$\{s_2\}$	$\{s_0, s_1\}$	$\partial_a(s_2) = s_2, \partial_b(s_2) = s_2$
3	$\emptyset$	$\{s_0, s_1, s_2\}$	true

where i is the iteration number, and S_i and H_i are the arguments of EQUIVP *in that same iteration. The trace terminates with $S_2 = \emptyset$ and thus we can conclude that $\alpha \sim \beta$.*

4.3. Implementation

4.3.1. Representation of Derivatives

The main data type used in EQUIVP is the type of pairs of sets of regular expressions. Each pair (S_α, S_β) represents a word derivative $(\partial_w(\alpha), \partial_w(\beta))$, where $w \in \Sigma^\star$. The type of derivatives `Drv` is defined as follows:

```
Record Drv (α β:re) := mkDrv {
 dp :> set re * set re ;
 w  : word ;
 cw : dp = (∂w(α),∂w(β))
}.
```

The type `Drv` is a *dependent record* composed of three parameters: a pair of sets of regular expressions `dp` that corresponds to the actual pair (S_α, S_β); a word `w`; a proof term `cw` that ensures that $(S_\alpha, S_\beta) = (\partial_w(\alpha), \partial_w(\beta))$. The use of the type `Drv` instead of a pair of sets of regular expressions is necessary because EQUIVP's domain is the set of pairs resulting from derivations and not arbitrary pairs of sets of regular expressions on Σ.

The equality relation defined over `Drv` terms considers only the projection `dp`, that is, two terms `d1` and `d2` of type `Drv α β` are equal if `(dp d1) = (dp d2)`. This implies that each derivative will be considered only once along the execution of EQUIVP. If the derivative `d1` is already in the accumulator set, then all derivatives `d2` that are computed afterwards will fail the membership test of line 10 of Algorithm 1. This directly implies the impossibility of the eventual non-terminating computations due to the repetition of derivatives.

As a final remark, the type `Drv` also provides a straightforward way to relate the result of the computation of EQUIVP to the (in-)equivalence of α and β: on one hand, if H is the set returned by EQUIVP, then checking the nullability of its elements is tantamount to proving the equivalence of the corresponding regular expressions, since we expect H to contain all the derivatives; on the other hand, if EQUIVP returns a term `t:Drv α β`, then ε(`t`) = `false`, which implies that the word `w t` is a witness of in-equivalence, and can be presented to the user.

4.3.2. Extended Derivation and Nullability

The notions of derivative with respect to a symbol and with respect to a word are also extended to the type `Drv`. The derivation of a value of type `Drv α β` representing the pair (S_α, S_β) is obtained by calculating the derivative $\partial_a(S_\alpha, S_\beta)$, updating the word w, and also by automatically building the associated proof term for the parameter `cw`. The function implementing the derivation of `Drv` terms, and its extension to sets of `Drv` terms, and to the derivation with respect to a word, are given below[2]. Note that $\partial_a(S_\alpha, S_\beta) = (\partial_a(S_\alpha), \partial_a(S_\beta))$, and therefore $\partial_a(\partial_w(\alpha), \partial_w(\beta)) = (\partial_{wa}(\alpha), \partial_{wa}(\beta))$.

```
Definition Drv_pdrv(α β:re)(x:Drv α β)(a:A) : Drv α β.
refine(match x with mkDrv α β K w P ⇒ mkDrv α β (pdrvp K a) (w++[a]) _ end).
abstract((* Proof that ∂a(∂w(α),∂w(β)) = (∂wa(α),∂wa(β)) *)).
Defined.

Definition Drv_pdrv_set(x:Drv α β)(s:set A) : set (Drv α β) := fold (fun y:A ⇒ add (Drv_pdrv x y))
     s ∅.

Definition Drv_wpdrv (α β:re)(w:word) : Drv α β.
refine(mkDrv α β (∂w(α),∂w(β)) w _).
abstract((* Proof that (∂w(α),∂w(β)) = (∂w(α),∂w(β)) *)).
Defined.
```

We also extend the notion of nullable regular expression to terms of type `Drv`, and to sets of values of type `Drv`. Checking the nullability of a `Drv` term denoting the pair (S_α, S_β) is tantamount at checking that $\varepsilon(S_\alpha) = \varepsilon(S_\beta)$.

```
Definition c_of_rep(x:set re * set re) := Bool.eqb (c_of_re_set (fst x)) (c_of_re_set (snd x)).

Definition c_of_Drv(x:Drv α β) := c_of_rep (dp x).

Definition c_of_Drv_set (s:set (Drv α β)) : bool := fold (fun x ⇒ andb (c_of_Drv x)) s true.
```

[2]For the sake of clarity we briefly describe de purpose of the tactic `abstract` that is used for building these definition. The tactic `abstract` saves the proof of the goal under consideration as an auxiliary lemma. This makes the actual proof term opaque in the context that `abstract` is used, which makes computation much more efficient in terms containing proofs as (dependent) arguments.

All the previous functions were implemented using the proof mode of Coq instead of trying a direct definition, that is, we used tactics to construct the definitions instead of providing the lambda term that implements them, which in this case facilitated the implementation. In particular, in this way we are able to wrap the proofs in the tactic **abstract**, which dramatically improves the performance of the computation.

4.3.3. Computation of New Derivatives

The *while-loop* of EQUIVP – lines 2 to 14 of Algorithm 1 – describes the process of testing the equivalence of the derivatives of two given regular expressions α and β. In each iteration, either a witness of inequivalence is found, or new derivatives (S_α, S_β) are computed and the sets S and H are updated accordingly. The expected behaviour of each iteration of the loop is implemented by the function **step**, presented below, and which also corresponds to the *for-loop* from lines 8 to 13 of Algorithm 1.

```
Definition step (H S:set (Drv α β))(Σ:set A) : ((set (Drv αβ) * set (Drv α β)) * step_case α β)
    :=
 match choose S with
 |None ⇒  ((H,S),termtrue α β H)
 |Some (S_α,S_β) ⇒
   if c_of_Drv _ _ (S_α,S_β) then
    let H' := add (S_α,S_β) H in
     let S' := remove (S_α,S_β) S in
      let ns := Drv_pdrv_set_filtered α β (S_α,S_β) H' Σ in
       ((H',ns ∪ S'),proceed α β)
   else
    ((H,S),termfalse α β (S_α,S_β))
 end.
```

The **step** function proceeds as follows: it obtains a pair (S_α, S_β) from the set S, and tests it for equi-nullability. If S_α and S_β are not equi-nullable, then **step** returns a pair $((H,S),$**termfalse** $\alpha\,\beta\,(S_\alpha, S_\beta))$, that serves as a witness of $\alpha \nsim \beta$. If, on the contrary, S_α and S_β are equi-nullable, then **step** generates a new set of derivatives by the symbols $a \in \Sigma$, $(S'_\alpha, S'_\beta) = (\partial_a(S_\alpha), \partial_a(S_\beta))$, such that (S'_α, S'_β) are not elements of $\{(S_\alpha, S_\beta)\} \cup H$. These new derivatives are added to S and (S_α, S_β) is added to H. The computation of new derivatives is performed by the function **Drv_pdrv_set_filtered**, defined as follows:

```
Definition Drv_pdrv_set_filtered(x:Drv α β)(H:set (Drv α β))(sig:set A) : set (Drv α β) :=
  filter (fun y ⇒ negb (y ∈ H)) (Drv_pdrv_set x sig).
```

Note that this is precisely what prevents the whole process from entering potential infinite loops, since each derivative is considered only once during the execution of EQUIVP and because the number of derivatives is always finite.

Finally, we present the type **step_case** below. This type is built from three constructors: the constructor **proceed** represents the fact that there is not yet information that allows to decide if the regular expressions under consideration are equivalent or not; the constructor **termtrue** indicates that no more elements exist in S, and that H should contain all the derivatives; finally, the constructor **termfalse** indicates that **step** has found a proof of in-equivalence of the regular expressions under consideration.

```
Inductive step_case (α β:re) : Type :=
|proceed   : step_case α β
|termtrue  : set (Drv α β) → step_case α β
|termfalse : Drv α β → step_case α β.
```

4.3.4. Termination

Clearly, the procedure EQUIVP is general recursive. This means that the procedure's iterative process cannot be directly encoded in Coq's underlying type system. Therefore, we have devised a well-founded relation establishing a recursive measure that defines the course-of-values that makes EQUIVP terminate. This well-founded relation will be the structural recursive argument for our encoding of EQUIVP. The decreasing measure (of the recursive calls) used in EQUIVP is defined as follows: in each recursive call, the cardinality of the accumulator set H increases by one unit due to the computation of **step**. The maximum size that H can reach is upper bounded by $2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1$ due

to the upper bounds of the cardinalities of both $PD(\alpha)$ and $PD(\beta)$, the cardinality of the cartesian product, and the cardinality of the powerset. Therefore, if `step` $H\ S\ \_ = (H', \_, \_)$, then the following relation

$$(2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - |H'| < (2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - |H|, \tag{5}$$

holds. In terms of its implementation in Coq, we first define and prove the following:

```
Definition lim_cardN (z:N) : relation (set A) :=
  fun x y:set A ⇒ nat_of_N z − (cardinal x) < nat_of_N z − (cardinal y).
Lemma lim_cardN_wf : ∀ z, well_founded (lim_cardN z).
```

Next, we establish the upper bound of the number of derivatives, and define the relation `LLim` that is the relation that actually implements (5). The encoding in Coq goes as follows:

```
Definition MAX_re(α:re) := |α|_Σ + 1.
Definition MAX(α β:re) := (2^MAX_re(α) × 2^MAX_re(β)) + 1.
Definition LLim(α β:re) := lim_cardN (Drv α β) (MAX α β).

Theorem LLim_wf(α β:re) : well_founded (LLim α β).
```

4.3.5. The Iterator

We now present the development of a recursive function that implements the main loop of Algorithm 1. This recursive function is an iterator that calls the function `step` a finite number of times starting with two initial sets S and H. This iterator, named `iterate`, is defined as follows:

```
Function iterate(α β:re)(H S:set (Drv α β))(sig:set A)(D:DP α β H S){wf (LLim α β) H}: term_cases
      α β :=
  let ((H',S',next) := step H S in
   match next with
    |termfalse x ⇒ NotOk α β x
    |termtrue  h ⇒ Ok α β h
    |proceed     ⇒ iterate α β H' S' sig (DP_upd α β H S sig D)
  end.
Proof.
 abstract(apply DP_wf).
 exact(guard α β 100 (LLim_wf α β)).
Defined.
```

The function `iterate` is recursively decreasing on a proof that `LLim` is well-founded. The type annotation `wf LLim` $\alpha\ \beta$ adds this information to the inner mechanisms of `Function`, so that `iterate` is constructed in such a way that Coq's type-checker accepts it. The proof that `LLim` is well-founded is computed by the function `guard`. This function was introduced by Barras and Gonthier[3] and builds a term made of 2^{100} constructors `Acc` on the front of the actual proof of the well-foundness of `LLim`, which turns out to be also a proof of the well-foundness of `LLim` as well. The number of such constructors may vary, and we have chose this because it is sufficiently large to cover our practical experiments.

Moreover, in order to validate `LLim` along the computation of `iterate`, we must provide evidence that the sets S and H remain disjoint in all the recursive calls of `iterate`. The last parameter of the definition of `iterate`, D, has the type `DP` which packs together a proof that the sets H and S are disjoint (in all recursive calls) and that all the elements in the set H are equi-nullable. The proof that S and H are disjoint is needed to ensure that `LLim` is valid in all recursive calls, whereas the proof that all the elements of H are equi-nullable is required to prove the equivalence of the regular expressions under consideration, following Equation (3). The definition of type `DP` is the following:

```
Inductive DP (α β:re)(H S: set (Drv α β)) : Prop :=
| is_dp : H ∩ S = ∅ → c_of_Drv_set α β H = true → DP α β H S.
```

In the definition of the recursive branch of `iterate`, the function `DP_upd` is used to build a new term of type `DP` that proves that the updated sets H and S remain disjoint, and that all the elements in H remain equi-nullable.

[3]This idea was proposed by Barras, and then improved by Gonthier in a discussion that occured in the Coq-Club mailing list.

```
Lemma DP_upd : ∀ (α β:re)(H S : set (Drv α β)) (sig : set A),
  DP α β H S → DP α β (fst (fst (step α β H S sig))) (snd (fst (step α β H S sig))).
```

The output of `iterate` is a value of type `term_cases`, which is defined as follows:

```
Inductive term_cases (α β:re) : Type :=
|Ok : set (Drv α β) → term_cases α β
|NotOk : Drv α β → term_cases α β.
```

The type `term_cases` is made of two constructors that determine what possible outcome we can obtain from computing `iterate`: either it returns a set S of derivatives, packed in the constructor `Ok`, or it returns a sole pair (S_α, S_β), packed in the constructor `NotOk`. The first should be used to prove equivalence, whereas the second should be used for exhibiting a witness of in-equivalence.

The `Function` command produces proof obligations that have to be discharged in order to be accepted by Coq's type checker. One of the proof obligations generated by `iterate` is that, when performing a recursive call, the new cardinalities of H and S still satisfy the underlying well-founded relation. The lemma `DP_wf` serves this purpose and is defined as follows:

```
Lemma DP_wf : ∀ (α β:re)(H S : set (Drv α β)) (sig : set A),
  DP α β H S → snd (step α β H S sig) = proceed α β → LLim α β (fst (fst (step α β H S sig))) H.
```

The second proof obligation generated by `Function` is discharged by the exact term that represents the well-founded relation under consideration. In the code below we give the complete definition of EQUIVP. The function `equivP` is simply a wrapper defined over `iterate`: it establishes the correct input for the arguments H and S and pattern matches over the result of `iterate`, returning the expected Boolean value.

```
Definition equivP_aux(α β:re)(H S:set(Drv α β))(Σ:set A)(D:DP α β H S):=
 let H' := iterate α β H S Σ D in match H' with | Ok _ ⇒true | NotOk _ ⇒false end.

Definition mkDP_ini : DP α β ∅ {Drv_1st α β}.
abstract(constructor;[split;intros;try(inversion H)|vm_compute];reflexivity).
Defined.

Definition equivP (α β:re) := equivP_aux α β ∅ {Drv_1st α β} (setSy α ∪ setSy β) (mkDP_ini α β).
```

The function `mkDP_ini` builds the term of type `DP` that ensures that $\{(\{\alpha\}, \{\beta\})\} \cap \emptyset = \emptyset$ and that $\varepsilon(\emptyset) =$ `false` holds. The final decision procedure, `equivP`, calls the function `equivP_aux` with the adequate arguments, and the function `equivP_aux` simply pattern matches over a term of `term_cases` and returns a Boolean value accordingly.

We note that in the definition of `equivP` we instantiate the parameter representing the input alphabet by the union of two sets, both computed by the function `setSy`. This function returns the set of all symbols that exist in a given regular expression. It turns out that for deciding regular expressions (in)equivalence we need not to consider a fixed alphabet Σ, since only the symbols that exist in the regular expressions being tested are important and used in the derivations. In fact, the input alphabet can even be an infinite alphabet.

4.4. Correctness

In order to prove the correctness of `equivP` with respect to language equivalence, we proceed as follows. Suppose that `equivP` $\alpha\ \beta$ = `true`. To prove that this implies regular expression equivalence we must prove that the set of all the derivatives is computed by the function `iterate`, and also that all the elements of that set are equi-nullable. This leads to (3), which in turn implies language equivalence.

To prove that `iterate` computes the desired set of derivatives we must show that, in each of its recursive calls, the accumulator set H keeps a set of values whose derivatives have been already computed (they are also in H), or that such derivatives are still in the working set S, waiting to be selected for further processing. This property is formally defined in Coq as follows:

```
Definition invP (α β:re)(H S:set (Drv α β))(Σ:set A) := ∀ x:Drv α β, x ∈ H → ∀ a:A, a ∈ Σ → (
    Drv_pdrv α β x a) ∈ (H∪S).
```

We must prove that `invP` is an invariant of `iterate`. This requires a proof asserting that `invP` is satisfied by the computation of `step` as stated in the next proposition.

Proposition 1. *Let α and β be two regular expressions, and let S, S', H, and H' be finite sets of values of type* `Drv` α β. *If* `invP`*(H,S) holds and if* `step` α β H S Σ = ((H',S'),`proceed` α β), *then* `invP`*(H',S') also holds.*

The next step is to prove that `invP` is an invariant of `iterate`. This proof indeed shows that if `invP` is satisfied in all the recursive calls of `iterate`, then this function must return a value `Ok` α β H' and `invP` H' $\emptyset$ must be satisfied. This is stated in the next proposition.

Proposition 2. *Let α and β be two regular expressions. Let S, H, and H' be finite sets of values of type* `Drv` α β, *and let Σ be an alphabet. If* `invP`*(α,β,H,S) holds, and if* `iterate` α β H S Σ D = `Ok` α β H', *then* `invP`*(α,β,H',$\emptyset$) also holds.*

In Coq, the two previous propositions are defined as follows:

```
Lemma invP_step : ∀ α β H S Σ,
  invP α β H S Σ → invP α β (fst (fst (step α β H S Σ))) (snd (fst (step α β H S Σ))) Σ.

Lemma invP_iterate : ∀ α β H S Σ D x,
  invP α β H S Σ → iterate α β H S Σ D = Ok α β x → invP α β x ∅.
```

The Propositions 1 and 2 are not enough to prove the correctness of `equivP` with respect to language equivalence. We still have to prove that the derivatives that are computed are all equi-nullable, and also prove that the pair containing the regular expressions being tested for equivalence is in the set of derivatives returned by `iterate`. For that, we strengthen the invariant `invP` with as follows:

```
Definition invP_final(α β:re)(H S:set (Drv α β))(s:set A) :=
  (Drv_1st α β) ∈ (H ∪ S) ∧
   ( ∀ x:Drv α β, x ∈ (H ∪ S) → c_of_Drv α β x = true) ∧
      invP α β H S s.
```

We start by proving that, if we are testing $\alpha \sim \beta$, then the pair $\{(\{\alpha\}, \{\beta\})\}$ is an element of the set returned by `iterate`. But first we must introduce two generic properties that will allow us to conclude that.

Proposition 3. *Let α and β be two regular expressions. Let H, H', and S' be sets of values of type* `Drv` α β. *Finally, let Σ be an alphabet, and let D be a value of type* `DP` α β H S. *If it holds that* `iterate` α β H S Σ D = `Ok` α β H', *then it also holds that $H \subseteq H'$.*

Corollary 1. *Let α and β be two regular expressions. Let γ be a value of type* `Drv` α β. *Let H, H', and S' be sets of values of type* `Drv` α β. *Finally, let Σ be an alphabet, and let D be a value of type* `DP` α β H S. *If it holds that* `iterate` α β H S Σ D = `Ok` α β H' *and that* `choose` S = `Some` γ, *then it also holds that $\{\gamma\} \cup H \subseteq H'$.*

From Proposition 3 and Corollary 1 we are able to prove that the original pair is always returned by the `iterate` function, whenever it returns a value `Ok` α β H.

Proposition 4. *Let α and β be two regular expressions, let H' be a finite set of values of type* `Drv` α β, *let Σ be an alphabet, and let D be a value of type* `DP` α β $\emptyset$ $\{(\{\alpha\}, \{\beta\})\}$. *Hence,*

$$\texttt{iterate}\ \alpha\ \beta\ \emptyset\ \{(\{\alpha\}, \{\beta\})\}\ \Sigma\ D = \texttt{Ok}\ \alpha\ \beta\ H' \rightarrow (\{\alpha\}, \{\beta\}) \in H'.$$

Now, we proceed in the proof by showing that all the elements of the set packed in a value `Ok` α β H' enjoy equi-nullability. This is straightforward, due to the last parameter of `iterate`. Recall that a value of type `DP` always contains a proof of that fact.

Proposition 5. *Let α and β be two regular expressions. Let H, H', and S' be set of values of type* `Drv` α β. *Finally, let Σ be an alphabet and D be a value of type* `DP` α β H S. *If it holds that* `iterate` α β H S Σ D = `Ok` α β H', *then it also holds that* $\forall \gamma \in H',\ \varepsilon(\gamma) =$ `true`.

Using Propositions 4 and 5 we can establish the intermediate result that will take us to prove the correctness of `equivP` with respect to language equivalence.

Proposition 6. *Let α and β be two regular expressions. Let H, H', and S' be set of values of type* `Drv` α β. *Finally, let Σ be an alphabet, and let D be a value of type* `DP` α β H S. *If it holds that* `iterate` α β H S Σ D = `Ok` α β H', *then* `invP_final` α β H' $\emptyset$.

The last intermediate logical condition that we need to establish is that `invP_final` implies language equivalence, when instantiated with the correct parameters. The following lemma gives us exactly that.

Proposition 7. *Let α and β be two regular expressions. Let H' be a set of values of type* `Drv` α β. *If it holds that* `invP_final` α β H' $\emptyset$ (`setSy` $\alpha \cup$ `setSy` β), *then α and β are equivalent.*

Finally, we can state the theorem that ensures that if `equivP` returns `true`, then we have the equivalence of the regular expressions under consideration.

Lemma 1. *Let α and β be two regular expressions. Thus, if* `equivP` α β = `true` *holds, then α and β are equivalent.*

4.5. Completeness

To prove that `equivP` α β = `false` implies the inequivalence of two given regular expressions α and β, we must prove that the value γ in the term `NotOk` α β γ returned by `iterate` α β S H Σ D is a witness that there is a word $w \in \Sigma^\star$ such that $w \in \mathcal{L}(\alpha)$ and $w \notin \mathcal{L}(\beta)$, or the other way around. This leads us to the following lemma about `iterate`.

Proposition 8. *Let α and β be regular expressions, let S and H be set of values of type* `Drv` α β. *Let Σ be an alphabet, γ a term of type* `Drv`, *and D a value of type* `DP` α β S H. *If* `iterate` α β S H Σ D = `NotOk` α β γ, *then, considering that γ represents the pair of sets of regular expressions (S_α, S_β), we have $\varepsilon(S_\alpha) \neq \varepsilon(S_\beta)$.*

Next, we just need to prove that the pair in the value returned by `iterate` does imply inequivalence.

Proposition 9. *Let α and β be regular expressions, let S and H be set of values of type* `Drv` α β, *let Σ be an alphabet, and let D be a value of type* `DP` α β S H. *Hence, if* `iterate` α β S H Σ D = `NotOk` α β γ *then α and β are not equivalent.*

The previous two lemmas allow us to conclude that `equivP` is correct with respect to the in-equivalence of regular expressions.

Lemma 2. *Let α and β be two regular expressions. Hence, if* `equivP` α β = `false` *then α and β are not equivalent.*

4.6. Tactics and Automation

In this section we describe two Coq proof tactics that are able to automatically prove the (in)equivalence of regular expressions, as well as relational algebra equations.

4.6.1. Tactic for Deciding Regular Expressions Equivalence

The expected way to prove the equivalence of two regular expressions α and β, using our development, can be summarised as follows: first we look into the goal, which must be of the form $\alpha \sim \beta$ or $\alpha \nsim \beta$; secondly, we transform such goal into the equivalent one that is formulated using `equivP`, on which we can perform computation. The main tactic, `dec_re`, pattern matches on the goal and decides whether the goal is an equivalence, an in-equivalence, or a subset relation. In the former two cases, `dec_re` applies the corresponding auxiliary tactics, `re_inequiv` or `re_equiv`, and reduces the equivalence into a call to `equivP`, and then performs computation in order to try to solve the goal by reflexivity. In the case of a goal representing a subset relation, `dec_re` first changes it into an equivalence (since we know that $\alpha \leq \beta = \alpha + \beta \sim \beta$) and, after that, call the auxiliary tactic `re_equiv` to prove the goal.