Deciding Kleene algebra terms equivalence in Coq

Nelma Moreira, David Pereira, Simão Melo de Sousa

A B S T R A C T

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 on 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) [1]. 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 [2] and Esterel [3]. More recently, regular expressions have been successfully applied in the runtime verification of programs [4,5].

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 [6] provided a first formalization of the Kleene theorem for regular languages [1] within the Coq proof assistant [7]. Höfner and Struth [8] investigated the automated reasoning in variants of Kleene algebras with Prover9 and Mace4 [9]. Pereira and Moreira [10] implemented in Coq an abstract specification of *Kleene algebra with tests* (KAT) [11] 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 [12]. In this paper we describe the mechanization of a decision procedure based on partial derivatives that was

proposed by Almeida et al. [13], and that is a functional variant of the rewrite system introduced by Antimirov and Mosses in [14]. 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 [15] presented an almost linear algorithm for equivalence of two DFAs. By transforming regular expressions into equivalent DFAs, Hopcroft and Karp's method can be used for regular expressions equivalence. A comparison of that method with the method proposed here is discussed by Almeida et al. [16,17]. There it is conjectured that a direct method should perform better on average, and that is corroborated by theoretical studies based on analytic combinatorics [18]. Hopcroft and Karp's method was used by Braibant and Pous [19] to formally verify Kozen's proof of the completeness of Kleene algebra [20] in Coq.

Independently of the work presented here, Coquand and Siles [21] mechanically verified an algorithm for deciding regular expression equivalence based on Brzozowski's derivatives [22] and an inductive definition of finite sets called Kuratowski-finite sets. Based on the same notion of derivative, Krauss and Nipkow [23] provide an elegant and concise formalization of Rutten's co-algebraic approach of regular expression equivalence [24] in the Isabelle proof assistant [25], but they do not address the termination of the decision procedure. Komendantsky provides a novel functional construction of the *partial derivative automaton* [26], and also made contributions [27] to the mechanization of concepts related to Mirkin's construction [28] of that automata. More recently, Andrea Asperti formalized a decision procedure for the equivalence of *pointed regular expressions* [29], that is both compact and efficient.

Besides avoiding the need for building DFAs, our use of partial derivatives also avoids the necessary normalization 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 [19], 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 on 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 provably terminating functions. Finally, a formal proof must be provided in order to ensure that the functional behavior of the implemented procedure is correct w.r.t. 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 formalize in order to implement the decision procedure; in Section 4 we describe the formalization of the decision procedure, its proofs of correctness and completeness, and comment on the procedure's computational efficiency; in Section 5 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 6 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 [30,31], and the corresponding development in Coq is available at [32].

2. An overview of the Coq proof assistant

The Coq proof assistant [7] is an implementation of Paulin-Mohring's *Calculus of Inductive Constructions* (CIC) [33]. The CIC is a rich typed λ-calculus that features polymorphism, dependent types, and that extends Coquand and Huet's *Calculus of Constructions* (CC) [34] with very expressive (co-)inductive types.

The CIC is built upon the *Curry–Howard Isomorphism* (CHI) *programs-as-proofs* principle [35], 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 extend 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 :=
| O : 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 than `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 [7]; 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 $(x, x_0, x_1, x_2, \ldots)$ 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` [36] 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 [7], of Chlipala [37], and of Pierce et al. [38].

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 [39] or the one by Kozen [40]. The encoding in Coq of the several definitions that we are about to introduce can be seen in [31].

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 $L_1 L_2 = \{wu \mid w \in L_1 \wedge u \in L_2\}$. 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 $\mathcal{D}_w(L) = \{v \mid wv \in L\}$. 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 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

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

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 $\varepsilon(\alpha) = \mathtt{true}$ 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 $\varepsilon(\alpha) = \mathtt{true}$ 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 $S \odot \alpha = \{\beta\alpha \mid \beta \in S\}$ otherwise. We usually omit the operator $\odot$ and write $S\alpha$ instead.

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

3.3. Derivatives of regular expressions

The notion of *derivative* of a regular expression α was introduced by Brzozowski in the 1960s [22], 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 recognize the corresponding languages [28]. Mirkin's definition is a generalization of Brzozowski's derivatives for NFA and was independently re-discovered almost thirty years later by Antimirov [41], 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:

$$\partial_a(0) = \emptyset \qquad \partial_a(\alpha + \beta) = \partial_a(\alpha) \cup \partial_a(\beta)$$

$$\partial_a(1) = \emptyset \qquad \partial_a(\alpha\beta) = \begin{cases} \partial_a(\alpha)\beta \cup \partial_a(\beta) & \text{if } \varepsilon(\alpha) = \texttt{true}, \\ \partial_a(\alpha)\beta & \text{otherwise.} \end{cases}$$

$$\partial_a(b) = \begin{cases} \{\varepsilon\} & \text{if } a \equiv b, \\ \emptyset & \text{otherwise.} \end{cases} \qquad \partial_a(\alpha^\star) = \partial_a(\alpha)\alpha^\star$$

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:

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

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

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

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

Antimirov proved in [41] 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 [42] 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:

$$\pi(0) = \emptyset \qquad \pi(\alpha + \beta) = \pi(\alpha) \cup \pi(\beta)$$
$$\pi(1) = \emptyset \qquad \pi(\alpha\beta) = \pi(\alpha)\beta \cup \pi(\beta)$$
$$\pi(a) = \{\varepsilon\} \qquad \pi\big(\alpha^\star\big) = \pi(\alpha)\alpha^\star$$

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. [13], and has its origins in the rewrite system proposed by Antimirov and Mosses [14] 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

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

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

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

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 $\varepsilon(\partial_w(\alpha)) = \mathtt{true}$, we obtain the following equivalence:

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

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

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

for $w \in \Sigma^\star$. Eq. (3) can be seen as an iterative process of testing regular expression equivalence by testing the equivalence of their derivatives. Eq. (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 Eq. (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. 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 Eq. (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 iff $\alpha \sim \beta$, false otherwise

```
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 Eq. (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

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

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).
```

[2] For the sake of clarity we briefly describe the purpose of the tactic **abstract** that is used for building these definitions. 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 where **abstract** is used, which makes computation much more efficient in terms containing proofs as (dependent) arguments.

```
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 to 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.
```

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 behavior 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 (Sα, Sβ) ⇒
   if c_of_Drv _ _ (Sα, Sβ) then
    let H' := add (Sα, Sβ) H in
     let S' := remove (Sα, Sβ) S in
      let ns := Drv_pdrv_set_filtered α β (Sα, Sβ) H' Σ in
       ((H',ns ∪ S'),proceed α β)
   else
    ((H,S),termfalse α β (Sα, Sβ))
 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 $\texttt{step}\, H\, S\, \_ = (H', \_, \_)$, then the following relation

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

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** α β) H 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 chosen 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 Eq. (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.

```
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))).
```

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

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 $\texttt{equivP}\ \alpha\ \beta = \texttt{true}$. 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`(α,β,H,S) *holds and if* `step` α β H S $\Sigma = ((H',S'),$ `proceed` α $\beta)$, *then* `invP`(α,β,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`$(\alpha,\beta,H',\emptyset)$ 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`$(\alpha,\beta,H',\emptyset)$ *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 ∅.
```

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` 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 $\beta = $ `true` *holds, then α and β are equivalent.*

4.5. Completeness

To prove that `equivP` $\alpha\ \beta =$ `false` implies the inequivalence of two given regular expressions α and β, we must prove that the value γ in the term `NotOk` $\alpha\ \beta\ \gamma$ returned by `iterate` $\alpha\ \beta\ S\ H\ \Sigma\ D$ 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` $\alpha\ \beta$. *Let Σ be an alphabet, γ a term of type* `Drv`, *and D a value of type* `DP` $\alpha\ \beta\ S\ H$. *If* `iterate` $\alpha\ \beta\ S\ H\ \Sigma\ D =$ `NotOk` $\alpha\ \beta\ \gamma$, *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` $\alpha\ \beta$, *let Σ be an alphabet, and let D be a value of type* `DP` $\alpha\ \beta\ S\ H$. *Hence, if* `iterate` $\alpha\ \beta\ S\ H\ \Sigma\ D =$ `NotOk` $\alpha\ \beta\ \gamma$ *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` $\alpha\ \beta =$ `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 summarized 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.

4.6.2. Tactic for deciding relation algebra equations

One of the applications of our development is the automation of proofs for equations of the algebra of relations defined over a unique datatype (i.e., endorelations). The idea of using regular expression equivalence to decide relation equations, based on derivatives, was first pointed out by Nipkow and Krauss [23]. In this section we adapt their idea to our development and give the definitions that are needed for the reflection based tactic that we have implemented. The tactic essentially reduces an equation between relations to the equivalence of two regular expressions. Then it is enough to apply our tactic and decide (in)equivalence.

A formalization of relations is already provided in Coq's standard library, but no support for automation is given. Here, we consider the following encoding of binary relations:

```
Parameter B : Type.

Definition EmpRel : relation B := fun _ _:B ⇒ False.

Definition IdRel : relation B := fun x y :B ⇒x = y.

Definition UnionRel (R S:relation B) : relation B := union _ R S.

Definition CompRel (R S: relation B): relation B := fun i k ⇒ ∃ j, R i j ∧ S j k.

Inductive TransRefl (R:relation B) : relation B :=
| trr : ∀ x, TransRefl R x x
| trt : ∀ x y, R x y → ∀ z, TransRefl R y z →TransRefl R x z.

Definition rel_eq (R S:relation B) : Prop := same_relation B R S.
```

The definitions `EmpRel`, `IdRel`, `UnionRel`, `CompRel` and `TransRefl` represent, respectively, the empty relation, the identity relation, the union of relations, the composition of relations, and the transitive and reflexive closure of a relation. If R_1 and

R_2 are relations, denote the previous constants and operations on relations by $\emptyset$, $\mathcal{I}$, $R_1 \cup R_2$, $R_1 \circ R_2$, and $R^\star$, respectively. The definition `rel_eq` corresponds to the equivalence of relations, and is denoted by $R_1 \sim_{\mathcal{R}} R_2$.

Regular expressions can be easily transformed into binary relations. For that, we need to consider a function v that maps each symbol of the alphabet under consideration into a relation. With this function, we can define another recursive function that, by structural recursion on a given regular expression α, computes the corresponding relation. Such a function is defined in Coq as follows:

```
Fixpoint reRel(v:nat→relation B)(α:re) : relation B :=
 match r with
 | 0 ⇒EmpRel
 | 1 ⇒IdRel
 | 'a ⇒ v a
 | x + y ⇒UnionRel (reRel v x) (reRel v y)
 | x · y ⇒CompRel (reRel v x) (reRel v y)
 | x* ⇒TransRefl (reRel v x)
 end.
```

The following example shows how `reRel` is, in fact, a function that generates a relation considering a particular definition of the function `v`.

Example 3. Let $\Sigma = \{a, b\}$, let R_a and R_b be two binary relations over a set of values of type `B`, and let $\alpha = a(b+1)$ be a regular expression. Moreover, let v be a function that maps the symbols a to the relation R_a, and that maps b to the relation R_b. The computation of `reRel` α `v` gives the relation $R_a \circ (R_b \cup \mathcal{I})$, and can be described as follows:

$$\begin{aligned}
\texttt{reRel}\ \alpha\ v &= \texttt{reRel}\ (a(b+1))\ v \\
&= \texttt{CompRel}\ (\texttt{reRel}\ a\ v)\ (\texttt{reRel}\ (b+1)\ v) \\
&= \texttt{CompRel}\ R_a\ (\texttt{reRel}\ (b+1)\ v) \\
&= \texttt{CompRel}\ R_a\ (\texttt{UnionRel}\ (\texttt{reRel}\ b\ v)\ (\texttt{reRel}\ 1\ v)) \\
&= \texttt{CompRel}\ R_a\ (\texttt{UnionRel}\ R_b\ (\texttt{reRel}\ 1\ v)) \\
&= \texttt{CompRel}\ R_a\ (\texttt{UnionRel}\ R_b\ \texttt{IdRel})
\end{aligned}$$

Naturally, a word $w = a_1 a_2 \ldots a_n$ can also be interpreted as a relation, namely, the composition of the relations $v(a_i)$, where v is the function that maps a symbol a_i to a relation R_{a_i}, with $1 \leq i \leq n$. Such interpretation of words as relations is implemented as follows:

```
Fixpoint reRelW (v:A →relation B)(w:word) : relation B :=
 match w with
 | [] ⇒ IdRel
 | x::xs ⇒CompRel (v x) (reRelW v xs)
 end.
```

Example 4. Let $\Sigma = \{a, b\}$, let R_a and R_b be two binary relations over a set of values of type `B`. Let w be a word defined by $w = abba$. Moreover, let v be a function that maps the symbol a to the relation R_a, and that maps b to the relation R_b. The function `reRelW` v w yields the relation $R_a \circ R_b \circ R_b \circ R_a \circ \mathcal{I}$, and its computation is summarized as

$$\begin{aligned}
\texttt{reRelW}\ v\ w &= \texttt{reRelW}\ v\ abba \\
&= R_a \circ (\texttt{reRelW}\ v\ bba) \\
&= R_a \circ R_b \circ (\texttt{reRelW}\ v\ ba) \\
&= R_a \circ R_b \circ R_b \circ (\texttt{reRelW}\ v\ a) \\
&= R_a \circ R_b \circ R_b \circ R_a \circ (\texttt{reRelW}\ v\ \epsilon) \\
&= R_a \circ R_b \circ R_b \circ R_a \circ \mathcal{I}.
\end{aligned}$$

Now we connect the previous interpretations to regular expression equivalence and relation equivalence. First we present the following inductive predicate, `ReL`, which defines a relation that contains all the pairs (a, b) such that a and b are related by the interpretation of `reRelW` over the elements of the language denoted by some regular expression α.

```
Inductive ReL (v:A →relation B) : re → relation B :=
| mkRel : ∀ α:re, ∀ w:word, w ∈ re2rel α → ∀ a b, (reRelW v w) a b →ReL α a b.
```

If two regular expressions α and β are equivalent, then their interpretations in terms of the function `reRelW` must necessarily lead to the equivalence of the values returned by `reRelW` v α w and by `reRelW` v α w, for $w \in \Sigma^\star$. This means that the pairs (a, b) in `ReL` v α and in `ReL` v β must also be the same. This takes us to the main property that is necessary to establish to rely on regular expression equivalence to decide equations involving relations.

Lemma 3. *Let α and β be regular expressions. Let v be a function that maps symbols to relation. Hence, $\alpha \sim \beta \rightarrow$* `ReL` v $\alpha \sim_{\mathcal{R}}$ `ReL` v β.

Table 1
Performance results of the tactic **dec_re**.

k	$n = 25$		$n = 50$		$n = 100$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
10	0.151	0.026	0.416	0.032	1.266	0.044
20	0.176	0.042	0.442	0.058	1.394	0.072
30	0.183	0.050	0.478	0.074	1.338	0.097
40	0.167	0.058	0.509	0.088	1.212	0.108
50	0.167	0.065	0.521	0.097	1.457	0.141
k	$n = 250$		$n = 500$		$n = 1000$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
10	12.049	0.058	38.402	0.081	–	0.125
20	5.972	0.083	24.674	0.105	58.904	0.181
30	5.511	0.128	17.408	0.157	43.793	0.226
40	5.142	0.147	19.961	0.181	43.724	0.271
50	5.968	0.198	17.805	0.203	46.037	0.280

In order to use Lemma 3 to decide relation equivalence, we must relate **ReL** and **reRel**. But, as the next lemma show, both these notions end up being equivalent relations.

Lemma 4. *Let α be a regular expression, and let ν be a function mapping symbols of the alphabet to relations. Thus* **reRel** $\nu\ \alpha \sim_{\mathcal{R}}$ **ReL** $\nu\ \alpha$.

Together, Lemma 3 and Lemma 4 allow us to prove that if two regular expressions are equivalent, then so are their interpretations on binary relations.

Theorem 1. *Let α and β be regular expressions. Let ν be a function that maps symbols to relation. Hence, $\alpha \sim \beta \rightarrow$* **reRel** $\nu\ \alpha \sim_{\mathcal{R}}$ **reRel** $\nu\ \beta$.

With Theorem 1 we are able to implement a tactic that transforms a goal of the form **reRel** $\nu\ \alpha \sim_{\mathcal{R}}$ **reRel** $\nu\ \beta$ into a goal stating that $\alpha \sim \beta$, and that applies the tactic for regular expressions (in-)equivalence to close the proof. Here we omit the technical details of such tactic.

4.7. Performance

Although the main goal of our development was to provide certified evidence that the decision algorithm suggested by Almeida et al. [17] is correct, it is of obvious interest to understand the usability and efficiency of **equivP** and of the corresponding tactic while being computed within Coq's interactive environment. For that, we have experimented our tactic with several data sets of randomly generated regular expressions, in a uniform way. The data sets were generated by the FAdo tool [43], and each such data set is composed of 10 000 pairs of regular expressions, so that the results are statistically relevant, that is, the size of each sample is more than enough to ensure results statistically significant with a 95% confidence level within a 5% error margin. The experiments were conducted on a Virtual Box environment with 8 Gb of RAM, using coq-8.3pl4. The virtual environment executes on a dual six-core processor AMD Opteron(tm) 2435 processor with 2.60 GHz, and with 16 Gb of RAM. Table 1 provides the results obtained from our experiments.

Each entry in Table 1 corresponds to the average time that was required to compute the decision procedure over 10 000 pairs of regular expressions. The tests consider both equivalent – denoted by the rows labeled by *eq* – and inequivalent regular expressions – denoted by the rows labeled by *ineq*. The variable k ranges over the sizes of the sets of symbols from which the regular expressions are built. The variable n ranges over the sizes of the regular expressions generated, that is, the total number of constants, symbols and operators of the regular expression. The results presented in Table 1 (given in seconds) allow us to conclude that the procedure is efficient, since it is able to decide the equivalence of large regular expressions in less than 1 min. However, the procedure has its pitfalls: whenever the size of the alphabet is small and the size of the regular expressions is considerably large, e.g., for configurations where $k = 10$ and the size of the regular expressions is 1000, or where $k = 2$ and the size of the regular expressions is 250, the decision procedure – and therefore, the tactic – take a long time to give a result. This is due to the fact that the derivations computed along the execution of the procedure tend to produce few derivatives resulting in the pair $(\emptyset, \emptyset)$ and so, more recursive calls are needed.

Table 2
Comparison of the performances.

alg./(k, n)	$(2, 5)$		$(2, 10)$		$(2, 20)$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
equivP	0.003	0.002	0.008	0.003	0.020	0.004
ATBR	0.059	0.016	0.080	0.042	0.258	0.099
	$(4, 20)$		$(4, 50)$		$(10, 100)$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
equivP	0.035	0.004	0.172	0.010	0.776	0.016
ATBR	0.261	0.029	0.436	0.358	1.525	0.874
	$(20, 200)$		$(50, 500)$		$(50, 1000)$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
equivP	2.211	0.048	9.957	0.121	17.768	0.149
ATBR	3.001	1.654	5.876	2.724	16.682	12.448

Our decision procedure is very efficient in deciding inequivalences, even for the larger families of regular expressions considered. This brings advantages when using our tactic, for instance, as an argument for the **`try`** tactic.[4] In the next section we present a comparison of the performance exhibited by our procedure with other developments available.

4.8. Related work

The subject of developing certified algorithms for deciding regular expression equivalence within theorem provers is not new. In recent years, much attention was directed to this particular subject, resulting in several formalizations, some of which are based on derivatives, and spawn along three different interactive theorem provers, namely Coq, Isabelle [25] and Matita [44].

The most complete of the developments is the one of Braibant and Pous [19]: the authors formalized Kozen's proof of the completeness of KA [20] and developed also efficient tactics to decide KA equalities by computational reflection. Their construction is based on the classical automata process for deciding regular expressions equivalence without minimization of the involved automata. Moreover, they use a variant of Ilie and Yu's method [45] for constructing automata from regular expressions, and the comparison is performed using Karp's [15] direct comparison of automata. The resulting development is quite general (it is able to prove (in)equivalence of expression of several models of Kleene algebra) and is also quite efficient due to a careful choice of the data structures involved.

The works that are closer to ours are the works of Coquand and Siles [21], and of Nipkow and Krauss [23]. Coquand and Siles implemented a procedure for regular expression equivalence based on Brzozowski's derivative method, supported by a new construction of finite sets in type theory. They prove their algorithm correct and complete. Nipkow and Krauss' development is also based on Brzozowski's derivative, and it is a compact and elegant development carried out in the Isabelle theorem prover. However, the authors did not formalize the termination and completeness of the algorithm. In particular, the termination is far from being a trivial subject, as demonstrated by the work presented in this paper and in the work of Coquand and Siles.

More recently, Asperti presented a development [29] of an algorithm based on pointed regular expressions, which are regular expressions containing internal points. These points serve as indicators of the part of the regular expression that was already processed (transformed into a DFA) and therefore which part of the regular expression remains to be processed. The development is also quite short and elegant and provides an alternative to the algorithms based on Brzozowski's derivatives, since it does not require normalization modulo a suitable set of axioms to prove the finiteness of the number of the states of the corresponding DFA.

In Table 2 we provide results about a comparison between our development and the one of Braibant and Pous [19].[5] We do not present comparison with the other two Coq developments since they clearly exhibit worse performances than ours and the previous one. For technical reasons, we were not able to test the development of Asperti. In these experiments we have used datasets of 1000 uniform randomly generated regular expressions, and they were conducted on a Macbook Pro 15″, with a 2.3 GHz Intel Core i7 processor with 4 GB of RAM memory.

It is clear from Table 2 that the work of Braibant and Pous scales better than ours for larger families of regular expressions but it is drastically slower than ours with respect to regular expression in-equivalence. For smaller families of regular expressions, our procedure is also faster than theirs in both cases. The values k and n in Table 2 are the same measures that were used in Table 1, presented in the previous section for the analysis of the performance of **`equivP`**.

[4] The **`try`** tactic tries to apply another tactic given as argument into the current goal. If the applied tactic fails, then the proof state remains as it was, and no error is reported.

[5] The particular version of ATBR used in the performance comparison was the one developed for Coq version 8.3, and is available at `http://coq.inria.fr/pylons/pylons/contribs/view/ATBR/v8.3`.

5. Equivalence of KAT terms

Kleene algebra with tests (KAT) [46,47] is an algebraic system that extends Kleene algebra, the algebra of regular expressions, by considering a subset of *tests* whose elements satisfy the axioms of Boolean algebra. With the tests embedded in the expressions, we are able to express imperative program constructions, rather than just non-deterministic choice, sequential composition and iteration on a set of actions, as in the case of regular expressions.

KAT is particularly interesting to capture and verify properties of simple imperative programs since it provides an equational way to deal with partial correctness and program equivalence. KAT subsumes *propositional Hoare logic* (PHL) [48,49] since PHL's deductive rules can be encoded in KAT and we can prove that the encodings are theorems of KAT. Consequently, proving that a given program C is partially correct using the deductive system of PHL is tantamount to checking if C is partially correct by equational reasoning in KAT. Moreover, some Horn formulas [50,51] of KAT can be reduced to standard equalities and the equalities can be decided automatically using one of the available methods [47,52,53].

Formally, a KAT is an algebraic structure $(K, T, +, \cdot, ^\star, \bar{}, 0, 1)$ such that $(K, +, \cdot, ^\star, 0, 1)$ is a KA, $(T, +, \cdot, \bar{}, 0, 1)$ is a Boolean algebra and $T \subseteq K$. Therefore, KAT satisfies the axioms of KA and the axioms of Boolean algebra.

5.1. The language model of KAT

Let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a non-empty finite set whose elements are called *primitive tests*. Let $\overline{\mathcal{B}} = \{\bar{b} \mid b \in \mathcal{B}\}$ be the set such that each element $l \in \mathcal{B} \cup \overline{\mathcal{B}}$ is called a *literal*. An *atom* is a finite sequence of literals $l_1 l_2 \ldots l_n$, such that each l_i is either b_i or $\bar{b_i}$, for $1 \leq i \leq n$, where $n = |\mathcal{B}|$. We will refer to atoms by $\alpha, \alpha_1, \alpha_2, \ldots$, and to the set of all atoms on $\mathcal{B}$ by At. The set At can be regarded as the set of all truth assignments to elements of $\mathcal{B}$. Consequently, there are exactly $2^{|\mathcal{B}|}$ atoms.

Given an atom $\alpha \in$ At and a primitive test $b \in \mathcal{B}$, we write $\alpha \leq b$ if $\alpha \to b$ is a propositional tautology. For each primitive test $b \in \mathcal{B}$ and for each atom $\alpha \in$ At we always have $\alpha \leq b$ or $\alpha \leq \bar{b}$.

Besides primitive tests we also have to consider a finite set of symbols representing atomic programs, whose role is the same of the alphabet in the case of regular expressions. Such a set in KAT is called the set of *primitive actions* $\Sigma = \{p_1, \ldots, p_m\}$ and is represented in our formalization by the type of integers `Z` available in Coq's standard library.

In the Coq development we have encoded primitive tests and atoms as terms of the type of finite ordinal numbers `Ord` n which we present below. The type `Ord` n consists of two parameters: a natural number `nv` and a proof term `nv_lt_n` witnessing that `nv` is strictly smaller than `nv`. The functions for constructing an ordinal containing the value zero and for calculating the successor of an ordinal are also included in the code excerpt. We consider the existence of a global value `ntests` of type `nat` that establishes the cardinality of $\mathcal{B}$. The value of the parameter `ntests` is defined by instantiating the module type `KAT_Alph`. This module is a parameter for the rest of the modules that compose our development.

```
Module Type KAT_Alph.
  Parameter ntests : nat.
End KAT_Alph.

Record Ord (n:nat) := mk_Ord {
 nv :> nat;
 nv_lt_n : ltb nv n = true
}.

Definition ord0(n:nat) : Ord (S n) := @mk_Ord (S n) 0 eq_refl.

Definition ordS(n:nat)(m:Ord n) := @mk_Ord (S n) (S m) (ord_lt n m).
```

A member in $\mathcal{B}$ is a term of type `Ord ntests` and an atom in At is an inhabitant of the type `Ord` $(2^{\texttt{ntests}})$. We can calculate the set of all atoms as given by the function `ords_up_to` introduced below. The statement that proves the fact that `ords_up_to` calculates the set At on $\mathcal{B}$ is the lemma `all_atoms_in_ords_up_to` given below.

```
Definition ordS_map (n:nat) (s:set (ord n)) : set (Ord (S n)) := map (@ordS n) s.

Fixpoint ords_up_to (n:nat) {struct n} : set (ord n) :=
  match n with
  |0 → ∅
  |S m →add (ord0 m) (@ordS_map m (ords_up_to m))
  end.

Lemma all_atoms_in_ords_up_to : ∀ (n:nat)(i:ord n), i ∈ (ords_up_to n).
```

In order to finish the development of primitive tests and atom related concepts we need to define how we evaluate tests with respect to these structures. Let $b \in \mathcal{B}$ be a term of type `ord` n, and let m be the natural number it represents. In order to check that $\alpha \leq b$, where α is represented by a value of type `ord` $(2^{\texttt{ntests}})$, we simply look at the mth bit of the value of α. If that bit is 1 then $\alpha \leq b$ is true, and false otherwise. The code below presents the Coq code for making this decision,

where **N.testbit_nat** is a function that converts a value of type **nat** into its corresponding bit representation and returns the nth bit, where n is given as argument.

```
Definition nth_bit(m:nat)(k:Ord m)(n:nat) : bool := N.testbit_nat (N.of_nat k) n.
```

The syntax of KAT terms extends the syntax of regular expressions – or the syntax of KA – with elements called *tests*, which can be regarded as Boolean expressions on the underlying Boolean algebra of any KAT. A test is inductively defined as follows: the constants 0 and 1 are tests; if $b \in \mathcal{B}$ then b is a test; if t_1 and t_2 are tests, then $t_1 + t_2$, $t_1 \cdot t_2$, and $\overline{t_1}$ are tests. We denote the set of tests on $\mathcal{B}$ by T. In this setting, the operators $\cdot$, $+$ and $\bar{}$ are interpreted as Boolean operations of conjunction, disjunction and negation, respectively. The operators $\cdot$ and $+$ are naturally overloaded with respect to their interpretation as operators over elements of the underlying KA, where they correspond to non-deterministic choice and sequence, respectively. Like primitive tests, tests are evaluated with respect to atoms for validity. The function **evalT** below implements this evaluation following the inductive structure of tests. For $t \in \mathsf{T}$ and $\alpha \in \mathsf{At}$, we denote evaluation of tests with respect to atoms by $\alpha \leq t$.

A KAT *term* e is inductively defined as follows: if t is a test then t is a KAT term; if $p \in \Sigma$, the p is a KAT term; if e_1 and e_2 are KAT terms, then so are their union $e_1 + e_2$, their concatenation $e_1 e_2$, and their Kleene star $e_1^\star$. The set of all KAT terms is denoted by $\mathsf{K}_{\mathcal{B},\Sigma}$, and we denote syntactic equality between $e_1, e_2 \in \mathsf{K}_{\mathcal{B},\Sigma}$ by $e_1 \equiv e_2$. In Coq, the type of KAT terms and the type of tests are defined as expected.

A *guarded string* [54,55] is a sequence $x = \alpha_0 p_0 \alpha_1 p_1 \dots p_{(n-1)} \alpha_n$, with $\alpha_i \in \mathsf{At}$ and $p_i \in \Sigma$. Guarded strings start and end with an atom. When $n = 0$, then the guarded string is a single atom $\alpha_0 \in \mathsf{At}$. We use $x, y, x_0, y_0, \dots$ to refer to guarded strings. The set of all guarded strings over the sets $\mathcal{B}$ and Σ is denoted by $\mathsf{GS}_{\mathcal{B},\Sigma}$. For guarded string x we define $\mathsf{first}(x) \stackrel{\text{def}}{=} \alpha_0$ and $\mathsf{last}(x) \stackrel{\text{def}}{=} \alpha_n$. We say that two guarded strings x and y are *compatible* if $\mathsf{last}(x) = \mathsf{first}(y)$. If two guarded strings x and y are compatible, then the *fusion product* $x \diamond y$, or simply xy, is the standard word concatenation but omitting one of the common atoms $\mathsf{last}(x)$, or $\mathsf{first}(y)$. The fusion product of two guarded strings x and y is a partial function since it is only defined when x and y are compatible. In Coq we have implemented the fusion product of two guarded strings x and y by means of a dependent recursive function that takes has arguments x and y, and an explicit proof of the compatibility of x and y, i.e., a term of type **compatible** x y. The function **fusion_prod** implements the fusion product based on this criteria.

```
Lemma compatible_tl: ∀ (x y x':gs)(α:atom)(p:sy)(h:compatible x y)(l:x = gs_conc x p x'), compatible x' y.

Fixpoint fusion_prod (x y:gs)(h:compatible x y) : gs :=
 match x as x' return x=x' →gs with
 | gs_end _ ⇒
    fun (_:(x = gs_end _)) ⇒ y
 | gs_conc k s t ⇒
    fun (h0:(x = gs_conc k s t)) ⇒
      let h' := compatible_tl x y h k s t h0 in
        gs_conc k s (fusion_prod t y h')
end (refl_equal x).
```

Since the parameter h depends on the guarded strings x and y it must recursively decrease accordingly. The lemma **compatible_tl** states that if two guarded strings x and y are compatible, and if $x = \alpha p :: x'$, for some $\alpha p \in (\mathsf{At} \cdot \Sigma)$ and $x' \in \mathsf{GS}_{\mathcal{B},\Sigma}$, then x' and y remain compatible. The main properties of the fusion product over compatible guarded strings are the following: if $x, y, z \in \mathsf{GS}_{\mathcal{B},\Sigma}$ then the fusion product is associative, i.e., $(xy)z = x(yz)$; the fusion product of a guarded string x with a compatible atom α is an absorbing operation on the left or right of α, i.e., $\alpha x = x$ and $x\alpha = x$; the function last is left-invariant with respect to the fusion product, i.e., $\mathsf{last}(xy) = \mathsf{last}(y)$; conversely, the function first is right-invariant with respect to the fusion product, i.e. x $\mathsf{first}(xy) = \mathsf{first}(x)$.

In the language theoretic model of KAT, a *language* is a set of guarded strings over the sets $\mathcal{B}$ and Σ, i.e., a subset of $\mathsf{GS}_{\mathcal{B},\Sigma}$. KAT terms are syntactic expressions that denote languages of guarded strings. Thus, given a KAT term e, the language that e denotes, $\mathsf{G}(e)$, is recursively defined on the structure of e as follows: the language of a primitive program $p \in \Sigma$ is the set of all guarded strings that correspond at p prefixed and suffixed by atoms, that is, $\mathsf{G}(p) = \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}\}$; the language of a test $t \in T$ is the set of all atoms that satisfy t, that is, $\mathsf{G}(t) = \{\alpha \in \mathsf{At} \mid \alpha \leq t\}$; if $\mathsf{G}(e_1)$ and $\mathsf{G}(e_2)$ are the languages of the KAT terms e_1 and e_2, then their union, concatenation, and Kleene's star are the languages $\mathsf{G}(e_1 + e_2) = \mathsf{G}(e_1) \cup \mathsf{G}(e_2)$, $\mathsf{G}(e_1 e_2) = \mathsf{G}(e_1)\mathsf{G}(e_2)$, and $\mathsf{G}(e^\star) = \bigcup_{n \geq 0} \mathsf{G}(e)^n$, respectively. From the previous definition it is easy to conclude that $\mathsf{G}(1) = \mathsf{At}$ and that $\mathsf{G}(0) = \emptyset$. If $x \in \mathsf{GS}_{\mathcal{B},\Sigma}$, then its language is $\mathsf{G}(x) = \{x\}$. If e_1 and e_2 are two KAT terms, we say that e_1 and e_2 are equivalent, and write $e_1 \sim e_2$, if and only if $\mathsf{G}(e_1) = \mathsf{G}(e_2)$. We naturally extend the function G to sets S of KAT terms by $\mathsf{G}(S) \stackrel{\text{def}}{=} \bigcup_{e \in S} \mathsf{G}(e)$. If S_1 and S_2 are sets of KAT terms then $S_1 \sim S_2$ if and only if $\mathsf{G}(S_1) = \mathsf{G}(S_2)$. Moreover, if e is a KAT term and S is a set of KAT terms then $e \sim S$ if and only if $\mathsf{G}(e) = \mathsf{G}(S)$. We also have to consider the left-quotient of languages $L \subseteq \mathsf{GS}_{\mathcal{B},\Sigma}$. Quotients with respect to words $w \in (\mathsf{At} \cdot \Sigma)^\star$ are defined by $\mathcal{D}_w(L) \stackrel{\text{def}}{=} \{x \mid wx \in L\}$, and are specialized to elements $\alpha p \in (\mathsf{At} \cdot \Sigma)$ by $\mathcal{D}_{\alpha p}(L) \stackrel{\text{def}}{=} \{x \mid \alpha p x \in L\}$.

5.2. Partial derivatives of KAT terms

The notion of derivative of a KAT term was introduced by Kozen in [53] as an extension of Brzozowski's derivatives. In the same work, Kozen also introduces the notion of *set derivative*, which we will call partial derivative of a KAT term. Before formally introducing partial derivatives, we have to introduce the notion of nullability of a KAT term. Given an atom α and a KAT term e, we say that e is nullable if $\varepsilon_\alpha(e) = \mathtt{true}$, such that $\varepsilon_\alpha(\cdot)$ is inductively defined as follows: $\varepsilon_\alpha(p) = \mathtt{false}$ and $\varepsilon_\alpha(e^\star) = \mathtt{true}$ always hold; $\varepsilon_\alpha(t) = \mathtt{true}$ if $\alpha \leq t$ holds, and $\varepsilon_\alpha(t) = \mathtt{false}$, otherwise; $\varepsilon_\alpha(e_1 + e_2) = \mathtt{true}$ if at least one of $\varepsilon_\alpha(e_1)$ or $\varepsilon_\alpha(e_1)$ returns true; finally, $\varepsilon_\alpha(e_1 e_2) = \mathtt{true}$ if both $\varepsilon_\alpha(e_1) = \mathtt{true}$ and $\varepsilon_\alpha(e_2) = \mathtt{true}$ hold. The function ε is extended to the set of all atoms At by $\mathsf{E}(e) \stackrel{\text{def}}{=} \{\alpha \in \mathsf{At} \mid \varepsilon_\alpha(e) = \mathtt{true}\}$. Like in the case of the nullability of regular expression, we can relate the results of $\varepsilon_\alpha(e)$ with language membership: if $\varepsilon_\alpha(e) = \mathtt{true}$ then $\alpha \in \mathsf{G}(e)$; symmetrically, if $\varepsilon_\alpha(e) = \mathtt{false}$ then $\alpha \notin \mathsf{G}(e)$. For KAT terms e_1 and e_2, if $\varepsilon_\alpha(e_1) = \varepsilon_\alpha(e_2)$ holds for all $\alpha \in \mathsf{At}$, then we say that e_1 and e_2 are *equi-nullable*. Nullability is extended to sets in the following way: $\varepsilon_\alpha(S) = \mathtt{true}$ if there is at least one KAT term e in S such that $\varepsilon_\alpha(e) = \mathtt{true}$; conversely, $\varepsilon_\alpha(S) = \mathtt{false}$ if for all $e \in S$, $\varepsilon_\alpha(e) = \mathtt{false}$ holds. Two sets S_1 and S_2 of KAT terms are equi-nullable if $\varepsilon_\alpha(S_1) = \varepsilon_\alpha(S_2)$. For sets of KAT terms we also define the concatenation of a set with a KAT term, denoted $S \odot e$, as follows: $S \odot e = \emptyset$ and $S \odot e = S$ if $e \equiv 0$ and $e \equiv 1$, respectively; otherwise, $S \odot e = \{e'e \mid e' \in S\}$. As usual, we omit the operator $\odot$ whenever possible.

Let $\alpha p \in (\mathsf{At} \cdot \Sigma)$, with $\alpha \in \mathsf{At}$ and $p \in \Sigma$, and let e be a KAT term. The *set* $\partial_{\alpha p}(e)$ *of partial derivatives* of e with respect to αp is inductively defined by

$$\partial_{\alpha p}(t) = \emptyset \qquad \partial_{\alpha p}(q) = \begin{cases} \{1\} & \text{if } p \equiv q, \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\partial_{\alpha p}(e_1 + e_2) = \partial_{\alpha p}(e_1) \cup \partial_{\alpha p}(e_2) \qquad \partial_{\alpha p}(e_1 e_2) = \begin{cases} \partial_{\alpha p}(e_1)e_2 \cup \partial_{\alpha p}(e_2) & \text{if } \varepsilon_\alpha(e_1) = \mathtt{true}, \\ \partial_{\alpha p}(e_1)e_2, & \text{otherwise.} \end{cases}$$

$$\partial_{\alpha p}(e^\star) = \partial_{\alpha p}(e)e^\star$$

Partial derivatives of KAT terms are extended to words $w \in (\mathsf{At} \cdot \Sigma)^\star$, inductively by $\partial_\epsilon(e) = \{e\}$, and $\partial_{w\alpha p}(e) = \partial_{\alpha p}(\partial_w(e))$, where ϵ is the empty word. The set of all partial derivatives of a KAT term is the set

$$\partial_{(\mathsf{At}\cdot\Sigma)^\star}(e) \stackrel{\text{def}}{=} \bigcup_{w \in (\mathsf{At}\cdot\Sigma)^\star} \left\{ e' \;\middle|\; e' \in \partial_w(e) \right\}.$$

Example 5. Let $\mathcal{B} = \{b_1, b_2\}$, $\Sigma = \{p, q\}$, and $e = b_1 p(b_1 + b_2)q$. The partial derivative of e with respect to the sequence $b_1 b_2 p \overline{b_1} b_2 q$ is the following:

$$\begin{aligned}
\partial_{b_1 b_2 p \overline{b_1} b_2 q}(e) &= \partial_{b_1 b_2 p \overline{b_1} b_2 q}\big(b_1 p(b_1 + b_2)q\big) \\
&= \partial_{\overline{b_1} b_2 q}\big(\partial_{b_1 b_2 p}\big(b_1 p(b_1 + b_2)q\big)\big) \\
&= \partial_{\overline{b_1} b_2 q}\big(\partial_{b_1 b_2 p}(b_1)\big(p(b_1 + b_2)q\big) \cup \partial_{b_1 b_2 p}\big(p(b_1 + b_2)q\big)\big) \\
&= \partial_{\overline{b_1} b_2 q}\big(\partial_{b_1 b_2 p}(b_1)\big(p(b_1 + b_2)q\big)\big) \cup \partial_{\overline{b_1} b_2 q}\big(\partial_{b_1 b_2 p}\big(p(b_1 + b_2)q\big)\big) \\
&= \partial_{\overline{b_1} b_2 q}\big(\partial_{b_1 b_2 p}(p)(b_1 + b_2)q\big) \\
&= \partial_{\overline{b_1} b_2 q}\big((b_1 + b_2)q\big) \\
&= \partial_{\overline{b_1} b_2 q}(b_1 + b_2)q \cup \partial_{\overline{b_1} b_2 q}(q) \\
&= \partial_{\overline{b_1} b_2 q}(q) \\
&= \{1\}.
\end{aligned}$$

Similarly to partial derivatives of regular expressions, the language of partial derivatives of KAT terms are the left-quotients, that is, for $w \in (\mathsf{At} \cdot \Sigma)^\star$ and for $\alpha p \in (\mathsf{At} \cdot \Sigma)$, the following equalities $\mathsf{G}(\partial_w(e)) = \mathcal{D}_w(\mathsf{G}(e))$ and $\mathsf{G}(\partial_{\alpha p}(e)) = \mathcal{D}_{\alpha p}(\mathsf{G}(e))$ hold.

Kozen showed [53] that the set of partial derivatives is finite by means of the closure properties of a sub-term relation over KAT terms. As we have seen in the previous section, in the case of regular expressions the same problem can be solved using Mirkin's *pre-bases* [28]. Here we extend this method to KAT terms. We obtain an upper bound on the number of partial derivatives that is bounded by the number of primitive programs of Σ, and not the number of sub-terms as in [53].

Definition 1. Let e be a KAT term. The function $\pi(e)$ from KAT terms to sets of KAT terms is recursively defined as follows: for $p \in \Sigma$ and $t \in T$, we have $\pi(t) = \emptyset$ and $\pi(p) = \{1\}$, respectively; if e_1 and e_2 are KAT terms, then $\pi(e_1 + e_2) = \pi(e_1) \cup \pi(e_2)$ and $\pi(e_1 e_2) = \pi(e_1)e_2 \cup \pi(e_2)$ hold; finally, we have $\pi(e^\star) = \pi(e)e^\star$.

We now show that this is an upper bound of $\pi(e)$, which requires a lemma stating that π is a closed operation on KAT terms.

Proposition 10. *Let e be a KAT term over the set of primitive tests $\mathcal{B}$ and the set of primitive programs Σ. Hence, it holds that*

$$\forall e\, e',\ e' \in \pi(e) \rightarrow \forall e'',\ e'' \in \pi\left(e'\right) \rightarrow e'' \in \pi(e).$$

Lemma 5. *Let e be a KAT term over the set of primitive tests $\mathcal{B}$ and the set of primitive programs Σ. Hence, $|\pi(e)| \leq |e|_{\Sigma}$.*

Now let $\mathsf{KD}(e) \stackrel{\text{def}}{=} \{e\} \cup \pi(e)$, with e being a KAT term. It is easy to see that $|\mathsf{KD}(e)| \leq |e|_{\Sigma} + 1$, since $|\pi(e)|_{\Sigma} \leq |e|_{\Sigma}$. We will now show that $\mathsf{KD}(e)$ established an upper bound on the number of partial derivatives of e. For that, we prove that $\mathsf{KD}(e)$ contains all the partial derivatives of e. First we prove that the partial derivative $\partial_{\alpha p}(e)$ is a subset of $\pi(e)$, for all $\alpha \in \mathsf{At}$ and $p \in \Sigma$. Next, we prove that if $e' \in \partial_{\alpha p}(e)$ then $\pi(e')$ is a subset of $\pi(e)$, which allows us to prove, by induction on the length of a word $w \in (\mathsf{At} \cdot \Sigma)^{\star}$ that all the derivatives of e are members of $\mathsf{KD}(e)$.

Lemma 6. *Let e be a KAT term, and let $\alpha p \in (\mathsf{At} \cdot \Sigma)$. Hence, if the KAT term e' is a member of $\pi(e)$, then $\partial_{\alpha p}(e') \subseteq \pi(e)$.*

Theorem 2. *Let e be a KAT term, and let $w \in (\mathsf{At} \cdot \Sigma)^{\star}$. Thus, $\partial_w(e) \subseteq \mathsf{KD}(e)$.*

We finish this section by establishing a comparison between our method for establishing the finiteness of partial derivatives with respect to the one introduced by Kozen [53]. The method introduced by Kozen establishes an upper bound on the number of derivatives of a KAT term considering its number of sub-terms. In particular, Kozen establishes that, given a KAT term e, the number of derivatives is upper bounded by $|e| + 1$ elements, where here $|e|$ denotes the number of sub-terms of e given by a closure $cl(e)$. This upper bound is larger than the one we obtain using our definition of $\mathsf{KD}(e)$.

5.3. A procedure for KAT terms equivalence

In this section we introduce a procedure for deciding KAT terms equivalence that is based on the notion of partial derivative. This procedure is the natural extension of the procedure EQUIVP described in the last section. Most of the structures of both the development on regular expressions and this one overlap and, for this reason, we will mainly focus on the details where the difference between the two developments are most notorious.

The kind of reasoning that takes us from partial derivatives of KAT terms into solving their (in)equivalence is very similar to the one we have followed with respect to regular expression (in)equivalence. Given a KAT term e we know that

$$e \sim \mathsf{E}(e) \cup \left(\bigcup_{\alpha p \in (\mathsf{At} \cdot \Sigma)} \alpha p \partial_{\alpha p}(e) \right).$$

Therefore, if e_1 and e_2 are KAT terms, we can reformulate the equivalence $e_1 \sim e_2$ as

$$\mathsf{E}(e_1) \cup \left(\bigcup_{\alpha p \in (\mathsf{At} \cdot \Sigma)} \alpha p \partial_{\alpha p}(e_1) \right) \sim \mathsf{E}(e_2) \cup \left(\bigcup_{\alpha p \in (\mathsf{At} \cdot \Sigma)} \alpha p \partial_{\alpha p}(e_2) \right),$$

which is tantamount to checking that $\forall \alpha \in \mathsf{At}$, $\varepsilon_{\alpha}(e_1) = \varepsilon_{\alpha}(e_2)$ and $\forall \alpha p \in (\mathsf{At} \cdot \Sigma)$, $\partial_{\alpha p}(e_1) \sim \partial_{\alpha p}(e_2)$ hold. Since we know that to check if a guarded string x is a member of the language denoted by some KAT term e we need to prove that the derivative of e with respect to x must be nullable, we can finitely iterate over the previous equations and reduce the (in)equivalence of e_1 and e_2 to one of the next equivalences:

$$e_1 \sim e_2 \leftrightarrow \forall \alpha \in \mathsf{At}, \forall w \in (\mathsf{At} \cdot \Sigma)^{\star}, \varepsilon_{\alpha}\left(\partial_w(e_1)\right) = \varepsilon_{\alpha}\left(\partial_w(e_2)\right) \tag{6}$$

and

$$\left(\exists w\, \exists \alpha,\ \varepsilon_{\alpha}\left(\partial_w(e_1)\right) \neq \varepsilon_{\alpha}\left(\partial_w(e_2)\right)\right) \leftrightarrow e_1 \nsim e_2. \tag{7}$$

The terminating decision procedure EQUIVKAT, presented in Algorithm 2, describes the computational interpretation of the equivalences (6) and (7). This procedure corresponds to the iterated process of deciding the equivalence of their partial derivatives.

The computational behavior of EQUIVKAT is very similar to the behavior of EQUIVP, described in the previous section. Both are iterative processes that decide (in)equivalences by testing the (in)equivalence of the corresponding partial derivatives.

Clearly, EQUIVKAT is computationally more expensive than EQUIVP: the code in lines 4 to 8 performs $2^{|\mathcal{B}|}$ comparisons to determine if the components of the derivative (Γ, Δ) are equi-nullable or not, whereas EQUIVP performs one single operation to determine equi-nullability; the code in lines 10 to 15 performs $2^{|\mathcal{B}|}|\Sigma|$ derivations, while in the case of EQUIVP only $|\Sigma|$ derivations are calculated. In the next section we describe the implementation of EQUIVKAT in Coq.

We finish this section by providing two examples that describe the course of values produced by EQUIVKAT, one for the equivalence of KAT terms, and another for the case of in-equivalence.

Algorithm 2 The procedure EQUIVKAT.

Require: $S = \{(\{e_1\}, \{e_2\})\}$, $H = \emptyset$
Ensure: true iff $e_1 \sim e_2$, false otherwise

```
1: procedure EQUIVKAT(S, H)
2:     while S ≠ ∅ do
3:         (Γ, Δ) ← POP(S)
4:         for α ∈ At do
5:             if ε_α(Γ) ≠ ε_α(Δ) then
6:                 return false
7:             end if
8:         end for
9:         H ← H ∪ {(Γ, Δ)}
10:        for αp ∈ (At · Σ) do
11:            (Λ, Θ) ← ∂_αp(Γ, Δ)
12:            if (Λ, Θ) ∉ H then
13:                S ← S ∪ {(Λ, Θ)}
14:            end if
15:        end for
16:    end while
17: return true
18: end procedure
```

Example 6. Let $\mathcal{B} = \{b\}$ and let $\Sigma = \{p\}$. Suppose we want to prove that $e_1 = (pb)^\star p$ and $e_2 = p(bp)^\star$ are equivalent. Considering $s_0 = (\{(pb)^\star p\}, \{p(bp)^\star\})$, it is enough to show that EQUIVKAT$(\{s_0\}, \emptyset) =$ `true`. The first step of the computation generates the two new following pairs of derivatives:

$$\partial_{bp}\big(\{e_1\}, \{e_2\}\big) = \big(\{1, b(pb)^\star\}, \{(bp)^\star\}\big),$$
$$\partial_{\bar{b}p}\big(\{e_1\}, \{e_2\}\big) = \big(\{1, b(pb)^\star\}, \{(bp)^\star\}\big).$$

Since the new pairs are the same, only one of them is added to the working set S, and the original pair $(\{e_1\}, \{e_2\})$ is added to the historic set H. Hence, in the next iteration of EQUIVKAT considers $S = \{s_1\}$, with $s_1 = (\{1, b(pb)^\star\}, \{(bp)^\star\})$, and $H = \{s_0\}$. Once again, new derivatives are calculated and they are the following:

$$\partial_{bp}\big(\{1, b(pb)^\star\}, \{(bp)^\star\}\big) = \big(\{b(pb)^\star\}, \{(bp)^\star\}\big),$$
$$\partial_{\bar{b}p}\big(\{1, b(pb)^\star\}, \{(bp)^\star\}\big) = (\emptyset, \emptyset).$$

The next iteration of the procedure will have $S = \{s_2, s_3\}$ and $H = \{s_0, s_1\}$, considering that $s_2 = (\{b(pb)^\star\}, \{(bp)^\star\})$ and $s_3 = (\emptyset, \emptyset)$. Since the derivative of s_2 is either s_2 or s_3 and since the same holds for the derivatives of s_3, the procedure will terminate in two iterations with $S = \emptyset$ and $H = \{s_0, s_1, s_2, s_3\}$. Hence, we conclude that $e_1 \sim e_2$.

5.4. Implementation, correctness and completeness

The implementation of EQUIVKAT in the Coq proof assistant follows very closely the same approach that we have taken to implement EQUIVP. The only differences between the implementations of the two procedures are the type of terms used (KAT terms instead of regular expressions), and the modification that this difference of terms causes in the definitions of the intermediate functions and logical properties. Such differences can be seen in detail in [31].

5.5. Performance and usability of EQUIVKAT

As pointed out earlier, the performance of EQUIVKAT is not expected to be as efficient as EQUIVP. The algorithm contains two exponential computations, one for checking if a derivative is equi-nullable and another to compute the set of new derivatives.

In its current state of development, our procedure can be used to automatically decide (in)equivalence involving small KAT terms. Such small terms occur very frequently in proofs of KAT equations and our procedure can be used to help on finishing such proofs, by automatically solving eventual sub-goals. Moreover, and due to the capability of extraction of Coq, we can obtain a program that is correct by construction and that can be used outside Coq's environment and exhibit better performances.

5.6. Application to program verification

As we have written earlier, KAT is suited to several verification tasks, as proved by several reported experiments. In this section we try to motivate the reader to the reasons that lead us to formalize KAT in Coq: to have a completely correct

development which can serve as a certified environment to build proofs of the correctness and equivalence of simple imperative programs.

In this section, we present some examples borrowed from [46] that show how KAT can be useful to prove the equivalence between certain classes of simple imperative programs. Moreover, we show how KAT and Hoare logic are related by PHL and how deductive reasoning in PHL reduces to equational reasoning in KAT. We also present some motivating examples about the application of KAT to build proofs about the partial correctness of imperative programs. We consider the well-known IMP [56] programming language in our examples.

5.6.1. Equivalence of programs through KAT

Recall that the terms of KAT are regular expressions enriched with Boolean tests. The addition of tests provides extra expressivity to KAT terms when compared to regular expressions because they allow us to represent imperative program constructions such as conditionals and while loops. Since KAT is propositional, it does not allow to express assignments or other first order constructions. Nevertheless, and under an adequate encoding of the first order constructions at the propositional level, we can encode programs as KAT terms. Under this assumption, if e_1 and e_2 are terms encoding the IMP programs C_1 and C_2, and if the Boolean test B is encoded by the KAT test t, then we can encode sequence, conditional instructions and while loops in KAT as follows.

$$C_1; C_2 \stackrel{\text{def}}{=} e_1 e_2,$$
$$\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi} \stackrel{\text{def}}{=} (te_1 + \bar{t}e_2),$$
$$\text{while } B \text{ do } C_1 \text{ end} \stackrel{\text{def}}{=} (te_1)^\star \bar{t}.$$

We now present an example that illustrates how we can address program equivalence in KAT.

Example 7. Let $\mathcal{B} = \{b, c\}$ and $\Sigma = \{p, q\}$ be the set of primitive tests and set of primitive programs, respectively, and let P_1 and P_2 be the following two programs:

$$P_1 \stackrel{\text{def}}{=} \texttt{while } B \texttt{ do } C_1\texttt{; while } B' \texttt{ do } C_2 \texttt{ end end}$$
$$P_2 \stackrel{\text{def}}{=} \texttt{if } B \texttt{ then } C_1\texttt{; while } B + B' \texttt{ do if } B' \texttt{ then } C_2 \texttt{ else } C_1 \texttt{ fi end else skip fi}$$

Let $C_1 = p$, $C_2 = q$, $B = b$ and $B' = c$. The programs P_1 and P_2 are encoded in KAT as

$$e_1 = \big(bp\big((cq)^\star \bar{c}\big)\big)^\star \bar{b} \quad \text{and} \quad e_2 = bp\big((b + c)(cq + \bar{c}p)\big)^\star \overline{(b + c)} + \bar{b},$$

respectively. The procedure decides the equivalence $e_1 \sim e_2$ in 0.053 s.

5.6.2. Hoare logic and KAT

Hoare logic uses triples of the form $\{P\}\, C\, \{Q\}$, where P is the precondition and Q is the postcondition of the program C. The meaning of these triples, called Hoare triples or *partial correctness assertions* (PCA), is the following: *if P holds when C starts executing then Q will necessarily hold when C terminates, if that is the case*. Hoare logic consists of a set of inference rules which we can successively apply to triples in order to prove the partial correctness of the underlying program.

PHL is a weaker Hoare logic that does not come equipped with the assignment inference rule, since it is a propositional logic. KAT subsumes PHL and therefore the inference rules of PHL can be encoded as KAT theorems. This implies that deductive reasoning within PHL proof system reduces to equational reasoning in KAT. In KAT Hoare triples $\{P\}\, C\, \{Q\}$ are expressed by the KAT equations $t_1 e_1 = t_1 e_1 t_2$, or equivalently by $t_1 e_1 \overline{t_2} = 0$, or by $t_1 e_1 \leq t_2$, with $t_1, t_2 \in T$ and e_1 a KAT term. The inference rules of PHL are encoded as follows:

Sequence:

$$t_1 e_1 = t_1 e_1 t_2 \wedge t_2 e_2 = t_2 e_2 t_3 \to t_1 e_1 e_2 = t_1 e_1 e_2 t_3, \tag{8}$$

Conditional:

$$t_1 t_2 e_1 = t_1 t_2 e_1 t_3 \wedge \bar{t_1} t_2 e_2 = \bar{t_1} t_2 e_2 t_3 \to t_2(t_1 e_1 + \bar{t_1} e_2) = t_2(t_1 e_1 + \bar{t_1} e_2) t_3, \tag{9}$$

While-loop:

$$t_1 t_2 e_1 = t_1 t_2 e_1 t_2 \to t_2 (t_1 e_1)^* \bar{t_1} = t_2 (t_1 e_1)^* \bar{t_1} \bar{t_1} t_2. \tag{10}$$

Eqs. (8), (9) and (10) were mechanically checked during our development and correspond to the usual deductive rules of PHL. No assignment rule is considered here because PHL is propositional. Our decision procedure may be of little or no help here since we need to reason under sets of equational hypotheses, and we need to used them in a way that cannot be fully automated [46]. However, derivable PHL rules of the form

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \cdots \{P_n\}\, C_n\, \{Q_n\}}{\{P\}\, C\, \{Q\}} \tag{11}$$

correspond to KAT equational implication

$$t_1 e_1 \overline{t'_1} = 0 \wedge \ldots \wedge t_n e_n \overline{t'_n} = 0 \wedge \ldots \wedge t_{n+1} \leq t'_{n+1} \wedge \ldots\ t_{n+m} \leq t'_{n+m} \rightarrow t e \overline{t'} = 0. \tag{12}$$

It has been shown in [50] that for all KAT terms $r_1, \ldots, r_n, e_1, e_2$, over a set of primitive programs $\Sigma = \{p_1, \ldots, p_m\}$, the equational implications of the form $r_1 = 0 \wedge \ldots \wedge r_n = 0 \rightarrow e_1 = e_2$ is a theorem of KAT if and only if $e_1 + uru = e_2 + uru$, where $u \stackrel{\text{def}}{=} (p_1 + \ldots + p_m)^\star$ and $r \stackrel{\text{def}}{=} r_1 + \ldots + r_n$. At this point our decision procedure can be used to decide $e_1 \sim e_2$. In order to infer the set of hypotheses that is required to obtain the previous equation, we need annotated programs. These encoded programs have the following encoding:

$$\begin{aligned} C_1; \{P\}\, C_2 &\stackrel{\text{def}}{=} e_1 t e_2, \\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi} &\stackrel{\text{def}}{=} t e_1 + \bar{t} e_2, \\ \texttt{while}\ B\ \texttt{do}\ \{I\}\ C\ \texttt{end} &\stackrel{\text{def}}{=} (tip)^\star \bar{t}, \end{aligned}$$

with C_1 and C_2 encoding the KAT terms e_1 and e_2, and with P and I encoding the tests t and i, respectively. Given an IMP program we can obtain its annotated counterpart using a *verification conditions generator* (VCG) algorithm such as the one presented by Frade and Pinto [57] or, at the level of KAT only, using the algorithm proposed by Almeida et al. [58]. Below, we give an example on how this can be performed for a simple IMP program, and how to use our decision procedure to prove the program's partial correctness.

Example 8. Let us consider the following program:

$$\mathsf{Fact} \stackrel{\text{def}}{=} y := 1;\ z := 0;\ \mathsf{while}\ \neg(z = x)\ \mathsf{do}\ z := z + 1;\ \ y := y * z\ \mathsf{end}.$$

The program Fact computes the factorial of the value given by the variable x. The specification we wish to prove partially correct is thus $\{\mathsf{true}\}\,\mathsf{Fact}\,\{y = x!\}$. The annotated version of Fact is the following:

$$\mathsf{AnnFact} \stackrel{\text{def}}{=} y := 1; \{y = 0!\}\ z := 0; \{y = z!\}\ \mathsf{while}\ \neg(z = x)\ \mathsf{do}\ \{y = z!\}\ z := z + 1;\ \{y \times z = z!\}\ y := y * z;\ \mathsf{end}$$

Considering $\mathcal{B} = \{t_0, t_1, t_2, t_3, t_4, t_5\}$ and $\Sigma = \{p_1, p_2, p_3, p_4\}$, and assuming that $t_0 \stackrel{\text{def}}{=} \mathsf{true}$ and that $t_5 \stackrel{\text{def}}{=} y = x!$, the Hoare triple $\{\mathsf{true}\}\,\mathsf{AnnFact}\,\{y = x!\}$ is encoded as the equality

$$t_0 p_1 t_1 p_2 t_2 (t_3 t_2 p_3 t_4 p_4)^\star \overline{t_3} \overline{t_5} = 0. \tag{13}$$

To prove (13) we need to obtain a set of hypotheses, that can be obtained in a backward fashion [58] using a *weakest precondition generator*. The set of hypotheses for (13) is the following:

$$\Gamma = \{t_0 p_1 \overline{t_1} = 0, t_1 p_2 \overline{t_2} = 0, t_3 t_2 p_3 \overline{t_4} = 0, t_4 p_2 \overline{t_2} = 0, t_2 \overline{t_3} \overline{t_5} = 0\}.$$

With Γ and $\Sigma = \{p_1, p_2, p_3, p_4\}$, we know that

- $u = (p_1 + p_2 + p_3 + p_4)^\star$;
- $r = t_0 p_1 \overline{t_1} + t_1 p_2 \overline{t_2} + t_3 t_2 p_3 \overline{t_4} + t_4 p_2 \overline{t_2} + t_2 \overline{t_3} \overline{t_5}$.

The equation $t_0 p_1 t_1 p_2 t_2 (t_3 t_2 p_3 t_4 p_4)^\star \overline{t_3} \overline{t_5} + uru = 0 + uru$ is provable by the decision procedure in 22 s.

5.7. Related work

Although KAT can be applied to several verification tasks, there exists few tool support for it. Kozen and Aboul-Hosn [59] developed the KAT-ML proof assistant. KAT-ML allows one to reason about KAT terms. It also provides support to reason with assignments and other first-order programming constructs, since the underlying theory of KAT-ML is *schematic Kleene algebra with tests* (SKAT) [60], an extension of KAT with a notion of assignment, characterized by an additional set of axioms. However, KAT-ML provides no automation.

Worthington [52] presented a conversion from KAT terms into regular expressions and decides the equivalence of these regular expressions using Kozen's procedure based on automata. The formalization of this approach requires working with matrices, which diverges from the approach followed in this paper.

Almeida et al. [58,61] presented a new development of a decision procedure for KAT equivalence. The implementation was made using the OCaml programming language, is not mechanically certified, but includes a new method for proving the partial correctness of programs that dispenses the burden of constructing the terms r and u introduced in the previous section.

Finally, the work that is more related to ours is the one of Pous [62]. This work extends the previous work of the author in the automation of Kleene algebra in Coq which was already discussed previously. The author decides KAT term via a procedure based on partial derivatives like we do. The decision procedure suffers from the same exponential behavior on

the size of terms involved, and no performance evaluation was carried out. This development provides also a powerful tactic that automatically solves KAT equations with hypotheses of the kind of the ones used for proving the partial correctness that we have used earlier. Moreover, the development contains a completeness proof of KAT following along the lines of the work of Kozen and Smith [47].

Recently, Broda et al. [63] presented a new automata based approach towards deciding the equivalence of KAT terms that improves on the previous work by avoiding the extra burden of computing with all atoms at each iteration of the algorithm.

6. Conclusions and future work

In this paper we have presented a decision procedure for regular expression equivalence, and an extension of this procedure to decide the equivalence of KAT terms. We have described their implementation in Coq, as well their proofs of correctness and completeness with respect to the underlying model of languages. Both procedures are based on the notion of partial derivative. The main advantage of our method, when compared to equivalent methods based on Brzozowski's derivatives, is that in our case there is no need for normalization – modulo the associativity, commutativity and idempotence of the $+$ operator – of the expressions in order to prove the finiteness of the number of derivatives, and consequently, of the termination of the corresponding algorithms. The performances exhibited by our algorithm are satisfactory. Yet, there is always space to improve.

In what the procedure for regular expressions is concerned, we intend to improve its performance by using an alternative comparison between expressions, based on *hash consing* [64]. Moreover, we are interested in implementing the linear partial derivative's approach followed by Almeida et al. [17]. This approach is less expensive that ours because it discards symbols from the derivation steps of the procedure as soon as those symbols become absent from the derivatives that are yet to be processed. We are interested also in extending our development for regular expressions augmented with intersection and complement operations. These regular expressions are adequate for runtime verifications, and we believe that our formalized functions can be used as a trust-based framework for runtime verification of programs.

In the case of the development of the decision procedure for KAT terms equivalence, we are mainly interested in extending the current development in the following ways: first, we are interested in improving the performance of the KAT equivalence procedure along the ideas of Broda et al. [63,65] in which a more efficient way of dealing with the propositional layer is proposed; next, we wish to improve the performance of the decision procedure by following along the ideas introduced by Almeida et al. [58,61] and whose method proposed dispenses the creation of the KAT terms r and u that are required to automate the proof of partial correctness of imperative programs encoded as KAT terms; the other way we wish to improve in our development is to add support for SKAT [59,60], which we believe that it will approximate the usage of KAT to a more realistic notion of program verification, since at the level of SKAT we have access to first order constructions in programs.

Although KAT works at the propositional level, it still can be used as a framework to perform several verifications tasks, namely, program equivalence and partial correctness of programs. However, in such approaches, we must provide the necessary abstractions of the first order constructions as new tests and primitive actions and, some times, consider extra commutativity conditions over these abstractions. Moreover, verification tasks of this kind must still rely on external tools that must ensure that the first order constructions considered are valid.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects PTDC/EIA/65862/2006 (RESCUE), PTDC/EIA-CCO/101904/2008 (CANTE), FCOMP-01-0124-FEDER-020486 (AVIACC), and FCOMP-01-0124-FEDER-037281 (CISTER). The authors also wish to acknowledge the anonymous reviewers for the detailed comments that helped to improve considerably this manuscript.

References

[1] S. Kleene, Representation of events in nerve nets and finite automata, in: C. Shannon, J. McCarthy (Eds.), Automata Studies, Princeton University Press, 1956, pp. 3–42, Ch. 2.
[2] The Perl development team, The Perl programming language, http://www.perl.org, last accessed: 03.06.2013.
[3] G. Berry, L. Cosserat, The ESTEREL synchronous programming language and its mathematical semantics, in: S.D. Brookes, A.W. Roscoe, G. Winskel (Eds.), Seminar on Concurrency, in: LNCS, vol. 197, Springer, 1984, pp. 389–448.
[4] U. Sammapun, A. Easwaran, I. Lee, O. Sokolsky, Simulation of simultaneous events in regular expressions for run-time verification, Electron. Notes Theor. Comput. Sci. 113 (2005) 123–143.
[5] K. Sen, G. Rosu, Generating optimal monitors for extended regular expressions, Electron. Notes Theor. Comput. Sci. 89 (2) (2003) 226–245.
[6] J.-C. Filliâtre, Finite automata theory in Coq: a constructive proof of Kleene's theorem, Research report 97-04, LIP – ENS Lyon, February 1997.
[7] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer-Verlag, 2004.
[8] P. Hofner, G. Struth, Automated reasoning in Kleene algebra, in: F. Pfenning (Ed.), CADE 2007, in: LNAI, vol. 4603, Springer-Verlag, 2007, pp. 279–294.
[9] W. McCune, Prover9 and Mace4, http://www.cs.unm.edu/smccune/mace4, access date: 1.10.2011.
[10] D. Pereira, N. Moreira, KAT and PHL in Coq, Comput. Sci. Inf. Syst. (ISSN 1820-0214) 05 (02) (2008).
[11] D. Kozen, Kleene algebra with tests, ACM Trans. Program. Lang. Syst. 19 (3) (1997) 427–443.

[12] J.B. Almeida, N. Moreira, D. Pereira, S. Melo de Sousa, Partial derivative automata formalized in Coq, in: M. Domaratzki, K. Salomaa (Eds.), CIAA 2010, in: LNCS, vol. 6482, Springer-Verlag, 2011, pp. 59–68.
[13] M. Almeida, N. Moreira, R. Reis, Antimirov and Mosses's rewrite system revisited, Int. J. Found. Comput. Sci. 20 (4) (2009) 669–684.
[14] V.M. Antimirov, P.D. Mosses, Rewriting extended regular expressions, in: G. Rozenberg, A. Salomaa (Eds.), DLT, World Scientific, 1994, pp. 195–209.
[15] J. Hopcroft, R. Karp, A linear algorithm for testing equivalence of finite automata, Tech. rep. TR 71-114, University of California, Berkeley, California, 1971.
[16] M. Almeida, N. Moreira, R. Reis, Testing regular languages equivalence, J. Autom. Lang. Comb. 15 (1/2) (2010) 7–25.
[17] M. Almeida, Equivalence of regular languages: an algorithmic approach and complexity analysis, Ph.D. thesis, FCUP, 2011, http://www.dcc.fc.up.pt/~mfa/thesis.pdf.
[18] S. Broda, A. Machiavelo, N. Moreira, R. Reis, The average transition complexity of Glushkov and partial derivative automata, in: G. Mauri, A. Leporati (Eds.), Proc. of 15th DLT 2011, in: LNCS, vol. 6795, Springer, 2011, pp. 93–104.
[19] T. Braibant, D. Pous, An efficient Coq tactic for deciding Kleene algebras, in: Proc. of 1st ITP, in: LNCS, vol. 6172, Springer, 2010, pp. 163–178.
[20] D. Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, Inf. Comput. 110 (2) (1994) 366–390.
[21] T. Coquand, V. Siles, A decision procedure for regular expression equivalence in type theory, in: J.-P. Jouannaud, Z. Shao (Eds.), CPP 2011, Kenting, Taiwan, December 7–9, 2011, in: LNCS, vol. 7086, Springer-Verlag, 2011, pp. 119–134.
[22] J.A. Brzozowski, Derivatives of regular expressions, J. ACM 11 (4) (1964) 481–494.
[23] A. Krauss, T. Nipkow, Proof Pearl: regular expression equivalence and relation algebra, J. Autom. Reason. 49 (1) (2012) 95–106.
[24] J.J.M.M. Rutten, Automata and coinduction (an exercise in coalgebra), in: D. Sangiorgi, R. de Simone (Eds.), CONCUR, in: LNCS, vol. 1466, Springer, 1998, pp. 194–218.
[25] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283, Springer, 2002.
[26] V. Komendantsky, Reflexive toolbox for regular expression matching: verification of functional programs in Coq + Ssreflect, in: PLPV, 2012, pp. 61–70.
[27] V. Komendantsky, Computable partial derivatives of regular expressions, http://www.cs.st-andrews.ac.uk/~vk/papers.html, 2011, access date: 1.07.2011.
[28] B. Mirkin, An algorithm for constructing a base in a language of regular expressions, Eng. Cybern. 5 (1966) 110–116.
[29] A. Asperti, A compact proof of decidability for regular expression equivalence, in: L. Beringer, A. Felty (Eds.), ITP 2012, Princeton, NJ, USA, August 13–15, 2012, in: LNCS, vol. 7406, Springer-Verlag, 2012, pp. 283–298.
[30] N. Moreira, D. Pereira, S.M. de Sousa, Deciding regular expressions (in-)equivalence in Coq, in: W. Kahl, T.G. Griffin (Eds.), RAMICS, in: LNCS, vol. 7560, Springer, 2012, pp. 98–113.
[31] D. Pereira, Towards certified program logics for the verification of imperative programs, Ph.D. thesis, FCUP & MAPi Doctoral Programme in Computer Science, April 2013, http://www.liacc.up.pt/~kat/thesis.pdf.
[32] N. Moreira, D. Pereira S, Melo de Sousa, PDCoq: deciding regular expression and KAT terms (in)equivalence in Coq through partial derivative, http://www.liacc.up.pt/~kat/pdcoq/.
[33] F. Pfenning, C. Paulin-Mohring, Inductively defined types in the calculus of constructions, in: M.G. Main, A. Melton, M.W. Mislove, D.A. Schmidt (Eds.), Mathematical Foundations of Programming Semantics, in: LNCS, vol. 442, Springer, 1989, pp. 209–228.
[34] T. Coquand, G.P. Huet, The calculus of constructions, Inf. Comput. 76 (2/3) (1988) 95–120.
[35] W.A. Howard, The formulas-as-types notion of construction, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980, pp. 479–490.
[36] G. Barthe, P. Courtieu, Efficient reasoning about executable specifications in Coq, in: V. Carreño, C. Muñoz, S. Tahar (Eds.), TPHOLs, in: LNCS, vol. 2410, Springer, 2002, pp. 31–46.
[37] A. Chlipala, Certified Programming with Dependent Types, MIT Press, 2011, http://adam.chlipala.net/cpdt/.
[38] B.C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu, V. Sjoberg, B. Yorgey, Software foundations, Electronic textbook http://www.cis.upenn.edu/~bcpierce/sf, 2012.
[39] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison–Wesley Publishing Company, Reading, Massachusetts, USA, 1979.
[40] D. Kozen, Automata and Computability, Springer-Verlag, New York, 1997.
[41] V.M. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theor. Comput. Sci. 155 (2) (1996) 291–319.
[42] J.-M. Champarnaud, D. Ziadi, From Mirkin's prebases to Antimirov's word partial derivatives, Fundam. Inform. 45 (3) (2001) 195–205.
[43] A. Almeida, M. Almeida, J. Alves, N. Moreira, R. Reis, FAdo and GUItar: tools for automata manipulation and visualization, in: S. Maneth (Ed.), Proc. 14th CIAA'09, in: LNCS, vol. 5642, Springer-Verlag, 2009, pp. 65–74.
[44] A. Asperti, W. Ricciotti, C.S. Coen, E. Tassi, The Matita interactive theorem prover, in: N. Bjørner, V. Sofronie-Stokkermans (Eds.), CADE, in: LNCS, vol. 6803, Springer, 2011, pp. 64–69.
[45] L. Ilie, S. Yu, Follow automata, Inf. Comput. 186 (1) (2003) 140–162.
[46] D. Kozen, Kleene algebra with tests, ACM Trans. Program. Lang. Syst. 19 (3) (1997) 427–443.
[47] D. Kozen, F. Smith, Kleene algebra with tests: completeness and decidability, in: Proc. 10th Int. Workshop Computer Science Logic, CSL'96, in: LNCS, vol. 1258, Springer-Verlag, 1996, pp. 244–259.
[48] D. Kozen, J. Tiuryn, On the completeness of propositional Hoare logic, in: Relational Methods in Computer Sciece, Inf. Sci. 139 (3–4) (2001) 187–195.
[49] D. Kozen, On Hoare logic and Kleene algebra with tests, ACM Trans. Comput. Log. 1 (1) (2000) 60–76.
[50] C. Hardin, Modularizing the elimination of $r = 0$ in Kleene algebra, Log. Methods Comput. Sci. 1 (3) (2005).
[51] C. Hardin, D. Kozen, On the elimination of hypotheses in Kleene algebra with tests, Tech. rep. TR2002-1879, Computer Science Department, Cornell University, October 2002.
[52] J. Worthington, Automatic proof generation in Kleene algebra, in: RelMiCS'08/AKA'08, in: LNCS, vol. 4988, Springer-Verlag, Berlin/Heidelberg, 2008, pp. 382–396.
[53] D. Kozen, On the coalgebraic theory of Kleene algebra with tests, Computing and information science technical reports, Cornell University, March 2008.
[54] D.M. Kaplan, Regular expressions and the equivalence of programs, J. Comput. Syst. Sci. 3 (4) (1969) 361–386.
[55] D. Kozen, Automata on guarded strings and applications, Mat. Contemp. 24 (2003) 117–139.
[56] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, Cambridge, MA, USA, 1993.
[57] M.J. Frade, J.S. Pinto, Verification conditions for source-level imperative programs, Comput. Sci. Rev. 5 (3) (2011) 252–277.
[58] R. Almeida, Decision algorithms for Kleene algebra with tests and Hoare logic, Master's thesis, Faculdade de Ciências da Universidade do Porto, 2012.
[59] K. Aboul-Hosn, D. Kozen, KAT-ML: an interactive theorem prover for Kleene algebra with tests, J. Appl. Non-Class. Log. 16 (1–2) (2006) 9–33.
[60] A. Angus, D. Kozen, Kleene algebra with tests and program schematology, Tech. rep. TR2001-1844, Cornell University, 2001.
[61] R. Almeida, S. Broda, N. Moreira, Deciding KAT and Hoare logic with derivatives, in: GandALF, Electronic Proceedings in Theoretical Computer Science, 2012, pp. 127–140.
[62] D. Pous, Relation algebra and KAT in Coq, http://perso.ens-lyon.fr/damien.pous/ra/, December 2012, last accessed: 30.12.2012.

[63] S. Broda, A. Machiavelo, N. Moreira, R. Reis, On the average size of Glushkov and equation automata for KAT expressions, in: 19th Inter. Symposium on Fundamentals of Computation Theory, in: LNCS, vol. 8070, Springer-Verlag, 2013, pp. 72–83.
[64] J.-C. Filliâtre, S. Conchon, Type-safe modular hash-consing, in: Proceedings of the 2006 Workshop on ML, ML '06, ACM, New York, NY, USA, 2006, pp. 12–19.
[65] S. Broda, A. Machiavelo, N. Moreira, R. Reis, On the equivalence of automata for KAT-expressions, in: A. Beckmann, E. Csuhaj-Varjú, K. Meer (Eds.), CiE, in: LNCS, vol. 8493, Springer, 2014, pp. 73–83.