Deciding Regular Expressions (In-)Equivalence in Coq*

Nelma Moreira[1], David Pereira[1**] and Simão Melo de Sousa[1]

[1] DCC-FC – University of Porto
Rua do Campo Alegre 1021, 4169-007, Porto, Portugal
nam@dcc.fc.up.pt,dpereira@ncc.up.pt
[2] LIACC & DI – University of Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001, Covilhã, Portugal
desousa@di.ubi.pt

Abstract. This work presents a mechanically verified implementation of an algorithm for deciding regular expression (in-)equivalence within the Coq proof assistant. This algorithm decides regular expression equivalence through an iterated process of testing the equivalence of their partial derivatives and also does not construct the underlying automata. Our implementation has a refutation step that improves the general efficiency of the decision procedure by enforcing the in-equivalence of regular expressions at early stages of computation. Recent theoretical and experimental research provide evidence that this method is, on average, more efficient than the classical methods based in automata. We present some performance tests and comparisons with similar approaches.

1 Introduction

Recently, much attention has been given to the mechanization of Kleene algebra (KA) within proof assistants. J.-C. Filliâtre [1] provided a first formalisation of the Kleene theorem for regular languages [2] within the Coq proof assistant [3]. Höfner and Struth [4] investigated the automated reasoning in variants of Kleene algebras with Prover9 and Mace4 [5]. Pereira and Moreira [6] implemented in Coq an abstract specification of Kleene algebra with tests (KAT) [7] 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 presented in [8]. In this paper, our goal is to mechanically verify a decision procedure based on partial derivatives proposed by Almeida *et*

* This work was partially funded by the European Regional Development Fund through the programme COMPETE and by the Portuguese Government through the FCT under the projects PEst-OE/EEI/UI0027/2011, PEst-C/MAT/UI0144/2011, and CANTE-PTDC/EIA-CCO/101904/2008.
** David Pereira is funded by FCT grant SFRH/BD/33233/2007

al. [9] that is a functional variant of the rewrite system proposed by Antimirov and Mosses [10]. 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 [11] presented an almost linear algorithm for equivalence of two deterministic finite automata (DFA). By transforming regular expressions into equivalent DFAs, Hopcroft and Karp method can be used for regular expressions equivalence. A comparison of that method with the method here proposed is discussed by Almeida et. al [12, 13]. There it is conjectured that a direct method should perform better on average, and that is corroborated by theoretical studies based on analytic combinatorics [14]. Hopcroft and Karp method was used by Braibant and Pous [15] to formally verify Kozen's proof of the completeness of Kleene algebra [16] in Coq. Although the relative inefficiency of the method chosen, and as we will note in Section 4, it seems the more competitive (and most general) implementation available currently in Coq.

Independently of the work here presented, Coquand and Siles [17] mechanically verified an algorithm for deciding regular expression equivalence based on Brzozowski's derivatives [18] and an inductive definition of finite sets called Kuratowski-finite sets. Also based on Brzozowski's derivatives, Krauss and Nipkow [19] provide an elegant and concise formalisation of Rutten's co-algebraic approach of regular expression equivalence [20] in the Isabelle proof assistant [21], but they do not address the termination of the formalized decision procedure. Vladimir Komendantsky provides a novel functional construction of the *partial derivative automaton* [22], and also made contributions [23] to the mechanization of concepts related to the Mirkin's construction [24] of that automata. More recently, Andrea Asperti formalized a decision procedure for the equivalence of *pointed regular expressions* [25], that is both compact and efficient.

Besides avoiding the need of building DFAs, our use of partial derivatives avoids also the necessary normalisation of regular expressions modulo ACI (i.e associativity, idempotence and commutativity of union) in order to ensure the finiteness of Brzozowski's derivatives. Like in other approaches [15], our method also includes a refutation step that improves the detection of inequivalent regular expressions. One of our goals is to use the procedure as a way to automate the process of reasoning about programs encoded as KAT terms in a certified framework. A first step towards this goal is reported in [26].

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 such as the one behind the Coq proof assistant raises several issues which are quite different from an 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 gen-

eral recursive one, whereas Coq's type system accepts only provable terminating functions. Finally, a formal proof must be provided in order to ensure that the functional behavior of the implemented procedure is correct wrt. regular expression (in-)equivalence. Moreover, the encoding effort must be conducted with care in order to obtain a solution that is able to compute inside Coq with a reasonable performance.

2 Some Basic Notions of Regular Languages

This section presents some basic notions of regular languages. These definitions can be found on standard books such as Hopcroft *et al.* [27], and their formalisation in the Coq proof assistant are presented by Almeida *et al.* [8].

2.1 Alphabets, Words, Languages and Regular Expressions

Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$ be an *alphabet* (non-empty set of symbols). A *word* w over Σ is any finite sequence of symbols. The *empty word* is denoted by ε and the *concatenation* of two words w_1 and w_2 is the word $w = w_1 w_2$. Let $\Sigma^\star$ be the set of all words over Σ. A *language* over Σ is a subset of $\Sigma^\star$. If L_1 and L_2 are two languages, then $L_1 L_2 = \{w_1 w_2 \,|\, w_1 \in L_1, w_2 \in L_2\}$. The *power* of a language is inductively defined by $L^0 = \{\varepsilon\}$ and $L^n = LL^{n-1}$, with $n \geq 1$. The *Kleene star* $L^\star$ of a language L is $\cup_{n \geq 0} L^n$. Given a word $w \in \Sigma^\star$, the *(left-)quotient* of L by the word w is the language $w^{-1}(L) = \{v \,|\, wv \in L\}$.

A *regular expression* (*re*) α over Σ represents a *regular language* $\mathcal{L}(\alpha) \subseteq \Sigma^*$ and is inductively defined by: $\emptyset$ is a *re* and $\mathcal{L}(\emptyset) = \emptyset$; ε is a *re* and $\mathcal{L}(\varepsilon) = \{\epsilon\}$; $\forall\, a \in \Sigma$, a is a *re* and $\mathcal{L}(a) = \{a\}$; if α and β are *re*'s, $(\alpha + \beta)$, $(\alpha\beta)$ and $(\alpha)^*$ are *re*'s, respectively with $\mathcal{L}(\alpha + \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$ and $\mathcal{L}(\alpha^\star) = \mathcal{L}(\alpha)^\star$. If Γ is a set of *re*'s, then $\mathcal{L}(\Gamma) = \cup_{\alpha \in \Gamma} \mathcal{L}(\alpha)$. The *alphabetic size* of a *re* α is the number of symbols of the alphabet in α and is denoted by $|\alpha|_\Sigma$. The *empty word property* (*ewp* for short) of a *re* α is denoted by $\varepsilon(\alpha)$ and is defined by $\varepsilon(\alpha) = \varepsilon$ if $\varepsilon \in \mathcal{L}(\alpha)$ and by $\varepsilon(\alpha) = \emptyset$, otherwise. If $\varepsilon(\alpha) = \varepsilon(\beta)$ we say that α and β *have the same ewp*. Given a set of *re*'s Γ we define $\varepsilon(\Gamma) = \varepsilon$ if there exists a *re* $\alpha \in \Gamma$ such that $\varepsilon(\alpha) = \varepsilon$ and $\varepsilon(\Gamma) = \emptyset$, otherwise. Two *re*'s α and β are *equivalent* if they represent the same language, that is, if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, and we write $\alpha \sim \beta$.

2.2 Partial Derivatives

The notion of *derivative* of a *re* was introduced by Brzozowski [18]. Antimirov [10] extended this notion to the one of set of *partial derivatives*, which correspond to a finite set representation of Brzozowski's derivatives.

Let α be a *re* and let $a \in \Sigma$. The *set* $\partial_a(\alpha)$ *of partial derivatives* of the *re* w.r.t. the symbol a is inductively defined as follows:

$$\begin{array}{ll} \partial_a(\emptyset) = \emptyset & \partial_a(\alpha+\beta) = \partial_a(\alpha) \cup \partial_a(\beta) \\ \partial_a(\varepsilon) = \emptyset & \partial_a(\alpha\beta) \quad = \begin{cases} \partial_a(\alpha)\beta \cup \partial_a(\beta) & \text{if } \varepsilon(\alpha) = \varepsilon \\ \partial_a(\alpha)\beta & \text{otherwise} \end{cases} \\ \partial_a(b) = \begin{cases} \{\varepsilon\} \text{ if } a \equiv b \\ \emptyset \quad \text{otherwise} \end{cases} & \partial_a(\alpha^\star) \quad = \partial_a(\alpha)\alpha^\star, \end{array}$$

where $\Gamma\beta = \{\alpha\beta \,|\, \alpha \in \Gamma\}$ if $\beta \neq \emptyset$ and $\beta \neq \varepsilon$, and $\Gamma\emptyset = \emptyset$ and $\Gamma\varepsilon = \Gamma$ otherwise (in the same way we define $\beta\Gamma$). Moreover one has

$$\mathcal{L}(\partial_a(\alpha)) = a^{-1}(\mathcal{L}(\alpha)). \tag{1}$$

The definition of set of partial derivatives is extended to sets of *re*'s and to words. Given a *re* α, a symbol $a \in \Sigma$, a word $w \in \Sigma^\star$, and a set of *re*'s Γ, we define $\partial_a(\Gamma) = \cup_{\alpha\in\Gamma}\partial_a(\alpha)$, $\partial_\varepsilon(\alpha) = \{\alpha\}$, and $\partial_{wa} = \partial_a(\partial_w(\alpha))$. Equation (1) can be extended to words $w \in \Sigma^\star$. The *set of partial derivatives* of a *re* α is defined by $PD(\alpha) = \cup_{w\in\Sigma^\star}(\partial_w(\alpha))$. This set is always finite and its cardinality is bounded by $|\alpha|_\Sigma + 1$.

Champarnaud and Ziadi show in [28] that partial derivatives and Mirkin's prebases [24] lead to identical constructions. Let $\pi(\alpha)$ be a function inductively defined as follows:

$$\begin{array}{llll} \pi(\emptyset) = \emptyset & \pi(\alpha+\beta) & = \pi(\alpha) \cup \pi(\beta) \\ \pi(\varepsilon) = \emptyset & \pi(\alpha\beta) & = \pi(\alpha)\beta \cup \pi(\beta) \\ \pi(a) = \{\varepsilon\} & \pi(\alpha^\star) & = \pi(\alpha)\alpha^\star. \end{array} \tag{2}$$

In his original paper, Mirkin proved that $|\pi(\alpha)| \leq |\alpha|_\Sigma$, while Champarnaud and Ziadi established that $PD(\alpha) = \{\alpha\} \cup \pi(\alpha)$. These properties were proven correct in Coq by Almeida *et al.* [8] and will be used to prove the termination of the decision procedure described in this paper.

An important property of partial derivatives is that given a *re* α we have

$$\alpha \sim \varepsilon(\alpha) + \sum_{a\in\Sigma} a\partial_a(\alpha) \tag{3}$$

and so, checking if $\alpha \sim \beta$ can be reformulated as

$$\varepsilon(\alpha) + \sum_{a\in\Sigma} a\partial_a(\alpha) \sim \varepsilon(\beta) + \sum_{a\in\Sigma} a\partial_a(\beta). \tag{4}$$

This will be an essential ingredient to our decision method because deciding if $\alpha \sim \beta$ is tantamount to check if $\varepsilon(\alpha) = \varepsilon(\beta)$ and if $\partial_a(\alpha) \sim \partial_a(\beta)$, for each $a \in \Sigma$. We also note that testing if a word $w \in \Sigma^\star$ belongs to $\mathcal{L}(\alpha)$ can be reduced to the purely syntactical operation of checking if

$$\varepsilon(\partial_w(\alpha)) = \varepsilon. \tag{5}$$

By (4) and (5) we have that

$$(\forall w \in \Sigma^\star, \varepsilon(\partial_w(\alpha)) = \varepsilon(\partial_w(\beta))) \leftrightarrow \alpha \sim \beta. \tag{6}$$

3 The Decision Procedure

In this section we describe the implementation in Coq of a procedure for deciding the equivalence of *re*'s based on partial derivatives. First we give the informal description of the procedure and afterwards we present the technical details of its implementation in Coq's type theory. The Coq development presented in this paper is available online in [29].

3.1 Informal Description

The procedure for deciding the equivalence of *re*'s, which we call EQUIVP, is presented in Fig.1. Given two *re*'s α and β this procedure corresponds to the iterated process of deciding the equivalence of their derivatives, in the way noted in equation (4). The procedure EQUIVP works over pairs of *re*'s (Γ, Δ) such that $\Gamma = \partial_w(\alpha)$ and $\Delta = \partial_w(\beta)$, for some word $w \in \Sigma^\star$. The notion of set of partial derivatives can also be extended to these pairs that we refer from now on by *derivatives*. To check if $\alpha \sim \beta$ it is enough to test the *ewp* 's of the derivatives, *i.e.*, if (Γ, Δ) verify the condition $\varepsilon(\Gamma) = \varepsilon(\Delta)$.

Algorithm 1 The procedure EQUIVP.

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

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

Two finite sets of derivatives are required for implementing EQUIVP: a set H that serves as an accumulator for the derivatives already processed by the procedure, and a set S which serves as a working set that gathers new derivatives yet to be processed. The set H ensures the termination of EQUIVP due to the finiteness of the number of derivatives.

When EQUIVP terminates, either the set H of all the derivatives of α and β has been computed, or a counter-example (Γ, Δ) has been found, *i.e.*, $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$. By equation (6), in the first case we conclude that $\alpha \sim \beta$ and, in the second case we conclude that $\alpha \not\sim \beta$. The correctness of this method can be found in Almeida *et al.* [9, 12]. As an illustration of how EQUIVP computes, we present below two small examples of its execution, the first considering the equivalence of two *re*'s, and the second considering the in-equivalence of two *re*'s.

Example 1. Suppose that we want to prove that the *re*'s $\alpha = (ab)^\star a$ and $\beta = a(ba)^\star$ are equivalent. Considering $s_0 = (\{a(ab)^\star\}, \{a(ba)^\star\})$, it is enough to show that

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

The computation of EQUIVP is 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 $\alpha \sim \beta$.

Example 2. Suppose that now we want to check if the *re*'s $\alpha = b^\star a$ and $\beta = b^\star ba$ are not equivalent. Considering $s_0 = (\{b^\star a\}, \{b^\star ba\})$, to prove so it is enough to check if

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

In this case, the computation of EQUIVP creates the new derivatives , $s_1 = (\{1\}, \emptyset)$ and $s_2 = (\{b^\star a\}, \{a, b^\star ba\})$, and takes two iterations to halt and return `false`. The counter example found is the pair s_1, as it is easy to see in the trace of computation presented in the table below.

i	S_i	H_i	*drvs.*
0	$\{s_0\}$	$\emptyset$	$\partial_a(s_0) = s_1, \partial_b(s_2) = s_2$
1	$\{s_1, s_2\}$	$\{s_0\}$	$\varepsilon(s_1) =$ `false`

3.2 Implementation in Coq

In this section we describe the mechanically verified formalisation of EQUIVP in the Coq proof assistant and show its termination and correctness.

The Coq Proof Assistant

The Coq proof assistant is an implementation of the Calculus of Inductive Constructions (CIC) [30], a typed λ-calculus that features polymorphism, dependent types and very expressive (co-)inductive types. Coq provides users with the means to define data-structures and functions, as in standard functional languages, and also allows to define specifications and to build proofs in the same language, if we consider the underlying λ-calculus as an higher-order logic. In CIC, every term has a type and also every type has its own type, called *sort*. The universe of sorts in Coq is defined as the set $\{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}$, where Prop is the type of propositions and Set is the type of program specifications. Both Prop and Set are of type Type(0). This distinction between the type of propositions and the type of program specifications permits Coq to provide a mechanism that extracts functional programs directly from Coq scripts, by ignoring all the propositions and extracting only the computationally meaningful definitions. More details on the way certified program development and proof construction are carried out can be found in [3].

In the formalisation below we use only the libraries and certified programming and proving mechanisms provided by the Coq official distribution [31]. In particular, our implementation is not axiom-free, as it depends on set extensionally, but which does not interfere with the consistency of the development. We also use a specific library (which is not in Coq's standard library) to deal with finite sets: in this case we use Stephane Lescuyer's Containers library [32], which is a re-implementation of Coq's finite sets library using *typeclasses*. This library eases the implementation of functions that deal with finite sets and also provides facilities to handle ordered types. In particular, using this library we obtain the type of finite sets of a finite set for free. These properties revealed themselves quite handy for our development which is based mostly on sets, and sets of sets of *re*'s (and extensions).

Certified Pairs of Derivatives

The main data structures underlying the implementation of EQUIVP are pairs of sets of *re*'s and sets of these pairs. Each pair (Γ, Δ) corresponds to a word derivative $(\partial_w(\alpha), \partial_w(\beta))$, where $w \in \Sigma^\star$ and α and β are the *re*'s being tested by EQUIVP. The pairs (Γ, Δ) are encoded by the type `Drv` α β, presented in Fig.1. This is a *dependent record* built from three parameters: a pair of sets of *re*'s `dp` that corresponds to the actual pair (Γ, Δ), a word `w`, and a proof term `cw` that certifies that $(\Gamma, \Delta) = (\partial_{\mathtt{w}}(\alpha), \partial_{\mathtt{w}}(\beta))$. The dependency of `Drv` α β comes from `cw`, which is a proof depending on the values of the *re*'s α and β, and on the word parameter `w`. This dependency ensures, at compilation time, that EQUIVP will only accept as input pairs of *re*'s that correspond to derivatives of α and β.

The type `Drv` α β provides also an easy way to relate the computation of EQUIVP and the equivalence of α and β: if H is the set returned by EQUIVP, then the equation (6) is tantamount to check the *ewp* of the elements of H.

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

Program Definition Drv_1st (α β:re) : Drv α β.
refine(Build_Drv ({r1},{r2}) nil _).
(* Now comes the proof that ({α},{β}) = (∂_ε(α),∂_ε(β)) *)
abstract(unfold wpdrvp;simpl;constructor;
         unfold wpdrv_set;simpl;normalize_notations;auto).
Defined.

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).
(* Now comes the proof that ∂_a(∂_w(α),∂_w(β)) = (∂_wa(α),∂_wa(β)) *)
abstract(unfold pdrvp;inversion_clear P;simpl in *;
         constructor;normalize_notations;simpl;
         [rewrite H|rewrite H0];rewrite wpdrv_set_app;
         unfold wpdrv_set;simpl;reflexivity).
Defined.

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

Definition Drv_wpdrv (α β:re)(w:word) : Drv α β.
refine(mkDrv α β (∂_w(α),∂_w(β)) w _).
(* Now comes the proof that (∂_w(α),∂_w(β)) = (∂_w(α),∂_w(β)) *)
abstract(reflexivity).
Defined.

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.
```

Fig. 1: Definition of the type `Drv` and the extension of derivatives and *ewp* functions.

Furthermore, this type allows to keep the set of words from which the set of derivatives of α and β has been obtained. For that it is enough to apply the projection `w` to each pair $(\Gamma, \Delta) \in H$.

The notions of derivative and of *ewp* are extended to the type `Drv` α β as implemented by the functions `Drv_pdrv` and `c_of_Drv`, and to sets of terms `Drv` α β by the functions `Drv_pdrv_set` and `c_of_Drv_set`, respectively. Note that part of the implementation of these functions is done by explicitly building proof terms using Coq's tactical language. In order to improve the performance of the computation of these functions we have wrapped the corresponding proofs in the `abstract` tactic, which defines these proofs as external lemmas and, as a consequence, replaces the explicit computation of the proof terms by a function call to the corresponding external lemma.

Computation of New Derivatives

The *while-loop* of EQUIVP describes the process of testing the equivalence of the derivatives of α and β. In each iteration, new derivatives (Γ, Δ) are computed until either the set S becomes empty, or a pair (Γ, Δ) such that $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$ is found. This is precisely what the function `step` presented in Fig.2 does (which corresponds to the *for-loop* from line 8 to line 12 of EQUIVP's pseudocode).

```
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).

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.

Definition step (H S:set (Drv α β))(sig: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 (Γ,Δ) =>
   if c_of_Drv _ _ (Γ,Δ) then
    let H' := add (Γ,Δ) H in
     let S' := remove (Γ,Δ) S in
      let ns :=  Drv_pdrv_set_filtered α β (Γ,Δ) H' sig in
       ((H',ns ∪ S'),proceed α β)
   else
    ((H,S),termfalse α β (Γ,Δ))
 end.
```

Fig. 2: The function `step`.

The `step` function proceeds as follows: it obtains a pair (Γ, Δ) from the set S, generates new derivatives by a symbol $a \in \Sigma$

$$(\Lambda, \Theta) = (\partial_a(\Gamma), \partial_a(\Delta))$$

and adds to S all the (Λ, Θ) that are not elements of $\{(\Gamma, \Delta)\} \cup H$. This is implemented by `Drv_pdrv_set_filtered` which prevents the whole process from entering potential infinite loops since each derivative is considered only once during the execution of EQUIVP and the overall number of derivatives is finite. The return type of `step` is

```
((set (Drv α β) * set (Drv α β)) * step_case).
```

The first component corresponds to the pair (H,S), constructed as described above. The second component is a term of type `step_case` which guides the iterative process of computing the equivalence of the derivatives of α and β: if it is the term `proceed`, then the iterative process should continue; if it is a term `termtrue` H then the process should terminate and H contains the set of all the

derivatives of α and β. Finally, if it is a term `termfalse` (Γ, Δ), then the process should terminate. The pair (Γ, Δ) is a witness that $\alpha \not\sim \beta$, since $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$.

Implementation and Termination of EQUIVP

The formalisation of EQUIVP in the Coq proof assistant is presented in Fig.4, and corresponds to the function `equivP`. Its main component is the function `iterate` which is responsible for the iterative process of calculating the derivatives of α and β, or to find a witness that $\alpha \not\sim \beta$ if that is the case. The function `iterate` executes recursively until `step` returns either a term `termtrue` H, or returns a term `termfalse` (Γ, Δ). Depending on the result of `step`, the function `iterate` returns a term of type `term_cases`, which can be the term `Ok` H indicating that $\alpha \sim \beta$, or the term `NotOk` (Γ, Δ) indicating that $\alpha \not\sim \beta$, respectively.

A peculiarity of the Coq proof assistant is that it only accepts terminating functions, and more precisely, it only accepts *structurally decreasing* functions. Nevertheless, *provably terminating functions* can be expressed via encoding into structural recusive functions. The `Function` [33] command helps users to define such functions which are not structurally decreasing along with an evidence of its termination, as an illustration of the *certified programming paradigm* that Coq promotes. In the case of `iterate` such evidence is given by the proof that its recursive calls follow a *well-founded relation*.

The decreasing measure (of the recursive calls) for `iterate` is defined as follows: in each recursive call the cardinal of the accumulator set H increases by one element due to the computation of `step`. This increase of H can occur only less than $2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1$ times, due to the upper bounds of the cardinalities of $PD(\alpha)$ and of $PD(\beta)$. Therefore, in each recursive call of `iterate`, if `step` $H\ S\ \_ = (H', \_, \_)$ then the following condition holds:

$$(2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - |H'| < (2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - |H| \quad (7)$$

The relation `LLim` presented in Fig.3 defines the decreasing measure imposed by equation (7). Furthermore, the definition of `iterate` requires an argument of type `DP` α β which determines that the sets H and S are invariably disjoint along the computation of `iterate` which is required to ensure that the set H is always increased by one element at each recursive call.

Besides the requirement of defining `LLim` to formalise `iterate`, we had to deal with two implementation details: first, we have used the type `N` which is a binary representation of natural numbers provided by Coq's standard library, instead of the type `nat` so that the computation of `MAX` becomes feasible for large natural numbers. The second detail is related to the computation over terms representing well founded relations: instead of using the proof `LLim_wf` directly in `iterate`, we follow a technique proposed by Bruno Barras that uses the proof returned by the call to the function `guard`, that lazily adds 2^n constructors `Acc_intro` in front of `LLim_wf` so, that the actual proof is never reached in practice, while

```
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).

Section WfIterate.
  Variables α β : re.

  Definition MAX_fst := |α|_Σ + 1.
  Definition MAX_snd := |β|_Σ + 1.

  Definition MAX := (2^MAX_fst × 2^MAX_snd) + 1.
  Definition LLim := lim_cardN (Drv α β) MAX.

  Theorem LLim_wf : well_founded LLim.

  Fixpoint guard (n : nat)(wfp : well_founded (LLim)) : well_founded (LLim):=
   match n with
   |O => wfp
   |S m => fun x => Acc_intro x (fun y _ => guard m (guard m wfp) y)
   end.

End WfIterate.
```

Fig. 3: The decreasing measure of `iterate`.

maintaining the same logical meaning. This technique avoids normalisation of well founded relation proofs which is usually highly complex and may take too much time to compute.

Finally, the function `equivP` is defined as a call to `equivP_aux` with the correct input, *i.e.*, with the accumulator set $H = \emptyset$ and with the working set $S = \{(\{\alpha\}, \{\beta\})\}$. The function `equivP_aux` is a wrapper that pattern matches over the term of type `term_cases` returned by `iterate` and returns the corresponding Boolean value.

Correctness and Completeness

To prove the correctness of `equivP` we must prove that, if `equivP` returns `true`, then `iterate` generates all the derivatives and prove that all these derivatives agree on the *ewp* of its components. To prove that all derivatives are computed, it is enough to ensure that the `step` function returns a new accumulator set H' such that:

$$\texttt{step}\ H\ S\ sig = (H', S', \_) \rightarrow \forall (\Gamma, \Delta) \in H', \forall a \in \Sigma, \partial_a(\Gamma, \Delta) \in (H' \cup S') \quad (8)$$

The predicate `invP` and the lemma `invP_step` presented in Fig.5 prove this property. This means that, in each recursive call to `iterate`, the sets H and S hold all the derivatives of the elements in H. At some point of the execution, by the finiteness of the number of derivatives, H will contain all such derivatives and S will eventually become empty. Lemma `invP_iterate` proves this fact by a proof by functional induction over the structure of `iterate`. From lemma

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

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.

Lemma DP_upd : ∀ (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))).

Lemma DP_wf : ∀ (h s : set (Drv r1 r2)) (sig : set A),
      DP _ _ h s → snd (StepFast' _ _ h s sig) = Process' _ _ →
      LLim _ _ (fst (fst (StepFast' _ _ h s sig))) h.

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
   |progress    => iterate α β H' S' sig (DP_upd α β H S sig D)
  end.
Proof.
 (* Now comes the proof that LLim is a decreasing measure for iterate *)
 abstract(apply DP_wf).
 (* Now comes the proof that LLim is a well founded relation. *)
 exact(guard r1 r2 100 (LLim_wf r1 r2)).
Defined.

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

Definition mkDP_ini : DP α β ∅ {Drv_1st α β}.
(* Now comes the proof that {({α},{β})} ∩ ∅ = ∅ and that ε(∅) = true *)
abstract(constructor;[split;intros;try (inversion H)|vm_compute];reflexivity).
Defined.

Definition equivP (α β:re)(sig:set A) :=
 equivP_aux α β ∅ {Drv_1st α β} sig (mkDP_ini α β).
```

Fig. 4: Implementation of `equivP`.

`invP_equivP` we can prove that

$$\forall w \in \Sigma^\star,\ (\partial_w(\alpha), \partial_w(\beta)) \in \texttt{equivP}\ \{(\{\alpha\},\{\beta\})\}\ \emptyset\ \Sigma \tag{9}$$

by induction on the length of the word w and using the invariants presented above.

To finish the correctness proof of `equivP` one needs to make sure that all the derivatives (Γ, Δ) verify the condition $\varepsilon(\Gamma) = \varepsilon(\Delta)$. For that, we have defined the predicate `invP_final` which strengthens the predicate `invP` by imposing that the previous property is verified. The predicate `invP_final` is proved to be an invariant of `equivP` and this implies *re* equivalence by equation (6), as stated by theorem `invP_final_eq_lang`.

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

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

Lemma invP_iterate : ∀ H S sig D,
 invP H S sig → invP (iterate α β H S sig D) ∅ sig.

Lemma invP_equivP :
 invP (equivP α β Σ) ∅ Σ.

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

Lemma invP_final_eq_lang :
 invP_final α β (equivP α β Σ) ∅ Σ → α ∼ β.

Theorem equivP_correct : ∀ α β, equivP α β sigma = true → α ∼ β.
Theorem equivP_complete : ∀ α β, α ∼ β → equivP α β sigma = true.
Theorem equivP_correct_dual : ∀ α β, equivP α β sigma = false → α ≁ β.
Theorem equivP_complete_dual : ∀ α β, α ≁ β → equivP α β sigma = false.
```

Fig. 5: Invariants of `step` and `iterate`.

For the completeness, it is enough to reason by contradiction: assuming that $\alpha \sim \beta$ then it must be true that $\forall w \in \Sigma^\star, \varepsilon(\partial_w(\alpha)) = \varepsilon(\partial_w(\beta))$ which implies that `iterate` may not return a set of pairs that contain a pair (Γ, Δ) such that $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$ and so, `equivP` must always answer `true`.

Using the lemmas `equivP_correct` and `equivP_correct_dual` of Fig.5 a tactic was developed to prove automatically the (in)equivalence of any two *re*'s α and β. This tactic works by reducing the logical proof of the (in)equivalence of *re*'s into a Boolean equality involving the computation of `equivP`. After effectively computing `equivP` into a Boolean constant, the rest of the proof amounts at applying the reflexivity of Coq's primitive equality. Note that this tactic is also able to solve *re* containment due to the equivalence $\alpha \leq \beta \leftrightarrow \alpha + \beta \sim \beta$.

4 Performance

Although the main goal of our development was to provide a certified evidence that the decision algorithm suggested by Almeida *et. al.* is correct, it is of obvious interest to infer the level of usability of `equivP` (and corresponding tactic) for conducting proofs involving *re*'s (in-)equivalence within the Coq proof assistant. We have cerried out two types of performance evaluation of the decision procedure. The first evaluation consisted in experimenting[3] the tactic developed

[3] The experiments were conducted on a Virtual Box environment with six cores and 8 Gb of RAM, using coq-8.3pl4. The virtual environment executes on a dual six-core processor AMD Opetron(tm) 2435 processor with 2.60 GHz, and with 16 Gb of RAM.

over data sets of 10000 pairs of uniform-randomly generated *re*'s using the FAdo tool [34] so that the results are statistically relevant. Some results are presented in the table below. The value n is the size of the syntactic tree[4] of each *re*'s generated. The value of k is the number of symbols of the alphabet. The columns *eq* and *ineq* are the average time (in seconds) spent to decide equivalence and inequivalence of two *re*'s, respectively. The column *iter* is the average number of recursive calls needed for `equivP` to terminate. The equivalence tests were performed by comparing a *re* with itself, whereas the inequivalence tests were performed by comparing two consecutive *re*'s randomly generated, with the same value of n.

k	$n = 25$				$n = 50$				$n = 100$			
	eq	*iter*	*ineq*	*iter*	*eq*	*iter*	*ineq*	*iter*	*eq*	*iter*	*ineq*	*iter*
10	0.142	9.137	0.025	1.452	0.406	16.746	0.033	1.465	1.568	34.834	0.047	1.510
20	0.152	9.136	0.041	1.860	0.446	16.124	0.060	1.795	1.028	30.733	0.081	1.919
30	0.163	9.104	0.052	2.060	0.499	15.857	0.080	2.074	1.142	29.713	0.112	2.107
40	0.162	9.102	0.056	2.200	0.456	15.717	0.105	2.178	0.972	29.152	0.148	2.266
50	0.158	9.508	0.065	2.392	0.568	15.693	0.125	2.272	1.182	28.879	0.170	2.374

In the second evaluation[5] have compared the performance of our development with the developments [15, 19, 17, 25]. The results are presented below. The equivalence tests $A(n, m, o) \equiv (a^o)a^\star + (a^n + a^m)^\star \sim (a^n + a^m)^\star$ and the equivalence tests $B(n) \equiv (\epsilon + a + aa + \ldots + a^{n-1})(a^n)^\star \sim a^\star$ were borrowed from [25]. The test $C(n)$ is the equivalence $\alpha_n \sim \alpha_n$, for $\alpha_n = (a + b)^\star(a(a + b)^n)$. The entries "–" and "$\geq 600s$" refer, respectively, to tests that were not performed and tests which took more than 10 minutes to finish[6].

	$A(n, m, o)$			$B(n)$			$C(n)$		
	$(4, 5, 12)$	$(5, 6, 20)$	$(5, 7, 24)$	18	100	500	5	10	15
`equivP`	0.15	0.20	0.29	0.05	1.30	37.06	1.31	77.61	$\geq 600s$
[15]	0.01	0.01	0.02	0.03	1.33	50.53	0.04	3.06	15.26
[19]	2.78	2.94	2.88	2.94	3.37	158.94	2.87	$\geq 600s$	$\geq 600s$
[17]	8.73	46.70	102.69	98.84	$\geq 600s$	$\geq 600s$	28.97	$\geq 600s$	$\geq 600s$
[25]	0.24	0.43	0.57	0.80	–	–	–	–	–

The development of Braibant and Pous is globally the more efficient in the tests we have selected, thanks to their *reification* mechanism and efficient representation of automata. `equivP` is able to outperform it only for the family of equivalences $B(n)$. When compared to the other formalizations, `equivP` exhibits better performances, which suggests that algorithms based on partial derivatives should be considered wrt. to other approaches.

[4] This corresponds to the sum of the number of constants, symbols and regular operators of the *re*.

[5] These tests were performed in a Macbook Pro 13", with a 2.53 GHz Inter Core 2 Duo, with 4 GB 1067 MHz DD3 of RAM memory, using coq-8.3pl4.

[6] The times for [25] are the ones given in the refered article.

5 Concluding Remarks and Applications

In this paper we have described the formalisation, within the Coq proof assistant, of the procedure EQUIVP for deciding *re* equivalence based in partial derivatives. This procedure has the advantage of not requiring the normalisation modulo *ACI* of *re*'s in order to prove its termination. We presented some performance tests and comparisons with similar approaches that suggest the acceptable behavior of our decision procedure. However, there is space for improvement of its performance and more throughout comparisons with the other developments should take place.The purpose of this research is part of a broader project where the equivalence of Kleene algebra with tests (KAT) terms is used to reason about the partial correctness of programs [35]. The development in [26] is a mechanization of KAT in the Coq proof assistant containing the extention of the decision procedure here presented for KAT terms (in-)equivalence.

Acknowledgments: We thank the anonymous referees for their constructive comments and criticisms, from which this paper has clearly benefited.

References

1. Filliâtre, J.C.: Finite Automata Theory in Coq: A constructive proof of Kleene's theorem. Research Report 97–04, LIP - ENS Lyon (February 1997)
2. Kleene, S. In: Representation of Events in Nerve Nets and Finite Automata. Shannon, C. and McCarthy, J. edn. Princeton University Press 3–42
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
4. Höfner, P., Struth, G.: Automated reasoning in Kleene algebra. In Pfenning, F., ed.: CADE 2007. Number 4603 in LNAI, Springer-Verlag (2007) 279–294
5. McCune, W.: Prover9 and Mace4. `http://www.cs.unm.edu/smccune/mace4`. Access date: 1.10.2011.
6. Moreira, N., Pereira, D.: KAT and PHL in Coq. CSIS **05**(02) (December 2008) ISSN: 1820-0214.
7. Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems **19**(3) (May 1997) 427–443
8. Almeida, J.B., Moreira, N., Pereira, D., Melo de Sousa, S.: Partial derivative automata formalized in Coq. In Domaratzki, M., Salomaa, K., eds.: CIAA 2010. Number 6482 in LNCS, Springer-Verlag (2011) 59–68
9. Almeida, M., Moreira, N., Reis, R.: Antimirov and Mosses's rewrite system revisited. Int. J. Found. Comput. Sci. **20**(4) (2009) 669–684
10. Antimirov, V.M., Mosses, P.D.: Rewriting extended regular expressions. In Rozenberg, G., Salomaa, A., eds.: DLT, World Scientific (1994) 195 – 209
11. Hopcroft, J., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Technical Report TR 71 -114, University of California, Berkeley, California (1971)
12. Almeida, M., Moreira, N., Reis, R.: Testing regular languages equivalence. JALC **15**(1/2) (2010) 7–25
13. Almeida, M.: Equivalence of regular languages: an algorithmic approach and complexity analysis. PhD thesis, FCUP (2011) `http://www.dcc.fc.up.pt/~mfa/thesis.pdf`.

14. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: The average transition complexity of Glushkov and partial derivative automata. In Mauri, G., Leporati, A., eds.: 15th DLT 2011 Proc. Volume 6795 of LNCS., Springer (2011) 93–104
15. Braibant, T., Pous, D.: An efficient Coq tactic for deciding Kleene algebras. In: Proc. 1st ITP. Volume 6172 of LNCS., Springer (2010) 163–178
16. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Infor. and Comput. **110**(2) (May 1994) 366–390
17. Coquand, T., Siles, V.: A decision procedure for regular expression equivalence in type theory. In Jouannaud, J.P., Shao, Z., eds.: CPP 2011, Kenting, Taiwan, December 7-9, 2011. Number 7086 in LNCS, Springer-Verlag 119–134
18. Brzozowski, J.A.: Derivatives of regular expressions. JACM **11**(4) (1964) 481–494
19. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. Journal of Automated Reasoning (2011) Published online.
20. Rutten, J.J.M.M.: Automata and coinduction (an exercise in coalgebra). In Sangiorgi, D., de Simone, R., eds.: CONCUR. Volume 1466 of LNCS., Springer (1998) 194–218
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
22. Komendantsky, V.: Reflexive toolbox for regular expression matching: verification of functional programs in Coq+Ssreflect. In Claessen, K., Swamy, N., eds.: PLPV, ACM (2012) 61–70
23. Komendantsky, V.: Computable partial derivatives of regular expressions. `http://www.cs.st-andrews.ac.uk/~vk/papers.html`
24. Mirkin, B.: An algorithm for constructing a base in a language of regular expressions. Engineering Cybernetics **5** (1966) 110–116
25. Asperti, A.: A compact proof of decidability for regular expression equivalence. In Beringer, L., Felty, A., eds.: Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Number 7406 in LNCS, Springer-Verlag
26. Moreira, N., Pereira, D., Melo de Sousa, S.: Deciding KAT terms equivalence in Coq. Technical Report DCC-2012-04, DCC-FC & LIACC, Universidade do Porto (2012)
27. Hopcroft, J., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison Wesley (2000)
28. Champarnaud, J.M., Ziadi, D.: From Mirkin's prebases to Antimirov's word partial derivatives. Fundam. Inform. **45**(3) (2001) 195–205
29. Moreira, N., Pereira, D., Melo de Sousa, S.: Source code of the formalization. `http://www.liacc.up.pt/~kat/equivP.tgz`
30. Paulin-Mohring, C.: Inductive definitions in the system Coq: Rules and properties. Proceedings of the International Conference on Typed Lambda Calculi and Applications **664** (1993) 328–345
31. The Coq Development Team. `http://coq.inria.fr`
32. Lescuyer, S.: First-class containers in coq. Studia Informatica Universalis **9**(1) (2011) 87–127
33. Barthe, G., Courtieu, P.: Efficient reasoning about executable specifications in Coq. In Carreño, V., Muñoz, C., Tahar, S., eds.: TPHOLs. Volume 2410 of LNCS., Springer (2002) 31–46
34. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: tools for automata manipulation and visualization. In Maneth, S., ed.: Proc. 14th CIAA'09. Volume 5642. (2009) 65–74
35. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Transactions on Computational Logic (TOCL) **1**(1) (2000) 60–76