Computing Certificates of Regular Expression Equivalence

Benoit Razet\(^1\) and Bodhayan Roy\(^2\)

\(^1\) Bucknell University benoit.razet@bucknell.edu
\(^2\) IIT Bombay bodhayan.roy@gmail.com

Abstract. Deciding the equivalence of regular expressions is a fundamental problem of theoretical computer science. On one hand, there are several decision procedures to solve this problem. On the other hand, there are several axiomatisations of the equivalence of regular expression. The axiomatisations are proved to be complete and the proofs consists of showing that each steps of a decision procedure correspond to using certain axioms from the axiomatisation. In principle, any axiomatisation that is proved to be complete, should correspond to an algorithm computing formal certificates. We present the design of a program to compute certificates of equivalence of regular expressions based on Salomaa’s axiomatisation. It includes a language to express certificates and a simple proof system to validate certificates. We implemented two decision procedures producing certificates. One based on Salomaa’s original proof of completeness and the other, more efficient, based on nondeterministic finite automata and their determinisation. We report on experiments producing and validating certificates for these two decision procedures. We hope that the same methodology would apply to other problems and axiomatisations related to Kleene Algebras.

1 Introduction

The concept of regular expressions is fundamental to computer science. They were studied as early as the 1950s by Kleene [Kle56]. They are used to study theoretical problems in the field of Programming Language Design or to study Computational Complexity problems. The problem is known to be PSPACE-complete [SM73]. Regular expressions are also used in practice for string manipulation techniques.

Every regular expression describes a set of strings and a problem that arises early on is to decide whether two regular expressions denote the same set of strings. This problem is called the regular expression equivalence problem. It was identified since the early work of Kleene [Kle56] and it has been solved in different ways since then. Kleene noticed a set of axioms for regular expressions equivalence but could not provide a complete description. The first solution to this problem was found by Salomaa [Sal66] who proposed a finite set of axioms and proved that his system was complete, which means that any two regular expressions denoting the same set of strings can be proved to be equal using only the axioms of the system.
Salomaa’s axiomatisation is one of several axiomatisations with different features. An interesting property of Salomaa’s axiomatisation is the finiteness of its set of axioms. The axioms are made of equations of the form $A = B$ where $A$ and $B$ are two generic regular expressions, and an additional axiom that is not of this form equational form, but of the more elaborated form (axiom $\text{EquaSol}$ in Figure 1). This particular axiom prevents Salomaa’s axiomatisation from enjoying nice algebraic properties and from being applicable to more models than just regular languages. Further research, initiated by Conway [Con71], led to two different axiomatisations. The first axiomatisation investigated by Bloom et. al. [BE93] consists of an infinite set of purely equational axioms. The impossibility of having a finite set of axioms and their being purely equational, was established by Redko [Red64]. The second axiomatisation was further investigated by Kozen [Koz94]. It consists of a finite set of equational and implicational axioms (of the form $A = B \Rightarrow C = D$). Nowadays, this is referred to as the axiomatisation of Kleene Algebras. It is also worth mentioning Pratt’s axiomatisation of Action Algebras [Pra90], a conservative extension of Kleene Algebras, which has the interesting property of being a purely equational finite axiomatisation, at the cost of adding two additional operators, the residuals.

In this article, we show how to lever the completeness of an axiomatisation to design of a program that produces certificate of equivalence based on the axiomatisation. The certificate can be independently checked by a simple proof system. Although we have the choice between several axiomatisations mentioned above, we found Salomaa’s axiomatisation – with its relatively simple proof of completeness – to be a good candidate for this endeavor.

We implemented two decision procedures producing certificates, one based on Salomaa’s original proof of completeness and a more efficient one based on the standard method of converting NFAs to DFAs. The source of the program is available [RR16]. In this article, we present the design of the proof system and the programs computing certificates. It is presented in a small subset of the functional programming language OCaml, such that it should be accessible to a wide audience with basic knowledge of functional programming, and should be easily translated to other programming languages.

The remainder of the article is organized as follows. Section 2 introduces regular expressions and Salomaa’s axiomatisation. Section 3 shows how the list of axioms can be implemented as rewriting rules over regular expressions. Section 4 explains the equational reasoning part of the axiomatisation and the notion of pointers to regular expressions as a key element of the implementation. Section 5 builds on top of the previous Sections to layout a simple proof system for Salomaa’s axiomatisation. Section 6 discusses Salomaa’s proof of completeness and the methodology to turn it into a program computing certificates of regular expression equivalence. We discuss the validation of our program through experiments in Section 7. Section 8 discusses related works and Section 9 concludes and proposes future works.
2 Salomaa’s axiomatisation

Let $\Sigma$ be a finite set of symbols, $\Sigma = \{a, b, c, \cdots\}$. A string is a finite sequence of symbols from $\Sigma$, $w = a_1a_2\cdots a_n$ is a string of length $n$. The concatenation of two strings $w_1$ and $w_2$ is written as $w_1 \cdot w_2$ or simply $w_1w_2$. The neutral element for the concatenation is the empty string, denoted as $\epsilon$. A language is defined as a set of strings, finite or infinite. Regular expressions are algebraic expressions to describe certain languages called regular languages. Regular expressions over the finite alphabet $\Sigma$ are defined inductively as the following:

- The constants 0 and 1 are regular expressions.
- Any symbol $x$ in $\Sigma$ is a regular expression.
- If $A$ and $B$ are two regular expressions then the union $A + B$, the concatenation $A \cdot B$ and the Kleene’s star $A^*$ are regular expressions.

There exists a natural function $L()$ mapping regular expressions to regular languages. $L()$ is defined inductively on the structure of the regular expression as:

- $L(0) = \emptyset$,
- $L(1) = \{\epsilon\}$,
- $L(A + B) = L(A) \cup L(B)$,
- $L(A \cdot B) = \{w_1w_2 \mid w_1 \in L(A) \land w_2 \in L(B)\}$

and for Kleene’s star operator, $L(A^*)$ is defined as the smallest language satisfying the equation $L(A^*) = \{\epsilon\} \cup L(A) \cdot L(A^*)$.

Regular expressions are straightforwardly defined in OCaml with a recursive datatype:

```ocaml
type regexp = Zero | One | Symb of char
  | Union of (regexp * regexp) | Conc of (regexp * regexp)
  | Star of regexp
```

The `regexp` datatype has six different constructors for each operator or constant. Without loss of generality, we use the primitive type for characters `char` for the alphabet $\Sigma$. As an example, the regular expression $a(b^* + c^*)^*$ would be encoded as:

```ocaml
let ex1 = Conc(Symb 'a', Star(Union(Star(Symb 'b'), Star(Symb 'c'))))
```

To illustrate the use of the `regexp` datatype, let us define the function `ewp` that decides whether a regular expression denotes a language containing the empty string or not. `ewp` stands for epsilon word property:

```ocaml
(* val ewp: regexp -> bool *)
let rec ewp e = match e with
  | Zero           -> false
  | One            -> true
  | Symb s         -> false
  | Union (e1,e2)  -> ewp e1 || ewp e2
  | Conc (e1,e2)   -> ewp e1 && ewp e2
  | Star e1        -> true
```

The function is recursively defined by pattern matching on the input regular expression `e` using the `match with` construct. The result of `ewp` on the regular expression `ex1` returns `false`. Note that most functions appearing in the remaining of the article are defined by pattern matching in a similar fashion.
Regular expressions form a structure \( \mathbb{RE}(\Sigma) = (\Sigma, 0, 1, +, \cdot, \ast) \) of an idempotent semiring and satisfies three more axioms related to Kleene’s star operation. Salomaa’s axiomatisation is given in Fig. 1. Every axiom is universally quantified over regular expression variables \( A, B, C \) or \( X \). All the axioms are *equational* except the last one and we qualify this one as *implicational*. Let us call the last two axioms respectively the *fixed point axiom* and the *implicational axiom*.³

### 3 Axioms as rewriting rules

Now we show how to turn Salomaa’s axiomatisation into a program rewriting regular expressions given an axiom. First we define a datatype enumerating all the axioms:

```ocaml
type axiom = AssocU1 | AssocU2 | IdentU1 | IdentU2 | CommutU
  | AssocC1 | AssocC2 | IdentCL | IdentCR
  | DistrL1 | DistrL2 | DistrR1 | DistrR2
  | AnnihiL | Idempot | EpsStar | Fixedpt | EquaSol
```

Each one of these axioms is going to act differently on a regular expression given that the expression has the correct form. The function that transforms a regular expression according to an equational axiom is given below.

```ocaml
let apply_eq_axiom ax e = match ax, e with
  | AssocU1, Union (a, Union (b,c)) -> Union (Union (a,b), c)
  | AssocU2, Union (Union (a,b), c) -> Union (a, Union (b,c))
  | AssocC1, Conc (a, Conc (b,c)) -> Conc (Conc (a,b), c)
  | AssocC2, Conc (Conc (a,b), c) -> Conc (a, Conc (b,c))
```

³ Our presentation differs from Salomaa’s in several minor points: Constant 1 corresponds to \( \phi^\ast \) and the axioms for the Kleene’s star are mirrored with respect to the concatenation, compare the fixed point axiom with Salomaa’s original \( \alpha^\ast = \phi^\ast + \alpha^\ast \alpha \).
This function is assuming that the application of an axiom of type `axiom` to a regular expression is possible. When this is not possible, because the input regular expression does not match an expected form, the function will throw an exception `assert false`. In general, each equational axiom can be interpreted as a left and a right rewriting rule. Notice that not every possible left or right rewriting rule is actually listed in the type `axiom` and function `apply_eq_axiom`. We leave some of rewriting rules out because they are unnecessary to write the certificates of equivalence.

The previous function takes care of all necessary equational axioms, we still have to encode the one implicational axiom `EquaSol`. This axiom works on equations as opposed to regular expressions. An equation can simply be encoded as a pair of regular expressions and the function implementing the application of the implicational axiom is the following:

```ocaml
type equality = (regexp * regexp)

let apply_impl_axiom ax eq = match ax, eq with
| EquaSol, (x1, Union (Conc (a,x2),b)) ->
  if x1 = x2 && not (ewp a) then (x1, Conc (Star a, b)) else assert false
| _ -> assert false
```

4 Equational reasoning using pointers

Now we show the part of proof system that implements the equational reasoning aspect of the axiomatisation. First, the equality is an equivalence relation, thus it is reflexive, transitive and symmetric:

\[
A = A \quad \text{(Reflex)}
\]
\[
A = B \Rightarrow B = A \quad \text{(Symm)}
\]
\[
[A = B \land B = C] \Rightarrow A = C \quad \text{(Trans)}
\]
Second, equalities may be used to rewrite inside regular expressions, like when one infers \((C+A)^* = (C+B)^*\) under the assumption \(A = B\). We need additional rules to substitute inside a regular expression using equalities. It is common practice to solve this problem by adding specific rules for each constructor. The traditional rules can be summarized as: If \(A = B\) then all the following equalities are valid, \(A + C = B + C\), \(C + A = C + B\), \(AC = BC\) and \(A^* = B^*\). An inconvenience of such rules is that the precondition \(A = B\) does not contain all the information necessary to derive the result, say \(A + C = B + C\), which also depends on \(C\). That is the reason why we designed an alternative solution.\(^4\) Moreover, our solution allows to follow more closely the conventional method of writing mathematical equations, manipulating iteratively the right-hand side of an equation. For example, suppose we have to prove the following equivalence:

\[
(aA + bB) + (bB + cC) = aA + (bB + cC)
\]

A natural proof goes along the lines of using a sequence of axioms of associativity, idempotence and distributivity\(^5\). A formal mathematical proof could be:

\[
\begin{align*}
(aA + bB) + (bB + cC) &= aA + (bB + (bB + cC)) \quad \text{(by AssocU2)} \\
 &= aA + ((bB + bB) + cC) \quad \text{(by AssocU1)} \\
 &= aA + ((b(B + B)) + cC) \quad \text{(by DistrL2)} \\
 &= aA + (bB + cC) \quad \text{(by Idempot)}
\end{align*}
\]

What is implicit in this reasoning is the exact position where each axiom is applied. The reader has no difficulty filling this gap by inferring the missing information, given the axiom being used and given the equations before and after the axiom is used. However, we have to make explicit this information to our program. Regular expressions need to be extended with a notion of pointer. The pointer points to a specific subexpression in a regular expression. The pointer may move at each step of the reasoning in order to point to the next subexpression to be rewritten. To illustrate this concept, let us revise the previous example to demonstrate the use of pointers. A bullet symbol • shows which subexpression is pointed by the pointer. It moves locally inside the regular expression, using the instructions Right, Left and Up.

Example 1.

\[
\begin{align*}
(aA + bB) + (bB + cC) &= ((aA + bB) + (bB + cC))^* \quad \text{(by Reflex)} \\
 &= aA + (bB + (bB + cC))^* \quad \text{(by AssocU2, Right)} \\
 &= aA + ((bB + bB)^* + cC) \quad \text{(by AssocU1, Left)} \\
 &= aA + ((b(B + B))^* + cC) \quad \text{(by DistrL2, Right)} \\
 &= (aA + (bB + cC))^* \quad \text{(by Idempot, Up, Up, Up)}
\end{align*}
\]

\(^4\) This is related to the subformula property in formal logic.

\(^5\) The axiom of distributivity is actually not necessary, but we thought it would be helpful to use more axioms in this example.
To implement the pointer feature, we adapt the standard technique known as the Zipper [Hue97] to regular expressions. The Zipper is a generic applicative method to define a notion of pointer for algebraic datatypes. The key idea is to see a pointer to a subtree in a treelike structure as a pair whose first element is a linked list encoding the path from the subtree to the root of the tree, and the second element of the pair is the subtree itself. The zipped version of the `regexp` type is the type `zregexp` defined using the `path` type:

```
type path = Top
  | LeftUnion of (path * regexp) | RightUnion of (regexp * path)
  | LeftConc of (path * regexp) | RightConc of (regexp * path)
  | DownStar of path

type zregexp = (path * regexp)
```

In order to understand these datatypes, we need to look at the function that moves the pointer according to a command:

```
type regexp_move = Left | Right | Down | Up

let apply_re_move cmd (path,e) =
  match cmd, e with
  | Left, Union (e1,e2) -> (LeftUnion (path,e2) ,e1)
  | Left, Conc (e1,e2) -> (LeftConc (path,e2) ,e1)
  | Right, Union (e1,e2) -> (RightUnion (e1,path) ,e2)
  | Right, Conc (e1,e2) -> (RightConc (e1,path) ,e2)
  | Down, Star e1 -> (DownStar path, e1)
  | Up, _ ->
    match path with
    | LeftUnion (path1, e2) -> (path1, Union(e,e2))
    | RightUnion (e1, path2) -> (path2, Union(e1,e))
    | LeftConc (path1, e2) -> (path1, Conc(e,e2))
    | RightConc (e1, path2) -> (path2, Conc(e1,e))
    | DownStar path1 -> (path1, Star e)
    | Top -> assert false
  | _, _ -> assert false
```

We also alter the type `equality` to be able to point inside the right-hand side expression of an equality:

```
type zequality = ZEq of (regexp * zregexp)
```

As an example, we use the type `zequality` to define the function applying any axiom from Salomaa’s axiomatisation, whether it is equational or implicational:

```
let apply_sal_ax ax zeq =
  match ax, zeq with
  | EquaSol, ZEq (e1,(Top,e2)) -> let eq2zeq (e1,e2) = ZEq (e1,(Top,e2)) in
    eq2zeq (apply_impl_axiom ax (e1,e2))
  | ax , ZEq (e1,(path,e2)) -> ZEq (e1,(path, apply_eq_axiom ax e2))
```

Now that we have introduced the notion of pointer inside regular expressions, we are ready to implement the equational reasoning part of the proof system. The datatype for the axioms of equational reasoning is an enumerated type:

```
type equiv_axiom = Reflex of regexp | Symm | Trans | Subst
```
The first constructor for the axiom of reflexivity is parametrized by a regular expression. The other axioms for symmetry, transitivity and substitution need not be parametrized. Their implementation is described as follows by the function `apply_eq_ax`:

```ml
let apply_eq_ax ax eq1 eq2 =
match ax, eq1, eq2 with
| Reflex a, _, _ -> ZEq (a,(Top,a))
| Symm, _, Some (ZEq (a,(Top,b))) -> ZEq (b,(Top,a))
| Trans, Some (ZEq (c,(Top,d))), Some (ZEq (a,(Top,b))) ->
  if b = d then ZEq (a,(Top,c)) else assert false
| Subst, Some (ZEq(e1,(Top,e2))), Some (ZEq (e,(path,e3))) ->
  if e1 = e3 then ZEq (e,(path,e2)) else assert false
| _ -> assert false
```

This function applies an axiom to produce an equality, given at most two equalities passed as the arguments `eq1` and `eq2` to the function. Depending on the axiom being applied, an argument may be absent, which is encoded using a `option` type.

### 5 A simple proof system

Let us piece together the different components we have described so far, and define a simple proof system for Salomaa’s axiomatisation.

A proof environment is described as a sequent $\Gamma \vdash A = B$ where $\Gamma$ is a set of equations that have been proven successfully in the past, and $A = B$ is the goal that is currently worked on. When one of the axioms of substitution or transitivity is applied, we need to indicate specifically one more equation than the current goal. In the sequent notation, it would have the form $\Gamma, C = D \vdash A = B$. This is the reason why there are two values of type `zequality option` in the triple encoding a sequent given below. The first one is for this optional equality $(C = D)$, and the one in the last position of the triple is the goal of the sequent $(A = B)$.

```ml
type proof_env = (zequality list * zequality option * zequality option)
```

To start a proof without any assumption, the environment is initially defined as:

```ml
let init_env = ([], None, None)
```

Three operations are needed for manipulating a proof environment. We need to load an equation from $\Gamma$ in the goal position (`LoadGoal`) or in the additional argument position (`LoadMore`), and we need to store the current goal in $\Gamma$:

```ml
type env_command = LoadGoal of regexp | LoadMore of regexp | Store
```

The function `apply_env_cmd` applies a command to an environment. It uses a helper function `find` that finds in the list of equations `eqs` the first equation that has a right-hand side equal to an input expression `e`.

```ml
let apply_env_cmd cmd (eqs, more, goal) =
let rec find eqs e = match eqs with
```
ZEq (e1, ze) :: eqs ->
if e1 = e then Some (ZEq (e1,ze)) else find eqs e in
match cmd with
| LoadGoal e -> (eqs, more, find eqs e)
| LoadMore e -> (eqs, find eqs e, goal)
| Store -> match goal with
  | Some eq -> (eq :: eqs, more, None)
  | _ -> assert false

Now we are ready to define the datatype for certificates. A certificate is a list of proof steps, and a proof step is a disjoint union of types we have encountered:

```ocaml
type proof_step = ReMv of regexp_move | SalAx of axiom
  | EqAx of equiv_axiom | EnvCmd of env_command
```

declared in the previous example. The proof checker is implemented as a function verifying each step of the proof to modify the environment according to the sequence of axioms in the certificate.

```ocaml
let onestep ax (eqs, more, goal) = match ax, goal with
| ReMv mv, Some (ZEq (e,ze)) -> let newze = apply_re_move mv ze in
  (eqs, more, Some (ZEq (e,newze)))
| SalAx ax, Some zeq -> (eqs, more, Some (apply_sal_ax ax zeq))
| EqAx ax, _ -> (eqs, more, Some (apply_eq_ax ax more goal))
| EnvCmd c, _ -> apply_env_cmd c (eqs,more,goal)
| _, _ -> assert false

let rec verify_all cert (eqs,more,goal) = match cert with
| [] -> (eqs,more,goal)
| ax :: axs -> let env = onestep ax (eqs,more,goal) in verify_all axs env

let verify_proof cert = verify_all cert init_env
```

The proof of equivalence discussed previously as Example 1 is fully described by the following sequence of axioms. The sequence constitutes the certificate which is verified by executing the function `verify_proof` on it:

```ocaml
let exi = let aA = Conc (Symb 'a', Symb 'A') in
  let bB = Conc (Symb 'b', Symb 'B') in
  let cC = Conc (Symb 'c', Symb 'C') in
  Union (Union (aA, bB), Union (bB, cC))

let proof1 = [EqAx (Reflex exi); SalAx AssocU2; ReMv Right; SalAx AssocU1;
  ReMv Left; SalAx DistrL2; ReMv Right; SalAx Idempot; ReMv Up; ReMv Up; ReMv Up]

verify_proof proof1;;
```

Notice that the first axiom to be used is the axiom of reflexivity when the environment is empty. The output of the proof checker after executing each axiom/command in the certificate is the following where the brackets `< E >@
plays the role of the bullet ⋅, pointing at various positions in the right-hand side of equalities.

| |- (aA + bB) + (bB + cC) = <(aA + bB) + (bB + cC)>@
| |- (aA + bB) + (bB + cC) = <aA + (bB + (bB + cC))>@
| |- (aA + bB) + (bB + cC) = (aA + (bB + (bB + cC))@
| |- (aA + bB) + (bB + cC) = (aA + (<bB + bB>)@ + cC)
| |- (aA + bB) + (bB + cC) = (aA + (<b(B + B)>@ + cC))
| |- (aA + bB) + (bB + cC) = (aA + (<bB>@ + cC))
| |- (aA + bB) + (bB + cC) = (aA + (b<B + B>@ + cC))
| |- (aA + bB) + (bB + cC) = (aA + (b<B>@ + cC))
| |- (aA + bB) + (bB + cC) = (aA + <bB + cC>@)

6 Computing certificates from a proof of completeness

Let $A$ and $B$ be two regular expressions. Salomaa’s proof of completeness proceeds in three successive steps, and our program follows this same design:

1. Compute systems of characteristic equations for both $A$ and $B$.
2. Duplicate equations in both systems such that the systems become equivalent.
3. Solve both systems simultaneously in order to find a common regular expression $U$ such that $A = U = B$.

The first step is considered as the most important part of the proof of completeness.

**Definition 1 ([Sal66] page 162).** A regular expression $A$ is equationally characterized if there exists a finite set of regular expressions $A_1, \ldots, A_n$ such that $A$ is $A_1$ and such that for each $i \leq n$, the following equivalence is provable:

\[ \vdash A_i = \sum_{a \in \Sigma} aA_{i,a} + \delta_{A_i} \]

where $\delta_{A_i}$ is either 0 or 1, and for each $i \leq n$ and $a \in \Sigma$, there exists a $k \leq n$ such that $A_{i,a}$ is $A_k$.

Note that a system of characteristic equations corresponds to a deterministic finite state automaton (DFA). The regular expressions $A_i$ are the states of the DFA, the transitions starting from a state $A_i$ are encoded as the summation $\sum_{a \in \Sigma} aA_{i,a}$, and a state $A_i$ is accepting if $\delta_{A_i}$ is 1. The definition differs slightly from the original one by Salomaa by mirroring the concatenation to describe an automaton recognizing strings from left to right instead of right to left.

The regular expression appearing on the right-hand side of a characteristic equation has a very specific form. We have the option to use the zequation type to encode these specific regular expressions, but we prefer to encode such specific equations using a more adequate structure:
type characteristic_eq = (regexp * (char * regexp) list * bool)

where the first element is $A_i$, the second element which is the list of pairs of characters and regular expressions encoding $\sum_{a \in \Sigma} a A_i.a$, and the third element is the for the boolean of acceptance $\delta_{A_i}$.

Lemma 1 ([Sal66] Lemma 4). Every regular expression is equationally characterized.

The proof of this lemma is by induction on the structure of the regular expression. That is, we need to prove that the base case regular expressions 0, 1 and $a$ are equationally characterized, and that assuming $A$ and $B$ are two regular expressions being equationally characterized by systems $A_i$ with $i \leq n$ and $B_j$ with $j \leq m$ then $A + B$, $A B$ and $A^*$ are all equationally characterized. The three base cases are straightforward. Let us show how to compute certificates building a system of characteristic equations in the case of $A + B$. We do that by proving that for any pair of regular expressions $A_u$ and $B_v$, then $A_u + B_v$ is equationally characterized. Let $E_1$ be the equation $A_u = \sum_{a \in \Sigma} a A_u.a + \delta_{A_u}$ and $E_2$ be the equation $B_v = \sum_{a \in \Sigma} a B_v.a + \delta_{B_v}$, the sequence of steps are the following:

$$E_1, E_2 \vdash A_u + B_v = A_u + B_v$$

(reflex)

$$\quad = \sum_{a \in \Sigma} a A_u.a + \delta_{A_u} + \sum_{a \in \Sigma} a B_v.a + \delta_{B_v}$$

(load_and_subst)

$$\quad = \sum_{a \in \Sigma} a A_u.a + \sum_{a \in \Sigma} a B_v.a + \delta_{A_u} + \delta_{B_v}$$

(rearrange)

$$\quad = \sum_{a \in \Sigma} a A_u.a + \sum_{a \in \Sigma} a B_v.a + \delta$$

(simp1_delta)

$$\quad = \sum_{a \in \Sigma} a (A_u.a + B_v.a) + \delta$$

(merge_sums)

The sequence of steps above shows that any expression $A_u + B_v$ has a characteristic equation. Applying this sequence of steps on any of the $n \times m$ regular expressions of the form $A_i + B_j$ leads to a system of characteristic equations, and since $A_1 + B_1$ is $A + B$, we conclude that $A + B$ is equationally characterized.

We show how to translate the sequence of steps above into a program computing them. The reader may refer to the names used to label each equation in the reasoning above. The names correspond to the encoding of either a certificate, or a function computing the certificate. We present each of them in order:

let reflex e1 e2 = [ EqAx (Reflex (Union (e1, e2))) ]

let load_and_subst e1 e2 = [ ReMv Left ; EnvCmd (LoadMore e1) ; EqAx Subst ; ReMv Up ; ReMv Right ; EnvCmd (LoadMore e2) ; EqAx Subst ; ReMv Up ]

let rearrange = [ SalAx AssocU2 ; ReMv Right ; SalAx AssocU1 ; ReMv Left ; SalAx CommutU ; ReMv Up ; SalAx AssocU2 ; ReMv Up ; SalAx AssocU1 ]

The two following helper functions will be convenient to write the next certificates:
let applyl cert = [ReMv Left]@cert@[ReMv Up]
let applyr cert = [ReMv Right]@cert@[ReMv Up]

The simplification of the sum of booleans $\delta_A u + \delta_B v$ is done by the following function returning $\delta$ with the certificate producing it:

let simpl_delta b1 b2 = match (b1,b2) with
| (false,false) -> (false, [ SalAx IdentU2 ])
| (true,false) -> (true, [ SalAx IdentU2 ])
| (false,true) -> (true, [ SalAx CommutU ; SalAx IdentU2 ])
| (true,true) -> (true, [ SalAx Idempot ])

The most difficult function is the one merging the two summations $\sum_{a\in\Sigma} aA_{u,a} + \sum_{a\in\Sigma} aB_{v,a}$. With a generic alphabet $\Sigma$ and assuming that both summations are encoded as lists lst1 and lst2 enumerated in the same order with respect to $a \in \Sigma$, the following function computes the resulting list of transitions and the certificate producing it:

let rec merge_sums lst1 lst2 = match lst1, lst2 with
| [[a,x]], [[a1,y]] -> ([(a1,Union (x,y))], [ SalAx DistrL2 ])
| (a,x)::xs, (a1,y)::ys -> let (xs_ys, cert_xs_ys) = merge_sums xs ys in
  let lst = (a1,Union (x,y)) :: xs_ys in
  let pcert = [ SalAx AssocU2 ; ReMv Right ; SalAx AssocU1
               ; ReMv Left ; SalAx CommutU ; ReMv Up ; SalAx AssocU2 ; ReMv Up
               ; SalAx AssocU1 ; ReMv Left ; SalAx DistrL2 ; ReMv Up ] in
  let cert = pcert @ (applyr (cert_xs_ys)) in
  (lst,cert)
|_ -> assert false

The function merge_sums recursively adds up the piece of certificate named pcert, in the same fashion as one zips two lists into a list of pairs. Finally, we assemble together all these certificates in order to define the function that takes two characteristic equations, respectively for $A_u$ and $B_v$, and returns the characteristic equation for $A_u + B_v$ together with the corresponding certificate:

let merge_union (e1,lst1,b1) (e2,lst2,b2) = let cert1 = reflex e1 e2 in
  let cert2 = load_and_subst e1 e2 in
  let cert3 = rearrange in
  let (b,cert4) = simpl_delta b1 b2 in
  let (lst,cert5) = merge_sums lst1 lst2 in
  let cert = cert1@cert2@cert3@cert4@applyr (cert5) in
  ((Union(e1,e2),lst,b), cert)

This concludes the description of the part of our program computing a system of characteristic equations for $A_u + B_v$. The same approach is used for computing the system of characteristic equations for $AB$ and $A^*$, and for implementing the steps 2 and 3 listed at the beginning of the Section 6. We leave these details out and refer the reader to the source code of our program.

The functions and type definitions we have presented are available as the file simple_proof_system.ml. They are a simplified version of corresponding
definitions and functions used in our real program which is available as the two files proof_system.ml and salomaa_certificate.ml. In particular, the assumptions we did on the list of transitions are impractical for a large alphabet. Also, the notion of proof environment must be extended with significantly more commands to be able to deal with systems of equations and indexing them efficiently. All in all, the proof system used for the entire decision procedure is about 300 lines of code, to be compared to the 200 lines of code for the proof system we described until Section 5. The program deciding the equivalence and computing the certificates is about 1,500 lines.

7 Validation and experimental results

We implemented two decision procedures producing certificates. One that follows closely Salomaa’s original proof of completeness, and a new one that follows the standard approach of converting regular expressions into NFAs and then convert the NFAs to DFAs. Both programs use the same method described in [Sal66] to decide the equivalence of two systems of characteristic equations associated to two DFAs.

Both programs are made of the superposition of two main components: the decision procedure on one side, and the production of certificates on the other side. Thus, to validate our entire program, we must ensure that the two following properties hold:

1. **Correctness of the decision procedure** – the program correctly decides whether two regular expressions are equivalent or not.
2. **Validity of certificates** – a certificate is a valid proof of the equivalence of two regular expressions.

We address the first property by testing the output of the decision procedure component against another algorithm that we trust to be correct. This trustworthy algorithm consists of the successive application of three algorithms from the folklore of Automata Theory: the compilation of regular expressions to $\epsilon$-NFAs, the determinisation of NFA to DFA using the subset construction, and Hopcroft-Karp’s algorithm [HK71] to decide the equivalence of two DFAs. We implemented each of the three algorithms naively, avoiding optimizations to maximize the level of correctness. They consist of a total of 200 lines of code. Assuming that the trustworthy algorithm is correct, in order to provide a high level of confidence that our decision procedure is correct, the benchmark compares the outputs of both the trustworthy algorithm and our program for a large number of regular expressions. The benchmark consists of regular expressions of various sizes, generated at random using a BST type of distribution [NPR10] over an alphabet of 2 letters $\Sigma = \{a, b\}$. For each size of regular expression from 5 to 30, we generated 50,000 pairs of regular expressions, and we verified that for each pair of regular expressions the program returns the same result as

\footnote{We also ran similar tests with success for regular expressions over an alphabet of size 4.}
the trustworthy algorithm. Despite the fact that it is quite unlikely that two random regular expressions are equivalent, for every size we found between 100 and 1000 pairs of equivalent regular expressions. A small sample of pairs of equivalent regular expressions found by our generator and proved to be equivalent is:

\[
\begin{align*}
((aa)a + (b + a))*b &= (b*((a*)* + a)*b \\
(a + (b + a))*(a(ab)) &= b*(((b + a)*((aa)b)) \\
(ab*)* + (a* + a) &= (a(((b + a) + a)*))** \\
((ab)((a + a)*b*)* + a)* &= ((ab* + a*) + ab*)(ab)* \\
(b((a + a)* + a*(bb)))* &= b((b + a)(b + (1 + a))*))** \\
(a + ab)(ba + (((ba + b)* + a)* &= a((b(a + b)* + (a + b*) ) + ((aa)a)a)* \\
(b+a)((b+a)+(aa)*)*(b*+((aa)**a)))* &= (a(bb)+((a+b)+b))(((a+b)*)**+a(ab)
\end{align*}
\]

To conclude on the experimental results for the first property, since our program and the trustworthy algorithm are very different in nature and they both agree on a large set of tests we have conducted, this provides a high confidence that our programs are correct.

For the second property, we verified that the certificates produced are valid. To check that a certificate is valid, we run it through our proof checker and make sure that it proves the expected equivalence. We used the same benchmark as for the first property, running the proof checker everytime a certificate is produced, all the certificates were correct.

Now, we propose to compare the performance of the two decision procedures producing certificates in terms of the length of certificate they produce. Figure 2 shows the length of certificates produced by the original algorithm by Salomaa’s and the revised version based on the standard algorithm. The figure on the left shows the maximum length of certificates, whereas the figure of the right shows the average length of certificates. For every size of regular expression between 5 and 30, the figures report the maximum and average length of certificate over several hundreds equivalent regular expressions. The figure showing the maximum length of certificate demonstrates a growth that is more than exponential, this is consistent with the worst case complexity of Salomaa’s algorithm, which is a tower of exponential $2^{2^{2^\ldots \cdot 2}}$ of height $n$, as explained in the original paper [Sal66]. In contrast the decision procedure based on the standard algorithm does not indicate worse than exponential complexity for the maximum length and even a slightly sub-exponential complexity for the average length.

The source of our programs is available [RR16]. It consists of several files: regexp.ml for regular expressions, simple_proof_system.ml is the proof system described in this article, salomaa_certificate.ml is the decision procedure producing certificates following Salomaa’s original method, nfa_det_certificate.ml is the decision procedure producing certificates following the standard approach, hk.ml is the naive implementation of Hopcroft-Karp algorithm, and test_original_vs_standard.ml is the source of the benchmark presented above.
8 Related works

The main contribution of this article is the program computing axiomatic certificates for the equivalence of regular expressions. Our approach is different from any other related work we have encountered.

A series of related works [CS11, Asp12, NT14, MPdS15] present various formalizations of the proof of correctness of decision procedures for the regular expression equivalence problem. These works focus mainly on the proof of the correctness of the decision procedure and leaves unexplored the production of certificates using a particular axiomatisation.

In Section 1, we already mentioned the difference between Kleene Algebras [Koz94] and Salomaa’s axiomatisation. One advantage of Kleene Algebras is to capture more models than the mere regular expressions. Nevertheless, what we have achieved with Salomaa’s axiomatisation – the computation of certificates expressed at the level of the axiomatisation – seems to be a challenge for Kleene Algebras. More specifically, a key feature of the proof of completeness of a Kleene Algebra $K$ is the use of a second Kleene Algebra for matrices $M(K)$. The proof of completeness for $K$ is mostly performed at the higher level of matrices $M(K)$ and it is unclear to us how to eliminate this extra layer in order to obtain certificates expressed directly at the level of the Kleene Algebra $K$.

In [BP10], Braibant and Pous describe a formalization in the Coq proof assistant of the completeness of Kleene Algebras following Kozen’s original work [Koz94]. In principle, since the development is done in Coq, it is possible to extract certificates of equivalence of regular expressions, but they would be written in Coq’s elaborated logic in relation to their formalization. Communicating such a proof to another proof assistant would require translating proofs from one proof assist-
tant to another, and is much more difficult than communicating our certificates
which are expressed in the simple logic of Salomaa’s axiomatisation.

An interesting work that focuses on the production of proofs/certificates is by
Worthington [Wor08]. It follows Kozen’s proof of completeness and introduces a
proof system, specifically tailored to deal with the Kleene algebra of matrices over
a Kleene algebra. It describes the design of a PSPACE transducer to compute the
certificates for their proof system. In principle, this transducer achieve similar
goals than our program computing certificates. Nevertheless, for similar reasons
we mentioned above, it is unclear how to eliminate the use of matrices that is
an integral part of their proof system.

Another line of related works is by Antimirov and Mosses [AM95], and more
recently by Almeida et al. [AMR09]. Their approach is to turn axiomatisations
into rewriting systems. To establish an equivalence between two regular expres-
sions, the rewriting system transforms regular expressions according to rewriting
rules until it proves the equivalence. A sequence of rewriting steps performed by
the rewriting system constitute a certificate of equivalence.

9 Conclusion and future work

The problem of equivalence of regular expressions and its axiomatisation has
interested the community since the early development of the field in the 1950’s.
The main property of an axiomatisation is its completeness. The completeness
guarantees that when two regular expressions are equivalent then there exists
a proof of this equivalence in the axiomatisation. Nevertheless, a proof of com-
pleteness is not exactly an algorithm computing axiomatic certificates. Ideally,
it would be possible to extract from a proof of completeness some algorithm
computing axiomatic certificates of regular expression equivalence. In this article,
we have described a simple proof system for Salomaa’s axiomatisation and
how to design of a program computing axiomatic certificates with Salomaa’s
axiomatisation. In particular we implemented one program that follows closely
Salomaa’s original proof of completeness [Sal66] and we also revisited this proof
of completeness to produce exponentially smaller certificates in the worst case,
following the standard technique based on NFA to DFA conversion. This demonstra-
tes that our approach is not limited to Salomaa’s original work but could be
applied to other related works.

We plan to investigate how to apply our methodology to efficient decision
procedures like the bisimulation upto congruence introduced by Bonchi and
Pous [BP13] or the antichain algorithm introduced by De Wulf et al. [DWDHR06].
Another interesting future work would be to apply our methodology to axiomati-
sations with many potential applications like Kleene Algebras [Koz94], Action
Algebras [Pra90], KAT [Koz97] or NetKAT [AFG+14].

Our program could be used by proof assistants as an external tool to solve
problems related to the regular expression equivalence problem. In this case, our
program would communicate its result to the proof assistant in the form of a
certificate of equivalence, independently checkable by the proof assistant.
Acknowledgments  We would like to thank Paritosh Pandya for all the enthusiastic discussions we had at TIFR about this work.

References


