A compact proof of decidability for regular expression equivalence

Andrea Asperti

Department of Computer Science, University of Bologna
asperti@cs.unibo.it

Abstract. The article describes a *compact* formalization of the relation between regular expressions and deterministic finite automata, and a formally verified, *efficient* algorithm for testing regular expression equivalence, both based on the notion of *pointed regular expression* [8].

1 Introduction

In this paper, we give a simple formalization of the construction of a deterministic finite automaton associated with a given regular expression, and of a bisimilarity algorithm to check regular expression equivalence. Our approach is based on the notion of *pointed regular expression* (pre), introduced in [8] (a similar notion has been independently presented in [14]). A pointed regular expression is just a regular expression internally labelled with some additional points. Intuitively, points mark the positions inside the regular expression which have been reached after reading some prefix of the input string, or better the positions where the processing of the remaining string has to be started. Each pointed expression for e represents a state of the *deterministic* automaton associated with e; since we obviously have only a finite number of possible labellings, the number of states of the automaton is finite.

Pointed regular expressions provide the tool for an *algebraic* revisitation of McNaughton and Yamada's algorithm for position automata [19], making the proof of its correctness, that is far from trivial (see e.g. [9, 11, 12]), particularly clear and simple. In particular, pointed expressions offer an appealing alternative to Brzozowski's derivatives (see e.g. [20] for a recent revisitation), avoiding their weakest point, namely the fact of being forced to quotient derivatives w.r.t. a suitable notion of equivalence in order to get a finite number of states (that is not essential for recognizing strings, but is crucial for comparing regular expressions).

All the proofs in this paper have been formalized in the Interactive Theorem Prover Matita [6].

2 Preliminaries

An *alphabet* is an arbitrary set of elements, equipped with a decidable equality:

```
record DeqSet : Type :=
{  carr :>Type;                  (* coercion *)
   eqb: carr → carr → bool;      (* notation:  a == b *)
   eqb_true: ∀x,y. (eqb x y = true) ↔ (x = y)
}.
```

A string (or word) over the alphabet S is just an element of *list S*. We need to deal with languages, that is, sets of strings. A traditional way to encode sets of elements in a given universe U in type theory is by means of predicates over U, namely elements of $U \to Prop$. A language over an alphabet S is hence an element of $list\ S \to Prop$.

Languages inherit all the basic operations for sets, namely union, intersection, complementation, substraction, and so on. In addition, we may define some new operations induced by string concatenation, and in particular the *concatenation* $A \cdot B$ of two languages A and B, the so called *Kleene's star* A^* of A and the *derivative* of a language A w.r.t. a given character a:

```
definition cat :=λS,A,B.λw:word S.
  ∃w1,w2.w1 @ w2 = w ∧ A w1 ∧ B w2.

definition star :=λS,A,λw:word S.
  ∃lw. flatten  S lw = w ∧ list_forall  S lw A.

definition deriv :=λS,A,a,w. A (a::w).
```

In the definition of star, $flatten$ and $list\_forall$ are standard functions over lists, respectively mapping $[l_1, \ldots, l_n]$ to $l_1@l_2 \ldots @l_n$ and $[w_1, w_2, \ldots, w_n]$ to $(A\ w_1) \wedge (A\ w_2) \cdots \wedge (A\ w_n)$.

Two languages are equal if they are equal as sets, namely if they contain the same words. This notion of equality, called `eqP` and denoted with the infix operator $\simeq$, is an extensional equality, different from the primitive intensional equality of Matita. In particular, we can *rewrite* with an equation $A \simeq B$ inside a context $C[A]$, only if the context is compatible with "$\simeq$".

```
definition eqP :=λA:Type.λP,Q:A → Prop.∀a:A.P a ↔Q a.
```

The main equations between languages that we shall need for the purposes of this paper (in addition to the set theoretic ones, and those expressing extensionality of operations) are listed below; the simple proofs are omitted.

```
lemma epsilon_cat_r: ∀S.∀A:word S → Prop. A ·{ϵ} ≃ A.
lemma epsilon_cat_l: ∀S.∀A:word S → Prop. {ϵ} ·A ≃ A.
lemma distr_cat_r: ∀S.∀A,B,C:word S → Prop. (A ∪B) ·C ≃  A · C ∪ B · C.
lemma deriv_union: ∀S,A,B,a. deriv (A ∪B) a ≃ (deriv A a) ∪ (deriv B a).
lemma deriv_cat: ∀S,A,B,a. ¬ A ϵ→ deriv (A·B) a ≃ (deriv A a) · B.
lemma star_fix_eps : ∀S.∀A:word S → Prop. A* ≃ (A − {ϵ}) ·A* ∪ {ϵ}.
```

3 Regular Expressions

The type *re* of regular expressions over an alphabet S is the smallest collection of objects generated by the following constructors:

```
inductive re (S: DeqSet) : Type :=
   z: re S                    (* empty *)
 | e: re S                    (* epsilon *)
 | s: S → re S                (* symbol *)
 | c: re S → re S → re S      (* concatenation *)
 | o: re S → re S → re S      (* plus *)
 | k: re S → re S.            (* kleene's star *)
```

In Matita, similarly to most interactive provers, we provide mechanisms to let the user define his own notation for syntactic constructs, and in the rest of the paper we shall use the traditional notation for regular expressions, namely $\emptyset, \epsilon, a, e_1 \cdot e_2, e_1 + e_2, e^*$.

The language *sem r* (notation: $[\![r]\!]$) associated with the regular expression r is defined by the following function:

```
let rec sem (S : DeqSet) (r : re S) on r : word S → Prop :=
  match r with
  [ z ⇒ ∅
  | e ⇒ {ϵ}
  | s x ⇒ {[x]}
  | c r1 r2 ⇒ ⟦r1⟧ · ⟦r2⟧
  | o r1 r2 ⇒ ⟦r1⟧ ∪ ⟦r2⟧
  | k r1 ⇒ ⟦r1⟧^* ].
```

4 Pointed regular expressions

A pointed item is a data type used to encode a *set of positions* inside a regular expression. The idea of formalizing pointers inside a data type by means of a labelled version of the data type itself is probably one of the first, major lessons learned in the formalization of the metatheory of programming languages (see e.g. [16] for a precursory application to residuals in lambda calculus). For our purposes, it is enough to mark positions preceding individual characters, so we shall have two kinds of characters $\bullet a$ (*pp a*) and a (*ps a*) according to the case a is pointed or not.

```
inductive pitem (S: DeqSet) : Type :=
   pz: pitem S
 | pe: pitem S
 | ps: S → pitem S
 | pp: S → pitem S
 | pc: pitem S → pitem S → pitem S
 | po: pitem S → pitem S → pitem S
 | pk: pitem S → pitem S.
```

A *pointed regular expression* (pre) is just a pointed item with an additional boolean, that must be understood as the possibility to have a trailing point *at the end* of the expression. As we shall see, pointed regular expressions can be understood as states of a DFA, and the boolean indicates if the state is final or not.

```
definition pre :=λS.pitem S × bool.
```

The *carrier* $|i|$ of an item i is the regular expression obtained from i by removing all the points. Similarly, the *carrier* of a pointed regular expression is the carrier of its item. The formal definition of this functions are straightforward, so we omit them. In the sequel, we shall use the same notation for functions defined over items or pres, leaving to the reader the simple disambiguation task (matita is also able to solve autonomously this kind of notational overloading).

The intuitive semantic of a point is to mark the position where we should start reading the regular expression. The language associated to a pre is the union of the languages associated with its points. Here is the straightforward definition (the question mark is an implicit parameter):

```
let rec semi (S : DeqSet) (i : pitem S) on i : word S → Prop :=
match r with
[ pz ⇒ ∅
| pe ⇒ ∅
| ps _ ⇒ ∅
| pp x ⇒ {[x]}
| pc i1 i2 ⇒ (semi ? i1) · ⟦|i2|⟧ ∪ (semi ? i2)
| po i1 r2 ⇒ (semi ? i1) ∪ (semi ? i2)
| pk i1 ⇒ (semi ? i1) · ⟦|i1|⟧* ].

definition semp :=λS : DeqSet.λp:pre S.
  if (snd p) then semi ? (fst p) ∪ {ϵ} else semi ? (fst p).
```

In the sequel, we shall often use the same notation for functions defined over re, items or pres, leaving to the reader the simple disambiguation task (matita is also able to solve autonomously this kind of notational overloading). In particular, we shall denote with $[\![e]\!]$ all semantic functions *sem*, *semi* and *semp*.

Example 1.

1. If e contains no point then $[\![e]\!] = \emptyset$
2. $[\![(a + \bullet bb)^*]\!] = [\![bb(a + bb)^*]\!]$

$\square$

Here are a few, simple, semantic properties of items

```
lemma not_epsilon_item : ∀S:DeqSet.∀i:pitem S. ¬ (⟦i⟧ ϵ).
lemma epsilon_pre : ∀S.∀e:pre S. (⟦i⟧ ϵ) ↔ (snd e = true).
lemma minus_eps_item: ∀S.∀i:pitem S. ⟦i⟧ ≃ ⟦i⟧−{ϵ}.
lemma minus_eps_pre: ∀S.∀e:pre S. ⟦fst e⟧ ≃ ⟦e⟧−{ϵ}.
```

The first property is proved by a simple induction on i; the other results are easy corollaries.

4.1 Intensional equality of pres

Items and pres are a very concrete datatype: they can be effectively compared, and enumerated. This is important, since pres *are* the states of our finite automata, and we shall need to compare states for bisimulation in Section 7.

In particular, we can define *beqitem* and *beqitem_true* enriching the set (*pitemS*) to a *DeqSet*.

```
definition DeqItem :=λS.
  mk_DeqSet (pitem S) (beqitem S) (beqitem_true S).
```

Matita's mechanism of *unification hints* [7] allows the type inference system to look at (*pitemS*) as the carrier of *DeqSet*, and at *beqitem* as if it was the equality function of *DeqSet*.

The product of two DeqSets is clearly still a DeqSet. Via unification hints, we may enrich a product type to the corresponding DeqSet; since moreover the type of booleans is a DeqSet too, this means that the type of pres *automatically inherits* the structure of a DeqSet (in Section 7, we shall deal with pairs of pres, and in this case too, without having anything to declare, the type will inherit the structure of a DeqSet).

Items and Pres can also be enumerated. In particular, it is easy to define a function *pre_enum* that takes in input a regular expression and gives back the list of all pres having e for carrier. Completeness of *pre_enum* is stated by the following lemma:

```
lemma pre_enum_complete : ∀S.∀e:pre S.
  memb ? e (pre_enum S (|fst e|)) = true.
```

5 Broadcasting points

Intuitively, a regular expression e must be understood as a pointed expression with a single point in front of it. Since however we only allow points before symbols, we must broadcast this initial point inside e traversing all nullable subexpressions, that essentially corresponds to the ϵ-closure operation on automata. We use the notation $\bullet(\cdot)$ to denote such an operation; its definition is the expected one: let us start discussing an example.

Example 2. Let us broadcast a point inside $(a+\epsilon)(b^*a+b)b$. We start working in parallel on the first occurrence of a (where the point stops), and on ϵ that gets traversed. We have hence reached the end of $a+\epsilon$ and we must pursue broadcasting inside $(b^*a+b)b$. Again, we work in parallel on the two additive subterms b^*a and b; the first point is allowed to both enter the star, and to traverse it, stopping in front of a; the second point just stops in front of b. No point reached that end of b^*a+b hence no further propagation is possible. In conclusion:

$$\bullet((a+\epsilon)(b^*a+b)b) = \langle(\bullet a+\epsilon)((\bullet b)^*\bullet a+\bullet b)b, \mathit{false}\rangle$$

□

Broadcasting a point inside an item generates a pre, since the point could possibly reach the end of the expression. Broadcasting inside a pair $i_1 + i_2$ amounts to broadcast in parallel inside i_1 and $i2$. If we define

$$\langle i_1, b_1' \rangle \oplus \langle i_2, b_2 \rangle = \langle i_1 + i_2, b_1 \vee b_2 \rangle$$

then, we just have $\bullet(i_1 + i_2) = \bullet(i_1) \oplus \bullet(i_2)$.

Concatenation is a bit more complex. In order to broadcast an item inside $i_1 \cdot i_2$ we should start broadcasting it inside i_1 and then proceed into i_2 if and only if a point reached the end of i_1.

This suggests to define $\bullet(i_1 \cdot i_2)$ as $\bullet(i_1) \triangleright i_2$, where $e \triangleright i$ is a general operation of concatenation between a pre and item (named *pre_concat_l*) defined by cases on the boolean in e

$$\langle i_1, \mathit{true} \rangle \triangleright i_2 = i_1 \triangleleft \bullet(i_2) \qquad \langle i_1, \mathit{false} \rangle \triangleright i_2 = \langle i_1 \cdot i_2, \mathit{false} \rangle$$

In turn, $\triangleleft$ (named *pre_concat_r*) says how to concatenate an item with a pre, that is however extremely simple:

$$i_1 \triangleleft \langle i_1, b \rangle = \langle i_1 \cdot i_2, b \rangle$$

The different kinds of concatenation between items and pres are summarized in Fig. 1, where we also depict the concatenation between two pres of Section 5.3.

	item	pre
item	$i_1 \cdot i_2$	$i_1 \triangleleft e_2$ $i_1 \triangleleft \langle i_1, b \rangle := \langle i_1 \cdot i_2, b \rangle$
pre	$e_1 \triangleright i_2$ $\langle i_1, \mathit{true} \rangle \triangleright i_2 := i_1 \triangleleft \bullet(i_2)$ $\langle i_1, \mathit{false} \rangle \triangleright i_2 := \langle i_1 \cdot i_2, \mathit{false} \rangle$	$e1 \odot e_2$ $e_1 \odot \langle i_2, b \rangle :=$ let $\langle i', b' \rangle = e_1 \triangleright i_2$ in $\langle i', b \vee b' \rangle$

Fig. 1. Concatenations between items and pres and respective equations

The definition of $\bullet(\cdot)$ (*eclose*) and $\triangleright$ (*pre_concat_l*) are mutually recursive. In this situation, a viable alternative that is usually simpler to reason about, is to abstract one of the two functions with respect to the other.

```
definition pre_concat_l :=λS.λbcast:∀S.pitem S → pre S.λe1:pre S.λi2:pitem S.
  let ⟨i1,b1⟩ := e1 in
  if b1 then i1 ◃ (bcast ? i2) else ⟨i1 · i2, false⟩.

let rec eclose (S: DeqSet) (i: pitem S) on i : pre S :=
 match i with
  [ pz ⇒ ⟨pz S, false⟩
  | pe ⇒ ⟨pe S, true⟩
  | ps x ⇒ ⟨ps S x, false⟩
  | pp x ⇒ ⟨pp S x, false⟩
  | po i1 i2 ⇒ •i1 ⊕ •i2
  | pc i1 i2 ⇒ •i1 ◃ i2
  | pk i ⇒ ⟨(fst (•i))^*,true⟩ ].
```

The definition of *eclose* can then be *lifted* from items to pres:

```
definition lift :=λS.λf:pitem S → pre S.λe:pre S.
  let ⟨i,b⟩ := e in ⟨fst (f i), snd (f i) ∨ b⟩.

definition preclose :=λS. lift S (eclose S).
```

By induction on the item i it is easy to prove the following result:

```
lemma erase_bullet : ∀S.∀i:pitem S. |fst (•i)| = |i|.
```

5.1 Semantics

We are now ready to state the main semantic properties of $\oplus, \triangleright, \triangleleft$ and $\bullet(-)$:

```
lemma sem_oplus: ∀S:DeqSet.∀e1,e2:pre S.
  ⟦e1 ⊕ e2⟧ ≃ ⟦e1⟧ ∪ ⟦e2⟧.

lemma sem_pre_concat_r : ∀S,i.∀e:pre S.
  ⟦i ▷ e⟧ ≃ ⟦i⟧ · ⟦|fst e|⟧ ∪ ⟦e⟧.

lemma sem_pre_concat_l : ∀S.∀e1:pre S.∀i2:pitem S.
  ⟦e1 ◁ i2⟧ ≃ ⟦e1⟧ · ⟦|i2|⟧ ∪ ⟦i2⟧.

theorem sem_bullet: ∀S:DeqSet. ∀i:pitem S.
  ⟦•i⟧ ≃ ⟦i⟧ ∪ ⟦|i|⟧.
```

The proofs of *sem_oplus* and *sem_pre_concat_r* are straightforward. For the others, we proceed as follow: we first prove the following auxiliary lemma, that assumes *sem_bullet*

```
lemma sem_pre_concat_l_aux : ∀S.∀e1:pre S.∀i2:pitem S.
   ⟦•i2⟧ ≃ ⟦i2⟧ ∪ ⟦|i2|⟧ →
     ⟦e1 ◁ i2⟧ ≃ ⟦e1⟧ · ⟦|i2|⟧ ∪ ⟦i2⟧.
```

Then, using the previous result, we prove *sem_bullet* by induction on i. Finally, *sem_pre_concat_l_aux* and *sem_bullet* give *sem_pre_concat_l*.

It is important to observe that all proofs have an algebraic flavor. Let us consider for instance the proof of *sem_pre_concat_l_aux*. Assuming $e_1 = \langle i_1, b_1\rangle$ we proceed by cases on b_1. If b_1 is false, the result is trivial; if b_1 is true, we have

$$\begin{array}{rll}
[\![\langle i_1, true\rangle \triangleleft i_2]\!] & \simeq [\![i_1]\!] \triangleright \bullet(i_2) & \text{by def. of } \triangleleft \\
& \simeq [\![i_1]\!] \cdot [\![|fst\ \bullet(i_2)|]\!] \cup [\![\bullet(i_2)]\!] & \text{by sem\_pre\_concat\_r} \\
& \simeq [\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] \cup [\![|i_2|]\!] & \text{by erase\_bullet and sem\_bullet} \\
& \simeq [\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![|i_2|]\!] \cup [\![i_2]\!] & \text{by assoc. and comm.} \\
& \simeq ([\![i_1]\!] \cup \{\epsilon\}) \cdot [\![|i_2|]\!] \cup [\![i_2]\!] & \text{by distr\_cat\_r} \\
& \simeq [\![\langle i_1, true\rangle]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] & \text{by the semantics of pre}
\end{array}$$

As another example, let us consider the proof of *sem_bullet*. The proof is by induction on i; let us consider the case of $i_1 \cdot i_2$. We have:

$$\begin{aligned}
[\![\bullet(i_1 \cdot i_2)]\!] &\simeq [\![\bullet(i_1)]\!] \triangleleft [\![i_2]\!] && \text{by definition of } \bullet(\cdot)\\
&\simeq [\![\bullet(i_1)]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by sem\_pre\_concat\_l}\\
&\simeq ([\![i_1]\!] \cup [\![|i_1|]\!]) \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by induction hypothesis}\\
&\simeq [\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![|i_1|]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!] && \text{by distr\_cat\_r}\\
&\simeq ([\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!]) \cup [\![|i_1 \cdot i_2|]\!] && \text{by assoc. and comm.}\\
&\simeq [\![(i_1 \cdot i_2)]\!] \cup [\![|i_1 \cdot i_2|]\!] && \text{by definition of } [\![\_]\!]
\end{aligned}$$

5.2 Initial state

As a corollary of theorem *sem_bullet*, given a regular expression e, we can easily find an item with the same semantics of e: it is enough to get an item (*blank* e) having e as carrier and no point, and then broadcast a point in it:

$$[\![\bullet(blank\ e)]\!] \simeq [\![(blank\ e)]\!] \cup [\![e]\!] \simeq [\![e]\!]$$

The definition of *blank* is straightforward; its main properties (both proved by an easy induction on e) are the following:

lemma forget_blank: ∀S.∀e:re S.|blank S e| = e.
lemma sem_blank: ∀S.∀e:re S. ⟦blank S e⟧ ≃ ∅.
theorem re_embedding: ∀S.∀e:re S. ⟦•(blank S e)⟧ ≃ ⟦e⟧.

5.3 Lifted operators

Plus and bullet have been already lifted from items to pres. We can now do a similar job for concatenation (⊙) and and Kleene's star (⊛).

definition lifted_cat :=λS:DeqSet.λe:pre S. lift S (pre_concat_l S eclose e).

definition lk :=λS:DeqSet.λe:pre S.
  let ⟨i1,b1⟩ := e **in** **if** b1 **then** ⟨(fst (eclose ? i1))*, true⟩ **else** ⟨i1*, false⟩.

We can easily prove the following properties:

lemma sem_odot: ∀S.∀e1,e2: pre S.
  ⟦e1 ⊙ e2⟧ ≃ ⟦e1⟧ · ⟦|fst e2|⟧ ∪ ⟦e2⟧.

theorem sem_ostar: ∀S.∀e:pre S.
  ⟦e^⊛⟧ ≃ ⟦e⟧ · ⟦|fst e|⟧*.

For example, let us look at the proof of the latter. Given $e = \langle i, b\rangle$ we proceed by cases on b. If b is false the result is trivial; if b is true we have:

$$\begin{aligned}
[\![\langle i, true\rangle^{\circledast}]\!] &\simeq [\![(fst\ \bullet(i))^*]\!] \cup \{\epsilon\} && \text{by definition of } \circledast\\
&\simeq [\![fst\ \bullet(i)]\!] \cdot [\![fst\ |\bullet(i)|]\!]^* \cup \{\epsilon\} && \text{by definition of } [\![\_]\!]\\
&\simeq [\![fst\ \bullet(i)]\!] \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by erase\_bullet}\\
&\simeq ([\![\bullet(i)]\!] - \{\epsilon\}) \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by minus\_eps\_pre}\\
&\simeq (([\![i]\!] \cup [\![|i|]\!]) - \{\epsilon\}) \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by sem\_bullet}\\
&\simeq (([\![i]\!] - \{\epsilon\}) \cup ([\![|i|]\!] - \{\epsilon\})) \cdot [\![|i|]\!]^* \cup \{\epsilon\} && \text{by distr\_minus}
\end{aligned}$$

$$
\begin{array}{ll}
\simeq(\llbracket i\rrbracket \cup (\llbracket |i| \rrbracket - \{\epsilon\}))\cdot \llbracket |i| \rrbracket^* \cup \{\epsilon\} & \text{by minus\_eps\_item}\\
\simeq\llbracket i\rrbracket \cdot \llbracket |i| \rrbracket^* \cup (\llbracket |i| \rrbracket - \{\epsilon\})\cdot \llbracket |i| \rrbracket^* \cup \{\epsilon\} & \text{by distr\_cat\_r}\\
\simeq\llbracket i\rrbracket \cdot \llbracket |i| \rrbracket^* \cup \llbracket |i| \rrbracket^* & \text{by star\_fix\_eps}\\
\simeq(\llbracket i\rrbracket \cup \{\epsilon\})\cdot \llbracket |i| \rrbracket^* & \text{by distr\_cat\_r}\\
\simeq\llbracket \langle i, \mathit{true}\rangle \rrbracket \cdot \llbracket |i| \rrbracket^* & \text{by definition of } \llbracket \_ \rrbracket
\end{array}
$$

6 Moves

We now define the move operation, that corresponds to the advancement of the state in response to the processing of an input character a. The intuition is clear: we have to look at points inside e preceding the given character a, let the point traverse the character, and broadcast it. All other points must be removed.

We can give a particularly elegant definition in terms of the lifted operators of the previous section:

```
let rec move (S: DeqSet) (x:S) (E: pitem S) on E : pre S :=
 match E with
  [ pz ⇒ ⟨pz S, false⟩
  | pe ⇒ ⟨pe S, false⟩
  | ps y ⇒ ⟨ps S y, false⟩
  | pp y ⇒ ⟨ps S, x == y⟩  (* the point is advanced if x==y, erased otherwise *)
  | po e1 e2 ⇒ (move ? x e1) ⊕ (move ? x e2)
  | pc e1 e2 ⇒ (move ? x e1) ⊙ (move ? x e2)
  | pk e ⇒ (move ? x e)^⊛ ].
```

Example 3. Let us consider the pre $(\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b)b$ and the two moves w.r.t. the characters a and b. For a, we have two possible positions (all other points gets erased); the innermost point stops in front of the final b, the other one broadcast inside $(b^*a + b)b$, so

$$\mathit{move}\ a\ ((\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b)b) = \langle (a+\epsilon)((\bullet b)^* \bullet a + \bullet b) \bullet b, \mathit{false}\rangle$$

For b, we have two positions too. The innermost point stops in front of the final b too, while the other point reaches the end of b^* and must go back through b^*a:

$$\mathit{move}\ b\ ((\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b) \bullet b) = \langle (a+\epsilon)((\bullet b)^* \bullet a + b) \bullet b, \mathit{false}\rangle$$

□

Obviously, a move does not change the carrier of the item, as one can easily prove by induction on the item

```
lemma same_carrier: ∀S:DeqSet.∀a:S.∀i:pitem S.
  | fst (move a i)| = |i |.
```

Here is our first, major result.

```
theorem move_ok: ∀S:DeqSet.∀a:S.∀i:pitem S.
   ⟦move a i⟧ ≃ deriv ⟦i⟧ a.
```

The proof is a simple induction on i. Let us see the case of concatentation:

$$\begin{array}{rll}
[\![move\ a\ (i_1 \cdot i_2)]\!] \simeq & [\![move\ a\ i_1 \odot move\ a\ i_2]\!] & \text{by def. of move} \\
\simeq & [\![move\ a\ i_1]\!] \cdot [\![|fst\ (move\ a\ i_2)|]\!] \cup [\![move\ a\ i_2]\!] & \text{by sem\_odot} \\
\simeq & [\![move\ a\ i_1]\!] \cdot [\![|i_2|]\!] \cup [\![move\ a\ i_2]\!] & \text{by same\_carrier} \\
\simeq & (deriv\ [\![i_1]\!]\ a) \cdot [\![|i_2|]\!] \cup (deriv\ [\![i_2]\!]\ a) & \text{by ind. hyp.} \\
\simeq & (deriv\ ([\![i_1]\!] \cdot [\![|i_2|]\!])\ a) \cup (deriv\ [\![i_2]\!]\ a) & \text{by deriv\_cat} \\
\simeq & deriv\ ([\![i_1]\!] \cdot [\![|i_2|]\!] \cup [\![i_2]\!])\ a & \text{by deriv\_union} \\
\simeq & deriv\ [\![i_1 \cdot i_2]\!]\ a & \text{by definition of } [\![\_]\!]
\end{array}$$

The move operation is generalized to strings in the obvious way:

```
let rec moves (S : DeqSet) w e on w : pre S :=
 match w with
  [ nil  ⇒ e
  | cons a tl   ⇒ moves S tl (move S a (fst e))].

lemma same_carrier_moves: ∀S:DeqSet.∀w.∀e:pre S.
  | fst  (moves ? w e)| = | fst  e |.

theorem decidable_sem: ∀S:DeqSet.∀w: word S. ∀e:pre S.
   (snd (moves ? w e) = true)  ↔ [[e]] w.
```

The proof of *decidable_sem* is by induction on w. The case $w = \epsilon$ is trivial; if $w = a :: w_1$ we have

$$\begin{array}{lll}
snd\ (moves\ (a :: w_1)\ e) = true & & \\
\leftrightarrow & snd\ (moves\ w_1\ (move\ a\ (fst\ e))) = true & \text{by def. of moves} \\
\leftrightarrow & [\![move\ a\ (fst\ e)]\!]\ w_1 & \text{by ind. hyp.} \\
\leftrightarrow & [\![e]\!]\ a :: w_1 & \text{by move\_ok}
\end{array}$$

It is now clear that we can build a DFA D_e for e by taking pre as states, and move as transition function; the initial state is $\bullet(e)$ and a state $\langle i, b\rangle$ is final if and only if $b = true$. The fact that states in D_e are finite is obvious: in fact, their cardinality is at most 2^{n+1} where n is the number of symbols in e. This is one of the advantages of pointed regular expressions w.r.t. derivatives, whose finite nature only holds after a suitable quotient.

Example 4. Figure 2 describes the DFA for the regular expression $(ac + bc)^*$.

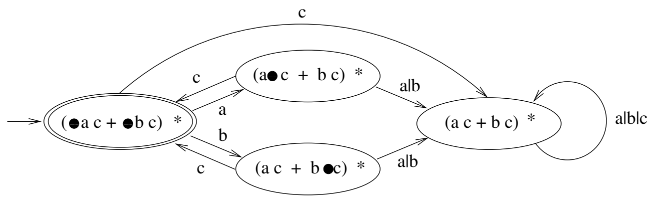

Fig. 2. DFA for $(ac + bc)^*$

The graphical description of the automaton is the traditional one, with nodes for states and labelled arcs for transitions. Unreachable states are not shown. Final states are emphasized by a double circle: since a state $\langle e, b\rangle$ is final if and only if b is true, we may just label nodes with the item.
The automaton is not minimal: it is easy to see that the two states corresponding to the pres $(a \bullet c + bc)^*$ and $(ac + b \bullet c)^*$ are equivalent (a way to prove it is to observe that they define the same language!). In fact, each state has a clear semantics given in terms of the associated pre e and not of the behaviour of the automaton. As a consequence, the construction of the automaton is not only *direct*, but also extremely *intuitive* and *locally verifiable*. □

Example 5. Starting from the regular expression $(a+\epsilon)(b^*a+b)b$, we obtain the automaton in Figure 3. Remarkably, this DFA is minimal, testifying the small

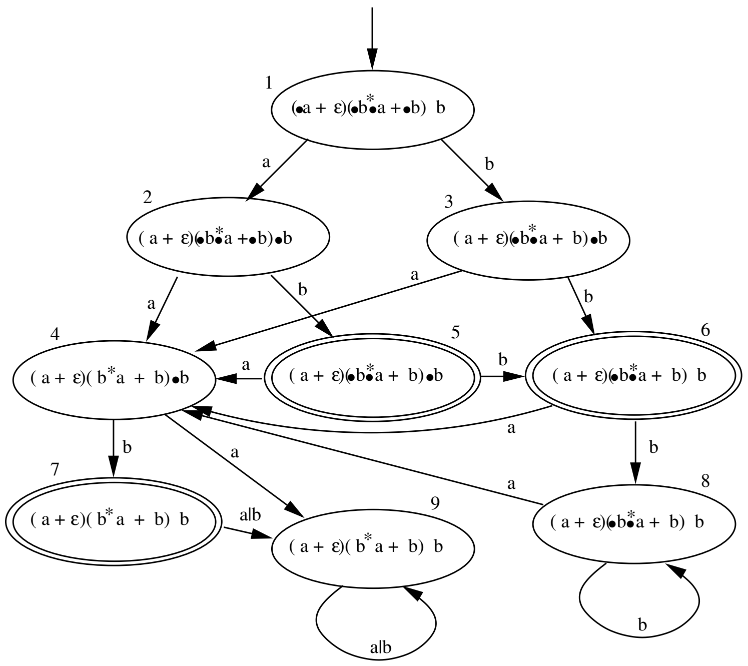

Fig. 3. DFA for $(a+\epsilon)(b^*a+b)b$

number of states produced by our technique (the pair of states $6 - 8$ and $7 - 9$ differ for the fact that 6 and 7 are final, while 8 and 9 are not). □

7 Equivalence

We say that two pres $\langle i_1, b_1\rangle$ and $\langle i_2, b_2\rangle$ are *cofinal* if and only if $b_1 = b_2$.
As a corollary of *decidable_sem*, we have that two expressions e_1 and e_2 are equivalent iff for any word w the states reachable through w are cofinal.

```
theorem equiv_sem: ∀S:DeqSet.∀e1,e2:pre S.
  ⟦e1⟧ ≃ ⟦e2⟧ ↔ ∀w.cofinal ⟨moves w e1,moves w e2⟩.
```

This does not directly imply decidability: we have no bound over the length of w; moreover, so far, we made no assumption over the cardinality of S. Instead of requiring S to be finite, we may restrict the analysis to characters occurring in the given pres. This means we can prove the following, stronger result:

```
lemma equiv_sem_occ: ∀S.∀e1,e2:pre S.(∀w.(sublist S w (occ S e1 e2))→
   cofinal ⟨moves w e1,moves w e2⟩) → ⟦e1⟧ ≃ ⟦e2⟧.
```

The proof essentially requires the notion of sink state and a few trivial properties:

```
definition sink_pre :=λS.λi.⟨blank S (| i |),  false ⟩.

lemma not_occur_to_sink: ∀S,a.∀i:pitem S. memb S a (occur S (|i|)) ≠ true →
  move a i  = sink_pre S i.

lemma moves_sink: ∀S,w,i. moves w (sink_pre S i) = sink_pre S i.
```

Let us say that a list of pairs of pres is a *bisimulation* if it is closed w.r.t. moves, and all its members are cofinal.

```
definition sons :=λS:DeqSet.λl:list S.λp:(pre S)×(pre S).
  map ?? (λa.⟨move a (fst (fst p)),move a (fst (snd p))⟩) l.

definition is_bisim :=λS:DeqSet.λl:list ?.λalpha:list S. ∀p:(pre S)×(pre S).
  memb ? p l = true → cofinal ? p ∧ (sublist ? (sons ? alpha p) l).
```

Using lemma *equiv_sem_occ* it is easy to prove

```
lemma bisim_to_sem: ∀S:DeqSet.∀l:list ?.∀e1,e2: pre S.
  is_bisim S l (occ S e1 e2) → memb ? ⟨e1,e2⟩ l = true → ⟦e1⟧ ≃ ⟦e2⟧.
```

As observed in [18] this is already an interesting result: checking if l is a bisimulation is decidable, hence we could generate l with some untrusted piece of code and then run a (boolean version of) *is_bisim* to check that it is actually a bisimulation. However, in order to prove that equivalence of regular expressions is *decidable* we must prove that we can always effectively build such a list (or find a counterexample). The idea is that the list we are interested in is just the set of all pair of pres *reachable* from the initial pair via some sequence of moves.

The algorithm for computing reachable nodes in a graph is a very traditional one. We split nodes in two disjoint lists: a list of *visited* nodes and a *frontier*, composed by all nodes connected to a node in visited but not visited already. At each step we select a node a from the frontier, compute its sons, add a to the set of visited nodes and the (not already visited) sons to the frontier.

Instead of fist computing reachable nodes and then performing the bisimilarity test we can directly integrate it in the algorithm: the set of visited nodes is closed by construction w.r.t. reachability, so we have just to check cofinality for any node we add to visited.

Here is the extremely simple algorithm

```
let rec bisim S l n (frontier,visited: list ?) on n :=
  match n with
  [ O ⇒ ⟨false,visited⟩ (* assert false *)
  | S m ⇒
    match frontier with
    [ nil ⇒ ⟨true,visited⟩
    | cons hd tl ⇒
      if beqb (snd (fst hd)) (snd (snd hd)) (* cofinality *) then
        bisim S l m (unique_append ? (filter ? (λx.notb (memb ? x (hd::visited)))
        (sons S l hd)) tl) (hd::visited)
      else ⟨false,visited⟩
    ]
  ].
```

The integer n is an upper bound to the number of recursive calls, equal to the dimension of the graph. It returns a pair composed by a boolean and the set of visited nodes; the boolean is true if and only if all visited nodes are cofinal.

The main test function is:

```
definition equiv :=λSig.λre1,re2:re Sig.
  let e1 :=•(blank ? re1) in
  let e2 :=•(blank ? re2) in
  let n :=S (length ? (space_enum Sig (|fst e1|) (|fst e2|))) in
  let sig :=(occ Sig e1 e2) in
  (bisim ? sig n [⟨e1,e2⟩] []).
```

We proved both correctness and completeness; in particular, we have

```
theorem euqiv_sem : ∀Sig.∀e1,e2:re Sig.
   fst (equiv ? e1 e2) = true ↔ ⟦e1⟧ ≃ ⟦e2⟧.
```

For correctness, we use the invariant that at each call of *bisim* the two lists *visited* and $frontier$ only contain nodes reachable from $\langle e_1, e_2\rangle$: hence it is absurd to suppose to meet a pair which is not cofinal. For completeness, we use the invariant that all the nodes in visited are cofinal, and the sons of *visited* are either in *visited* or in the $frontier$; since at the end $frontier$ is empty, *visited* is hence a bisimulation. All in all, correctness and completeness take little more than a few hundreds lines.

8 Discussion, related works, conclusions

Most of the formal proofs contained in this paper go back to 2009, preceding the technical report where we introduced the notion of pointed regular expression [8]; the long term idea, still in progress, was to use this material as a base for wrting an introductory tutorial to Matita. Since then, a small bunch of related works have appeared [2, 18, 13, 22], convincing us we could possibly add our two cents to this interesting, and apparently never exhausted topic.

Most of the above mentioned works are based on the notion of *derivative*, either in Brzozowski's acception [18, 13] or in Antimirov's one [2, 22]. This is not particularly surprising, since the algebraic nature of derivatives make them particularly appealing for a formal development. However, as remarked in [18], "in the large range of algorithms that turn regular expressions into automata, Brzozowski's procedure is on the elegant side, not the efficient one".

In order to get an efficient implementation, Braibant and Pous [10] resort to a careful implementation of finite state automata, encoding them as matrices over the given alphabet; automata are build using a variant of Thompson's technique [21] due to Ilie and Yu [17] (simpler to formalize then [21] but still complex).

Our approach based on pointed regular expressions provides a simple, algebraic revisitation of McNaughton and Yamada's algorithm [19] that, in contrast to Brzozowski's procedure, is traditionally reputed for its efficiency [1]; as a result, our approach is both *efficient* and *compact*.

Compactness, is maybe the most striking feature: from the definition of languages and regular expressions to the correctness proof of bisimilarity, our development takes less than 1200 lines. A *self contained* (not minimal) snapshot of the library can be found at `http:\\www.cs.unibo.it\~asperti\re.tar`, and it takes about 3400 lines. The development described in [18] has about the same size, but in this case the comparison is not fair, since they only check *correctness*, but do not address neither *termination* nor *completeness*. Especially, termination for Brzozowski's procedure is a delicate issue, taking quite an effort to [13].

The formalization in [13] is unexpectedly verbose: 7414 lines, *not including* relevant fragments of the standard library. This is particularly surprising since it has been written in ssreflect [15], that is reputed to be a compact dialect of Coq. We should observe that [13] contains two bisimilarity algorithms: one slow and naif (similar to that described in [18]) and one more complex, but more difficult to prove correct (taking, respectively, 1109 and 2576 lines). The point is that the efficiency of Brzozowski's procedure largely relies on the quotient made over derivatives: associative and commutative rewriting is enough for termination, but more complex rewritings are required to get a really performant implementation.

In spite of this huge effort, the actual performance of the bisimilarity test in [13] remains modest. Let us consider a couple of examples. The first one is an encoding of Bezout's identity discussed in [13]; exploiting the fact that set inclusion can be reduced to equality expressing $A \subseteq B$ as $A \cup B = B$, the arithmetical statement

$$\forall n \geq c. \exists x, y. n = xa + yb$$

can be expressed as the following regular expression problem

$$A(a, b, c) = (0^c)0^* + (0^a + 0^b)^* \simeq (0^a + 0^b)^*$$

The second problem, borrowed from [3], consists in proving the following equality:

$$B(n) = (\epsilon + a + aa + \cdots + a^{n-1})(a^n)^* \simeq a^*$$

In Figure 4 we compare our technique (pres) with that of [13]; execution times are expressed in seconds and have been computed on a machine with a Pentium M Processor 750 1.86GHz and 1GB of RAM.

problem	answer	*pres*	[13]
$A(3,5,8)$	yes	0.19	2.09
$A(4,5,11)$	no	0.18	5.26
$A(4,5,12)$	yes	0.24	5.26
$A(5,6,19)$	no	0.30	31.22
$A(5,6,20)$	yes	0.43	31.23
$A(5,7,23)$	no	0.38	70.09
$A(5,7,24)$	yes	0.57	70.19

problem	answer	*pres*	[13]
$B(6)$	yes	0.15	0.29
$B(8)$	yes	0.20	1.24
$B(10)$	yes	0.26	3.98
$B(12)$	yes	0.31	10.71
$B(14)$	yes	0.45	25.04
$B(16)$	yes	0.61	53.15
$B(18)$	yes	0.80	104.16

Fig. 4. Performance

The main achievement of our work, is however the very notion of *pointed regular expression*. The important facts are that

1. pointed expressions are in bijective correspondence with states of DFA
2. each pointed expression has a clear and intuitive semantics
3. the relation between a state and its sons is immediate and very natural

This allows a *direct*, *intuitive* and *locally verifiable* construction of the deterministic automaton for e, that is not only convenient for formalization, but also for didactic purposes. Since their discovery, we systematically used pointed expressions for teaching the argument to students and, according to our experience, they are *largely* superior to any other method we are aware of. In our opinion, pointed regular expressions are a nice example of the kind of results we may expect from the revisitation of methods and notions of computer science and mathematics induced by mechanical formalization, and which is probably the most ambitious and challenging objective of this discipline (see e.g. [4, 5]).

Acknowledgments We thank Claudio Sacerdoti Coen for contributing to the notion of pointed regular expression and Enrico Tassi for collaborating to an earlier, partial implementation in Matita. The work has been partially supported by the European Project CerCo.[1]

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson Education Inc., 2006.
2. José Bacelar Almeida, Nelma Moreira, David Pereira, and Simão Melo de Sousa. Partial derivative automata formalized in coq. In *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada*, volume 6482 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2010.

[1] CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

3. Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155:291–319, 1996.
4. Andrea Asperti and Cristian Armentano. A page in number theory. *Journal of Formalized Reasoning*, 1:1–23, 2008.
5. Andrea Asperti and Jeremy Avigad. Zen and the art of formalization. *Mathematical Structures in Computer Science*, 21(4):679–682, 2011.
6. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE-2011), Wroclaw, Poland*, volume 6803 of *LNCS*, 2011.
7. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 84–98. Springer-Verlag, 2009.
8. Andrea Asperti, Enrico Tassi, and Claudio Sacerdoti Coen. Regular expressions, au point. *eprint arXiv:1010.2604*, 2010.
9. Gérard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(3):117–126, 1986.
10. Thomas Braibant and Damien Pous. An efficient coq tactic for deciding kleene algebras. In *Proceedings of Interactive Theorem Proving, ITP 2010, Edinburgh, UK*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.
11. Anne Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.
12. Chia-Hsiang Chang and Robert Paige. From regular expressions to dfa's using compressed nfa's. In *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*, volume 644 of *Lecture Notes in Computer Science*, pages 90–110. Springer, 1992.
13. Thierry Coquand and Vincent Siles. A decision procedure for regular expression equivalence in type theory. In *Proceedings of Certified Programs and Proofs, CPP 2011, Kenting, Taiwan*, volume 7086 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2011.
14. Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions: functional pearl. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland.*, pages 357–368. ACM, 2010.
15. Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
16. Gérard P. Huet. Residual theory in lambda-calculus: A formal development. *J. Funct. Program.*, 4(3):371–394, 1994.
17. Lucian Ilie and Sheng Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.
18. Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, published on line, 2011.
19. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Ieee Transactions On Electronic Computers*, 9(1):39–47, 1960.
20. Scott Owens, John H. Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, 2009.
21. Ken Thompson. Regular expression search algorithm. *Communications of ACM*, 11:419–422, 1968.
22. Chunhan Wu, Xingyuan Zhang, and Christian Urban. A formalisation of the myhill-nerode theorem based on regular expressions (proof pearl). In *Proceedings of Interactive Theorem Proving ITP 2011, Berg en Dal, The Netherlands*, volume 6898 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2011.