Pointed Regular Expressions

Andrea Asperti[1], Claudio Sacerdoti Coen[1], and Enrico Tassi[2]

[1] Department of Computer Science, University of Bologna
asperti@cs.unibo.it sacerdot@cs.unibo.it
[2] INRIA-Micorsoft
tassi@cs.unibo.it

Abstract. We introduce a new technique for constructing a finite state deterministic automaton from a regular expression, based on the idea of marking a suitable set of positions *inside* the expression, intuitively representing the possible points reached after the processing of an initial prefix of the input string. *Pointed* regular expressions provide an algebraic counterpart to position automata, joining the elegance and the symbolic appealingness of Brzozowski's derivatives, with the effectiveness of McNaughton and Yamada's labelling technique, and essentially combining the best of the two approaches.

1 Introduction

There is hardly a subject in Theoretical Computer Science that, in view of its relevance and elegance, has been so thoroughly investigated as the notion of *regular expression* and its relation with *finite state automata* (see e.g. [18, 9] for some recent surveys). All the studies in this area have been traditionally inspired by two precursory, basilar works: Brzozowski's theory of *derivatives* [6], and McNaughton and Yamada's algorithm [15] (attributed to Glushkov [11] in the east). The main advantages of derivatives are that they are syntactically appealing, easy to grasp and to prove correct (see [17] for a recent revisitation). On the other side, McNaughton and Yamada's approach results in a particularly efficient algorithm, still used by most pattern matchers like the popular grep and egrep utilities. The relation between the two approaches has been deeply investigated too, starting from the seminal work by Berry and Sethi [4] where it is shown how to refine Brzozowski's method to get to the efficient algorithm (Berry and Sethi' algorithm has been further improved by later authors [5, 7]).

We introduce in this paper an algebraic variation of position automata based on *pointed* regular expression. 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 allow the *direct* construction of the DFA [14] associated with a regular expression, in a way that is simple, intuitive, and

efficient (the task is traditionally considered as *very involved* in the literature: see e.g [18], pag.71). At the same time, their algebraic nature allows a clean and comparison with derivatives (see the extended version of this paper [3]).

In the imposing bibliography on regular expressions - as far as we could discover - the only author mentioning a notion close to ours is Watson [19, 20]. However, he only deals with single points, while the most interesting properties of *pre* derive by their implicit additive nature (such as the possibility to compute the *move* operation by a single pass on the marked expression: see definition 11).

2 Regular expressions

Definition 1. *A regular expression over the alphabet Σ is an expression e generated by the following grammar, where $a \in \Sigma$:*

$$E ::= \emptyset | \epsilon | a | E + E | EE | E^*$$

Definition 2. *The language $L(e)$ associated with the regular expression e is defined by the following rules:*

$$\begin{array}{ll} L(\emptyset) = \emptyset & L(\epsilon) = \{\epsilon\} \\ L(a) = \{a\} & L(e_1 + e_2) = L(e_1) \cup L(e_2) \\ L(e_1 e_2) = L(e_1) \cdot L(e_2) & L(e^*) = L(e)^* \end{array}$$

where ϵ is the empty string, $L_1 \cdot L_2 = \{\ l_1 l_2 \mid l_1 \in L_1,\ l_2 \in L_2\}$ is the concatenation of L_1 and L_2 and L^* is the so called Kleene's closure of L: $L^* = \bigcup_{i=0}^{\infty} L^i$, with $L^0 = \epsilon$ and $L^{i+1} = L \cdot L^i$.

Definition 3 (nullable). *A regular expression e is* nullable if $\epsilon \in L(e)$.

The fact of being nullable is decidable; it is easy to prove that the characteristic function $\nu(e)$ can be computed by the following rules:

$$\begin{array}{ll} \nu(\emptyset) = \mathit{false} & \nu(\epsilon) = \mathit{true} \\ \nu(a) = \mathit{false} & \nu(e_1 + e_2) = \nu(e_1) \vee \nu(e_2) \\ \nu(e_1 e_2) = \nu(e_1) \wedge \nu(e_2) & \nu(e^*) = \mathit{true} \end{array}$$

Definition 4. *A deterministic finite automaton (DFA) is a quintuple (Q, Σ, q_0, t, F) where*

* *Q is a finite set of states;*
* *Σ is the input alphabet;*
* *$q_0 \in Q$ is the initial state;*
* *$t : Q \times \Sigma \to Q$ is the state transition function;*
* *$F \subseteq Q$ is the set of final states.*

The transition function t is extended to strings in the following way:

$$t^*(q, w) = \begin{cases} t(q, \epsilon) = q \\ t(q, aw') = t^*(t(q, a), w') \end{cases}$$

Definition 5. *Let $A = (Q, \Sigma, q_0, t, F)$ be a DFA; the language recognized A is defined as follows:*

$$L(A) = \{w | t^*(q_0, w) \in F\}$$

3 Pointed regular expressions

Definition 6.

1. *A* pointed item *over the alphabet Σ is an expression e generated by following grammar, where $a \in \Sigma$:*

$$E ::= \emptyset | \epsilon | a | \bullet a | E + E | EE | E^*$$

2. *A* pointed regular expression *(pre) is a pair $\langle e, b \rangle$ where b is a boolean and e is a pointed item.*

The term $\bullet a$ is used to point to a position inside the regular expression, preceding the given occurrence of a. In a pointed regular expression, the boolean must be intuitively understood as the possibility to have a trailing point at the end of the expression.

Definition 7. *The* carrier $|e|$ *of an item e is the regular expression obtained from e by removing all the points. Similarly, the* carrier *of a pointed regular expression is the carrier of its item.*

In the sequel, we shall often use the same notation for functions defined over items or pres, leaving to the reader the simple disambiguation task. Moreover, we use the notation $\epsilon(b)$, where b is a boolean, with the following meaning:

$$\epsilon(true) = \{\epsilon\} \qquad \epsilon(false) = \emptyset$$

Definition 8.

1. *The language $L_p(e)$ associated with the item e is defined as follows:*

$$\begin{array}{ll} L_p(\emptyset) = \emptyset & L_p(\epsilon) = \emptyset \\ L_p(a) = \emptyset & L_p(\bullet a) = \{a\} \\ L_p(e_1 + e_2) = L_p(e_1) \cup L_p(e_2) & L_p(e_1 e_2) = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2) \\ L_p(e^*) = L_p(e) \cdot L(|e|^*) & \end{array}$$

2. *For a pointed regular expression $\langle e, b \rangle$ we define*

$$L_p(\langle e, b \rangle) = L_p(e) \cup \epsilon(b)$$

Example 1.

1. If e contains no point (i.e. $e = |e|$) then $L_p(e) = \emptyset$
2. $L_p((a + \bullet b)^*) = L(b(a + b)^*)$

Let us observe that, as shown by point 2. above (replacing b with a more complex expression), pointed regular expressions can provide a more compact syntax for denoting languages than traditional regular expressions. This may have important applications to the investigation of the descriptional complexity (succinctness) of regular languages (see e.g. [10, 12, 13]).

Lemma 1. *If e is a pointed item then $\epsilon \notin L_p(e)$. Hence, $\epsilon \in L_p(\langle e, b\rangle)$ if and only if $b = true$.*

Proof. A trivial structural induction on e.

3.1 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 over symbols, we must broadcast this initial point inside the expression, that essentially corresponds to the ϵ-closure operation on automata. We use the notation $\bullet(\cdot)$ to denote such an operation.

The broadcasting operator is also required to lift the item constructors (choice, concatenation and Kleene's star) from items to pres: for example, to concatenate a pre $\langle e_1, true\rangle$ with another pre $\langle e_2, b_2\rangle$, we must first broadcast the trailing point of the first expression inside e_2 and then pre-pend e_1; similarly for the star operation. We could define first the broadcasting function $\bullet(\cdot)$ and then the lifted constructors; however, both the definition and the theory of the broadcasting function are simplified by making it co-recursive with the lifted constructors.

Definition 9.

1. *The function $\bullet(\cdot)$ from pointed item to pres is defined as follows:*

$$\begin{array}{ll}
\bullet(\emptyset) = \langle \emptyset, false\rangle & \bullet(\epsilon) = \langle \epsilon, true\rangle \\
\bullet(a) = \langle \bullet a, false\rangle & \bullet(\bullet a) = \langle \bullet a, false\rangle \\
\bullet(e_1 + e_2) = \bullet(e_1) \oplus \bullet(e_2) & \bullet(e_1 e_2) = \bullet(e_1) \odot \langle e_2, false\rangle \\
\bullet(e^*) = \langle e'^*, true\rangle \ \textit{where} \ \bullet(e) = \langle e', b'\rangle &
\end{array}$$

2. *The lifted constructors are defined as follows*

$$\langle e'_1, b'_1\rangle \oplus \langle e'_2, b'_2\rangle = \langle e_1 + e_2, b'_1 \vee b'_2\rangle$$

$$\langle e'_1, b'_1\rangle \odot \langle e'_2, b'_2\rangle = \begin{cases} \langle e'_1 e'_2, b'_2\rangle & \textit{when } b'_1 = false \\ \langle e'_1 e''_2, b'_2 \vee b''_2\rangle & \textit{when } b'_1 = true \textit{ and } \bullet(e'_2) = \langle e''_2, b''_2\rangle \end{cases}$$

$$\langle e', b'\rangle^\star = \begin{cases} \langle e'^*, false\rangle & \textit{when } b' = false \\ \langle e''^*, true\rangle & \textit{when } b' = true \textit{ and } \bullet(e') = \langle e'', b''\rangle \end{cases}$$

The apparent complexity of the previous definition should not hide the extreme simplicity of the broadcasting operation: on a sum we proceed in parallel; on a concatenation $e_1 e_2$, we first work on e_1 and in case we reach its end we pursue broadcasting inside e_2; in case of e^* we broadcast the point inside e recalling that we shall eventually have a trailing point.

Example 2. Suppose to 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:

$$\bullet((a+\epsilon)(b^*a+b)b) = (\bullet a+\epsilon)((\bullet b)^* \bullet a+\bullet b)b$$

Definition 10. *Broadcasting is extended to pres in the obvious way:*

$$\bullet(\langle e, b\rangle) = \langle e', b \vee b'\rangle \text{ where } \bullet(e) = \langle e', b'\rangle$$

As we shall prove in Corollary 2, broadcasting an initial point may reach the end of an expression e if and only if e is nullable. The following theorem characterizes the broadcasting function and also shows that the semantics of the lifted constructors on pres is coherent with the corresponding constructors on items.

Theorem 1.

1. $L_p(\bullet e) = L_p(e) \cup L(|e|)$
2. $L_p(e_1 \oplus e_2) = L_p(e_1) \cup L_p(e_2)$
3. $L_p(e_1 \odot e_2) = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2)$
4. $L_p(e^\star) = L_p(e) \cdot L(|e|)^*$

We do first the proof of 2., followed by the simultaneous proof of 1. and 3., and we conclude with the proof of 4.

Proof (of 2.). We need to prove $L_p(e_1 \oplus e_2) = L_p(e_1) \cup L_p(e_2)$.

$$\begin{aligned} L_p(\langle e_1', b_1'\rangle \oplus \langle e_2', b_2'\rangle) &= L_p(\langle e_1' + e_2', b_1' \vee b_2'\rangle) = L_p(e_1' + e_2') \cup \epsilon(b_1') \cup \epsilon(b_2') \\ &= L_p(e_1') \cup \epsilon(b_1') \cup L_p(e_2') \cup \epsilon(b_2') = L_p(e_1) \cup L_p(e_2) \end{aligned}$$

Proof (of 1. and 3.). We prove 1. ($L_p(\bullet e) = L_p(e) \cup L(|e|)$) by induction on the structure of e, assuming that 3. holds on terms structurally smaller than e.

* $L_p(\bullet(\emptyset)) = L_p(\langle \emptyset, \mathit{false}\rangle) = \emptyset = L_p(\emptyset) \cup L(|\emptyset|)$.
* $L_p(\bullet(\epsilon)) = L_p(\langle \epsilon, \mathit{true}\rangle) = \{\epsilon\} = L_p(\epsilon) \cup L_p(|\epsilon|)$.
* $L_p(\bullet(a)) = L_p(\langle a, \mathit{false}\rangle) = \{a\} = L_p(a) \cup L(|a|)$.
* $L_p(\bullet(\bullet a)) = L_p(\langle \bullet a, \mathit{false}\rangle) = \{a\} = L_p(\bullet a) \cup L(|\bullet a|)$.
* Let $e = e_1 + e_2$. By induction we know that $L_p(\bullet(e_i)) = L_p(e_i) \cup L(|e_i|)$ Thus, by 2., we have

$$\begin{aligned} L_p(\bullet(e_1+e_2)) &= L_p(\bullet(e_1) \oplus \bullet(e_2)) = L_p(\bullet(e_1)) \cup L_p(\bullet(e_2)) \\ &= L_p(e_1) \cup L(|e_1|) \cup L_p(e_2) \cup L(|e_2|) = L_p(e_1+e_2) \cup L(|e_1+e_2|) \end{aligned}$$

* Let $e = e_1 e_2$. By induction we know that $L_p(\bullet(e_i)) = L_p(e_i) \cup L(|e_i|)$. Thus, by 3. over the structurally smaller terms e_1 and e_2

$$\begin{aligned} L_p(\bullet(e_1e_2)) &= L_p(\bullet(e_1) \odot \langle e_2, \mathit{false}\rangle) \\ &= L_p(\bullet(e_1)) \cdot L(|e_2|) \cup L_p(e_2) = (L_p(e_1) \cup L(|e_1|)) \cdot L(|e_2|) \cup L_p(e_2) \\ &= L_p(e_1) \cdot L(|e_2|) \cup L(|e_1|) \cdot L(|e_2|) \cup L_p(e_2) = L_p(e_1e_2) \cup L(|e_1e_2|) \end{aligned}$$

$*$ Let $e = e_1^*$. By induction we know that $L_p(\bullet(e_1)) = L_p(e_1') \cup \epsilon(b_1') = L_p(e_1) \cup L(|e_1|)$ and in particular, since by Lemma 1 $\epsilon \notin L_p(e_1)$,

$$L_p(e_1') = L_p(e_1) \cup (L(|e_1|) \setminus \epsilon(b_1'))$$

Then,

$$\begin{aligned} L_p(\bullet(e_1^*)) &= L_p(\langle e_1'^*, true\rangle) = L_p(e_1'^*) \cup \epsilon \\ &= L_p(e_1')L(|e_1^*|) \cup \epsilon = (L_p(e_1) \cup (L(|e_1|) \setminus \epsilon(b_1')))L(|e_1^*|) \cup \epsilon \\ &= L_p(e_1)L(|e_1^*|) \cup (L(|e_1|) \setminus \epsilon(b_1'))L(|e_1^*|) \cup \epsilon = L_p(e_1)L(|e_1^*|) \cup L(|e_1^*|) \\ &= L_p(e_1^*) \cup L(|e_1^*|) \end{aligned}$$

Having proved 1. for e assuming that 3. holds on terms structurally smaller than e, we now assume that 1. holds for e_1 and e_2 in order to prove 3, i.e. $L_p(e_1 \odot e_2) = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2)$. We distinguish the two cases of the definition of $\odot$:

$$\begin{aligned} L_p(\langle e_1', false\rangle \odot \langle e_2', b_2'\rangle) &= L_p(\langle e_1'e_2', b_2'\rangle) = L_p(e_1'e_2') \cup \epsilon(b_2') \\ &= L_p(e_1') \cdot L(|e_2'|) \cup L_p(e_2') \cup \epsilon(b_2') = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2) \end{aligned}$$

$$\begin{aligned} L_p(\langle e_1', true\rangle \odot \langle e_2', b_2'\rangle) &= L_p(\langle e_1'e_2'', b_2' \vee b_2''\rangle) = L_p(e_1'e_2'') \cup \epsilon(b_2') \cup \epsilon(b_2'') \\ &= L_p(e_1') \cdot L(|e_2''|) \cup L_p(e_2'') \cup \epsilon(b_2') \cup \epsilon(b_2'') \\ &= L_p(e_1') \cdot L(|e_2''|) \cup L_p(e_2') \cup L(|e_2'|) \cup \epsilon(b_2') \\ &= (L_p(e_1') \cup \epsilon(true)) \cdot L(|e_2|) \cup L_p(e_2') \cup \epsilon(b_2') = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2) \end{aligned}$$

Proof (of 4.). We need to prove $L_p(e^\star) = L_p(e) \cdot L(|e|)^*$. We distinguish the two cases of the definition of $\cdot^\star$:

$$\begin{aligned} L_p(\langle e', false\rangle^\star) &= L_p(\langle e'^*, false\rangle) = L_p(e'^*) \\ &= L_p(e') \cdot L(|e'|)^* = (L_p(e') \cup \epsilon(false)) \cdot L(|e'|)^* = L_p(e) \cdot L(|e|)^* \end{aligned}$$

$$\begin{aligned} L_p(\langle e', true\rangle^\star) &= L_p(\langle e''^*, true\rangle) \cup \epsilon = L_p(e''^*) \cup \epsilon = L_p(e'') \cdot L(|e''|)^* \cup \epsilon \\ &= (L_p(e') \cup L(|e'|)) \cdot L(|e''|)^* \cup \epsilon = L_p(e') \cdot L(|e''|) \cup L(|e'|) \cdot L(|e''|)^* \cup \epsilon \\ &= L_p(e') \cdot L(|e''|) \cup L(|e'|)^* = (L_p(e') \cup \epsilon(true)) \cdot L(|e''|) = L_p(e) \cdot L(|e|)^* \end{aligned}$$

Corollary 1. *For any regular expression e, $L(e) = L_p(\bullet e)$.*

Another important corollary is that an initial point reaches the end of a (pointed) expression e if and only if e is able to generate the empty string.

Corollary 2. *$\bullet e = \langle e', true\rangle$ if and only if $\epsilon \in L(|e|)$.*

Proof. By theorem 1 we know that $L_p(\bullet e) = L_p(e) \cup L(|e|)$. So, if $\epsilon \in L_p(\bullet e)$, since by Lemma 1 $\epsilon \notin L_p(e)$, it must be $\epsilon \in L(|e|)$. Conversely, if $\epsilon \in L(|e|)$ then $\epsilon \in L_p(\bullet e)$; if $\bullet e = \langle e', b\rangle$, this is possible only provided $b = true$.

We conclude this section mentioning a few additional properties of the broadcasting function whose simple proofs can be found in the extended version [3].

Theorem 2.

1. $\bullet(e_1 \oplus e_2) = \bullet(e_1) \oplus \bullet(e_2)$
2. $\bullet(e_1 \odot e_2) = \bullet(e_1) \odot e_2$
3. $\bullet(\bullet(e)) = \bullet(e)$

3.2 The move operation

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.

Definition 11.

1. *The function $move(e, a)$ taking in input a pointed item e, a character $a \in \Sigma$ and giving back a pointed regular expression is defined as follow, by induction on the structure of e:*

$$\begin{array}{ll} move(\emptyset, a) = \langle \emptyset, false \rangle & move(\epsilon, a) = \langle \epsilon, false \rangle \\ move(b, a) = \langle b, false \rangle & move(\bullet a, a) = \langle a, true \rangle \\ move(\bullet b, a) = \langle b, false \rangle & move(e_1 + e_2, a) = move(e_1, a) \oplus move(e_2, a) \\ move(e^*, a) = move(e, a)^\star & move(e_1 e_2, a) = move(e_1, a) \odot move(e_2, a) \end{array}$$

2. *The move function is extended to pres by just ignoring the trailing point: $move(\langle e, b \rangle, a) = move(e, a)$*

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

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

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 $b^* a$:

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

Theorem 3. *For any pointed regular expression e and string w,*

$$w \in L_p(move(e, a)) \Leftrightarrow aw \in L_p(e)$$

Proof. The proof is by induction on the structure of e.

- if e is atomic, and e is not a pointed symbol, then both $L_p(move(e, a))$ and $L_p(e)$ are empty, and hence both sides are false for any w;
- if $e = \bullet a$ then $L_p(move(\bullet a, a)) = L_p(\langle a, true \rangle) = \{\epsilon\}$ and $L_p(\bullet a) = \{a\}$;
- if $e = \bullet b$ with $b \neq a$ then $L_p(move(\bullet b, a)) = L_p(\langle b, false \rangle) = \emptyset$ and $L_p(\bullet b) = \{b\}$; hence for any string w, both sides are false;
- if $e = e_1 + e_2$ by induction hypothesis $w \in L_p(move(e_i, a)) \Leftrightarrow aw \in L_p(e_i)$, hence,

$$\begin{array}{l} w \in L_p(move(e_1 + e_2, a)) \Leftrightarrow \\ \quad \Leftrightarrow w \in L_p(move(e_1, a) \oplus move(e_2, a)) \\ \quad \Leftrightarrow w \in L_p(move(e_1, a)) \cup L_p(move(e_2, a)) \\ \quad \Leftrightarrow (w \in L_p(move(e_1, a))) \vee (w \in L_p(move(e_2, a))) \\ \quad \Leftrightarrow (aw \in L_p(e_1)) \vee (aw \in L_p(e_2)) \\ \quad \Leftrightarrow aw \in L_p(e_1) \cup L_p(e_2) \\ \quad \Leftrightarrow aw \in L_p(e_1 + e_2) \end{array}$$

* suppose $e = e_1 e_2$, by induction hypothesis $w \in L_p(move(e_i, a)) \Leftrightarrow aw \in L_p(e_i)$, hence,

$$\begin{aligned}
&w \in L_p(move(e_1 e_2, a)) \Leftrightarrow \\
&\quad \Leftrightarrow w \in L_p(move(e_1, a) \odot move(e_2, a)) \\
&\quad \Leftrightarrow w \in L_p(move(e_1, a)) \cdot L|e_2| \cup L_p(move(e_2, a)) \\
&\quad \Leftrightarrow w \in L_p(move(e_1, a)) \cdot L|e_2| \vee w \in L_p(move(e_2, a)) \\
&\quad \Leftrightarrow (\exists w_1, w_2, w = w_1 w_2 \wedge w_1 \in L_p(move(e_1, a)) \\
&\qquad \wedge w_2 \in L(|e_2|)) \vee w \in L_p(move(e_2, a)) \\
&\quad \Leftrightarrow (\exists w_1, w_2, w = w_1 w_2 \wedge a w_1 \in L_p(e) \\
&\qquad \wedge w_2 \in L(|e_2|)) \vee aw \in L_p(e_2) \\
&\quad \Leftrightarrow (aw \in L_p(e_1) \cdot L|e_2|) \vee (aw \in L_p(e_2)) \\
&\quad \Leftrightarrow aw \in L_p(e_1) \cdot L|e_2| \cup \in L_p(e_2) \\
&\quad \Leftrightarrow aw \in L_p(e_1 e_2)
\end{aligned}$$

* suppose $e = e_1^*$, by induction hypothesis $w \in L_p(move(e_1, a)) \Leftrightarrow aw \in L_p(e_1)$, hence,

$$\begin{aligned}
&w \in L_p(move(e_1^*, a)) \Leftrightarrow \\
&\quad \Leftrightarrow w \in L_p(move(e_1, a))^\star \\
&\quad \Leftrightarrow w \in L_p(move(e_1, a)) \cdot L(|move(e_1, a)|)^* \\
&\quad \Leftrightarrow \exists w_1, w_2, w = w_1 w_2 \wedge w_1 \in L_p(move(e_1, a)) \\
&\qquad \wedge w_2 \in L(|e_1|)^* \\
&\quad \Leftrightarrow \exists w_1, w_2, w = w_1 w_2 \wedge a w_1 \in L_p(e_1) \wedge w_2 \in L(|e_1|)^* \\
&\quad \Leftrightarrow aw \in L_p(e_1) \cdot L(|e_1|)^* \\
&\quad \Leftrightarrow aw \in L_p(e_1^*)
\end{aligned}$$

We extend the move operations to strings in the usual way:

Definition 12.

$$move^*(e, \epsilon) = e \qquad move^*(e, aw) = move^*(move(e, a), w)$$

Theorem 4. *For any pointed regular expression e and all strings α, β,*

$$\beta \in L_p(move^*(e, \alpha)) \Leftrightarrow \alpha\beta \in L_p(e)$$

Proof. A trivial induction on the length of α, using theorem 3.

Corollary 3. *For any pointed regular expression e and any string α,*

$$\alpha \in L_p(e) \Leftrightarrow \exists e', L_p(move^*(e, \alpha)) = \langle e', true \rangle$$

Proof. By Theorems 4 and Lemma 1.

3.3 From regular expressions to DFAs

Definition 13. *To any regular expression e we may associate a DFA $D_e = (Q, \Sigma, q_0, t, F)$ defined in the following way:*

* Q *is the set of all possible pointed expressions having* e *as carrier;*
* Σ *is the alphabet of the regular expression*
* q_0 *is* $\bullet e$*;*
* t *is the move operation of definition 11;*
* F *is the subset of pointed expressions* $\langle e, b\rangle$ *with* $b = true$*.*

Theorem 5. $L(D_e) = L(e)$

Proof. By definition,

$$w \in L(D_e) \leftrightarrow move^*(\bullet(e), w) = \langle e', true\rangle$$

for some e'. By the previous theorem, this is possible if an only if $w \in L_p(\bullet(e))$, and by corollary 1, $L_p(\bullet(e)) = L(e)$.

Remark 1. The fact that the set Q of states of D_e is finite is obvious: its 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, and is not so trivial to prove (see [6]).

The automaton D_e just defined may have many inaccessible states. We can provide another algorithmic and direct construction that yields the same automaton restricted to the accessible states only.

Definition 14. *Let* e *be a regular expression and let* q_0 *be* $\bullet e$*. Let also*

$$\begin{aligned} Q_0 &:= \{q_0\} \\ Q_{n+1} &:= Q_n \cup \{e' | e' \notin Q_n \wedge \exists a. \exists e \in Q_n. move(e, a) = e'\} \end{aligned}$$

Since every Q_n *is a subset of the finite set of pointed regular expressions, there is an* m *such that* $Q_{m+1} = Q_m$*. We associate to* e *the DFA* $D_e = (Q_m, \Sigma, q_0, F, t)$ *where* F *and* t *are defined as for the previous construction.*

Example 4. In Figure 1 we describe the DFA associated with the regular expression $(ac + bc)^*$. The graphical description of the automaton is the traditional

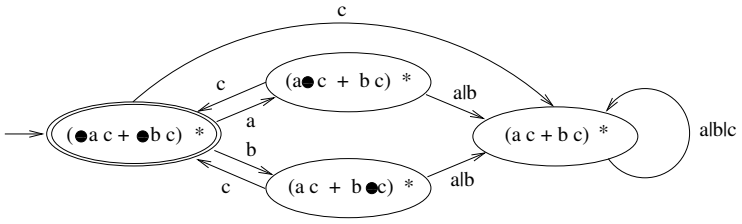

Fig. 1. DFA for $(ac + bc)^*$

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 automata 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!).

Example 5. Starting form the regular expression $(a+\epsilon)(b^*a+b)b$, we obtain the automata of Figure 2

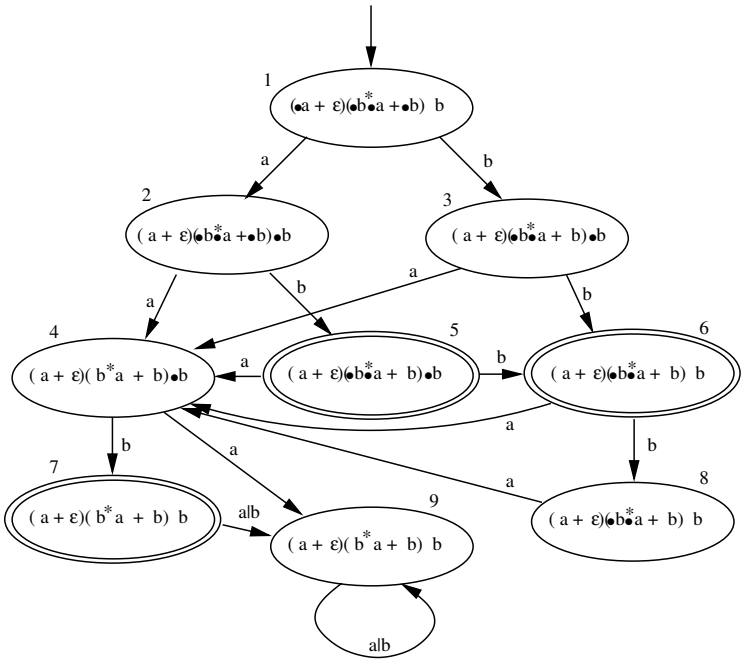

Fig. 2. DFA for $(a+\epsilon)(b^*a+b)b$

Quite remarkably, this DFA is minimal! (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).

4 Conclusions and future work

We introduced in this paper the notion of pointed regular expression. Points are used to mark the positions inside the regular expression which are reached after reading some prefix of the input string, and where the processing of the remaining string should start. In particular, each pointed expression has a clear semantics. Since each pointed expression for e represents a state of the *deterministic* automaton associated with e, this means we may associate a semantics

to each state in terms of the specification e and not of the behaviour of the automaton. This allows a *direct*, *intuitive* and *easily verifiable* construction of the deterministic automaton for e. We used pointed expressions during the last couple of years for teaching the argument to students and, according to our experience, they are clearly superior to any other method we are aware of.

From a theoretical point of view, pointed expressions seem to open a wide range of novel perspectives and original approaches, partially outlined in the extended version [3], that also contains a discussion of minimization and a detailed comparison with Brzozowski's derivatives (that turned out to be less trivial than expected, and is likely to be improved passing through Antimirov's partial derivatives [1]).

The most interesting operation is *merging* (denoted with a $\dagger$), defined by overlapping two pointed expression with a same carrier. As a matter of fact, the very reason allowing a direct construction of the DFA is that the semantics of pres behaves additively w.r.t. merging, that is

$$L_p(e_1 \dagger e_2) = L_p(e_1) \cup L_p(e_2)$$

This also explains why the techinque cannot be trivially extended to intersection and complement, since merging is no longer additive. The problem has a deep theoretical reason: indeed, even if these operators do not increase the expressive power of regular expressions they can have a drastic impact on succinctness. In particular it is well known that expressions with complements can provide descriptions of certain languages which are non-elementary more compact than standard regular expression [16]. Gelade [10] has recently proved that for any natural number n there exists a regular expression with intersection of size $\mathcal{O}(n)$ such that any DFA accepting its language has a double-exponential size, i.e. it contains at least 2^{2^n} states (see also [12]). Hence, marking positions with points is not enough, just because we would not have enough states. This seems to suggest the possibility to exploit more complex variants of pres, using for instance colors to recover the additive property.

A large amount of research has been recently devoted to the so called succinteness problem, namely the investigation of the descriptional complexity of regular languages (see e.g. [10, 12, 13]). Since, as observed in Example1, pointed expression can provide a more compact description for regular languages than traditional regular expression, it looks interesting to better investigated this issue (that seems to be related to the so called star-height [8] of the language).

It could also be worth to investigate variants of the notion of pointed expression, allowing different positioning of points inside the expressions. Merging must be better investigated, and the whole equational theory of pointed expressions, both with different and (especially) fixed carriers must be entirely developed.

All the results is the paper have been formalized and checked in the interactive theorem prover Matita [2], that actually provided the original motivation for our work.

References

1. Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155:291–319, 1996.
2. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.
3. Andrea Asperti, Enrico Tassi, and Claudio Sacerdoti Coen. Regular expressions, au point. *eprint arXiv:1010.2604*, 2010.
4. Gérard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(3):117–126, 1986.
5. Anne Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.
6. Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
7. 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.
8. Lawrence C. Eggan. Transition graphs and the star-height of regular events. *Michigan Mathematical Journal*, 10(4):385–397, 1963.
9. Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming wei Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.
10. Wouter Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theor. Comput. Sci.*, 411(31-33):2987–2998, 2010.
11. V.M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
12. Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. In *ICALP*, volume 5126 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2008.
13. Markus Holzer and Martin Kutrib. Nondeterministic finite automata - recent results on the descriptional and computational complexity. *Int. J. Found. Comput. Sci.*, 20(4):563–580, 2009.
14. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.
15. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Ieee Transactions On Electronic Computers*, 9(1):39–47, 1960.
16. Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 125–129. IEEE, 1972.
17. Scott Owens, John H. Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, 2009.
18. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 1: word, language, grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.
19. Bruce W. Watson. A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata. *South African Computer Journal*, 27:12–17, 2001.
20. Bruce W. Watson. Directly constructing minimal dfas : combining two algorithms by brzozowski. *South African Computer Journal*, 29:17–23, 2002.