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 join the elegance and the symbolic appealingness of Brzozowski’s derivatives, with the effectiveness of McNaughton and Yamada’s labelling technique, essentially combining the best of the two approaches.

Categories and Subject Descriptors F.1.1 [Models of Computation]

General Terms Theory

Keywords Regular expressions, Finite States Automata, Derivatives

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. [1,2] for some recent surveys). All the studies in this area have been traditionally inspired by two precursory, basilar works: Brzozowski’s theory of derivatives [3], and McNaughton and Yamada’s algorithm [4]. The main advantages of derivatives are that they are syntactically appealing, easy to grasp and to prove correct (see [5] 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 [6] 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 [7,8]).

Regular expressions are such small world that it is much at no one’s surprise that all different approaches, at the end, turn out to be equivalent; still, their philosophy, their underlying intuition, and the techniques to be deployed can be sensibly different. Without having the pretension to say anything really original on the subject, we introduce in this paper a notion of pointed regular expression, that provides a cheap palliative for derivatives and allows a simple, direct and efficient construction of the deterministic finite automaton. Remarkably, the formal correspondence between pointed expressions and Brzozowski’s derivatives is unexpectedly entangled (see Section 4.1) testifying the novelty and the not-so-trivial nature of the notion.

The idea of pointed expressions was suggested by an attempt of formalizing the theory of regular languages by means of an interactive prover[9]. At first, we started considering derivatives, since they looked more suitable to the kind of symbolic manipulations that can be easily dealt with by means of these tools. However, the need to consider sets of derivatives and, especially, to reason modulo associativity, commutativity and idempotence of sum, prompted us to look for an alternative notion. Now, it is clear that, in some sense, the derivative of a regular expression $e$ is a set of “subexpressions” of $e$; the only, crucial, difference is that we cannot forget their context. So, the natural solution is to point at subexpressions inside the original term. This immediately leads to the notion of pointed regular expression (pre), that is just a normal regular expression where some positions (it is enough to consider individual characters) have been pointed out. Intuitively, the 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 [9] 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. [1], pag.71).

In the imposing bibliography on regular expressions - as far as we could discover - the only author mentioning a notion close to ours is Watson [10,11]. 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 [21]).

\[1\] The rule of the game was to avoid overkilling, i.e. not make it more complex than deserved.
\[2\] This is also the reason why, at the end, we only have a finite number of derivatives.
2. Regular expressions

**Definition 1.** A regular expression over the alphabet Σ is an expression \( e \) generated by the following grammar:
\[
E ::= \emptyset \mid e \mid a \mid E + E \mid EE \mid E^*
\]
with \( a \in \Sigma \).

**Definition 2.** The language \( L(e) \) associated with the regular expression \( e \) is defined by the following rules:
\[
\begin{align*}
L(\emptyset) & = \emptyset \\
L(e) & = \{e\} \\
L(a) & = \{a\} \\
L(e_1 + e_2) & = L(e_1) \cup L(e_2) \\
L(e_1 e_2) & = L(e_1) \cdot L(e_2) \\
L(e^*) & = L(e)^*
\end{align*}
\]
where \( e \) is the empty string, \( L_1 \cdot L_2 = \{ 1tL_2 | t \in L_1 \} \) \( , L_1 \cup L_2 \) 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 \( L^0 = e \) and \( L^{i+1} = L \cdot L^i \).

**Definition 3 (nullable).** A regular expression \( e \) is said to be nullable if \( e \in L(e) \).

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:
\[
\begin{align*}
\nu(\emptyset) & = \text{false} \\
\nu(e) & = \text{true} \\
\nu(a) & = \text{false} \\
\nu(e_1 + e_2) & = \nu(e_1) \lor \nu(e_2) \\
\nu(e_1 e_2) & = \nu(e_1) \land \nu(e_2) \\
\nu(e^*) & = \text{true}
\end{align*}
\]

**Definition 4.** A deterministic finite automaton (DFA) is a quintuple \( (Q, \Sigma, q_0, t, F) \) where
- \( Q \) is a finite set of states;
- \( \Sigma \) 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:

**Definition 5.** Given a function \( t : Q \times \Sigma \to Q \), the function \( t^* : Q \times \Sigma^* \to Q \) is defined as follows:
\[
t^*(q, w) = \begin{cases} 
  t(q, a) = q & \text{if } w = a \in \Sigma \\
  t(q, aw') = t^*(t(q, a), w') & \text{if } w \notin \Sigma
\end{cases}
\]

**Definition 6.** 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 7.**
1. A pointed item over the alphabet \( \Sigma \) is an expression \( e \) generated by following grammar:
\[
E ::= \emptyset | e | a | E + E | EE | E^* \\
\]
with \( a \in \Sigma \).
2. A pointed regular expression \( (pe) \) is a pair \( (e, b) \) where \( b \) is a boolean and \( e \) is a pointed item.

The term \( \bullet e \) 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 8.** 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 \( e(b) \), where \( b \) is a boolean, with the following meaning:
\[
e(\text{true}) = \{e\} \quad e(\text{false}) = \emptyset
\]

**Definition 9.**
1. The language \( L_p(e) \) associated with the item \( e \) is defined by the following rules:
\[
\begin{align*}
L_p(\emptyset) & = \emptyset \\
L_p(e) & = \emptyset \\
L_p(a) & = \emptyset \\
L_p(\bullet a) & = \{a\} \\
L_p(e_1 + e_2) & = L_p(e_1) \cup L_p(e_2) \\
L_p(e_1 e_2) & = L_p(e_1) \cdot L_p(e_2) \\
L_p(e^*) & = L_p(e) \cdot L_p(e)^*
\end{align*}
\]
2. For a pointed regular expression \( (e, b) \) we define
\[
L_p((e, b)) = L_p(e) \cup \{e\}
\]

**Example 10.**
\[
L_p((a + \bullet b)^*) = L(b(a + b)^*)
\]

Indeed,
\[
L_p((a + \bullet b)^*) = L_p(a + \bullet b)^* = L_p(a) \cup L_p(\bullet b)^* = \{b\} \cdot L(a + b)^* = L(b(a + b)^*)
\]

Let us incidentally observe that, as shown by the previous example, 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. \[12\] [13] [14]).

**Example 11.** If \( e \) contains no point (i.e. \( e = |e| \)) then \( L_p(e) = \emptyset \)

**Lemma 12.** If \( e \) is a pointed item then \( e \notin L_p(e) \). Hence, \( e \in L_p((e, b)) \) if and only if \( b = \text{true} \).

**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 \( e \)-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 \( (e_1, \text{true}) \) with another \( (e_2, b) \), we must first broadcast the trailing point of the first expression inside \( e_2 \) and then prepend \( e_1 \); similarly for the star operation. We could define first the broadcasting function \( \circ (\cdot) \) 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 13.

1. The function $\bullet(\cdot)$ from pointed item to pres is defined as follows:
   - $\bullet(\emptyset) = (\emptyset, \text{false})$
   - $\bullet(e) = (e, \text{true})$
   - $\bullet(a) = (a, \text{false})$
   - $\bullet(e_1 + e_2) = (e_1) \bullet (e_2)$
   - $\bullet(e_1 \cdot e_2) = (e_1 \odot (e_2, \text{false}))$
   - $\bullet(e^*) = (e^*, \text{true})$ where $\bullet(e) = (e^*, b')$

2. The lifted constructors are defined as follows
   - $\langle e_1, b'_1 \rangle \oplus (e_2, b'_2) = (e_1 + e_2, b'_1 \lor b'_2)$ when $b'_1 = \text{false}$
   - $\langle e_1, b'_1 \rangle \odot (e_2, b'_2) = (e_1 e_2, b'_1 \lor b'_2)$ when $b'_1 = \text{true}$
   - $\langle e', b' \rangle^* = \begin{cases} (e'^*, \text{false}) & \text{when } b' = \text{false} \\ (e'^*, \text{true}) & \text{when } b' = \text{true} \\ \bullet(e') = (e^*, b'') \end{cases}$

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 14. Suppose to broadcast a point inside

$$ (a + e)(b^* a + b)b $$

We start working in parallel on the first occurrence of $a$ (where the point stops), and on $e$ that gets traversed. We have hence reached the end of $a + e$ 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:

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

Definition 15. The broadcasting function is extended to pres in the obvious way:

$$ \bullet(e, b) = (e^*, b \lor b') $$

As we shall prove in Corollary 13 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 16.

1. $L_p(e) = L_p(e) \cup L([e])$.
2. $L_p(e_1 + e_2) = L_p(e_1) \cup L_p(e_2)$
3. $L_p(e_1 \cdot e_2) = L_p(e_1) \cdot L([e_2]) \cup L_p(e_2)$
4. $L_p(e^*) = L_p(e) \cdot L([e])^*$

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 $L_p(e_1 + e_2) = L_p(e_1) \cup L_p(e_2)$.

$$ L_p((e', b'_1) \oplus (e'_2, b'_2)) = $$
$$ = L_p(e'_1 + e'_2, b'_1 \lor b'_2) = $$
$$ = L_p(e'_1) \cup e'_1(b'_2) \cup L_p(e'_2) \lor e'_2(b'_2) = $$
$$ = L_p(e'_1) \cup L_p(e'_2) $$

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

- $L_p(e) = L_p(e) \cup L([e]) = \emptyset = L_p(\emptyset) \cup L([\emptyset])$.
- $L_p(e) = L_p(e) \cup L([e]) = \{e\} = L_p(e) \cup L_p(e)$(e).
- $L_p(a) = L_p(a) \cup L([a]) = \{a\} = L_p(a) \cup L([a])$.
- $L_p(e_1 + e_2) = L_p(e_1) \cup L([e_1]) \cup L_p(e_2) \cup L([e_2]) = L_p(e_1) \cup L_p(e_2)$.

Thus, by 2., we have

$$ L_p(e_1 + e_2) = $$
$$ = L_p(e_1) \cup L([e_1]) \cup L_p(e_2) \cup L([e_2]) = L_p(e_1) \cup L_p(e_2) $$

Let $e = e_1 + e_2$. By induction hypothesis we know that

$$ L_p(e_1 + e_2) = L_p(e_1) \cup L([e_1]) $$

Thus, by 3. over the structurally smaller terms $e_1$ and $e_2$

- $L_p(e_1 + e_2) = L_p(e_1) \cup e_1(b'_2) = L_p(e_1) \cup L([e_1]) \cup L_p(e_2) \cup L([e_2])$

Thus, by 3. over the structurally smaller terms $e_1$ and $e_2$

- $L_p(e_1 + e_2) = L_p(e_1) \cup L([e_1]) \cup L_p(e_2) \cup L([e_2])$

and in particular, since by Lemma 12 $e \not\in L_p(e_1)$,

$$ L_p(e_1) = L_p(e_1) \cup L([e_1]) \cup L_p(e_1) $$

Then,

$$ L_p(e_1^*) = $$
$$ = L_p(e_1^*, \text{true}) $$
$$ = L_p(e_1^*) \cup e $$
$$ = L_p(e_1^*) \cup L([e_1] \cup e_1(b'_2)) \cup L([e_1]) \cup e $$
$$ = L_p(e_1) \cup L([e_1]) \cup e_1(b'_2) \cup L([e_1]) \cup e $$

Haven 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.: $L_p(e_1 \cdot e_2) = L_p(e_1 \cdot L([e_2]) \cup L_p(e_2)$

We distinguish the two cases of the definition of $\odot$:
\[ L_p((e_1', false) \odot (e_2', b_2')) = \\
= L_p((e_1' \odot (e_2', b_2')) = \\
= L_p(e_1') \cdot L(\{(e_2', b_2')\}) \cup L_p(e_2') \cup \epsilon(b_2') \\
= L_p(e_1') \cdot L(\{(e_2', b_2')\}) \cup L_p(e_2') \cup \epsilon(b_2') \\
\]

\[ L_p((e_1', true) \odot (e_2', b_2')) = \\
= L_p((e_1' \odot (e_2', b_2')) = \\
= L_p(e_1') \cdot L(\{(e_2', b_2')\}) \cup L_p(e_2') \cup \epsilon(b_2') \\
= L_p(e_1') \cdot L(\{(e_2', b_2')\}) \cup L_p(e_2') \cup \epsilon(b_2') \\
\]

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

\[ L_p((e', false)^*) = \\
= L_p((e'^* \cdot false)) = \\
= L_p(e'^*) = \\
= L_p(e') \cdot L(|e'|)^* = \\
= L_p(e') \cdot L(|e'|)^* \\
\]

\[ L_p((e', true)^*) = \\
= L_p((e'^* \cdot true)) \cup \epsilon = \\
= L_p(e'^*) \cup \epsilon = \\
= L_p(e') \cdot L(|e'|)^* \cup \epsilon = \\
= L_p(e') \cdot L(|e'|)^* \cup \epsilon \\
\]

Corollary 17. For any regular expression e, \( L(e) = L_p(\epsilon e) \).

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 18. \( \epsilon e = (\epsilon', true) \) if and only if \( e \in L(|e|) \).

Proof. By theorem 16, we know that \( L_p(e \odot e) = L_p(e) \cup L(\epsilon) \).

To conclude this section, let us prove the lemmata about the •( ) function (it will only be used in Section 5 and can be skipped at a first reading). To this aim we need a technical lemma whose straightforward proof by case analysis is omitted.

Lemma 19. 1. \( \bullet(e_1 \odot e_2) = \bullet(e_1) \odot \bullet(e_2) \)

2. \( \bullet(e_1) \odot \bullet(e_2) = \bullet(e_1 \odot e_2) \)

Theorem 20. \( \bullet(\bullet(e)) = \bullet(e) \)

Proof. The proof is by induction on e.

- \( \bullet(\emptyset) = (\emptyset, false) = (\emptyset, false \lor false) = \bullet(\emptyset) \)
- \( \bullet(e') = (e', true \lor false) = (e', true \lor true) = (e') \)
- \( \bullet(a) = (a, false \lor false) = (a, false \lor false) = \bullet(a) \)
- \( \bullet(e \odot a) = (e, false \lor false) = \bullet(e) \)
- If e is \( e_1 + e_2 \) then
  \[ \bullet(e_1 + e_2) = \bullet(e_1) \odot \bullet(e_2) = (\bullet(e_1)) \odot \bullet(e_2) = (\bullet(e_1)) \odot \bullet(e_2) = \]

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 21.

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

   \[ move(\emptyset, a) = (\emptyset, false) \]
   \[ move(e, a) = (e, false) \]
   \[ move(b, a) = (b, false) \]
   \[ move(\bullet a, a) = (a, true) \]
   \[ move(\bullet b, a) = (b, false) \]
   \[ move(c_1 + e_2, a) = move(c_1, a) \odot move(e_2, a) \]
   \[ move(c_1 : e_2, a) = move(e_1, a) \odot move(e_2, a) \]
   \[ move(e'^* \cdot a, a) = move(e', a) \]

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

Example 22. Let us consider the pre ((\bullet a + \epsilon)((\bullet b)^* \cdot a + \bullet b)b and the two moves w.r.t. the characters a and b.

Theorem 23. 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 then \( L_p(move(e, a)) = L_p(\epsilon) \) and hence both sides are true.
- If e = \( \bullet a \) then \( L_p(move(\bullet a, a)) = L_p(\epsilon) \) and hence both sides are false.
- If e = \( \bullet b \) with \( b \neq a \) then \( L_p(move(\bullet b, a)) = L_p(\emptyset) \) and hence both sides are false.
- If e = \( \bullet c_1 + e_2 \) then induction hypothesis \( w \in L_p(move(c_1, a)) \Leftrightarrow aw \in L_p(e) \), hence.

short description of paper
for some $e'$. By the previous theorem, this is possible if an only if $w \in L_p(\bullet(e))$, and by corollary 17 $L_p(\bullet(e)) = L(e)$.

**Remark 29.** The fact that the set $Q$ of states of $D_n$ 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 a relatively complex property to prove (see \cite{3}). The automaton $D_n$ 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 30.** Let $e$ be a regular expression and let $q_0$ be $\bullet e$. Let also

$$Q_0 := \{q_0\} \quad Q_{n+1} := Q_n \cup \{e'|e' \notin Q_n \land \exists \alpha. \exists e \in Q_n. \text{move}(e, a) = e'\}$$

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.

![Figure 1. DFA for $(a + e)(b * a + b)b$](image)

In Figure 1 we describe the DFA associated with the regular expression $(a + e)(b * a + b)b$. 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 $(e, b)$ is final if and only if $b$ is true, we may just label nodes with the item (for instance, the pair of states $6 - 8$ and $7 - 9$ only differ for the fact that 6 and 7 are final, while 8 and 9 are not).

### 3.3 From regular expressions to DFAs

**Definition 27.** 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;
- $\Sigma$ is the alphabet of the regular expression $e$;
- $q_0$ is $\bullet e$;
- $t$ is the move operation of definition 27;
- $F$ is the subset of pointed expressions $(e, b)$ with $b = \text{true}$.

**Theorem 28.** $L(D_e) = L(e)$

**Proof.** By definition,

$$w \in L(D_e) \iff \text{move}^*(\bullet(e), w) = (e', \text{true})$$

### 3.4 Admissible relations and minimization

The automaton in Figure 1 is minimal. This is not always the case. For instance, for the expression $(ac + bc)^*$ we obtain the automaton of Figure 2 and it is easy to see that the two states corresponding to the pres $(a \bullet c + c \bullet e)^*$ and $(ac + b \bullet e)^*$ are equivalent (a way to prove it is to observe that they define the same language). The latter remark, motivates the following definition.

---

**short description of paper**

5 2010/10/14
The set of admissible equivalence relations over $e$ is a bounded lattice, ordered by refinement, whose bottom element is syntactic identity and whose top element is $e_1 \approx e_2$ if $L(e_1) = L(e_2)$. Moreover, if $\approx_1 \subset \approx_2$ (the first relation is a strict refinement of the second one), the number of states of $D_{e_1}/\approx_1$ is strictly larger than the number of states of $D_{e_2}/\approx_2$.

**Theorem 36.** If $\approx$ is the top element of the lattice, then $D_{e_1}/\approx$ is the minimal automaton that recognizes $L(e)$.

**Proof.** By the previous theorem, $D_{e}/\approx$ recognizes $L(e)$ and has no unreachable states. By absurd, let $D' = (Q', \Sigma', q_0', \tau', F')$ be another smaller automaton that recognizes $L(e)$. Since the two automata are different, recognize the same languages and have no unreachable states, there exists two words $w_1, w_2$ such $\tau'(q_0, w_1) = \tau'(q_0, w_2)$ but $[e_1]/\approx = [move^*]/\approx([q_0]/\approx, w_1) \neq [move^*]/\approx([q_0]/\approx, w_2) = [e_2]/\approx$ where $e_1$ and $e_2$ are any two representatives of their equivalence classes and thus $e_1 \neq e_2$. By definition of $\approx$, $L(p(e_1)) \neq L(p(e_2))$. Without loss of generality, let $w_3 \in L(p(e_1)) \setminus L(p(e_2))$. We have $w_3 w_2 \in L(e)$ and $w_2 w_3 \notin L(e)$ because $D_{e_1}/\approx$ recognizes $L(e)$, which is absurd since $\tau'(q_0, w_1 w_3) = \tau'(q_0, w_2 w_3)$ and $D'$ also recognizes $L(e)$.

The previous theorem tells us that it is possible to associate to each state of an automaton for $e$ (and in particular to the minimal automaton) a pre $e'$ over $e$ so that the language recognized by the automaton in the state $e'$ is $L(p(e'))$, that provides a very suggestive labelling of states.

The characterization of the minimal automaton we just gave does not seem to entail an original algorithmic construction, since does not suggest any new effective way for computing $\approx$. However, similarly to what has been done for derivatives (where we have similar problems), it is interesting to investigate admissible relations that are easier to compute and tend to produce small automata in most practical cases. In particular, in the next section, we shall investigate one important relation providing a common quotient between the automata built with pres and with Brzozowski’s derivatives.

**4. Read back**

Intuitively, a pointed regular expression corresponds to a set of regular expressions. In this section we shall formally investigate this “read back” function; this will allow us to establish a more syntactic relation between traditional regular expressions and their pointed version, and to compare our technique for building a DFA with that based on derivatives.

In the following sections we shall frequently deal with sets of regular expressions (to be understood additively), that we prefer to the treatment of regular expressions up to associativity, commutativity and idempotence of the sum (ACI) that is for instance typical of the traditional theory of derivatives (this also clarifies that ACI-rewriting is only used at the top level).

It is hence useful to extend some syntactic operations, and especially concatenation, to sets of regular expressions, with the usual distributive meaning: if $e$ is a regular expression and $S$ is a set of regular expressions, then

$$Se = \{e' | e' \in S\}$$

We define $eS$ and $S_1 S_2$ in a similar way. Moreover, every function on regular expressions is implicitly lifted to sets of regular expressions by taking its image. For example,

$$L(S) = \bigcup_{e \in S} L(e)$$
\textbf{THEOREM 46.} For any pointed regular expression $c,$

\[ R(\bullet(e)) = nf_c(|e|) \cup R(e) \]

\textit{Proof.} Let $\bullet(e) = \langle \epsilon', b' \rangle;$ then $e \in R(\bullet(e))$ iff $b' = true,$ iff $\nu(|e|) = true.$ Hence the goal reduces to prove that $R(\epsilon') = nf(|e|) \cup R(e).$ We proceed by induction on the structure of $e.$

\begin{itemize}
  \item $e = \emptyset : \bullet(\emptyset) = \langle \emptyset, false \rangle$ and $R(\emptyset) = \emptyset = nf(\emptyset)$
  \item $e = e_1 \bullet (e) = \langle e, true \rangle$ and $R(e) = \emptyset = nf(e)$
  \item $e = a : \bullet(a) = \langle a, false \rangle$ and $R(a) = \{a\} = nf(a)$
  \item $e = \bullet a : \bullet(\bullet(a)) = \langle a, false \rangle$ and $R(\bullet a) = \{a\} = nf(a)$
  \item $e = e_1 + e_2 : let \bullet(\epsilon_1 + e_2) = (\epsilon'_1 + e_2, b);$ then
    \[ R(\epsilon'_1 + e_2) = R(\epsilon'_1) \cup R(e_2) \]
    \[ = nf(\epsilon'_1) \cup R(e_2) \]
    \[ = nf(\epsilon'_1) \cup nf(|e_2|) \cup R(e_2) \]
    \[ = nf(\epsilon'_1) \cup R(e_2) \]
    \[ = (nf(\epsilon'_1) \cup R(\epsilon'_1)) | e_2 | \cup R(e_2) \]
    \[ = nf(\epsilon'_1) | e_2 | \cup R(e_2) \]
    \[ = (nf(\epsilon'_1) | e_2 |) \cup R(e_2) \]
    \[ = (nf(\epsilon'_1) | e_2 |) \cup R(\epsilon_1 + e_2) \]
    \[ = \epsilon \in \emptyset. \]
  \end{itemize}

\section{THEOREM 47. For all regular expression $e,$ $R(\bullet(e)) = nf_e(e)$}

To conclude this section, in analogy with what we did for the semantic function in Theorem 16, we express the behaviour of $R$ in terms of the \textit{lifted} algebraic constructors. This will be useful in Theorem 51.

\begin{itemize}
  \item $R(\epsilon_1 + \epsilon_2) = R(\epsilon_1) \cup R(\epsilon_2)$
  \item $R(\epsilon_1 + \epsilon_2) \circ \epsilon_2 = R(\epsilon_1) | e_2 | \cup R(e_2)$
  \item $R(\epsilon_1 \bullet \epsilon_2) = R(\epsilon_1) | e_2 | = R(\epsilon_1) | e_2 | \cup R(\epsilon_2)$
  \item $R(\epsilon'_1, false')^* = R(\epsilon'_1)^*| e_1^* \cup R(e_1)^* | e_1^* \cup R(e_2)$
  \item $R(\epsilon'_1, true')^* = R(\epsilon'_1)^*| e_1^* \cup R(e_1)^* | e_1^* \cup R(e_2)$
\end{itemize}

\textit{Proof.} Let $e = \langle \epsilon'_1, b' \rangle$:

\begin{itemize}
  \item $R(\epsilon_1 + \epsilon_2) = R(\epsilon_1) \cup \text{langl}(\epsilon'_1, b'_1) \cup \text{langl}(\epsilon'_2, b'_2)$
  \item $R(\epsilon_1 \bullet \epsilon_2) = R(\epsilon_1) | e_2 | = R(\epsilon_1) | e_2 | \cup R(\epsilon_2)$
  \item $R(\epsilon_1 \bullet \epsilon_2) = R(\epsilon_1) | e_2 | = R(\epsilon_1) | e_2 | \cup R(\epsilon_2)$
  \item $R(\epsilon_1) \cup \epsilon(\epsilon_1) \cup R(\epsilon_2) \cup \epsilon(\epsilon_2)$
  \item $R(\epsilon'_1) \cup \epsilon(\epsilon'_1) \cup R(\epsilon'_2) \cup \epsilon(\epsilon'_2)$
\end{itemize}
2.  \( R((e_1', false) \odot (e_2', b_2')) = \)
\[ = R(e_1'e_2', b_2') \]
\[ = R(e_1'e_2') \cup R(e_2') \cup \epsilon(b_2') \]
\[ = R(e_1'e_2') \cup R(e_2') \cup \epsilon(b_2') \]
\[ = (R(e_1')|e_2') \cup n_{nf}(e_2') \cup R(e_2') \cup \epsilon(b_2') \]
\[ = R(e_1'e_2') \cup n_{nf}(e_2') \cup R(e_2') \]
3. let \( \bullet(e_2') = (e_2', b_2') \)
\[ R((e_1', true) \odot (e_2', b_2')) = \]
\[ = R(e_1'e_2', b_2' \cup b_2') \]
\[ = R(e_1'e_2') \cup R(e_2') \cup \epsilon(b_2') \cup \epsilon(b_2') \]
\[ = (R(e_1')|e_2') \cup R(e_2') \cup \epsilon(b_2') \]
\[ = (R(e_1')|e_2') \cup n_{nf}(e_2') \cup R(e_2') \cup \epsilon(b_2') \]
4.  \( R((e_1', false)^* ) = R((e_1'^*, false)) = R(e_1'^* ) = R(e_1' | e_1 | e_1 | ... | e_1) \)
5. let \( \bullet(e_1') = (e_1'^*, b_1'^* ) \); then \( R(\bullet(e_1')) = R(e_1'^* ) \cup \epsilon(b_1'^* ) = n_{nf}(e_1'^* ) \cup R(e_1') \)
\[ R((e_1', true)^*) = \]
\[ = R((e_1'^*, true)) \]
\[ = R(e_1'^* | e_1' \cup \epsilon(true) \]
\[ = (R(e_1') | \epsilon b_{nf}(e_1')) | e_1'^* \cup \epsilon(true) \]
\[ = R(e_1'| e_1' \cup n_{nf}(e_1')) \cup \epsilon(true) \]
\[ = R(e_1'| e_1' \cup n_{nf}(e_1')) \]
\[ & \quad \text{4.1 Relation with Brzozowski’s Derivatives} \]

We are now ready to formally investigate the relation between pointed expressions and Brzozowski’s derivatives. As we shall see, they give rise to quite different constructions and the relation is less obvious than expected.

Let’s start with recalling the formal definition.

**DEFINITION 49.**
\[
\partial_a(\emptyset) = \emptyset \\
\partial_a(e) = e \\
\partial_a(a) = e \\
\partial_a(b) = \emptyset \\
\partial_a(e_1 + e_2) = \partial_a(e_1) + \partial_a(e_2) \\
\partial_a(c(e_1e_2)) = \partial_a(e_1)c(e_2) \text{ if not } \nu(e_1) \\
\partial_a(e_1^*) = \partial_a(e_1^*) + \partial_a(e_2) \text{ if } \nu(e_1) \\
\partial_a(e^*) = \partial_a(e)^* 
\]

**DEFINITION 50.**
\[
\partial_a(e) = e \\
\partial_{aw}(e) = \partial_a(\partial_a(e)) 
\]

In general, given a regular expression \( e \) over the alphabet \( \Sigma \), the set \( \{ \partial_w(e) | w \in \Sigma^* \} \) of all its derivatives is not finite. In order to get a finite set we must suitably quotient derivatives according to algebraic equalities between regular expressions. The choice of different set of equations gives rise to different quotients, and hence to different automata. Since for finiteness it is enough to consider associativity, commutativity and idempotence of the sum (ACI), the traditional theory of Brzozowski’s derivatives is defined according to these laws (although this is probably not the best choice from a practical point of view).

As a practical example, in Figure 3 we describe the automaton obtained using derivatives relative to the expression \((ac + bc)^* \) (compare it with the automata of Figure 2). Also, note that the vertically aligned states are equivalent.

Let us remark, first of all, the heavy use of ACI. For instance
\[
\partial_a((ac + bc)^*) = (ac + bc)(ac + bc)^* \\
\partial_a((ac + bc)^*) = (bc + cc)(ac + bc)^* 
\]
and they can be approximated only up to commutativity of the sum. As another example,
\[
\partial_a((ac + bc)(ac + bc)(ac + bc)^*) = \\
= (ac + bc)(ac + bc)^* + (bc + cc)(ac + bc)(ac + bc)^* \\
= (ac + bc)(ac + bc)(ac + bc)(ac + bc)^* + (bc + cc)(ac + bc)(ac + bc)^* \\
\]

only using associativity and idempotence of the sum.

The second important remark is that, in general, it is not true that we may obtain the pre-automata by quotienting the derivative one (nor the other way round). For instance, from the initial state, the two arcs labelled \( a \) and \( b \) lead to a single state in the automata of Figure 3 but in different states in the automata of Figure 2.

A natural question is to understand if there exists a common algebraic quotient between the two constructions (not exploiting minimization).

As we shall see, this can be achieved by identifying states with a same readback in the case of pres, and states with similar look-ahead normal form in the case of derivatives.

For instance, in the case of the two automata of Figures 2 and 3 we would obtain the common quotient of Figure 4.

**Figure 3. Automaton with Brzozowski’s derivatives**

**Figure 4. A quotient of the two automatons**

The general picture is described by the commuting diagram of Figure 5 whose proof will be the object of the next section (in Figure 5 \( w \) obviously stands for the string \( a_1 \ldots a_n \)).

**4.2 Formal proof of the commuting diagram in Figure 5**

Part of the diagram has already proved: the leftmost triangle, used to relate the initial state of the two automata, is Corollary 57; the two triangles at the right, used to relate the final states, just states the trivial properties that \( e \in R(e, b) \) iff and only if \( b = true \) (since no expression in \( R(e) \) is nullable), and \( e \in n_{nf}(e) \) if and only if \( e \) is nullable (see Remark 52).

We start proving the upper part. We prove it for a pointed item \( e \) and leave the obvious generalization to a pointed expression to the reader (the move operation does not depend from the presence of a trailing point, and similarly the derivative of \( e \) is empty).

**THEOREM 51.** For any pointed item \( e \),
\[
R(move(e, a)) = n_{nf}(\partial_a(R(e))) 
\]

**Proof.** By induction on the structure of \( e \):
- the cases ∅, ε, a and b are trivial
- if e = a then move(a, a) = (a, true) and R(a, true) = {e}. On the other side, nf(∂(R(a))) = nf(∂(a)) = ε.
- if e = e₁ + e₂ then
  \[
  R(\text{move}(e₁ + e₂)) = \begin{cases}
  R(\text{move}(e₁)) \cup R(\text{move}(e₂)) & \text{if } e₁ \neq e₂ \\
  R(\text{move}(e₁ + e₂)) & \text{if } e₁ = e₂
  \end{cases}
  \]
- let e = e₁e₂, and let us suppose that move(e₁, a) = (e₁, false) and thus R(move(e₁, a)) = R(e₁) and ν(∂(R(e₁))) = false. We have then:
  \[
  R(\text{move}(e₁e₂)) = \begin{cases}
  R(\text{move}(e₁)) \cup R(\text{move}(e₂)) & \text{if } e₁ \neq e₂ \\
  R(\text{move}(e₁e₂)) & \text{if } e₁ = e₂
  \end{cases}
  \]
- let e = e₁, and let us suppose that move(e₁, a) = (e₁, false). Thus e ∉ nf(∂(R(e₁))). Then
  \[
  R(\text{move}(e₁, a)) = \begin{cases}
  R(\text{move}(e₁, a)) & \text{if } e₁ \neq e₂ \\
  R(e₁e₂) & \text{if } e₁ = e₂
  \end{cases}
  \]
  We pass now to prove the lower part of the diagram in Figure 5 namely that for any regular expression e,
  \[
  nf(∂(e)) = nf(∂(nf(e)))
  \]
  Since however, nf(∂(e)) = nf(∂(nf(e))) (the derivative of e is empty), this is equivalent to prove the following result.

**Theorem 52.** nf(∂(e)) = nf(∂(nf(e)))

**Proof.** The proof is by induction on e. Any induction hypothesis over a regular expression e₁ can be strengthened to nf(∂(e₁))e₂ = nf(∂(nf(e₁)))e₂ for all e₂ since
  \[
  nf(∂(e₁))e₂ = nf(∂(nf(e₁)))e₂ 
  \]
(observe that ν(∂(e₁)) = ν(∂(nf(e₁)))) since the languages denoted by ∂(e₁) and ∂(nf(e₁)) are equal.

We must consider the following cases.
- If e is ε, ∅ or a symbol b different from a then both sides of the equation are empty
- If e is a, nf(∂(a)) = {ε} = nf(∂(a)) = nf(∂(nf(a)))
- If e is e₁ + e₂,
  \[
  nf(∂(e₁ + e₂)) = \begin{cases}
  nf(∂(e₁)) + nf(∂(e₂)) & \text{if } e₁ \neq e₂ \\
  nf(∂(e₁)) & \text{if } e₁ = e₂
  \end{cases}
  \]
- If e is εe₂ and ν(e₁) = false,
  \[
  nf(∂(εe₂)) = nf(∂(ε))e₂ = nf(∂(nf(ε)))e₂ = nf(∂(∂(nf(ε))))e₂ = nf(∂(∂(nf(ε))))e₂
  \]
- If e is εe₂ and ν(e₁) = true,
  \[
  nf(∂(εe₂)) = nf(∂(ε))e₂ = nf(∂(nf(ε)))e₂ = nf(∂(∂(nf(ε))))e₂
  \]

Figure 5. Pointed regular expressions and Brzozowski’s derivatives
If $e$ is $e_1^*$,
\[ nf_\partial(\partial_w (e_1^*)) = nf_\partial(\partial_w (e_1)e_1^*) = nf_\partial(\partial_w (nf_\epsilon (e_1))e_1^*) = \]
\[ = nf_\partial(\partial_w (nf_\epsilon (e_1)e_1^*)) = nf_\partial(\partial_w (nf_\epsilon (e_1))e_1^*) \]

**Lemma 53.** $R(e) = nf_\epsilon (R(e))$

**Proof.** We proceed by induction over $e$:
- $R(\emptyset) = \emptyset = nf_\epsilon (\emptyset) = nf_\epsilon (R(\emptyset))$
- $R(e) = 0 = nf_\epsilon (R(e))$
- $R(a) = 0 = nf_\epsilon (R(a))$
- $R(a_1) = nf_\epsilon (a_1) = nf_\epsilon (R(a_1))$
- $R(e_1 \cup e_2) = R(e_1) \cup R(e_2) = nf_\epsilon (R(e_1)) \cup nf_\epsilon (R(e_2)) = nf_\epsilon (R(e_1 \cup R(e_2)) = nf_\epsilon (R(e_1) \cup R(e_2))$
- $R(e_1)e_2 = R(e_1)e_2 = nf_\epsilon (R(e_1))e_2 = nf_\epsilon (R(e_1)e_2) = nf_\epsilon (R(e_1)e_2) = nf_\epsilon (R(e_1)e_2)$
- $R(e_1^*) = R(e)e_1^* = nf_\epsilon (R(e))e_1^* = nf_\epsilon (R(e)e_1^*) = nf_\epsilon (R(e^*))$

We are now ready to prove the commutation of the outermost diagram.

**Theorem 54.** For any pointed item $e$,
\[ R(\text{move}^* (e, w)) = nf_\partial (\partial_w (R(e))) \]

**Proof.** The proof is by induction on the structure of $w$. In the base case, $R(\text{move}^* (e, e)) = R(e) = nf_\epsilon (R(e)) = nf_\partial (\partial_w (R(e)))$. In the inductive step, by Step 52
\[ R(\text{move}^* (e, a)) = \]
\[ = R(\text{move}^* (\text{move} (e, a), w)) = nf_\partial (\partial_w (R(\text{move}^* (e, a)))) = nf_\partial (\partial_w (R(e))) = nf_\partial (\partial_w (R(e))) \]

**Corollary 55.** For any regular expression $e$,
\[ R(\text{move}^* (\cdot, w)) = nf_\partial (\partial_w (R(e))) \]

**Theorem 56.** $kn(R(\cdot))$ (the kernel of $R(\cdot)$) is an admissible equivalence relation over $\mathcal{P}$.

**Proof.** By Lemma 59 we derive that for all pres $e_1, e_2$, if $R(e_1) = R(e_2)$ then $L(e_1) = L(e_2)$. We also need to prove that for all pres $e_1, e_2$ and all symbol $a$, if $R(e_1) = R(e_2)$ then $R(\text{move} (e_1, a)) = R(\text{move} (e_2, a))$. By Theorem 51
\[ R(\text{move} (e_1, a)) = nf_\partial (\partial_w (R(e_1))) = nf_\partial (\partial_w (R(e_2))) = \]
\[ = R(\text{move} (e_2, a)) \]

**Theorem 57.** $kn(nf_\epsilon (e))$ is an admissible equivalence relation over regular expressions

**Proof.** By Lemma 57 we derive that for all regular expressions $e_1, e_2$, if $nf_\epsilon (e_1) = nf_\epsilon (e_2)$ then $L(e_1) = L(e_2)$. We also need to prove that for all regular expressions $e_1, e_2$ and all symbol $a$, if $nf_\epsilon (e_1) = nf_\epsilon (e_2)$ then $nf_\partial (\partial_w (e_1)) = nf_\partial (\partial_w (e_2))$.

**Theorem 58.** For each regular expression $e$, let $D_e^* = (Q^*, \Sigma, e, \epsilon^*, F^*)$ be the automaton for $e$ built according to Definition 10 and let $D_e^* = (Q^*, \Sigma, e, \epsilon^*, F^*)$ be the automaton for $e$ obtained with derivatives. Let $kn(R)$ and $kn(nf_\epsilon)$ be the kernels of $R$ and $nf_\epsilon$, respectively. Then $D_e^*/kn(R) = D_e^*/kn(nf_\epsilon)$.

**Proof.** The results hold by commutation of Figure 5 that is granted by the previous results, in particular by Corollary 35 and Theorem 56 and the commutation of the triangles relative to the initial and final states.

Theorem 58 relates our finite automata with the infinite states ones obtained via Brzozowski’s derivatives before quotienting the automata states by means of ACI to make them finite. The following easy lemma shows that $kn(nf_\epsilon)$ is an equivalence relation finer than ACI and thus Theorem 58 also holds for the standard finite Brzozowski’s automata since we can quotient with ACI first.

**Lemma 59.** Let $e_1$ and $e_2$ be regular expressions. If $e_1 =\text{ACI} e_2$ then $nf_\epsilon (e_1) = nf_\epsilon (e_2)$.

5. **Merging**

By Theorem 16 $L_p (\cdot) = L_p (\cdot) \cup L(|e|)$. A more syntactic way to look at this result is to observe that $\bullet (e)$ can be obtained by “merging” together the points in $\epsilon$ and $\bullet (|e|)$, and that the language defined by merging two pointed expressions $e_1$ and $e_2$ is just the union of the two languages $L_p (e_1)$ and $L_p (e_2)$. The merging operation, that we shall denote with $\bullet$, does also provide the relation between deterministic and nondeterministic automata where, as in Watson [10][11], we may label states with expressions with a single point (for lack of space, we shall not explicitly address the latter issue in this paper, that is however a simple consequence of Theorem 57). Finally, the merging operation will allow us to explain why the technique of pointed expressions cannot be (naively) generalized to intersection and complement (see Section 5.1).

**Definition 60.** Let $e_1$ and $e_2$ be two items on the same carrier $|e|$. The merge of $e_1$ and $e_2$ is defined by the following rules by recursion over the structure of $e$:
\[ \emptyset \uparrow \emptyset = \emptyset \]
\[ \epsilon \uparrow \epsilon = \epsilon \]
\[ \epsilon \uparrow a = a \]
\[ a \uparrow \epsilon = a \]
\[ a \uparrow a = a \uparrow a \]
\[ \bullet \uparrow a = a \uparrow a \]
\[ a \uparrow \bullet a = a \uparrow a \]
\[ (e_1 + e_2) \uparrow (e_1 + e_2) = (e_1 \uparrow e_2) + (e_2 \uparrow e_1) \]
\[ (e_1e_2) \uparrow (e_1e_2) = (e_1 \uparrow e_2)(e_1 \uparrow e_2) \]
\[ e_1 \uparrow e_2 = e_1 \uparrow e_2 \]

The definition is extended to pres as follows:
\[ (e_1, b_1) \uparrow (e_2, b_2) = (e_1 \uparrow e_2, b_1 \vee b_2) \]

**Theorem 61.** $\uparrow$ is commutative, associative and idempotent

**Proof.** Trivial by induction over the structure of the carrier of the arguments.

**Theorem 62.** $L_p (e_1 \uparrow e_2) \subseteq L_p (e_1) \cup L_p (e_2)$

**Proof.** Trivial by induction on the common carrier of the items of $e_1$ and $e_2$.

All the constructions we presented so far commute with the merge operation. Since merging essentially corresponds to the subset construction over automata, the following theorems constitute the proof of correctness of the subset construction.
THEOREM 63. \((e_1^1 + e_1^2) \oplus (e_2^1 + e_2^2) = (e_1^1 + e_2^1) \oplus (e_1^2 + e_2^2)\)

Proof: Trivial by expansion of definitions.

THEOREM 64.

1. for \(e_1\) and \(e_2\) items on the same carrier,
   \[\bullet(e_1 \uplus e_2) = \bullet(e_1) \uplus \{e_2, \text{false}\}\]
2. for \(e_1\) and \(e_2\) pres on the same carrier,
   \[\bullet(e_1 \uplus e_2) = \bullet(e_1) \uplus e_2\]
3. \((e_1^1 + e_1^2) \odot (e_2^1 + e_2^2) = (e_1^1 \odot e_2^1) \uplus (e_1^2 \odot e_2^2)\)

COROLLARY 65.
\[\bullet(e_1 \uplus e_2) = e_1 \uplus \bullet(e_2) = \bullet(e_1) \uplus \bullet(e_2)\]

Proof of the corollary The corollary is a simple consequence of commutativity of \(\uplus\) and idempotence of \(\bullet\):
\[
\bullet(e_1 \uplus e_2) = \bullet(e_1 \uplus e_2) = \bullet(e_1) \uplus \bullet(e_2) = \bullet(e_1) \uplus \bullet(e_2)
\]

Proof of 1. We first prove \(\bullet(e_1 \uplus e_2) = \bullet(e_1) \uplus \{e_2, \text{false}\}\) by induction over the structure of the common carrier of \(e_1\) and \(e_2\), assuming that 3. holds on terms whose carrier is structurally smaller than \(e\).

- If \(e_1 = \emptyset\), \(e, \alpha\) then trivial
- If \(e_1 = e_1^1 + e_1^2\) and \(e_2 = e_2^1 + e_2^2\):
  \[
  \bullet((e_1^1 + e_1^2) \uplus (e_2^1 + e_2^2)) = \bullet(e_1) \uplus (e_2^1 + e_2^2) = (e_1^1 \uplus e_2^1) \odot (e_1^2 \uplus e_2^2)
  \]

- If \(e_1 = e_1^1\) and \(e_2 = e_2^1\) then, using 3. on items whose carrier is structurally smaller than \(e_1\),
  \[
  \bullet((e_1^1 + e_1^2) \uplus (e_2^1 + e_2^2)) = \bullet(e_1^1 \uplus e_2^1) \odot (e_1^2 + e_2^2)
  \]

Thus
\[
(e_1^1 \uplus e_2^1) \odot (e_1^2 \uplus e_2^2) = (e_1^1 \uplus e_1^2 \odot e_2^1 \odot e_2^2) = (e_1^1 \uplus e_1^2 \odot e_2^1 \odot e_2^2)
\]

THEOREM 66. \((e_1 \uplus e_2) \ast = e_1 \uplus e_2\)

Proof. Let \(e_1 = (e_1^1, b_1)\) and \(e_2 = (e_2^1, b_2)\). Thus
\[
\langle (e_1^1, b_1 \uplus e_2^1, b_2) \rangle = \langle (e_1^1, b_1) \uplus (e_2^1, b_2) \rangle
\]

Let define \(e', e_1^1\) and \(e_2^2\) by cases on \(b_1\) and \(b_2\) with the property that \(e' = e_1 \uplus e_2^2\):

- If \(b_1 = b_2 = \text{false}\) then let \(e' = e_1^1\) and \(e' = e_1^1 \uplus e_2^2\). Obviously \(e' = e_1^1 \uplus e_2^2\).
- If \(b_1 = \text{true}\) and \(b_2 = \text{false}\) then let \(e' = e_1^1\) and \(e' = e_1^1 \uplus e_2^2\). Let \(e_1^1 = e_1^1\) and let \(\bullet(e_1^1) \uplus e_1^2 = \bullet(e_1) \ast = \langle e', b' \rangle\).

In all cases,
\[
\langle (e_1^1 \uplus e_2^1, b_1 \uplus b_2) \rangle = \langle (e_1^1 \uplus e_2^2, b_1 \uplus b_2) \rangle = (e_1^1, b_1) \ast = (e_1^1 \uplus e_2^2, b_1 \uplus b_2)
\]

THEOREM 67. \(\text{move}(e_1 \uplus e_2, a) = \text{move}(e_1, a) \uplus \text{move}(e_2, a)\)

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

- the cases \(\emptyset, b \neq a\) are trivial by computation
- the case \(a\) has four sub-cases: if \(e_1\) and \(e_2\) are both \(a\), then \(\text{move}(a \uplus a, a) = (\emptyset, \text{false}) = \text{move}(a, a) \uplus \text{move}(a, a)\), otherwise at least one in \(e_1\) or \(e_2\) is \(\neq a\) and \(\text{move}(e_1 \uplus e_2, a) = \text{move}(e_1, a) \uplus \text{move}(e_2, a)\)
  - if \(e = e_1 \uplus e_2\) then
    \[
    \text{move}((e_1 \uplus e_2), a) = \text{move}(e_1 \uplus e_2, a) \uplus \text{move}(e_1, a) \uplus \text{move}(e_2, a)
    \]
  - if \(e = e_1 \uplus e_2\) then
    \[
    \text{move}((e_1 \uplus e_2), a) = \text{move}(e_1 \uplus e_2, a) \uplus \text{move}(e_1, a) \uplus \text{move}(e_2, a)
    \]

5.1 Intersection and complement

Pointed expressions cannot be generalized in a trivial way to the operations of intersection and complement. Suppose to extend the definition of the language in the obvious way, letting \(L_{\mu}(e_1 \cap e_2) =\)
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.

As explained in the introduction, the notion of pointed expression was suggested by an attempt of formalizing the theory of regular languages by means of an interactive prover. This testify the relevance of the choice of good data structures not just for the design of algorithms but also for the formal investigation of a given field, and is a nice example of the kind of interesting feedback one may expect by the interplay with automated devices for proof development.

References

[1] G. Rozenberg and A. Salomaa, eds., Handbook of formal languages, vol. 1: word, language, grammar. New York, NY, USA: Springer-Verlag New York Inc., 1997.

[2] K. E. B. Krawetz, J. Shallit, and M. Wei Wang, “Regular expressions: New results and open problems,” Journal of Automata, Languages and Combinatorics, vol. 10, no. 4, pp. 407–437, 2005.

[3] J. A. Brzozowski, “Derivatives of regular expressions,” J. ACM, vol. 11, no. 4, pp. 481–494, 1964.

[4] R. McNaughton and H. Yamada, “Regular expressions and state graphs for automata,” IEEE Transactions on Electronic Computers, vol. 9, no. 1, pp. 39–47, 1960.

[5] S. Owens, J. H. Reppy, and A. Turon, “Regular-expression derivatives re-examined,” J. Funct. Program., vol. 19, no. 2, pp. 173–190, 2009.

[6] G. Berry and R. Sethi, “From regular expressions to deterministic automata,” Theor. Comput. Sci., vol. 48, no. 3, pp. 117–126, 1986.

[7] A. Brüggemann-Klein, “Regular expressions into finite automata,” Theor. Comput. Sci., vol. 120, no. 2, pp. 197–213, 1993.

[8] C.-H. Chang and R. 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, vol. 644 of Lecture Notes in Computer Science, pp. 90–110, Springer, 1992.

[9] S. C. Kleene, “Representation of events in nerve nets and finite automata,” in Automata Studies (C. E. Shannon and J. McCarthy, eds.), pp. 3–42, Princeton University Press, 1956.

[10] B. W. Watson, “A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata,” South African Computer Journal, vol. 27, pp. 12–17, 2001.

[11] B. W. Watson, “Directly constructing minimal dfas : combining two algorithms by brzozowski,” South African Computer Journal, vol. 29, pp. 17–23, 2002.

[12] W. Gelade, “ Succinctness of regular expressions with interleaving, intersection and counting,” Theor. Comput. Sci., vol. 411, no. 31-33, pp. 2987–2998, 2010.

[13] H. Gruber and M. Holzer, “Finite automata, digraph connectivity, and regular expression size,” in ICALP, vol. 5126 of Lecture Notes in Computer Science, pp. 39–50, Springer, 2008.

[14] M. Holzer and M. Kutrib, “Nondeterministic finite automata - recent results on the descriptive and computational complexity,” Int. J. Found. Comput. Sci., vol. 20, no. 4, pp. 563–580, 2009.

[15] A. R. Meyer and L. J. Stockmeyer, “The equivalence problem for regular expressions with squaring requires exponential space,” in 13th Annual Symposium on Switching and Automata Theory (FOCS), pp. 125–129, IEEE, 1972.

[16] L. C. Eggan, “Transition graphs and the star-height of regular events,” Michigan Mathematical Journal, vol. 10, no. 4, pp. 385–397, 1963.