Computable partial derivatives of regular expressions

Vladimir Komendantsky[1]

School of Computer Science
University of St Andrews
St Andrews, KY16 9SX, UK
vk10@st-andrews.ac.uk

Abstract. A declarative approach to theorem prover formalisation of partial derivatives of regular expressions is presented. Partial derivatives of regular expressions are a computation trace of (total) Brzozowski derivatives. Namely, partial derivatives can be viewed as states of a non-deterministic automaton recognising the language of a regular expression, while Brzozowski derivatives are viewed as states of a deterministic automaton. Common decision procedures on regular expressions compute Brzozowski derivatives and involve determinisation of an intermediate non-deterministic automaton structure, a step that is avoided if partial derivatives are computed instead. Therefore we obtain a more straightforward method to implement decision procedures on regular languages in a provably-correct way in Coq.

1 Introduction

The most widely employed form of regular expression derivatives is due to Brzozowski [5]. Their straightforward application is to deterministic finite automaton construction and termination of regular expression matching. Matching a word w against a regular expression E reduces to construction of an equivalent regular expression in normal form, the word derivative of E with respect to w, and checking whether the obtained word derivative is *nullable*, that is, matches the empty word. Since, for every finite word, the process of obtaining the word derivative with respect to a regular expression is terminating, and nullability is decidable by a straightforward recursive algirithm, Brzozowski derivatives give a simple and effective notion of regular expression matching.

Partial derivatives of regular expressions were introduced by Mirkin [16] in an elegant declarative style. The set of partial derivatives of E forms a *prebase* which is an object of inductive type. The notion of prebase is a non-deterministic generalisation of the notion of *base* [5], that is, the set of word derivatives of E. Having a type of prebase is quite remarkable is we look at it from the point of view of theorem proving because it provides us with sufficient structure, a *spec*, to guide through inductive proofs. An equivalent definition where derivatives form a kind of computational family with the inductive structure considered as

meta-data would require recovering this inductive structure for efficient theorem proving. Such an algorithmic definition does exist in the form of partial derivatives of Antimirov [2], an independently discovered notion. In fact, it was also considered by Mirkin much earlier in another paper, [15], where the author described an algorithm computing the same results as the one by Antimirov. The two approaches, declarative and algorithmic, can be seen as two sides of the same coin: The former uses induction to define inhabitants of an inductive type and therefore lends itself to formalisation in a theorem prover, while the latter uses recursion to define inhabitants of a computational family, and so, is a natural functional programming approach.

Contribution. By building a structured and extendable development of partial derivatives of regular languages in Coq we contribute to a thorough understanding of the declarative side of partial derivatives and thus complement the recent formal proofs that follow the algorithmic paradigm and use inductive families, [1]. Moreover, in contrast with [1], our basic language membership relation is decidable, which facilitates proof development by computational reflection. Although the paper does not contain novel theoretic results, formalisation of the partial derivative construction is a non-trivial result in its own right. It can be used in further research, for example, as computational content for decision procedures for Kleene algebras in Coq.

Motivation. A decision procedure for Kleene algebra statements in Coq should provide effective, accessible and provably correct notions of regular expression containment and equivalence proof search. Practical applications of such notions can include correctness proofs for XML DTD compression algorithms or even static XML type inference.

Outline. Below in this section we state related work, especially in the field of interactive theorem proving. In Section 2, we provide a brief introduction to defined constructs that we require from Ssreflect libraries. In Section 3, definitions of regular languages and regualr expressions are given. Since we do not rely on an axiom of extensional equality of languages, we have to deal with extensional constructs using appropriate equivalence notions. Such two notions are discussed in Section 4. The main part of the formalisation, including the Main Theorem, is contained in Section 5. From the partial derivative construction we can obtain the construction of Brzozowski derivatives, using the method illustrated in Section 6. Topics for future work are discussed in Section 7. We give our conclusions in Section 8.

Related work. In [3], a related technique of *pointed regular expressions* for constructing a deterministic finite automaton from a regular expression was discussed and a development in Matita was presented. The authors rely on the axiom of decidability of extensional equality of regular languages. Pointed regular expressions are an application of Brzozowski derivative construction to McNaughton and Yamada's labelling technique. The latter is traditionally used in

determinisation of non-deterministic finite automata. The relationship to our work is that we use (a matrix representation of) non-deterministic automata directly, without their determinisation, having applied the corresponding partial derivative construction instead of a total one. Nevertheless, our development does not require an axiom of extensional equality of languages because operations of regular languages are computable in Coq, and hence regular languages enjoy decidable *equivalence*.

Another recent study [4] introduced a reflexive Coq tactic for deciding Kleene algebras. Regular languages are the initial model of Kleene algebras [13]. Binary relations are another significant model. The authors of [4] achieve a broad goal of solving equations in any given Kleene algebra using the initiality of the standard language model. In comparison, our computable specification could also be used for deciding Kleene algebras, right in Coq. Also, we share a similar matrix-based approach to representing automata. In contrast with [4], we use neither the tactic language of Coq (which provides access to the internal representation of terms in Coq) nor the Type Classes enrichment of type theory of Coq to represent hierarchies of mathematical structures. The former is not required in our development because we do not pattern-match on internal representation; and the latter is replaced by the packed class methodology of Ssreflect that does not require extensions of type theory.

Applications of Brzozowski derivatives include terminating algorithms for parsing [7] and subtyping [10, 11]. The aspect of derivatives is exploited which is related to a notion of proof search by delayed applications of the non-deterministic sum.

Partial derivatives find their application in the same areas as Brzozowski derivatives, for example, Antimirov derivatives in regular expression matching [17], with the additional benefit being the linear upper bound on the number of partial derivatives — a dramatic improvement compared to word derivatives. A comparison of the two kinds of partial derivative (Mirkin derivative and Antimirov derivative) was performed in [6], where these two are virtually identified. However, as we mentioned earlier, the important difference — which comes into view when we program in a theorem prover — is implicit in different style of definitions.

2 Library definitons

In this section we introduce required notions from Ssreflect libraries [9]. These libraries do not change the type theory of Coq or add new axioms. A characteristic feature is a systematic use of dependent types and unification hints (Implicit Coercions and Canonical Structures) implemented in Coq, which allows to construct hierarchies of types of mathematical structure [8].

Probably the most important type in our development is the type 'I_n of natural number bounded by n:

```
Inductive ordinal (n : nat) : Type :=
  Ordinal : ∀ m : nat, m < n → 'I_n
```

Therefore a bounded number, and *ordinal*, in Ssreflect terminology, is a natural number together with a proof that it is less than the given upper bound.

Programming as well as theorem proving with Ssreflect has the advantage that there are many predefined library functions on primitive structures such as lists (called seq in Ssreflect, merely for disambiguation with the main libraries of Coq), together with proofs of their properties. For working with formal languages there are indespensable functions such as take n s, the list containing only the first n items of the given list s, or the list s if size s $\leq$ n. The dual function is drop n s which computes the list s less its first n items, or the empty list if size s $\leq$ n.

Ssreflect features two related kinds of predicate. The type pred T of *applicative predicate* on a given type T is simply an alias for the function type T $\rightarrow$ bool. If P is an applicative predicate, the proposition "the expression e satisfies P" can be written applicatively, as P e. On the other hand, Ssreflect allows to define *collective predicates* as instances of a type of generic predicate predType. If P is a collective predicate, the proposition "e satisfies P" is collectively written as e $\in$ P. The library maintains the prefix and infix style of notation for these two kinds of predicate respectively.

In Ssreflect, finite objects can be defined in a canonical way as inhabitants of the type finType. The set of symbols of a regular expression is finite, and so, we can define the type of *symbol* to be a finType. For i an element of the ordinal 'I_#|symbol| (the type of natural numbers bounded by the size of symbol), we also define the i-th symbol in the canonical enumeration of symbol using the library function enum_val simply as enum_val i.

Remark 1. Both kinds of predicate are decidable, namely, bool-valued. This ensures that predicate definitions are computable in Coq by the internal reduction and type inference, as opposed to the requirement to prove, by hand or by tactic, that a general deductive Prop-valued predicate holds for every given object in the domain. This gives us a fair advantage of deciding language containments by simplification and rewriting as opposed to induction and inductive inversion.

Iterated operators, that arise naturally in enumerative computation, are introduced by notation

```
\big[op/idx]_(i < n) F
```

This provides a generic definition for iterating an operator over a set of indices, implicitly parametrised by the type of indices, the operator op, the initial value idx applied when the set of indices is empty and the expression F we are iterating.

3 Definition of regular expressions and languages

Now we can use basic notions introduced in Section 2 to define regular languages and regular expressions. Like in Coq, we do not display implicit arguments that can be inferred by the typechecker of Coq.

The type of regular expression is the simple inductive type below:

```
Inductive re :=
  | Void | Eps | Atom : symbol → re
  | Alt : re → re → re | Conc : re → re → re | Star : re → re.
```

The six possible forms of regular expression are, respectively: Void denoting the empty language; Eps denoting the language of the empty string; Atom a denoting the language of the string a; Alt E F and Conc E F denoting the union and concatenation of the languages of E and F respectively; and Star E denoting the iteration language of E.

Remark 2. It can be also possible to define a *dependently-typed* regular expression by parametrising each kind of regular expression with the language it denotes. However, in that case, some advanced dependent inversion techniques would be required to relate languages denoted by the components of a compound regular expression to its own language. Since inversion is a deductive method, and we work in the alternative, computational paradigm, we relate regular expressions to their languages using reflection lemmas rather than inversion.

Now we will use notions from Section 2 for programming computable regular languages, that is, we will define several decidable predicates on words that form the complete functional basis of regular languages. First, we define a word as a list of symbols, a seq, in Ssreflect terminology, and supply it with a canonical instance of the type of types with decidable equality:

```
Definition word := seq symbol.
Canonical Structure word_eqType := [eqType of word].
```

We also define languages to be simple applicative predicates using the abbreviation pred:

```
Definition language := pred word.
Definition void : language := pred0.
Definition eps : language := pred1 [::].
Definition atom a : language := pred1 [:: a].
Definition alt L1 L2 : language := [predU L1 & L2].
Definition conc L1 L2 : language :=
  fun w ⇒ existsb i : 'I_(size w).+1, L1 (take i w) && L2 (drop i w).
Definition star L : language :=
  fix star w := if w is a :: u then conc (residual a L) star u else true.
```

We will explain language definitions in detail. The void language is a predicate that is always false. The empty string language eps is a singleton predicate that is true only on the empty list of symbols, denoted [::] in Ssreflect. The language of a symbol a is another singleton predicate that is true only on the one-symbol word [:: a]. The alternation of two languages is the strainghtforward union of predicates. The concatenation of two languages is defined by the notation existsb x : T , F that denotes negb (#|(fun x : T ⇒ F)| == 0), namely, the computable boolean statement that the size of fun x : T ⇒ F *considered as a finite function* is not equal to 0. Finally, the iteration of a language L is defined by removing

the first symbol a from the word w, residuation of the given language L with respect to a and concatenating the result with the iteration landuage taken as a thunk; in the case when the word w is empty, it can be trivially iterated, and so, the predicate is true on the empty word. Since we remove a symbol on each application of star, the function passes the termination check.

The nullability test mentioned in Introduction is performed by the following function:

```
Fixpoint is_output (e : re) :=
  match e with
    | Void | Atom _ ⇒ false
    | Eps | Star _ ⇒ true
    | Alt e1 e2 ⇒ is_output e1 || is_output e2
    | Conc e1 e2 ⇒ is_output e1 && is_output e2
  end.
```

The nullability test has a direct relationship to automaton representation of a regular expression. In fact, a nullable regular expression is such that its recognising automaton halts on the empty string. The function output below computes the regular expression denoting this fact:

```
Definition output e := if is_output e is true then Eps else Void.
```

The language semantics of regular expressions is computed by the function lang that recursively maps constructors of re to their language counterparts.

```
Fixpoint lang e :=
  match e with
  | Void ⇒ void
  | Eps ⇒ eps
  | Atom x ⇒ atom x
  | Alt e1 e2 ⇒ alt (lang e1) (lang e2)
  | Conc e1 e2 ⇒ conc (lang e1) (lang e2)
  | Star e1 ⇒ star (lang e1)
  end.
```

As explained in the following section, the function lang is instrumental in the canonical interpretation of regular expressions as sets of words.

4 Extensional equalities

It is easy to see that given two languages in the sense of Section 3, being functions, cannot be effectively tested for equality but only equivalence. Namely, two languages are said to be equivalent if they contain the same words. This can be captured using two different relations in Ssreflect. The first possibility is the extensional equality of functions denoted $=_1$. For example, we consider the extensional additive monoid structure on regular languages. That is, we prove the following lemmas containing statements of monoidal laws up to extensional equality:

```
Lemma altA : ∀ L M N, alt L (alt M N) =1 alt (alt L M) N.
Lemma altIl : ∀ L, alt void L =1 L.
Lemma altIr : ∀ L, alt L void =1 L.
```

Similarly, as part of the extensional multiplicative structure, we can prove the following theorem:

```
Lemma conclr : ∀ L, conc L eps =1 L.
```

The second possibility is to use the set-based equality denoted $=_i$. This is required for regular expressions considered as the sets of words they denote. The implementation of this involves an implicit coercion from regular expressions to languages which is in fact the function lang. It also involves a specifying a canonical instance of predicates on words on the type of regular expression. Thus, the semantic equivalence of the languages a pair of regular expressions can be defined as follows, where the first line defines the canonical instance of collective predicates on regular expressions.

```
Canonical Structure re_predType := mkPredType lang.
Definition equire (e1 e2 : re) : Prop := e1 =i e2.
```

5 Partial derivatives

Now we come to the main structure of the paper, the type of prebase for a given regular expression. We design a specification in which we capture the required parameters and properties of a prebase:

```
Record prebase (E0 : re) : Type := Prebase {
  pN : nat;
  pP :> (1 + pN).-tuple re;
  pM : matrix (seq re) (1 + pN) #|symbol|;
  _ : tnth pP ord0 = E0;
  _ : ∀ i j (E : re), E ∈ pM i j → E ∈ behead_tuple pP;
  _ : ∀ i,
          lang (tnth pP i) =1
          (\big[ alt / lang (output (tnth pP i)) ]_(j < #|symbol|)
                \big[ alt / void ]_(k < size (pM i j))
                    conc (atom (enum_val j)) (lang (nth Void (pM i j) k)))
}.
```

Therefore a prebase of E0 is a 6-tuple consisting of a natural number pN, a tuple of length pN containing regular expressions and a rectangular matrix of size 1 + pN by #|symbol| whose cells are sequences of regular expressions, all satisfying the three properties:

1. The 0-th element of the prebase tuple is the given regular expression E0.
2. Every cell in the prebase matrix is a subset of the tail of the prebase tuple, that is, the tuple with the 0-th element being removed.

3. For every i-th element of the prebase tuple, the corresponding language equivalence holds, which can be written in mathematical notation, assuming that the number of elements in the finite type symbol is m:

$$[\![\mathsf{E}_i]\!] =_1 \bigoplus_{j<m} \Big(\bigoplus_{k<\mathsf{size}(\mathsf{pM}_{i,j})} \big(a_j \cdot [\![(\mathsf{pM}_{i,j})_k\ ;\ 0]\!]\ ;\ [\![0]\!]\big) \Big)\ ;\ [\![\mathsf{output}(\mathsf{E}_i)]\!] \qquad (1)$$

The l.h.s. of the equivalence (1) contains the language denoted by E_i. The r.h.s. contains the union of languages obtained by finite iteration of the operation alt on the expression before the semicolon and, when the list of indices empties, applying the language written after the semicolon. This is a particular case of notation \big[op/idx]_(i < n)F for finite iterative operators we discussed in Section 2.

Our goal now is to define the prebase for a given regular expression E. For this, we formalise the mathematical proof of the theorem of Mirkin. In mathematical notation, the theorem is stated below. The constructed prebase is denoted $[E]$.

Main Theorem (Mirkin [16]). For any given regular expression E, we can construct a prebase $[E]$ whose tuple consists of regular expressions $E_0, \ldots, E_n$ such that

- $E = E_0$;
- cells of the matrix of $[E]$ may only contain elements of $\{E_1, \ldots, E_n\}$;
- the corresponding language equivalence (1) holds for each element of the prebase tuple.

We skip the mathematical proof and let the reader refer to the formal proofs supplied with the paper [12].

We produce the formal proof of the Main Theorem by structural induction on E:

```
Theorem mirkin_prebase : ∀ E : re, prebase E.
```

For the proof it is required to construct the prebase tuple and the prebase matrix for each possible kind of regular expression, and prove the corresponding three properties.

6 Connection with Brzozowski derivatives

Using partial derivatives, a representation of Brzozowski derivatives can be obtained in the following way. Suppose we are given a symbol a whose index in the canonical enumeration of the alphabet is j. Then the following sum is a representation of Brzozowski derivative of a regular expression E with respect to the symbol a:

$$E_1 + \cdots + E_n$$

where $E_1, \ldots, E_n$ are elements of the cell $(0, j)$ in the computed matrix of the prebase of E.

We can define the above sum formally:

```
Definition der E j := foldr Alt Void (pM (mirkin_prebase E) ord0 j).
```

The reflexive and transitive closure of der is the well-known Brzozowski derivative of a regular expression with respect to a word, defined as follows:

```
Fixpoint mem_der E w :=
  if w is a :: u
  then mem_der (der E (enum_rank a)) u
  else is_output E.
```

The following theorem establishes the formal connection between computation of the Brzozowski derivative and decidability of regular language membership testing:

```
Theorem mem_derE : ∀ w E, mem_der E w = (w ∈ lang E).
```

This statement is proved by structural induction on w. The base case is straightforward. For the induction step, we prove the following lemma:

```
Lemma derE : ∀ j E, der E j =i residual (enum_val j) (mem E).
```

Remark 3. Decidability of language membership is a straightforward application of Brzozowski derivatives. However, direct application of partial derivatives is also possible if we do not restrict derivative search to the row 0 of the prebase matrix but use the whole matrix instead.

7 Discussion and further directions

In addition to the three properties listed in the type of prebase in Section 5, the paper [16] added one more that characterises the upper bound on the size of the prebase tuple: For a given regular expression E, the size of its prebase tuple is less than or equal to the number of distinct symbols in E plus 1. We have not yet proved this property formally. An Ssreflect proof is possible that should use the defined operations on finite sets and established properties of those.

In the present state of program extraction in Coq, functional programs defined in Ssreflect cannot be natually extracted to target programming languages. The current limitation of extraction does not allow to treat anonymous record fields in modules consistently. This limitation unfortunately includes the most generic type eqType of types with decidable equality. The limitation can be removed still. Hopefully, this will be done in future releases of Coq. As for now, it is quite impossible to extract computational content of our proofs in order to use it in a target programming language.

It can further be investigated by extending this development how prebases generalise to the case of regular types (expressions whose semantics is finite and infinite regular trees rather than regular languages) [14], and whether their use can help to define efficient subtyping algorithms either for regular expressions [10] or for regular types. There is a stimulating opportunity of applying proof enumeration methods in subtyping by using partial derivatives, which is stronger than proof search that total derivatives are fitted for. Using proof enumeration can enable us to find most efficient proofs rather than a single canonical proof.

8 Conclusions

We define the structure of partial derivatives (prebase) of a regular expression using Coq and Ssreflect libraries. This project allows to apply the expressive power of Coq to building computational decision procedures for formal languages. At present we can envisage a number of possible applications of partial derivatives, including proof search for regular expression containment and decision procedures for Kleene algebras. Further research is possible on how our development can be further extended to the more general case of regular types.

At the time of submission we have to admit several technical lemmas on the inductive step for the proof of the last property of prebase, the language equivalence. Learning a new proof library is a research project of its own, and as such, does take time. The positive circumstance is that the critical parts of the development are now complete and the question now is to simply find the right lemmas to converge the proofs that should converge. The current state of the development in Coq 8.3pl2 with Ssreflect 1.3pl1 is accessible from the author's webpage [12].

Acknowledgements. The author is grateful to Georges Gonthier and Prof. Boris Mirkin. This research is supported by the research fellowship EU FP7 Marie Curie IEF 253162 'SImPL'.

References

1. J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Implementation and Application of Automata 2010*, volume 6482/2011 of *Lecture Notes in Computer Science*, pages 59–68, 2011.
2. V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
3. A. Asperti, C. Sacerdoti Coen, and E. Tassi. Regular expressions, au point, 2010. Draft submitted to arXiv.org and available at http://arxiv.org/abs/1010.2604.
4. T. Braibant and D. Pous. Deciding Kleene algebras in Coq, 2011. Submitted.
5. J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
6. J.-M. Champarnaud and D. Ziadi. From Mirkin's prebases to Antimirov's word partial derivatives. *Fundam. Inf.*, 45:195–205, January 2001.
7. N. A. Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 285–296, New York, NY, USA, 2010. ACM.
8. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics (2009)*, volume 5674 of *LNCS*, 2009.
9. G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2011.
10. F. Henglein and L. Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). Technical Report 612, Department of Computer Science, University of Copenhagen (DIKU), February 2010.

11. F. Henglein and L. Nielsen. Regular expression containment: Coinductive axiomatization and computational interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, January 2011.
12. V. Komendantsky. Formal proofs of the prebase theorem of Mirkin, 2011. Coq script available at http://www.cs.st-andrews.ac.uk/~vk/doc/mir.v.
13. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, 1994.
14. C. McBride. Clowns to the left of jokers to the right (Pearl): Dissecting data structures. *SIGPLAN Not.*, 43:287–295, January 2008.
15. B. G. Mirkin. External algorithm for construction of base for the language of regular expressions. In M. A. Bogomolov and B. V. Korobov, editors, *Computational methods and programming for computers Ural-2 and Ural-4*, pages 161–166. Saratov University Press, Saratov, Russia, 1966.
16. B. G. Mirkin. New algorithm for construction of base in the language of regular expressions. *Tekhnicheskaya Kibernetika*, 5:113–119, 1966. English translation in *Engineering Cybernetics*, No. 5, Sept.–Oct. 1966, pp. 110-116.
17. M. Sulzmann and K. Z. M. Lu. Regular expression matching using partial derivatives, 2010. Draft.

9 Mathematical commentary to Section 5

We will define tuples and matrices of a prebase of a given regular expression in this Appendix. This is a mathematical commentary to Section 5 and the formal Coq development. As it is common for theorem proving in a proof assistant, in our development, precision of definitions had to be improved with respect to [16] to resolve certain ambiguity of mathematical notation. Therefore the theorem statement and proof outline below serve for indicative purposes only. The reader is advised to refer to [12] for exact definitions.

We let the *alphabet* of symbols be a finite ordered set $A = \{a_0, \ldots, a_{n-1}\}$, for some n. *Regular expressions* are generated by the following grammar:

$$E, F ::= 0 \mid 1 \mid a_i \mid E + F \mid E \times F \mid E^*$$

Let $||E||$ denote the number of distinct alphabet symbols in the regular expression E, and let $o(E)$ be 1 if E is nullable and 0 otherwise.

For some positive natural number m, an m-tuple of regular expressions $P = \{E_0, \ldots, E_{m-1}\}$ is called a *prebase* if the following semantic language equivalences hold for $i \in [0, m-1]$ and $j \in [0, n-1]$:

$$E_i \simeq a_0 \times \big(\sum M_{i,0}\big) + \cdots + a_{n-1} \times \big(\sum M_{i,n-1}\big) + o(E_i) \tag{2}$$

where M_{ij} is a finite subset of P, that is,

$$M_{i,j} = \bigcup_{k \in I_{ij}} E_k, \quad I_{ij} \subseteq [0, m-1]$$

and the regular expression denoted $\sum M_{i,j}$ is the sum of elements of $M_{i,j}$ obtained by folding (left or right) by $+$ with the initial value 0. The result of the

fold is 0 if and only if $M_{i,j}$ is empty. These individual sets $M_{i,j}$ will be viewed by us as cells of the *matrix* $[M]_{i,j}$ *of the prebase* P.

We remark that the *base* B of E, that is, the set of all Brzozowski word derivatives of E, is a special case of prebase, the one where each of $M_{i,j}$ in (2) is a singleton set containing a regular expression from the set B.

The order-aware definition of prebase gives a certain amount freedom to define an algorithm computing prebases for a given regular expression. Mirkin [16] developed a generic algorithm by structural induction on the regular expression. Therefore we state the result as a theorem where prebases and their corresponding matrices are parametrised by a regular expression:

Main Theorem (Mirkin [16]). For any given regular expression E, we can construct a prebase $P(E)$ such that

- $E \in P(E)$, namely, $E = E_0$;
- cells of the matrix of $P(E)$ may only contain elements of $\{E_1, \ldots, E_n\}$;
- and $|P(E)| \leq ||E|| + 1$.

Proof (Outline). By induction on E, we define:
Basis.

1. $P(0) = \{0\}$ and $M(0) = \lambda\ i\ j.\ \emptyset$.
2. $P(1) = \{1\}$ and $M(1) = \lambda\ i\ j.\ \emptyset$.
3. $P(a_k) = \{a_k, 1\}$ and $M(a_k) = \lambda\ i\ j.\ \begin{cases} \text{if } i = 0 \text{ and } j = k \text{ then } \{1\} \\ \text{else } \emptyset. \end{cases}$

Induction step. We will use vertical concatenation $\boxdot$ of two matrices.

1. $P(E + F) = \{E_0 + F_0, E_1, \ldots, E_{|P(E)|-1}, F_1, \ldots, F_{|P(F)|-1}\}$.
$$M(E + F) = \left[M(E)_{0,j} \cup M(F)_{0,j}\right]_{0,j} \boxdot \left[M(E)\right]_{0<i,j} \boxdot \left[M(F)\right]_{0<i,j}$$
2. $P(E{\times}F) = \{E_0{\times}F_0, E_1{\times}F_0, \ldots, E_{|P(E)|-1}{\times}F_0, F_1, \ldots, F_{|P(F)|-1}\}$.
$$M(E{\times}F) = \begin{array}{c} \left[\left(\left(\sum M(E)_{i,j}\right){\times}F_0\right) \cup \left(o(E_i){\times}\left(\sum M(F)_{0,j}\right)\right)\right]_{i,j} \\ \boxdot \\ \left[M(F)\right]_{0<i,j} \end{array}$$
3. $P(E^*) = \{E_0^*, E_1{\times}E_0^*, \ldots, E_{|P(E)|-1}{\times}E_0^*\}$ where E_0^* should be read as $(E_0)^*$.
$$M(E^*) = \begin{array}{c} \left[\left(\sum M(E)_{0,j}\right){\times}E_0^*\right]_{0,j} \\ \boxdot \\ \left[\left(\left(\sum M(E)_{i,j}\right){\times}E_0^*\right) \cup \left(o(E_i){\times}\left(\sum M(E)_{0,j}\right){\times}E_0^*\right)\right]_{0<i,j} \end{array}$$

□

Remark 4. Although the prebase matrix is quite natural to consider, it did not appear in the original papers [16, 15]. Meanwhile, by manifesting the matrix structure to Ssreflect we can reduce low-level clutter in proofs.