Reflexive Toolbox for Regular Expression Matching

Verification of Functional Programs in Coq+Ssreflect

Vladimir Komendantsky

School of Computer Science
University of St Andrews
St Andrews, KY16 9SX, UK
http://www.st-andrews.ac.uk/~vk/

Abstract

We study a derivative method allowing to prove termination of computations on regular expressions. A Coq formalisation of a canonical non-deterministic finite automaton construction on a regular expression is presented. The correctness of the functional definitions is formally verified in Coq using the libraries and the small-scale reflection tools of Ssreflect. We propose to extend the proofs further, and this is a work in progress, to study termination of containment and equivalence in terms of partial derivatives. This serves as a major motivation and intended application of the presented approach. A method that we develop in the paper, called *shadowing*, allows for a smooth program extraction from decision procedures whatever the complexity of the dependently typed proofs.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

General Terms Verification, Theory

Keywords Partial derivatives of regular expressions, Mirkin prebases, Coq, Ssreflect

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 algorithm, 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 an inductive type.

PLPV'12, January 24, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1125-0/12/01…$10.00

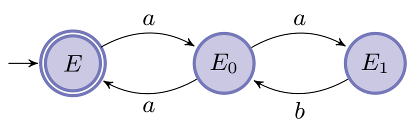

Figure 1. The automaton of partial derivatives of $((a{\times}(a{\times}b)^*){\times}a)^*$

The notion of prebase is a non-deterministic generalisation of the notion of *base* [5], that is, the set of word derivatives of E.

Let us start with an example. Consider the following non-deterministic automaton: The automaton can be represented in the following table:

	a	b
E	$\{E_0\}$	$\emptyset$
E_0	$\{E, E_1\}$	$\emptyset$
E_1	$\emptyset$	$\{E_0\}$

In the table representation, we have the vector of states of the NFA:

$$\begin{bmatrix} E \\ E_0 \\ E_1 \end{bmatrix}$$

the transition matrix:

$$\begin{bmatrix} \{E_0\} & \emptyset \\ \{E, E_1\} & \emptyset \\ \emptyset & \{E_0\} \end{bmatrix}$$

and also the ordered set of alphabetic symbols in which the set of regular expressions is parametric: $[a; b]$.

This automaton recognises the language denoted by the regular expression $E = ((a{\times}(a{\times}b)^*){\times}a)^*$ (Figure 1). It can be observed that E_0 and E_1 recognise *derivative languages*, those obtained by residuating the language of E with a word (for example, with a and aa respectively). The expression E is its own derivative in the trivial way: with respect to the empty word. In fact, this automaton is a minimal NFA recognising E which is obtained by an *optimised* prebase. The construction we present in this paper is simpler and is not optimised (see Example 2).

Methodology. We use *shadowing* of richly typed regular expressions of type re (Section 8) by plain regular expressions of type $\underline{\mathsf{re}}$ (Section 6). This allows to overcome the incompatibility of the current implementation of program extraction of Coq [19] with module types and unnamed record fields[1] that appear for technical reasons in the type hierarchy of Ssreflect [10]. A form of this method seems to have been used recently in [4]. In addition to that, we make use of the $\mathsf{subType}$ canonical structure in the Ssreflect libraries to establish a connection between the shadow (simply typed) regular expressions and the richly typed ones. The shadowing approach is based on the idea of adjunction between a given type T and its subtype S which can be described as the type of those objects of T that satisfy a given predicate P . More precise definitions of subtypes are given in 4. Using generic notation, this adjunction can be stated as a logical equivalence proposition:

$$(F\ t = s) \leftrightarrow (\mathsf{Some}\ t = G\ s)$$

where F is an injection from the subtype to the underlying type, and G is a *partial* injection from the underlying type onto the subtype. Partiality here means that G maps a given s to an object of the kind $\mathsf{Some}\ t$ if $F\ t$ is defined and is s. Otherwise, if $F\ t$ is undefined, $G\ s$ evaluates to the special value None representing the objects of the underlying type for which there is no corresponding object in the subtype.

Suppose the underlying type T enjoys a decidable equality. This is the case with the type $\mathbb{N}$ of Peano numbers, for example, that is essential in many algorithmic constructions. Let us denote this equality by $==$, so that $t_1 == t_2$ is either true or false. Then the above adjunction can be simplified as follows:

$$(F\ t == s) = (\mathsf{Some}\ t == G\ s)$$

where instead of the propositional equivalence we now use equality of boolean values. This simplified adjunction together with the fact that the relation $==$ *reflects* the usual equality $=$ (and so, $==$ and $=$ can be seen as mutually interchangeable up to a coercion of boolean values into propositions), allows to map computations of values in T that satisfy the predicate P to the corresponding values in S. For example, if T is a simple type of computational values and P is a predicate that expresses the property of the result of the computation being correct (for example, being within specified bounds), then we can choose between the dependently-typed representation of computations in S (that are ideal for theorem proving) and the simply-typed one (for the purposes of programming or, vice versa, for extraction of the computational content). Section 12 contains an example of an application of this technique.

Another useful method, the *small-scale reflection* [10], allows for a generic approach to equivalences of the following kind. Let $P : \mathsf{Prop}$ and $b : \mathsf{bool}$ such that

$$P \quad \leftrightarrow \quad b = \mathsf{true}$$

We can define an eliminator that allows to reduce a proof of P to a proof of $b = \mathsf{true}$, and vice versa. This is very efficient when we work with regular languages, i.e., recursively defined boolean predicates. The rich library of small-scale reflection facts allows to alternate between computational statements such as $(w \in L) = (w \in M)$, which is a an equality of boolean terms, and the corresponding statements of logic, for example,

$$(w \in L) = \mathsf{true} \to (w \in M) = \mathsf{true} \quad \wedge$$
$$(w \in M) = \mathsf{true} \to (w \in L) = \mathsf{true}$$

Related work. Our work is related to a non-computational formalisation of partial derivatives in [1]. To obtain a computational behaviour in the type theory of Coq, we have to employ quite different foundational definitions. The shadowing approach we follow allows for direct functional programming in Coq, and hence for direct proofs of termination-by-construction. Another difference is that we have an explicit simply typed representation of the transition matrix in Coq (Section 6), possibly for the first time in the field of interactive theorem proving. We propose to use the matrix for computational decision algorithms in the paper [14]. The formal proofs of that proposal is currently a work in progress.

Other related works that require slightly different foundations include the library for AC-rewriting and for deciding equivalence of regular languages by Braibant and Pous [4]. It uses dependently-typed matrices with an analogue of the shadowing technique, associating simply-typed matrices with a richly-typed equivalence relation for matrix bounds checking. This library allows to build computational definitions inside Coq and uses the extension of first-order Type Classes in Coq that interacts well with Extraction. There is also an original work by Krauss and Nipkow on Brzozowski derivatives in the Isabelle theorem prover [15]. They provide a remarkable algorithm for closure-computation of the binary bisimilarity relation on a pair of regular expressions that does not require a proof of its termination. Such computations are definable in Isabelle thanks to a special *while-option* combinator. In contrast, Coq requires different approaches, such as those of [4] or the present paper. The termination of the non-empty bisimilarity computation algorithm is proved in [15] by adapting original inductive proofs by Brzozowski [5]. If it exists and once computed, the non-empty bisimilarity relation can be given as input to an equivalence checking algorithm also provided in [15] that infers equivalence of the languages denoted by the two regular expressions as a corollary of the lemma of Rutten [17].

Although the derivative approach is not very popular among the programmers of practical regular expression libraries, a number of promising applications is available from the research-active community. Brzozowski derivatives have been successfully applied to definitions of terminating algorithms for parsing [7] in dependent type theory, and also to regular expression containment [11, 12]. The aspect of derivatives is exploited in the latter paper that is related to a notion of proof search by delayed applications of the non-deterministic sum. The resulting construction is very interesting in that in makes promising connections with proof theory. At the same time, the formal calculus is very large, which has been a source of difficulties in achieving a soundness proof of the proof search procedure.

Partial derivatives find their application in the same areas as Brzozowski derivatives, for example, Antimirov derivatives in regular expression matching [18], 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 derivatives (Mirkin derivative and Antimirov derivative) was performed in [6].

Notational conventions. For the sake of presentation, we do not provide listings of Coq code, which would be plain ASCII. Instead, throughout the paper, we use a human-oriented type-theoretic notation, where Type denotes the universe of types, and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines. For example, the inductive definition of the type Σ of dependent sum is written in two steps. First, we define the *universe*, of which our type is inhabitant (to the right of the semicolon):

$$\Sigma \quad : \quad \forall_{(A:\mathsf{Type})}.\, (A \to \mathsf{Type}) \to \mathsf{Type}$$

and second, we define the *constructors* of the type by providing a natural deduction rule for each constructor. In the case of Σ, there

[1] There has been a report that extraction has been patched for unnamed fields in a development version.

is only one constructor, and so, only one rule:

$$\cfrac{\cfrac{x : A}{p : P\,x}}{\textsf{exist}_{\,A\ P\ x}\ p : \Sigma_{\,A}\ P}$$

with exist being the name of the constructor. The structure of a rule is a finite tree whose root contains the conclusion of the rule. Implicit parameters are written as subscripts. They are omitted when a definition is applied.

2. Motivation

We start with an algorithmic motivation. First let us define a *monomial* to be a regular expression whose main operation is not $+$, and a *polynomial* to be a finite, possibly empty sum of monomials. We say that a monomial E is in *head normal form* if it is of the kind $a_j \times F$ for some symbol a_j and a regular expression F. Suppose we are given a polynomial $E = E_1 + \cdots + E_k$. The polynomial E can be reduced to a polynomial with all monomials in head normal form by recursive application of the following rewrite rule schemes to every monomial (where the default association of operations is to the left):

$$(F_1 \times \cdots \times F_l \times F_{l+1}) \times G \mapsto (F_1 \times \cdots \times F_l) \times (F_{l+1} \times G) \tag{1}$$

$$(F_1 + \cdots + F_l) \times G \mapsto F_1 \times G + \cdots + F_l \times G \tag{2}$$

$$F^* \times G \mapsto F \times (F^* \times G) + G \tag{3}$$

$$F^* \mapsto F \times F^* + 1 \tag{4}$$

Then, the set of monomials can be rearranged using the fact that the operation $+$ is associative, commutative and idempotent expressed by the following *ACI equivalences*:

$$\begin{aligned} F + (G + H) &=_L (F + G) + H \\ F + G &=_L G + F \\ F + F &=_L F \end{aligned} \tag{5}$$

as well as using the following equivalence from the structure of additive monoid on regular languages on A:

$$F =_L F + 0 \tag{6}$$

Therefore E can be reduced to a polynomial

$$E^1 = a_{i_1} \times E^1_1 + \cdots + a_{i_{p_1}} \times E^1_{p_1} + o(E) \tag{7}$$

where p_1 is the number of monomials that are neither 0 nor 1, and where $o(E)$ denotes the regular expression 1 if E is nullable (meaning that it denotes a language containing the empty string), and 0 otherwise. The formal definition of o is given in Section 6. Then, we apply the same recursive algorithm to reduce $E^1_1, \dots, E^1_{p_1}$, and so on, until the obtained polynomial is of the form

$$E^q = a_{j_1} \times E^q_1 + \cdots + a_{j_{p_q}} \times E^q_{p_q} + o(E^q)$$

where $E^q_1, \dots, E^q_{p_q}$ are already defined, that is, these regular expressions appear in the list

$$E^1_1, \dots, E^1_{p_1}, \dots, E^{q-1}_1, \dots, E^{q-1}_{p_{q-1}}$$

where some duplicates may occur already. Termination of this recursive process is the goal of the Main Theorem (Section 9) in this paper for which we provide a formalised proof by structural induction on E.

Example 1. Consider the concrete alphabet $A = [a; b]$ and the regular expression $E = ((a \times (a \times b))^* \times a)^*$ over this alphabet. Using the four rewrite rule schemes (1)–(4), the ACI-equivalences (5), the equivalence (6) and the obvious congruence rules for $=_L$, we obtain the following system of equations describing the automaton in Figure 1:

$$\begin{aligned} E &=_L a \times (a \times b)^* \times a \times E + 1 \\ &=_L a \times E_0 + o(E), \\ E_0 &= (a \times b)^* \times a \times E \\ &=_L (a \times b) \times E_0 + a \times E \\ &=_L a \times E_1 + a \times E + o(E_0), \\ E_1 &= b \times E_0 \\ &=_L b \times E_0 + o(E_1) \end{aligned}$$

Thus the regular expression E is described in terms of its output and the regular expression E_0 which denotes a residual language that results from taking those words in the language of E that start from the symbol a and removing that occurrence of a. In the terminology of this paper, the vector of partial derivatives of E consists of regular expressions E, E_0 and E_1. Note that we can safely add 0 denoted by $o(E_0)$ in the last line of the equivalence describing E_0 to get the required form of the polynomial (7). Similarly, we also add 0 denoted by $o(E_1)$ when describing E_1.

3. Basic Finite Objects

It is standard to have a list-based representation of finite structures. Advanced finite types may hide the low-level representation. This is also the case with Ssreflect libraries. However, due to technical complexity of the dependently typed hierarchy of structures, it is often impossible to use other facilities such as program extraction in combination with advanced finite types. Our solution is to decouple the simply typed part where computations are defined and the dependently typed part where computations are proved correct. We will define computation-related notions. Lists are defined as follows:

$$\textsf{seq} : \textsf{Type} \to \textsf{Type}$$

$$\cfrac{}{\textsf{nil}_{\,A} : \textsf{seq}\ A} \qquad \cfrac{a : A \quad s : \textsf{seq}\ A}{\textsf{cons}_{\,A}\ a\ s : \textsf{seq}\ A}$$

We will abbreviate nil as [] and cons a s as a::s. Also we employ a list concatenation function cat $_A$: seq A → seq A → seq A with the usual recursive definition and abbreviate cat s_1 s_2 as s_1++s_2.

Below are a few basic definitions on lists. First, the map combinator:

map : $\forall_{\,T_1\ T_2}.\ (T_1 \to T_2) \to \textsf{seq}\ T_1 \to \textsf{seq}\ T_2$
map f [] = []
map f (x::s) = f x :: map f s

The abbreviation for map $(\lambda\, x.\ g\ x)\ s$ is $[g\ x \mid x \in s]$. Next, the *head* of a list (with a fallback value):

head : $\forall_{\,T}. T \to \textsf{seq}\ T \to T$
head x [] = x
head x (y::_) = y

and the *tail* function:

behead : $\forall_{\,T}.\ \textsf{seq}\ T \to \textsf{seq}\ T$
behead [] = []
behead (_::s) = s

We will now define an appropriate notion of a bounded number. It should encapsulate an upper bound and a proof that the bounded number is less than the bound. First, take the usual inductive definition of the type of (unary) natural number:

$$\mathbb{N} : \textsf{Type}$$

$$\frac{}{0 : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{S}\, n : \mathbb{N}}$$

We define the (truncated) predecessor of a natural number as follows:

$\mathsf{predn} : \mathbb{N} \to \mathbb{N}$
$\mathsf{predn}\ 0 = 0$
$\mathsf{predn}\ (\mathsf{S}\ n) = n$

The definition of natural addition, addn, is standard and such that $1 + n$ is convertible with $S\ n$. Natural numbers enjoy a decidable less-than relation. It can be defined via the usual truncated subtraction and decidable equality, that is, a relation with values true or false of type bool. Let us recall the equality relation on natural numbers as follows:

$$\begin{array}{lcl} \_ == \_ & : & \mathbb{N} \to \mathbb{N} \to \mathsf{bool} \\ 0 == 0 & = & \mathsf{true} \\ S\, m == S\, n & = & m == n \\ 0 == S\, n & = & \mathsf{false} \\ S\, m == 0 & = & \mathsf{false} \end{array}$$

The less-then relation is then a function

$$m < n \quad = \quad S\, m - n == 0$$

Thus the type of bounded numbers can be defined as follows:

$$\mathbb{I} : \mathbb{N} \to \mathsf{Type}$$

$$\frac{p : m < n}{\mathsf{Ordinal}\; n\; m\; p : \mathbb{I}\; n}$$

So, we have a dependent inductive type here, with the type of the variable p depending on the value m. The type $\mathbb{I}\ n$ is a special dependent pair type, simpler than Σ from the Introduction, specified on the concrete boolean predicate $\lambda\ m.\ m < n$. The proof of $m < n$ is encoded as a boolean value. This has two outcomes. On one hand, it is easier to reason by cases on boolean-valued relations than on more general relations with values in Type in situations where the relation can be computed inside Coq, provided that the relevant inversion principles exist. Therefore this definition of bounded number facilitates proof by cases in the reflexive interpretation. On the other hand, the type $\mathbb{I}\ n$ is a subtype of $\mathbb{N}$ by the coercion

$$\mathsf{nat\_of\_ord} : \forall_{(n:\mathbb{N})}.\ (\mathbb{I}\ n) \to \mathbb{N}$$

$$\mathbb{N}\mathrm{of}\mathbb{I}\ i \quad = \quad \textbf{let}\ \mathsf{Ordinal}\ \_\ m\ \_ = i\ \textbf{in}\ m$$

This permits application of lemmas for $\mathbb{N}$ to statements about $\mathbb{I}\ n$ without recursive conversion of finite numbers to natural numbers.

The following function is used to obtain a list of natural numbers from m to $n - 1$:

$\mathsf{iota} : \mathbb{N} \to \mathbb{N} \to \mathsf{seq\ nat}$
$\mathsf{iota}\ m\ 0 = []$
$\mathsf{iota}\ m\ (\mathsf{S}\ n) = m :: \mathsf{iota}\ (\mathsf{S}\ m)\ n$

There is also a generic iterator iter:

$\mathsf{iter} : \forall_{\ (T:\mathsf{Type})}.\ \mathbb{N} \to (T \to T) \to T \to T$
$\mathsf{iter}\ 0\ f\ x = x$
$\mathsf{iter}\ (\mathsf{S}\ n)\ f\ x = f\ (\mathsf{iter}\ n\ f\ x)$

The iterator can be used, for example, to define a list of n copies of x:

$\mathsf{nseq} : \forall_{\ T}.\ \mathbb{N} \to T \to \mathsf{seq}\ T$
$\mathsf{nseq}_{\ T}\ n\ x = \mathsf{iter}\ n\ (\mathsf{cons}\ x)\ []$

The types seq T, for a given T, and $\mathbb{N}$ are inhabited by infinitely many finite objects. Therefore, although their inhabitants are finite, the types themselves are not. However, given these types, one can think of other types where finiteness is expressed as part of their structure. The type $\mathbb{I}\ n$, with a bound n, is a basic example of a type that is inhabited by finitely many objects and is a subtype of $\mathbb{N}$ in the precise sense of Section 4. More generic finite types of which $\mathbb{I}\ n$ is itself a particular case have an underlying canonical enumeration of elements. This enumeration is exactly a list of type seq T of elements of the underlying type T. We describe this kind of finite types in Section 5.

4. Subtypes

In this section we will see exactly how $\mathbb{I}\ n$ is presented in Ssreflect as a subtype of $\mathbb{N}$. In general, subtypes can be modelled in type theory by dependent pairs $\Sigma_{\ A}\ P$ with the definition in the Introduction where A is a type and P is an informative predicate on A. There are two projections associated with $\Sigma_{\ A}\ P$. The first projection π_1 maps a term x of type $\Sigma_{\ A}\ P$ to its component in A. The second projection π_2 maps x to the proof that P holds for the component in A, that is, to $P\ (\pi_1\ x)$. This modelling method is very generic, which has its conveniences and discomforts. The terms of the subtype can be easily coerced to the corresponding terms in the underlying type by the first projection. On the other hand, comparison of two terms both of type $\Sigma_{\ A}\ P$ requires to compare first and then second components of the terms pairwise. Often, comparing the first components does not present a difficulty as long as the relation on A that is being checked is decidable. It is the comparison of the second components that can become quite technical and eventually requires to actively use axioms that allow comparison of dependent components. In a development that significantly relies on the possibility of explicit expression of given types being subtypes of other types, there is therefore a worthwhile task to find less general methods that make comparison of dependent pairs more tangible, even though that might restrict the class of acceptable dependent pair as well.

For decidable predicates, that is, for predicates of type

$$\mathsf{pred}\ T = T \to \mathsf{bool}$$

a method to define subtypes in Coq has been introduced in [8]. This is the method employed in the Ssreflect libraries. We explain it here for completeness reasons. First, let us restrict the type of the second component and define subType as follows:

$$\mathsf{subType} : \forall_{\ T:\mathsf{Type}}.\ \mathsf{pred}\ T \to \mathsf{Type}$$

The next crucial step is to syntactically separate the actual subtype from the underlying type T. This is done by introducing a new parameter (which can also be defined as a coercion) sub_sort representing the subtype, along with the first projection val that maps the subtype to the underlying type, and the dependent injection Sub that constructs an object in the subtype given an object x in the underlying type such that x satisfies the predicate P. Additionally, we require an eliminator el and a proof q that val cancels Sub.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{sub\_sort : \mathsf{Type}}{val : sub\_sort \to T}}{Sub : \forall\, x.\ P\, x \to sub\_sort}}{el : \forall\, K.\ (\forall\, x\, p.\ K\ (Sub\, x\, p)) \to \forall\, u.\ K\ u}}{q : \forall\, x\, p.\ val\ (Sub\, x\, p) = x}}{\mathsf{SubType}_{\ T\ P}\ sub\_sort\ val\ Sub\ el\ k : \mathsf{subType}_{\ T}\ P}$$

Behind the complexity of this definition there is a clean functionality based on the idea of irrelevance of proofs of boolean predicates. Using this notion of a subtype, we can compare any two objects of a given subtype of a type T by comparing their first projections in T, for which, as we already discussed, a decidable comparison relation on T suffices.

The definition of $\mathbb{I}$ in the previous section gives rise to an eliminator $\mathbb{I}\_\mathsf{rect}$ which is derived automatically by Coq. This eliminator can be given to the constructor $\mathsf{SubType}$ together with the function ($\mathsf{nat\_of\_ord}\ _n$) that stands for the first projection val. Then, the rest of the constructor arguments can be inferred. This application defines subtyping of $\mathbb{I}\ n$ in $\mathbb{N}$.

To obtain the adjunction between the subtype S and the underlying type T, as discussed in the Introduction, we used functions F which is val here, and G, called insub in Ssreflect libraries, which can be defined essentially as follows:

$$\mathsf{insub} : \forall\ _{T\ P}\ (S : \mathsf{subType}\ _{T\ P}).\ T \to \mathsf{option}\ S$$

Noting that T and P are implicit from the argument S, we can write

```
insub $S\ t$ =
  let $p = P\ t$ in
  if $p$ is true then Some (Sub $t\ p$) else None
```

We need one more detail, that is, the canonical finite structure on $\mathbb{I}\ n$, in order to be able to define shadowing.

5. Advanced Finite Types

Advanced finite types feature in Ssreflect libraries [8, 10]. There is a generic type of finite structures $\mathsf{finType} : \mathsf{Type}$ supporting, among others, the following operations, for some $T : \mathsf{finType}$:

1. The expression $\mathsf{enum}\ T$ yields a duplicate-free, canonical list of all elements of T.
2. $\#|T|$ denotes the cardinality of T.
3. $\mathsf{enum\_val}\ i$ is the i-th element of $\mathsf{enum}\ T$ where $i : \mathbb{I}\ \#|T|$.
4. $\mathsf{enum\_rank}\ t$ is the index $i : \mathbb{I}\ \#|T|$ such that $\mathsf{enum\_val}\ i = t$.

Our brief exposition of $\mathsf{finType}$ lacks the full definition of the canonical type of types with decidable equality $==$, called eqType in [8, 10]. It nevertheless appears to be sufficient for the purposes of this paper to describe $\mathsf{finType}$ as a subtype of a type T with decidable equality such that it possesses a canonical enumeration, that is, a list of objects of type T such that each object in the canonical enumeration appears there exactly once. The reader can find formal definitions in the aforementioned sources.

One can now observe that the inhabitants of $\mathbb{I}\ n$ form a canonical enumeration with respect to the relation $<$ from Section 3. Therefore $\mathbb{I}\ n$ has a straightforward canonical finite structure.

6. Prebases

In this section we give recursive definitions for the straightforward non-optimised prebase construction.

First, we define the simply typed regular expressions as follows (where n stands for the number of symbols in the underlying alphabet):

$$\underline{\mathsf{re}} : \mathbb{N} \to \mathsf{Type}$$

$$\frac{}{\underline{\mathsf{Void}}\ _n : \underline{\mathsf{re}}\ n} \qquad \frac{}{\underline{\mathsf{Eps}}\ _n : \underline{\mathsf{re}}\ n} \qquad \frac{i : \mathbb{I}\ n}{\underline{\mathsf{Atom}}\ _n\ i : \underline{\mathsf{re}}\ n}$$

$$\frac{E : \underline{\mathsf{re}}\ n \qquad F : \underline{\mathsf{re}}\ n}{\underline{\mathsf{Alt}}\ _n\ E\ F : \underline{\mathsf{re}}\ n} \qquad \frac{E : \underline{\mathsf{re}}\ n \qquad F : \underline{\mathsf{re}}\ n}{\underline{\mathsf{Conc}}\ _n\ E\ F : \underline{\mathsf{re}}\ n}$$

$$\frac{E : \underline{\mathsf{re}}\ n}{\underline{\mathsf{Star}}\ _n\ E : \underline{\mathsf{re}}\ n}$$

The following output function is straightforwardly terminating:

$$\begin{aligned}
&\underline{\mathsf{o}} : \forall\ _n.\ \underline{\mathsf{re}}\ n \to \mathsf{bool}\\
&\underline{\mathsf{o}}\ \underline{\mathsf{Void}} = \mathsf{false}\\
&\underline{\mathsf{o}}\ \underline{\mathsf{Eps}} \ = \mathsf{true}\\
&\underline{\mathsf{o}}\ (\underline{\mathsf{Atom}}\ i) \qquad = \mathsf{false}\\
&\underline{\mathsf{o}}\ (\underline{\mathsf{Alt}}\ E\ F) \quad\ = \underline{\mathsf{o}}\ E\ ||\ \underline{\mathsf{o}}\ F\\
&\underline{\mathsf{o}}\ (\underline{\mathsf{Conc}}\ E\ F) = \underline{\mathsf{o}}\ E\ \&\&\ \underline{\mathsf{o}}\ F\\
&\underline{\mathsf{o}}\ (\underline{\mathsf{Star}}\ E) \qquad = \mathsf{true}
\end{aligned}$$

Below are the functions that derive vectors of partial derivatives and transition matrices for component regular expressions in a purely syntactic way. Transition matrices inhabit the following simple type:

$$\mathsf{mat}\ T = \mathsf{seq}\ (\mathsf{seq}\ T)$$

It can be seen that $\mathsf{mat}\ T$ allows to represent arbitrary list structures that may not correspond to matrices. The fact that only correct matrices are generated in the prebase formation, given that the input matrices are correct, is proved as part of the correctness criterion stated in Section 9 and formalised in Section 11.

The base cases of the recursive computation of the prebase vector and matrix involve constructors $\underline{\mathsf{Void}}$, $\underline{\mathsf{Eps}}$ and $\underline{\mathsf{Atom}}$. The first two cases are similar because, except for the empty word, whatever word we choose to residuate the these regular expressions with, we obtain $\underline{\mathsf{Void}}$, with no derivatives. Therefore we still have to provide one row of the prebase matrix in $\mathsf{pmatVoid}$ and $\mathsf{pmatEps}$ just to express the fact that these regular expressions have no derivatives with respect to any symbol.

$$\begin{aligned}
&\mathsf{ptupVoid}\ : \forall\ _n.\ \mathsf{seq}\ (\underline{\mathsf{re}}\ n)\\
&\mathsf{ptupVoid}\ n = [\underline{\mathsf{Void}}\ _n]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{pmatVoid}\ : \mathbb{N} \to \mathsf{mat}\ (\mathsf{seq}\ \mathbb{N})\\
&\mathsf{pmatVoid}\ n = \mathsf{nseq}\ n\ []
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ptupEps}\ : \forall\ _n.\ \mathsf{seq}\ (\underline{\mathsf{re}}\ n)\\
&\mathsf{ptupEps}\ n = [\underline{\mathsf{Eps}}\ _n]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{pmatEps}\ : \mathbb{N} \to \mathsf{mat}\ (\mathsf{seq}\ \mathbb{N})\\
&\mathsf{pmatEps}\ n = \mathsf{pmatVoid}\ n
\end{aligned}$$

The third base case is only slightly bigger. There is a new derivative with respect to the given symbol with number i.

$$\begin{aligned}
&\mathsf{ptupAtom} : \forall\ _n.\ \mathbb{I}\ n \to \mathsf{seq}\ (\underline{\mathsf{re}}\ n)\\
&\mathsf{ptupAtom}\ i = [\underline{\mathsf{Atom}}\ i;\ \underline{\mathsf{Eps}}]
\end{aligned}$$

The prebase matrix must have two rows, with the first row containing the reference to the second row in the column of the given symbol. This reference is denoted by 0 (recall the numbering schema in Example 2). Other columns are empty, and all the columns in the second row are empty as well since $\underline{\mathsf{Eps}}$ has no derivatives with respect to symbols.

```
pmatAtom : $\mathbb{N} \to \mathbb{N} \to$ mat (seq $\mathbb{N}$)
pmatAtom $n\ a$ =
  [if $j == a$ then [0] else [] | $j \in$ iota 0 $n$] ::
  nseq $n$ []
```

There are three recursive cases where vectors are obtained from vectors of the component regular expressions. Let us consider the case of $\underline{\mathsf{Alt}}$ first. We construct a vector of length $1 + (\mathsf{size}\ t_1 - 1) + (\mathsf{size}\ t_2 - 1)$. The vector of derivatives of $\underline{\mathsf{Alt}}\ E\ F$ consist of this expression itself (trivially), the non-trivial derivatives of E, and the non-trivial derivatives of F. Here we have a choice whether to perform optimisations, for example, by comparing the languages of derivatives obtained from different components, in order to achieve the best space characteristics. However, in this paper we work with the simplest computational definitions which are purely syntactic in that no language denotations are computed at this stage. The first element of the vector of a component regular expression is this regular expression itself, by construction. Note that, since the sizes of the component vectors are are always greater than zero, which is according to the base cases, the simply-typed computation of the component regular expressions does never default to the value $\underline{\mathsf{Void}}$ which is provided here solely for the reason that the function head requires a default value. Dually, the computation of the tail by the function behead never defaults to the empty list.

```
ptupAlt : $\forall_n$. seq ($\underline{\text{re}}$ $n$) → seq ($\underline{\text{re}}$ $n$) → seq ($\underline{\text{re}}$ $n$)
ptupAlt $t_1$ $t_2$ =
  ($\underline{\text{Alt}}$ (head $\underline{\text{Void}}$ $t_1$) (head $\underline{\text{Void}}$ $t_2$)) ::
  behead $t_1$ ++ behead $t_2$
```

With the dimensions of the matrix known from the vector construction, we have to fill in the first row of the matrix with disjoint unions of references to the corresponding rows in the submatrices of non-trivial derivatives of the component regular expressions. The disjoint union is implemented by list concatenation. To access the cells of the component matrices we use a function $\mathsf{nth}_T\ x\ s\ n$ that computes the n-th element of the list s of type $\mathsf{seq}\ T$, with the mandatory default value x. Therefore the cell (i, j) can be computed by the following function where M is a matrix of type $\mathsf{mat}\ (\mathsf{seq}\ T)$:

$$\mathsf{cell}_T\ M\ i\ j = \mathsf{nth}\ []\ (\mathsf{nth}\ []\ M\ i)\ j$$

The rows below the first one contain copies of the corresponding rows from the component matrices, however, the rows from the second matrix have been shifted by the size of the first matrix less the first row. Therefore all the references to rows originating from the second matrix should be shifted accordingly. Note, similar to the atomic case, the behaviour of the predecessor function on natural numbers is non-trivial provided that correct matrices are given as input.

```
pmatAlt  : ℕ → mat (seq ℕ) →
             mat (seq ℕ) → mat (seq ℕ)
pmatAlt$_n$ $M_1$ $M_2$ =
  let $m_1$ = size $M_1$ in
  let $m_2$ = size $M_2$ in
  [cell $M_1$ 0 $j$ ++
     [addn (predn $m_1$) $k$ | $k$ ∈ cell $M_2$ 0 $j$]
  | $j$ ∈ iota 0 $n$]  ::
  behead $M_1$ ++
  [
    [
      [addn (predn $m1$) $k$ | $k$ ∈ cell $M_2$ $i$ $j$]
    | $j$ ∈ iota 0 $n$]
  | $i$ ∈ iota 1 predn $m_2$]
```

Turning to the case of $\underline{\mathsf{Conc}}$, we remark that it requires no new auxiliary function definitions. Moreover, the vector is of the same size as in the previous case. However, the first size t_1 elements are concatenations of a derivative of the first component regular expression with the second component expression. The remaining elements represent derivatives for the case when the first expression matches the empty string.

```
ptupConc : $\forall_n$. seq ($\underline{\text{re}}$ $n$) → seq ($\underline{\text{re}}$ $n$) → seq ($\underline{\text{re}}$ $n$)
ptupConc $t_1$ $t_2$ =
  [$\underline{\text{Conc}}$ $E$ (head $\underline{\text{Void}}$ $t_2$) | $E$ ∈ $t_1$] ++ behead $t_2$
```

Concatenation behaves differently depending on whether or not the first component regular expression is nullable. This transforms to the derivative setting as well. So, we add an extra argument of type seq bool to the function constructing the concatenation matrix to represent the nullability characteristic of each of the derivatives of the first expression. Then we shift the values of the references to the rows from the second component matrix just as we did in the case of $\underline{\mathsf{Alt}}$.

```
pmatConc : ℕ → mat (seq ℕ) →
             mat (seq ℕ) → seq bool → mat (seq ℕ)
pmatConc $n$ $M_1$ $M_2$ $o_1$ =
  let $m_1$ = size $M_1$ in
  let $m_2$ = size $M_2$ in
  [
    [
      cell $M_1$ $i$ $j$ ++
      if nth false $o_1$ $i$
      then [addn (predn $m_1$) $k$ | $k$ ∈ cell $M_2$ 0 $j$]
      else []
    | $j$ ∈ iota 0 $n$]
  | $i$ ∈ iota 0 $m_1$] ++
  [
    [
      [addn (predn $m_1$) $k$ | $k$ ∈ cell $M_2$ $i$ $j$]
    | $j$ ∈ iota 0 $n$]
  | $i$ ∈ iota 1 (predn $m_2$)]
```

The last is the case of $\underline{\mathsf{Star}}$. Here we have similarity with the case of $\underline{\mathsf{Conc}}$, which is intuitively clear, and even more simple than that because there is only one component regular expression.

```
ptupStar : $\forall_n$. seq ($\underline{\text{re}}$ $n$) → seq ($\underline{\text{re}}$ $n$)
ptupStar $t$ =
  let $E$ = $\underline{\text{Star}}$ (head $\underline{\text{Void}}$ $t$) in
  $E$ :: [$\underline{\text{Conc}}$ $E'$ $E$ | $E'$ ∈ behead $t$]
```

Like in the case of concatenation, we make use of a list of nullability characteristics.

```
pmatStar  : ℕ → mat (seq ℕ) →
              seq bool → mat (seq ℕ)
pmatStar $n$ $M$ $o_1$ =
  let $m$ = size $M$ in
  head [] $M$  ::
  [
    [
      cell $M$ $i$ $j$ ++
      if nth false $o_1$ $i$ then cell $M$ 0 $j$ else []
    | $j$ ∈ iota 0 $n$]
  | $i$ ∈ iota 1 (predn $m$)]
```

Putting it all together, the vector of partial derivatives and the transition matrix are computed by the following functions:

```
ptup : $\forall_n$. $\underline{\text{re}}$ $n$ → seq ($\underline{\text{re}}$ $n$)
ptup $\underline{\text{Void}}$ = ptupVoid
ptup $\underline{\text{Eps}}$  = ptupEps
ptup ($\underline{\text{Atom}}$ $i$)     = ptupAtom $i$
ptup ($\underline{\text{Alt}}$ $E$ $F$)   = ptupAlt (ptup $E$) (ptup $F$)
ptup ($\underline{\text{Conc}}$ $E$ $F$) = ptupConc (ptup $E$) (ptup $F$)
ptup ($\underline{\text{Star}}$ $E$)     = ptupStar (ptup $E$)
```

```
pmat : $\forall_n$. $\underline{\text{re}}$ $n$ → mat (seq ℕ)
pmat $\underline{\text{Void}}$ = pmatVoid
pmat $\underline{\text{Eps}}$  = pmatEps
pmat ($\underline{\text{Atom}}$ $i$)     = pmatAtom (ℕofI $i$)
pmat ($\underline{\text{Alt}}$ $E$ $F$)   = pmatAlt (pmat $E$) (pmat $F$)
pmat ($\underline{\text{Conc}}$ $E$ $F$) = pmatConc (pmat $E$) (pmat $F$)
                        [$\underline{\text{o}}$ $i$ | $i$ ∈ ptup $E$]
pmat ($\underline{\text{Star}}$ $E$)     = pmatStar (pmat $E$)
                        [$\underline{\text{o}}$ $i$ | $i$ ∈ ptup $E$]
```

The computed values are correct vectors and matrices given that the inputs are correct. Type theory offers many possible choices of proving this. In the remaining sections we argue in favour of one particular choice of implementation involving type-rich regular expressions because type enrichment allows to define the computable language interpretation of a rich regular expression. This would not be possible without an explicit type of finite structures finType, with its special instrumentation. At the same time, by proving the correctness of the simply-typed construction, we establish a subtype relation between the richly typed and simply typed versions of regular expressions. This allows to extract computations from dependently typed constructions inside Coq.

To illustrate the use of the simply typed construction, we provide a computation of a non-optimised prebase for the regular expression we are already familiar with since the Introduction.

Example 2 (Expression $((a{\times}(a{\times}b)^*){\times}a)^*$).

$$\begin{array}{l}
\mathsf{E} = \underline{\mathsf{Star}} \\
\quad (\underline{\mathsf{Conc}} \\
\quad\quad (\underline{\mathsf{Conc}}\ (\underline{\mathsf{Atom}}\ a) \\
\quad\quad\quad (\underline{\mathsf{Star}}\ (\underline{\mathsf{Conc}}\ (\underline{\mathsf{Atom}}\ a)\ (\underline{\mathsf{Atom}}\ b)))) \\
\quad\quad (\underline{\mathsf{Atom}}\ a))
\end{array}$$

The vector of partial derivatives $\mathsf{P}_E = \mathsf{ptup}\ \mathsf{E}$ contains representations of the following expressions:

$$\begin{array}{lcl}
\mathsf{E} & = & ((a{\times}(a{\times}b)^*){\times}a)^* \\
\mathsf{E}_0 & = & ((1{\times}(a{\times}b)^*){\times}a)^*{\times}((a{\times}(a{\times}b)^*){\times}a)^* \\
\mathsf{E}_1 & = & (((1{\times}b){\times}(a{\times}b)^*){\times}a)^*{\times}((a{\times}(a{\times}b)^*){\times}a)^* \\
\mathsf{E}_2 & = & ((1{\times}(a{\times}b)^*){\times}a)^*{\times}((a{\times}(a{\times}b)^*){\times}a)^* \\
\mathsf{E}_3 & = & 1{\times}((a{\times}(a{\times}b)^*){\times}a)^*
\end{array}$$

The matrix of transitions is computed as follows:

$$\begin{array}{l}
\mathsf{M_E} = \mathsf{pmat}\ \mathsf{E} = \\
\quad [[[0]; \quad []]; \\
\quad\ [[1;3]; \quad []]; \\
\quad\ [[]; \quad [2]]; \\
\quad\ [[1;3]; \quad []]; \\
\quad\ [[0]; \quad []]]
\end{array}$$

Observe that the following two pairs of regular expressions have the same languages: E and E_3; E_0 and E_2. The algorithm is unaware of the denotational semantics, and therefore these equivalences cannot be expressed at this stage. Semantic definitions are provided in the next two sections.

7. Regular Languages

In this section we provide definitions of regular languages according to [9]. These definition give rise to a regular expression pattern matching algorithm already without a notion of derivative. The interest in derivatives comes from the fact that it is difficult to see how can we decide relations on regular expressions such as equivalence or containment in Coq without constructing some intermediate, derivative regular expressions. Partial derivatives are in general more efficient compared to Brzozowski derivatives, which motivates our study.

Ssreflect distinguishes between two kinds of predicates: collective and applicative ones. Both can be seen as functions of the kind $T \to$ bool. One of the differences between these is that the former allows a more general usage compatible with notation $x \in P$ while the latter is simply a function that is applied as usual: $P\ x$. In fact, the predicates in the definition of regular languages can be used either way, however, since the notation $x \in P$ does not simplify, there should be a special coercion mem that transforms a collective predicate to its simplifiable applicative counterpart.

We parametrise *words* over finite types. The latter play the role of the alphabet.

$$\begin{array}{l}
\mathsf{word} : \mathsf{finType} \to \mathsf{Type} \\
\mathsf{word}\ X = \mathsf{seq}\ X
\end{array}$$

The most basic kind of a predicate is $\mathsf{pred} = \lambda\ (T : \mathsf{Type}).\ T \to$ bool. We use straightforward predicates such as the predicate pred0 that always fails, the predicate pred1 that succeeds only for a given object, and the disjunction predU of two predicates. From that, we define the notion of a *language* over a finite alphabet and the basic regular language constants and operations: the empty language void, the language of the empty string eps, the language of a one-symbol word atom, alternation (union) of two languages alt, the concatenation of two languages conc and the iteration (Kleene closure) of a language star.

$$\begin{array}{l}
\mathsf{language} : \mathsf{finType} \to \mathsf{Type} \\
\mathsf{language}\ X = \mathsf{pred}\ (\mathsf{word}\ X) \\
\mathsf{void} : \forall_X .\ \mathsf{language}\ X \\
\mathsf{void} = \mathsf{pred0} \\
\mathsf{eps} : \forall_X .\ \mathsf{language}\ X \\
\mathsf{eps} = \mathsf{pred1}\ [] \\
\mathsf{atom} : \forall_X .\ X \to \mathsf{language}\ X \\
\mathsf{atom}\ a = \mathsf{pred1}\ [a] \\
\mathsf{alt}\ : \forall_X .\ \mathsf{language}\ X \to \mathsf{language}\ X \to \mathsf{language}\ X \\
\mathsf{alt}\ L_1\ L_2 = \mathsf{predU}\ (\mathsf{mem}\ L_1)\ (\mathsf{mem}\ L_2) \\
\mathsf{conc} : \forall_X .\ \mathsf{language}\ X \to \mathsf{language}\ X \to \mathsf{language}\ X \\
\mathsf{conc}\ L_1\ L_2\ w = \\
\quad \#\big|\lambda\ (i : \mathbb{I}\ (1 + (\mathsf{size}\ w)). \\
\quad\quad L_1\ (\mathsf{take}\ i\ w)\ \&\&\ L_2\ (\mathsf{drop}\ i\ w)\big|\ \mathrel{!=}\ 0
\end{array}$$

The use of the cardinality function $\#|\_|$ in the definition of conc is remarkable in that it forces unification of the function inside with its canonical representation of type finType, which allows to access the function graph as a canonical enumeration. The cardinality function then computes the number of elements of the underlying type satisfying the finite predicate expressed by the function. This number is non-zero if and only if the concatenation of the two languages is non-empty. Therefore the cardinality function helps to define a computational existential quantifier.

Before giving the definition of the star language, we require a computation of a *residual language* by the following function:

$$\begin{array}{l}
\mathsf{residual} : \forall_X .\ X \to \mathsf{language}\ X \to \mathsf{language}\ X \\
\mathsf{residual}\ a\ L = \lambda\ w.\ L\ (\mathsf{cons}\ a\ w)
\end{array}$$

Now we can define a structurally recursive version of language iteration:

$$\begin{array}{l}
\mathsf{star} : \forall_X .\ \mathsf{language}\ X \to \mathsf{language}\ X \\
\mathsf{star}\ L = \\
\quad \mathbf{letrec}\ \mathsf{star}' : \mathsf{word}\ X \to \mathsf{bool}\ \mathbf{in} \\
\quad\quad \mathsf{star}'\ (a{::}w) = \mathsf{conc}\ (\mathsf{residual}\ a\ L)\ \mathsf{star}'\ w \\
\quad\quad \mathsf{star}'\ [] = \mathsf{true}
\end{array}$$

A non-trivial organisation of finType that shields away the body of the cardinal function in the definition of conc from direct evaluation allows to satisfy the requirement of having a structurally decreasing argument in the definition of the star language. This is a situation where type enrichment in fact *helps* in definition of a structurally recursive function.

8. Richly Typed Regular Expressions

With all the computational definitions for regular languages at hand, it is possible to define the type-rich version of $\underline{\mathsf{re}}$:

$$\mathsf{re} : \mathsf{finType} \to \mathsf{Type}$$

$$\frac{}{\mathsf{Void}_X : \mathsf{re}\ X} \qquad \frac{}{\mathsf{Eps}_X : \mathsf{re}\ X} \qquad \frac{a : X}{\mathsf{Atom}_X\ a : \mathsf{re}\ X}$$

$$\frac{E : \mathsf{re}\ X \qquad F : \mathsf{re}\ X}{\mathsf{Alt}_X\ E\ F : \mathsf{re}\ X} \qquad \frac{E : \mathsf{re}\ X \qquad F : \mathsf{re}\ X}{\mathsf{Conc}_X\ E\ F : \mathsf{re}\ X}$$

$$\frac{E : \mathsf{re}\ X}{\mathsf{Star}_X\ E : \mathsf{re}\ X}$$

The corresponding output function o is little changed from $\underline{\mathsf{o}}$:

$\mathsf{o} : \forall_X.\ \mathsf{re}\ X \to \mathsf{bool}$
$\mathsf{o}\ \mathsf{Void} = \mathsf{false}$
$\mathsf{o}\ \mathsf{Eps} = \mathsf{true}$
$\mathsf{o}\ (\mathsf{Atom}\ a) = \mathsf{false}$
$\mathsf{o}\ (\mathsf{Alt}\ E\ F) = \mathsf{o}\ E\ ||\ \mathsf{o}\ F$
$\mathsf{o}\ (\mathsf{Conc}\ E\ F) = \mathsf{o}\ E\ \&\&\ \mathsf{o}\ F$
$\mathsf{o}\ (\mathsf{Star}\ E) = \mathsf{true}$

Finally, thanks to these definitions, we can define the semantic representation of a type-rich regular expression:

$\mathsf{lang} : \forall_X.\ \mathsf{re}\ X \to \mathsf{language}\ X$
$\mathsf{lang}\ \mathsf{Void} = \mathsf{void}$
$\mathsf{lang}\ \mathsf{Eps} = \mathsf{eps}$
$\mathsf{lang}\ (\mathsf{Atom}\ a) = \mathsf{atom}\ a$
$\mathsf{lang}\ (\mathsf{Alt}\ E\ F) = \mathsf{alt}\ (\mathsf{lang}\ E)\ (\mathsf{lang}\ F)$
$\mathsf{lang}\ (\mathsf{Conc}\ E\ F) = \mathsf{conc}\ (\mathsf{lang}\ E)\ (\mathsf{lang}\ F)$
$\mathsf{lang}\ (\mathsf{Star}\ E) = \mathsf{star}\ (\mathsf{lang}\ E)$

The relation between re and $\underline{\mathsf{re}}$ is formalised in the proof of the Main Theorem.

9. Correctness Criterion for Prebases

Before we give a final formalised definition of the prebase construction, it is interesting to see how it corresponds to paper-and-pencil constructions found in literature [2, 16]. In the typed version of the theorem of Mirkin we assume an iterated operator $\bigoplus$ whose precise definition follows shortly in the next section. It is a generalised version of the language operator alt obtained by folding with a default value that is shown here by the tail expression with a $+$.

Main Theorem (Mirkin). *For any given regular expression* E *of type* re X*, we can construct a vector of partial derivatives* P *of length* $m > 0$ *and a matrix of transitions* M *of size* $m \times n$*, where* n *is the number of symbols in the alphabet* X*, such that*

(i) The expression located by the coordinate 0 in P *is* E.

(ii) The cells of the matrix M *are subsets of the set* $[0, m-1]$ *of natural numbers.*

(iii) Let $\mathsf{P} = [\mathsf{E}_0; \ldots; \mathsf{E}_{m-1}]$. *For each* $i < m$*, the following language equivalence holds:*

$$\mathsf{E}_i =_L \bigoplus_{0 \le j < n} \Big(\bigoplus_{k \in \mathsf{M}\ i\ j} (\mathsf{conc}\ \mathsf{a}_j\ \mathsf{E}_k) + \mathsf{void} \Big) + \mathsf{o}\ \mathsf{E}_i$$

We note that the size of P under the non-optimised prebase construction does not adhere to the better upper bound $|\mathsf{P}| \le ||\mathsf{E}|| + 1$ that the optimised construction enjoys, where $||\mathsf{E}||$ denotes the number of the alphabetic symbols in E.

10. Iterated Language Operators

Now we will describe our extension of canonical iterated operators of [3]. The general scheme reducebig for iterating an operator op on a list r over a function F with a default (terminal) value idx and a filter predicate P is the following:

$\mathsf{reducebig} : \forall_{(R\ I:\mathsf{Type})}.\ R \to (R \to R \to R) \to$
$\quad \mathsf{seq}\ I \to \mathsf{pred}\ I \to (I \to R) \to R$
$\mathsf{reducebig}\ idx\ op\ r\ P\ F =$
$\quad \mathsf{foldr}\ (\lambda\ i\ x.\ \mathbf{if}\ P\ i\ \mathbf{then}\ op\ (F\ i)\ x\ \mathbf{else}\ x)\ idx\ r$

We define the iterated version of alt as

$$\mathsf{reducebig}\ idx\ \mathsf{alt}\ r\ \mathsf{predT}\ F$$

where predT is a predicate that is always true. We simply do not use the predicate argument in this instance. This expression is denoted by

$$\mathsf{alt}^{\mathsf{idx}}_{i \in r}\ F\ i$$

There is a technical subtlety in the actual definition in Coq that allows to control the flow of term simplification. So, in effect, the iterated operators have to be unfolded by special lemmas.

The approach of [3] does not straightforwardly allow for extensional use which would be required for working with iterated collective predicates. Therefore we extend it with an appropriate notion of an *extensional morphism* and provide technical lemmas allowing for equational reasoning (by rewriting) about such predicates in general, and languages in particular. We treat extensional equalities with the following lemma:

$\mathsf{eq\_xbig} : \forall_T\ (op : \mathsf{pred}\ T \to \mathsf{pred}\ T \to \mathsf{pred} T)$
$\quad (opb : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool})\ (idx_1\ idx_2 : \mathsf{pred}\ T).$
$\quad\quad (\forall\ (P\ Q : \mathsf{pred}\ T)\ (x : T).$
$\quad\quad\quad (x \in op\ P\ Q) = opb\ (x \in P)\ (x \in Q)) \to$
$\quad\quad (\forall\ y.\ (y \in idx_1) = (y \in idx_2)) \to$
$\quad\quad \forall\ (r : \mathsf{seq}\ T)\ (F_1\ F_2 : \mathsf{seq}\ T \to \mathsf{pred}\ T).$
$\quad\quad\quad (\forall\ i\ y.\ (y \in F_1\ i) = (y \in F_2\ i)) \to$
$\quad\quad\quad \forall y.\ (y \in op^{idx_1}_{i \in r}\ F_1\ i) = (y \in op^{idx_2}_{i \in r}\ F_2\ i)$

This particular lemma allows to rewrite subterms of the iterated operation with extensionally equal terms.

11. Prebases, Formalised

Using iterated operators, it is easy to formalise the part (iii) of the Main Theorem. Thus we obtain a formalised version of the definition of a prebase. It's type signature is as follows:

$$\mathsf{prebase} : \forall_X.\ \mathsf{re}\ X \to \mathsf{Type}$$

In Section 6, we have already seen the function nth that computes the n-th element of a list, with a mandatory default value. Instead of lists, here we use a more expressive type tuple that has an explicit length parameter, with a proof that the length of the underlying list of the tuple is equal to this parameter. Therefore we have a dependently typed extension of the function nth, called tnth, that does not require a default element. More precisely, this default element is inferred automatically from the parameters of the constructor of tuple. Also, let us denote the canonical zero of a type $\mathbb{I}\ n$ by ord0, for any n. The difference concerns only the proof-irrelevant part. Lastly, the version of richly typed matrices we use is matrix of type $\mathsf{Type} \to \mathbb{N} \to \mathbb{N} \to \mathsf{Type}$, such that $\mathsf{matrix}\ T\ m\ n$ is the type of m by n matrices whose cells are of type T. This is implemented in terms of tuples as an $\#\big|(\mathbb{I}\ m) \times (\mathbb{I}\ n)\big|$-tuple of T, which in fact can be endowed with a canonical finite structure.

For a given E, an object of type $\mathsf{prebase}\ E$ is defined by induction with a single constructor

$$\mathsf{Prebase}_X\ pN\ pP\ pM\ L0\ Li$$

whose arguments are explained below:

$pN : \mathbb{N}$
$pP : \mathsf{tuple}\ (1 + pN)\ (\mathsf{re}\ X)$
$pM : \mathsf{matrix}\ (\mathsf{seq}\ (\mathbb{I}\ pN))\ (1 + pN)\ \#|X|$
$L0 : \forall\ w.\ (w \in (\mathsf{lang}\ (\mathsf{tnth}\ pP\ \mathsf{ord0}))) =$
$\quad\quad (w \in \mathsf{lang}\ E)$
$Li : \forall\ i\ w.$
$\quad\quad (w \in \mathsf{lang}\ (\mathsf{tnth}\ pP\ i)) =$
$\quad\quad (w \in$
$\quad\quad\quad (\mathsf{alt}^{\mathsf{lang}\ (\mathsf{o}\ (\mathsf{tnth}\ pP\ i))}_{j < \#|X|}$
$\quad\quad\quad\quad (\mathsf{alt}^{\mathsf{void}}_{k \in pM\ i\ j}$
$\quad\quad\quad\quad\quad (\mathsf{conc}$

$$
\begin{array}{l}
\quad(\mathsf{atom}\ (\mathsf{enum\_val}\ j))\\
\quad(\mathsf{lang}\ (\mathsf{tnth}\ pP\ k))\\
))))
\end{array}
$$

The Coq proofs are contained in [13].

12. Example of a Problem Solvable by Computation

Below we are analysing in detail one characteristic step in the proof of the correctness criterion. It shows advantages of the computational encoding of regular expressions that allows to perform crucial deduction steps effectively by rewriting in the goal formula.

The example problem originates from our proof of the inductive step of the Main Theorem for the case where the head constructor of E is Alt. We require the dependently typed left and right shifting operations on bounded numbers. Let us now skip the technical definitions and give their types only:

$$\mathsf{lshift} : \forall_{\ m\ n:\mathbb{N}}.\ \mathbb{I}\,m \rightarrow \mathbb{I}\,(m+n)$$

and

$$\mathsf{rshift} : \forall_{\ m\ n:\mathbb{N}}.\ \mathbb{I}\,n \rightarrow \mathbb{I}\,(m+n)$$

Example 3. Let E_1 and E_2 be regular expressions. Assume that E_1 and E_2 both have the prebase structure. Let N_1, P_1, M_1, L_1^0 and L_1^i be the prebase structure of E_1, and let N_2, P_2, M_2, L_2^0 and L_2^i be the prebase structure of E_2.

Using the straightforward regular expression and matrix conversion functions, we can define

$$
\begin{array}{l}
E = \mathsf{Alt}\ E_1\ E_2\\
N = N_1 + N_2\\
P = [\mathsf{re}{\uparrow}\ (\mathsf{erefl}\ \#|X|)\ i\\
\qquad \mid i \in \mathsf{ptup\_alt}\\
\qquad\qquad [\mathsf{re}{\downarrow}\ (\mathsf{erefl}\ \#|X|)\ i \mid i \in P_1]\\
\qquad\qquad [\mathsf{re}{\downarrow}\ (\mathsf{erefl}\ \#|X|)\ i \mid i \in P_2]]\\
M = \mathsf{mat}\ (1+N)\ \ \#|X|\ \ N\\
\qquad\qquad (\mathsf{pmat\_alt}\ \ \#|X|\ \ (\mathsf{mx}{\downarrow}\ \ M_1)\ (\mathsf{mx}{\downarrow}\ \ M_2))
\end{array}
$$

So, for instance, one can show the following equivalence of languages:

$$
\begin{array}{l}
(w \in \mathsf{lang}\ (\mathsf{tnth}\ P\ (\mathsf{lshift}\ N\ \mathsf{ord0}))) =\\
(w \in\\
\quad \mathsf{alt}_{j<\#|X|}^{\mathsf{lang}\ (\mathsf{o}\ (\mathsf{tnth}\ P\ (\mathsf{lshift}\ N\ \mathsf{ord0})))}\\
\qquad \mathsf{alt}_{k\in M\ (\mathsf{lshift}\ N\ \mathsf{ord0})\ j}^{\mathsf{void}}\\
\qquad\quad \mathsf{conc}\\
\qquad\qquad (\mathsf{atom}\ (\mathsf{enum\_val}\ j))\\
\qquad\qquad (\mathsf{lang}\ (\mathsf{tnth}\ P\ (\mathsf{rshift}\ 1\ k))))
\end{array}
$$

This example is interesting in that it allows to apply extensional equality lemma eq_xbig as well as the standard morphism lemma from the Ssreflect libraries. The full proof is a little bit technical, but the interested reader is welcome to refer to the accompanying proofs in [13]. In the proof, we use the fact that the injections re$\uparrow$ and re$\downarrow$ cancel each other.

In relationship to the corresponding functions on representations of matrices, mat$\uparrow$ and mat$\downarrow$, is different. In fact, the function mat$\downarrow$ may eventually help to define a formal notion of subtyping of dependently typed matrices of the kind $\mathsf{matrix}\ (\mathsf{seq}\ (\mathbb{I}\ pN))\ (1+pN)\ \#|X|$ in simply typed matrices of the kind $\mathsf{mat}\ (\mathsf{seq}\ \mathbb{N})$. This is what seem to be hinted by the proofs but has not yet been verbalised.

13. Conclusions and Further Work

We presented a novel formalisation of a functional construction [16] of a non-deterministic finite automaton recognising the language of a given regular expression and introduced notions that contribute to this formalisation. One of the basic methods we employ is the clean separation between the functional programming part and the theorem proving part, the idea that appears, for example, in [4]. Among other outcomes, this allows to have the functional programming part always extractable to Ocaml, Haskell or similar languages, while the type-rich theorem proving counterpart may not have this property being constructed in a manner which cannot be treated directly by automated program extraction procedures. Still, the connection between the two parts is fully proved, machine checked, and allows to alternate between one and the other. The proofs are provided in the contributed proof script [13].

We are working on mechanisation of computational decision procedures for regular language containment and equivalence, for which we gave preliminary paper-and-pencil proofs in terms of prebases in [14]. The formalised proofs of correctness of these decision procedures are not quite ready yet, but we are moving towards a complete formalisation. Also we are working on the optimised prebase construction that has the best upper bound on the number of partial derivatives.

Acknowledgments

I am grateful to the Automata Group of the University of Porto and especially to Nelma Moreira for the kind invitation and interesting discussions on the subject. I would also like to thank PLPV referees for their constructive suggestions to improve the paper. The research is supported by the Marie Curie fellowship "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] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*, LNCS, pages 86–101. Springer-Verlag, 2008.

[4] T. Braibant and D. Pous. An efficient coq tactic for deciding kleene algebras. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-14051-8.

[5] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4): 481–494, 1964. doi: 10.1145/321239.321249.

[6] J.-M. Champarnaud and D. Ziadi. From Mirkin's prebases to Antimirov's word partial derivatives. *Fundam. Inf.*, 45:195–205, January 2001. ISSN 0169-2968.

[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. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863585. URL `http://doi.acm.org/10.1145/1863543.1863585`.

[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. Expressions regulieres, May 2010. E-mail correspondence.

[10] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2011. URL `http://hal.inria.fr/inria-00258384/en/`.

[11] 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.

[12] 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.

[13] V. Komendantsky. Formal proofs of the prebase theorem of Mirkin, 2011. Coq script available at `http://www.cs.st-andrews.ac.uk/~vk/doc/prebase.v`.

[14] V. Komendantsky. Regular expression containment as a proof search problem. In S. Lengrand, editor, *Proceedings of the International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT'11)*, Wrocław, Poland, 30 July 2011.

[15] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, March 2011.

[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] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorigi and R. de Simone, editors, *CONCUR '98*, volume 1466 of *LNCS*, pages 194–218. Springer, 1998.

[18] M. Sulzmann and K. Z. M. Lu. Regular expression matching using partial derivatives, 2010. Draft.

[19] The Coq development team. The Coq proof assistant reference manual. `http://coq.inria.fr/refman/`.