Certified Parsing of Regular Languages

Denis Firsov and Tarmo Uustalu

Institute of Cybernetics, Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
{denis,tarmo}@cs.ioc.ee

Abstract. We report on a certified parser generator for regular languages using the Agda programming language. Specifically, we programmed a transformation of regular expressions into a Boolean-matrix based representation of nondeterministic finite automata (NFAs). And we proved (in Agda) that a string matches a regular expression if and only if the NFA accepts it. The proof of the if-part is effectively a function turning acceptance of a string into a parse tree while the only-if part gives a function turning rejection into a proof of impossibility of a parse tree.

1 Introduction

Parsing is the process of structuring a linear representation (sentence, a computer program, etc.) in accordance with a given grammar. Parsers are procedures which perform parsing. They are being used extensively in a number of disciplines: in computer science (for compiler construction, database interfaces, artificial intelligence), in linguistics (for text analysis, corpora analysis, machine translation, textual analysis of biblical texts), in typesetting chemical formulae, in chromosome recognition, and so on [4]. It is therefore clear that having correct parsers is important for all these disciplines. Surprisingly, relatively little research had been done in field of certified parsing.

However, with the recent development of programming languages with dependent types it has become possible to encode useful invariants in the types and prove properties of a program while implementing it. In this paper we use the system of dependent types of the Agda language [8], which is based on Martin-Löf's type theory. One of the basic ideas behind Martin-Löf's type theory is the Curry-Howard interpretation of propositions as types. A proposition is proved by writing a program of the corresponding type.

Dependent types allow types to talk about values. A classical example of a dependent type is the type of lists of a given length: $\mathsf{Vec\ A\ n}$. Here A is the type of the elements and n is the length of the list. Having such a definition of vector, we can define "safe" functions. Let us look at definition of a safe head function:

$$\mathsf{head} : \forall \{\mathsf{A\ n}\} \to \mathsf{Vec\ A\ (suc\ n)} \to \mathsf{A}$$
$$\mathsf{head\ (x :: xs)} = \mathsf{x}$$

This definition says that the function head accepts only vectors with at least one element. So, it is now the responsibility of the programmer to provide non-empty vectors, otherwise compilation will fail.

G. Gonthier and M. Norrish (Eds.): CPP 2013, LNCS 8307, pp. 98–113, 2013.

In this paper, we have adopted the same general technique of expressing properties of data in their types to implement a library of matrix operations, program a transformation of regular expressions into a Boolean matrix based representation of nondeterministic finite automata (NFAs) and prove it correct. The correctness proof turns NFAs effectively into parsers: the proof of acceptance or rejection of a string gives us a parse tree (a witness of matching) or a proof of impossibility of one.

Our Agda development can be found online at `http://cs.ioc.ee/~denis/cert-reg` .

2 Regular Expressions

Before we start, let us take care of the alphabet (Σ) of our interest. Since many functions require Σ as a parameter and an NFA must share its alphabet with the regular expressions, we define Σ as a global parameter for each module of the parser-generator library.

```
module ModuleName (Σ : Set) (_≟_ : Decidable (_≡_ {A = Σ}))
```

It states that the module is parametrized by an alphabet and also a decidable equality on the alphabet.

The datatype for regular expressions is called RegExp and is defined as

```
data RegExp : Set where
  ε    : RegExp
  ′_   : Σ → RegExp
  _∪_  : RegExp → RegExp → RegExp
  _·_  : RegExp → RegExp → RegExp
  _⁺   : RegExp → RegExp
```

The base cases are regular expressions for the empty string (ε) and for single-character strings ($'\_$). The step cases are given by the regular operations: $\_\cup\_$ for union, $\_\cdot\_$ for concatenation and $\_^+$ for iteration at least once. Note that instead of the Kleene star ($\_^*$) we use plus ($\_^+$). This is more convenient for us and does not restrict generality, as star is expressible as choice between the empty string and plus.

Now, we need to specify when a string (an element of type List Σ) is in the language of the regular expression, which is also called matching. This is done by introducing a *parsing (or matching) relation* (denoted by $\_\blacktriangleright\_$) between strings and regular expressions.

```
String : Set
String = List Σ

data _▶_ : String → RegExp → Set where
  empt  : [] ▶ ε
  symb  : {x : Σ} → [x] ▶ ′ x
  unionl : {u : String} {e₁ e₂ : RegExp} → u ▶ e₁ → u ▶ e₁ ∪ e₂
  unionr : {u : String} {e₁ e₂ : RegExp} → u ▶ e₂ → u ▶ e₁ ∪ e₂
```

```
con    : {u v : String} {e₁ e₂ : RegExp} → u ▶ e₁ → v ▶ e₂
           → u ++ v ▶ e₁ · e₂
plus1  : {u : String} {e : RegExp} → u ▶ e → u ▶ e⁺
plus2  : {u v : String} {e : RegExp}
           → u ▶ e → v ▶ e⁺ → u ++ v ▶ e⁺
```

(Arguments enclosed in curly braces are implicit. The type checker will try to figure out the argument value for you. If the type checker cannot infer an implicit argument, then it must be provided explicitly, e.g., `symb {x}`.)

Let us now examine the constructors of the relation `_▶_`:

1. The constructor `empt` states that the empty string is in the language of the regular expression `ε`.
2. The constructor `symb` states that the string consisting of a single character `x` is in the language of the regular expression `' x`.
3. The constructor `unionl` (`unionr`) states that if a string `u` is in the language defined by e_1 (e_2), then `u` is also in the language of $e_1 \cup e_2$ for any e_2 (e_1).
4. The constructor `con` states that if a string `u` is in the language e_1 and a string `v` is in the language of e_2 then the concatenation of both strings is in the language of $e_1 \cdot e_2$.
5. The constructor `plus1` states that if a string `u` is in the language of `e`, then it is also in the language of e^+.
6. The constructor `plus2` states that if a string `u` is in the language of `e` and a string `v` is in the language of e^+, then concatenation of both strings is in the language of e^+.

Note that a proof that a string is in the parsing relation with a regular expression is a parse tree. Note that we do not introduce any notion of "raw" parse trees, a parse tree is always a parse tree of a specific string.

3 A Matrix Library

The transition relation of a nondeterministic finite automaton (NFA) can be viewed as a labeled directed graph. So it can be expressed as a family of incidence matrices (one matrix per label). In addition to the nice algebraic properties that this approach highlights, it allows us to compose automata (expressed with matrices) in various ways by using block operations.

We therefore formalize matrices and some important matrix operations and their properties.

3.1 Matrices and Matrix Operations

What sort of elements can a matrix contain? Our approach abstracts from the type of elements. But in order for matrix addition and multiplication to be well-defined and satisfy the standard properties, it must form a commutative semiring. In Agda we introduce a parametrized module

```
module Data.Matrix (sr : CommSemiRing)
```

This declaration says that the module Data.Matrix is parametrized by a commutative semiring sr. A semiring will be a record containing the carrier type R, the operations, and the proofs of the laws of the semiring.

Next we define a representation for matrices. A matrix is a vector of vectors, therefore the matrix type can be defined as

```
_×_ : ℕ → ℕ → Set
k × l = Vec (Vec R l) k
```

where k denotes the number of rows and l the number of columns in a matrix. Let us implement some of the most important operations on matrices:

Null Matrix. The zero or null matrix is a matrix with all entries zero.

```
null : {k l : ℕ} → k × l
null = replicate (replicate zero)
```

Here zero is the additive identity element of the semiring. Note that the arguments k and l are implicit (enclosed in curly braces), so in the most cases we can omit them and the type checker will try to infer their values automatically.

Identity Matrix. The identity or unit matrix of size k is the k × k square matrix with ones on the main diagonal and zeros elsewhere.

```
id : {k : ℕ} → k × k
id {0} = []
id {suc k} = (one :: replicate zero) :: zipWith _+_ null (id {k})
```

Here one is the multiplicative identity element.

Addition. Matrix addition is the operation of adding two matrices by adding corresponding entries together.

```
_⊕_ : {k l : ℕ} → k × l → k × l → k × l
_⊕_ [] [] = []
_⊕_ (rowA :: A′) (rowB :: B′) = zipWith _+_ rowA rowB :: A′ ⊕ B′
```

Note, that signature of addition requires the dimensions of the two input matrices to be equal.

Transposition. The transpose of a matrix A is another matrix ⟨ A ⟩ (or A^T in mathematical notation) created by writing the rows of A as the columns of ⟨ A ⟩.

```
⟨_⟩ : {k l : ℕ} → k × l → l × k
⟨ [] ⟩ = replicate []
⟨ rowA :: A′ ⟩ = zipWith _::_ rowA ⟨ A′ ⟩
```

Multiplication. Matrix multiplication is a binary operation that takes a pair of matrices and produces another matrix. If A is an k × l matrix and B is an l × m matrix, the result A ⊗ B of their multiplication is an k × m matrix

defined only if the number l of columns of the first matrix A is equal to the number of rows of the second matrix B.

```
_⊗_ : {k l m : ℕ} → k × l → l × m → k × m
_⊗_ [] _ = []
_⊗_ (rowA :: A′) B = multRow :: A′ ⋆ B
  where
    multRow = map
      (λ colB → (foldr (_+_) zero (zipWith (_*_) rowA colB))) ⟨ B ⟩
```

The result of matrix multiplication is a matrix whose elements are found by multiplying the elements within a row from the first matrix by the associated elements within a column from the second matrix and summing the products.

In our library we have proved a number of basic properties of matrix transposition, addition, multiplication.

3.2 Block Operations

How to create matrices from smaller matrices systematically? This section describes an approach to block operations on matrices that has been advocated by Macedo and Oliveira [6]. It is centered around four operations corresponding to the injections and projections of the biproducts on the category of natural numbers and matrices.

1. $\iota_1 : \{k\ l : \mathbb{N}\} \to (k + l) \times k$
   $\iota_1 = \mathsf{id} \mathbin{+\!\!+} \mathsf{null}$
2. $\iota_2 : \{k\ l : \mathbb{N}\} \to (k + l) \times l$
   $\iota_2 = \mathsf{null} \mathbin{+\!\!+} \mathsf{id}$
3. $\pi_1 : \{k\ l : \mathbb{N}\} \to k \times (k + l)$
   $\pi_1 = \mathsf{zipWith}\ \_ \mathbin{+\!\!+} \_\ \mathsf{id}\ \mathsf{null}$
4. $\pi_2 : \{k\ l : \mathbb{N}\} \to l \times (k + l)$
   $\pi_2 = \mathsf{zipWith}\ \_ \mathbin{+\!\!+} \_\ \mathsf{null}\ \mathsf{id}$

Example 1. Let us show some instances of these operations:

$$\iota_1\{3\}\{1\} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 0 \end{bmatrix} \qquad \iota_2\{2\}\{2\} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \hline 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\pi_1\{3\}\{1\} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] \qquad \pi_2\{2\}\{2\} = \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right]$$

The main block operations are now defined as follows:

Concatenation (copairing) is the operation of placing two matrices next to each other.

```
[_|_] : {k l m : ℕ} → k × l → k × m → k × (l + m)
[ A | B ] = A ⊗ π₁ ⊕ B ⊗ π₂
```

Stacking (pairing) is the operation of placing two matrices on top of each other.

$$[\_/\_] : \{k\ l\ m : \mathbb{N}\} \to k \times m \to l \times m \to (k + l) \times m$$
$$[\ A\ /\ B\ ] = \iota_1 \otimes A \oplus \iota_2 \otimes B$$

Example 2. Let $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$. Then

$$[\ A\ |\ B\ ] = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{array}\right] \text{ and } [\ A\ /\ B\ ] = \left[\begin{array}{c} 1\ 2 \\ 3\ 4 \\ \hline 5\ 6 \\ 7\ 8 \end{array}\right]$$

3.3 Properties of Block Operations

The main advantage of working with injections and projections is that they structure proofs about block operations. Next, we list some properties of block operations.

- Multiplying the concatenation of A and B by ι_1 (ι_2) yields A (B).

$$\begin{aligned} &m_1m_2\text{-con-}\iota_1 : \{k\ l\ m : \mathbb{N}\}\ (A : k \times l)\ (B : k \times m) \\ &\quad \to [\ A \mid B\ ] \otimes (\iota_1\ \{l\}\ \{m\}) \equiv A \\ &m_1m_2\text{-con-}\iota_2 : \{k\ l\ m : \mathbb{N}\}\ (A : k \times l)\ (B : k \times m) \\ &\quad \to [\ A \mid B\ ] \otimes (\iota_2\ \{l\}\ \{m\}) \equiv B \end{aligned}$$

- Multiplying π_1 (π_2) by the stacking of A and B gives A (B).

$$\begin{aligned} &\pi_1\text{-}m_1m_2\text{-stack} : \{k\ l\ m : \mathbb{N}\}\ (A : k \times l)\ (B : m \times l) \\ &\quad \to \pi_1\ \{k\}\ \{m\} \otimes [\ A\ /\ B\ ] \equiv A \\ &\pi_2\text{-}m_1m_2\text{-stack} : \{k\ l\ m : \mathbb{N}\}\ (A : k \times l)\ (B : m \times l) \\ &\quad \to \pi_2\ \{k\}\ \{m\} \otimes [\ A\ /\ B\ ] \equiv B \end{aligned}$$

- The product of C and the concatenation of A and B is equal to the concatenation of $C \otimes A$ and $C \otimes B$.

$$\begin{aligned} &\text{distrib-lft} : \{k\ l\ m\ n : \mathbb{N}\}\ (A : k \times l)\ (B : k \times m)\ (C : n \times k) \\ &\quad \to C \otimes [\ A \mid B\ ] \equiv [\ C \otimes A \mid C \otimes B\ ] \end{aligned}$$

- The product of a stacking reduces to a stacking of products.

$$\begin{aligned} &\text{distrib-rgt} : \{k\ l\ m\ n : \mathbb{N}\}\ (A : k \times l)\ (B : m \times l)\ (C : l \times n) \\ &\quad \to [\ A\ /\ B\ ] \otimes C \equiv [\ A \otimes C\ /\ B \otimes C\ ] \end{aligned}$$

- Multiplying the concatenation of A and B by the stacking of C and D yields the sum of $A \otimes C$ and $B \otimes D$.

$$\begin{aligned} &\text{con-}\otimes\text{-stack} : \{k\ l\ m\ n : \mathbb{N}\}\ (A : k \times l)\ (B : k \times m) \\ &\quad \to (C : l \times n) \to (D : m \times n) \\ &\quad \to [\ A \mid B\ ] \otimes [\ C\ /\ D\ ] \equiv A \otimes C \oplus B \otimes D \end{aligned}$$

4 NFAs and Parsing with NFAs

We are now in the position to implement a parser generator for regular languages. We parse strings with nondeterministic finite automata and represent them in terms of Boolean matrices.

From now on we therefore use matrices over the commutative semiring of Booleans, with false as zero, disjunction as addition, true as one, conjunction as multiplication.

4.1 Nondeterministic Finite Automata

A Σ-NFA can be defined as a record with four fields.

```
record NFA : Set where
  field ∇ : ℕ
        δ : Σ → ∇ × ∇
        I : 1 × ∇
        F : ∇ × 1
```

- ∇ is the *size of the state space*. We do not name states. Instead we identify them with positions in rows and columns of matrices.
- δ specifies the *transition function*. In our implementation δ is a total function from letters of the alphabet to incidence matrices such that for any $x : \Sigma$ the function call $\delta\ x$ will return an incidence matrix D of size $\nabla \times \nabla$ where $D_{ij} = 1$ iff q_j is a successor of q_i for character x.
- I specifies the *set of initial states*. The initial states can be represented by a $1 \times \nabla$ matrix (row vector) where the element I_{1i} is 1 iff q_i is an initial state.
- F specifies the *set of final states*. The final states can be represented by a $\nabla \times 1$ matrix (column vector) where the element F_{i1} is 1 iff q_i is a final state.

4.2 Running an NFA

Running an NFA on the string $x_0 \ldots x_n$ from a set of states X represented by a row vector can be implemented as the series of multiplications $X \otimes \delta\ nfa\ x_0 \otimes \ldots \otimes \delta\ nfa\ x_n$. This computes the row vector of all states reachable from the states X by following the transitions corresponding to the individual letters of the string $x_0 \ldots x_n$.

```
run : (nfa : NFA) → String → 1 × ∇ nfa → 1 × ∇ nfa
run nfa u X = foldl (λ A x → A ⊗ δ nfa x) X u
```

If we take X to be I and multiply the matrix further with the column vector F of the final states, we get the 1×1 matrix $\mathsf{id}\ \{1\}$ (i.e. $[[\mathsf{true}]]$), if there is an overlap between the states reachable from some initial state and the final states, i.e., if the string is accepted, and null (i.e. $[[\mathsf{false}]]$) otherwise.

```
runNFA : NFA → String → 1 × 1
runNFA nfa u = (run nfa u (I nfa)) ⊗ F nfa
```

4.3 Converting Regular Expressions to NFAs (Parsers)

We have introduced and defined the types RegExp and NFA. We now implement a conversion from RegExp to NFA, which we will use as a parser generator.

```
reg2nfa : RegExp → NFA
reg2nfa ε          = ε′
reg2nfa (′ a)      = ′′ a
reg2nfa (e₁ ∪ e₂)  = (reg2nfa e₁) ∪′ (reg2nfa e₂)
reg2nfa (e⁺)       = (reg2nfa e) ⁺′
reg2nfa (e₁ · e₂)  = (reg2nfa e₁) ·′ (reg2nfa e₂)
```

This function recurses over the regular expression and replaces every constructor with a corresponding operation on NFAs. We describe each case:

- `e = ε`:

  ```
  ε′ = record {
    ∇ = 1 ;
    δ = λ x → null;
    I = id;
    F = id}
  ```

  Clearly, an NFA that accepts only the empty string can be given by one state `0` that is both initial and final.
- `e = ′ a`:

  In this case, the regular expression describes the single-character string `a`. So, the corresponding NFA should accept only this string.

  ```
  ′′ a = record {
    ∇ = 2;
    δ = λ x → if a ≟ x then [ [ null  |  id {1} ]  /
                              [ null  |   null   ] ]
                       else null;
    I = [ id {1}  |  null ];
    F = [ null  /  id {1} ]}
  ```

  This NFA has two states `0` and `1` such that `0` is an initial and `1` a final state. The transition function compares the character `x` with the expected character `a`. If they coincide, then it returns the incidence matrix for the graph with a single edge from `0` to `1`, otherwise, the null matrix for the empty graph.
- `e = e₁ ∪ e₂`:

  Recall that strings of both languages must be accepted. So we must run both NFAs.

  ```
  nfa₁ ∪′ nfa₂ = record {
    ∇ = ∇ nfa₁ + ∇ nfa₂;
    δ = λ x → [ [ δ nfa₁ x |   null   ]  /
                [   null   | δ nfa₂ x ] ];
    I = [ I nfa₁ | I nfa₂ ];
    F = [ F nfa₁  /  F nfa₂ ]}
  ```

The resulting NFA is built of nfa_1 and nfa_2 as follows:

(∇) The state space of the resulting NFA must contain states from nfa_1 and nfa_2. So $\nabla\ =\ \nabla\ \mathsf{nfa}_1 + \nabla\ \mathsf{nfa}_2$.

(δ) The transition function of the resulting NFA is composed of four blocks.

1. The top left block is the incidence matrix of the first NFA.
2. The top right block is null. So, no transitions from nfa_1 to nfa_2.
3. The bottom left block is null. In terms of incidence matrices this signals that there are no transitions from nfa_2 to nfa_1.
4. The bottom right block is the incidence matrix of the second NFA.

(I) The set of initial states in the resulting NFA is the union of the sets of initial states of nfa_1 and nfa_2.

(F) The set of final states in the resulting NFA is the union of the sets of final states of nfa_1 and nfa_2.

– $\mathsf{e}\ =\ \mathsf{e}'^{+}$:

$$
\begin{array}{l}
\mathsf{nfa}^{+\prime}\ =\ \textbf{record}\ \{ \\
\quad \nabla\ =\ \nabla\ \mathsf{nfa}; \\
\quad \delta\ =\ \lambda\, \mathsf{x} \to (\mathsf{id} \oplus \mathsf{F}\ \mathsf{nfa} \otimes \mathsf{I}\ \mathsf{nfa}) \otimes (\delta\ \mathsf{nfa}\ \mathsf{x}) \\
\quad \mathsf{I}\ =\ \mathsf{I}\ \mathsf{nfa}; \\
\quad \mathsf{F}\ =\ \mathsf{F}\ \mathsf{nfa}\,\}
\end{array}
$$

The difference between $\mathsf{nfa}^{+\prime}$ and nfa is in δ only. Specifically, we add a new edge from each final state to each successor of an initial state. This is achieved by $\mathsf{F}\ \mathsf{nfa} \otimes \mathsf{I}\ \mathsf{nfa} \otimes \delta\ \mathsf{nfa}\ \mathsf{x}$, where $\mathsf{I}\ \mathsf{nfa} \otimes \delta\ \mathsf{nfa}\ \mathsf{x}$ stands for edges reachable from initial state by reading the token x. And $\mathsf{F}\ \mathsf{nfa} \otimes \mathsf{I}\ \mathsf{nfa} \otimes \delta\ \mathsf{nfa}\ \mathsf{x}$ puts an edge from each final state to each successor of an initial state.

– $\mathsf{e}\ =\ \mathsf{e}_1 \cdot \mathsf{e}_2$:

$$
\begin{array}{l}
\mathsf{nfa}_1 \cdot' \mathsf{nfa}_2\ =\ \textbf{record}\ \{ \\
\quad \nabla\ =\ \nabla\ \mathsf{nfa}_1 + \nabla\ \mathsf{nfa}_2; \\
\quad \delta\ =\ \lambda\, \mathsf{x} \to [\,[\,\delta\ \mathsf{nfa}_1\ \mathsf{x} \mid \mathsf{F}\ \mathsf{nfa}_1 \otimes \mathsf{I}\ \mathsf{nfa}_2 \otimes \delta\ \mathsf{nfa}_2\ \mathsf{x}\,]\ /\ [\,\ \mathsf{null}\ \mid\ \delta\ \mathsf{nfa}_2\ \mathsf{x}\ \,]\,]; \\
\quad \mathsf{I}\ =\ [\,\mathsf{I}\ \mathsf{nfa}_1 \mid \mathsf{null}\,]; \\
\quad \mathsf{F}\ =\ [\,\mathsf{F}\ \mathsf{nfa}_1 \otimes \mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2\ /\ \mathsf{F}\ \mathsf{nfa}_2\,]\,\}
\end{array}
$$

The fields of nfa_1 and nfa_2 are combined in the following way:

(∇) The state space of the resulting NFA consists of the disjoint union of state spaces of nfa_1 and nfa_2, i.e. $\nabla\ =\ \nabla\ \mathsf{nfa}_1 + \nabla\ \mathsf{nfa}_2$.

(δ) The transition function constructs incidence matrices from four blocks.

1. The top left block contains the incidence matrix of nfa_1. Hence, the transition relation between the states of nfa_1 is not changed.
2. The top right block is $\mathsf{F}\ \mathsf{nfa}_1 \otimes \mathsf{I}\ \mathsf{nfa}_2 \otimes \delta\ \mathsf{nfa}_2\ \mathsf{x}$. This expression constructs an incidence matrix with transitions from all final states of nfa_1 to all successors of initial states in nfa_2. In other words, it says that upon reaching a final state of nfa_1, it is time to transition to nfa_2.
3. The bottom left block is null. Hence, there are no transitions from nfa_2 back to nfa_1.

4. The bottom right block consists of the transition function of nfa_2. So, the transitions between the states of nfa_2 are unchanged.

(I) The resulting NFA's initial states must contain only initial states of nfa_1.

(F) Clearly, the final states of the resulting NFA must contain all final states of the second NFA. But what if the second NFA accepts the empty string? Then the resulting NFA must also accept the language of the first NFA, hence, contain all its final states. The desired behaviour is achieved by

$$\mathsf{F}\ =\ [\ \mathsf{F}\ \mathsf{nfa}_1 \otimes \mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2\ /\ \mathsf{F}\ \mathsf{nfa}_2\ ]$$

The top block of this column vector is equal to either the empty vector or $\mathsf{F}\ \mathsf{nfa}_1$ depending on the result of $\mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2$. The latter multiplication is equal to $\mathsf{id}\ \{1\}$, if nfa_2 accepts the empty string, and null, if it does not. The bottom block of F is always equal to $\mathsf{F}\ \mathsf{nfa}_2$. So, the desired behaviour is achieved.

4.4 Correctness

The correctness of an NFA with respect to a regular expression consists of completeness and soundness. Completeness guarantees that every string matching a regular expression will be accepted by the NFA. However, completeness alone is not enough, since an NFA accepting all strings is also complete. Soundness in turn guarantees that, if the NFA accepts, the string matches the regular expression. Similarly to the case of completeness, soundness is not sufficient alone, because an NFA rejecting every string is sound.

Completeness. Completeness states that, if a string is matched by the given regular expression, then the constructed NFA $\mathsf{reg2nfa}\ \mathsf{e}$ accepts it.

$$\begin{aligned}&\mathsf{complete}\ :\ (\mathsf{e}\ :\ \mathsf{RegExp}) \to (\mathsf{u}\ :\ \mathsf{String}) \to \mathsf{u} \blacktriangleright \mathsf{e}\\ &\quad \to \mathsf{runNFA}\ (\mathsf{reg2nfa}\ \mathsf{e})\ \mathsf{u} \equiv \mathsf{id}\ \{1\}\end{aligned}$$

We prove this theorem by induction on the proof of $\mathsf{u} \blacktriangleright \mathsf{e}$. Hence, all shapes of parse trees must be considered. We describe only the cases for union and concatenation, the others can be found in the Agda code.

Union In this case $\mathsf{e}\ =\ \mathsf{e}_1 \cup \mathsf{e}_2$.

$$\mathsf{complete}\ (\mathsf{e}_1 \cup \mathsf{e}_2)\ \mathsf{u}\ \mathsf{parseTree}\ =\ \ldots$$

We start with pattern matching on $\mathsf{parseTree}$.

$$\begin{aligned}&\mathsf{complete}\ (\mathsf{e}_1 \cup \mathsf{e}_2)\ \mathsf{u}\ (\mathsf{unionl}\ \mathsf{parseTree}')\ =\ \ldots\\ &\mathsf{complete}\ (\mathsf{e}_1 \cup \mathsf{e}_2)\ \mathsf{u}\ (\mathsf{unionr}\ \mathsf{parseTree}')\ =\ \ldots\end{aligned}$$

This yields two cases for the last rule used in the parse tree: unionl or unionr. Since both cases are proved in the same way, we describe only the first one.

The main idea is to show that a run of $\mathsf{nfa}_1 \cup' \mathsf{nfa}_2$ can be split into runs of nfa_1 and nfa_2. It is proved by the lemma $\mathsf{union\text{-}split}$:

$$\begin{aligned}&\mathsf{union\text{-}split} : (\mathsf{nfa}_1\ \mathsf{nfa}_2 : \mathsf{NFA})\ (\mathsf{u} : \mathsf{String})\ (\mathsf{X}_1 : 1 \times \_)\ (\mathsf{X}_2 : 1 \times \_)\\ &\quad \to \mathsf{run}\ (\mathsf{nfa}_1 \cup' \mathsf{nfa}_2)\ \mathsf{s}\ [\ \mathsf{X}_1 \mid \mathsf{X}_2\ ] \equiv [\ \mathsf{run}\ \mathsf{nfa}_1\ \mathsf{s}\ \mathsf{X}_1 \mid \mathsf{run}\ \mathsf{nfa}_2\ \mathsf{s}\ \mathsf{X}_2\ ]\end{aligned}$$

Next, by using the previously described property

$$\begin{aligned}&\mathsf{con\text{-}\otimes\text{-}stack} : \{\mathsf{k}\ \mathsf{l}\ \mathsf{m}\ \mathsf{n} : \mathbb{N}\}\ (\mathsf{A} : \mathsf{k} \times \mathsf{l})\ (\mathsf{B} : \mathsf{k} \times \mathsf{m})\ (\mathsf{C} : \mathsf{l} \times \mathsf{n})\ (\mathsf{D} : \mathsf{m} \times \mathsf{n})\\ &\quad \to [\ \mathsf{A} \mid \mathsf{B}\ ] \otimes [\ \mathsf{C}\ /\ \mathsf{D}\ ] \equiv \mathsf{A} \otimes \mathsf{C} \oplus \mathsf{B} \otimes \mathsf{D}\end{aligned}$$

we complete the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{id}\ \{1\} \oplus \mathsf{id}\ \{1\} \equiv \mathsf{id}\ \{1\}}\ \text{Boolean arithm.}}{(\mathsf{run}\ \mathsf{nfa}_1\ \mathsf{s}\ \mathsf{X}_1 \otimes (\mathsf{F}\ \mathsf{nfa}_1)) \oplus (\mathsf{run}\ \mathsf{nfa}_2\ \mathsf{s}\ \mathsf{X}_2 \otimes (\mathsf{F}\ \mathsf{nfa}_2)) \equiv \mathsf{id}\ \{1\}}\ \text{IHs}}{[\ \mathsf{run}\ \mathsf{nfa}_1\ \mathsf{s}\ \mathsf{X}_1 \mid \mathsf{run}\ \mathsf{nfa}_2\ \mathsf{s}\ \mathsf{X}_2\ ] \otimes [\ \mathsf{F}\ \mathsf{nfa}_1\ /\ \mathsf{F}\ \mathsf{nfa}_2\ ] \equiv \mathsf{id}\ \{1\}}\ \text{con-}\otimes\text{-stack}}{\mathsf{run}\ (\mathsf{nfa}_1 \cup' \mathsf{nfa}_2)\ \mathsf{s}\ [\ \mathsf{X}_1 \mid \mathsf{X}_2\ ] \otimes [\ \mathsf{F}\ \mathsf{nfa}_1\ /\ \mathsf{F}\ \mathsf{nfa}_2\ ] \equiv \mathsf{id}\ \{1\}}\ \text{union-split}$$

Plus In this case $\mathsf{e} = \mathsf{e'}^{+}$.

$$\mathsf{complete}\ (\mathsf{e'}^{+})\ \mathsf{u}\ \mathsf{parseTree} = \ldots$$

Pattern matching on parseTree yields two cases. We examine them in turn:

1. In the first case the last rule of the parse tree is plus1.

   $$\mathsf{complete}\ (\mathsf{e'}^{+})\ \mathsf{u}\ (\mathsf{plus1}\ \mathsf{parseTree'}) = \ldots$$

   Recall that plus1 is a constructor which states that, if u is in language of e′, then it is also in language of $(\mathsf{e'}^{+})$. Hence, the main lemma for this case can be stated as

   $$\begin{aligned}&\mathsf{plus\text{-}weak} : (\mathsf{nfa} : \mathsf{NFA})\ (\mathsf{u} : \mathsf{String})\ (\mathsf{X} : 1 \times (\nabla\ \mathsf{nfa}))\\ &\quad \to \mathsf{run}\ \mathsf{nfa}\ \mathsf{u}\ \mathsf{X} \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}\\ &\quad \to \mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}\ \mathsf{X} \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}\end{aligned}$$

   It is proved by induction on the length of the string u.
2. In the second case the last rule of the parse tree is plus2.

   $$\mathsf{complete}\ (\mathsf{e'}^{+})\ .\ (\mathsf{u}_1 +\!\!+ \mathsf{u}_2)\ (\mathsf{plus2}\ \{\mathsf{u}_1\}\ \{\mathsf{u}_2\}\ \mathsf{tree}_1\ \mathsf{tree}_2) = \ldots$$

   Note that string u is now split into u_1 and u_2 such that u_1 is in the language of e′ and u_2 is in the language of $\mathsf{e'}^{+}$. We must prove that $\mathsf{u}_1 +\!\!+ \mathsf{u}_2$ are in the language of $\mathsf{e'}^{+}$. To do so, we first introduce some useful lemmas.
   - We show that run on $\mathsf{u}_1 +\!\!+ \mathsf{u}_2$ can be split into a run on u_1 and a run on u_2.

     $$\begin{aligned}&\mathsf{plus\text{-}split} : (\mathsf{nfa} : \mathsf{NFA})\ (\mathsf{u}_1\ \mathsf{u}_2 : \mathsf{String})\\ &\quad \to \mathsf{run}\ (\mathsf{nfa}^{+\prime})\ (\mathsf{u}_1 +\!\!+ \mathsf{u}_2)\ (\mathsf{I}\ \mathsf{nfa}) \otimes \mathsf{F}\ \mathsf{nfa}\\ &\qquad \equiv \mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}_2\ (\mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}_1\ (\mathsf{I}\ \mathsf{nfa})) \otimes \mathsf{F}\ \mathsf{nfa}\end{aligned}$$

   - We also show that, if the automaton $\mathsf{nfa}^{+\prime}$ accepts a string from the initial states, then it will also accept that string from any final state.

     $$\begin{aligned}&\mathsf{plus\text{-}fin} : (\mathsf{nfa} : \mathsf{NFA})\ (\mathsf{u} : \mathsf{String})\ (\mathsf{X} : 1 \times \nabla\ \mathsf{nfa})\\ &\quad \to \mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}\ (\mathsf{I}\ \mathsf{nfa}) \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}\\ &\quad \to \mathsf{X} \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}\\ &\quad \to \mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}\ \mathsf{X} \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}\end{aligned}$$

Finally, the big picture of the proof looks like this:

$$\dfrac{\dfrac{}{\mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}_2\ (\mathsf{I}\ \mathsf{nfa}) \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}}\ \text{IH} \quad \dfrac{\dfrac{}{\mathsf{run}\ \mathsf{nfa}\ \mathsf{u}_1\ (\mathsf{I}\ \mathsf{nfa}) \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}}\ \text{IH}}{\mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}_1\ (\mathsf{I}\ \mathsf{nfa}) \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}}\ \text{plus-weak}}{\dfrac{\mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}_2\ (\mathsf{run}\ (\mathsf{nfa}^{+\prime})\ \mathsf{u}_1\ (\mathsf{I}\ \mathsf{nfa})) \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}}{\mathsf{run}\ (\mathsf{nfa}^{+\prime})\ (\mathsf{u}_1\ +\!\!+\ \mathsf{u}_2)\ (\mathsf{I}\ \mathsf{nfa}) \otimes \mathsf{F}\ \mathsf{nfa} \equiv \mathsf{id}\ \{1\}}\ \text{plus-split}}\ \text{plus-fin}$$

Soundness. Showing that our NFA generation is sound is more complicated, but also more interesting. Let us look at the signature of the soundness theorem:

```
sound : (e : RegExp) → (u : String)
  → runNFA (reg2nfa e) u ≡ id {1} → u ▶ e
```

It states that, if the NFA accepts a string, then it matches the regular expression. sound is a proposition, but it is also a type! Its proof is a function that delivers parse trees. We prove this theorem by induction on the argument e : RegExp.

As in case of completeness, our aim is to explain the high-level ideas of the proof. We skip most of the details and describe only two cases.

Single character. This case is interesting because it demonstrates the essence of all soundness cases. We are given an accepting run of the automaton. Using this fact we must construct a parse tree. However, most of the cases generated by pattern matching are discharged by showing that they contradict with the accepting run we have at our disposal.

```
sound (′ a) u run = ...
```

We pattern match on the string u and examine three different cases in turns:

1. u *is the empty string.*

   ```
   sound (′ a) u run = ...
   ```

   We must show that in this case it is impossible to give run. We do so by pattern matching on run and the rest is taken care of by Agda's type checker.
2. u *is a string of one symbol.*

   ```
   sound (′ a) (x :: [ ]) run = ...
   ```

   This is the only situation when the automaton can accept u. Still, we must check if x is equal to a. We case analyse on the decidable equality of a and x:

   ```
   sound (′ a) (x :: [ ]) run with a ≟ x
   sound (′ a) (x :: [ ]) | eq = ...
   sound (′ a) (x :: [ ]) | neq = ...
   ```

   Then two cases must be discharged.

   (a) x *is equal to* a.
   This is exactly the case when the automaton finishes in the accepting state. To close this case, we rewrite the context using a ≡ x and provide the constructor symb {x} as the required proof.

(b) x *is not equal to* a.
Then Agda computes that runNFA (reg2nfa (′ a)) [x] is equal to null, but this contradicts the assumptions. Hence, the case is discharged.

3. u *is a string of two or more characters.*

```
sound (′ a) (x₁ :: x₂ :: xs) run = ...
```

Similarly to the first case, our goal is to show that runNFA (reg2nfa (′ a)) u will never accept a string consisting of two or more characters. This is done by observing the fact that, even if the automaton reaches the second state by reading the first character, then by reading the second character the automaton will lose all active states, since there are no transitions going out of the second state. Hence, runNFA (reg2nfa (′ a)) (x₁ :: x₂ :: xs) cannot return id {1}. Therefore, the case is discharged.

Concatenation. We are in the case

```
sound (e₁ · e₂) u run = ...
```

Fortunately, there is only one possible constructor for this case in the parsing relation, namely con. It states that to prove u ▶ e₁ · e₂ we must show that u₁ ▶ e₁ and u₂ ▶ e₂ for some splitting of u into u₁ and u₂. Hence, we must be able to extract from run two shorter runs and use them to get u₁ ▶ e₁ and u₂ ▶ e₂ by induction hypothesis. To express this in Agda we use sigma-types, corresponding to existentials.

```
cons-split : (nfa₁ nfa₂ : NFA) (u : String)
  → run (nfa₁ ·′ nfa₂) u [ I nfa₁ | null ]
          ⊗ [ F nfa₁ ⊗ I nfa₂ ⊗ F nfa₂ / F nfa₂ ] ≡ id {1}
  → ∃ [u₁ : String] ∃ [u₂ : String] u ≡ u₁ ++ u₂
          ∧ run nfa₁ u₁ (I nfa₁) ⊗ F nfa₁ ≡ id {1}
          ∧ run nfa₂ u₂ (I nfa₂) ⊗ F nfa₂ ≡ id {1}
```

Note that we split the string u into u₁ and u₂, but must also provide a proof that u ≡ u₁ ++ u₂. To prove it, we will need a variant of cons-split.

```
cons-split-state : (nfa₁ nfa₂ : NFA) (x : Σ) (u : String) (X : 1 × _)
  → run (nfa₁ ·′ nfa₂) (x :: u) [ X | null ] ⊗ (null ++ F nfa₂) ≡ id {1}
  → ∃ [u₁ : String] ∃ [u₂ : String] (x :: u) ≡ u₁ ++ u₂
          ∧ run nfa₁ u₁ X ⊗ F nfa₁ ≡ id {1}
          ∧ run nfa₂ u₂ (I nfa₂) ⊗ F nfa₂ ≡ id {1}
```

The important differences between cons-split and cons-split-state are the following:

- In cons-split, the given run starts from [I nfa₁ | null], where I nfa₁ is the set of initial states of nfa₁, but in cons-split-state, we use a more general variant [X | null], where X is a given parameter.
- In cons-split, the run of the automaton can terminate either in the final states of nfa₁ or in the final states of nfa₂, but in cons-split-state, the set of final states is limited to those of nfa₂.

– In contrast with cons-split-state, the string xs can be empty in cons-split.

The restrictions present in cons-split-state force it to address the specific and most complicated case when the given run starts in some states of nfa_1, but terminates in nfa_2. The lemma states that in this case we can break an accepting run of $\mathsf{nfa}_1 \cdot' \mathsf{nfa}_2$ down into two smaller accepting runs.

We will not describe here how cons-split-state is proved. Instead we show how to reduce cons-split to cons-split-state. To do so, we perform multiple case analyses. First, we distinguish the empty string case from the cons-case.

1. If $\mathsf{u} \equiv [\,]$, then $\mathsf{u}_1 \equiv [\,]$ and $\mathsf{u}_2 \equiv [\,]$ and we must show that $\mathsf{I}\ \mathsf{nfa}_1 \otimes \mathsf{F}\ \mathsf{nfa}_1 \equiv \mathsf{id}\ \{1\}$ and $\mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2 \equiv \mathsf{id}\ \{1\}$. Both proofs are easily derived from the given premise:

$$[\ \mathsf{I}\ \mathsf{nfa}_1 \mid \mathsf{null}\ ] \otimes [\ \mathsf{F}\ \mathsf{nfa}_1 \otimes (\mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2)\ /\ \mathsf{F}\ \mathsf{nfa}_2\ ] \equiv \mathsf{id}\ \{1\}$$

2. If $\mathsf{u} \equiv \mathsf{x} :: \mathsf{xs}$, then we perform an additional case analysis: whether the second automaton has final states among its initial states:
   (a) $\mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2 \equiv \mathsf{null}$. In this case, the problem is clearly an instance of cons-split-state.
   (b) $\mathsf{I}\ \mathsf{nfa}_2 \otimes \mathsf{F}\ \mathsf{nfa}_2 \equiv \mathsf{id}\ \{1\}$. In this case, we perform a third level of case analysis: whether the first automaton nfa_1 accepts the whole string.
      i. $\mathsf{run}\ \mathsf{nfa}_1\ \mathsf{u}\ (\mathsf{I}\ \mathsf{nfa}_1) \otimes \mathsf{F}\ \mathsf{nfa}_1 \equiv \mathsf{id}\ \{1\}$.
      If we take $\mathsf{u}_1 \equiv \mathsf{u}$ and $\mathsf{u}_2 \equiv [\,]$, then case is immediately discharged.
      ii. $\mathsf{run}\ \mathsf{nfa}_1\ \mathsf{u}\ (\mathsf{I}\ \mathsf{nfa}_1) \otimes \mathsf{F}\ \mathsf{nfa}_1 \equiv \mathsf{null}$.
      In this case, we need an additional lemma

$$\begin{aligned}&\mathsf{consnd2} : (\mathsf{nfa}_1\ \mathsf{nfa}_2 : \mathsf{NFA})\ (\mathsf{u} : \mathsf{String})\\ &\quad \to (\mathsf{run}\ \mathsf{nfa}_1\ \mathsf{u}\ (\mathsf{I}\ \mathsf{nfa}_1)) \otimes \mathsf{F}\ \mathsf{nfa}_1 \equiv \mathsf{null}\\ &\quad \to (\mathsf{run}\ (\mathsf{nfa}_1 \cdot' \mathsf{nfa}_2)\ \mathsf{u}\ [\ \mathsf{I}\ \mathsf{nfa}_1 \mid \mathsf{null}\ ]) \otimes [\ \mathsf{F}\ \mathsf{nfa}_1\ /\ \mathsf{F}\ \mathsf{nfa}_2]\\ &\quad \equiv (\mathsf{run}\ (\mathsf{nfa}_1 \cdot' \mathsf{nfa}_2)\ \mathsf{u}\ [\ \mathsf{I}\ \mathsf{nfa}_1 \mid \mathsf{null}\ ]) \otimes [\ \mathsf{null}\ /\ \mathsf{F}\ \mathsf{nfa}_2]\end{aligned}$$

      It states that, if the first automaton does not accept the whole string, then running the automaton $\mathsf{nfa}_1 \cdot' \mathsf{nfa}_2$ with the final states of nfa_1 is equivalent to running the automaton $\mathsf{nfa}_1 \cdot' \mathsf{nfa}_2$ without the final states of nfa_1. So, this branch of cons-split is also reduced to cons-split-state.

sound returns one parse tree. If there are multiple parse trees for a single string, it prefers unionl over unionr and also plus1 over plus2. It also never invokes plus2 with the first string empty, as in this case no progress is made. In fact makes sense to restrict the first string argument of plus2 to be a cons-string—this removes the possibility for a string to have an infinite number of parse trees.

4.5 Parsing

Correctness of reg2nfa turns the NFA for a regular expression effectively into a parser. Since we can decide whether a 1×1 matrix contains true or false, using sound and complete, for any string we can have a parse tree or a proof of that there cannot be one.

$$\mathsf{parse} : (\mathsf{e} : \mathsf{RegExp}) \to (\mathsf{u} : \mathsf{String}) \to \mathsf{u} \blacktriangleright \mathsf{e} \uplus (\mathsf{u} \blacktriangleright \mathsf{e} \to \bot)$$

5 Related Work

Braibant and Pous [1] implement a Coq tactic for deciding equational theory of Kleene algebras. The work is based on checking if two regular expressions represent the same language. This is done in four steps. First, regular expressions are converted into ε-NFAs. Then ε-transitions are removed to get NFAs. Next, determinisation procedure converts NFAs into DFAs. Finally, they check whether the DFAs are equivalent. This results in a general decision procedure for Kleene algebras. In principle, it can be used to solve the recognition problem: to check whether a word w is in the language defined by a regular expression r, we can check if $\mathsf{w} \cup \mathsf{r}$ and r define the same language. But as this requires go through all four steps for each query, it is impractical.

In contrast, we focus only on the recognition problem. The main difference of our work is that we convert regular expressions directly into NFAs without ε-transitions. This makes the overall process simpler, since then we do not have to find ε-closures and remove ε-transitions afterward and pepper all that, as Braibant and Pous confirm, with quite tricky proofs of correctness.

Many works formalizing recognition of regular languages are based on the concept of the derivative of a language [5,2,3,7]. This is not accidental, since derivatives have nice algebraic properties which make them attractive for a formal development.

It seems that the alternative approach of converting regular expressions to finite automata is believed to be a messy procedure with too much low-level detail involved. For instance, Krauss and Nipkow [5] discuss the link between regular expressions and finite automata in the context of lexing, but point out that encoding finite automata as graphs involves a painful amount of detail and a higher-level approach is desirable.

Wu et al. [9] show how to formalize the Myhill-Nerode theorem by only using regular expressions and the motivation behind this approach is again to avoid the trouble of representing automata as graphs.

We have shown that conversion from regular expressions to finite state automata encoded as Boolean matrices can be done in a concise and high-level way by using block operations on matrices. Proofs in this setting benefit significantly from lemmas about block operations.

6 Conclusion

We presented an implementation of a certified parser generator for regular languages. In particular we showed how to reduce operations and proofs about NFAs into linear-algebra operations and proofs. The practical part of this work was divided into two parts. In the first part, we implemented a generic library for matrices, focusing on block operations. Besides an implementation of basic matrix operations, we also proved many well-known properties of these functions.

In the second part of the practical work, we implemented a transformation of regular expressions to NFAs and proved its correctness. A string is parsed by checking whether the NFA accepts it, as soundness turns the positive answer into a parse tree while completeness can be used to conclude impossibility of a parse tree in the negative case.

This work could be continued in several directions. The implemented framework (the matrix library, RegExp and NFA libraries) can be used to formalize different aspects of regular language theory: minimizing NFAs, showing equivalence of regular expressions, conversion of NFAs to regular expressions, etc.

One variation on the theme of this work would be to consider matrices over natural numbers instead of Booleans. This would allow counting of accepting runs of an NFA (paths from an initial to a final state). Soundness and completeness of the transformation of regular expressions to NFAs would establish a bijection between the parse trees of a given string and the accepting runs of the NFA.

Acknowledgements. This work was supported by the ERDF funded CoE project EXCS, the Estonian Ministry of Education and Research target-financed theme no. 0140007s12 and the Estonian Science Foundation grant no. 9475.

References

1. Braibant, T., Pous, D.: An efficient Coq tactic for deciding Kleene algebras. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 163–178. Springer, Heidelberg (2010)
2. Coquand, T., Siles, V.: A decision procedure for regular expression equivalence in type theory. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 119–134. Springer, Heidelberg (2011)
3. Danielsson, N.A.: Total parser combinators. In: Proc. of 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2010, pp. 285–296. ACM (2010)
4. Grune, D.: Parsing Techniques: A Practical Guide, 2nd edn. Springer (2010)
5. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. J. of Autom. Reasoning 49(1), 95–106 (2012)
6. Macedo, H.D., Oliveira, J.N.: Typing linear algebra: A biproduct-oriented approach. Sci. of Comput. Program. 78(11), 2160–2191 (2013)
7. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: better, faster, stronger SFI for the x86. In: Proc. of 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2012, pp. 395–404. ACM (2012)
8. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
9. Wu, C., Zhang, X., Urban, C.: A formalisation of the Myhill-Nerode theorem based on regular expressions (Proof pearl). In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 341–356. Springer, Heidelberg (2011)