Semester project report

Bence Czipó

January 24, 2018

1 Introduction

Formal languages are abstract mathematical structures, sets of string generated from a finite alphabet.

Definition (Alphabet). An *alphabet* A is a finite set of symbols $\{a_1, a_2, a_3, \dots a_n\}$ ▪

1.1 Words

Definition (Words). A *string* (or *word*) is a sequence of symbols. The length of a word is the number of symbols in the sequence. ▪

For words, we can define the following operations.

- **Length:** Let $\text{len}(w)$ denote the number of characters in w.
- **Indexing:** For a word w and an integer $i \in 0..\text{len}(w) - 1$ $w[i]$ denote the i'th character of w (starting from 0).
- **Range indexing:** For a word w let $w[i : j]$ denote a word such that $\text{len}(w[j : i]) = j - i + 1$ $w[i : j][0] = w[i], w[i : j][1] = w[i + 1], \dots w[i : j][j - i] = w[j]$. For notation, lets define $w[: i] = w[0 : i]$ and $w[i :]$ as $w[i : \text{len}(w) - 1]$.
- **Concatenation:** Let $w_c = w_1 \cdot w_2$ be a word, such that $\text{len}(w_c) = \text{len}(w_i) + \text{len}(w_2)$ and $w_c[: \text{len}(w_1) - 1] = w_1$ and $w_c[\text{len}(w_1) :] = w_2$. In some cases for simplification, we will notate word concatenation with just the sequentiality of the two words, and leave the $\cdot$ character.

1.2 Languages

Let A^i denote the set of the words created from the symbols of A that has length i.

Let A^* denote $A^0 \cup A^1 \cup \dots$, which in other words is the set of finite sequences created by the symbols of the alphabet.

Definition (Formal language). We call the set$L$ a *formal language* if $L \subseteq A^*$ ▪

Note that the definition of A^* allows the *empty word* (which is a symbol sequence of length 0). The empty word is denoted by ϵ

We can define two distinguished languages, that exists over every alphabet.

Definition (Empty language). *Empty language* is a language that does not contain any word, so $L_0 = \emptyset$ ▪

Definition (Unit language). *Unit language* is a language, that contains only one word, ϵ. So in other words, $L_\epsilon = \{\epsilon\}$ ▪

1.2.1 Operations Over Languages

Since languages are set of words, we can define basic set operations such as

- union
- intersection
- subtraction
- complement (with respect to A^*)
- inclusion ($L_1 \subseteq L_2$)

We can also define a concatenation operation for languages, that takes all the words from the first language and appends all of them to the second language. Formally, let $L_1 \subseteq A^*$ and $L_2 \subseteq A^*$ be two languages. The concatenation of two languages is $L_1 \cdot L_2 = \{u_1u_2 \mid u_1 \in L_1, u_2 \in L_2\}$

With the notion of concatenation, we can also define power of languages. The n'th power of a language Lis L combined to itself n-1 times. Formally, we can define L^n inductively: let $L^0 = \{\epsilon\}$ and $L^{i+1} = L^i \cdot L$.

We can also define the *Kleene star* of a language as $L^* = \bigcup_n L^n$. Or we can rephrase it as $L^* = \{w_1 \dots w_n \mid n \geq 0, \forall i \in [1, n].w_i \in L\}$.

This approach is quite similar to the implementation with Sets, it also ensures uniqueness, and the values, for which the function is true can not be enumerated. However, with functions we can implement infinite languages, on the other hand with the other two approach we can only model languages with arbitrary many words.

1.3 Regular Expressions

Languages can be constructed using regular expressions. A regular expression can contain the following constants:

- The empty language $\emptyset$.
- The unit language $\{\epsilon\}$ (denoted by simply ϵ)
- A language of one word $\{w\}$ (denoted by w)

It defines the following operations.

- **Concatenation** of sets of words.
- **Union** of sets of words.
- **Kleene star** of a set of words.

2 Implementation

In order to verify theorems about languages in Stainless, they have to be implemented in Scala.

This section first introduces a way to represent symbols and words in Stainless. Later, I present and compare different approaches to implement languages. They can be represented in various ways, in this work I highlight three of them.

1. Languages are set of words, so they can be represented through a Set of List of T-s.
2. Languages can be represented as a unique List of List of T-s.
3. They can also be represented with a function, that maps List[T]-s to boolean values, true if the word is in the language and false otherwise

Note that in order to be able to efficiently verify theorems in Stainless, we have to use the collection implementations of Stainless that are located in the packages `stainless.lang` and `stainless.collection`. From now on, whenever Set or List are mentioned, they refer to `stainless.lang.Set` and `stainless.collection.List`.

2.1 Representing Symbols and Words

In languages, symbols are frequently characters, however, by definition they can be any type, so they are represented as generic type `T`. Words are ordered, not unique collections of symbols, thus they are represented as a `List[T]`. With the usage of Lists, the other operation intuitively come:

- ϵ can be represented as `Nil[T]`,
- $\text{len}(w)$ can be represented as `w.size`
- indexing can be represented using the indexing operator of lists,
- range indexing can be implemented combining `take` and `drop`
- concatenation of words w_1 and w_2 can be expressed as `w1 ++ w2`

2.2 Representing Languages with Set

Languages are set of words, so using Set of words seems to be the most convenient way to represent them.

This way, all basic set operations can be implemented by calling the corresponding method of Set. Moreover, uniqueness is also ensured, because the used data structure ensures it.

The drawback of this solution is that Set does not have any functions that allows us to iterate through its elements.

```
case class Lang[T](set: Set[List[T]]) {

  def concat(that: Lang[T]): Lang[T] = ???

  def ++(that: Lang[T]):Lang[T] = Lang[T](this.set ++ that.set)

  def contains(word: List[T]): Boolean = set.contains(word)

  [...]

}
```

Listing 1: Sketch of a class representing Languages using sets

2.3 Representing Languages with Lists

To overcome that, we can represent languages with unique lists, as sketched in Listing 2.

This implementation has the advantage that items can be iterated through (e.g. applying structural induction on the list), however, in this case we have to deal with the issue that words in languages can have arbitrary order. Moreover, with list operations, uniqueness is not ensured. This can be worked around two ways:

1. For each operation we require the input lists to have unique words in a total order, and with our implementation we ensure that the resulting lists will also be unique, and its words will follow the same total order.
2. We state our theorems not for the list, but for their content, which has the type Set. This way, uniqueness in ensured, and the order of items does not matter any more. The big advantage of this approach is that if two lists are equal (structurally) their contents are also equal. This means that sometimes it can be enough to prove a stricter, but more easily provable theorem.

The implementation sketch for the second approach is presented in Listing 2 uses the second approach.

2.4 Representing Languages with Functions

The third approach is to represent a language with a function, that maps words (lists of symbols) to boolean values. For each language $L \in A^*$ we can define a function f_L such that a word w over A

$$f_L(w) = \begin{cases} \text{true} & \text{if } w \in L \\ \text{false} & \text{otherwise} \end{cases}$$

```scala
case class Lang[T](list: List[List[T]]) {

  def appendToAll( l: List[List[T]],
                   suffix: List[T] ):List[List[T]] = l match {
    case Nil() => Nil[List[T]]()
    case Cons(x,xs) => (x ++ suffix)::appendToAll(xs, suffix)
  }

  def concatLists(
          l1: List[List[T]],
          l2: List[List[T]]):  List[List[T]] = l2 match {
    case Nil() => Nil[List[T]]()
    case Cons(x,xs) => appendToAll(l1,x)++combineLists(l1,xs)
  }

  def concat(that: Lang[T]): Lang[T] = {
    Lang[T](concatLists(this.list, that.list))
  }

  def ++(that: Lang[T]): Lang[T] = (
    Lang[T](this.list ++ that.list)
  )

  def == (that: Lang[T]): Boolean = {
    this.list.content == that.list.content
  }

  def contains(word: List[T]): Boolean = list.contains(word)

  [...]

}
```

Listing 2: Implementing Languages using sets

One of the challenges with this implementation that the set operations are not always trivial. To define the combination of L_1 and L_2, we have to define a function $f_{L_1 \cdot L_2}$. One definition for such function is the following: $f_{L_1 \cdot L_2}(w) = \exists i \in 0..\text{len}(w).\ w[: i-1] \in L_1 \wedge w[i :] \in L_2$, informally, this means that the word can be split into two parts, such that the first part is in L_1 and the second is in L_2.

The sketch for such implementation is presented in Listing 3.

Note that due to the current limitations of the Stainless framework, instead of $\exists$ we have to

```scala
case class Lang[T](f: List[T] => Boolean) {

  def concat(that: Lang[T]): Lang[T] = {
    Lang[T](l => !forall( (i: BigInt) => !(
      i <= l.size &&
      i >= 0 &&
      this.contains(l.take(i)) &&
      that.contains(l.drop(i))
    )))
  }

  def ++(that: Lang[T]): Lang[T] = {
    Lang[T](w => this.f(w) || that.f(w))
  }

  def == (that: Lang[T]): Boolean = {
    forall((x:List[T]) => contains(x) ==> that.contains(x)) &&
    forall((x:List[T]) => that.contains(x) ==> contains(x))
  }

  def contains(word: List[T]): Boolean = f(word)
}
```

Listing 3: Implementing languages using functions

use $\forall$, applying the fact that for a predicate p we have $\exists(p) = !\forall(!p)$

2.5 Comparison

The comparison of different ways of implementation can be found in Table 1.

For my work, I selected the non-unique/ordered list representation of languages. The core motivation behind that decision was that this way languages can be iterated using structural induction, even though this way theorems get a bit more complicated than in the other cases. The other main reason for that was that this way, the words are not required to have a total order, unlike the unique and ordered case.

2.6 Regular Expressions

In the implementation of regular expressions I differed from their original definition. For sake of simplicity lets restrict them to allow only two constants to appear in the regular

	Sets	Lists (unique and ordered)	Lists (content)	Functions
Unique	Yes	Yes	No	Yes
Iterable (structural induction)	No	Yes	Yes	No
Can express infinite languages	No	No	No	Yes
Equality of languages	Trivial	Content equality	Content equality	Complex, $\forall$ expressions
Concatenation of languages	Complex (no structural induction on sets)	Structural induction applied twice (combination will be unique)	Structural induction applied twice	Complex
Word containment	Trivial	Trivial	Trivial	Trivial
Set operations ($\cup$, $\cap, \ldots$)	Trivial	Uniqueness has to be ensured	Trivial	Trivial
Phrasing theorems and lemmas	Trivial	Trivial	Theorems about content of list	Trivial

Table 1: Comparison of different ways of implementation

expressions, but they can be any language L_1, L_2. Let the set of such regular expression be denoted by R_{L_1,L_2}.

Constants and operations are defined as case classes. `L1` and `L2` has no arguments, `Union` and `Concat` take two regular expressions, whereas `Pow` (that replaces the star in the implementation) takes a regular expression and a natural number.

The regular expression implementation has a method
`eval(l1: Lang[T], l2: Lang[T]): Lang[T]` that takes two languages as parameters and evaluates the value of the regular expression with respect to it.

3 Theorems

In this section I will present the theorems that I proved for formal languages using Stainless, and the sketch of their implementation. Note that this enumeration is not an exhaustive list, some helper lemmas might not be presented. In some cases, theorem proofs are omitted because their simplicity or analogousness to other proofs, or because they would require too much space. For the list and proofs of all theorems, see the source code, available on GitHub.

3.1 Extending the implementation

In this section, all theorems and stainless proofs are phrased and presented considering an implementation with not unique and not ordered Lists. A sketch for such implementation has been presented in Section 2.3, however it misses some functionality. This section details the methods for this implementation and introduces notations and abbreviations used in this report.

3.1.1 Equalities

In the sketch, we used the `==` operator to check if two languages are the same.However, overriding such operator can cause unexpected behavior in Stainless, and for this reason, we need to define our own operator for that.

```
def sameAs(that: Lang[T]): Boolean = {
  this.list.content == that.list.content
}
```

3.1.2 Set operations

The sketch presented the union operation `++` and the containment operation `contains`. For some cases, we might like to add an other single word to the language, which can be done with the `::` operator.

```
def ::(t: List[T]): Lang[T] = Lang[T](t :: this.list)
```

An other convenient is inclusion, to check if a language is contained in an other.

```
def subsetOf(that: Lang[T]): Boolean =
                  this.list.content subsetOf that.list.content
```

Finally, it could be also benefitial to not only allow adding new elements, but also allow removing elements from the language. For this reason lets introduce the `--` operator.

```
def --(that: Lang[T]): Boolean = Lang[T](this.list--that.list)
```

3.1.3 Language operations

The implementation for concatenation of languages has already been presented. However, for efficiency, we can define more operations.

The power of a language can be defined the following way:

```
def ^ (i: BigInt): Lang[T] = i match {
  case BigInt(0) => Lang[T](List(Nil()))
  case _ => this concat (this ^ (i-1))
}
```

Note that this operation unfolds to the right. We could have defined it in an other way, as

```
def :^ (i: BigInt): Lang[T] = i match {
  case BigInt(0) => Lang[T](List(Nil()))
  case _ =>  (this ^ (i-1)) concat this
}
```

We will later show that these two operations are equivalent but until then, lets keep both definitions.

Having defined the power of a language, we can take a step towards having the star of a language by defining the n-close of a language. First lets define it formally the following way:

Definition (Close of a language). Let L be a language. The n'th close of the language can be defined as $L^{(n)} = \bigcup_{i=0}^{n} L^i$ ▪

This definition has to really useful properties:

- if $w \in L^{(n)}$ then $\forall n' \geq n.\ w \in \text{close}_{n'}(L)$ and $w \in L^*$
- if $w \in L^*$ then $\exists n.\ w \in L^{(n)}$

In other words, if we would like to prove that a word is in the star of a language, it is enough to construct an integer n such that the word will be the n'th close of the language. The big advantage compared to the power of languages is that in this case we only have to tell an upper bound for j for which the word is in L^j.

Such function can be implemented the following way:

```
def close(i: BigInt): Lang[T] = i match {
  case BigInt(0) => this ^ i
  case _ => (this close (i-1)) ++ (this ^ i)
}
```

3.1.4 Helper methods

In order to be able to efficiently define the language operations, some helper methods have been already introduced. Namely `concatList` which in functionality is basically the same as the concat operation, just it is for lists not languages. The *appendToAll* function takes a word and a language and appends the word to the end of each word in that language.

Intuitively, as we defined `appendToAll`, we might want to define a similar operation `prependToAll`, which prepends a single word to any other words in a language.

```
def prependToAll[T](prefix:List[T],
                    l:List[List[T]]):List[List[T]] = l match {
  case Nil() => Nil[List[T]]()
  case hd::tl => (prefix ++ hd) :: prependToAll(prefix, tl)
}
```

3.2 Notations

Writing long proofs using operation names and strict syntax can be space demanding. For sake of clearness and easier understanding, lets introduce some notations.

From now on, lets use `==` for language equality, and `===` for strict equality, namely marking the equality of the representing lists.

Instead of `++` and `--` operators, use the corresponding set operations, $\cup$ and $\setminus$.

Also let $w \in L$ denote the `contains` function, and use $\subseteq$ as a shorthand for `subsetOf`. For concatenation, we can use the $\cdot$ operator, or to make it more function-like, we can use $cl(L_1, L_2)$. Since *concat* and *concatLists* are functionally really similar, we can use the same notation for both. Lets also abbreviate the function names, `appendToAll` as aa and `prependToAll` as pa.

We can also define notations for the types. We can get rid of the type parameter T, since all operations are only valid on languages and words over the same alphabet. Let `Word` denote `List[T]` and let `Lang` denote `Lang[T]`. For the sake of simplicity lets also suppose that all list operations can be invoked on languages, without accessing the underlying list (so for a language `l` and a list method `f(...)` let `l.f(...)` denote `l.list.f(...)`).

Also for notation, let $\emptyset$ denote the empty language, which is with this representation is equivalent to `Nil[List[T]]()`. The empty word with this representation would be `Nil[T]()` so the unit language $\{\epsilon\}$ is `List(Nil[T]())`

3.3 Theorems about words

Words are list of symbols, thus many theorems about them hold, because they hold for any kind of list. It is not easy to see that since a symbol can be literally anything, every theorem that holds for a word constructed from symbols must hold for lists of any type.

3.3.1 Empty word concatenation

We can state a theorem that the concatenation of an empty word is an identity operation.

Theorem 1 *Let w be a word. In this case*

- $w \cdot \epsilon = w$
- $\epsilon \cdot w = w$

We can also state some kind of associativity for word concatenation:

Theorem 2 *Let w_1, w_2, w_3 be words over the same alphabet A. In that case,* $(w_1 \cdot w_2) \cdot w_3 = w_1 \cdot (w_2 \cdot w_3)$

The corresponding theorems and proofs for all the above are implemented in Stainless, in `stainless.collection.ListSpecs`.

Theorem 3 (Cancellation Laws) *For words w_1, w_2, w_3 over the same alphabet A, we have*

- *if* $w_1 \cdot w_2 = w_1 \cdot w_3$ *then* $w_2 = w_3$
- *if* $w_2 \cdot w_1 = w_3 \cdot w_1$ *then* $w_2 = w_3$

3.4 Theorems About Languages

As it was mentioned in Section 2.5, I used list of words to represent a language, so whenever a theorem is phrased, we actually phrase it of a list of list of T, or more precisely, since I chose the non-unique variant, about the content of such lists of lists.

3.4.1 Combination to the Empty Language

Theorem 4 (Null Combination) *Any language concatenated to the empty language results in the empty language, formally,*

- $\forall L.\ L \cdot \emptyset = \emptyset$
- $\forall L.\ \emptyset \cdot L = \emptyset$

Converting this theorems into stainless format is straightforward, however, as a reference, this time it is included.

Actually, we can convert them differently:

```
def rightNullConcat[T](l: Lang[T]): Boolean = {
  l.concat(nullLang[T]()) sameAs nullLang[T]()
}.holds

or

def rightNullConcat[T](l1: Lang[T]): Boolean = {
  forall( (l: Lang[T]) =>
      l.concat(nullLang[T]()) sameAs nullLang[T]()
  )
}.holds

or directly on lists of lists

def rightNullConcat[T](l: List[List[T]]): Boolean = {
   concatLists(l, Nil[List[T]]()).content ==
                                       Nil[List[T]]().content
}.holds
```

The difference between the first and the second is that when we will later reference a theorem (or lemma) in a proof of an other theorem, in the first case we can exactly state for which language should the solver apply the theorem, which enhances the performance. Comparing the first and the third, they are basically the same, and if Stainless can not prove a theorem, eventually we will have to give hints to the solver based on the underlying data structures, which in this case is a list.

Recall that in the implementation presented in Section 2.3, we applied structural induction on the right hand operand of the concatenation. For this reason the first case is straightforward, because the operation will yield an empty list.

Note that in this case, we proved the equality of the lists, that are representing the languages, instead of proving the equality of their content. However it is straightforward that if two lists are identical, their content are the same.

Similarilly, we can state the second part of the theorem:

```
def leftNullConcat[T](l: Lang[T]): Boolean = {
  (nullLang[T]()).concat(l) sameAs nullLang[T]()
}.holds
```

Stainless can verify the theorem above without further aid, however we can also prove it by hand. If L is an empty list, the proof is just as straightforward as in the first case. On the other hand if it has some elements, we can apply induction.

```
cl(∅, hd::tl) ==
aa(∅, hd) ++ cl(∅, tl) ==
∅ ++ cl([∅], tl) ==
cl(∅, tl) == //by the induction hypothesis
∅
```

3.4.2 Structural Theorems and Lemmas

The previous theorems were straightforward from the definition, Stainless can prove them without a hint. However, in order to prove more complex theorems, we have to define some helper lemmas and theorems.

First of all, we would like to show, that even though we defined list concatenation with appending, we could have defined it with perpending, namely that

Lemma 1 *For any languages L_1, L_2 over the same alphabet, where $L_1 = hd1 :: tl1$ we have cl(hd1::tl1, L_2) == pa(hd1,L_2) ++ cl(tl1,L_2)*

If L_2 is $\emptyset$, we can apply the previous theorem and have `cl((hd1::tl1),∅) =∅` and `[pa(hd,∅)] =∅` and `cl(tl,∅) =∅`, and the proof is straightforward.

Otherwise, we can apply induction. We know that L_2= `hd2 :: tl2`, so we have

```
cl(hd1::tl1, hd2::tl2) ===
//by definition
aa(hd1::tl1, hd2) ++ cl(hd1::tl1, tl2) ===
//by definition
[hd1 ++ hd2] ++ aa(tl1, hd2) ++ cl(hd1::tl, tl2)  ===
// by the I.H.
[hd1 ++ hd2] ++ aa(tl1, hd2) ++ pa(hd1, tl2) ++ cl(tl1, tl2)
```

Similarly, we can also show that

```
[hd1++hd2] ++ pa(hd1, tl2) ++ aa(tl1, hd2) ++ cl(tl1, tl2) ===
pa(hd1, hd2 :: tl2) ++ aa(tl1, hd2) ++ cl(tl1, tl2) ===
pa(hd1, hd2 :: tl2) ++ cl(tl1, hd2 :: tl2)
```

See that in the two claims above, we used strict equality instead of set equality, so we can apply that if for two languages $L_1 === L_2$ then $L_1 == L_2$.

So we can state that:

```
cl(hd1::tl1, hd2::tl2) ==
[hd1++hd2] ++ aa(tl1, hd2) ++ pa(hd1, tl2) ++ cl(tl1, tl2) ==
[hd1++hd2] ++ pa(hd1, tl2) ++  aa(tl1, hd2) ++ cl(tl1, tl2) ==
pa(hd1, hd2 :: tl2) ++ cl(tl1, hd2 :: tl2)
```

The proof of this lemma can be found in the code in function `clInductLeft`.

Intuitively, we can generalize the following lemma to some kind of distributivity.

Lemma 2 (Left Distributivity) *For any languages* L_1, L_2, L_3 *we have* $(L_1{+}{+}L_2) \cdot L_3 == L_1 \cdot L_3 + +L_2 \cdot L_3$.

Lemma 3 (Right Distributivity) *For any languages* L_1, L_2, L_3 *we have* $(L_1 + +L_2) \cdot L_3 == L_1 \cdot L_3 + +L_2 \cdot L_3$.

The proof for these two lemmas are not included, but in can be seen that they can be devised inductively. They are implemented in functions `clLeftDistributiveAppend` and `clRightDistributiveAppend`.

3.4.3 Concatenation to the Unit Language

Theorem 5 (Unit Concatenation (Right)) $\forall L \subseteq A^*$. *we have* $L \cdot \{\epsilon\} = L$

If L is an empty language, the solution is straightforward, we can apply the previous theorem, $\emptyset \cdot \{\epsilon\} = \emptyset$. Otherwise, we can apply structural induction on `L = hd :: tl` the following way:

```
cl(hd::tl, {ϵ})              == // LM1
pa(hd, {ϵ}) ++ cl(tl, {ϵ})  == // LM2
[hd] ++ cl(tl, {ϵ})          == // by the I.H.
[hd] ++ tl                   ==
L
```

Note that the step marked with LM1 applies Lemma 1. The LM2 step is almost trivial to see, however Stainless can't prove it on its own. For the complete proof see the function `prependToEmptyList`. The proof of the theorem can be found in function `rightUnitConcat`.

Theorem 6 (Unit Concatenation (Left)) $\forall L \subseteq A^*$. *we have* $\{\epsilon\} \cdot L = L$

For this theorem, the solution is even easier. If L is an empty language, the solution is also straightforward, we have, $\{\epsilon\} \cdot \emptyset = \emptyset$.

If not, we know that `L == hd :: tl` so we can write

```
cl([ϵ], hd::tl) ==
aa([ϵ], hd) ++ cl([ϵ], tl) == // Theorem~\ref{the:emptyword}
[hd] ++ cl([ϵ], tl) == //by the I.H.
[hd] ++ tl ==
L
```

The implementation for this theorem is proved in the function `leftUnitConcat`.

3.4.4 Operations on Equivalent Languages

Since different lists can represent the same language, it is not straightforward that applying the same operation on the same languages leads to the same result. We can, however, state some lemmas that will be useful in the next sections.

First of all, define that if two languages are the same, concatenating them to the same language yields the same language.

Lemma 4 *Let* L_1, L_2, L_3 *be three languages in the same alphabet, and let* $L_1 == L_2$*. Then* $L_1 \cdot L_3 == L_2 \cdot L_3$

We can also state the lemma for the other case, where the left hand side operator is fixed.

Lemma 5 *Let* L_1, L_2, L_3 *be three languages in the same alphabet, and let* $L_2 == L_3$*. Then* $L_1 \cdot L_1 == L_1 \cdot L_3$

The proof for these lemmas are not included, but shown in functions `clContentEquals` and `clContentEquals2` and in other, referenced sub-lemmas.

3.4.5 Associativity

We have seen that for every $L \in A^*$, $L \cdot \{\epsilon\} = \{\epsilon\} \cdot L = L$. Now the question arises if $(A^*; \cdot)$ is a monoid. For this, we have to prove associativity of languages over the concatenation.

Before that, lets define a helper lemma that will help us to prove associativity.

Lemma 6 *For any word w and languages L_1 and L_2 we have $pa(w, L_1) \cdot L_2 == pa(w, L_1 \cdot L_2)$*

The lemma can be proved applying structural induction, and a proof for it can be found in the code under the function `replaceConcatPrepend`.

Theorem 7 (Associativity) *For every $L_1, L_2, L_3 \subseteq A^*$ we have $(L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$*

A convenient proof for the theorem would be to show that each element in one language is contained in the other language. So if there exists an injection from one set to the other, and the other set to the first, then the two are identical. However, Stainless is more efficient when something is proved with structural induction so we will show associativity with structural induction.

We will apply induction on L_1, first lets examine the case when it is $\emptyset$. In this case on the left hand side of the equality, we have $(\emptyset \cdot L_2) \cdot L_3 = \emptyset \cdot L_3 = \emptyset$ and on the right hand side we have $\emptyset \cdot (L_2 \cdot L_3) = \emptyset$, so the two are equal.

Otherwise, there is some hd and tl such that $L_1 = hd :: tl$.

Now we can apply the following steps:

$((hd :: tl) \cdot L_2) \cdot L_3$ `== // (1)`
$(pa(hd, L_2) \cup (tl \cdot L_2)) \cdot L_3$ `== // (2)`
$(pa(hd, L_2) \cdot L_3) \cup ((tl \cdot L_2) \cdot L_3)$ `== // (3)`
$(pa(hd, L_2) \cdot L_3) \cup (tl \cdot (L_2 \cdot L_3))$ `== // (4)`
$pa(hd, L_2 \cdot L_3) \cup (tl \cdot (L_2 \cdot L_3))$ `== // (5)`
$(hd :: tl) \cdot (L_2 \cdot L_3)$

The explanation of each step is the following:

1. We apply Lemma 1 and 4. The first ensures that the two left hand sides are the same and the second ensures that the results of the combination are the same.
2. We apply that concatenation is distributive over union, as stated in Lemma 2.
3. We apply the induction step, and apply the theorem for the right hand side.
4. We apply the associativity in small. This step was proved in Lemma 6.
5. We apply Lemma 1 once more.

3.4.6 Theorems about subset operation

First, lets present some lemmas that are trivial for stainless.

Lemma 7 (Reflexivity) *Let L_1, L_2 be two languages such that $L_1 == L_2$. In this case, we know that $L_1 \subseteq L_2$.*

Lemma 8 (Transitivity) *Let L_1, L_2, L_3 be three languages. If $L_1 \subseteq L_2$ and $L_2 \subseteq L_3$ then $L_1 \subseteq L_3$.*

Applying these, we can state two new lemmas:

Lemma 9 *Let L_1, L_2, L_3 be three languages. If $L_1 \subseteq L_2$ and $L_2 == L_3$ then $L_1 \subseteq L_3$.*

Lemma 10 *Let L_1, L_2, L_3 be three languages. If $L_1 == L_2$ and $L_2 \subseteq L_3$ then $L_1 \subseteq L_3$.*

We can use the subset operation to prove equivalence of languages.

Lemma 11 (Subset-Equivalence) *For all languages L_1, L_2 such that $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$ we have $L_1 == L_2$*

We can also state, that if a language is subset to an other, than there has to be a language such that if we take the union of that and the smaller language, we get the bigger one.

Lemma 12 *For all languages L_1, L_2 such that $L_1 \subseteq L_2$ there is a language L_3 such that $(L_1 \cup L_3) == L_2$.*

However, instead of using the existential quantifier, it would be easier to simply specify L_3 as $L_2 \setminus L_1$.

We can also state some subset lemmas with the union operation.

Lemma 13 *For all languages L_1, L_2 we have $L_1 \subseteq (L_1 \cup L_2)$ and $L_2 \subseteq (L_1 \cup L_2)$*

We can also say that if two languages are separately subset of an other, their union will also be subset of that language.

Lemma 14 *Let L_1, L_2, L_3 be three languages. If $L_1 \subseteq L_3$ and $L_2 \subseteq L_3$ then $(L_1 \cup L_2) \subseteq L_3$*

Based on these, we can state the following lemmas.

Lemma 15 *Let L_1, L_2, L_3 be three languages. If $L_1 \subseteq L_2$ then $L_1 \cdot L_3 \subseteq L_2 \cdot L_3$.*

Stainless can not prove this lemma on its own, but we can combine previous lemmas to show that it actually holds.

$L_1 \cdot L_3 \subseteq$ `//(1)`
$(L_1 \cdot L_3) \cup ((L_2 \setminus L_1) \cdot L_3)$ `==` `//(2)`
$(L_1 \cup (L_2 \setminus L_1)) \cdot L_3$ `==` `//(3)`
$L_2 \cdot L_3$

In step 1) we applied Lemma 13, in step 2) we used distributivity as stated in Lemma 3 and in step 3) we applied Lemma 12. Eventually we have to apply Lemma 9 to show that subset relation is transitive through equality.

The following lemma can be proved similarly:

Lemma 16 *Let L_1, L_2, L_3 be three languages. If $L_2 \subseteq L_3$ then $L_1 \cdot L_2 \subseteq L_1 \cdot L_3$.*

The proofs for the previous two lemmas are presented in functions `concatSubset` and `concatSubset2`.

3.4.7 Theorems about the power of languages

We defined the power operation in two different ways, one that unfolds to the left and one that unfolds to the right. Having proved associativity, we feel that eventually, these two definition are the same.

Theorem 8 (Power definition equality) *For all language L and $i \in \mathbb{N}$ we have $L\,\hat{}\;i == L\,\hat{}{:}\;i$*

To prove this theorem we have to separate three cases.

If i is 0, by definition both sides will be $\{\epsilon\}$.

If i is 1 then

```
L^1 == //by definition
L·(L^0) == //by definition
L·{ϵ} == // (1)
L == //(2)
{ϵ}·L == //by definition
(L:^0)·L == //by definition
L:^1
```

In step 1) we applied Theorem 5, and in step 2) we applied Theorem 6.

Otherwise, we can apply induction. We know that $i \geq 2$.

```
L·(L^i-1) == // (1)
L·(L:^i-1) == //(2)
L·((L:^i-2)·L) == //(3)
(L·(L:^i-2))·L == //(4)
(L·(L^i-2))·L == //(5)
(L^i-1) ·L == //(6)
(L:^i-1)·L == //(7)
L:^1
```

During the proof, we applied induction three times, in steps (1), (4) and (6). We used associativity (Theorem 7) in step (3). In all other steps the proof only uses the definition and the lemmas about combining equivalent languages (Lemmas 4 and 5). Note that the termination is alway ensured as we always in each induction step, the value of i decreases. Furthermore, since in this case we know that $i \geq 2$, in the recursive calls $i \geq 0$ is ensured.

For the implementation of the proof, see `couldHaveDefinedOtherWay`.

From now on, lets simply refer to both as L^i.

We can see that any language to the power of zero is a unit language just like in the case of the power of real numbers. But we can state more lemmas similar to that:

Lemma 17 (First power of languages) *For any language L we have $L^1 == L$.*

The proof for this lemma can be seen in function `langToFirst` and it is really similar to the $i = 1$ case in Theorem 8.

Lemma 18 (Power of unit language) *For any $i \in \mathbb{N}$ we have $\{\epsilon\}^i == \{\epsilon\}$*

The proof for this can be shown using induction and is presented in function `unitLangPow`.

An other useful lemma would be to show that sums in the exponent can be expanded into concatenations. However, for now it is enough to prove something weaker, for only two languages.

Lemma 19 (Language to the sum) *For any language L and numbers $a, b \in \mathbb{N}$ we have $L^{a+b} == (L^b) \cdot (L^b)$.*

The proof for this lemma can be shown applying induction and associativity, as it is implemented in function `powSum`.

3.4.8 Theorems About the Close of Languages

We defined the close operator the get closer to the notion of star. With this implementation, we can not implement infinite languages, but we can say that a word $w \in L^*$ iff. $\exists i \in \mathbb{N}.w \in L^{(i)}$

We can also state some lemmas concerning the close of empty and unit languages.

Lemma 20 (Close of Empty Language) *For every $i \in \mathbb{N}$ we have $\emptyset^{(i)} == \{\epsilon\}$*

Lemma 21 (Close of Unit Language) *For every $i \in \mathbb{N}$ we have $\{\epsilon\}^{(i)} == \{\epsilon\}$*

These two can be proved easily using induction, as shown in the code at functions `nullLangClose` and `unitLangClose`.

A useful property of close that $L^{(i)} \subseteq L^{(j)}$ is equivalent to $i \leq j$.

Lemma 22 *$\forall L.\ L^{(i)} \subseteq L^{(j)}$ iff. $i \leq j$.*

This can also be proved inductively, as shown in function `subsetCloseLe`.

Finally, we would like to define something like Lemma 19 for the close operation.

Lemma 23 *For every language L and $a, b \in \mathbb{N}$ we have $L^{(a+b)} == L^{(a)} \cdot L^{(b)}$*

However, proof of this would be really difficult, so instead, lets just use a weaker form of this lemma.

Lemma 24 *For every language L and $a, b \in \mathbb{N}$ we have $L^{(a)} \cdot L^{(b)} \subseteq L^{(a+b)}$*

For the proof of the latter see function `sumClose` in the code.

3.5 Theorems About Regular Expressions

We would like to show that for our regular expression implementation (as defined in Section 2.6) some properties hold. Recall that the implementation only allows two languages, L_1 and L_2. We would like to prove that $\forall r \in R_{L_1,L_2}$. if a language L is defined by r then $L \in (L_1 \cup L_2)^*$. Eliminate the star from that expression!

Theorem 9 *For every regular expression r defined over the languages L_1, L_2, if L is defined by r, $\exists i \in \mathbb{N}.\ L \subseteq (L_1 \cup L_2)^{(i)}$*

In order to prove this, for each regular expression we have to guess the exponent i. For this, we have to define an other method `evalExp():BigInt`. The value of it:

- for `L1` and `L2` is 1,
- for `Union(r1,r2)` is `max(r1.evalExp(), r2.evalExp())`,
- for `Conc(r1,r2)` is `r1.evalExp() + r2.evalExp()`
- for `Pow(r,n)` is `r.evalExp() * n`

We can state a lemma nearly equivalent to the previous theorem:

Lemma 25 *For every regular expression r defined over the languages L_1, L_2, if L is defined by r, if $i =$ `r.evalExp()` then $L \subseteq (L_1 \cup L_2)^{(i)}$*

For `L1` and `L2` it is trivial that $i = 1$ is sufficient because of Lemma 13.

For `Union(r1,r2)` we can say that $(L_1 \cup L_2)^{(a)} \subseteq (L_1 \cup L_2)^{(max(a,b))}$ and $(L_1 \cup L_2)^{(b)} \subseteq (L_1 \cup L_2)^{(a,b)}$ (a,b are `r1.evalExp()` and `r2.evalExp()`). From there we can apply the subset distributivity over union (Lemma 14) and with induction and subset transitivity (Lemma 8) we can prove the statement.

For `Concat(r1,r2)` we chose `evalExp = r1.evalExp() + r2.evalExp()`. Here the proof is similar to the previous case, but now we use Lemma 24 about the union of closes being contained in the close of sum. Since we only want to prove inclusion, the weaker (and proved) for of the lemma is sufficient.

We can say that `Pow(r,n)` can be expressed as
`Conc(r, Conc(r, Conc(..., Conc(r, Pow(r,1)) ...)))` which is
`Conc(r, Conc(r, Conc(..., Conc(r, r) ...)))`, so this is only a syntactic sugar in regular expressions. For this reason, lets omit this case in the proofs.

Even though the proof is sound in paper, its Stainless implementation is not yet implemented completely, some steps are not verifying. The initial version of the proof is included in function `regexSubsetStar` in the file `RegEx.scala`.

For the previous proof to be correct, we have to show that while taking the close of a language we do not perform an invalid operation, namely each exponent is nonnegative.

Lemma 26 *For each regular expression r we have* `r.evalExp() > 0`*.*

The proof for this lemma is trivial as for the constant values the exponent is positive, and both maximum selection and addition (of positive numbers) preserves positiveness.

4 Performance

All lemmas and theorems are shown to be verifying and terminating except for the ones presented in Section 3.5. The total time required for verification on a pc with Dual-Core Intel® Core™ i5-4210U CPU @ 1.70GHz processor and 8GB memory is 1670.178s with the longest time for a single step being 241.744s. For termination, the total time required on the same architecture is 1936.337s, with the longest step lasting for 613.622s (`couldHaveDefinedOtherWay`).