Verification of Formal Languages

Bence Czipó

École Polytechnique Fédérale de Lausanne

bence.czipo@epfl.ch

January 25, 2018

Introduction

- Formal Languages
- Regular Expressions
- Deterministic Finite Automaton

Overview

Formal Languages

Definitions

Definition (Alphabet)

An *alphabet* A is a finite set of symbols $\{a_1, a_2, a_3, \ldots a_n\}$

Definitions

Definition (Alphabet)

An *alphabet* A is a finite set of symbols $\{a_1, a_2, a_3, \dots a_n\}$

Definition (Words)

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

Definitions

Definition (Alphabet)

An *alphabet* A is a finite set of symbols $\{a_1, a_2, a_3, \ldots a_n\}$

Definition (Words)

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

Definition (Formal language)

Let A^i denote the set of the words created from the symbols of A that has length i.
Let A^* denote $A^0 \cup A^1 \cup \ldots$, which in other words is the set of finite sequences created by the symbols of the alphabet.
We call the set L a *formal language* if $L \subseteq A^*$

Operations

- **union** $(L_1 \cup L_2)$
- **subtraction** $(L_1 \setminus L_2)$
- **inclusion** $(L_1 \subseteq L_2)$

Operations

- **union** $(L_1 \cup L_2)$
- **subtraction** $(L_1 \setminus L_2)$
- **inclusion** $(L_1 \subseteq L_2)$
- **concatenation** $(L_1 \cdot L_2)$

Definition (Concatenation)

Let $L_1 \subseteq A^*$ and $L_2 \subseteq A^*$ be two languages. The concatenation of two languages is $L_1 \cdot L_2 = \{u_1 u_2 \mid u_1 \in L_1, u_2 \in L_2\}$

Operations

- **union** $(L_1 \cup L_2)$
- **subtraction** $(L_1 \setminus L_2)$
- **inclusion** $(L_1 \subseteq L_2)$
- **concatenation** $(L_1 \cdot L_2)$
- **power** (L^i)

Definition (Power of languages)

let $L^0 = \{\epsilon\}$ and $L^{i+1} = L \cdot L^i$

Operations

- **union** $(L_1 \cup L_2)$
- **subtraction** $(L_1 \setminus L_2)$
- **inclusion** $(L_1 \subseteq L_2)$
- **concatenation** $(L_1 \cdot L_2)$
- **power** (L^i)
- **Kleene star** (L^*)

Definition (Kleene star)

$$L^* = \{w_1 \ldots w_n \mid n \geq 0, \forall i \in [1, n]. w_i \in L\}$$

Lemma

$$L^* = \bigcup_n L^n$$

Operations

- **union** $(L_1 \cup L_2)$
- **subtraction** $(L_1 \setminus L_2)$
- **inclusion** $(L_1 \subseteq L_2)$
- **concatenation** $(L_1 \cdot L_2)$
- **power** (L^i)
- **Kleene star** (L^*)
- complement (with respect to A^*)
- intersection $(L_1 \cap L_2)$

Last two are not part of the implementation

Distinguished Languages

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, $L_\epsilon = \{\epsilon\}$

Representation in Scala

- Symbols: $\rightarrow$ `Any` (T - generic type)
- Words $\rightarrow$`List[T]`

Representation in Scala

- Symbols: $\rightarrow$ `Any` (T - generic type)
- Words $\rightarrow$`List[T]`
  - 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`

Representation in Scala

- Symbols: $\rightarrow$ `Any` (T - generic type)
- Words $\rightarrow$`List[T]`
  - ϵ can be represented as `Nil[T]`,
  - 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`
- Languages $\rightarrow$???

Representation in Scala

- Symbols: $\rightarrow$ `Any` (T - generic type)
- Words $\rightarrow$`List[T]`
    - ϵ can be represented as `Nil[T]`,
    - 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`
- Languages $\rightarrow$
    - `Set[Words]`
    - `Words -> Boolean`
    - Unique (and ordered) `List[Words]`
    - `List[Words]`

Languages as Set

```
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)
 [...]

}
```

Languages as Function

```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.f(l.take(i)) && that.f(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]) =>  this.f(x) == that.f(x))

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

Languages as List (non-unique)

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

  [...]
}
```

Languages as List (non-unique)

```
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 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)
}
```

Comparison

	Sets	Lists - 1	Lists - 2	Functions
Unique	Yes	Yes	**No**	Yes
Iterable	No	<u>Yes</u>	<u>Yes</u>	No
Infinity	No	No	No	<u>Yes</u>
Equality	Trivial	Content $=$	Content $=$	$\forall$ **expr**
Concat	**???**	<u>S.I. 2x</u>	<u>S.I. 2x</u>	**Complex**
Contain	Trivial	Trivial	Trivial	Trivial
Set ops ($\cup$, $\cap$, $\ldots$)	Trivial	**Uniqueness**	Trivial	Trivial
Lemmas	Trivial	Trivial	**About cont**	Trivial

Extending implementation with List

How to implement Kleene star?

Extending implementation with List

How to implement Kleene star?

Theorem

L^* is only finite if $L = \emptyset$ or $L = \{\epsilon\}$.

We can not express it using finite language representations.

Extending implementation with List

How to implement Kleene star?

Theorem

L^* is only finite if $L = \emptyset$ or $L = \{\epsilon\}$.

We can not express it using finite language representations.

Definition (Close)

Let $n \in \mathbb{N}$. The n'th close of the language can be defined as $L^{(n)} = \bigcup_{i=0}^{n} L^i$.

Extending implementation with List

How to implement Kleene star?

Theorem

L^* is only finite if $L = \emptyset$ or $L = \{\epsilon\}$.

We can not express it using finite language representations.

Definition (Close)

Let $n \in \mathbb{N}$. The n'th close of the language can be defined as $L^{(n)} = \bigcup_{i=0}^{n} L^i$.

Lemma

If $w \in L^*$ then $\exists n \in \mathbb{N}$. such that $w \in L^{(n)}$

Theorems and Lemmas About Languages

Distributivity

Theorem (Left Distributivity)

For any languages L_1, L_2, L_3 *we have* $(L_1 \cup L_2) \cdot L_3 == L_1 \cdot L_3 \cup L_2 \cdot L_3$.

Theorem (Right Distributivity)

For any languages L_1, L_2, L_3 *we have* $(L_1 \cup L_2) \cdot L_3 == L_1 \cdot L_3 \cup L_2 \cdot L_3$.

Concatenation special cases

Theorem (Null Concatenation - Right)

$\forall L \subseteq A^*.\ L \cdot \emptyset = \emptyset$

Concatenation special cases

Theorem (Null Concatenation - Right)

$\forall L \subseteq A^*.\ L \cdot \emptyset = \emptyset$

Theorem (Null Concatenation - Left)

$\forall L \subseteq A^*.\ \emptyset \cdot L = \emptyset$

Concatenation special cases

Theorem (Null Concatenation - Right)

$\forall L \subseteq A^*.\ L \cdot \emptyset = \emptyset$

Theorem (Null Concatenation - Left)

$\forall L \subseteq A^*.\ \emptyset \cdot L = \emptyset$

Theorem (Unit Concatenation - Right)

$\forall L \subseteq A^*.\ we\ have\ L \cdot \{\epsilon\} = L$

Concatenation special cases

Theorem (Null Concatenation - Right)

$\forall L \subseteq A^*.\ L \cdot \emptyset = \emptyset$

Theorem (Null Concatenation - Left)

$\forall L \subseteq A^*.\ \emptyset \cdot L = \emptyset$

Theorem (Unit Concatenation - Right)

$\forall L \subseteq A^*.\ we\ have\ L \cdot \{\epsilon\} = L$

Theorem (Unit Concatenation - Left)

$\forall L \subseteq A^*.\ we\ have\ \{\epsilon\} \cdot L = L$

Equivalence Lemmas

Same operation performed on the same languages.

Equivalence Lemmas

Same operation performed on the same languages.

Due to the selected implementation, equivalence is not always trivial.

Equivalence Lemmas

Same operation performed on the same languages.

Due to the selected implementation, equivalence is not always trivial.

Lemma

Let L_1, L_2, L_3 be three languages over 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

Let L_1, L_2, L_3 be three languages over the same alphabet, and let $L_2 = L_3$. Then $L_1 \cdot L_2 = L_1 \cdot L_3$

Associativity

Theorem (Associativity)

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

Proof.

$\square$

Associativity

Theorem (Associativity)

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

Proof.

1. $((\{hd\} \cup tl) \cdot L_2) \cdot L_3 =$ applying the distributive law

$\square$

Associativity

Theorem (Associativity)

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

Proof.

1. $((\{hd\} \cup tl) \cdot L_2) \cdot L_3 =$ applying the distributive law
2. $((\{hd\} \cdot L_2) \cup (tl \cdot L_2)) \cdot L_3 =$ applying the distributive law

$\square$

Associativity

Theorem (Associativity)

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

Proof.

1. $((\{hd\} \cup tl) \cdot L_2) \cdot L_3 =$ applying the distributive law
2. $((\{hd\} \cdot L_2) \cup (tl \cdot L_2)) \cdot L_3 =$ applying the distributive law
3. $((\{hd\} \cdot L_2) \cdot L_3) \cup ((tl \cdot L_2) \cdot L_3) =$ applying induction

$\square$

Associativity

Theorem (Associativity)

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

Proof.

1. $((\{hd\} \cup tl) \cdot L_2) \cdot L_3 =$ applying the distributive law
2. $((\{hd\} \cdot L_2) \cup (tl \cdot L_2)) \cdot L_3 =$ applying the distributive law
3. $((\{hd\} \cdot L_2) \cdot L_3) \cup ((tl \cdot L_2) \cdot L_3) =$ applying induction
4. $((\{hd\} \cdot L_2) \cdot L_3) \cup (tl \cdot (L_2 \cdot L_3)) =$ cheating

$\square$

Associativity

Theorem (Associativity)

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

Proof.

1. $((\{hd\} \cup tl) \cdot L_2) \cdot L_3 =$ applying the distributive law
2. $((\{hd\} \cdot L_2) \cup (tl \cdot L_2)) \cdot L_3 =$ applying the distributive law
3. $((\{hd\} \cdot L_2) \cdot L_3) \cup ((tl \cdot L_2) \cdot L_3) =$ applying induction
4. $((\{hd\} \cdot L_2) \cdot L_3) \cup (tl \cdot (L_2 \cdot L_3)) =$ cheating
5. $(\{hd\} \cdot (L_2 \cdot L_3)) \cup (tl \cdot (L_2 \cdot L_3)) =$ applying the distributive law

$\square$

Associativity

Theorem (Associativity)

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

Proof.

1. $((\{hd\} \cup tl) \cdot L_2) \cdot L_3 =$ applying the distributive law
2. $((\{hd\} \cdot L_2) \cup (tl \cdot L_2)) \cdot L_3 =$ applying the distributive law
3. $((\{hd\} \cdot L_2) \cdot L_3) \cup ((tl \cdot L_2) \cdot L_3) =$ applying induction
4. $((\{hd\} \cdot L_2) \cdot L_3) \cup (tl \cdot (L_2 \cdot L_3)) =$ cheating
5. $(\{hd\} \cdot (L_2 \cdot L_3)) \cup (tl \cdot (L_2 \cdot L_3)) =$ applying the distributive law
6. $(\{hd\} \cup tl) \cdot (L_2 \cdot L_3)$

$\square$

Cheating?

Lemma

For any word w and languages L_1 and L_2 we have
$(\{w\} \cdot L_1) \cdot L_2 = \{w\} \cdot (L_1 \cdot L_2)$

Proof.

Applying induction once more... □

Theorems About Power of Languages

Definition of the power operation only states $L^0 = \{\epsilon\}$ explicitly.

This is similar to the case in real numbers where $a^0 = 1$.

Can we have a rule like $a^1 = a$

Theorems About Power of Languages

Definition of the power operation only states $L^0 = \{\epsilon\}$ explicitly.

This is similar to the case in real numbers where $a^0 = 1$.

Can we have a rule like $a^1 = a$

Lemma (First power of languages)

For any language L we have $L^1 = L$.

Theorems About Power of Languages

Definition of the power operation only states $L^0 = \{\epsilon\}$ explicitly.

This is similar to the case in real numbers where $a^0 = 1$.

Can we have a rule like $a^1 = a$

Lemma (First power of languages)

For any language L we have $L^1 = L$.

Proof.

$L^1 =$ (by definition)
$L \cdot (L^0) =$ (by definition)
$L \cdot \{\epsilon\} =$ (by unit concatenation lemma)
L □

Theorems About Power of Languages

We also know that $1^n = 1$, but what about languages?

Lemma (Power of unit language)

For any $i \in \mathbb{N}$ we have $\{\epsilon\}^i = \{\epsilon\}$

Proof.

Applying the definition, induction on i and after the unit combination lemma, we get: $\{\epsilon\}^i = \{\epsilon\} \cdot \{\epsilon\}^{i-1} = \{\epsilon\} \cdot \{\epsilon\} = \{\epsilon\}$ $\square$

Theorems About Power of Languages

The decision that power unfolds to the left was ad-hoc

It could have been defined the other way

Theorem (Power definition equality)

For all language L and $i \in \mathbb{N}$ we have $L^i = L \cdot L^{i-1} = L^{i-1} \cdot L$

Proof.

- case $i = 0$: trivial
- case $i = 1$: trivial (applying first power lemma)
- case $i > 1$: apply induction on i, or simply unfold and apply associativity $i - 2$ times

$\square$

Theorems About Power of Languages

Lemma (Language to the sum)

For any language L and numbers $a, b \in \mathbb{N}$ we have $L^{a+b} = (L^b) \cdot (L^b)$.

Proof.

Apply induction on a (or b) $\square$

Theorems About Close of Languages

Lemma (Close of Empty Language)

For every $i \in \mathbb{N}$ *we have* $\emptyset^{(i)} = \{\epsilon\}$

Theorems About Close of Languages

Lemma (Close of Empty Language)

For every $i \in \mathbb{N}$ *we have* $\emptyset^{(i)} = \{\epsilon\}$

Lemma (Close of Unit Language)

For every $i \in \mathbb{N}$ *we have* $\{\epsilon\}^{(i)} = \{\epsilon\}$

Theorems About Close of Languages

Lemma (Close of Empty Language)

For every $i \in \mathbb{N}$ *we have* $\emptyset^{(i)} = \{\epsilon\}$

Lemma (Close of Unit Language)

For every $i \in \mathbb{N}$ *we have* $\{\epsilon\}^{(i)} = \{\epsilon\}$

Lemma (Close Order)

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

And so on...

Theorems About Close of Languages

We would also like to define something similar to "Language to the sum"

Lemma

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

Theorems About Close of Languages

We would also like to define something similar to "Language to the sum"

Lemma

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

Does it even hold?

Theorems About Close of Languages

We would also like to define something similar to "Language to the sum"

Lemma

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

Does it even hold?

Yes, but it would be hard to prove, and something weaker will be enough

Lemma

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

Regular Expressions

"Official" definition

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 (denoted by $\cdot$ or sequentiality)
- **Union** of sets of words (denoted by $|$)
- **Kleene star** (repetition) of a set of words (denoted by *)

Example

*(abc)*d(e|f)*, matches for example *"abcde"*, *"abcabcdf"* and *"de"*

Implementation

How to represent RegExes in Scala?

Instead of these constant use any language as building block, but limit their number to 2

Have a function
`eval[T](l1: Lang[T], l2: Lang[T]): Lang[T]` to evaluate the value

Use case classes

```
sealed abstract class RegEx {
  //evaluate the regular expression to a language
  def eval[T](l1: Lang[T], l2: Lang[T]): Lang[T]
}
```

Case Classes

```
case class L1() extends RegEx {
  override def eval[T](...): Lang[T] = l1
}
case class L2() extends RegEx {
  override def eval[T](...): Lang[T] = l2
}

case class Union(l:RegEx, r:RegEx) extends RegEx {
  override def eval[T](...): Lang[T] =
                    l.eval(...) ++ r.eval(...)
}

case class Conc(l:RegEx, r:RegEx) extends RegEx {
  override def eval[T](...): Lang[T] =
                   l.eval(...) concat r.eval(...)
}
```

Dealing with Star

- Still can not represent infinite languages
- We could apply the close-trick
- Only a syntactic sugar for finite union of pows

```
case class Pow(r:RegEx, n:BigInt) extends RegEx {
  override def eval[T](...): Lang[T] =
           r.eval(...) ^ n
}
```

Theorems and Lemmas

Theorems

Theorem

For every regular expression r defined over the languages L_1, L_2, if r evaluates to L, then $L \subseteq (L_1 \cup L_2)^*$

Still can not handle * properly

Theorems

Theorem

For every regular expression r defined over the languages L_1, L_2, if r evaluates to L, then $L \subseteq (L_1 \cup L_2)^*$

Still can not handle * properly

Lemma

For every regular expression r defined over the languages L_1, L_2, if r evaluates to L, then $\exists i \in \mathbb{N}.\ L \subseteq (L_1 \cup L_2)^{(i)}$

Theorems

Theorem

For every regular expression r defined over the languages L_1, L_2, if r evaluates to L, then $L \subseteq (L_1 \cup L_2)^*$

Still can not handle * properly

Lemma

For every regular expression r defined over the languages L_1, L_2, if r evaluates to L, then $\exists i \in \mathbb{N}.\ L \subseteq (L_1 \cup L_2)^{(i)}$

Try to construct such i manually

Finding exponent

Lets define a function `evalExp(): BigInt` such that

- for `L1` and `L2` it is 1

Finding exponent

Lets define a function `evalExp(): BigInt` such that

- for `L1` and `L2` it is 1
- for `Union(l,r)` it is `max(l.evalExp(), r.evalExp())`

Finding exponent

Lets define a function `evalExp(): BigInt` such that

- for `L1` and `L2` it is 1
- for `Union(l,r)` it is `max(l.evalExp(), r.evalExp())`
- for `Conc(l,r)` it is `l.evalExp() + r.evalExp()`

Finding exponent

Lets define a function `evalExp(): BigInt` such that

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

Finding exponent

Lets define a function `evalExp(): BigInt` such that

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

Lemma

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

Proving suitability - Constants

Case `L1` and `L2` it is trivial because of the following lemma.

Lemma

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

Proving suitability - Union

Case `Union` we can say that $(L_1 \cup L_2)^{(a)} \subseteq (L_1 \cup L_2)^{(max(a,b))}$ and $(L_1 \cup L_2)^{(b)} \subseteq (L_1 \cup L_2)^{(max(a,b))}$ because of the "Close order lemma" $(a \leq max(a, b)$ and $b \leq max(a, b))$

a,b are `r1.evalExp()` and `r2.evalExp()`

Lemma (Distributivity of subset)

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

Lemma (Transitivity of subset)

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 this two we can prove the statement

Proving suitability - Concatenation

Recall the following lemma:

Lemma

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

Case `Conc` the proof is similar to the previous case, but now we use the lemma above. Since we only want to prove inclusion, the weaker (and proved) form of the lemma is sufficient.

Proving suitability - Power

`Pow(r,n)` can be expressed as:

`Conc(r, Conc(r, Conc(..., Conc(r, Pow(r,1)) ...)))`
Which is equivalent to:

`Conc(r, Conc(r, Conc(..., Conc(r, r) ...)))`

So power is only a syntactic sugar in regular expressions

We can deduce it from the previous case, where we get $\sum_{i=1}^{n}$ r.evalExp() $=$ n * r.evalExp()

Last missing piece

Lemma

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

Proof.

- In case of the constants this is trivial as $1 > 0$

$\square$

Last missing piece

Lemma

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

Proof.

- In case of the constants this is trivial as $1 > 0$
- In case of `Union(l,r)`, we can apply induction. We know that `l.evalExp()` ≤ 0 and `r.evalExp()` ≤ 0. The maximum of two non-negative numbers will be non-negative.

□

Last missing piece

Lemma

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

Proof.

- In case of the constants this is trivial as $1 > 0$
- In case of `Union(l,r)`, we can apply induction. We know that `l.evalExp()` ≤ 0 and `r.evalExp()` ≤ 0. The maximum of two non-negative numbers will be non-negative.
- In case of `Cons(l,r)`, we can apply induction. We know that `l.evalExp()` ≤ 0 and `r.evalExp()` ≤ 0. The sum of two non-negative numbers will be non-negative.

$\square$

Last missing piece

Lemma

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

Proof.

- In case of the constants this is trivial as $1 > 0$
- In case of `Union(l,r)`, we can apply induction. We know that `l.evalExp()` ≤ 0 and `r.evalExp()` ≤ 0. The maximum of two non-negative numbers will be non-negative.
- In case of `Cons(l,r)`, we can apply induction. We know that `l.evalExp()` ≤ 0 and `r.evalExp()` ≤ 0. The sum of two non-negative numbers will be non-negative.
- In case of `Pow(r,n)`, we can apply induction. We know that $n \leq 0$ and `r.evalExp()` ≤ 0. The product of two non-negative numbers will be also non-negative.

$\square$

Sounds good, doesn't work?

Every theorem about words and languages has been verified by Stainless.

Sounds good, doesn't work?

Every theorem about words and languages has been verified by Stainless.

About regular expressions, only the exponent positiveness lemma has been verified, the other has been proven for the constant and union cases.

Sounds good, doesn't work?

Every theorem about words and languages has been verified by Stainless.

About regular expressions, only the exponent positiveness lemma has been verified, the other has been proven for the constant and union cases.

Every verifying theorem and lemma was cleared to be terminating.

Sounds good, doesn't work?

Every theorem about words and languages has been verified by Stainless.

About regular expressions, only the exponent positiveness lemma has been verified, the other has been proven for the constant and union cases.

Every verifying theorem and lemma was cleared to be terminating.

Metrics (excluding lemmas about regular expressions):

- 95 functions (methods and lemmas)
- 651 verified conditions
- 5127.148s of total time for verification (longest: 318.947s)
- 1936.337s of total time for checking termination

CPU: Dual-Core Intel® Core$^{\text{TM}}$ i5-4210U CPU @ 1.70GHz processor
RAM: 8GB

Conclusion and Future Work

Summary:

- analyzed different language representations
- presented a formal language implementation using lists
- based on that implementations proved properties of languages
- applied implemented abstractions to represent regular expressions and prove theorems about them

Conclusion and Future Work

Summary:

- analyzed different language representations
- presented a formal language implementation using lists
- based on that implementations proved properties of languages
- applied implemented abstractions to represent regular expressions and prove theorems about them

Possible future work:

- Experiment with other language representation
- Finish incomplete lemmas about regular expressions
- Try to prove further statements about regular expressions
- Examine other abstractions using formal languages

Questions?