Bachelor thesis
Computer Science

Radboud University

From μ-regular expressions to context-free grammars and back

Author:
Bart Gruppen
s4465784

First supervisor/assessor:
dr. J.C. (Jurriaan) Rot
`jrot@cs.ru.nl`

Second assessor:
prof. dr. J.H. Geuvers
(Herman)
`herman@cs.ru.nl`

July 2, 2018

Abstract

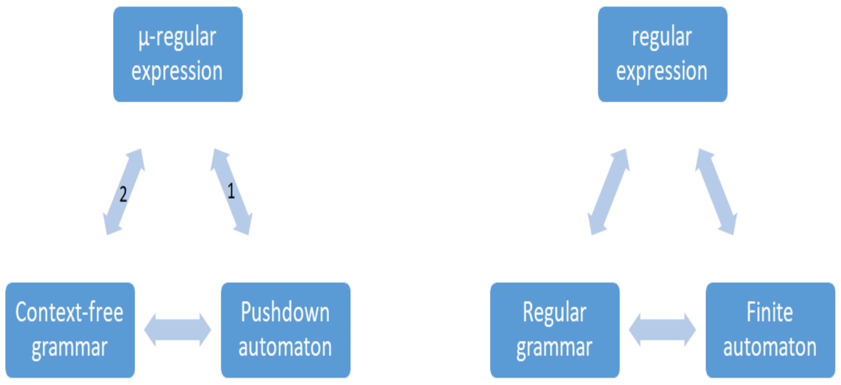
Peter Thiemann showed how to go from μ-regular expressions to a push-down automata [7]. In that context, it is also possible to go from μ-regular expressions to context-free grammars and to go directly from context-free grammars to μ-regular expressions. In this thesis we will show how this works and additionally we will show how the Kleene star is redundant when having a μ-regular expression.

Contents

Chapter 1

Introduction

Regular expressions and regular grammars are used to describe and produce regular languages. This is widely used in the theory of formal languages [6]. However, regular languages turn out to be not expressive enough in some cases. Context-free grammars have more expressive power and are therefore capable to express different languages. Languages that are produced by a context-free grammar are called context-free languages. We know that context-free languages are accepted by pushdown automata, but are there also expressions that describe context-free languages? The answer is positive and the expressions are called μ-regular expressions. The phenomen of μ-regular expressions are recently being studied by various people, like P. Thiemann [7], N. Krishnaswami [4] and H. Leiss [5]. Thiemann for example showed that the languages described by a μ-regular expression are exactly the same as the languages accepted by pushdown automata. This is the relation in Figure 1.1a that contains a "1" in the double arrow.



(a) Context-free languages (b) Regular languages

Figure 1.1

We also see that Figure 1.1b displays three blocks that correspond with regular languages. However, in this thesis we will mainly focus on the relations that are in 1.1a, So the blocks that correspond with context-free languages. We see three relations in there.

- As said before, the relation between μ-regular expressions and pushdown automata is proved by Thiemann [7].
- The relation between context-free grammars and pushown automata is very well known [3].
- The relation between μ-regular expressions and context-free grammars is the one that we mainly focus on in this thesis. We want to check whether the language of a μ-regular expression is the same as a language of a context-free grammar and the other way around as well.

In this thesis we will show how to make a μ-regular expression from a context-free grammar and vice versa, so the relation in Figure 1.1a that contains a "2" in the double arrow. This includes algorithms and correctness proofs. This relation is considered a folklore theorem, but has never actually been proved. The actual proof of it is therefore needed. Going from a μ-regular expression to a context-free grammar basically comes down to mapping from a starting symbol to the elements of the expression. The provided algorithm is made for theoretical purposes and is not optimized for implementation. To go from a context-free grammar to a μ-regular expression means that one has to substitute all free variables with their corresponding μ-regular expressions.

Furthermore, we will show that the Kleene star from the regular expressions can be replaced by the least fixed point operator μ. This means that the syntax of the grammar can be smaller, but the expressive power remains the same. The intuition here is that the least fixed point operator μ can bind a variable and can therefore produce in an repetitive way. For all three cases the provided solutions (algorithms and correctness proofs) achieve their goals, because the solutions are proved to be correct.

Chapter 2

Preliminaries

Definition 1 (Language)**.** *Let Σ be an alphabet and Σ^* all words over this alphabet. Then a language L is a subset of Σ^*.*

Example 2.0.1 (Languages)**.**

- $\emptyset$ is the language with no words.
- $\{\epsilon\}$ is the language which just contains the empty word.
- $\{\epsilon, a, aa\}$ is the language containing the three words ϵ, a and aa.
- $\{w \in \{a,b\}^* \mid w = w^R\}$ is the language containing only palindromes.

We consider two different type of languages: Regular languages and context-free languages. Every regular language is context-free, but the other way around is not true.

Given languages L, K we write concatenation as usual by $LK := \{wv \mid w \in L \text{ and } v \in K\}$ and the Kleene star as $L^* := \bigcup_{i\in\mathbb{N}} L^i = \{\epsilon\} \cup L \cup LL \cup LLL...$

Example 2.0.2 (Kleene star)**.** If $L = \{a\}$ then L^* contains λ, a, aa, aaa, ...

Definition 2 (Regular expression)**.** *The set $Reg(\Sigma)$ is defined recursively with the syntax:*

$$r, s ::= 0 \mid 1 \mid a \mid rs \mid r+s \mid r^*$$

where a ranges over Σ.

The language of a regular expression is given by:

$$\begin{aligned}
\mathcal{L}(0) &= \emptyset \\
\mathcal{L}(1) &= \{\epsilon\} \\
\mathcal{L}(a) &= \{a\} \\
\mathcal{L}(rs) &= \mathcal{L}(r)\mathcal{L}(s) \\
\mathcal{L}(r+s) &= \mathcal{L}(r) \cup \mathcal{L}(s) \\
\mathcal{L}(r^*) &= (\mathcal{L}(r))^*
\end{aligned}$$

where $\mathcal{L} : Reg(\Sigma) \to \mathcal{P}(\Sigma^*)$ and $\mathcal{P}(\Sigma^*)$ denotes the powerset of Σ^*.

Definition 3 (Regular language)**.** *A language L is called regular if $L = \mathcal{L}(r)$ for some r, where r is a regular expression.*

Example 2.0.3 (Regular expressions and regular languages)**.** The regular expression $a(a+b)^*b$ represents the set of strings starting with an a, followed by strings of a's and b's of any length and in any sequence, ending with a b. $\mathcal{L}(a(a+b)^*b) = \{w \mid w \text{ begins with an } a \text{ and ends with } b\}$ is the corresponding language. The language $L = \mathcal{L}(a(a+b)^*a)$ is regular because it can be made of the expression $(a(a+b)^*a)$.

Definition 4 (Context-free grammar)**.** *A context-free grammar is a four-tuple $G = (V, P, \Sigma, S)$ where*

- *V is a finite set. Each element $v \in V$ is a variable (non-terminal). Non-terminals (or variables) are the capital letters in the examples. So in $A \to a$, A is a non-terminal and a is a terminal. In this thesis, we will mix the use of the terms variable and non-terminal.*
- *$P \subseteq V \times (\Sigma \cup V)^*$ are the actual (rewrite) rules from the grammar. This is also called the production. It is a finite relation between the non-terminals and the union of the terminals and the non terminals. There are no terminals on the left-hand side of the relation.*
- *Σ is a finite set of terminals. This set is also called the alphabet of G.*
- *S is the starting symbol of the grammar G. S is a non-terminal and has to occur in the set V.*

Definition 5 (Language of a context-free grammar)**.** *We denote derivation by $\Rightarrow$, and $\overset{+}{\Rightarrow}$ denotes one or more derivations. This is the expansion of productions, described in Definition 4, to words. We say that*

$$w \overset{+}{\Rightarrow} v \iff \exists w_1, w_2, T, R : w = w_1 T w_2 \,\&\, v = w_1 R w_2 \,\&\, (T, R) \in P$$

Given a context-free grammar G, the language generated by G is:

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\Rightarrow} w\}.$$

A language $K \subseteq \Sigma^*$ is a context-free language iff $K = L(G)$ for some context-free grammar G.

Example 2.0.4 (Context-free grammar)**.**
Consider the following grammar $G : (\{S\}, \{(S, aSb), (S, \epsilon)\}, \{a, b\}, S)$, or, written differently:

$$S \to aSb \mid \epsilon$$

It is customary to not write this as a four-tuple but as productions. Both notations will be used in this thesis. This CFG corresponds to the language $L(G) = \{a^n b^n \mid n \in \mathbb{N}\}$.

Chapter 3

Context-free languages by equations

In Definition 5 we have seen a way to describe the semantics of a context-free grammar. However, this turns out to not be the smartest way for us to do so. In this section we will discuss an alternative way. It turns out that languages of context-free grammars can be seen as a system of equations. If one takes the smallest set of this system of equations, one will get the corresponding language. To formalize this, we introduce the least fixed point (lfp). A good thing about describing a language in this way is that it already looks a bit like the language of a μ-regular expression, which will be introduced in Chapter 4.

So what is a least fixed point? To understand this, we first need to have a look at "normal" fixed points. A function is said to have a fixed point if $f(x) = x$. An example with natural numbers is: $f(x) = x^3 + 2x - 10$. When one takes $x = 2$, the result will be 2 as well. So $f(2) = 2$. This means 2 is a fixed point of $f(x)$. In this thesis we do not focus on numbers, but rather on languages. Let us go back to the languages of context-free grammars, written as a system of equations. Here is an example: Consider the context-free grammar H:

$$S \to aSb \mid \epsilon$$

The corresponding equation looks like this:

$$L = \{a\}L\{b\} \cup \{\epsilon\} \tag{3.1}$$

The unique solution of this equation is the language that is produced by grammar H. We formalize this as a least fixed point, which happens to be unique. Because we are talking about languages now, we also have inclusion. So we do not just look at fixed points, but at least fixed points. A function does not always have a least fixed point, but there is a condition under which there actually always is one. That is, if the function is monotone. In natural language, a function is called monotone if the following is true: if the input of a function increases, the output of the function does not decrease. So either it stays the same, or it increases. For the next definition we assume that the arity of the function is one. This means that the function has one argument as input.

Definition 6 (Monotone function with one argument)**.**
A function $f : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ on languages is monotone if

$$\forall L, K \in \mathcal{P}(\Sigma^*) : L \subseteq K \to f(L) \subseteq f(K)$$

Theorem 1. *If a function f is monotone, f has a least fixed point* $\mathrm{lfp}L.f(L)$ *(This is the smallest L such that $f(L) = L$).*

Example 3.0.1 (Context-free grammar as an equation)**.** Consider the grammar G given by: $S \to abS \mid \epsilon$. This can be seen as the equation: $L = \{a\}\{b\}L \cup \{\epsilon\}$. The least solution can be approximated as follows:

$$\emptyset \subseteq \{a\}\{b\}\emptyset \cup \{\epsilon\} \subseteq \{a\}\{b\}(\{a\}\{b\}\emptyset \cup \{\epsilon\}) \cup \{\epsilon\} \subseteq ...$$

But we know that $L\emptyset = \emptyset$ and $L\{\epsilon\} = L$. So we can write this as:

$$\emptyset \subseteq \{\epsilon\} \subseteq \{a\}\{b\} \cup \{\epsilon\} \subseteq ...$$

Because our function f, mentioned in Definition 6, is monotone we can state that:

$$\emptyset \subseteq f(\emptyset) \subseteq f(f(\emptyset)) \subseteq ...$$

And then the least fixed point would be:

$$\mathrm{lfp}(f) = \bigcup_{i \geq 0} f^i(\emptyset)$$

This raises the question: Why do we need the *least* fixed point of an equation to get its language? Consider the grammar with the following production: $S \to S \mid \epsilon$. This corresponds to all the languages containing the empty word. We can even remove the ϵ, so the production would look like this: $S \to S$. This might seem a little bit strange. But this produces all the languages. Since we only need one, we take the smallest solution.

Now let us see what least fixed point of grammar H from (3.1) is. We already set up the equation $S = \{a\}S\{b\} \cup \{\epsilon\}$. The corresponding least fixed point equals:

$$\mathrm{lfp}L.\{a\}L\{b\} \cup \{\epsilon\}$$

which means the function is

$$f(L) = \{a\}L\{b\} \cup \{\epsilon\}$$

This gives the solution of this fixed point as:

$$\{a^n b^n \mid n \geq 0\}$$

We have seen monotone functions with arity one in Definition 6. This was because we only had one non-terminal in our grammar, namely the starting symbol. But there can be more non-terminals in a grammar. This means that we need to generalize to allow this. Now that we have more variables to deal with, we will make use of assignment of languages which we call η. The type of η is:

$$\eta : V \to \mathcal{P}(\Sigma^*)$$

But we are not just dealing with more variables. We are dealing with words (so non-terminals and terminals). Therefore we extend η by $\bar{\eta}$.

Given η, the map $\bar{\eta}$ is defined by: $\forall X \in V$, $w \in (V \cup \Sigma)$:

$$\bar{\eta} : (V \cup \Sigma)^* \to \mathcal{P}(\Sigma^*)$$

$$\bar{\eta}(\epsilon) = \{\epsilon\}$$
$$\bar{\eta}(Xw) = \eta(X)\bar{\eta}(w)$$
$$\bar{\eta}(aw) = \{a\}\bar{\eta}(w)$$

The map $\bar{\eta}$ makes a singleton letter word for each terminal and passes the non-terminals on to the original function: η. Now that we can handle multiple variables and words, we define the following function, where $X^Y = \{f \mid f : Y \to X\}$.

Definition 7 (Monotone function with multiple arguments)**.** *If V is the set of non-terminals (variables), then:*

$$f : \mathcal{P}(\Sigma^*)^V \to \mathcal{P}(\Sigma^*)^V$$

is monotone if for all $\eta_1, \eta_2 \in \mathcal{P}(\Sigma^*)^V$:

$$(\forall T \in V : \eta_1(T) \subseteq \eta_2(T)) \to (\forall T \in V : f(\eta_1)(T) \subseteq f(\eta_2)(T))$$

We here explain that we can get the language of a grammar by considering it as a system of equations and taking the least fixed point of our carefully chosen function f^G and take the starting symbol of that result. Our aim is to prove that $L(G) = \text{lfp}(f^G)(S)$.

Our carefully chosen f^G will calculate the language for all the non-terminals in V. We will work with one variable at the time. So for the definition of f^G we will have to break the function in smaller parts which is done down here:

$$f^G(\eta) : V \to \mathcal{P}(\Sigma^*)$$

$$f^G(\eta)(X) = f_X(\eta)$$

Where f_X will be defined later on. After this the heart of the method comes. The next function will take one variable (which is possible because of the previous function) and gives the language of the terminals and non-terminals. We will explain this method via an example grammar. We start with the first two steps:

Example 3.0.2 (From CFG to a language)**.** Consider the CFG with the following productions:

$$S \to abT \mid \epsilon$$

$$T \to aTb \mid a \mid S$$

The first step is to calculate the language for both non-terminals. We break up the calculation and we get the two functions $f(\eta)(S)$ and $f(\eta)(T)$. These are respectively equal to $f_S(\eta)$ and $f_T(\eta)$. This is the moment where the new method comes in. We define the following function:

$$f_X : \mathcal{P}(\Sigma^*)^V \to \mathcal{P}(\Sigma^*)$$

$$f_X(\eta : V \to \mathcal{P}(\Sigma^*)) = \bigcup_{(X,w)\in P} \bar{\eta}(w)$$

Now we can apply this on our two functions $f_S(\eta)$ and $f_T(\eta)$. The answer is worked out below:

$$f_S(\eta) = \bigcup_{(S,w)\in P} \bar{\eta}(w) = \{a\}\{b\}\eta(T) \cup \{\epsilon\}$$

$$f_T(\eta) = \bigcup_{(T,w)\in P} \bar{\eta}(w) = \{a\}\eta(T)\{b\} \cup \{a\} \cup \eta(S)$$

If one would take the grammar from Example 3.0.2 to a system of equations, it would look like this:

$$L = \{a\}\{b\}K \cup \{\epsilon\}$$

$$K = \{a\}K\{b\} \cup \{a\} \cup L$$

It already looks very similar to what is calculated above. Because we are dealing with multiple variables, it is convenient to present the result as a tuple. This would make $(L, K) = (\{a\}\{b\}K \cup \{\epsilon\}, \{a\}K\{b\} \cup \{a\} \cup L)$. But we want to keep track of the non-terminals from the context-free grammar, so we map the variables: $(S \mapsto L, T \mapsto K) = (S \mapsto \{a\}\{b\}K \cup \{\epsilon\}, T \mapsto \{a\}K\{b\} \cup \{a\} \cup L)$.

We have almost got a result now. But there is still the function η in the result. As said, this function maps a variable to a language. The only thing we need to do now is taking the least fixed point of the outcomes:

$$\mathrm{lfp} L.\{a\}\{b\}K \cup \{\epsilon\}$$

$$\text{lfp}K.\{a\}K\{b\} \cup \{a\} \cup L$$

Combining them to one least fixed point would give:

$$\text{lfp}(L, K).(\{a\}\{b\}K \cup \{\epsilon\}, \{a\}K\{b\} \cup \{a\} \cup L)$$

This is the outcome we want, since the tuple with application above maps to the exact same outcome and we also kept track of the non-terminals there, which we intended to capture by $\text{lfp}\, \eta.f^G(\eta)$

Now that we have Definition 7, with η-functions, we can define the language of a context-free grammar with non-terminals. This brings us closer to comparing the language of a context-free grammar with the language of a μ-regular expression.

Theorem 2 ([2])**.** *For a monotone function f and a context-free grammar G:* $\text{lfp}(f^G)(S) = L(G)$

Chapter 4

μ-regular expressions

In this chapter we will dive into μ-regular expressions and give an idea of why this is helpful for this thesis. We will also give the syntax and the semantics of it. We have already seen regular expressions in Definition 2. A μ-regular expression is actually nothing more than a regular expression extended with a least fixed point operator μ. We did not need this operator to compare the languages of regular expressions and the language of regular grammars, but now that we have context-free grammars we need to extend the scope. It will soon be clear how this fixed point operator will extend the scope.

Let us see an example of a μ-regular expression and the corresponding language. Say we have the μ-regular expression $e = \mu x.axb + 1$. The μ is, as we said, the fixed point operator. Roughly said this means that it replaces the starting symbol by the variable it binds. This is in our case the x.

Definition 8 (μ-regular pre-expressions)**.** *The set $R(\Sigma, X)$ of μ-regular expressions over the alphabet Σ and the unbound variables X is defined inductively by*

$$\begin{aligned}
&0 \in R(\Sigma, X)\\
&1 \in R(\Sigma, X)\\
&a \in \Sigma \to a \in R(\Sigma, X)\\
&r, s \in R(\Sigma, X) \to rs \in R(\Sigma, X)\\
&r, s \in R(\Sigma, X) \to r + s \in R(\Sigma, X)\\
&r \in R(\Sigma, X) \to r^* \in R(\Sigma, X)\\
&x \in X \to x \in R(\Sigma, X)\\
&r \in R(\Sigma, X \cup \{x\}) \to \mu x.r \in R(\Sigma, X)
\end{aligned}$$

Definition 9. *If there occurs an expression which contains a variable that is not bound (yet), we call the expression a μ-regular pre-expression.*

If all variables are bound, so $X = \emptyset$, we do not speak about a μ-regular pre-expression anymore, but we say that the expression is a μ-regular expression. From our example expression e we extract $axb+1$. The expression 1 is an inductive base case for μ-regular expressions. The same holds for a and b. But the x is not bound yet. So we cannot speak about a μ-regular expression yet. It still is a μ-regular pre-expression. Now that we know the syntax of a μ-regular pre-expression, it is time to go on to the semantics. Therefore we need to give meaning to the unbound variables in the μ-regular pre-expression. This is done by a variable environment, which we will call η. Let $\eta : X \to \mathcal{P}(\Sigma^*)$ be a function that maps these variables to languages. η can be extended by other mappings as concatenation. The language that is described by a μ-regular pre-expression will then be defined inductively as: $\mathcal{L} : R(\Sigma, X) \times (X \to \mathcal{P}(\Sigma^*)) \to \mathcal{P}(\Sigma^*)$.

$$\begin{array}{rcl}
\mathcal{L}(0,\eta) & = & \emptyset \\
\mathcal{L}(1,\eta) & = & \{\epsilon\} \\
\mathcal{L}(a,\eta) & = & \{a\} \\
\mathcal{L}(rs,\eta) & = & \mathcal{L}(r,\eta)\mathcal{L}(s,\eta) \\
\mathcal{L}(r+s,\eta) & = & \mathcal{L}(r,\eta) \cup \mathcal{L}(s,\eta) \\
\mathcal{L}(x,\eta) & = & \eta(x) \\
\mathcal{L}(\mu x.r,\eta) & = & \mathrm{lfp} L.\mathcal{L}(r,\eta[x \mapsto L])
\end{array}$$

If there are no unbound variables in an expression, so $r \in R(\Sigma)$, then

$$R(\Sigma) := R(\Sigma, \emptyset)$$

and we write

$$\mathcal{L}(r) := \mathcal{L}(r, \emptyset)$$

We will work out our example expression $e = \mu x.axb + 1$ down here and show the corresponding language:

Example 4.0.1 (Language of a μ-regular expression)**.**

$$\begin{array}{rl}
& \mathcal{L}(\mu x.axb+1,\eta) \\
= & \mathrm{lfp} L.\mathcal{L}(axb+1,\eta[x \mapsto L]) \\
= & \mathrm{lfp} L.\mathcal{L}(axb,\eta[x \mapsto L]) \cup \mathcal{L}(1,\eta[x \mapsto L]) \\
= & \mathrm{lfp} L.\mathcal{L}(a,\eta[x \mapsto L]) \cdot (x,\eta[x \mapsto L]) \cdot (b,\eta[x \mapsto L]) \cup \mathcal{L}(1,\eta[x \mapsto L]) \\
= & \mathrm{lfp} L.\{a\} \cdot \eta[x \mapsto L](x) \cdot \{b\} \cup \{\epsilon\} \\
= & \mathrm{lfp} L.\{a\}L\{b\} \cup \{\epsilon\}
\end{array}$$

This is the exact same answer as we found in Definition 6. So the language produced the context-free grammar $S \to aSb \mid \epsilon$ is the language as described by the μ-regular expression $\mu x.axb + 1$.

Chapter 5

Redundancy of the Kleene star

In this chapter we will discuss why we do not need the Kleene star in a μ-regular expression. The Kleene star can be characterized as follows: $L^* = \text{lfp}K.LK \cup 1$. We will now give an example to get the right intuition why the Kleene star can be abandoned and later we will show this is general. We start by giving an example grammar and its corresponding μ-regular expression. Consider the following context-free grammar:

$$S \to a^*$$

One could alternatively write this as follows:

$$S \to aS \mid \epsilon$$

Not only does this grammar not include the Kleene star, it is also easy to see that the language produced by this grammar is equivalent to the following μ-regular expression:

$$\mu x.ax + 1$$

This μ-regular expression does not contain the Kleene star anymore but still describes the same language as the grammar produces. We will prove this in a theorem later on. But before we go there, we need to prove a lemma. In the following proof of this lemma we will use regular pre-expressions, because we prove a more general statement, since this turns out to be easier. So we will have $R(\Sigma, X)$ instead of $(\Sigma, \emptyset)$. The lemma will say that if an application does not apply to the current variables, one can ignore it and leave it out of the equation.

Lemma 1. *For every language L, every $r \in R(\Sigma, X)$, every $\eta : X \to \mathcal{P}(\Sigma^*)$ and every x such that $x \notin X$:*

$$\mathcal{L}(r, \eta[x \mapsto L]) = \mathcal{L}(r, \eta)\,.$$

So x maps to L and all the other variables go to η of an element.The type of the operation is $\eta[x \mapsto L] : X \cup \{x\} \to \mathcal{P}(\Sigma^*)$.

Proof. We will prove this by induction on r by treating each of the cases starting with the base cases where $r = 0$, $r = 1$ and $r = a$.

$$\begin{aligned}\mathcal{L}(0, \eta[x \mapsto L]) &= \emptyset \\ &= \mathcal{L}(0, \eta)\end{aligned}$$

$$\begin{aligned}\mathcal{L}(1, \eta[x \mapsto L]) &= \{\epsilon\} \\ &= \mathcal{L}(1, \eta)\end{aligned}$$

$$\begin{aligned}\mathcal{L}(a, \eta[x \mapsto L]) &= \{a\} \\ &= \mathcal{L}(a, \eta)\end{aligned}$$

Now suppose $r = y$ such that $y \in X$ and so $y \neq x$.

$$\begin{aligned}\mathcal{L}(y, \eta[x \mapsto L]) &= \eta[x \mapsto L](y) \\ &= \eta(y) \\ &= \mathcal{L}(y, \eta)\end{aligned}$$

Now assume it holds for r_1 and r_2. We will now show that the union (+) of r_1 and r_2 is correct. The proofs of concatenation ($\cdot$) and the Kleene star (*) looks the same as the proof of the union, so we will skip these two in the proof.

$$\begin{aligned}\mathcal{L}(r_1 + r_2, \eta[x \mapsto L]) &= \mathcal{L}(r_1, \eta[x \mapsto L]) \cup \mathcal{L}(r_2, \eta[x \mapsto L]) \\ &\overset{IH}{=} \mathcal{L}(r_1, \eta) \cup \mathcal{L}(r_2, \eta) \\ &= \mathcal{L}(r_1 + r_2, \eta)\end{aligned}$$

The last induction step is on an expression which starts with a μ. We assume that this lemma holds for r. We can make the second step, so swapping $[x \mapsto L]$ and $[y \mapsto K]$, because we know that $x \neq y$, so the order does not matter.

$$\begin{aligned}\mathcal{L}(\mu y.r, \eta[x \mapsto L]) &= \text{lfp}K.\mathcal{L}(r, \eta[x \mapsto L][y \mapsto K]) \\ &= \text{lfp}K.\mathcal{L}(r, \eta[y \mapsto K][x \mapsto L]) \\ &\overset{IH}{=} \text{lfp}K.\mathcal{L}(r, \eta[y \mapsto K]) \\ &= \mathcal{L}(\mu y.r, \eta)\end{aligned}$$

□

We can now prove that we do not need the Kleene star in a μ-regular expression. This means that there are no free variables in the expression.

Theorem 3. $\forall r \in R(\Sigma, X) : \mathcal{L}(r^*) = \mathcal{L}(\mu x.rx + 1)$

Proof. The proof of this theorem by substitution will be given below:

$$\begin{aligned}
\mathcal{L}(\mu x.rx+1) &= \mathcal{L}(\mu x.rx+1, \emptyset) \\
&= \mathrm{lfp} L.\mathcal{L}(rx+1, [x \mapsto L]) \\
&= \mathrm{lfp} L.\mathcal{L}(r, [x \mapsto L]) \cdot (x, [x \to L]) \cup \{\epsilon\} \\
&= \mathrm{lfp} L.\mathcal{L}(r, \emptyset) \cdot L \cup \{\epsilon\} && \text{(Lemma 1)} \\
&= (\mathcal{L}(r, \emptyset))^* \\
&= \mathcal{L}(r^*, \emptyset) \\
&= \mathcal{L}(r^*)
\end{aligned}$$

□

Note that the first and the last step can be made because there are no unbound variables and in that case $R(\Sigma) := R(\Sigma, \emptyset)$ holds.

Chapter 6

From context-free grammars to μ-regular expressions

In this section we will present another main contribution. We will now show how to go from a context-free grammar to a μ-regular expression.

Example 6.0.1. We have already seen that that the grammar

$$S \to aSb \mid \epsilon$$

produces the same language as the μ-regular expression

$$\mu S.aSb + 1$$

Looking closely at the grammar and the μ-regular expression we can see a pattern. The disjunction bar of the grammar is replaced by a plus in the μ-regular expression and the starting symbol S is replaced by μS. This grammar actually has two productions: aSb and ϵ. We denote the μ-regular expression by a summation as follows: $\mu S. \sum\limits_{(S,w)\in P} w$.

In the example above, it was clear that we were dealing with the same language. But for more complex cases, we will now provide an algorithm to go from a CFG to a μ-regular expression.

1: Start with the expression as the starting symbol S.

2: Substitute all free variables in the expression with their corresponding μ-regular expressions. The sequence does not matter. All the appearances of the variable have to be replaced. Continue until there are no free variables left.

In (pseudo)code, this would be:

```
$e := S$
while $e$ contains free variables do
   let $x$ be a free variable in $e$
   $e := e[x := \mu S. \sum_{(S,w)\in P} w]$
end while
return $e$
```

We will now demonstrate our algorithm on the following example.

Example 6.0.2. Consider the context-free grammar G which has the following productions:

$$\begin{array}{rcl} S & \rightarrow & aST \mid U \\ T & \rightarrow & TU \mid S \\ U & \rightarrow & \epsilon \end{array}$$

Following the algorithm step by step we could get a table like this :

e	Free variables	Next assignment
S	S	$S[S := \mu S.aST + U]$
$\mu S.aST + U$	T, U	$T[T := \mu T.(TU + S)]$
$\mu S.aS(\mu T.TU + S) + U$	U	$U[U := \mu U.\epsilon]$
$\mu S.aS(\mu T.T(\mu U.\epsilon) + S) + \mu U.\epsilon$		

The algorithm seems to work for any example. Nevertheless, it turned out to be very difficult to prove that it works on a general basis. This is why we decided to turn the algorithm upside down. In the previous algorithm, we started with something small (the starting symbol) and we keep expending it. But we can also do it the other way around. So we start with all the productions and we combine them into one expression. This is exactly what the algorithm will do. The new algorithm looks in (pseudo)code like this:

```
$F := V$
$E(F) := \{ \sum_{(X,w)\in P} w \mid X \in V\}$
while $F \neq \{S\}$ do
   let $Y \in F$, $Y \neq S$
   $e_Y := E(Y)$
   $F := F \setminus \{Y\}$
   $\forall X \in F : E(X) := E(X)[Y \mapsto \mu Y.e_Y]$
end while
return $\mu S.E(S)$
```

We will demonstrate this algorithm on the same example as above to show that the outcome is the same. But first we will demonstrate it on another example below.

Example 6.0.3. Consider the grammar H, which productions are in the column "Grammar rules".

Grammar rules	First iteration	Second iteration
$X \to aYX$	$E(X) = a(\mu Y.XYbZ)X$	$E(X) = a(\mu Y.XYb(\mu Z.X(\mu Y.XYbZ)Z))X$
$Y \to XYbZ$	$E(Z) = X(\mu Y.XYbZ)Z$	
$X \to XYZ$		

One can see that the algorithm runs two iterations. From three rules to one expression. The final step is to add the starting symbol bound by a mu: $\mu X.a(\mu Y.XYb(\mu Z.X(\mu Y.XYbZ)Z))X$.

Example 6.0.4. And now the application of the algorithm to our grammar G from Example 6.0.2.

Grammar rules	First iteration	Second iteration
$S \to aST \mid U$	$E(S) = aS(\mu T.TU + S) + U$	$E(S) = aS(\mu T.T(\mu U.\epsilon) + S) + \mu U.\epsilon$
$T \to TU \mid S$	$E(U) = \epsilon$	
$U \to \epsilon$		

And again the final step is binding the the starting symbol to a mu: $\mu S.aS(\mu T.T(\mu U.\epsilon) + S) + \mu U.\epsilon$. This is exactly the same outcome we got with our first algorithm. So the algorithms seem to give the same result. The big difference here is that it is a lot easier to prove the correctness of the algorithm which we will do below.

6.1 Correctness

We will now prove that the language of a μ-regular expression is the same language as the language of a CFG after it has been transformed to a μ-regular expression. We will prove that our (last) algorithm from above is correct. Since we use a while loop, it is convenient that we will prove correctness with an invariant. The invariant is:

$$\forall U \in F \setminus \{Y\} : (\mathrm{lfp} f^G)(U) = (\mathrm{lfp}\, \eta.(\lambda x.\mathcal{L}(E(X), \eta)))(U)$$

We have to prove the correctness on three places in the algorithm:

(1) Before the loop.

(2) In the loop, after one iteration.

(3) After the loop.

The hardest one is the one inside the while loop, so number (2). This will be the first of the three to prove. We have to prove that the invariant is preserved after the body. So that the language before one iteration has to be the same as after one iteration, because if it is preserved after one iteration, it will be preserved after multiple iterations as well. The language will be determined via the variables. Before we give the actual proof, we first state two lemmas which we are using.

Lemma 2. $\mathcal{L}(E(X), \eta[Y \mapsto \text{lfp}L.\mathcal{L}(e_Y, \eta[Y \mapsto L])]) = \mathcal{L}(E(X)[Y \mapsto \mu Y.e_Y], \eta)$

Proof. This proof will be done by induction on $E(X)$. The only interesting case is when $E(X) = x$ and $x = y$. So that case will be handled below. The other cases are rather uninteresting, because all the variables will just drop away, since they do not occur in the formula.

$$\begin{aligned}
\mathcal{L}(x, \eta[x \mapsto \text{lfp}L.\mathcal{L}(e_x, \eta[x \mapsto L])]) &= \eta[x \mapsto \text{lfp}L.\mathcal{L}(e_x, \eta[x \mapsto L])])(X) \\
&= \text{lfp}L.\mathcal{L}(e_x, \eta[x \mapsto L]) \\
&= \mathcal{L}(\mu x.e_x, \eta) \\
&= \mathcal{L}(x[x \mapsto \mu x.e_x], \eta)
\end{aligned}$$

□

We will use Bekić's bisection lemma [1] to prove the correctness. We will not fully take his lemma, but an slightly adapted version so that it fits in our expressions.

Lemma 3 (Bekić's bisection lemma). *For all* $U \in V \setminus \{Y\}$*:*

$$(\text{lfp}\eta.\lambda X.f_X(\eta))(U) = (\text{lfp}\eta.\lambda X.f_X(\eta[Y \mapsto \text{lfp}L.f_Y(\eta[Y \mapsto L])]))(U)$$

The proof that the language remains the same after one iteration can now be made.

Theorem 4. *For all* $U \in F \setminus \{Y\}$*:*

$$(\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X), \eta)))(U) = (\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X)[Y \mapsto \mu Y.e_Y], \eta)))(U)$$

Proof.

$$\begin{aligned}
&\forall U \in F \setminus \{Y\} : (\text{lfp} f^G)(U) \\
= \; &(\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X), \eta)))(U) && \text{(Invariant)} \\
= \; &(\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X), \eta[Y \mapsto \text{lfp}L.f_y(\eta[Y \mapsto L])])))(U) && \text{(Lemma 3)} \\
= \; &(\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X), \eta[Y \mapsto \text{lfp}L.\mathcal{L}(E(Y), \eta[Y \mapsto L])])))(U) \\
= \; &(\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X), \eta[Y \mapsto \text{lfp}L.\mathcal{L}(e_Y, \eta[Y \mapsto L])])))(U) \\
= \; &(\text{lfp}\,\eta.(\lambda x.\mathcal{L}(E(X)[Y \mapsto \mu Y.e_Y], \eta)))(U) && \text{(Lemma 2)}
\end{aligned}$$

□

We now have proved that the invariant stays intact after an iteration of the while loop, so number (2). We still need to prove that our invariant is correct before the while loop and after it. Before the while loop is straightforward, because the invariant is about F and E and those are the variables that are being assigned. This means they are set perfectly for the invariant and so the invariant is correct there. So that is number (1).

After the while loop is even more straightforward, because there are no further assignments there. The only thing that happens is that the starting symbol is bound to a mu. This means number (3) is correct too. So all on all three places the invariant is proved to be correct so the invariant is correct and with that the algorithm is correct.

Chapter 7

From μ-regular expressions to context-free grammars

In this section we present one of our main contributions. We will show how to go from a μ-regular expression to a context-free grammar. We will do this by defining a function by induction, but first let us build up some intuition. We know from Definition 4 and from Example 2.0.4 what a context-free grammar looks like. We have also seen some μ-regular expressions already. We will now show two more μ-regular expressions and their corresponding context-free grammars to get some more intuition.

$$\mu x.abx + c$$

corresponds to

$$X \to abX \mid c$$

One can see that the transformation is quite easy to compute. The x is bound by the μ, so this roughly means that the x will be a non-terminal in the produced grammar. The a, b and c are just letters from the alphabet. It gets harder when the μ-regular expression is nested, as in the next example.

$$\mu x.ab(\mu y.cxa) + 1$$

corresponds to

$$X \to abY \mid \epsilon$$

$$Y \to cXa$$

Again, one can see the substitution of variables into the μ-regular expressions. Now that we have seen some examples, we will continue to the formal definition of our transformation. We will define this by induction on μ-regular pre-expressions. The function will have a μ-regular expression as input and a context-free-grammar as output. The type of a context-free grammar is given in Definition 4 in the Preliminaries. However, we will not entirely follow this type in our definition. Because we are dealing with μ-regular pre-expressions, we need to add the set of unbound variables X to the grammars, where $V \cap X = \emptyset$. We define a CFG with variables as a tuple (V, P, Σ, S, X). The production P now is the relation: $P \subseteq V \times (\Sigma \cup V \cup X)^*$. Furthermore note that the starting symbol $S \in V$ and Σ is still the alphabet. We need the CFG with variables, because the proof requires intermediate steps. In this intermediate steps we are talking about μ-regular pre-expressions, which might have free variables.

This extension is needed, because if a free variable would be the argument of the η function of Chapter 3, the function would not be able to work properly. So this extra variable to the grammar turns out to be necessary in the proof we will show after this section, because intermediate results might have unbound variables. The function that converts μ-regular expressions into context-free grammars will be called ϕ. The type of ϕ will be: $\phi : R(\Sigma, X) \to \{(V, P, \Sigma, S, X) \mid (V, P, \Sigma, S, X) \text{ is a CFG with variables }\}$. We will now inductively define ϕ by:

Definition 10 (ϕ)**.**

$$\begin{aligned}
&\phi(0) = (\{S\}, \{(S, \emptyset)\},\ \Sigma,\ S, \emptyset)\\
&\phi(1) = (\{S\}, \{(S, \epsilon)\},\ \Sigma,\ S, \emptyset)\\
&\phi(a) = (\{S\}, \{(S, a)\},\ \Sigma,\ S, \emptyset)\\
&\phi(x) = (\{S, x\},\ \{(S, x)\},\ \Sigma,\ S, X)\\
&\phi(r + s) = (V_r \cup V_s \cup \{S\},\ P_r \cup P_s \cup \{(S, S_r),\ (S, S_s)\},\ \Sigma, S, X_r \cup X_s)\\
&\phi(r \cdot s) = (V_r \cup V_s \cup \{S\},\ P_r \cup P_s \cup \{(S,\ S_r S_s)\},\ \Sigma,\ S, X_r \cup X_s)\\
&\phi(\mu x.r) = (V_r \cup \{x\},\ P_r \cup \{(x, S_r)\},\ \Sigma,\ x, X_r \setminus \{x\})
\end{aligned}$$

Where $\phi(r) = (V_r, P_r, \Sigma, S_r, X_r)$ and $\phi(s) = (V_s, P_s, \Sigma, S_s, X_s)$, and we assume that the variables in 'r' are not the same as the variables in 's'. If so, we rename them so that they do not overlap anymore, but we do not rename the variables that are in X. In each equation S is a fresh starting symbol and is not occurring in V_r, V_s or in X. Furthermore, V_r and V_s are disjoint, meaning that $V_r \cap V_s = \emptyset$. Note that in the case of $\phi(x)$ we have $x \in X$ and in the case of $\phi(\mu x.r)$ we have $x \in x_r$.

The whole point of this converting of context-free grammars and μ-regular expressions is to check whether the language remains. So we extended with variables X and we say that $\phi(r) = \phi(r, \emptyset)$.

7.1 Correctness

We will now prove the correctness of ϕ. This will be done via induction. The goal is to show that the language of a μ-regular expression is the same as the language of that μ-regular expression transformed into a CFG by ϕ.

As said in the previous section, we need to prove the correctness of intermediate steps. And in this steps we are dealing with unbound variables. This means that we are not dealing with μ-regular expressions but with μ-regular *pre*-expressions. So when we would follow the algorithm and come to the point of having $\eta(X)$ where X is an unbound variable, the input would not have the right type. This means we have to add a function that can map to languages from the unbound variables in the semantics of context-free grammars with variables. We introduce δ, which takes the unbound variables of a grammar as input and maps to the language of them. We say that:

- $\eta : V \to \mathcal{P}(\Sigma^*)$. The function that maps unbound variables to languages.
- $\delta : X \to \mathcal{P}(\Sigma^*)$. The function η from Chapter 3 named differently.
- $\delta \cup \eta : V \cup X \to \mathcal{P}(\Sigma^*)$. The combination of functions where

$$(\delta \cup \eta)(x) = \left\{ \begin{array}{ll} \delta(x), & \text{if } x \in V \\ \eta(x), & \text{if } x \in X \end{array} \right\}$$

The function $\overline{\delta \cup \eta}$ is then specified the same as $\bar{\eta}$ of Chapter 3. Having extra functions also means that our function f_T from Chapter 3 has to be updated from $f_X : \mathcal{P}(\Sigma^*)^V \to \mathcal{P}(\Sigma^*)$ into $f_T : \mathcal{P}(\Sigma^*)^{V \cup X} \to \mathcal{P}(\Sigma^*)$. The transformation of a context-free grammar into a language is as follows:

$$L(G, \eta) = (\text{lfp}\delta.\lambda T.f_T(\delta \cup \eta))(S)$$

Finally we can move on to the actual proof where we will prove that the language of a μ-regular expression is the same as the language of that same μ-regular expression transformed into a context-free grammar by ϕ:

Theorem 5. $\forall \eta \in X \to \mathcal{P}(\Sigma^*), r \in R(\Sigma, X) : \mathcal{L}(r, \eta) = L(\phi(r), \eta)$

Proof. Proof by induction, the first four are the base cases:

$$\begin{aligned}
L(\phi(0),\eta) &= L((\{S\},\,\{(S,0)\},\,\Sigma,\,S,\emptyset),\eta)(S)\\
&= (\text{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S)\\
&= (\text{lfp}\delta.S\mapsto\overline{\delta\cup\eta}(0))(S)\\
&= (S\mapsto\emptyset)(S)\\
&= \emptyset\\
&= \mathcal{L}(0,\eta)
\end{aligned}$$

$$\begin{aligned}
L(\phi(1),\eta) &= L((\{S\},\,\{(S,\epsilon)\},\,\Sigma,\,S,\emptyset),\eta)(S)\\
&= (\text{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S)\\
&= (\text{lfp}\delta.S\mapsto\overline{\delta\cup\eta}(\epsilon))(S)\\
&= (\text{lfp}\delta.S\mapsto\{\epsilon\})(S)\\
&= \{\epsilon\}\\
&= \mathcal{L}(1,\eta)
\end{aligned}$$

$$\begin{aligned}
L(\phi(a),\eta) &= L((\{S\},\,\{(S,a)\},\,\Sigma,\,S,\emptyset),\eta)(S)\\
&= (\text{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S)\\
&= (\text{lfp}\delta.S\mapsto\overline{\delta\cup\eta}(a))(S)\\
&= (\text{lfp}\delta.S\mapsto\{a\})(S)\\
&= \{a\}\\
&= \mathcal{L}(a,\eta)
\end{aligned}$$

$$\begin{aligned}
L(\phi(x),\eta) &= L((\{S,x\},\,\{(S,x)\},\,\Sigma,\,S,X),\eta)(S)\\
&= (\text{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S)\\
&= (\text{lfp}\delta.(S\mapsto\overline{\delta\cup\eta}(x),x\mapsto\emptyset))(S)\\
&= (\text{lfp}\delta.(S\mapsto\eta(x),x\mapsto\emptyset))(S)\\
&= \eta(x)\\
&= \mathcal{L}(x,\eta)
\end{aligned}$$

Assume it holds for the base cases and continue with the induction steps:

$$\begin{aligned}
L(\phi(r+s),\eta) &= L((V_r \cup V_s \cup \{S\},\, P_r \cup P_s \cup \{(S,S_r),\,(S,S_s)\},\, \Sigma, S, X_r \cup X_s),\eta)(S)\\
&= (\mathrm{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S_r) \cup (\mathrm{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S_s)\\
&= (\mathrm{lfp}\delta.(S_r \mapsto \eta(r))(S_r)) \cup (\mathrm{lfp}\delta.(S_s \mapsto \eta(s))(S_s))\\
&\overset{IH}{=} \mathcal{L}(r,\eta) \cup \mathcal{L}(s,\eta)\\
&= \mathcal{L}(r+s,\eta)
\end{aligned}$$

$$\begin{aligned}
L(\phi(r\cdot s),\eta) &= L((V_r \cup V_s \cup \{S\},\, P_r \cup P_s \cup \{(S,\, S_rS_s)\},\, \Sigma,\, S, X_r \cup X_s),\eta)(S)\\
&= (\mathrm{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S_r) \cdot (\mathrm{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(S_s)\\
&\overset{IH}{=} \mathcal{L}(r,\eta) \cdot \mathcal{L}(s,\eta)\\
&= \mathcal{L}(r\cdot s,\eta)
\end{aligned}$$

The last case is the one with a μ in front of the expression:

$$\begin{aligned}
L(\phi(\mu x.r),\eta) &= L(V_r \cup \{x\},\, P_r \cup \{(x,S_r)\},\, \Sigma,\, x, X_r \setminus \{x\}),\eta)\\
&= (\mathrm{lfp}\delta.\lambda T.f_T(\delta\cup\eta))(x)\\
&= (\mathrm{lfp}\delta.(x \mapsto \overline{\delta\cup\eta}(S_r), \lambda T.f_T(\delta\cup\eta)))(x)\\
&= \mathrm{lfp}K.((\mathrm{lfp}\delta.\lambda T.f_T(\delta\cup\eta[x\mapsto K]))(S_r)) && \text{(Bekić)}\\
&= \mathrm{lfp}K.L(\phi(r,\eta[x\mapsto K]))\\
&\overset{IH}{=} \mathrm{lfp}K.\mathcal{L}(r,\eta[x\mapsto K])\\
&= \mathcal{L}(\mu x.r,\eta)
\end{aligned}$$

All cases are proved to be correct so the algorithm is proven to be correct. $\square$

Chapter 8

Conclusions & Related Work

The appearance of μ-regular expressions is relatively new in the world of formal languages. Nevertheless, they can be useful and we gave a little peek on how they might be. They can for example help with the substitution of the Kleene star and they complete the triangle from Figure 1.1a. Hans Leiss also wrote about using a μ instead of a Kleene Star [5]. The main difference here is the scope of the usage and approach.

This thesis also showed how to go from a μ-regular expression to a context-free grammar and how to go from a context-free grammar to a μ-regular expression. Both ways are captured in algorithms which we proved to be correct. The last two transformations are individual results but strongly cohere with each other since they are both about transformations in the same cycle from Figure 1.1a. The work of Thiemann [7] is therefore also strongly related to this thesis and has been the main inspiration of researching μ-regular expressions.

The definitions of Chapter 2 (Preliminaries) are mainly inspired by the book Languages and Machines [6]. The given definitions can somewhat differ in styling and the given examples are self made.

This thesis did not really have open research questions. It rather had folklore theorems to be proved and algorithms to be provided. These goals have been achieved and one can therefore say that main goal is achieved.

Bibliography

[1] Hans Bekic. *Programming Languages and Their Definition: Selected Papers*, volume 177. Springer Science & Business Media, 1984.

[2] Seymour Ginsburg and H Gordon Rice. Two families of languages related to algol. *Journal of the ACM (JACM)*, 9(3):350–371, 1962.

[3] John E Hopcroft. *Introduction to automata theory, languages, and computation.* Pearson Education India, 2008.

[4] NR Krishnaswami. A typed, algebraic approach to parsing, 2017.

[5] Hans Leiß. Towards kleene algebra with recursion. In *International Workshop on Computer Science Logic*, pages 242–256. Springer, 1991.

[6] Thomas A Sudkamp and Alan Cotterman. *Languages and machines: an introduction to the theory of computer science*, volume 2. Addison-Wesley Reading, Mass., 1988.

[7] Peter Thiemann. Partial derivatives for context-free languages - from μ-regular expressions to pushdown automata. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 248–264, 2017.