SAARLAND UNIVERSITY
FACULTY OF NATURAL SCIENCES AND TECHNOLOGY I

BACHELOR'S THESIS

VERIFIED ALGORITHMS FOR CONTEXT-FREE GRAMMARS IN COQ

Author:
Jana Hofmann

Advisor:
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Bernd Finkbeiner, Ph.D.

Submitted: 10th August 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath:

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent:

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 10th August, 2016

Abstract

We give a basic formalization of context-free grammars and the languages they describe in the proof assistant Coq. We show the decidability of the word and the emptiness problem of context-free languages by giving and verifying decision procedures. Furthermore, we describe four grammar transformations: The elimination of empty rules, the elimination of unit rules, the separation of characters from the rest of the grammar and the binarization of a grammar. For each transformation, we show that it preserves the language of the grammar. Together, they yield a grammar which is in Chomsky normal form.

We aim for algorithms that are easy to understand and verify. Many of our results are obtained with finite iteration on lists. By using abstractions of fixed point and closure iterations, the correctness of the algorithms is intuitive and formally obtained with short proofs.

Acknowledgements

In the course of writing this thesis and also during those three years of my Bachelor studies there were many people who supported and influenced me. First of all, I would like to thank my advisor Prof. Smolka for his guidance and countless hours in which he provided valuable ideas. They had a great impact on the thesis. But also concerning the rest of my studies, he supported me in many ways.

Next, I have to thank Yannick, Chris and many fellow students who showed me how incredibly exciting computer science can be. Special thanks go to Kathrin, Clara and Yannick for proofreading the thesis. Moreover, I would like to thank all those who made (and make) me have such a great time in Saarbrücken. Finally, I am so grateful for the support of my family through all these years.

Contents

1 Introduction **11**
- 1.1 Related Work 12
- 1.2 Contribution 12

2 Preliminaries **15**
- 2.1 Finite Fixed Point Iteration 15
- 2.2 Finite Closure Iteration 16
- 2.3 Sublists 17
- 2.4 Basic Definitions 18

3 Context-Free Grammars **21**
- 3.1 Derivations 21
- 3.2 Operations on Grammars 25

4 Decidability of Emptiness Problem **27**

5 Decidability of Word Problem **31**

6 Elimination of Epsilon Rules **37**

7 Elimination of Unit Rules **43**

8 Elimination of Deterministic Variables **47**

9 Separation of Grammars **53**

10 Binarization of Grammars **57**

11 Chomsky Normal Form **61**
- 11.1 Transformation to Chomsky Normal Form 61

12 Conclusion **65**
- 12.1 Future Work 65

A Coq Realization **67**

B Use of Derivation Predicates **69**

C Variable and Function Names **70**

C.1 Variable Names 70

C.2 Transformations and Function Names 70

Bibliography **71**

Chapter 1

Introduction

Context-free grammars (CFGs) are an important concept in the field of computer science. They were introduced by Noam Chomsky in 1956 [2]. Initially, he intended to use them to describe natural languages. But natural languages turned out to be too complex to be described by CFGs. Nevertheless, they became an important tool to describe languages, although not natural languages. They are, however, perfect to describe programming languages, because they capture their recursive structure. It has been found that efficient parsing algorithms that work on a context-free grammar have cubic time complexity [5]. Besides the word problem, there are several other decidable problems for context-free grammars. For example, it is also decidable whether a language is finite or empty [5]. These are reasons why context-free grammars are so popular.

In this thesis, we give verified algorithms to decide the word and the emptiness problem. In addition, we discuss and verify four grammar transformations which yield a grammar in Chomsky normal form (CNF). CNF is a popular basis for further reasoning on context-free grammars, since it guarantees strong assumptions about the grammar. As an example, the CYK algorithm for deciding the word problem assumes the grammar to be in CNF [5]. Furthermore, the proof of the pumping lemma for context-free languages is based on the Chomsky normal form [8]. We show that every grammar can be transformed to CNF without modifying its languages.

All functions and proofs are realized in the proof assistant Coq [11]. We primarily aim to describe algorithms that are easy to understand, explain and verify. In order to obtain simple algorithms and proofs, we forego achieving optimal complexity. We benefit from the functional context we work in — on the computational level, Coq is a dependently typed functional programming language. As context-free grammars have been studied for several decades there are many textbooks describing algorithms, popular are [8, 5]. Most of them follow an imperative way of

arguing. Hence, as a second consequence of the functional setting, our algorithms and especially our proofs often differ from existing literature. Thereby, the formalization of context-free grammars permits new insights, apart from the relevance of the topic in its own right.

In the course of our work, we discovered that most of the problems we approached can be solved using iteration. Therefore, we make use of abstractions described in [10] which formalize general closure and fixed point iterations. Thereby, we can condense our tasks to describing a step function we have to prove correct.

1.1 Related Work

The formalization of context-free grammars and corresponding results using proof assistants has become popular recently [1, 3, 9]. Work has been investigated especially in verifying parsers and parser generators. Firsov and Uustalu [3] formalize a CYK parser using Agda. Jourdan, Pottier and Leroy [6] describe a validation function for LR(1)-Parsers in Coq. A second publication in Coq by Koprowski and Binsztok [7] deals with the verification of a parser interpreter that is based on parsing expression grammars which represents an alternative to context-free grammars.

Concerning normalization, this thesis was inspired by a second work of Firsov and Uusatalu [4]. They describe grammar transformations to CNF verified in Agda. By using parse tree transformations for their proofs, they show how to convert every parse tree to a corresponding one in CNF. In her PhD thesis [1], Barthwal gives a formalization of CFGs including transformations to CNF and Greibach normal form, which is another important normal form. In addition, she also describes a possibility of eliminating useless symbols in grammars. Her work is carried out using higher-order logic (HOL) embedded in Isabelle. Ramos [9] gives a comprehensive formalization of context-free grammars in Coq. This includes a proof that for every CFG, there exists a CNF. In addition, he shows several closure properties and a proof of the pumping lemma for CFGs.

1.2 Contribution

To the best of our knowledge, concerning the decidability of the word and emptiness problem, we seem to be the first to formalize proofs in Coq. Our approach for deciding the word problem does not follow traditional parser algorithms. It can be described as a generalized variant of CYK without assuming the grammar to be in CNF. Moreover, we describe four transformations of context-free grammars. Together, they yield a grammar in CNF. Each transformation works on general grammars without making restrictive assumptions. The four transformations are

- Elimination of ε-rules
- Elimination of unit rules
- Separation of characters
- Binarization

We aimed for simplicity, thus our complete formalization counts about 2,000 lines. The development is carried out in constructive type theory. Therefore, all functions are computable.

Chapter 2

Preliminaries

We will formalize context-free grammars using lists, which allow us to describe algorithmic ideas in an intuitive and compact way. We assume the reader to be familiar with well known functions like *map*, *concat* and *filter*.

When talking about decidability, we refer to the definition given in [10]. As Coq is based on constructive type theory, decidability is defined with an informative type rather than propositionally.

Definition 2.1 *A proposition P is **decidable**, if the informative type $\{P\} + \{\neg P\}$ is inhabited. Consequently, a predicate $p : X \to Prop$ is decidable, if $\forall x : X.\ \{p\,x\} + \{\neg p\,x\}$ is inhabited.*

We use the notation $X \xrightarrow{dec} Prop$ for decidable predicates and X_{dec} for a type with decidable equality.

2.1 Finite Fixed Point Iteration

Many results of this thesis are obtained by finite fixed point and closure iterations on context-free grammars. For both, we can use a formalization described in [10]. We first discuss the more general finite fixed point iteration (FFPI). For this thesis, we will only apply it to lists. However, the abstraction is not restricted to a special type.

Remember that for a function f we call x a **fixed point** of f, if $f\ x = x$. Given x and f, we will describe under which conditions we can compute a fixed point of f by applying f at most n times to x.

Lemma 2.2 (Finite Fixed Point Iteration) *Let X be a type and $f : X \to X$ a function.*

1. *Fixed Point. Let $\sigma : X \to \mathbb{N}$ and $x \in X$ such that for every number n either $\sigma(f^n\ x) > \sigma(f^{n+1}\ x)$ or $f^n\ x$ is a fixed point of f. Then $f^{\sigma x}\ x$ is a fixed point of f.*

2. *Induction. Let* $p : X \to Prop$ *and* $x \in X$ *such that* $p\ x$ *and* $\forall z.\ p\ z \to p(f\ z)$*. Then* $p(f^n\ x)$ *for every number* n*.*

Lemma 2.2 states that if there is a size function σ for X, that decreases with every step of f until a fixed point is reached, then we will get a fixed point after at most $\sigma\ x$ steps. In addition, if f preserves some property x fulfils, then the fixed point (and any intermediate step) will also have this property.

2.2 Finite Closure Iteration

Finite closure iteration (FCI) is a technique that can only be applied to lists. It successively adds elements from a list N to a list M which is initially empty. The elements to be added to M are determined by a predicate $step$. For each list M and candidate x from N, $step$ returns $\top$ if x can be added to M and $\bot$ otherwise. Thereby, a list that is closed with respect to $step$ and N is constructed. FCI obviously terminates as N is finite.

Formally, FCI can be described with the following lemma.

Lemma 2.3 (Finite Closure Iteration) *Let* X *be a type with decidable equality,* $step : list\ X \to X \to Prop$ *be a decidable predicate, and* N *be a list over* X*. Then one can construct a list* $M \subseteq S$ *such that:*

1. *Closure. If* $step\ M\ x$ *and* $x \in N$*, then* $x \in M$*.*
2. *Induction. Let* $p : X \to Prop$ *such that* $step\ xs\ x \to p\ x$ *for all* $xs \subseteq p$ *and* $x \in N$*. Then* $M \subseteq p$*.*

This means that in order to apply this lemma we need:

1. a type X_{dec} which will be inferred automatically
2. a maximal list $N : list\ X$
3. a step predicate $step : list\ X \to X \to Prop$
4. a proof that $step$ is decidable.

For the resulting list M, Lemma 2.3 states that M is closed with respect to $step$ and N. Moreover, if $step$ inductively preserves some property for elements of N, then we can also assume it to hold for all elements of M.

Finite closure iteration is a strengthening of finite fixed point iteration. In fact, FCI can be proved using FFPI [10]. The proof shall not be discussed further at this point.

2.3 Sublists

Apart from iterations the notion of sublists and segments will prove itself helpful. We receive a sublist of a list $xs : list\ X$ by removing some elements from xs. Which elements can be removed is indicated by a predicate $p : X \to Prop$. In contrast to list inclusion ($\subseteq$), sublists respect the order and the number of appearances of elements in the original list. We realize the concept of sublists with an inductive definition. We write $xs \precsim_p ys$ if xs is a sublist of ys with respect to p.

$$\frac{}{nil \precsim_p nil} \qquad \frac{xs \precsim_p ys \quad p\ y}{xs \precsim_p y :: ys} \qquad \frac{xs \precsim_p ys}{y :: xs \precsim_p y :: ys}$$

Note that if xs is an element of the powerlist of ys, then $xs \precsim_\top ys$.

We give some basic but useful properties of sublists that can be proven easily. We use $+\!\!+$ for list concatenation.

Fact 2.4 (Properties of sublists)

1. $xs \precsim_p xs$ (Reflexivity)
2. *If* $xs \precsim_p ys$ *and* $ys \precsim_p zs$*, then* $xs \precsim_p zs$. (Transitivity)
3. *If* $xs_1 \precsim_p ys_1$ *and* $xs_2 \precsim_p ys_2$*,* *then* $xs_1 +\!\!+ xs_2 \precsim_p ys_1 +\!\!+ ys_2$. (Closure under Concatenation)
4. *If* p *and* p' *are equivalent for all* s *and* $xs \precsim_p ys$*, then* $xs \precsim_{p'} ys$.
5. *If* $xs \precsim_p ys$ *and* $p\ x$ *for all* x *in* xs*, then* $p\ x$ *for all* x *in* ys.
6. *If* $xs \precsim_p ys_1 +\!\!+ ys_2$ *then there exist* xs_1, xs_2 *such that* $xs_1 \precsim_p ys_1$ *and* $xs_2 \precsim_p ys_2$. (Split)
7. *If* $xs \precsim_p ys$*, then* $xs \subseteq ys$. (Inclusion)

The functional characterization of sublists is straightforward. The following function computes all sublists of a list.

```
slists : ∀ X. (X -dec-> Prop) → list X → list (list X)
slists p []       := [[]]
slists p (s::u)   := if p s then slists p u ++ map (cons s) (slists p u)
                     else map (cons s) (ppower p u)
```

Fact 2.5 $xs \precsim_p ys$ *iff* $xs \in$ slists $p\ ys$.

A segment is a sublist where all elemens appear next to each other in the original list. This leads directly to a definition of segments:

Definition 2.6 *A list xs is a **segment** of ys ($xs \precsim_s ys$), if there exists xs_1 and xs_2 such that $ys = xs_1 + xs + xs_2$.*

For example, the list $[1, 3, 4]$ is a sublist of $[1, 2, 3, 4]$ but not a segment. Every segment is also a sublist but not vice versa. Again we state properties of segments that we will be helpful in subsequent proofs.

Fact 2.7 (Properties of segments)

1. $xs \precsim_s xs$ (Reflexivity)
2. $\varepsilon \precsim_s xs$
3. *If $xs \precsim_s ys$ and $ys \precsim_s zs$, then $xs \precsim_s zs$.* (Transitivity)

We can compute all segments of a list:

```
segms : ∀ X_dec. list X → list (list X)
segms []         := [[]]
segms (x :: xs)  := let sxs := segms xs
                    in sxs ++ map (cons x)
                                  (filter (λys. ∃zs. xr = ys++zs) sxs)
```

The function segms is a functional characterization of $\precsim_s$.

Fact 2.8 $xs \precsim_s ys$ *iff* $xs \in$ segms ys.

2.4 Basic Definitions

We give some helpful list functions. We define $*_f$ to be an operation that concatenates all elements of two lists using a given function f.

```
• *_f • : ∀ X Y Z. list X → list Y → list Z
[] *_f ys         := []
(x :: xs) *_f ys  := map (f x) ys ++ xs *_f ys
```

For example, $a *_{pair} b$ yields the classical cross product using the pair constructor.

The notion of projection is well known and its lifting to lists is straightforward.

```
π1 : ∀ X Y. list (X × Y) → list X
π1 xs := map (λ(x, y). x) xs
```

$$\pi_2 : \forall\ X\ Y.\ \mathit{list}\ (X \times Y) \to \mathit{list}\ Y$$
$$\pi_2\ xs := \texttt{map}\ (\lambda(x,\ y).\ y)\ xs$$

Finally, we give a substitution function that replaces all appearances of an element in a list with a second list.

$$\begin{aligned}
&\bullet[\bullet \to \bullet] : \forall\ X_{dec}.\ \mathit{list}\ X \to X \to \mathit{list}\ X \to \mathit{list}\ X \\
&[\,][y \to ys] := [\,] \\
&(x :: xs)[y \to ys] := \texttt{if}\ x = y\ \texttt{then}\ ys +\!\!+ xs[y \to ys] \\
&\qquad\qquad\qquad\qquad \texttt{else}\ x :: xs[y \to ys]
\end{aligned}$$

We will make use of the following properties of substitution:

Fact 2.9 (Properties of substitution)

1. $(xs_1 +\!\!+ xs_2)[y \to ys] = xs_1[y \to ys] +\!\!+ xs_2[y \to ys]$ (Split)
2. *If* $y \notin xs$, *then* $xs[y \to ys] = xs$. (Skip)
3. $|\ xs\ | = |\ xs[y \to y']\ |$
4. *If* $y' \notin xs$, *then* $(xs[y \to y'])\ [y' \to y] = xs$. (Reversible substitution)

Chapter 3

Context-Free Grammars

To characterize context-free grammars in a formal way, we give the following definitions of **variables** (also called non-terminals) and **characters** (terminals).

$$\begin{aligned} var &:= n && (n \in \mathbb{N}) \\ char &:= n && (n \in \mathbb{N}) \\ symbol &:= var \mid char \end{aligned}$$

For variables we use Roman capital letters (A, B, C...) whereas for characters and symbols we use small letters (s for symbols and $a, b, c, \dots$ for characters). In Appendix C, you can find all variables namings. **Phrases** are lists of symbols, they are indicated by u, v, w. **Rules** are pairs of a variable and a phrase. A **grammar** G is a list of rules.

$$\begin{aligned} phrase &:= \mathscr{L}(symbol) \\ rule &:= var \times phrase \\ grammar &:= \mathscr{L}(rule) \end{aligned}$$

To improve readability, we write $A\backslash aBc$ for a rule $(A, [a, B, c])$. In addition, we write su for a phrase $s :: u$ and uv instead of $u +\!\!+ v$. The distinction between those two operators is possible with the naming of the variables.

3.1 Derivations

A context-free grammar describes a context-free rewriting system. We say that a phrase uAw rewrites to uvw, if there is a rule $A\backslash v$ in the grammar. Consequently, v is **derivable** from u (i.e. $u \overset{G}{\Rightarrow} v$), if v can be reached from u in arbitrary many rewriting steps. There are many ways to formalize the property of derivability. A

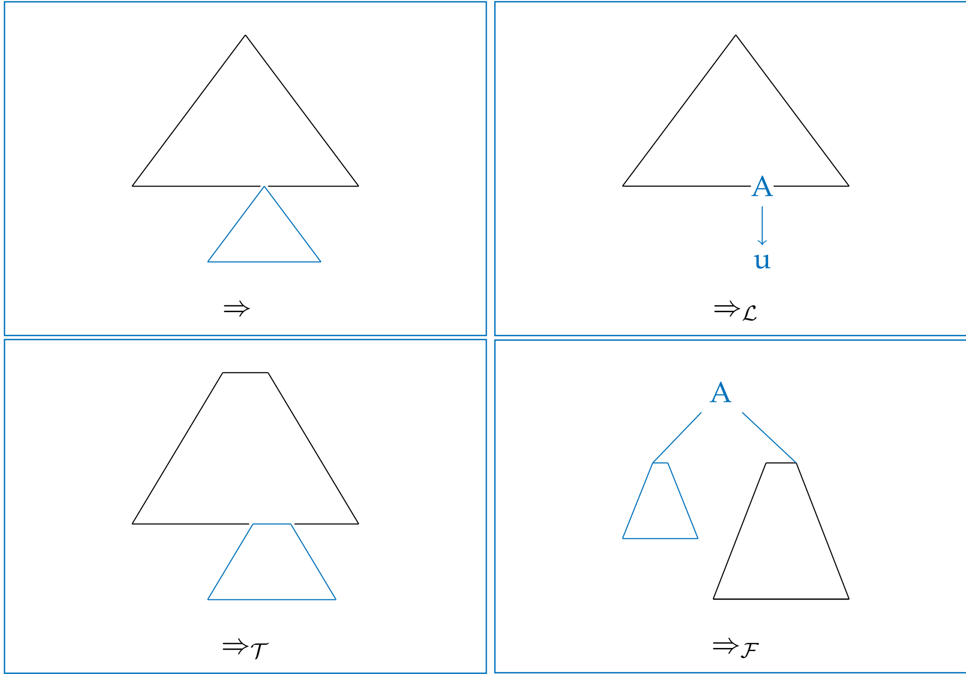

Figure 3.1: Different derivation predicates.

direct implementation of rewriting yields the following predicate:

$$\frac{}{u \overset{G}{\Rightarrow}_{\mathcal{I}} u} \qquad \frac{A \backslash v \in G}{uAw \overset{G}{\Rightarrow}_{\mathcal{I}} uvw} \qquad \frac{u \overset{G}{\Rightarrow}_{\mathcal{I}} v \quad v \overset{G}{\Rightarrow}_{\mathcal{I}} w}{u \overset{G}{\Rightarrow}_{\mathcal{I}} w}$$

The second rule performs one step of rewriting. The first and the third rule add reflexivity and transitivity. However, other characterizations of derivability turn out to be more convenient for formal proofs. Actually, we will make use of several definitions throughout this thesis. Thereby, we can choose from different induction principles, depending on which gives us the shortest and most intuitive proofs. Appendix B sums up which predicate is used in which chapter for the main proofs. In Fig. 3.1, we illustrate four different characterizations of derivability which are discussed below.

In contrast to $\Rightarrow_{\mathcal{I}}$, the following predicate defines an induction principle which isolates the application of a rule and adds the congruent transitive closure. Therefore, it implicitly states that derivations are context-free.

$$\frac{}{u \overset{G}{\Rightarrow}_{\mathcal{T}} u} \qquad \frac{A \backslash u \in G}{A \overset{G}{\Rightarrow}_{\mathcal{T}} u} \qquad \frac{u \overset{G}{\Rightarrow}_{\mathcal{T}} u_1 v u_2 \quad v \overset{G}{\Rightarrow}_{\mathcal{T}} w}{u \overset{G}{\Rightarrow}_{\mathcal{T}} u_1 w u_2}$$

As we often do grammar transformations, this predicate can be very useful, since in the second case we can concentrate on the rule while the third case is often quite simple as we can work with two inductive hypotheses.

In most cases, our lemmas will be statements about derivability beginning with a single variable. Therefore, we introduce a non-symmetric variant giving us an appropriate induction principle. We will mostly use this characterization, so we just use the $\Rightarrow$-notation.

$$\frac{}{A \overset{G}{\Rightarrow} A} \qquad \frac{A \backslash u \in G}{A \overset{G}{\Rightarrow} u} \qquad \frac{A \overset{G}{\Rightarrow} uBw \quad B \overset{G}{\Rightarrow} v}{A \overset{G}{\Rightarrow} uvw}$$

For some proofs it will be convenient to argue about a single derivation step. The following predicate captures the right-linear character of a derivation.

$$\frac{}{A \overset{G}{\Rightarrow}_{\mathcal{L}} A} \qquad \frac{A \overset{G}{\Rightarrow}_{\mathcal{L}} uBw \quad B \backslash v \in G}{A \overset{G}{\Rightarrow}_{\mathcal{L}} uvw}$$

We finally give a derivation predicate that is inspired by parse trees which often serve as basis for formalizations of context-free grammars. In contrast to linear derivations, trees allow a two-dimensional bottom-up perspective.

$$\frac{}{u \overset{G}{\Rightarrow}_{\mathcal{F}} u} \qquad \frac{A \backslash u \in G \quad u \overset{G}{\Rightarrow}_{\mathcal{F}} v}{A \overset{G}{\Rightarrow}_{\mathcal{F}} v} \qquad \frac{s \overset{G}{\Rightarrow}_{\mathcal{F}} u \quad v \overset{G}{\Rightarrow}_{\mathcal{F}} w}{sv \overset{G}{\Rightarrow}_{\mathcal{F}} uw}$$

Seen as a parse tree, the second rule implements the conversion of a forest (a list of trees) to a single tree with a rule. The second rule captures the concatenation of a derivation tree with a forest. Compared to conventional definitions of parse trees like in [4], this predicate is not nested inductive but nevertheless gives us strong inductive hypotheses.

Note that all characterizations are defined as predicates and not as types. Thus, we can only speak about *derivability*, not about concrete *derivations*. Actually, there might be several derivations for the same phrase. For our purpose, the notion of derivability turned out to be powerful enough and yields simple proofs. We want to prove that all these characterizations are equivalent. An illustration of how we will prove their equivalence can be found in Fig. 3.2. We start by proving that $\Rightarrow$, $\Rightarrow_{\mathcal{L}}$ and $\Rightarrow_{\mathcal{T}}$ are equivalent.

Lemma 3.1 $A \overset{G}{\Rightarrow} u \leftrightarrow A \overset{G}{\Rightarrow}_{\mathcal{L}} u$.

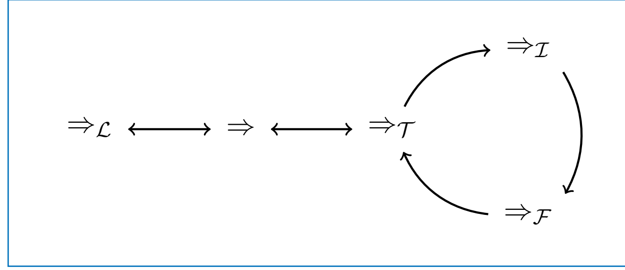

Figure 3.2: Equivalence proofs of derivation predicates.

Proof By induction on the derivation predicate for both parts. For the only-if part, we need a nested induction on the second inductive hypothesis. ∎

Lemma 3.2 $A \overset{G}{\Rightarrow} u \leftrightarrow A \overset{G}{\Rightarrow}_{\mathcal{T}} u$.

Proof By induction on $\Rightarrow$ and $\Rightarrow_{\mathcal{T}}$, respectively. ∎

To prove the equivalence of $\Rightarrow_{\mathcal{T}}$, $\Rightarrow_{\mathcal{F}}$ and $\Rightarrow_{\mathcal{I}}$, we first establish certain properties of $\Rightarrow_{\mathcal{F}}$.

Lemma 3.3 (Splitting) *The derivation of* $\Rightarrow_{\mathcal{F}}$ *can be split into two parts:*
Assume $u_1u_2 \overset{G}{\Rightarrow}_{\mathcal{F}} v$*. Then there are* v_1 *and* v_2 *such that* $v = v_1v_2$ *and* $u_1 \overset{G}{\Rightarrow}_{\mathcal{F}} v_1$ *and* $u_2 \overset{G}{\Rightarrow}_{\mathcal{F}} v_2$*.*

Proof By induction on $u_1u_2 \overset{G}{\Rightarrow}_{\mathcal{F}} v$ and a second induction on u_1u_2 in the first case. ∎

Lemma 3.4 (Compatibility with Concatenation) $\Rightarrow_{\mathcal{F}}$ *is compatible with concatenation:*
If $u_1 \overset{G}{\Rightarrow}_{\mathcal{F}} v_1$ *and* $u_2 \overset{G}{\Rightarrow}_{\mathcal{F}} v_2$*, then* $u_1u_2 \overset{G}{\Rightarrow}_{\mathcal{F}} v_1v_2$*.*

Proof By induction on $u_1 \overset{G}{\Rightarrow}_{\mathcal{F}} v_1$ and a second induction on u_1 in the first case. ∎

Lemma 3.5 (Transitivity) $\Rightarrow_{\mathcal{F}}$ *is transitive:*
If $u \overset{G}{\Rightarrow}_{\mathcal{F}} v$ *and* $v \overset{G}{\Rightarrow}_{\mathcal{F}} w$*, then* $u \overset{G}{\Rightarrow}_{\mathcal{F}} w$*.*

Proof By induction on $u \overset{G}{\Rightarrow}_{\mathcal{F}} v$ using the splitting property (Lemma 3.3). ∎

Now we are able to prove the equivalence of $\Rightarrow_{\mathcal{F}}$, $\Rightarrow_{\mathcal{I}}$ and $\Rightarrow_{\mathcal{T}}$.

Lemma 3.6 *We need three proofs:*

1. $u \overset{G}{\Rightarrow}_{\mathcal{F}} v \rightarrow u \overset{G}{\Rightarrow}_{\mathcal{T}} v$*.*

2. $u \overset{G}{\Rightarrow}_{\mathcal{T}} v \to u \overset{G}{\Rightarrow}_{\mathcal{I}} v$.
3. $u \overset{G}{\Rightarrow}_{\mathcal{I}} v \to u \overset{G}{\Rightarrow}_{\mathcal{F}} v$.

Proof All implications can be proved using induction on the derivation predicate:

1. By induction on $u \overset{G}{\Rightarrow}_{\mathcal{F}} v$.
2. By induction on $u \overset{G}{\Rightarrow}_{\mathcal{T}} v$.
3. By induction on $u \overset{G}{\Rightarrow}_{\mathcal{I}} v$ using transitivity and compatibility with concatenation of $\Rightarrow_{\mathcal{F}}$. ∎

Because of the established equivalences between the different predicates, we can assume the properties we proved for $\Rightarrow_{\mathcal{F}}$ to hold for all of them. An additional extension property will be helpful.

Lemma 3.7 (Extension) *Assume* $u \overset{G}{\Rightarrow}_{\mathcal{T}} v$ *and* $G \subseteq G'$. *Then* $u \overset{G'}{\Rightarrow}_{\mathcal{T}} v$.

Proof By induction on $u \overset{G}{\Rightarrow} v$. ∎

Phrases that contain only characters are called **words** (x, y, z). A **language** of a grammar is defined by the sets of words we can get by starting the derivation with a certain variable. We define the languages of a grammar using a predicate $\mathcal{L} : \mathit{grammar} \to \mathit{var} \to \mathit{phrase} \to \mathit{Prop}$.

$$\mathcal{L}_G^A\, x := A \overset{G}{\Rightarrow} x$$

Note that by writing x instead of u, we require x to be a word, which can not be derived further.

3.2 Operations on Grammars

The **domain** $\mathcal{D}$ and the **range** $\mathcal{R}$ are defined by the right- and left-hand sides of a grammar.

$$\begin{aligned}
&\mathcal{D} : \mathit{grammar} \to \mathit{list}\ \ \mathit{symbol} \\
&\mathcal{D}\ \texttt{[]} := \texttt{[]} \\
&\mathcal{D}\ (A\backslash u\ ::\ G) := A\ ::\ \mathcal{D}\ G
\end{aligned}$$

$$\begin{aligned}
&\mathcal{R} : \mathit{grammar} \to \mathit{list}\ \ \mathit{phrase} \\
&\mathcal{R}\ \texttt{[]} := \texttt{[]} \\
&\mathcal{R}\ (A\backslash u\ ::\ G) := u\ ::\ \mathcal{R}\ G
\end{aligned}$$

We can compute a list $\mathcal{S}$ of all symbols of a grammar.

$$\begin{aligned}
&\mathcal{S} : \mathit{grammar} \to \mathit{list}\ \ \mathit{symbol} \\
&\mathcal{S}\ \texttt{[]} := \texttt{[]} \\
&\mathcal{S}\ (A\backslash u\ ::\ G) := A\ ::\ u \mathbin{+\!\!+} \mathcal{S}\ G
\end{aligned}$$

We will write $\mathcal{S}_G$ instead of $\mathcal{S}\ G$ (equally for $\mathcal{D}$ and $\mathcal{R}$). For the sake of readability, we might also drop the index G, if it is clear from the context.

For some operations on context-free grammars we will need to introduce fresh variables to the grammar. As the definition of symbols is based on natural numbers, we can lift $<$ and $\leq$ up to symbols. We call the symbol s **fresh** for G, if $\forall s' \in \mathcal{S}_G.\ s' < s$.

Fact 3.8 *For every grammar G there is a variable A that is fresh for G.*

The functional characterizations of maximal and fresh symbols are obvious and shall be omitted here. We will assume a generator $\mathsf{fresh} : \mathit{grammar} \to \mathit{var}$ that gives us a variable that is fresh for agrammar.

Discussion

When Chomsky first described context-free grammars [2], he defined them as a tuple (N, T, P, S). The set N consists of all variables, T of all characters, P is the set of rules and S is the start variable. As the set N and T are given through P, we do not mention them explicitly. In addition, we do not care about start variables as our results will hold generally for all possible start variables.

The way we defined derivability is unusual. Some formalizations are based on parse trees [4], others define $\Rightarrow$ to describe a single step and then add the reflexive and transitive closure ($\Rightarrow^*$) [1]. Furthermore, there are many possibilities to describe derivations with inductive predicates that we did not discuss. A thesis like this develops in the curse of work and time is limited. After finishing our work, we consider other possibilities to define grammars and derivations which might be interesting for future work. For example, we could have assumed the grammar to be binary. This means that grammars are described without lists on the right-hand sides. Thereby, one may shorten some proofs. Alternatively, a derivation predicate that could have replaced $\Rightarrow_{\mathcal{F}}$ is the following:

$$\frac{}{\varepsilon \overset{G}{\Rightarrow}_{\mathcal{F}} \varepsilon} \qquad \frac{A \backslash u \in G \quad u \overset{G}{\Rightarrow}_{\mathcal{F}} u' \quad v \overset{G}{\Rightarrow}_{\mathcal{F}} v'}{Av \overset{G}{\Rightarrow}_{\mathcal{F}} u'v'} \qquad \frac{u \overset{G}{\Rightarrow}_{\mathcal{F}} u'}{su \overset{G}{\Rightarrow}_{\mathcal{F}} su'}$$

This predicate captures the intuition of rewriting parse trees in phrases (rule 2) and skipping characters (rule 3).

Chapter 4

Decidability of Emptiness Problem

We start by proving that for every grammar G and variable A it is decidable whether $\mathcal{L}_G^A$ is **empty**, i.e. $\neg\exists x.\ \mathcal{L}_G^A\ x$. Proving that the word problem is decidable is not sufficient since we would have to check infinitely many words. To decide whether $\mathcal{L}_G^A$ is empty, we need the notion of **productive variables**. We call a variable productive, if it derives a word x. Hence, a language $\mathcal{L}_G^A$ is empty, if A is *not* productive. Computing the set of productive variables is not difficult. If there is a rule $A\backslash a$ in G, then A is obviously productive. Additional, for a rule $A\backslash u$, A is productive, if every variable in u is productive. We compute the set of all productive variables and $\mathcal{L}_G^A$ is empty, if A is *not* in this set.

First, we give an inductive predicate $\mathit{productive}$. To simplify definitions and proofs, we lift the notion of productive variables to symbols.

$$\frac{}{\mathit{productive}_G\ a} \qquad \frac{A\backslash u \in G \quad \forall s \in u.\ \mathit{productive}_G\ s}{\mathit{productive}_G\ A}$$

We prove that the inductive characterization of $\mathit{productive}$ corresponds with our direct definition.

Lemma 4.1 $\mathit{productive}_G\ A$ *iff* $\exists x.\ A \overset{G}{\Rightarrow} x$.

Proof Instead of $\Rightarrow$ we prove the claim with $\Rightarrow_{\mathcal{F}}$ as this characterization matches the right-inductive definition of $\mathit{productive}$ best.

$\rightarrow$ By induction on $\mathit{productive}_G\ A$ and a nested induction on u in the second case (see the rules of $\mathit{productive}$).

$\leftarrow$ We prove the generalized statement $u \overset{G}{\Rightarrow}_{\mathcal{F}} v \rightarrow (\forall s \in v.\ \mathit{productive}_G\ s) \rightarrow \forall s \in u.\ \mathit{productive}_G\ s$. It subsumes the claim as in a word, all symbols are productive. The proof goes through by induction on $u \overset{G}{\Rightarrow}_{\mathcal{F}} v$. ∎

To compute the set of all productive symbols of a grammar, we make use of finite closure iteration (Section 2.2). Remember that we need to supply a maximal list N that contains all productive symbols. We can pick $\mathcal{S}_G$, as all productive symbols are symbols of G. In addition, we need to define a step predicate. The step predicate follows the structure of the inductive characterization of $\mathit{productive}$.

$$\begin{aligned}
&\mathsf{step} \;:\; \mathit{grammar} \to \mathit{list\ symbol} \to \mathit{symbol} \to \mathit{Prop} \\
&\mathsf{step}_G\ M\ s \;:=\; s = a \lor s = A \\
&\qquad\qquad\qquad\qquad \land\ \exists u.\ A\backslash u \in G \\
&\qquad\qquad\qquad\qquad\qquad \land\ \forall s' \in u.\ s' \in M
\end{aligned}$$

With step, every symbol which is added to the resulting list is a character or a productive variable, since it can be derived in one step to a list of productive symbols. Since G is finite, step is obviously decidable.

Fact 4.2 $\mathsf{step}_G\ M\ s$ *is decidable.*

FCI yields the required list of all productive symbols.

$$\mathbb{P}_G := \text{FCI}\ \mathsf{step}_G\ \mathcal{S}_G$$

To prove that $\mathbb{P}_G$ is correct, we prove that a symbol s is in $\mathbb{P}_G$ iff s is productive in G. Using the closure and the induction lemma proved for FCI (Lemma 2.3), the proof is straightforward.

Lemma 4.3 *Assume* $s \in \mathcal{S}_G$ *and* $\mathit{productive}_G\ s$*. Then* $s \in \mathbb{P}_G$*.*

Proof By induction on $\mathit{productive}$.

- $s = a$. With the closure lemma of FCI, we need to show that $s \in \mathcal{S}_G$ and step $\mathbb{P}_G$ s holds. The first statement holds by assumption. The latter holds because s is a character by induction.
- $s = A$ and $A\backslash u \in G$. Again, using the closure lemma and the assumption that $s \in \mathcal{S}_G$, only step $\mathbb{P}_G\ s$ is left to show. Because of the inductive hypothesis, all symbols in u are in $\mathbb{P}_G$. Therefore, step $\mathbb{P}_G\ A$ holds. ∎

Lemma 4.4 *If* $s \in \mathbb{P}_G$*, then* $\mathit{productive}_G\ s$*.*

Proof We apply the induction lemma of FCI. Thereby, we need to prove that if in a set of symbols M, all elements are productive and step $M\ s$ holds, then s is productive. Because of step $M\ s$, there are two cases.

- $s = a$. Then s is productive.

- $s = A$ and $A \backslash u \in G$. All symbols of u are in M. Then by induction, they are all productive and therefore, A is productive. ∎

Combining Lemma 4.3 and Lemma 4.4 gives us the intended property.

Corollary 4.5 *Assume $s \in \mathcal{S}_G$. Then $s \in \mathbb{P}_G$ iff $productive_G\ s$.*

As $\mathbb{P}_G$ is computable, we can decide whether $\mathcal{L}_G^A$ is empty, if A is a symbol of G. Otherwise, the language must be empty since a symbol that is not in $\mathcal{S}_G$ can not derive anything but itself.

Theorem 4.6 *It is decidable whether $\mathcal{L}_G^A$ is empty.*

Chapter 5

Decidability of Word Problem

We want to prove that context-free languages are decidable, i.e. we aim to decide $\mathcal{L}_G^A\, x$. Instead of deciding only $A \overset{G}{\Rightarrow} x$, we rather decide the more general problem $A \overset{G}{\Rightarrow} u$. We assume G and u to be fixed. Intuitively, we want to compute for all segments of u, from which symbols they can be derived.

To realize this idea, we introduce a type $item$.

$$item := symbol \times phrase$$

Our aim is to construct a list M : $list\ item$ such that $(s, v) \in M$ if and only if s derives v and v is a segment of u. We construct this list with a bottom-up algorithm: Every pair (s, s) is in M, if s appears in u. Then, we combine pairs $(s_0, v_0), \ldots, (s_n, v_n)$ of M, such that there is a rule $B \backslash s_0 \ldots s_n$ in G. We know that s_i derives v_i for every pair (s_i, v_i) and therefore, B derives $v_0 \ldots v_n$. So if $v_0 \ldots v_n$ is a segment of u, we can add the pair $(B, v_0 \ldots v_n)$ to M. Finally, as u is a segment of u, we have $(A, u) \in M$ iff A derives the phrase u.

Consider the following example.

Example 5.1

Let G and u be given as

$$\begin{aligned} G &:= A\backslash aBA \qquad B\backslash BB \\ & A\backslash a \qquad\quad\; B\backslash b \\ u &:= abba \end{aligned}$$

The following figure illustrates the algorithm.

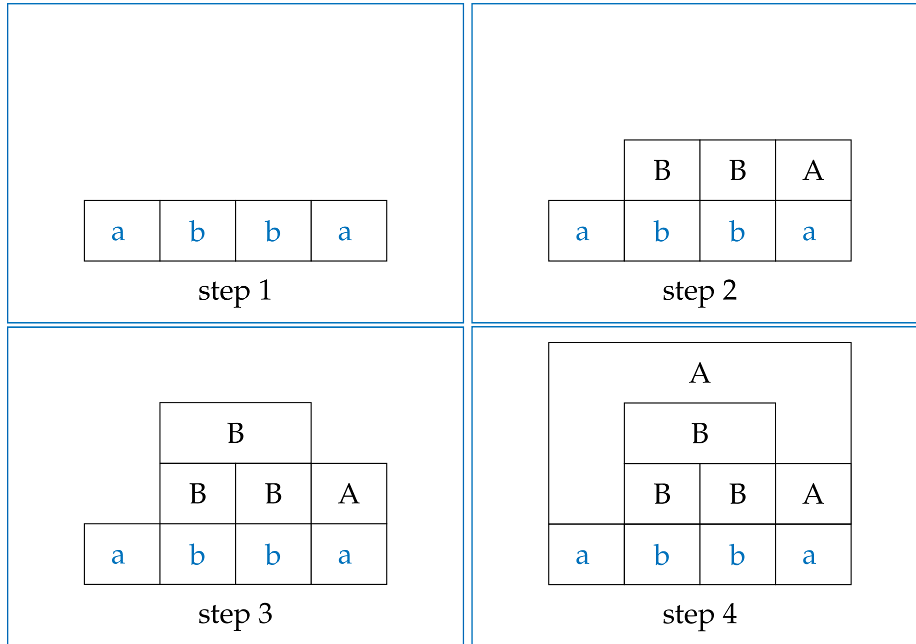

Figure 5.1: Illustration of the decision algorithm.

1. First, we add (a, a) and (b, b) to M.
2. Now we add (A, a) (B, b) since $(b, b) \in M$, $(a, a) \in M$, $A\backslash a \in G$ and $B\backslash b \in G$.
3. Next, we add (B, bb) because of $(B, b) \in M$, $B\backslash BB \in G$ and bb is a segment of u. Note that every element of M can be used multiple times to construct a new item.
4. Finally, we add $(A, abba)$ since $(a, a),\ (B, bb), (A, a) \in M$ and $A\backslash aBA \in G$. Of course, $abba$ is a segment of u.

Since (A, u) is in M, A can derive u.

We now want to formalize the idea. To prove it correct, the tree predicate $\Rightarrow_{\mathcal{F}}$ is suited best. Similar to the algorithm we describe, it is defined in a bottom-up manner. We aim to prove the following lemma.

Lemma 5.1 *Assume u and G are fixed and $s \in \mathcal{S}_G$. Then we can compute a list M such that $(s, v) \in M$ iff $s \overset{G}{\Rightarrow}_{\mathcal{F}} v$ and v is a segment of u.*

We will later see why $s \in \mathcal{S}_G$ is a necessary assumption. The intuition of M can be

described formally with the aid of an inductive predicate:

$$\frac{s \in u}{(s,s) \in M} \qquad \frac{v \precsim_s u \quad M' \subseteq M \quad A\backslash(\pi_1\ M') \in G \quad \mathsf{concat}(\pi_2\ M') = v}{(A,v) \in M}$$

We obtain the deciding item list with FCI.

$$\begin{aligned} \mathsf{items}_{G,u} &:= \mathcal{S}_G *_{pair} (\mathsf{segms}\ u) \\ \mathsf{step}_G\ M\ (s,v) &:= v = s \lor s = A \land \\ &\qquad\qquad \exists M' \subseteq M.\ A\backslash(\pi_1\ M') \in G \land v = \mathsf{concat}(\pi_2\ M') \\ \mathbb{D}_{G,u} &:= \mathsf{FCI}\ \mathsf{step}_G\ \mathsf{items}_{G,u} \end{aligned}$$

For the following, we fix the variables G and u and can therefore omit the index $_{G,u}$.

The predicate step implements the inductive intuition of M for FCI. The list items is the maximal list we have to provide to use FCI: Note that all pairs (s, v) for which $s \overset{G}{\Rightarrow}_{\mathcal{F}} v$ and $v \precsim_s u$ could hold are contained in items. Now it becomes clear why Lemma 5.1 assumes s to be in $\mathcal{S}_G$. Because of the definition of items, Lemma 5.1 only holds for $s \in \mathcal{S}_G$. What is left to prove is the decidability of step since $item$ is obviously of decidable equality.

Lemma 5.2 step M (s, v) *is decidable.*

Proof If s is a character, then step M (s, v) holds. The case of $s = A$ is more difficult as there are infinitely many M' such that $M' \subseteq M$. The isolated components $A\backslash(\pi_1\ M') \in G$ and $v = \mathsf{concat}(\pi_2\ M')$ are decidable. However, note that proving the existence of an M' such that $A\backslash(\pi_1\ M') \in G$ is not strong enough. But as G and M are finite, we can compute the set of *all* possible lists $M' \subseteq M$, where $A\backslash(\pi_1\ M') \in G$ holds. Then, we can decide for each element whether $v = \mathsf{concat}(\pi_2\ M')$. ∎

We first prove the only-if-part of Lemma 5.1. We generalize the lemma to do induction on the derivation predicate. Basically, we prove that every derivation can be split into parts such that every part is in $\mathbb{D}$.

Lemma 5.3 *Let $w \overset{G}{\Rightarrow}_{\mathcal{F}} v$ and $v \precsim_s u$. Assume that all symbols in w are in $\mathcal{S}_G$. Then there is a list M such that $\pi_1\ M = w$, $\mathsf{concat}(\pi_2\ M) = v$ and $M \subseteq \mathbb{D}$.*

Note that this statement gives us an inductive hypothesis that fits to step. If w consists of only one symbol, then M has only one element which is in $\mathbb{D}$.

Proof (Lemma 5.3) By induction on $w \overset{G}{\Rightarrow}_{\mathcal{F}} v$. We distinguish three cases.

- $w = v$. Let $v = [s_0, s_1, \dots]$. Choose $M := [(s_0, [s_0]), (s_1, [s_1]), \dots]$. Obviously, $\pi_1\ M = v$ and $\mathsf{concat}(\pi_2\ M) = v$. Because of the definition of step and $v \precsim_s u$, $\mathbb{D}$ contains all $(s, [s])$, if $s \in v$. Therefore, $M \subseteq \mathbb{D}$.
- $A\backslash w \in G$ and $w \overset{G}{\Rightarrow}_{\mathcal{F}} v$. We choose $M := [(A, v)]$. We have $\pi_1\ M = A$ and $\mathsf{concat}(\pi_2\ M) = v$. To prove $M \subseteq \mathbb{D}$, we apply the closure lemma of FCI and prove that $(A, v) \in$ items and step $\mathbb{D}\ (A, v)$. The first statement follows from our assumptions. For the second, we use the list $M' \subseteq \mathbb{D}$ we get by induction where $\pi_1\ M' = w$ and $\mathsf{concat}(\pi_2\ M') = v$ hold. By the definition of step, we get step $\mathbb{D}$ (A, v) .
- $s \overset{G}{\Rightarrow} w$ and $w' \overset{G}{\Rightarrow} v$. By induction, $M_1 \subseteq \mathbb{D}$ and $M_2 \subseteq \mathbb{D}$, where $\pi_1\ M_1 = s$, $\pi_1\ M_2 = w'$, $\mathsf{concat}(\pi_2\ M_1) = w$ and $\mathsf{concat}(\pi_2\ M_2) = v$. We choose $M := M_1 +\!\!+ M_2$ and all requirements are fulfilled. ∎

For the if-part, we assume (s, v) to be in $\mathbb{D}$. By construction of $\mathbb{D}$ with FCI and items, we know that v is a segment of u. Therefore, we only need to prove $s \overset{G}{\Rightarrow}_{\mathcal{F}} v$.

Lemma 5.4 *Let $(s, v) \in \mathbb{D}$. Then $s \overset{G}{\Rightarrow}_{\mathcal{F}} v$.*

Proof We apply the induction lemma of FCI. So assume $(s, v) \in$ items and a set M, where for every element (s', v'), $s' \overset{G}{\Rightarrow}_{\mathcal{F}} v'$ holds. Furthermore, we have step M (s, v). If $v = s$, then $s \overset{G}{\Rightarrow}_{\mathcal{F}} v$ follows by definition. Otherwise, we have $s = A$ and a set $M' \subseteq M$ where $A\backslash\pi_1\ M' \in G$ and $\pi_2\ M' = v$. With transitivity of derivations, we have to prove $(\pi_1\ M') \overset{G}{\Rightarrow}_{\mathcal{F}} v$. This can be done by induction on M' because of the assumption that $s' \overset{G}{\Rightarrow}_{\mathcal{F}} v'$ for every element (s', v') in M. ∎

Now we can conclude Lemma 5.1.

Proof (Lemma 5.1) Using Lemma 5.3 Lemma 5.4. ∎

To get rid of the restriction of s to $\mathcal{S}_G$, we observe the following fact.

Fact 5.5 *Assume $A \overset{G}{\Rightarrow} u$ and A is not in $\mathcal{S}_G$. Then $u = A$.*

By using FCI we showed that $\mathbb{D}$ is computable. From Lemma 5.1, we obtain our primary aim: The word problem for context-free languages is decidable.

Theorem 5.6 *$\mathcal{L}_G^A\ u$ is decidable.*

Proof If u is not a word, then $\neg\mathcal{L}_G^A\ u$. Otherwise, if $A \in \mathcal{S}_G$, then $A \overset{G}{\Rightarrow} u$ iff $(A, u) \in \mathbb{D}_{G,u}$ by Lemma 5.1. If $A \notin \mathcal{S}_G$, then $A \overset{G}{\Rightarrow} u$ iff $u = A$ by Fact 5.5. ∎

Remark

In this chapter, we presented a bottom-up chart parsing algorithm. One could describe it as a generalized version of a CYK parser because we do not expect the grammar to be in CNF. We examine all possible segments of the word we want to parse. In contrast, CYK can expect the grammar to be binary. Therefore, it is sufficient to just consider all possible segment pairs of the word.

Chapter 6

Elimination of Epsilon Rules

In the next chapters we describe several grammar transformations that apply to general context-free grammars. For each transformation we will have to prove two different lemmas:

1. After the transformation, the grammar fulfils the intended property.
2. The transformation preserves all languages of the grammar (except for the empty word ε).

In this chapter, we describe a grammar transformation that generates an **ε-free** grammar, i.e. a grammar without **ε-rule** (rules of the form $A\backslash\varepsilon$). We follow the algorithm described in [4, 1]. It consists of two steps. First, we generate the **ε-closure** of the grammar which makes every ε-rule redundant. This is achieved by adding all rules $A\backslash u$ we obtain from a rule $A\backslash u' \in G$ by deleting some variables that can derive ε from u'. In the second step, we delete every ε-rule. The generated grammar produces the same phrases except from ε. The following example illustrates the algorithm.

Example 6.2

Consider the following grammar:

$$G := A\backslash aBA \qquad B\backslash AA$$
$$A\backslash\varepsilon \qquad B\backslash b$$

Obviously, A can derive ε. With $A\backslash aBA$ and $B\backslash AA$ we add $A\backslash aB$, $B\backslash A$ and $B\backslash\varepsilon$ to the grammar. But then we know that B, too, can derive ε. Therefore, we can add $A\backslash aA$ and $A\backslash a$. We obtain the ε-closed grammar where $A\backslash\varepsilon$ and

$B\backslash\varepsilon$ are redundant.

$$\begin{aligned} G^{\varepsilon} := A\backslash aBA \quad & B\backslash AA \\ A\backslash aB \quad & B\backslash A \\ A\backslash aA \quad & B\backslash\varepsilon \\ A\backslash a \quad & B\backslash b \\ A\backslash\varepsilon \quad & \end{aligned}$$

Removing all ε-rules results in the ε-free grammar.

$$\begin{aligned} G^{\varepsilon^-} := A\backslash aBA \quad & B\backslash AA \\ A\backslash aB \quad & B\backslash A \\ A\backslash aA \quad & B\backslash b \\ A\backslash a \quad & \end{aligned}$$

The grammar is equivalent to the start grammar, except that neither A nor B can derive ε.

We have shown that the word problem of context-free languages is decidable. This means in particular that it is decidable whether a variable A can derive the empty word ε. We then call A **nullable**. We give an inductive characterization of nullable variables which is more convenient in proofs. It resembles the definition of productive variables in Chapter 4.

$$\frac{A\backslash u \in G \quad \forall s \in u.\, nullable_G\ s}{nullable_G\ A}$$

Fact 6.1 *$nullable_G\ A$ iff $A \overset{G}{\Rightarrow} \varepsilon$.*

To ease notation, we use $\mathcal{N}_G$ instead of $nullable_G$. In general, we do not indicate G if it is clear from the context. To formalize the elimination algorithm, we make use of sublists which turned out be very helpful in formal proofs. Note that the right-hand side of every rule in the ε-closure is a sublist with respect to $\mathcal{N}$ of a rule in the original grammar. For the following, we will assume the properties of sublists established in Fact 2.4. With sublists, we can state the property we want the ε-closure to fulfil.

Definition 6.2 *A grammar G is **ε-closed** if for every rule $A\backslash u$ in G the following holds: If $v \precsim_{\mathcal{N}} u$, then $A\backslash v$ is in G.*

The following function computes the closure of a grammar with respect to a predicate p:

$$\begin{array}{lll} \mathsf{closure} & : & (symbol \xrightarrow{dec} Prop) \to grammar \to grammar \\ \mathsf{closure}\ p\ [] & := & [] \\ \mathsf{closure}\ p\ (A \backslash u :: G) & := & \mathsf{map}\ (\lambda u'.\ A \backslash u')\ (\mathsf{slists}\ p\ u) +\!\!+\ \mathsf{closure}\ p\ G \end{array}$$

Fact 6.3 *$A\backslash v \in \mathsf{closure}\, p\, G$ iff there is a rule $A\backslash u \in G$ and $v \precsim_p u$.*

Instantiating closure results in the e-closure of a grammar:

$$G^\varepsilon := \mathsf{closure}\, \mathcal{N}\, G$$

Corollary 6.4 *$A\backslash v \in G^\varepsilon$ iff there is a rule $A\backslash u \in G$ and $v \precsim_{\mathcal{N}} u$.*

To prove that G^ε is ε-closed, we prove that G^ε has the same nullable variables as G.

Lemma 6.5 *$nullable_G\ A$ iff $nullable_{G^\varepsilon}\ A$.*

Proof By induction on $nullable$ for each direction using **??**. ∎

Lemma 6.6 *G^ε is ε-closed.*

Proof Let $A\backslash u \in G^\varepsilon$ and $v \precsim_{\mathcal{N}_{G^\varepsilon}} u$. Using Corollary 6.4, we know that there is a rule $A\backslash u'$ in G and $u \precsim_{\mathcal{N}_G} u'$. Using Lemma 6.5, transitively also $v \precsim_{\mathcal{N}_G} u'$. But then again with Corollary 6.4, $A\backslash v$ must be in G^ε. ∎

Next, we show that the ε-closure preserves the language of a grammar. More general, we prove $u \overset{G}{\Rightarrow}_{\mathcal{T}} v$ if and only if $u \overset{G^\varepsilon}{\Longrightarrow}_{\mathcal{T}} v$. We observe that if v is a sublist of u with respect to nullable variables, then we can derive v from u.

Lemma 6.7 *If $v \precsim_{\mathcal{N}} u$ then $u \overset{G}{\Rightarrow}_{\mathcal{T}} v$.*

Proof By induction on $v \precsim_{\mathcal{N}} u$. ∎

Lemma 6.8 *$A \overset{G}{\Rightarrow}_{\mathcal{T}} u$ iff $A \overset{G^\varepsilon}{\Longrightarrow}_{\mathcal{T}} u$.*

Proof By induction on $\Rightarrow_{\mathcal{T}}$ for both direction using Corollary 6.4 and Lemma 6.7. ∎

Corollary 6.9 *$\mathcal{L}_G^A \equiv \mathcal{L}_{G^\varepsilon}^A$.*

Proof By definition of $\mathcal{L}_G^A$ applying Lemma 6.8. ∎

We showed that the ε-closure of a grammar fulfils the desired ε-closedness property and that it preserves the language. In the next step we prove that removing all rules $A\backslash\varepsilon$ from G^ε preserves the language of every variable — except from ε which can not be derived anymore.

The following function computes the new grammar:

$$\bullet^{-} \; : \; grammar \rightarrow grammar$$
$$G^{-} \; := \; \mathsf{filter}\ (\lambda(A\backslash u).\ u \neq \varepsilon)\ G$$

The function G^- is obviously correct:

Fact 6.10 *G^- is ε-free.*

Fact 6.11 *$A\backslash u \in G^-$ iff $u \neq \varepsilon$ and $A\backslash u \in G$.*

We now aim to prove the following lemma:

Lemma 6.12 *Let G be ε-closed and $u \neq \varepsilon$. Then $A \overset{G}{\Rightarrow} u$ iff $A \overset{G^-}{\Longrightarrow} u$.*

The only-if-part is easy. Since G^- is a subset of G, we can just take the same derivation . For the if-part, we need to prove that we need no ε-rules. In [8], the proof is described as follows: Assume we do a derivation that uses an ε-rule $A\backslash\varepsilon$, i.e. we eliminate A. For a derivation in G^-, we have to decide that we want to eliminate A at the point where A is introduced. Instead of choosing the rule that contains A, in an ε-closed grammar, there is the similar rule without A.

Formally, we want to find another proof as formalizing the notion of introducing variables is unnecessarily complicated. Instead, we generalize the statement. We prove that if G can derive u, then G^- can derive every subphrase of u, where some nullable variables are deleted. This gives us a strong inductive hypothesis. We use the right-linear predicate $\Rightarrow_{\mathcal{L}}$ which allows us to argue about every rewriting step.

Lemma 6.13 *Assume G is ε-closed, $A \overset{G}{\Rightarrow}_{\mathcal{L}} u$, $u' \precsim_{\mathcal{N}} u$ and $u' \neq \varepsilon$. Then $A \overset{G^-}{\Longrightarrow}_{\mathcal{L}} u'$.*

Proof By induction on $A \overset{G}{\Rightarrow}_{\mathcal{L}} u$. We have two cases.

- $u = A$. As $u' \neq \varepsilon$, we have $u' = A$ and therefore $A \overset{G^-}{\Longrightarrow}_{\mathcal{L}} u'$.
- $A \overset{G}{\Rightarrow}_{\mathcal{L}} v_1 B v_3$ and $B\backslash v_2 \in G$. We have $u' \precsim_{\mathcal{N}} v_1 v_2 v_3$, so $u' = u_1 u_2 u_3$ and $u_1 \precsim_{\mathcal{N}} v_1$, $u_2 \precsim_{\mathcal{N}} v_2$ and $u_3 \precsim_{\mathcal{N}} v_3$. As G is ε-closed, we get $B\backslash u_2 \in G$. We distinguish between $u_2 = \varepsilon$ and $u_2 \neq \varepsilon$.

- $u_2 = \varepsilon$ and B is nullable. So $u_1 u_3 \precsim_{\mathcal{N}} v_1 B v_3$. By induction, we get directly $A \xRightarrow{G^-}_{\mathcal{L}} u_1 u_3$.
- $u_2 \neq \varepsilon$. As $u_1 B u_3 \precsim_s v_1 B v_3$, we have by induction $A \xRightarrow{G^-}_{\mathcal{L}} u_1 B u_3$. Therefore, $A \xRightarrow{G^-}_{\mathcal{L}} u_1 u_2 u_3$ by definition. ∎

As $\precsim$ is reflexive, with Lemma 6.13, G^- derives the same phrases as G except for ε. Therewith, we can prove Lemma 6.12.

Proof (Lemma 6.12) It is left to show that $A \xRightarrow{G^-} u$ implies $A \overset{G}{\Rightarrow} u$. Since $G^- \subseteq G$, the proof can be done with an easy induction on $A \xRightarrow{G^-} u$. ∎

Since G^ε is ε-closed, we apply the removal operation to it and receive a grammar that is ε-free.

Theorem 6.14 *$G^{\varepsilon-}$ is ε-free.*

Proof Follows from Fact 6.10. ∎

Theorem 6.15 $\mathcal{L}_G^A \backslash \{\varepsilon\} \equiv \mathcal{L}_{G^{\varepsilon-}}^A$

Proof Using Corollary 6.9 and Lemma 6.12. ∎

Discussion

Using the notion of sublists to prove ε-elimination correct is, as far as we know, a new approach. Thereby, we can avoid induction on the length of the derivation as proposed by Hopcroft and Ullman [5]. Previous formalizations like [4, 1, 9] seem to follow this idea. However, in our setting, sublists reduce the effort of formal proofs considerably.

Chapter 7

Elimination of Unit Rules

Unit rules are rules of the form $A\backslash B$. We remove those rules by replacing them by abbreviatory rules. Assume, $A\backslash B \in G$. For every rule $B\backslash u \in G$, we add $A\backslash u$. Then, we delete $A\backslash B$ from G, which is redundant now. Note that this technique preserves the domain and the range of a grammar. Therefore, we can again make use of finite closure iteration because all rules of the resulting grammar will be a combination of an old left-hand side and an old right-hand side.

The required grammar $G^{\mathcal{U}}$ can be best described by an inductive predicate.

$$\frac{A\backslash u \in G \quad \forall B.\, u \neq B}{A\backslash u \in G^{\mathcal{U}}} \qquad\qquad \frac{A\backslash B \in G \quad B\backslash u \in G^{\mathcal{U}}}{A\backslash u \in G^{\mathcal{U}}}$$

Every rule of G which is no unit rule is also a rule of $G^{\mathcal{U}}$. In addition, $G^{\mathcal{U}}$ contains all abbreviatory rules we get by simulating the use of a unit rule.

Similar to the algorithm that decides the word problem, we transfer this inductive characterization to FCI. To give the upper bound N, we make use of the generalized product function $*$ introduced in Chapter 2. Therewith, N can be defined in terms of a product using the rule constructor $\backslash$.

$$N_G := \mathcal{D}_G *_{\backslash} \mathcal{R}_G$$

The step predicate implements the inductive characterization of $G^{\mathcal{U}}$.

$$\begin{aligned} \mathsf{step}_G\ M\ (A\backslash u) := {} & A\backslash u \in G \ \wedge\ \neg\, \exists B.\, u = B \\ & \vee\ \exists B.\, A\backslash B \in G \ \wedge\ B\backslash u \in M \end{aligned}$$

Fact 7.1 step_G *is decidable.*

We now obtain a grammar without unit rules.

$$G^{\mathcal{U}} := \text{FCI } \mathsf{step}_G \; N_G$$

Remember that in order to prove $G^{\mathcal{U}}$ correct, we have to prove two statements. First, $G^{\mathcal{U}}$ does not contain any unit rule. In addition, it does not change the language $\mathcal{L}_G^A$ for any variable A. The first property follows directly from the definition of the step predicate and can be proved using the induction lemma of FCI (Lemma 2.3).

Theorem 7.2 *$G^{\mathcal{U}}$ does not contain any unit rule.*

Proof By applying the induction lemma of FCI. ∎

In order to prove the preservation of all languages we prove that every word derived by $G^{\mathcal{U}}$ can also be derived by G and vice versa. Note that this statement only holds for words, not for phrases in general: Right-hand sides of unit rules can be derived by G but not by $G^{\mathcal{U}}$. We make use of the tree-like derivation predicate $\Rightarrow_{\mathcal{F}}$, whose induction principle fits best to the definition of step. If we used $\Rightarrow_{\mathcal{L}}$ or $\Rightarrow_{\mathcal{T}}$, we could not apply the inductive hypothesis as the interim phrase uBw is not a word.

Soundness of $G^{\mathcal{U}}$ regarding derivability follows with a simple induction, we just have to prove that every rule in $G^{\mathcal{U}}$ can be simulated by G.

Lemma 7.3 *If $A \backslash u \in G^{\mathcal{U}}$, then $A \overset{G}{\Rightarrow}_{\mathcal{F}} u$.*

Proof Using the induction lemma of FCI. ∎

Lemma 7.4 *If $A \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} u$ then $A \overset{G}{\Rightarrow}_{\mathcal{F}} u$.*

Proof By induction on $A \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} u$ using Lemma 7.3 and the transitivity of $\Rightarrow_{\mathcal{F}}$. ∎

For completeness, we have to prove that the step function yields the expected grammar.

Lemma 7.5 *If $B \backslash u \in G^{\mathcal{U}}$ and $A \backslash B \in G$, then $A \backslash u \in G^{\mathcal{U}}$.*

Proof By applying the closure lemma of FCI. ∎

Lemma 7.6 *If $A \backslash u \in G$ and u is not a single variable, then $A \backslash u \in G^{\mathcal{U}}$.*

Proof By applying the closure lemma of FCI. ∎

Now we can prove what we aimed for.

Lemma 7.7 *If $A \overset{G}{\Rightarrow}_{\mathcal{F}} x$ then $A \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} x$.*

Proof By induction on $A \overset{G}{\Rightarrow}_{\mathcal{F}} x$. We distinguish three cases.

- $x = A$. By definition, $A \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} x$.
- $A\backslash u \in G$ and by induction, $u \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} x$. If $u \neq B$ for all B, then $A\backslash u \in G^{\mathcal{U}}$ by Lemma 7.6 and we are done. Otherwise, we have $u = B$ for some B. Because of $B \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} x$ and $x \neq B$ as x is a word, there must be a rule $B\backslash v \in G^{\mathcal{U}}$ and $v \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} x$ (this can be proved with a simple induction on $B \overset{G^{\mathcal{U}}}{\Longrightarrow}_{\mathcal{F}} x$). With Lemma 7.5, we have $A\backslash v \in G^{\mathcal{U}}$ and we are done.
- Follows directly from inductive hypotheses. ■

Having Lemma 7.4 and Lemma 7.7, we can conclude the preservation of languages.

Theorem 7.8 $\mathcal{L}_G^A \equiv \mathcal{L}_{G^{\mathcal{U}}}^A$

Discussion

We eliminated unit rules by replacing them with abbreviatory rules. This idea was inspired by Firsov and Uustalu [4]. However, they do not use an iterative context. An alternative approach is to insert all unit rules. This means that for $A\backslash B$ and $C\backslash uAv$, we add $C\backslash uBv$. We gave up on this idea since it does not preserve the range of a grammar. It would then have been much more difficult to use FCI, a technique that simplifies proofs significantly.

Chapter 8

Elimination of Deterministic Variables

Assume a grammar G which contains a rule $A\backslash u$. If this rule is the only rule with A on the left-hand side, we call A **deterministic**. Deterministic rules are, in a sense, superfluous, since one could replace every occurrence of A with u, an operation we call **inlining**. We now want to formalize on which conditions we can simplify a grammar by inlining one (or more) rules of the grammar. The following function substitutes a symbol in a grammar and can thus be used to inline a rule.

```
substG : grammar → symbol → phrase → grammar
substG G s u := map (λ(B\v). B\v[s → u]) G
```

We write $G[A \to u]$ for substG A u. For the following, we assume several properties of substG to hold. Some are properties of general substitution (Fact 2.9) lifted to substitution in grammars.

Fact 8.1

1. $(G_1 + G_2)[s \to u] = G_1[s \to u] + G_2[s \to u]$ (Split)
2. *If* $s \notin v$*, then* $(A\backslash v :: G)[s \to u] = A\backslash v :: G[s \to u]$. (Skip 1)
3. *If* $s \notin \mathcal{S}_G$*, then* $G[s \to u] = G$. (Skip 2)
4. *If* B *is fresh in* G*, then* $(G[s \to B])\ [B \to s] = G$. (Reversible substitution)
5. *If* $B\backslash v \in G$*, then* $u \overset{G}{\Rightarrow}_{\mathcal{T}} u[B \to v]$.
6. $\mathcal{D}_G = \mathcal{D}_{G[s \to u]}$

We aim to prove that eliminating a deterministic variable preserves the languages of the remaining variables of the grammar. Therefore, we consider the more general problem $A \overset{G}{\Rightarrow} u$ and state under which conditions we can prove that the inlined grammar preserves derivability.

Lemma 8.2 *We define* $G' := G[B \to v]$ *and want to derive a phrase* u *from* A*. Assume*

1. *$B \notin \mathcal{D}_G$ and*
2. *$B \notin v$ and*
3. *$B \notin u$ and*
4. *$A \neq B$.*

Then $A \xRightarrow{G'} u$ iff $A \xRightarrow{B\backslash v::G} u$.

A rule can be inlined if its left-hand side is unique in G and the rule is not recursive, i.e. the left-hand side does not appear on the right-hand side. In addition, the inlined grammar can not derive phrases that contain the inlined variable. Note that this is the case when we consider only words, which do not contain any variables. Moreover, the derivation must not start with the inlined variable.

We begin with the if-part of Lemma 8.2. We just have to prove that every rule in an inlined grammar can be simulated by a derivation.

Lemma 8.3 *If $A\backslash u \in G[B \to v]$, then $A \xRightarrow{B\backslash v::G} u$.*

Proof From $A\backslash u \in G[B \to v]$ we know that there is a rule $A\backslash u' \in G$ where $u = u'[B \to v]$. Moreover, we have $u' \xRightarrow{B\backslash v::G}_{\mathcal{T}} (u'[B \to v])$. The claim holds by transitivity of $\Rightarrow$. ∎

With this auxiliary lemma, we can prove the if-part. Here, the definition of $\Rightarrow$ allows us to directly apply Lemma 8.3 in the case of the third rule.

Lemma 8.4 *If $A \xRightarrow{G[B\to v]} u$, then $A \xRightarrow{B\backslash v::G} u$.*

Proof By induction on $A \xRightarrow{G[B\to v]} u$ using Lemma 8.3. ∎

For the only-if part we need to prove a corresponding lemma for substitution on phrases: With an inlined grammar we can derive inlined phrases. For this proof, the linear derivation predicate $\Rightarrow_{\mathcal{L}}$ is more convenient then $\Rightarrow$, as the numerous assumptions of the claim would complicate the two inductive hypotheses of the third rule of $\Rightarrow$.

Lemma 8.5 *Let $A \xRightarrow{B\backslash v::G}_{\mathcal{L}} u$ be given. We assume that $A \neq B$, $B \notin \mathcal{D}_G$ and $B \notin v$. Then $A \xRightarrow{G[B\to v]}_{\mathcal{L}} u[B \to v]$.*

Proof By induction on $A \xRightarrow{B\backslash v::G}_{\mathcal{L}} u$. We distinguish two cases.

- $u = A$. From $A \neq B$, we get $u[B \to v] = u$ and the claim holds by definition.

- $A \xrightarrow{B\backslash v::G}_{\mathcal{L}} uB'w$ and $B'\backslash v' \in B\backslash v :: G$. By induction we have $A \xrightarrow{G[B\to v]}_{\mathcal{L}} (uB'w)[B \to v]$. If $B' = B$, then $v = v'$ because $B \notin \mathcal{D}_G$. Then $(uB'w)[B \to v] = (uv'w)[B \to v]$ because $B \notin v$ and we are done. Otherwise, $B' \neq B$, and $B'\backslash v' \in G$. Therefore, $B'\backslash v'[B \to v] \in G[B \to v]$. So by definition of $\Rightarrow_{\mathcal{L}}$ and the inductive hypothesis we have $A \xrightarrow{G[B\to v]}_{\mathcal{L}} (uv'w)[B \to v]$ which is what we aimed for. ∎

This lemma directly yields the only-if-part.

Lemma 8.6 *Let $A \xrightarrow{B\backslash v::G}_{\mathcal{L}} u$ be given. We assume that $A \neq B$, $B \notin \mathcal{D}_G$, $B \notin v$ and additionally $B \notin u$. Then $A \xrightarrow{G[B\to v]}_{\mathcal{L}} u$.*

Proof From $B \notin u$ we have $u[B \to v] = u$ and we can apply Lemma 8.5. ∎

Together, we can prove what we aimed for, Lemma 8.2.

Proof (Lemma 8.2) By using Lemma 8.4 and Lemma 8.6. ∎

Theorem 8.7 *If $A \neq B$, $B \notin \mathcal{D}_G$ and $B \notin v$, then $\mathcal{L}^A_{G[B\to v]} \equiv \mathcal{L}^A_{B\backslash v::G}$.*

The technique of inlining rules of a grammar will turn out to be a useful tool for the grammar transformations that are yet to come. In the context of this thesis, we will only inline a single rule. However, we still discuss how to inline a list of deterministic rules in a grammar. The inlining function is defined as expected and is based on substG.

$$
\begin{aligned}
&\mathsf{inlL} \;:\; grammar \to list\;\; rule \to grammar \\
&\mathsf{inlL}\;\; G\;\; [\,] \qquad\qquad\quad := \; G \\
&\mathsf{inlL}\;\; G\;\; (A\backslash u\; ::\; M) := (\mathsf{inlL}\;\; G\;\; M)[A \to u]
\end{aligned}
$$

We lift the conditions stated in Lemma 8.2 to lists. They can be described with an inductive predicate that is inspired by a predicate defining unification in [10]. We use $\|$ to indicate that two lists are disjoint.

$$
\frac{}{\mathit{inlinable}_G\ \mathit{nil}} \qquad\qquad \frac{A \notin u \quad A \notin \mathcal{D}_M \quad u \parallel \mathcal{D}_M \quad \mathit{inlinable}_G\ M}{\mathit{inlinable}_G\ (A\backslash u :: M)}
$$

Of course, the empty list can be inlined into a grammar. Furthermore, assume that M can be inlined into G. Note, that inlL inlines a list of rules from right to left. Therefore, if we want to inline an additional rule on the left, its left-hand side should not appear in the rest of the list. Otherwise, we would inline a non-deterministic rule.

Furthermore, u must be disjoint from the left-hand sides of M because we drop rules we inline. With the same argument we state that the inlined rule should not be recursive.

Lemma 8.8 *We define* $G' := \mathsf{inlL}\ G\ M$. *Assume*

1. $inlinable_G\ M$ *and*
2. $u \parallel \mathcal{D}_M$ *and*
3. $A \in \mathcal{D}_G$.

Then $A \overset{G'}{\Longrightarrow} v$ *iff* $A \overset{G+\!\!+M}{\Longrightarrow} u$.

Both, the if-part and the only-if-part are similar to the case of inlining a single rule. For the if-part, we prove that every rule in the inlined grammar can be simulated by the old grammar in several steps.

Lemma 8.9 *If* $A \backslash u \in \mathsf{inlL}\ G\ M$, *then* $A \overset{G+\!\!+M}{\Longrightarrow} u$.

Proof By induction on M. ∎

With this lemma we conclude the if-part.

Lemma 8.10 *If* $A \overset{\mathsf{inlL}\ G\ M}{\Longrightarrow} u$, *then* $A \overset{G+\!\!+M}{\Longrightarrow} u$.

Proof By induction on $A \overset{\mathsf{inlL}\ G\ M}{\Longrightarrow} u$ using Lemma 8.9. ∎

For the only-if-part, we can profit from the only-if-part of Lemma 8.2 because $inlinable_G$ justs lifts the conditions of Lemma 8.2 to lists.

Lemma 8.11 *Let* $A \overset{G+\!\!+M}{\Longrightarrow} u$. *Assume* $inlinable_G\ M$, $u \parallel \mathcal{D}_M$ *and* $A \in \mathcal{D}_G$. *Then* $A \overset{\mathsf{inlL}\ G\ M}{\Longrightarrow} u$.

Proof By induction on M using Lemma 8.6. ∎

Now we can prove Lemma 8.8.

Proof (Lemma 8.8) Using Lemma 8.10 and Lemma 8.11. ∎

With Lemma 8.8, we obtain the desired property of inlining deterministic variables.

Theorem 8.12 *If* $inlinable_G\ M$ *and* $A \in \mathcal{D}_G$, *then* $\mathcal{L}^A_{\mathsf{inlL}\ G\ M} \equiv \mathcal{L}^A_{G+\!\!+M}$.

Remark

Note that we do not automatically eliminate all deterministic variables of a grammar. This would not be possible without changing the languages of the grammar (e.g. consider a deterministic rule $A\backslash x$). Instead, we explicitly state which rule (or list of rules) we inline into a given grammar. However, one could automatically eliminate all deterministic rules whose right-hand sides are not a word. This task might be interesting for future work.

Chapter 9

Separation of Grammars

In this chapter we describe how to generate a **uniform** grammar. A grammar is uniform, if every rule has a single character on its right-hand side or a list of variables. We achieve this by separating characters from the rest of the grammar: We introduce a new rule $A\backslash a$ for every character a that appears on the right-hand side of a rule; except from rules that are already of the form $B\backslash a$. Afterwards, a is replaced by A in the entire grammar, where A must be fresh.

We define uniformness formally:

$$uniform_G \coloneqq \forall A\backslash u \in G.\ \forall a.\ a \in u \to u = a.$$

We realize separation with finite fixed point iteration (FFPI). In every step, we separate one character from the grammar. Note that in this case, we can not apply finite closure iteration. We would have to give a list N such that every rule of the resulting grammar is in N. However, the resulting grammar contains rules with fresh variables. Therefore, giving N would require too much effort.

In order to apply FFPI, we need to provide

1. A step function $f : grammar \to grammar$ that separates one character if possible and does not change the grammar otherwise.
2. A size function $\sigma : grammar \to \mathbb{N}$ that decreases with every step $f\ G$, if G is not a fixed point of f.

Note that in contrast to FCI, f is a function and not a decidable predicate. We observe that if G is not uniform, then we can pick a character that can be separated from G. We prove this statement with the help of an informative type.

Lemma 9.1 $\{a \mid \exists A\backslash u \in G.\ a \in u \wedge |u| \geq 2\} + uniform_G$

Proof By induction on G. We use that if u is not character-free, then the type $\{a \mid a \in u\}$ is inhabited. ∎

With this lemma we construct a function getChar : $grammar \to option\ char$ which yields Some a, if there is a character a than can be separated and None otherwise. Below, we assume such a function getChar. Remember that for every grammar we can assume a generator for fresh variables (Fact 3.8). Therewith, we define the step function.

```
step : grammar → grammar
step G := match getChar G with
            | None ⇒ G
            | Some a ⇒ let A := fresh G
                        in A\a :: (G[a → A])
```

Furthermore, assume a function countChars : $phrase \to \mathbb{N}$ which counts the number of characters in a phrase. Its implementation is obvious and is omitted at this point. Therewith, we give a size function count that counts the number of characters in rules others than $A \backslash a$.

```
count : grammar → ℕ
count []          := 0
count (A\u :: G)  := if |u| < 2 then count G
                     else countChars u + count G
```

Now we define the uniform grammar with the help of FFPI.

$$G^{\mathcal{S}} := \text{FFPI step count}$$

Before proving that $G^{\mathcal{S}}$ preserves the languages of G, we prove that $G^{\mathcal{S}}$ is indeed uniform. Therefore, we show that count and step meet the requirements of FFPI such that it returns a fixed point. Afterwards, we prove that every fixed point of step must be a uniform grammar.

For the first point we need to prove that count decreases if we have not yet reached a fixed point (Lemma 2.2). We make two observations. First, if we replace a character with a variable, count will not increase. Second, if this character can be found in a rule that does not just map to a character, then count decreases.

Lemma 9.2 count $G \geq$ count $(G[a \to A])$

Proof By induction on G using that countChars $u \geq$ countChars $u[a \to A]$. ∎

Lemma 9.3 *Assume $A \backslash u \in G$, $a \in u$ and $u \neq a$. Then* count $G >$ count $(G[a \to B])$

Proof It suffices to prove count $[A \backslash u] >$ count $([A \backslash u][a \to B])$ as count will not decrease for the rest of the grammar (Lemma 9.2). But then we have countChars $u >$ countChars $u[a \to A]$ because $a \in u$. ∎

Now it is easy to prove what we aimed for.

Lemma 9.4 *If* $\mathsf{step}\ G \neq G$ *then* $\mathsf{count}\ G > \mathsf{count}(\mathsf{step}\ G)$*.*

Proof Using Lemma 9.3. ∎

With the fixed point lemma we get that $G^{\mathcal{S}}$ is a fixed point.

Lemma 9.5 $G^{\mathcal{S}}$ *is a fixed point of* step*.*

Proof We apply the fixed point lemma of FFPI Lemma 2.2. Therefore, we need to prove that for every n either $\mathsf{step}^n\ G$ is a fixed point or $\mathsf{count}\ G > \mathsf{count}\ (\mathsf{step}\ G)$. This follows with Lemma 9.4. ∎

It is left to prove is that a fixed point of step is uniform.

Lemma 9.6 *Let* G *be a fixed point of* step*. Then* G *is uniform.*

Proof A step $\mathsf{step}\ G$ yields either G or $B\backslash a :: G[a \to B]$ for some fresh variable B. In the first case, G is uniform by construction (Lemma 9.1). The second case is contradictory. We have $G = \mathsf{step} G = B\backslash a :: G[a \to B]$. But by construction, B is fresh in G. ∎

Theorem 9.7 $G^{\mathcal{S}}$ *is uniform.*

Proof $G^{\mathcal{S}}$ is a fixed point of step (Lemma 9.5) and every fixed point of step is uniform (Lemma 9.6). ∎

Next, we prove that $\mathcal{L}_G^A \equiv \mathcal{L}_{G^{\mathcal{S}}}^A$. As we might have added fresh variables to $G^{\mathcal{S}}$, we can only prove this statement for variables that are in the domain of G. Again, FFPI shortens our proofs. It suffices to show that the step function preserves the language because the induction lemma of FFPI lifts this proof to $G^{\mathcal{S}}$. We prove that we obtain the original grammar by inlining the rule that step might add. Since we proved that inlining preserves derivability (on certain conditions) in Chapter 8, we obtain the intended result. However, we need to assume the derived phase to be a word because otherwise, it could contain a freshly introduced variable.

Lemma 9.8 *Assume* $A \in \mathcal{D}_G$*. Then* $A \xRightarrow{\mathsf{step}\ G} x$ *iff* $A \xRightarrow{G} x$*.*

Proof We have $\mathsf{step}_G = G$ or $\mathsf{step}_G = B\backslash a :: G[a \to B]$ for some fresh variable B. The first case is easy. For the second, we prove both parts of the equivalence. In both proofs, we can substitute G with $(G[a \to B])[B \to a]$ as B is fresh. Now we can apply the correctness lemma of inlining (Lemma 8.2). ∎

The FFPI induction lemma yields the correctness of $G^{\mathcal{S}}$. Therefore, we need to prove that step G subsumes the domain of G.

Lemma 9.9 $\mathcal{D}_G \subseteq \mathcal{D}_{\mathsf{step}\ G}$

Proof By induction on G. ■

Lemma 9.10 *Assume $A \in \mathcal{D}_G$. Then $A \overset{G}{\Rightarrow} u$ iff $A \overset{G^{\mathcal{S}}}{\Longrightarrow} u$.*

Proof We generalize the claim to $A \overset{G}{\Rightarrow} u$ iff $A \xrightarrow{\mathsf{step}^n\ G} u$ and do induction on $n \in \mathbb{N}$ using the induction lemma of FFPI (Lemma 2.2), Lemma 9.9 and Lemma 9.8. ■

Theorem 9.11 *Assume $A \in \mathcal{D}_G$. Then $\mathcal{L}^A_{G^{\mathcal{S}}} \equiv \mathcal{L}^A_G$.*

Discussion

We assumed a function getChar : $grammar \to option\ char$ to exist. In Coq, we did not realize this with an option type. Instead, the step function is described with Coq tactics which work directly with the informative type in Lemma 9.1. The desired function is then given as a proof term that Coq constructs. These proof terms are meant to prove the existence of a functions with the desired properties, not to be executed in practice. However, in this case, this approach requires considerably less work than giving the function explicitly and then proving it correct. The proof of FCI in [10] follows the same approach.

In this chapter (and others), we profit from an abstraction for fixed point iteration. We only have to prove the correctness of a step function. However, there are algorithms that work more efficiently. As an example, one would expect the function getChar to search the whole grammar for a new character. This is done in every step, even though we could start the search at the point where the last character has been found.

Chapter 10

Binarization of Grammars

Context-free grammars are, in a sense, two-dimensional. Every grammar has arbitrarily many rules and the right-hand sides of the rules can be arbitrarily long. In a **binary** grammar, every right-hand side counts at most two symbols. This means that the grammar becomes one-dimensional and easier to handle in proofs. We show that every grammar can be transformed into a binary counterpart. As before, we make use of finite fixed point iteration and show that the transformation preserves the languages of the grammar.

If a grammar G is not binary, there is a rule $A\backslash su$ with three or more symbols on the right-hand side. We can shorten this rule by replacing $A\backslash su$ with two rules $A\backslash sB$ and $B\backslash u$ where B is a fresh variable. This is an operation inverse to inlining, similar to separation described in Chapter 9. Hence, we again use inlining to prove it correct. We implement the idea with the following step function.

```
step' : grammar → grammar → grammar
step' G' []                    := []
step' G' (A\[] :: G)           := A\[] :: step' G' G
step' G' (A\[s0] :: G)         := A\[s0] :: step' G' G
step' G' (A\[s0;s1] :: G)      := A\[s0;s1] :: step' G' G
step' G' (A\(s0::u) :: G)      := let B := fresh G'
                                  in A\[s0;B] :: B\u :: G

step : grammar → grammar
step G := step' G G
```

In contrast to prior step functions, we need to implement step with an auxiliary function step' which takes a second grammar as argument to remember which grammar to choose a new variable from. Apart from the step function, we pro-

vide a size function to apply FFPI. The following function adds the length of all rules that are not binary. This number decreases with every step.

```
count : grammar → ℕ
count []             := 0
count (A\u :: G)  := if |u| ≤ 2 then count G
                     else |u| + count G
```

We obtain the binary counterpart for each grammar with the help of FFPI.

$$G^2 := \text{FFPI step count}$$

Proving that G^2 is binary is straightforward. Similar to Chapter 9, we first prove that step decreases for grammars that are no fixed point of step. Together with the lemmas of FFPI we prove that G^2 is a fixed point of step. Afterwards, we show that every fixed point of step is binary. Therefrom, we then obtain that G^2 is binary.

Lemma 10.1 *If* step $G \neq G$*, then* count $G >$ count (step G)*.*

Proof By induction on G. The base case is contradictory. So assume $G = A\backslash u :: Gr$. If $A\backslash u$ is binary, then $Gr \neq \text{step}'\ G\ Gr$ and the statement follows with the inductive hypothesis. Otherwise, we have $u = s :: u'$ and we add a binary rule, which is not counted, and $|u'| < |u|$. ∎

Lemma 10.2 G^2 *is a fixed point of* step*.*

Proof We apply the fixed point lemma of FFPI (Lemma 2.2). Therefore, we need to prove that for every n either $\text{step}^n\ G$ is a fixed point or count $G >$ count (step G). This follows with Lemma 10.1. ∎

Lemma 10.3 *Let G be a fixed point of* step*. Then G is binary.*

Proof By induction on G. ∎

Theorem 10.4 G^2 *is binary.*

Proof G^2 is a fixed point of step (Lemma 10.2) and every fixed point of step is binary (Lemma 10.3). ∎

To prove that G^2 preserves the language of G we proceed similar to Chapter 9. It suffices to show that step preserves the language of G. Again, we can prove our goal only for variables from $\mathcal{D}_G$.

Lemma 10.5 *Assume, $A \in \mathcal{D}_G$. Then $A \overset{G}{\Rightarrow} x$ if and only if $A \xRightarrow{\text{step } G} x$.*

Proof We distinguish two cases. If $\mathsf{step}\ G = G$, then the claim follows directly. Otherwise, one rule of G has been shortened, i.e. $G \equiv A\backslash su :: Gr$, $\mathsf{step}_G \equiv A\backslash sB :: B\backslash u :: Gr$ and B is a fresh variable. Therefore, $G \equiv (A\backslash sB :: Gr)[B \to u]$. Since B is fresh and x is a word, the goal follows with the correctness lemma of inlining (Lemma 8.2). ∎

To lift this lemma to G^2, we need to prove that $\mathsf{step}\ G$ subsumes the domain of G.

Lemma 10.6 $\mathcal{D}_G \subseteq \mathcal{D}_{\mathsf{step}\ G}$

Proof By induction on G. ∎

With this lemma it follows the correctness of G^2.

Corollary 10.7 *Assume, $A \in \mathcal{D}_G$. Then $A \stackrel{G}{\Rightarrow} x$ if and only if $A \stackrel{G^2}{\Longrightarrow} x$.*

Proof We generalize to $A \stackrel{G}{\Rightarrow} u$ iff $A \xRightarrow{\mathsf{step}^n\ G} u$ and do induction on $n \in \mathbb{N}$ using the induction lemma of FFPI (Lemma 2.2), Lemma 10.5 and Lemma 10.6. ∎

Theorem 10.8 *Assume, $A \in \mathcal{D}_G$. Then $\mathcal{L}_G^A \equiv \mathcal{L}_{G^2}^A$.*

Discussion

The inlining tool to eliminate deterministic variables turned out to be useful in the last two chapters. It implements an inverse operation to both, a step of separation and a step of binarization. Therewith, proving the correctness of both transformations was easy. This kind of generalization has, as far as we know, not yet been discussed in formalizations of context-free grammars.

Chapter 11

Chomsky Normal Form

In the last chapters, we discussed all grammar transformations that are required for the transformation of a grammar to Chomsky normal form.

Definition 11.1 *A grammar G is in* ***Chomsky normal form*** *(CNF) if*

1. *G is ε-free and*
2. *G is binary and*
3. *G is uniform and*
4. *G does not contain any unit rules.*

Classically, G is not assumed to be completely ε-free as this would forbid ε to be in its language. Therefore, a rule $S\backslash\varepsilon$ is allowed, if S is the start symbol. As we do not care about start symbols, we assume ε not to be in any language of G. Otherwise — we proved that this is a decidable problem — it is easy to add rules $S\backslash A$ and $S\backslash\varepsilon$ after normalization that fixes S (or rather A, if we want to derive something else then ε) to be the start symbol.

11.1 Transformation to Chomsky Normal Form

For a grammar G, we obtain a grammar $G^{\mathcal{N}}$ in CNF by first applying binarization, followed by separation, elimination of ε-rules and elimination of unit rules. To prove that $G^{\mathcal{N}}$ is indeed in Chomsky normal form, we need to prove that non of our transformations destroys the progress we already achieved. Note that the order in which we apply the single transformations does matter. For example, the elimination of ε-rules could add new unit rules. Hence, we should eliminate unit rules after eliminating ε-rules. Since all described transformations work on general grammars, we can, for example, assume G^{ε^-} to be ε-free , irrespective of which transformations we applied before.

To obtain a grammar in CNF, we first apply binarization. Therefore, we prove that the other transformations do not create non-binary rules when applied to a binary grammar.

Lemma 11.2 *Separation and the elimination of ε-rules and unit rules preserve binarity.*

1. *If G is binary, then also $G^{\mathcal{S}}$ is.*
2. *If G is binary, then also G^{ε^-} is.*
3. *If G is binary, then also $G^{\mathcal{U}}$ is.*

Proof

1. We apply the induction lemma of FFPI and need to show that the separating step function (we call it $\mathsf{step}_{\mathcal{S}}$) preserves binarity for some G'. Assume $\mathsf{step}_{\mathcal{S}}\ G' = G'$, then the claim follows by assumption. Otherwise $\mathsf{step}_{\mathcal{S}}\ G' = B\backslash a :: G'[a \to B]$. As G' is binary, also $G'[a \to B]$ is. $B\backslash a$ is obviously binary.
2. Every right-hand side of a rule in G^{ε^-} is a sublist of a right-hand side in G. Since sublists to not increase the length of a list, G^{ε^-} is binary.
3. Follows directly as $\mathcal{R}_G = \mathcal{R}_{G^{\mathcal{U}}}$. ∎

Corollary 11.3 *$G^{\mathcal{N}}$ is binary.*

Proof Follows from Lemma 11.2 together with Theorem 10.4. ∎

Next, we prove that $G^{\mathcal{N}}$ is uniform.

Lemma 11.4 *The elimination of ε-rules and unit rules preserve uniformness of a grammar.*

1. *If G is uniform, then also G^{ε^-} is.*
2. *If G is uniform, then also $G^{\mathcal{U}}$ is.*

Proof

1. Assume $A\backslash u \in G^{\varepsilon^-}$ and $a \in u$. It follows that there is a u' such that $A\backslash u' \in G$ and $u \precsim_{\mathcal{N}} u'$. Therefore, $a \in u'$. As G is uniform, it follows $u' = a$. As u is a sublist of u' and G^{ε^-} is ε-free, $u = a$.
2. Follows directly as $\mathcal{R}_G = \mathcal{R}_{G^{\mathcal{U}}}$. ∎

Corollary 11.5 *$G^{\mathcal{N}}$ is uniform.*

Proof Follows from Lemma 11.4 together with Theorem 9.7. ∎

Finally. we need to prove that $G^{\mathcal{N}}$ is ε-free.

Lemma 11.6 *If G is ε-free, then also $G^{\mathcal{U}}$ is.*

Proof Follows directly as $\mathcal{R}_G = \mathcal{R}_{G^{\mathcal{U}}}$. ∎

Corollary 11.7 *$G^{\mathcal{N}}$ is ε-free.*

Proof Follows from Lemma 11.6 together with Theorem 6.14. ∎

Theorem 11.8 *$G^{\mathcal{N}}$ is in Chomsky normal form.*

Proof Follows from Corollary 11.3, Corollary 11.5 and Corollary 11.7 together with Theorem 7.2. ∎

Now that we know that $G^{\mathcal{N}}$ is in Chomsky normal form, it remains to prove that $G^{\mathcal{N}}$ has the same languages. Because of how we defined ε-elimination, we exclude ε from that statement. Moreover, as we added fresh symbols to the grammar during transformation, we only consider languages of variables that are in the domain of the original grammar. This is why we show that binarization subsumes the domain of a grammar in order to apply the correctness lemma of separation.

Lemma 11.9 $\mathcal{D}_G \subseteq \mathcal{D}_{G^2}$

Proof With the induction lemma of FFPI using that the step function used for binarization subsumes the domain (Lemma 10.6). ∎

Finally, we can prove that normalization preserves the languages of the grammar.

Theorem 11.10 *Assume $A \in \mathcal{D}_G$ and $u \neq \varepsilon$. Then $\mathcal{L}_G^A \equiv \mathcal{L}_{G^{\mathcal{N}}}^A$.*

Proof Follows from the correctness theorems of each chapter (Theorem 6.15, Theorem 7.8, Theorem 9.11, Theorem 10.8) together with Lemma 11.9. ∎

Chapter 12

Conclusion

In this thesis, we gave a formalization of context-free grammars based on lists. We described algorithms to decide the emptiness and the word problem. In addition, we discussed four grammar transformations (elimination of ε-rules, elimination of unit rules, separation and binarization) necessary to give the Chomsky normal form of a grammar. Both decidability results and the transformations can be applied to general CFGs, we do not assume it to be binary or in CNF.

It turned out that the decision functions and all transformations can be described as bounded iterations on grammars. Hence, we were able to apply abstractions for finite fixed point and closure iterations. Thereby, we obtained intuitive algorithms and simple correctness proofs.

12.1 Future Work

The theory of context-free grammars and languages is broad. This means that there are many results that we did not discuss in this thesis. First of all, it would be interesting to prove that the finiteness problem of context-free languages is decidable. Intuitively, a language should be finite if productive variables do not appear in circles. We already described how to detect productive variables. We assume recognizing circles in a grammar to be a feasible task. As far as we know, this problem has not yet been formalized in a proof assistant.

Furthermore, one could consider to formalize the elimination of useless symbols which is not included in the transformation to CNF. In [9], Ramos describes an approach which is formalized in Coq.

In addition, we did not discuss closure properties of context-free languages, some of them involving regular languages. In his PhD thesis [9], Ramos proves that context-free languages are closed under union, concatenation and Kleene star. The work of Barthwal [1] additionally includes that context-free languages are closed

under the image of a homomorphism and substitution. In this thesis, we already alluded to substitution (Chapter 8) which could be extended.

Appendix A

Coq Realization

This work is carried out in the proof assistant Coq [11], compiled with version 8.5pl2 (July 2016). All results can be found at `https://www.ps.uni-saarland.de/~hofmann/bachelor.php`. The development counts about 2000 lines (the `Base` library not included). The organization of the files mainly follows the chapters of this thesis.

Definitions and Preliminaries

We use the `Base.v` library that was developed for [10] as basis. Here, you can find the definition of finite closure and finite fixed point iteration. The file `Lists.v` adds some definitions and lemmas about lists. This includes everything which is discussed in Chapter 2, i.e. sublists, segments, substitution and other functions.

Context-Free Grammars

Context-free grammars are defined in `Definitions.v`. The definitions of the domain, range, etc. can be found in `Symbols.v`. This also includes several lemmas about fresh variables. All definitions and equivalences concerning derivability are proved in `Derivation.v`. The content of these three files is described in Chapter 3.

Decidability Results

In `Dec_Empty.v`, you can find the results of Chapter 4, i.e. the decidability of the emptiness problem. The decidability of the word problem is formalized in `Dec_Word.v`.

Transformations and Chomsky Normal Form

The elimination of ε-rules is given in `ElimE.v`, the elimination of unit rules in `ElimU.v`. In `Inlining.v`, we discuss substitution in grammars and the elimination of deterministic variables. The results are used in `Separate.v` and `Binarize.v`, where the separation of characters and the binarization of grammars can be found. Finally, in `Chomsky.v`, we prove a transformation to Chomsky normal form correct.

Appendix B

Use of Derivation Predicates

In this thesis, we make us of different characterizations of derivations. In the following table, you can find for each chapter that involves discussions about derivability which predicate we apply for the main proofs.

Chapter 4:	Decidability of Emptiness Problem	$\Rightarrow_{\mathcal{F}}$
Chapter 5:	Decidability of Word Problem	$\Rightarrow_{\mathcal{F}}$
Chapter 6:	Elimination of Epsilon Rules	$\Rightarrow_{\mathcal{T}}, \Rightarrow_{\mathcal{L}}$
Chapter 7:	Elimination of Unit Rules	$\Rightarrow_{\mathcal{F}}$
Chapter 8:	Elimination of Deterministic Variables	$\Rightarrow, \Rightarrow_{\mathcal{L}}$
Chapter 9:	Separation of Grammars	$\Rightarrow$
Chapter 10:	Binarization of Grammars	$\Rightarrow$

Appendix C

Variable and Function Names

C.1 Variable Names

grammars	G
phrases	u, v, w
words (terminal phrases)	x, y, z
rules	r
symbols	s
characters	a, b, c, $\ldots$
variables	A, B, C, $\ldots$

C.2 Transformations and Function Names

ε-free grammar	$G^{\varepsilon-}$
grammar without unit rules	$G^{\mathcal{U}}$
uniform grammar	$G^{\mathcal{S}}$
binary grammar	G^{2}
grammar in CNF	$G^{\mathcal{N}}$
domain	$\mathcal{D}_G$
range	$\mathcal{R}_G$
symbols	$\mathcal{S}_G$

Bibliography

[1] Aditi Barthwal. *A formalisation of the theory of context-free languages in higher order logic*. PhD thesis, The Australian National University, 2010.

[2] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[3] Denis Firsov and Tarmo Uustalu. Certified CYK parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming*, 83(5):459–468, 2014.

[4] Denis Firsov and Tarmo Uustalu. Certified normalization of context-free grammars. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 167–174. ACM, 2015.

[5] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Ma., USA, 1979.

[6] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR (1) parsers. In *European Symposium on Programming*, pages 397–416. Springer, 2012.

[7] Adam Koprowski and Henri Binsztok. Trx: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

[8] Dexter C. Kozen. *Automata and computability*. Springer, 1997.

[9] Marcus V. M. Ramos. *Formalization of context-free language theory*. PhD thesis, Universidade Federal de Pernambuco, 2016.

[10] Gert Smolka and Chad E. Brown. Introduction to computational logic. Lecture Notes [PDF], 2014. Retrieved from `https://www.ps.uni-saarland.de/courses/cl-ss14/script/icl.pdf`; 10th August, 2016.

[11] The Coq Development Team. The coq reference manual, version 8.5, July 2016. Retrieved from `http://coq.inria.fr/doc`; 10th August, 2016.