View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Universidade do Minho: RepositoriUM

On the Formalization of Some Results of Context-Free Language Theory

Marcus Vinícius Midena Ramos[1,2],
Ruy J. G. B. de Queiroz[2],
Nelma Moreira[3], and
José Carlos Bacelar Almeida[4,5]

[1] Universidade Federal do Vale do São Francisco, Juazeiro, BA, Brazil
marcus.ramos@univasf.edu.br
[2] Universidade Federal de Pernambuco, Recife, PE, Brazil
{mvmr,ruy}@cin.ufpe.br
[3] Universidade do Porto, Porto, Portugal
nam@dcc.fc.up.pt
[4] Universidade do Minho, Braga, Portugal
jba@di.uminho.pt
[5] HASLab - INESC TEC

Abstract. This work describes a formalization effort, using the Coq proof assistant, of fundamental results related to the classical theory of context-free grammars and languages. These include closure properties (union, concatenation and Kleene star), grammar simplification (elimination of useless symbols, inaccessible symbols, empty rules and unit rules), the existence of a Chomsky Normal Form for context-free grammars and the Pumping Lemma for context-free languages. The result is an important set of libraries covering the main results of context-free language theory, with more than 500 lemmas and theorems fully proved and checked. This is probably the most comprehensive formalization of the classical context-free language theory in the Coq proof assistant done to the present date, and includes the important result that is the formalization of the Pumping Lemma for context-free languages.

Keywords: Context-free language theory, language closure, grammar simplification, Chomsky Normal Form, Pumping Lemma, formalization, Coq.

1 Introduction

This work is about the mathematical formalization of an important subset of the context-free language theory, including some of its most important results such as the Chomsky Normal Form and the Pumping Lemma.

The formalization has been done in the Coq proof assistant. This represents a novel approach towards formal language theory, specially context-free language theory, as virtually all textbooks, general literature and classes on the subject rely on an informal (traditional) mathematical approach. The objective of this work, thus, is to elevate the status of this theory to new levels in accordance with the state-of-the-art in mathematical accuracy, which is accomplished with the use of interactive proof assistants.

The choice of using Coq comes from its maturity, its widespread use and the possibility of extracting certified code from proofs. HOL4 and Agda have also been used in the formalization of context-free language theory (see Section 7), however they do not comply to at least one of these criteria.

The formalization is discussed in Sections 2 (method used in most of the formalization), 3 (closure properties), 4 (grammar simplification), 5 (CNF - Chomsky Normal Form) and 6 (Pumping Lemma). The main definitions used in the formalization are presented in Appendix A. The library on binary trees and their relation to CNF derivations is briefly discussed in Appendix B.

Formal language and automata theory formalization is not a completely new area of research. In Section 7, a summary of these accomplishments is presented. Most of the formalization effort on general formal language theory up to date has been dedicated to the regular language theory, and not so much to context-free language theory. Thus, this constitutes the motivation for the present work. Final conclusions are presented in Section 8.

In order to follow this paper, the reader is required to have basic knowledge of Coq and context-free language theory. The recommended starting point for Coq is the book by Bertot and Castéran [1]. Background on context-free language theory can be found in [2] or [3], among others. A more detailed and complete discussion of the results of this work can be found in [4]. The source files of the formalization are available for download from [5].

2 Method

Except for the Pumping Lemma, the present formalization is essentially about context-free grammar manipulation, that is, about the definition of a new grammar from a previous one (or two), such that it satisfies some very specific properties. This is exactly the case when we define new grammars that generate the union, concatenation, closure (Kleene star) of given input grammar(s). Also, when we create new grammars that exclude empty rules, unit rules, useless symbols and inaccessible symbols from the original ones. Finally, it is also the case when we construct a new grammar that preserves the language of the original grammar and still observes the Chomsky Normal Form.

In the general case, the mapping of grammar $g_1 = (V_1, \Sigma, P_1, S_1)$ into grammar $g_2 = (V_2, \Sigma, P_2, S_2)$ requires the definition of a new set of non-terminal symbols N_2, a new set of rules P_2 and a new start symbol S_2. Similarly, the mapping of grammar $g_1 = (V_1, \Sigma, P_1, S_1)$ and grammar $g_2 = (V_2, \Sigma, P_2, S_2)$ into grammar $g_3 = (V_3, \Sigma, P_3, S_3)$ requires the definition of a new set of non-terminal symbols N_3, a new set of rules P_3 and a new start symbol S_3.

For all cases of grammar manipulation, we consider that the original and final sets of terminal symbols are the same. Also, we have devised the following common approach to constructing the desired grammars:

1. Depending on the case, inductively define the type of the new non-terminal symbols; this will be important, for example, when we want to guarantee that the start symbol of the grammar does not appear in the right-hand side of any rule or when we have to construct new non-terminals from the existing ones; the new type may use some (or all) symbols of the previous type (via mapping), and also add new symbols;
2. Inductively define the rules of the new grammar, in a way that it allows the construction of the proofs that the resulting grammar has the required properties; these new rules will likely make use of the new non-terminal symbols described above; the new definition may exclude some of the original rules, keep others (via mapping) and still add new ones;
3. Define the new grammar by using the new type of non-terminal symbols and the new rules; define the new start symbol (which might be a new symbol or an existing one) and build a proof of the finiteness of the set of rules for this new grammar;
4. State and prove all the lemmas and theorems that will assert that the newly defined grammar has the desired properties;
5. Consolidate the results within the same scope and finally with the previously obtained results.

In the following sections, this approach will be explored with further detail for each main result. The definitions of Appendix A are used throughout.

3 Closure Properties

The basic operations of union, concatenation and closure for context-free grammars are described in a rather straightforward way. These operations provide new context-free grammars that generate, respectively, the union, concatenation and the Kleene star closure of the language(s) generated by the input grammar(s).[6]

For the union, given two arbitrary context-free grammars g_1 and g_2, we want to construct g_3 such that $L(g_3) = L(g_1) \cup L(g_2)$ (that is, the language generated by g_3 is the union of the languages generated by g_1 and g_2).

The classical informal proof constructs $g_3 = (V_3, \Sigma, P_3, S_3)$ from g_1 and g_2 such that $N_3 = N_1 \cup N_2 \cup \{S_3\}$ and $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1, S_3 \rightarrow S_2\}$. With the appropriate definitions for the new set of non-terminal symbols, the new set of rules and the new start symbol, we are able to construct a new grammar `g_uni` such that `g3 = g_uni g1 g2`.

For the concatenation, given two arbitrary context-free grammars g_1 and g_2, we want to construct g_3 such that $L(g_3) = L(g_1) \cdot L(g_2)$ (that is, the language generated by g_3 is the concatenation of the languages generated by g_1 and g_2).

The classical informal proof constructs $g_3 = (V_3, \Sigma, P_3, S_3)$ from g_1 and g_2 such that $N_3 = N_1 \cup N_2 \cup \{S_3\}$ and $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 S_2\}$. With the appropriate definitions for the new set of non-terminal symbols, the new set of

[6] The results of this section are available in libraries `union.v`, `concatenation.v` and `closure.v`.

rules and the new start symbol, we are able to construct a new grammar `g_cat` such that `g3 = g_cat g1 g2`.

For the Kleene star, given an arbitrary context-free grammar g_1, we want to construct g_2 such that $L(g_2) = (L(g_1))^*$ (that is, the language generated by g_2 is the reflexive and transitive concatenation (Kleene star) of the language generated by g_1).

The classical informal proof constructs $g_2 = (V_2, \Sigma, P_2, S_2)$ from g_1 such that $N_2 = N_1 \cup N_2 \cup \{S_2\}$ and $P_2 = P_1 \cup P_2 \cup \{S_2 \to S_2S_1, S_2 \to S_1\}$. With the appropriate definitions for the new set of non-terminal symbols, the new set of rules and the new start symbol, we are able to construct a new grammar `g_uni` such that `g2 = g_clo g1`.

Although simple in their structure, it must be proved that the definitions `g_uni`, `g_cat` and `g_clo` always produce the correct result. In other words, these definitions must be "certified", which is one of the main goals of formalization. In order to accomplish this, we must first state the theorems that capture the expected semantics of these definitions. Finally, we have to derive proofs of the correctness of these theorems.

This can be done with a pair of theorems for each grammar definition: the first relates the output to the inputs, and the other one does the converse, providing assumptions about the inputs once an output is generated. This is necessary in order to guarantee that the definitions do only what one would expect, and no more.

For union, we prove (considering that g_3 is the union of g_1 and g_2 and S_3, S_1 and S_2 are, respectively, the start symbols of g_3, g_1 and g_2): $\forall g_1,\, g_2,\, s_1,\, s_2, (S_1 \Rightarrow^*_{g_1} s_1 \to S_3 \Rightarrow^*_{g_3} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2 \to S_3 \Rightarrow^*_{g_3} s_2)$. For the converse of union we prove: $\forall s_3, (S_3 \Rightarrow^*_{g_3} s_3) \to (S_1 \Rightarrow^*_{g_1} s_3) \vee (S_2 \Rightarrow^*_{g_2} s_3)$. Together, the two theorems represent the semantics of the context-free grammar union operation.

For concatenation, we prove (considering that g_3 is the concatenation of g_1 and g_2 and S_3, S_1 and S_2 are, respectively, the start symbols of g_3, g_1 and g_2): $\forall g_1,\, g_2,\, s_1,\, s_2, (S_1 \Rightarrow^*_{g_1} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2) \to (S_3 \Rightarrow^*_{g_3} s_1 \cdot s_2)$. For the converse of concatenation, we prove: $\forall g_3,\, s_3, (S_3 \Rightarrow^*_{g_3} s_3) \to \exists s_1,\, s_2, (S_1 \Rightarrow^*_{g_1} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2) \wedge (s_3 = s_1 \cdot s_2)$.

For closure, we prove (considering that g_2 is the Kleene star of g_1 and S_2 and S_1 are, respectively, the start symbols of g_2 and g_1): $\forall g_1,\, s_1,\, s_2, (S_2 \Rightarrow^*_{g_2} \epsilon) \wedge ((S_2 \Rightarrow^*_{g_2} s_2) \wedge (S_1 \Rightarrow^*_{g_1} s_1) \to S_2 \Rightarrow^*_{g_2} s_2 \cdot s_1)$. Finally: $\forall s_2, (S_2 \Rightarrow^*_{g_2} s_2) \to (s_2 = \epsilon) \vee (\exists\ s_1,\, s_2' \,|\, (s_2 = s_2' \cdot s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2') \wedge (S_1 \Rightarrow^*_{g_1} s_1))$.

In all three cases, the correctness proofs are straightforward and follow closely the informal proofs available in most textbooks. The formalization consists of a set of short and readable lemmas, except for the details related to mappings involving sentential forms. Since every grammar is defined with a different set of non-terminal symbols (i.e. uses a different type for these symbols), sentential forms from one grammar have to "mapped" to sentential forms of another grammar in order to be usable and not break the typing rules of Coq. This required a lot of effort in order to provide and use the correct mapping functions, and also to cope with it during proof construction. This is something that we don't see in informal proofs, and is definitely a burden when doing the formalization.

The completeness proofs, on the other hand, resulted in single lemmas with reasonably long scripts (~280 lines) in each case. Intermediate lemmas were not easily identifiable as in the correctness cases and, besides the initial induction of predicate `derives`, the long list of various types of case analysis increased the complexity of the scripts, which are thus more difficult to read.

It should be added that the closure operations considered here can be explained in a very intuitive way (either with grammars or automata), and for this reason many textbooks don't even bother going into the details with mathematical reasoning. Because of this, our formalization was a nice exercise in revealing how simple and intuitive proofs can grow in complexity with many details not considered before.

4 Simplification

The definition of a context-free grammar, and also the operations specified in the previous section, allow for the inclusion of symbols and rules that may not contribute to the language being generated. Besides that, context-free grammars may also contain rules that can be substituted by equivalent smaller and simpler ones. Unit rules, for example, do not expand sentential forms (instead, they just rename the symbols in them) and empty rules can cause them to contract. Although the appropriate use of these features can be important for human communication in some situations, this is not the general case, since it leads to grammars that have more symbols and rules than necessary, making difficult its comprehension and manipulation. Thus, simplification is an important operation on context-free grammars.

Let G be a context-free grammar, $L(G)$ the language generated by this grammar and ϵ the empty string. Different authors use different terminology when presenting simplification results for context-free grammars. In what follows, we adopt the terminology and definitions of [2].

Context-free grammar simplification comprises the manipulation of rules and symbols, as described below:

1. An *empty rule* $r \in P$ is a rule whose right-hand side β is empty (e.g. $X \to \epsilon$). We prove that for all G there exists G' such that $L(G) = L(G')$ and G' has no empty rules, except for a single rule $S \to \epsilon$ if $\epsilon \in L(G)$; in this case, S (the initial symbol of G') does not appear on the right-hand side of any rule of G';
2. A *unit rule* $r \in P$ is a rule whose right-hand side β contains a single non-terminal symbol (e.g. $X \to Y$). We prove that for all G there exists G' such that $L(G) = L(G')$ and G' has no unit rules;
3. A symbol $s \in V$ is *useful* ([2], p. 116) if it is possible to derive a string of terminal symbols from it using the rules of the grammar. Otherwise, s is called an *useless symbol*. A useful symbol s is one such that $s \Rightarrow^* \omega$,

with $\omega \in \Sigma^*$. Naturally, this definition concerns mainly non-terminals, as terminals are trivially useful. We prove that for all G such that $L(G) \neq \emptyset$, there exists G' such that $L(G) = L(G')$ and G' has no useless symbols;

4. A symbol $s \in V$ is *accessible* ([2], p. 119) if it is part of at least one string generated from the root symbol of the grammar. Otherwise, it is called an *inaccessible symbol*. An accessible symbol s is one such that $S \Rightarrow^* \alpha s \beta$, with $\alpha, \beta \in V^*$. We prove that for all G there exists G' such that $L(G) = L(G')$ and G' has no inaccessible symbols.

Finally, we prove a unification result: that for all G, if G is non-empty, then there exists G' such that $L(G) = L(G')$ and G' has no empty rules (except for one, if G generates the empty string), no unit rules, no useless symbols, no inaccessible symbols and the start symbol of G' does not appear on the right-hand side of any other rule of G'.[7]

In all these four cases and the five grammars that are discussed next (namely `g_emp`, `g_emp'`, `g_unit`, `g_use` and `g_acc`), the proof of `rules_finite` is based on the proof of the corresponding predicate for the argument grammar. Thus, all new grammars satisfy the `cfg` specification and are finite as well.

Result (1) is achieved in two steps. In the first step, we map grammar g_1 into an equivalent grammar g_2 (except for the empty string), which is free of empty rules and whose start symbol does not appear on the right-hand side of any rule. This is done by eliminating empty rules and substituting rules that have nullable symbols in the right-hand side by a set of equivalent rules. Next, we use g_2 to map g_1 into g_3 which is fully equivalent to g_1 (including the empty string if this is the case).

Observe that resulting grammar (`g_emp g` or g_2) does not generate the empty string, even if `g` (or g_1) does so. The second step, thus, consists of constructing g_3 such that it generates all the strings of g_2 plus the empty string if g_1 does so. This is done by conditionally adding a rule that maps the start symbol to the empty string.

We define `g_emp' g` (or g_3) such that `g_emp' g` generates the empty string if `g` generates the empty string. This is done by stating that every rule from `g_emp g` is also a rule of `g_emp' g` and also by adding a new rule that allow `g_emp' g` to generate the empty string directly if necessary.

The proof of the correctness of the previous definitions is achieved through the following Coq theorem:

```
Theorem g_emp'_correct: ∀ g: cfg non_terminal terminal,
g_equiv (g_emp' g) g ∧ (produces_empty g → has_one_empty_rule (g_emp' g)) ∧
(∼ produces_empty g → has_no_empty_rules (g_emp' g)) ∧
start_symbol_not_in_rhs (g_emp' g).
```

New predicates are used in this statement: `produces_empty`, for a grammar that produces the empty string, `has_one_empty_rule`, to describe a grammar that has a single empty rule among its set of rules (one whose left-hand side

[7] The results of this section are available in libraries `emptyrules.v`, `unitrules.v`, `useless.v`, `inaccessible.v` and `simplification.v`.

is the initial symbol), `has_no_empty_rules` for a grammar that has no empty rules at all and `start_symbol_not_in_rhs` to state that the start symbol does not appear in the right-hand side of any rule of the argument grammar.

The proof of `g_emp'_correct` is reduced to the proof of the equivalence of grammars `g` and `g_emp g`. The most complex part of this formalization, by far, is to prove this equivalence, as expressed by lemmas `derives_g_g_emp` and `derives_g_emp_g`. These lemmas state, respectively, that every sentential form of `g` (except for the empty string) is also generated by `g_emp` and that every sentential form of `g_emp` is also generated by `g`. While the second case was relatively straightforward, the first proved much more difficult. This happens because the application of a rule of `g` can cause a non-terminal symbol to be eliminated from the sentential form (if it is an empty rule), and for this reason we have to introduce a new structure and do many case analysis in the sentential form of `g` in order to determine the corresponding new rule of `g_emp g` that has to be used in the derivation. We believe that the root of this difficulty was the desire to follow strictly the informal proof of [2] (Theorem 5.1.5), which depends on an intuitive lemma (lemma 3.1.5), however not easily formalizable. Probably for this reason, the solution constructed in our formalization is definitely not easy or readable, and this motivates the continued search for a simpler and more elegant one.

Result (2) is achieved in only one step. We first define the relation `unit` such that, for any two non-terminal symbols X and Y, `unit X Y` is true when $X \Rightarrow^{+} Y$ ([2], p. 114). This means that Y can be derived from X by the use of one or more unit rules.

The mapping of grammar g_1 into an equivalent grammar g_2 such that g_2 is free of unit rules consists basically of keeping all non-unit rules of g_1 and creating new rules that reproduce the effect of the unit rules that were left behind. No new non-terminal symbols are necessary. The correctness of `g_unit` comes from the following theorem:

```
Theorem g_unit_correct: ∀ g: cfg non_terminal terminal,
g_equiv (g_unit g) g ∧ has_no_unit_rules (g_unit g).
```

The predicate `has_no_unit_rules` states that the argument grammar has no unit rules at all.

We find important similarities in the proofs of grammar equivalence for the elimination of empty rules (lemma `g_emp'_correct`) and the elimination of unit rules (lemma `g_unit_correct`). In both cases, going backwards (from the new to the original grammar) was relatively straightforward and required no special machinery. On the other hand, going forward (from the original to the new grammar) proved much more complex and required new definitions, functions and lemmas in order to complete the corresponding proofs.

The proof that every sentence generated by the original grammar is also generated by the transformed grammar (without unit rules) requires the introduction of the `derives3` predicate specially for this purpose. Because this definition represents the derivation of sentences directly from a non-terminal symbol, it is possible to abstract over the use of unit rules. Since `derives3` is a mutual inductive definition, we had to create a specialized induction principle (`derives3_ind_2`) and use it explicitly, which resulted in more complex proofs.

Result (3) is obtained in a single and simple step, which consists of inspecting all rules of grammar g_1 and eliminating the ones that contain useless symbols in either the left or right-hand side. The other rules are kept in the new grammar g_2. Thus, $P_2 \subseteq P_1$. No new non-terminals are required.

The `g_use` definition, of course, can only be used if the language generated by the original grammar is not empty, that is, if the start symbol of the original grammar is useful. If it were useless then it would be impossible to assign a root to the grammar and the language would be empty. The correctness of the useless symbol elimination operation is certified by proving theorem `g_use_correct`, which states that every context-free grammar whose start symbol is useful generates a language that can also be generated by an equivalent context-free grammar whose symbols are all useful.

```
Theorem g_use_correct: ∀ g: cfg non_terminal terminal,
non_empty g → g_equiv (g_use g) g ∧ has_no_useless_symbols (g_use g).
```

The predicates `non_empty`, and `has_no_useless_symbols` used above assert, respectively, that grammar `g` generates a language that contains at least one string (which in turn may or may not be empty) and the grammar has no useless symbols at all.

Result (4) is similar to the previous case: the rules of the original grammar g_1 are kept in the new grammar g_2 as long as their left-hand consist of accessible non-terminal symbols (by definition, if the left-hand side is accessible then all the symbols in the right-hand side of the same rule are also accessible). If this is not the case, then the rules are left behind. Thus, $P_2 \subseteq P_1$.

The correctness of the inaccessible symbol elimination operation is certified by proving theorem `g_acc_correct`, which states that every context-free grammar generates a language that can also be generated by an equivalent context-free grammar whose symbols are all accessible.

```
Theorem g_acc_correct: ∀ g: cfg non_terminal terminal,
g_equiv (g_acc g) g ∧ has_no_inaccessible_symbols (g_acc g).
```

In a way similar to `has_no_useless_symbols`, the absence of inaccessible symbols in a grammar is expressed by predicate `has_no_inaccessible_symbols` used above.

The proof of `g_acc_correct` is also natural when compared to the arguments of the informal proof. It has only 384 lines on Coq script and, despite the similarities between it and the proof of `g_use_correct`, it is still ~40% shorter than that. This is partially due to a difference in the definitions of `g_use_rules` and `g_acc_rules`: in the first case, in order to be eligible as a rule of `g_use`, a rule of `g` must provably consist only of useful symbols in both the left and right-hand sides; in the second, it is enough to prove that only the left-hand side is accessible (the rest is consequence of the definition). Since we have a few uses of the constructors of these definitions, the simpler definition of `g_acc_rules` resulted in simpler and shorter proofs. As a matter of fact, it should be possible to do something similar to the definition of g_use_rules, since the left-hand side of a rule is automatically useful once all the symbols in the right-hand side are proved useful (a consequence of the definition). This will be considered in a future review of the formalization.

So far we have only considered each simplification strategy independently of the others. If one wants to obtain a new grammar that is simultaneously free of empty and unit rules, and of useless and inaccessible symbols, it is not enough to consider the previous independent results: it is necessary to establish a suitable order to apply these simplifications, in order to guarantee that the final result satisfies all desired conditions. Then, it is necessary to prove that the claims do hold.

For the order, we should start with (i) the elimination of empty rules, followed by (ii) the elimination of unit rules. The reason for this is that (i) might introduce new unit rules in the grammar, and (ii) will surely not introduce empty rules, as long as the original grammar is free of them (except for $S \rightarrow \epsilon$, in which case S, the initial symbol of the grammar, must not appear on the right-hand side of any rule). Then, elimination of useless and inaccessible symbols (in either order) is the right thing to do, since they only remove rules from the original grammar (which is specially important because they do not introduce new empty or unit rules). The formalization of this result is captured in the following theorem:

```
Theorem g_simpl_ex_v1: ∀ g: cfg non_terminal terminal, non_empty g →
 ∃ g': cfg (non_terminal' non_terminal) terminal, g_equiv g' g ∧
 has_no_inaccessible_symbols g' ∧ has_no_useless_symbols g' ∧
(produces_empty g → has_one_empty_rule g') ∧
(∼ produces_empty g → has_no_empty_rules g') ∧
 has_no_unit_rules g' ∧ start_symbol_not_in_rhs g'.
```

The proof of g_simpl_ex_v1 demands auxiliary lemmas to prove that the characteristics of the initial transformations are preserved by the following ones. For example, that all of the unit rules elimination, useless symbol elimination and inaccessible symbol elimination operations preserve the characteristics of the empty rules elimination operation.

5 Chomsky Normal Form

The Chomsky Normal Form (CNF) theorem, proposed and proved by Chomsky in [6], asserts that $\forall\ G = (V, \Sigma, P, S),\ \exists\ G' = (V', \Sigma, P', S') \mid L(G) = L(G') \wedge \forall\ (\alpha \rightarrow_{G'} \beta) \in P', (\beta \in \Sigma) \vee (\beta \in N \cdot N)$.

That is, every context-free grammar can be converted to an equivalent one whose rules have only one terminal symbol or two non-terminal symbols in the right-hand side. Naturally, this is valid only if G does not generate the empty string. If this is the case, then the grammar that has this format, plus a single rule $S' \rightarrow_G \epsilon$, is also considered to be in the Chomsky Normal Form, and generates the original language, including the empty string. It can also be assured that in either case the start symbol of G' does not appear on the right-hand side of any rule of G'.

The existence of a CNF can be used for a variety of purposes, including to prove that there is an algorithm to decide whether an arbitrary context-free language accepts an arbitrary string, and to test if a language is not context-free (using the Pumping Lemma for context-free languages, which can be proved with the help of CNF grammars).

The idea of mapping G into G' consists of creating a finite number of new non-terminal symbols and new rules, in the following way:

1. For every terminal symbol σ that appears in the right-hand side of a rule $r = \alpha \rightarrow_G \beta_1 \cdot \sigma \cdot \beta_2$ of G, create a new non-terminal symbol $[\sigma]$, a new rule $[\sigma] \rightarrow_{G'} \sigma$ and substitute σ for $[\sigma]$ in r;
2. For every rule $r = \alpha \rightarrow_G N_1 N_2 \cdots N_k$ of G, where N_i are all non-terminals, create a new set of non-terminals and a new set of rules such that $\alpha \rightarrow_{G'} N_1[N_2 \cdots N_k], [N_2 \cdots N_k] \rightarrow_{G'} N_2[N_3 \cdots N_k], \cdots, [N_{k-2}N_{k-1}N_k] \rightarrow_{G'} N_{k-2} [\ N_{k-1}N_k\ ], [N_{k-1}N_k] \rightarrow_{G'} N_{k-1}N_k$.

Case (1) substitutes all terminal symbols of the grammar for newly created non-terminal symbols. Case (2) splits rules that have three or more non-terminal symbols on the right-hand side by a set of rules that have only two non-terminal symbols in the right-and side. Both changes preserve the language of the original grammar.

It is clear from above that the original grammar must be free of empty and unit rules in order to be converted to a CNF equivalent. Also, it is desirable that the original grammar contains no useless and no inaccessible symbols, besides assuring that the start symbol does not appear on the right-hand side of any rule. Thus, it will be required that the original grammar be first simplified according to the results of Section 4.

Given the original grammar g_1, we construct two new grammars g_2 and g_3. The first generates the same set of sentences of g_1, except for the empty string, and the second includes the empty string:

1. Construct g_2 such that $L(g_2) = L(g_1) - \epsilon$;
2. Construct g_3 (using g_2) such that $L(g_3) = L(g_2) \cup \{\epsilon\}$.

Then, either g_2 or g_3 will be used to prove the existence of a CNF grammar equivalent to g_1.

For step 1, the construction of g_2 (that is, `g_cnf g`) is more complex, as we need to substitute terminals for new non-terminals, introduce new rules for these non-terminals and also split the rules with three or more symbols on the right-hand side.

Next, we prove that g_2 is equivalent to `g` (or g_1). It should be noted, however, that the set of rules defined above do not generate the empty string. If this is the case, then we construct g_3 (that is, `g_cnf'`) with a new empty rule.

The statement of the CNF theorem can then be presented as:[8]

[8] The results of this section are available in library `chomsky.v`.

```
Theorem g_cnf_ex: ∀ g: cfg non_terminal terminal,
(produces_empty g ∨ ~produces_empty g) ∧
(produces_non_empty g ∨ ~produces_non_empty g) →
∃ g': cfg (non_terminal' (emptyrules.non_terminal' non_terminal) terminal)
terminal, g_equiv g' g ∧ (is_cnf g' ∨ is_cnf_with_empty_rule g') ∧
start_symbol_not_in_rhs g'.
```

The new predicates used above assert, respectively, that the argument grammar (i) produces at least one non-empty string (`produces_non_empty`), (ii) is in the Chomsky Normal Form (`is_cnf`) and (iii) is in the Chomsky Normal Form and has a single empty rule with the start symbol in the left-hand side (`is_cnf_with_empty_rule`).

It should be observed that the statement of `g_cnf_ex` is not entirely constructive, as we require, for any context-free grammar `g`, a proof that either `g` produces the empty string or `g` does not produce the empty string, and also that `g` produces a non-empty string or `g` does not produce a non-empty string. Since we have not yet included a proof of the decidability of these predicates in our formalization (something that we plan to do in the future), the statement of the lemma has to require such proofs explicitly. They are demanded, respectively, by the elimination of empty rules and elimination of useless symbols phases of grammar simplification.

The formalization of this section required a lot of insights not directly available from the informal proofs, the most important being the definition of the predicate `g_cnf_rules` (for the rules of the `g_cnf` grammar). In a first attempt, this inductive definition resulted with 14 constructors. Although correct, it was refined many times until it was simplified to only 4 after the definition of the type of the new non-terminals was adjusted properly, with a single constructor. This effort resulted in elegant definitions which allowed the simplification of the corresponding proofs, thus leading to a natural and readable formalization. In particular, the strategy used in the proof of lemma `derives_g_cnf_g` (which states that every sentence produced by `g_cnf g` is also produced by `g`) is a very simple and elegant one, which uses the information already available in the definition of `g_cnf_rules`.

6 Pumping Lemma

The Pumping Lemma is a property that is verified for all context-free languages (CFLs) and was stated and proved for the first time by Bar-Hillel, Perles and Shamir in 1961 ([7]). It does not characterize the CFLs, however, since it is also verified by some languages that are not context-free. It states that, for every context-free language and for every sentence of such a language that has a certain minimum length, it is possible to obtain an infinite number of new sentences that must also belong to the language. This minimum length depends only on the language defined. In other words (let $\mathcal{L}$ be defined over alphabet Σ): $\forall\ \mathcal{L}, (\text{cfl}\ \mathcal{L}) \rightarrow \exists\ n \,|\, \forall\ \alpha, (\alpha \in \mathcal{L}) \wedge (|\alpha| \geq n) \rightarrow \exists\ u, v, w, x, y \in \Sigma^* \,|\, (\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|vwx| \leq n)\ \wedge \forall\ i, uv^iwx^iy \in \mathcal{L}$.

The Pumping Lemma is stated in Coq as follows:[9, 10, 11, 12]

```
Lemma pumping_lemma: ∀ l: lang terminal,
(contains_empty l ∨ ∼ contains_empty l) ∧
(contains_non_empty l ∨ ∼ contains_non_empty l) → cfl l →
∃ n: nat, ∀ s: sentence, l s → length s ≥ n →
∃ u v w x y: sentence, s = u ++v ++w ++x ++y ∧
length (v ++x) ≥ 1 ∧ length (u ++y) ≥ 1 ∧ length (v ++w ++x) ≤ n ∧
∀ i: nat, l (u ++(iter v i) ++w ++(iter x i) ++y).
```

A typical use of the Pumping Lemma is to show that a certain language is not context-free by using the contrapositive of the statement of the lemma. The proof proceeds by assuming that the language is context-free, and this leads to a contradiction from which one concludes that the language in question can not be context-free.

The Pumping Lemma derives from the fact that the number of non-terminal symbols in any context-free grammar G that generates $\mathcal{L}$ is finite. There are different strategies that can be used to prove that the lemma can be derived from this fact. We searched through 13 proofs published in different textbooks and articles by different authors, and concluded that in 6 cases ([7], [8], [9], [10], [11] and [2]) the strategy uses CNF grammars and binary trees for representing derivations. Other 5 cases ([12], [13], [14], [15] and [16]) present tree-based proofs that however do not require the grammar to be in CNF. Finally, Harrison ([17]) proves the Pumping Lemma as a corollary to the more general Ogden's Lemma and Amarilli and Jeanmougin ([18]) use a strategy with pushdown automata instead of context-free grammars.

The difference between the proofs that use binary trees and those that use general trees is that the former uses $n = 2^k$ (where k is the number of non-terminal symbols the grammar) and the latter uses $n = m^k$ (where m is the length of the longest right-hand side among all rules of the grammar and k is the number of non-terminal symbols in the grammar). In both cases, the idea is the same: to show that sufficiently long sentences have parse trees for which a maximal path contains at least two instances of the same non-terminal symbol.

Since 11 out of 13 proofs considered use grammars and generic trees and, of these, 6 use CNF grammars and binary trees (including the authors of the original proof), this strategy was considered as the choice for the present work. Besides that, binary trees can be easily represented in Coq as simple inductive types, where generic trees require mutually inductive types, which increases the

[9] This statement contains the extra clause `length (u ++ y) >= 1`, corresponding to $|uy| \geq 1$, which is normally not mentioned in textbooks.

[10] Predicates `contains_empty` and `contains_non_empty` are indeed decidable and thus it would not be necessary to explicitly state that they satisfy the Law of the Excluded Middle. However, this property has not been addressed in the formalization yet, which justifies the statement of the lemma as it is.

[11] Application `iter l i` on a list `l` and a natural `i` yields list $\texttt{l}^{\texttt{i}}$.

[12] The results of this section are available in library `pumping.v`.

complexity of related proofs. Thus, for all these reasons we have adopted the proof strategy that uses CNF grammars and binary trees in what follows.

The classical proof considers that G is in the Chomsky Normal Form, which means that derivation trees have the simpler form of binary trees. Then, if the sentence has a certain minimum length, the frontier of the derivation tree should have two or more instances of the same non-terminal symbol in some path that starts in the root of this tree. Finally, the context-free character of G guarantees that the subtrees related to these duplicated non-terminal symbols can be cut and pasted in such a way that an infinite number of new derivation trees are obtained, each of which is related to a new sentence of the language. The formal proof presented here is based in the informal proof available in [3].

The proof of the Pumping Lemma starts by finding a grammar G that generates the input language L (this is a direct consequence of the predicate `cfl`, which states that the language is context-free). Next, we obtain a CNF grammar G' that is equivalent to G, using previous results. Then, G is substituted for G' and the value for n is defined as 2^k, where k is the length of the list of non-terminals of G' (which in turn is obtained from the predicate `rules_finite`). An arbitrary sentence α of $L(G')$ that satisfies the required minimum length n is considered. Lemma `derives_g_cnf_equiv_btree` is then applied in order to obtain a `btree` t that represents the derivation of α in G'. Naturally we have to ensure that $\alpha \neq \epsilon$, which is true since by assumption $|\alpha| \geq 2^k$.

The next step is to obtain a path (a sequence of non-terminal symbols ended by a terminal symbol) that has maximum length, that is, whose length is equal to the height of t plus 1. This is accomplished by means of the definition `bpath` and the lemma `btree_ex_bpath`. The length of this path (which is $\geq k+2$) allows one to infer that it must contain at least one non-terminal symbol that appears at least twice in it. This result comes from the application of the lemma `pigeon` which represents a list version of the well-known pigeonhole principle:

```
Lemma pigeon: ∀ A: Type, ∀ x y: list A, (∀ e: A, In e x → In e y) →
length x = length y + 1→ ∃ d: A, ∃ x1 x2 x3: list A, x = x1 ++[d] ++x2 ++[d] ++x3.
```

This lemma (and other auxiliary lemmas) is included in library pigeon.v, and its proof requires the use of classical reasoning (and thus library `Classical_Prop` of the Coq Standard Library). This is necessary in order to have a decidable equality on the type of the non-terminals of the grammar, and this is the only place in the whole formalization where this is required. Nevertheless, we plan to pursue in the future a constructive version of this proof.

Since a path is not unique in a tree, it is necessary to use some other representation that can describe this path uniquely, which is done by the predicate `bcode` and the lemma `bpath_ex_bcode`. A `bcode` is a sequence of boolean values that tell how to navigate in a `btree`. Lemma `bpath_ex_bcode` asserts that every path in a `btree` can be assigned a `bcode`.

Once the path has been identified with a repeated non-terminal symbol, and a corresponding `bcode` has been assigned to it, lemma `bcode_split` is applied twice in order to obtain the two subtrees t_1 and t_2 that are associated respectively to the first and second repeated non-terminals of t.

From this information it is then possible to extract most of the results needed to prove the goal, except for the pumping condition. This is obtained by an auxiliary lemma `pumping_aux`, which takes as hypothesis the fact that a tree t_1 (with frontier vwx) has a subtree t_2 (with frontier w), both with the same roots, and asserts the existence of an infinite number of new trees obtained by repeated substitution of t_2 by t_1 or simply t_1 by t_2, with respectively frontiers $v^iwx^i, i \geq 1$ and w, or simply $v^iwx^i, i \geq 0$.

The proof continues by showing that each of these new trees can be combined with tree t obtained before, thus representing strings $uv^iwx^iy, i \geq 0$ as necessary. Finally, we prove that each of these trees is related to a derivation in G', which is accomplished by lemma `btree_equiv_produces_g_cnf`.

The formalization of the Pumping Lemma is quite readable and easily modifiable to an alternative version that uses a smaller value of n (as in the original proof contained in [7]). It builds nicely on top of the previous results on grammar normalization, which in turn is a consequence of grammar simplification. It is however long (`pumping_lemma` has 436 lines of Coq script) and the key insights for its formalization were (i) the construction of the library trees.v, specially the lemmas that relate binary trees to CNF grammars; (ii) the identification and isolation of lemma `pumping_aux`, to show the pumping of subtrees in a binary tree and (iii) the proof of lemma `pigeon`. None of these aspects are clear from the informal proof, they showed up only while working in the formalization.

7 Related Work

Context-free language theory formalization is a relatively new area of research, when compared with the formalization of regular languages theory, with some results already obtained with the Coq, HOL4 and Agda proof assistants.

The pioneer work in context-free language theory formalization is probably the work by Filliâtre and Courant ([19]), which led to incomplete results (along with some important results in regular language theory) and includes closure properties (e.g. union), the partial equivalence between pushdown automata and context-free grammars and parts of a certified parser generator. No paper or documentation on this part of their work has been published however.

Most of the extensive effort started in 2010 and has been devoted to the certification and validation of parser generators. Examples of this are the works of Koprowski and Binsztok (using Coq, [20]), Ridge (using HOL4, [21]), Jourdan, Pottier and Leroy (using Coq, [22]) and, more recently, Firsov and Uustalu (in Agda, [23]). These works assure that the recognizer fully matches the language generated by the corresponding context-free grammar, and are important contributions in the construction of certified compilers.

On the more theoretical side, on which the present work should be considered, Norrish and Barthwal published on general context-free language theory formalization using the HOL4 proof assistant ([24], [25], [26]), including the existence of Chomsky and Greibach normal forms for grammars, the equivalence of pushdown automata and context-free grammars and closure properties. These results are from the PhD thesis of Barthwal ([27]), which includes also a proof of the Pumping Lemma for context-free languages. Thus, Barthwal extends our work with pushdown automata and Greibach Normal Form results, and for this reason it is the most complete formalization of context-free language theory up to date, in any proof assistant. Recently, Firsov and Uustalu proved the existence of a Chomsky Normal Form grammar for every general context-free grammar, using the Agda proof assistant ([28]). For a discussion of the similarities and differences of our work and those of Barthwal and Firsov, please refer to [4].

A special case of the Pumping Lemma for context-free languages, namely the Pumping Lemma for regular languages, is included in the comprehensive work of Doczkal, Kaiser and Smolka on the formalization of regular languages ([29]).

8 Conclusions

This is probably the most comprehensive formalization of the classical context-free language theory done to the present date in the Coq proof assistant, and includes the important result that is the second ever formalization of the Pumping Lemma for context-free languages (the first in the Coq proof assistant). It is also the first ever proof of the alternative statement of the Pumping Lemma that uses a smaller value of n (for more details, see [7] and [4]).

The whole formalization consists of 23,984 lines of Coq script spread in 18 libraries (each library corresponds to a different file), not including the example files. The libraries contain 533 lemmas and theorems, 99 constructors, 63 definitions (not including fixpoints), 40 inductive definitions and 20 fixpoints among 1,067 declared names.

The present work represents a relevant achievement in the areas of formal language theory and mathematical formalization. As explained before, there is no record that the author is aware of, of a project with a similar scope in the Coq proof assistant covering the formalization of context-free language theory. The results published so far are restricted to parser certification and theoretical results in proof assistants other than Coq. This is not the case, however, for regular language theory, and in a certain sense the present work can be considered as an initiative that complements and extends that work with the objective of offering a complete framework for reasoning with the two most popular and important language classes from the practical point of view. It is also relevant from the mathematical perspective, since there is a clear trend towards increased and widespread usage of interactive proof assistants and the construction of libraries for fundamental theories.

Plans for future development include the definition of new devices (e.g. pushdown automata) and results (e.g. equivalence of pushdown automata and context-free grammars), code extraction and general enhancements of the libraries, with migration of parts of development into SSReflect (to take advantage, for example, of finite type results).

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
2. Sudkamp, T.A.: Languages and Machines. 3rd edn. Addison-Wesley (2006)
3. Ramos, M.V.M., Neto, J.J., Vega, I.S.: Linguagens Formais: Teoria Modelagem e Implementação. Bookman (2009)
4. Ramos, M.V.M.: Formalization of Context-Free Language Theory. PhD thesis, Centro de Informática - UFPE (2016) `www.univasf.edu.br/~marcus.ramos/tese.pdf`, accessed May 05th, 2016.
5. Ramos, M.V.M.: Source files of [4] (2016) `https://github.com/mvmramos/v1`, accessed May 03rd, 2016.
6. Chomsky, A.N.: On certain formal properties of grammar. Information and Control **2** (1959) 137–167
7. Bar-Hillel, Y.: Language and information: selected essays on their theory and application. Addison-Wesley series in logic. Addison-Wesley Pub. Co. (1964)
8. Hopcroft, J.E., Ullman, J.D.: Formal Languages and Their Relation to Automata. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1969)
9. Davis, M.D., Sigal, R., Weyuker, E.J.: Computability, Complexity, and Languages (2Nd Ed.): Fundamentals of Theoretical Computer Science. Academic Press Professional, Inc., San Diego, CA, USA (1994)
10. Kozen, D.C.: Automata and Computability. Springer (1997)
11. Hopcroft, J.E., Motwani, R., Rotwani, Ullman, J.D.: Introduction to Automata Theory, Languages and Computability. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
12. Ginsburg, S.: The Mathematical Theory of Context-Free Languages. McGraw-Hill, Inc., New York, NY, USA (1966)
13. Denning, P.J., Dennis, J.B., Qualitz, J.E.: Machines, Languages and Computation. Prentice-Hall (1978)
14. Brookshear, J.G.: Theory of Computation: Formal Languages, Automata, and Complexity. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1989)
15. Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Second edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1998)
16. Sipser, M.: Introduction to the Theory of Computation. Second edn. International Thomson Publishing (2005)
17. Harrison, M.A.: Introduction to Formal Language Theory. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1978)
18. Amarilli, A., Jeanmougin, M.: A proof of the pumping lemma for context-free languages through pushdown automata. CoRR **abs/1207.2819** (2012)
19. INRIA: Coq users' contributions (2015) `http://www.lix.polytechnique.fr/coq/pylons/contribs/index`, accessed October 26th, 2015.
20. Koprowski, A., Binsztok, H.: TRX: A formally verified parser interpreter. In: Proceedings of the 19th European Conference on Programming Languages and Systems. ESOP'10, Berlin, Heidelberg, Springer-Verlag (2010) 345–365 `http://dx.doi.org/10.1007/978-3-642-11957-6%5f19`, accessed October 26th, 2015.
21. Ridge, T.: Simple, functional, sound and complete parsing for all context-free grammars. In: CPP. (2011) 103–118
22. Jourdan, J.H., Pottier, F., Leroy, X.: Validating LR(1) parsers. In: Proceedings of the 21st European Conference on Programming Languages and Systems. ESOP'12, Berlin, Heidelberg, Springer-Verlag (2012) 397–416

23. Firsov, D., Uustalu, T.: Certified {CYK} parsing of context-free languages. Journal of Logical and Algebraic Methods in Programming **83**(56) (2014) 459 – 468 The 24th Nordic Workshop on Programming Theory (NWPT 2012).
24. Barthwal, A., Norrish, M.: A formalisation of the normal forms of context-free grammars in HOL4. In Dawar, A., Veith, H., eds.: Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23–27, 2010. Proceedings. Volume 6247 of Lecture Notes in Computer Science., Springer (August 2010) 95–109
25. Barthwal, A., Norrish, M.: Mechanisation of PDA and grammar equivalence for context-free languages. In Dawar, A., de Queiroz, R.J.G.B., eds.: Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010. Volume 6188 of Lecture Notes in Computer Science. (2010) 125–135
26. Barthwal, A., Norrish, M.: A mechanisation of some context-free language theory in HOL4. Journal of Computer and System Sciences (WoLLIC 2010 Special Issue, A. Dawar and R. de Queiroz, eds.) **80**(2) (2014) 346 – 362
27. Barthwal, A.: A formalisation of the theory of context-free languages in higher order logic. PhD thesis, The Australian National University (2010) `https://digitalcollections.anu.edu.au/bitstream/1885/16399/1/Barthwal%20Thesis%202010.pdf`, accessed November 27th, 2015.
28. Firsov, D., Uustalu, T.: Certified normalization of context-free grammars. In: Proceedings of the 2015 Conference on Certified Programs and Proofs. CPP '15, New York, NY, USA, ACM (2015) 167–174
29. Doczkal, C., Kaiser, J.O., Smolka, G.: A constructive theory of regular languages in Coq. In Gonthier, G., Norrish, M., eds.: Certified Programs and Proofs. Volume 8307 of Lecture Notes in Computer Science. Springer International Publishing (2013) 82–97

A Definitions

We present next the main definitions used in the formalization.[13] Context-free grammars are represented in Coq very closely to the usual algebraic definition. Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The sets $N = V \backslash \Sigma$ and Σ are represented as types (usually denoted by names such as, for example, `non_terminal` and `terminal`), separately from G. The idea is that these sets are represented by inductive type definitions whose constructors are its inhabitants. Thus, the number of constructors in an inductive type corresponds exactly to the number of (non-terminal or terminal) symbols in a grammar.

Once these types have been defined, we can create abbreviations for sentential forms (`sf`), sentences (`sentence`) and lists of non-terminals (`nlist`). The first corresponds to the list of the disjoint union of the types `non-terminal` and `terminal`, while the other two correspond to simple lists of, respectively, `non-terminal` and `terminal` symbols.

The record representation `cfg` has been used for G. The definition states that `cfg` is a new type and contains three components. The first component is the `start_symbol` of the grammar (a non-terminal symbol) and the second is `rules`, that represents the rules of the grammar. Rules are propositions (represented

[13] The results of this appendix are available in libraries `cfg.v` and `cfl.v`.

in Coq by `Prop`) that take as arguments a non-terminal symbol and a (possibly empty) list of non-terminal and terminal symbols (corresponding, respectively, to the left and right-hand side of a rule). Grammars are parametrized by types `non_terminal` and `terminal`.

```
Record cfg (non_terminal terminal : Type): Type:= {
start_symbol: non_terminal;
rules: non_terminal → sf → Prop;
rules_finite:
   ∃ n: nat,
   ∃ ntl: nlist,
   ∃ tl: tlist,
   rules_finite_def start_symbol rules n ntl tl }.
```

The predicate `rules_finite_def` assures that the set of rules of the grammar is finite by proving that the length of right-hand side of every rule is equal or less than a given value, and also that both left and right-hand side of the rules are built from finite sets of, respectively, non-terminal and terminal symbols (represented here by lists). This represents an overhead in the definition of a grammar, but it is necessary in order to allow for the definition of `non_terminal` and `terminal` as generic types in Coq.

Since generic types might have an infinite number of elements, one must make sure that this is not the case when defining the `non_terminal` and `terminal` sets. Also, even if these types contain a finite number of inhabitants (constructors), it is also necessary to prove that the set of rules is finite. All of these is captured by predicate `rules_finite_def`. Thus, for every `cfg` defined directly of constructed from previous grammars, it will be necessary to prove that the predicate `rules_finite_def` holds.

The other fundamental concept used in this formalization is the idea of *derivation*: a grammar `g` *derives* a string `s2` from a string `s1` if there exists a series of rules in `g` that, when applied to `s1`, eventually results in `s2`. A direct derivation (i.e. the application of a single rule) is represented by $s_1 \Rightarrow s_2$, and the reflexive and transitive closure of this relation (i.e. the application of zero or more rules) is represented by $s_1 \Rightarrow^* s_2$. An inductive predicate definition of this concept in Coq (`derives`) uses two constructors:

```
Inductive derives
   (non_terminal terminal : Type)
   (g : cfg non_terminal terminal)
   : sf → sf → Prop :=
   | derives_refl :
      ∀ s : sf,
      derives g s s
   | derives_step :
      ∀ (s1 s2 s3 : sf)
      ∀ (left : non_terminal)
      ∀ (right : sf),
      derives g s1 (s2 ++inl left :: s3) →
      rules g left right → derives g s1 (s2 ++right ++s3)
```

The constructors of this definition (derives_refl and derives_step) are the axioms of our theory. Constructor derives_refl asserts that every sentential form s can be derived from s itself. Constructor derives_step states that if a sentential form that contains the left-hand side of a rule is derived by a grammar, then the grammar derives the sentential form with the left-hand side replaced by the right-hand side of the same rule. This case corresponds to the application of a rule in a direct derivation step.

A grammar generates a string if this string can be derived from its start symbol. Finally, a grammar produces a sentence if it can be derived from its start symbol.

Two grammars g_1 (with start symbol S_1) and g_2 (with start symbol S_2) are *equivalent* (denoted $g_1 \equiv g_2$) if they generate the same language, that is, $\forall s, (S_1 \Rightarrow^*_{g_1} s) \leftrightarrow (S_2 \Rightarrow^*_{g_2} s)$. This is represented in our formalization in Coq by the predicate g_equiv.

With these and other definitions (see [4]), it is possible to prove various lemmas about grammars and derivations, and also operations on grammars, all of which are useful when proving the main theorems of this work.

Library cfg.v contains 4,393 lines of Coq script (~18.3% of the total) and 105 lemmas and theorems (~19.7% of the total).

B Generic Binary Trees Library

In order to support the formalization of the Pumping Lemma in Section 6, an extensive library of definitions and lemmas on binary trees and their relation to CNF grammars has been developed.[14] This library is based in the definition of a binary tree (btree) whose internal nodes are non-terminal symbols and leaves are terminal symbols. The type btree is defined with the objective of representing derivation trees for strings generated by context-free grammars in the Chomsky Normal Form:

```
Inductive btree (non_terminal terminal: Type): Type:=
| bnode_1: non_terminal → terminal → btree
| bnode_2: non_terminal → btree → btree → btree.
```

The constructors of btree relate to the two possible forms that the rules of a CNF grammar can assume (namely with one terminal symbol or two non-terminal symbols in the right-hand side). Naturally, the inhabitants of the type btree can only represent the derivation of non-empty strings.

Next, we have to relate binary trees to CNF grammars. This is done with the predicate btree_cnf, used to assert that a binary tree bt represents a derivation in CNF grammar g. Now we can show that binary trees and derivations in CNF grammars are equivalent. This is accomplished by two lemmas, one for each direction of the equivalence. Lemma derives_g_cnf_equiv_btree asserts that for every derivation in a CNF grammar exists a binary tree that represents this derivation. It is general enough in order to accept that the input grammar

[14] The results of this appendix are available in library trees.v.

might either be a CNF grammar, or a CNF grammar with an empty rule. If this is the case, then we have to ensure that the derived sentence is not empty. Lemma btree_equiv_derives_g_cnf proves that every binary tree that satisfies btree_cnf corresponds to a derivation in the same (CNF) grammar.

Among other useful lemmas, the following one is of fundamental importance in the proof of the Pumping Lemma, as it relates the length of the frontier of a binary tree to its height:

```
Lemma length_bfrontier_ge:
∀ t: btree,
∀ i: nat,
length (bfrontier t) ≥ 2 ^ (i − 1) →
bheight t ≥ i.
```

The notion of subtree is also important, and is defined inductively as follows (note that a tree is not, in this definition, a subtree of itself):

```
Inductive subtree (t: btree): btree → Prop:=
| sub_br: ∀ tl tr: btree, ∀ n: non_terminal,
          t = bnode_2 n tl tr → subtree t tr
| sub_bl: ∀ tl tr: btree, ∀ n: non_terminal,
          t = bnode_2 n tl tr → subtree t tl
| sub_ir: ∀ tl tr t': btree, ∀ n: non_terminal,
          subtree tr t' → t = bnode_2 n tl tr → subtree t t'
| sub_il: ∀ tl tr t': btree, ∀ n: non_terminal,
          subtree tl t' → t = bnode_2 n tl tr → subtree t t'.
```

The following lemmas, related to subtrees, among many others, are also fundamental in the proof of the Pumping Lemma:

```
Lemma subtree_trans:
∀ t1 t2 t3: btree,
subtree t1 t2 → subtree t2 t3 → subtree t1 t3.

Lemma subtree_includes:
∀ t1 t2: btree,
subtree t1 t2 → ∃ l r : sentence,
bfrontier t1 = l ++bfrontier t2 ++r ∧ (l ≠ [] ∨ r ≠ []).
```

Library trees.v has 4,539 lines of Coq script (~18.9% of the total) and 84 lemmas (~15.7% of the total)).