Formalization of simplification for context-free grammars

Marcus Vinícius Midena Ramos
(UFPE/Recife & UNIVASF/Petrolina)

Ruy J. G. B. de Queiroz
(UFPE/Recife)

September 1st, 2015

10th LSFA, UFRN, Natal, RN

Scope

The objective of this work is to formalize a substantial part of context-free language theory in the Coq proof assistant, making it possible to reason about it in a fully checked environment, with all the related advantages.

- Formalization is the process of writing proofs such that they have a precise meaning over a simple and well-defined calculus whose rules can be automatically checked by a machine;
- Context-free language theory is fundamental in the representation and study of artificial languages, specially programming languages, and in the construction of their processors (compilers and interpreters);
- The formalization of context-free language theory is a key to the certification of compilers and programs, as well as to the development of new languages and tools for certified programming.

Overview

- Part of Formal Language Theory (Chomsky Hierarchy):
  - Regular Languages;
  - Context-Free Languages;
  - Context-Sensitive Languages;
  - Recursively Enumerable Languages.
- Developed from mid 1950s to late 1970s;
- Relevant to the representation, study and implementation of artificial languages;

Overview

Includes:

- Context-free grammars, pushdown automata and notations (e.g. BNF);
- Equivalence of grammars and automata;
- Grammar simplification;
- Normal forms;
- Derivation trees, parsing and ambiguity;
- Determinism and non-determinism;
- Closure properties;
- Decidable and undecidable problems;
- Relation with other language classes.

Related Work

- Regular languages have already been formalized to a large extend;
- Some formalization of context-free languages appeared in recent years, mostly in HOL4 and Agda;
- Mostly on parser certification or certified parser generation;
- Not much on theory.

Objectives

To formally state and prove the following fundamental results on context-free language theory:

1. Closure properties (9th LSFA 2014):
    - Union;
    - Concatenation;
    - Kleene star.
2. Grammar simplification:
    - Elimination of empty rules;
    - Elimination of unit;
    - Elimination of useless symbols;
    - Elimination of inaccessible symbols.
3. Chomsky Normal Form;
4. Pumping Lemma.

Context-Free Grammar

$G = (V, \Sigma, P, S)$, where:

- V is the vocabulary of G;
- Σ is the set of terminal symbols;
- $N = V \setminus \Sigma$ is the set of non-terminal symbols;
- P is the set of rules $\alpha \rightarrow \beta$, with $\alpha \in N$ and $\beta \in V^*$;
- $S \in N$ is the start symbol.

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

Context-Free Grammar

Making sure that `cfg` represents a context-free grammar:

- General types might have an infinite number of elements;
- We must check that the rules of the grammar are built from finite sets of terminal and non-terminal symbols;
- We must also check that the set of rules is finite;
- The predicate `rules_finite_def` is used to make sure that these conditions are satisfied for every grammar in the formalization, either user-defined or constructed;
- A list of non-terminal symbols (`ntl`), a list of terminal symbols (`tl`) and an upper bound on the length of the right-hand side of the rules (`n`) must be supplied.

Example

$G = (\{S', A, B, a, b\}, \{a, b\}, \{S' \to aS', S' \to b\}, S')$ generates the language a^*b.

```
Inductive nt1: Type:= | S' | A | B.
Inductive t1: Type:= | a | b.
Inductive rs1: nt1 → list (nt1 + t1) → Prop:=
  r1: rs1 S' [inr a; inl S']
| r2: rs1 S' [inr b].

Definition g1: cfg nt1 t1:= {|
start_symbol:= S';
rules:= rs1;
rules_finite:= rs1_finite |}.
```

Derivation

Substitution process:
s_1 *derives* s_2 by application of zero or more rules: $s_1 \Rightarrow^* s_2$.

```
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)
```

Derivation

- Predicate `generates`: a derivation that begins with the start symbol of the grammar;
- Predicate `produces`: a derivation that begins with the start symbol of the grammar and ends with a sentence.

$$\underbrace{\underbrace{S \Rightarrow \alpha_1 \Rightarrow \overbrace{\alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1}}^{\texttt{derives}}}_{\texttt{generates}} \Rightarrow \alpha_n \Rightarrow \omega}_{\texttt{produces}}$$

Example

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aab$

```
Lemma produces_g1_aab:
produces g1 [a; a; b].
Proof.
unfold produces.
unfold generates.
simpl.
apply derives_step with (s2:=[inr a; inr a])(left:=S')(right:=[inr b]).
apply derives_step with (s2:=[inr a])(left:=S')(right:=[inr a;inl S']).
apply derives_start with (left:=S')(right:=[inr a;inl S']).
apply r11.
apply r11.
apply r12.
Qed.
```

Grammar Equivalence

$g_1 \equiv g_2$
if they generate the same language, that is,
$\forall s, (S_1 \Rightarrow_{g_1}^{*} s) \leftrightarrow (S_2 \Rightarrow_{g_2}^{*} s)$

```
Definition g_equiv
(non_terminal1 non_terminal2 terminal : Type)
(g1: cfg non_terminal1 terminal)
(g2: cfg non_terminal2 terminal): Prop:=
∀ s: list terminal,
produces g1 s ↔ produces g2 s.
```

Context-Free Language

- A *language* is a set of strings over a given alphabet;
- A *context-free language* is a language that is generated by some context-free grammar: $L(G) = \{w \,|\, S \Rightarrow_g^* w\}$.

```
Definition lang (terminal: Type):= list terminal → Prop.

Definition lang_of_g (g: cfg): lang :=
fun w: list terminal ⇒ produces g w.

Definition lang_eq (l k: lang) :=
∀ w, l w ↔ k w.

Definition cfl (terminal: Type) (l: lang terminal): Prop:=
∃ non_terminal: Type,
∃ g: cfg non_terminal terminal,
lang_eq l (lang_of_g g).
```

Methodology

For closure properties, grammar simplification and Chomsky normal form:

1. Inductively define the new non-terminal symbols (if necessary);
2. Inductively define the rules of the new grammar;
3. Define the new grammar;
4. Show that the new grammar has the desired properties;
5. Consolidate the results.

Overview

Grammar simplification aims at obtaining new and simpler grammars that are equivalent to the original ones:

- Simpler means:
    - They contain only symbols and rules that are effectively used in the derivation of some sentence;
    - They do not contain unit rules (e.g. $A \rightarrow B$);
    - They do not contain empty rules (e.g. $A \rightarrow \epsilon$), except for a special case.
- Equivalent means that they generate the same language.

Important to reduce the complexity of grammars and thus (i) simplify its understanding, increase the efficiency of parsers obtained from them and (iii) allow their normalization.

Elimination of empty rules
Concept

- An *empty rule* $r \in P$ is a rule whose right-hand side β is empty (e.g. $X \rightarrow \epsilon$);
- We formalize that for all G, there exists G' such that $L(G) = L(G')$ and G' has no empty rules, except for a single rule $S \rightarrow \epsilon$ 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 in G'.

Elimination of empty rules

Definitions

```
Definition empty
(g: cfg terminal _) (s: non_terminal + terminal): Prop:=
derives g [s] [].

Inductive non_terminal'
(non_terminal : Type): Type:=
| Lift_nt: non_terminal → non_terminal'
| New_ss.

Definition g_emp
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal' terminal :=
  {| start_symbol:= New_ss;
     rules:= g_emp_rules g;
     rules_finite:= g_emp_finite g |}.
```

Elimination of empty rules

Definitions

```
Inductive g_emp_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' → sf' → Prop :=
| Lift_direct :
    ∀ left: non_terminal,
    ∀ right: sf,
    right ≠ [] → rules g left right →
    g_emp_rules g (Lift_nt left) (map symbol_lift right)
```

Elimination of empty rules

Definitions

```
| Lift_indirect:
    ∀ left: non_terminal,
    ∀ right: sf,
    g_emp_rules g (Lift_nt left) (map symbol_lift right)→
    ∀ s1 s2: sf,
    ∀ s: non_terminal,
    right = s1 ++(inl s) :: s2 →
    empty g (inl s) →
    s1 ++s2 ≠ [] →
    g_emp_rules g (Lift_nt left) (map symbol_lift (s1 ++s2))
| Lift_start_emp:
    g_emp_rules g New_ss [inl (Lift_nt (start_symbol g))].
```

Elimination of empty rules

Example

Suppose that X, A, B, C are non-terminals, of which A, B and C are nullable, a, b and c are terminals and $X \rightarrow aAbBcC$ is a rule of g. Then, the above definitions assert that $X \rightarrow aAbBcC$ is a rule of g_emp g, and also:

- $X \rightarrow aAbBc$;
- $X \rightarrow abBcC$;
- $X \rightarrow aAbcC$;
- $X \rightarrow aAbc$;
- $X \rightarrow abBc$;
- $X \rightarrow abcC$;
- $X \rightarrow abc$.

Elimination of empty rules

Definitions

```
Definition g_emp'
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg (non_terminal' _) terminal :=
  {| start_symbol:= New_ss _;
     rules:= g_emp'_rules g;
     rules_finite:= g_emp'_finite g |}.
```

Elimination of empty rules

Definitions

```
Inductive g_emp'_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' non_terminal → sf' → Prop :=
| Lift_all:
  ∀ left: non_terminal' _,
  ∀ right: sf',
  rules (g_emp g) left right → g_emp'_rules g left right
| Lift_empty:
  empty g (inl (start_symbol g)) →
  g_emp'_rules g (start_symbol (g_emp g)) [].
```

Elimination of empty rules

Correctness

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

Elimination of empty rules

Proof Outline

The definition of `g_equiv`, when applied to the previous theorem, yields:

```
∀ s: sentence,
produces (g_emp' g) s ↔ produces g s.
```

- For the $\rightarrow$ part, the strategy is to prove that for every rule $left \rightarrow_{g\_emp'} right$, either $left \rightarrow_g right$ is a rule of g or $left \Rightarrow_g^* right$;
- For the $\leftarrow$ part, the strategy is a more complicated one, and involves induction over the number of derivation steps in g.

Elimination of unit rules

Concept

- A *unit rule* $r \in P$ is a rule whose right-hand side β contains a single non-terminal symbol (e.g. $X \rightarrow Y$);
- We formalize that for all G, there exists G' such that $L(G) = L(G')$ and G' has no unit rules.

Elimination of unit rules

Definitions

```
Inductive unit
(terminal non_terminal : Type)
(g: cfg terminal non_terminal)
(a: non_terminal)
: non_terminal → Prop:=
| unit_rule:
  ∀ (b: non_terminal),
  rules g a [inl b] → unit g a b
| unit_trans:
  ∀ b c: non_terminal,
  unit g a b → unit g b c → unit g a c.
```

Elimination of unit rules

Definitions

```
Definition g_unit
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal terminal :=
  {| start_symbol:= start_symbol g;
     rules:= g_unit_rules g;
     rules_finite:= g_unit_finite g |}.
```

Elimination of unit rules

Definitions

```
Inductive g_unit_rules
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal → sf → Prop :=
| Lift_direct' :
   ∀ left: non_terminal,
   ∀ right: sf,
   (∀ r: non_terminal, right ≠ [inl r]) →
   rules g left right →
   g_unit_rules g left right
```

Elimination of unit rules

Definitions

```
Lift_indirect':
 ∀ a b: non_terminal,
 unit g a b →
 ∀ right: sf,
 rules g b right →
 (∀  c: non_terminal, right ≠ [inl c]) →
 g_unit_rules g a right.
```

Elimination of unit rules

Example

Suppose that $N = \{S', X, Y, Z\}$, $\Sigma = \{a, b, c\}$ and $P = \{S' \rightarrow X, X \rightarrow aX, X \rightarrow Y, Y \rightarrow XbY, Y \rightarrow Z, Z \rightarrow c\}$. The previous definitions assert that P' has the following rules:

- $S' \rightarrow aX$;
- $S' \rightarrow XbY$;
- $S' \rightarrow c$;
- $X \rightarrow aX$;
- $X \rightarrow XbY$;
- $X \rightarrow c$;
- $Y \rightarrow XbY$;
- $Y \rightarrow c$;
- $Z \rightarrow c$

Elimination of unit rules

Correctness

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

Elimination of unit rules

Proof Outline

Consider `g_equiv (g_unit g) g` of the previous statement:

- For the $\rightarrow$ part, the strategy adopted is to prove that for every rule $left \rightarrow_{g\_unit} right$ of (`g_unit g`), either $left \rightarrow_g right$ is a rule of g or $left \Rightarrow_g^* right$;
- For the $\leftarrow$ part, the strategy is also a more complicated one, and involves induction over a predicate that is equivalent to *derives* (*derives3*), but generates the sentence directly without considering the application of a sequence of rules, which allows one to abstract the application of unit rules in g.

Elimination of useless symbols

Concept

- A symbol $s \in V$ is *useful* if it is possible to derive a sentence 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 $\omega \in \Sigma^*$;
- We formalize 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.

Elimination of useless symbols

Definitions

```
Definition useful
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
(s: non_terminal + terminal): Prop:=
match s with
| inr t ⇒ True
| inl n ⇒ ∃ s: sentence, derives g [inl n] (map term_lift s)
end.
```

Elimination of useless symbols

Definitions

```
Definition g_use
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal terminal:=
  {| start_symbol:= start_symbol g;
     rules:= g_use_rules g;
     rules_finite:= g_use_finite g |}.
```

Elimination of useless symbols

Definitions

```
Inductive g_use_rules
(terminal non_terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal → sf → Prop :=
| Lift_use :
  ∀ left: non_terminal,
  ∀ right: sf,
  rules g left right →
  useful g (inl left) →
  (∀ s: non_terminal + terminal, In s right → useful g s) →
  g_use_rules g left right.
```

Elimination of useless symbols

Correctness

```
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).
```

Elimination of useless symbols

Proof Outline

Consider `g_equiv (g_use g) g` of the previous statement:

- The $\rightarrow$ part of the `g_equiv` proof is straightforward, since every rule of `g_use` is also a rule of `g`;
- For the converse, it is necessary to show that every symbol used in a derivation of `g` is useful, and thus all the rules used in this derivation also appear in `g_use`.

Elimination of inaccessible symbols

Concept

- A symbol $s \in V$ is *accessible* 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 formalize that for all G, there exists G' such that $L(G) = L(G')$ and G' has no inaccessible symbols.

Elimination of inaccessible symbols

Definitions

```
Definition accessible
(terminal non_terminal : Type)
(g : cfg non_terminal terminal)
(s: non_terminal + terminal): Prop:=
∃ s1 s2: sf, derives g [inl (start_symbol g)] (s1 ++s :: s2).
```

Elimination of inaccessible symbols

Definitions

```
Definition g_acc
(terminal non_terminal : Type)
(g : cfg non_terminal terminal)
: cfg non_terminal terminal :=
  {| start_symbol:= start_symbol g;
     rules:= g_acc_rules g;
     rules_finite:= g_acc_finite g |}.
```

Elimination of inaccessible symbols

Definitions

```
Inductive g_acc_rules
(terminal non_terminal : Type)
(g : cfg non_terminal terminal)
: non_terminal → sf → Prop :=
| Lift_acc : ∀ left: non_terminal,
  ∀ right: sf,
  rules g left right → accessible g (inl left) →
  g_acc_rules g left right.
```

Elimination of inaccessible symbols

Correctness

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

Elimination of inaccessible symbols

Proof Outline

Consider `g_equiv (g_acc g) g` of the previous statement:

- The $\rightarrow$ part of the `g_equiv` proof is also straightforward, since every rule of `g_acc` is also a rule of `g`;
- For the converse, it is necessary to show that every symbol used in the derivation of `g` is accessible, and thus the rules used in this derivation also appear in `g_acc`.

Unification

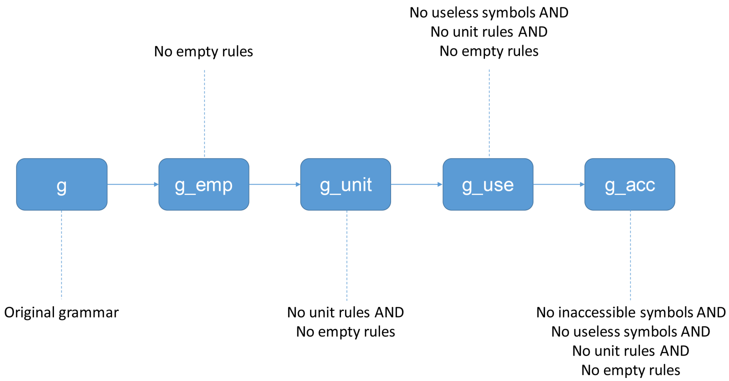
All in the Same Grammar

```
Theorem g_simpl:
∀ 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' ∧
(generates_empty g → has_one_empty_rule g') ∧
(∼ generates_empty g → has_no_empty_rules g') ∧
 has_no_unit_rules g' ∧
 start_symbol_not_in_rhs g'.
```

Unification

Proof Outline

Requires the proof that certain operations preserve some properties of the original grammar:



This Formalization

- Comprehensive set of fundamental results on context-free language theory;
- First formalization in Coq (preliminary work by Filliâtre);
- Enables the formalization of the Chomsky Normal Form and the Pumping Lemma;
- Framework to advance with the formalization of CFLs and related theories.

Current Status

- All objectives were reached in August/2015 (formalization complete);
- First formalization at all of the Pumping Lemma;
- 600+ lemmas and theorems, 20+ libraries, 25.000+ lines of scripts;
- 2 year effort;
- Declarative style;
  - Closer to textbook definitions;
  - More abstract to deal with;
  - Does not allow for the extraction of certified programs;
  - Efficiency issues;
  - Main objective was the Pumping Lemma.

Further Work

- Simplify the formalization with SSRreflect;
- Code extraction and certified algorithms;
- Formalize pushdown automata and other results of CFLs.