Formalization of Context-Free Language Theory

Marcus Vinícius Midena Ramos

Ruy J. G. B. de Queiroz
(Supervisor)

UFPE

January 18th, 2016

mvmr@cin.ufpe.br
(10 de janeiro de 2016, 19:23)

Introduction

Mathematical formalization

$+$

Context-free language theory

$=$

Formalization of context-free language theory

Introduction

Mathematical formalization

- Machine assisted proof construction;
- Machine verified proofs;
- Speed, reliability and reuse;
- Mathematics and computer science;
- Interactive theorem proving;
- Certified hardware and software development.

Introduction

Context-free language theory

- Language design, analysis and implementation;
- Computation theory;
- Fundamental in computing curricula and computation practice.

Introduction

Objectives

- Formalization of an important subset of context-free language theory;
- Using the Coq proof assistant (type theory).
- Research on:
  - Logics and natural deduction;
  - Lambda calculus;
  - Type theory;
  - Mathematical formalization;
  - Interactive theorem proving.
- Build a set of libraries that can be used in:
  - Education;
  - Certified software construction.
- Create and develop a culture of mathematical formalization.

Introduction

History

Background:

- 2011-2012: classes on lambda calculus, set theory and logic;
- 2013-2015: self study of proof theory, type theory and Coq;

Formalization:

- July 2013 until April 2014: regular languages, Coq as a functional programming language;
- April 2014 until August 2015: context-free languages, focus on lemmas and theorems;

Introduction

History

Presentations:

- 02/2014: WTA/EPUSP/USP;
- 02/2014: Thesis proposal examination;
- 09/2014: LSFA'14;
- 07/2015: DCC/FC/UP;
- 08/2015: LSFA'15.

Thesis writing:

- September 2015 until December 2015.

A Sampler of Formally Checked Projects

Mathematical formalization is a mature activity:

- Use over the years;
- Diversity of proof assistants and underlying theories;
- Development of proof assistants technology;
- Size, complexity and importance of many different projects;
- Theoretical and technologically oriented;
- Academy and industry oriented;
- A clear trend;
- A point of no return.

A Sampler of Formally Checked Projects

Some remarkable projects:

- Four Color Theorem;
- Odd Order Theorem;
- Kepler Conjecture;
- Homotopy Type Theory and Univalent Foundations of Mathematics;
- Compiler Certification;
- Microkernel Certification;
- Digital Security Certification.

Related Work

- Language and automata theory has been subject of formalization since the mid-1980s, when Kreitz used the Nuprl proof assistant to prove results about deterministic finite automata and the pumping lemma for regular languages;
- Since then, the theory of regular languages has been the subject of intense formalization by various researchers using many different proof assistants;
- The formalization of context-free language theory, on the other hand, is more recent and includes fewer accomplishments, mostly concentrated in certified parser generation.

Related Work

Context-free languages

- A recent and important reference is the work of Christian Doczkal, Jan-Oliver Kaiser and Gert Smolka;
- Following the structure of the book by Kozen, they did a fairly complete formalization of regular languages theory;
- All the development was done in Coq, is only 1,400 lines long, and benefited from the use of the SSReflect Coq plug-in.

Related Work

Context-free languages

- Most of the extensive effort, however, started in 2010 and has been devoted to the certification and validation of parser generators;
- On the more theoretical side, Norrish and Barthwal published in 2010 on general context-free language theory formalization using the HOL4 proof assistant, including:
  - The existence of normal forms for grammars;
  - Pushdown automata,
  - Closure properties and
  - A proof of the Pumping Lemma for context-free languages.
- In 2015, Firsov and Uustalu proved the existence of a Chomsky Normal Form grammar for every general context-free grammar, using the Agda proof assistant.

Related Work

Summary

	Norrish & Barthwal2010	**Firsov & Uustalu**
Proof assistant	`HOL4`	`Agda`
Closure	✓	×
Simplification	✓	*only empty and unit rules*
CNF	✓	✓
GNF	✓	×
PDA	✓	×
PL	✓	×

Related Work

Motivation

- Until 2015, the only comprehensive work is the one by Norrish and Barthwal (HOL4 in 2010);
- The Pumping Lemma has not been published;
- Firsov and Uustalu add a more limited implementation (Agda in 2015);
- No formalization in Coq.
- Formalization of the PL in HOL4 discovered only in november 2015.

Formalized Results

- Closure properties of context-free languages and grammars;
- Context-free grammar simplification;
- Chomsky Normal Form (CNF);
- Pumping Lemma (PL) for context-free languages.

PL depends on CNF, which in turn depends on grammar simplification.

Phases

1. Selection of an underlying formal logic to express the theory and then a tool that supports it adequately;
2. Representation of the objects of the universe of discourse in this logic;
3. Implementation of a set of basic transformations and mappings over these objects;
4. Statement of the lemmas and theorems that describe the properties and the behaviour of these objects, and establish a consistent and complete theory;
5. Formal derivation of proofs of these lemmas and theorems, leading to proof objects that can confirm their validity.

Definitions

- Symbols (including terminal and non-terminal);
- Sentential forms (strings of terminal and non-terminal symbols);
- Sentences (strings of terminal symbols);
- Context-free grammars;
- Derivations.

Sequence

1. General purpose libraries;
2. Closure properties;
3. Grammar simplification $\rightarrow$ Chomsky Normal Form $\rightarrow$ Pumping Lemma.

Support

- Basic lemmas on arithmetic, lists and logic;
- Basic lemmas on context-free languages and grammars;
- Basic lemmas on binary trees and their relation to CNF grammars;

Basic Definitions

Grammars

Terminal symbols as a type. Example:

```
Inductive nt: Type:=
| a
| b
| c.
```

Non-terminal symbols as a type. Example:

```
Inductive nt: Type:=
| X
| Y
| Z.
```

Basic Definitions

Grammars

Variables and notations:

```
Variables non_terminal terminal: Type.
Notation sf := (list (non_terminal + terminal)).
Notation sentence := (list terminal).
Notation nlist:= (list non_terminal).
```

Examples:

```
[inr a; inr a; inr b; inr c]
[inr a; inl X; inl Y; inr b]
[inl Z; inl Z; inl X]
```

Basic Definitions

Grammars

(V, Σ, P, S)

```
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 }.
```

Basic Definitions

Grammars

```
Definition rules_finite_def
   (non_terminal terminal : Type)
   (ss: non_terminal)
   (rules: non_terminal → sf → Prop)
   (n: nat)
   (ntl: list non_terminal)
   (tl: list terminal) :=
In ss ntl ∧
(∀ left: non_terminal,
 ∀ right: list (non_terminal + terminal),
 rules left right →
 length right ≤ n ∧
 In left ntl ∧
 (∀ s : non_terminal, In (inl s) right → In s ntl) ∧
 (∀ s : terminal, In (inr s) right → In s tl)).
```

Basic Definitions

Grammars

Example:

$$G = (\{S', A, B, a, b\}, \{a, b\}, \{S' \to aS', S' \to b\}, S')$$

that generates language a^*b:

```
Inductive nt: Type:= | S' | A | B.
Inductive t: Type:= | a | b.
Inductive rs: nt → list (nt + t) → Prop:=
  r1: rs S' [inr a; inl S']
| r2: rs S' [inr b].
```

Basic Definitions

Grammars

```
Lemma rs_finite:
∃ n: nat,
∃ ntl: nlist,
∃ tl: tlist,
In S' ntl ∧
∀ left: non_terminal,
∀ right: sf,
rs1 left right →
(length right ≤ n) ∧
(In left ntl) ∧
(∀  s: non_terminal, In (inl s) right → In s ntl) ∧
(∀  s: terminal, In (inr s) right → In s tl).
Proof.
admit.
Qed.
```

Basic Definitions

Grammars

```
Definition g: cfg nt t:= {|
start_symbol:= S';
rules:= rs;
rules_finite:= rs_finite |}.
```

Basic Definitions

Derivations

$s_1 \Rightarrow^* s_2$

```
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)
```

Basic Definitions

Derivations

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

```
Definition generates (g: cfg) (s: sf): Prop:=
derives g [inl (start_symbol g)] s.

Definition produces (g: cfg) (s: sentence): Prop:=
generates g (map terminal_lift s).
```

Basic Definitions

Derivations

Example:

```
Lemma derives_g_aab:
derives g [inl S'] [inr a; inr a; inr b].
Proof.
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 r1.
apply r1.
apply r2.
Qed.
```

Basic Definitions

Derivations

Examples:

- `derives g [inr a; inl S'] [inr a; inr b]`;
- `generates g [inl S'] [inr a; inl S']` and
- `produces g [inl S'] [inr a; inr b]`.

Basic Definitions

Derivations

```
Definition produces_empty
(g: cfg non_terminal terminal): Prop:=
produces g [].

Definition produces_non_empty
(g: cfg non_terminal terminal): Prop:=
∃ s: sentence, produces g s ∧ s ≠ [].
```

Basic Definitions

Derivations

```
Definition appears (g: cfg) (s: non_terminal + terminal): Prop:=
match s with
| inl n ⇒ ∃ left: non_terminal,
          ∃ right: sf,
          rules g left right ∧ ((n=left) ∨ (In (inl n) right))
| inr t ⇒ ∃ left: non_terminal,
          ∃ right: sf,
          rules g left right ∧ In (inr t) right
end.
```

Basic Definitions

Derivations

To map a sentence (sentence) into a sentential form (sf):

```
Definition terminal_lift (t: terminal):
non_terminal + terminal:=
inr t.
```

Basic Definitions

Derivations

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:

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

Basic Definitions

Languages

$$L(G) = \{w \mid S \Rightarrow_g^* w\}$$

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

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

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

Infix "==" := lang_eq (at level 80).
```

Basic Definitions

Languages

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

Definition contains_empty (l: lang): Prop:=
l [].

Definition contains_non_empty (l: lang): Prop:=
∃ w: sentence,
l w ∧ w ≠ [].
```

Generic CFG Library

General results on context-free gramars and languages:

- 4,393 lines of Coq script, $\sim$18.3% of the total;
- 105 lemmas and theorems;
- Alternative definitions for predicate `derives`;
- Supports the whole formalization;
- Some examples follow.

Generic CFG Library

- Derivation transitivity:
  $\forall g, s_1, s_2, s_3,\ (s_1 \Rightarrow_g^* s_2) \to (s_2 \Rightarrow_g^* s_3) \to (s_1 \Rightarrow_g^* s_3)$
- Context independence:
  $\forall g, s_1, s_2, s, s',\ (s_1 \Rightarrow_g^* s_2) \to (s \cdot s_1 \cdot s' \Rightarrow_g^* s \cdot s_2 \cdot s')$
- Concatenation:
  $\forall g, s_1, s_2, s_3, s_4,\ (s_1 \Rightarrow_g^* s_2) \to (s_3 \Rightarrow_g^* s_4) \to (s_1 \cdot s_3 \Rightarrow_g^* s_2 \cdot s_4)$
- Derivation independence: $\forall g, s_1, s_2, s_3,\ (s_1 \cdot s_2 \Rightarrow_g^* s_3) \to \exists s_1', s_2' \,|\, (s_3 = s_1' \cdot s_2') \wedge (s_1 \Rightarrow_g^* s_1') \wedge (s_2 \Rightarrow_g^* s_2')$
- Derivation of a string of terminals from a non-terminal symbol:
  $\forall g, s_1, s_2, n, w,\ (s_1 \cdot n \cdot s_2 \Rightarrow_g^* w) \to \exists\ w' \,|\, (n \Rightarrow_g^* w')$
- Direct or indirect derivation: $\forall g, n, w,\ (n \Rightarrow_g^* w) \to (n \to_g w) \vee (\exists\ right \,|\, n \to_g right \wedge right \Rightarrow_g^* w)$
- Grammar equivalence transitivity:
  $\forall g_1, g_2, g_3,\ (g_1 \equiv g_2) \wedge (g_2 \equiv g_3) \to (g_1 \equiv g_3)$

Generic CFG Library

Alternative definitions for predicate `derives`:

- Used to ease some proofs;
- Equivalence has been proved;
- Standard `derives` has been used in statements.

Generic CFG Library

```
Inductive derives2
  (non_terminal terminal : Type)
  (g : cfg non_terminal terminal)
  : sf → sf → Prop :=
  | derives2_refl :
     ∀ s : sf,
     derives2 g s s
  | derives2_step :
     ∀ (s1 s2 s3 : sf)
     ∀ (left : non_terminal)
     ∀ (right : sf),
     derives2 g (s1 ++right ++s2) s3 →
     rules g left right →
     derives2 g (s1 ++inl left :: s2) s3.
```

Generic CFG Library

```
Inductive derives3
   (g: cfg): non_terminal → sentence → Prop :=
   | derives3_rule:
     ∀ (n: non_terminal) (lt: sentence),
     rules g n (map inr lt) → derives3 g n lt
   | derives3_step:
     ∀ (n: non_terminal) (ltnt: sf) (lt: list terminal),
     rules g n ltnt → derives3_aux g ltnt lt → derives3 g n lt
with derives3_aux (g: cfg): sf → sentence → Prop :=
   | derives3_aux_empty:
     derives3_aux g [] []
   | derives3_aux_t:
     ∀ (t: terminal) (ltnt: sf) (lt: sentence),
     derives3_aux g ltnt lt → derives3_aux g (inr t :: ltnt) (t :: lt)
   | derives3_aux_nt:
     ∀ (n: non_terminal) (lt lt': sentence) (ltnt: sf),
     derives3_aux g ltnt lt → derives3 g n lt' →
   derives3_aux g (inl n :: ltnt) (lt' ++lt).
```

Generic CFG Library

```
Inductive derives6
  (non_terminal terminal : Type)
  (g : cfg non_terminal terminal)
  : nat → sf → sf → Prop :=
  | derives6_0 :
     ∀ s : sf,
     derives6 g 0 s s
  | derives6_sum :
     ∀ (left : non_terminal)
     ∀ (right : sf)
     ∀ (i : nat)
     ∀ (s1 s2 s3 : sf),
     rules g left right →
     derives6 g i (s1 ++right ++s2) s3 →
     derives6 g (S i) (s1 ++[inl left] ++s2) s3.
```

Generic CFG Library

The equivalence of definitions `derives`, `derives2`, `derives3` and `derives6` has been proved:

- `derives_equiv_derives2`, for
  `derives g s1 s2` $\leftrightarrow$ `derives2 g s1 s2`;
- `derives_equiv_derives3`, for
  `derives g n (map inr s)` $\leftrightarrow$ `derives3 g n s`;
- `derives_equiv_derives6`, for
  `derives g s1 s2` $\leftrightarrow \exists\, n$, `derives6 g n s1 s2`.

Method

Most of the work share a common objective: to construct a new grammar from an existing one (or two existing ones). This is the case of:

- Closure properties:
  - Union;
  - Concatenation;
  - Kleene star;
- Grammar simplification:
  - Elimination of empty rules;
  - Elimination of unit rules;
  - Elimination of useless symbols;
  - Elimination of inaccessible symbols;
- Chomsky Normal Form (CNF).

Thus, a common method to be used in all these cases has been devised.

Method

1. Depending on the case, define a new type of 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;
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;

Method

3. Define the new grammar by using the new 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.

Closure Properties

Union

Given two arbitrary context-free grammars g_1 and g_2, the following definitions are used 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).

Closure Properties

Union

- For the new set of non-terminals:
  - All the non-terminals of g_1;
  - All the non-terminals of g_2;
  - A fresh new non-terminal symbol (S_3).
- For the new set of rules:
  - All the rules of g_1;
  - All the rules of g_2;
  - Two new rules: $S_3 \to S_1$ and $S_3 \to S_2$.
- For the new grammar:
  - The new set of non-terminals;
  - The new set of rules;
  - The new non-terminal (S_3) as the start symbol.

Closure Properties

Union

```
Inductive g_uni_nt (non_terminal_1 non_terminal_2 : Type): Type:=
| Start_uni
| Transf1_uni_nt: non_terminal_1 → g_uni_nt
| Transf2_uni_nt: non_terminal_2 → g_uni_nt.

Notation sf1:= (list (non_terminal_1 + terminal)).
Notation sf2:= (list (non_terminal_2 + terminal)).
Notation sfu:= (list (g_uni_nt + terminal)).
```

Closure Properties

Union

```
Definition g_uni_sf_lift1 (c: non_terminal_1 + terminal)
: g_uni_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf1_uni_nt nt)
  | inr t  ⇒ inr t
  end.
Definition g_uni_sf_lift2 (c: non_terminal_2 + terminal)
: g_uni_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf2_uni_nt nt)
  | inr t  ⇒ inr t
  end.
```

Closure Properties

Union

```
Inductive g_uni_rules
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal): g_uni_nt → sfu → Prop :=
| Start1_uni:
  g_uni_rules g1 g2 Start_uni [inl (Transf1_uni_nt (start_symbol g1))]
| Start2_uni:
  g_uni_rules g1 g2 Start_uni [inl (Transf2_uni_nt (start_symbol g2))]
| Lift1_uni:
  ∀ nt: non_terminal_1, ∀ s: sf1,
  rules g1 nt s →
  g_uni_rules g1 g2 (Transf1_uni_nt nt) (map g_uni_sf_lift1 s)
| Lift2_uni:
  ∀ nt: non_terminal_2, ∀ s: sf2,
  rules g2 nt s →
  g_uni_rules g1 g2 (Transf2_uni_nt nt) (map g_uni_sf_lift2 s).
```

Closure Properties

Union

```
Definition g_uni
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal)
: (cfg g_uni_nt terminal):=
  {| start_symbol:= Start_uni;
     rules:= g_uni_rules g1 g2;
     rules_finite:= g_uni_finite g1 g2 |}.
```

Closure Properties

Union

Consider grammars G_1 and G_2:

- $G_1 = (\{S_1, X_1, a, b\}, \{a, b\}, \{S_1 \to aX_1, X_1 \to aX_1 \mid b\}, S_1)$;
- $G_2 = (\{S_2, X_2, a, b\}, \{a, b\}, \{S_2 \to aX_2, X_2 \to aX_2 \mid c\}, S_2)$.

Then, the new grammar G_3 that generates $L(G_1) \cup L(G_2)$ can be expressed as:

$$G_3 = (\{S_3, S_1, S_2, X_1, X_2, a, b, c\}, \{a, b, c\}, P_3, S_3)$$

with P_3 containing the following rules:

$$\begin{array}{rcl} S_3 & \to & S_1 \mid S_2 \\ S_1 & \to & aX_1 \\ X_1 & \to & aX_1 \mid b \\ S_2 & \to & aX_2 \\ X_2 & \to & aX_2 \mid c \end{array}$$

Closure Properties

Union

```
Inductive non_terminal1: Type:=
| S1
| X1.

Inductive non_terminal2: Type:=
| S2
| X2.

Inductive terminal: Type:=
| a
| b
| c.
```

Closure Properties

Union

```
Inductive rs1:
non_terminal1 → list (non_terminal1 + terminal) → Prop:=
| r11: rs1 S1 [inr a; inl X1]
| r12: rs1 X1 [inr a; inl X1]
| r13: rs1 X1 [inr b].

Definition g1: cfg non_terminal1 terminal := {|
start_symbol:= S1;
rules:= rs1;
rules_finite:= rs1_finite |}.
```

Closure Properties

Union

```
Inductive rs2:
non_terminal2 → list (non_terminal2 + terminal) → Prop:=
| r21: rs2 S2 [inr a; inl X2]
| r22: rs2 X2 [inr a; inl X2]
| r23: rs2 X2 [inr c].

Definition g2: cfg non_terminal2 terminal := {|
start_symbol:= S2;
rules:= rs2;
rules_finite:= rs2_finite |}.
```

Closure Properties

Union

```
Definition g3:= g_uni g1 g2.
```

Closure Properties

Concatenation

Given two arbitrary context-free grammars g_1 and g_2, the following definitions are used 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).

Closure Properties

Concatenation

- For the new set of non-terminals:
  - All the non-terminals of g_1;
  - All the non-terminals of g_2;
  - A fresh new non-terminal symbol (S_3).
- For the new set of rules:
  - All the rules of g_1;
  - All the rules of g_2;
  - One new rule: $S_3 \rightarrow S_1 S_2$.
- For the new grammar:
  - The new set of non-terminals;
  - The new set of rules;
  - The new non-terminal (S_3) as the start symbol.

Closure Properties

Concatenation

```
Inductive g_cat_nt (non_terminal_1 non_terminal_2 terminal : Type):
Type:=
| Start_cat
| Transf1_cat_nt: non_terminal_1 → g_cat_nt
| Transf2_cat_nt: non_terminal_2 → g_cat_nt.

Notation sf1:= (list (non_terminal_1 + terminal)).
Notation sf2:= (list (non_terminal_2 + terminal)).
Notation sfc:= (list (g_cat_nt + terminal)).
```

Closure Properties

Concatenation

```
Definition g_cat_sf_lift1 (c: non_terminal_1 + terminal):
g_cat_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf1_cat_nt nt)
  | inr t  ⇒ inr t
  end.

Definition g_cat_sf_lift2 (c: non_terminal_2 + terminal):
g_cat_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf2_cat_nt nt)
  | inr t  ⇒ inr t
  end.
```

Closure Properties

Concatenation

```
Inductive g_cat_rules
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal): g_cat_nt → sfc → Prop :=
| New_cat:
   g_cat_rules g1 g2 Start_cat
   ([ inl (Transf1_cat_nt (start_symbol g1))]++
    [ inl (Transf2_cat_nt (start_symbol g2))])
| Lift1_cat:
   ∀ nt s,
   rules g1 nt s →
   g_cat_rules g1 g2 (Transf1_cat_nt nt) (map g_cat_sf_lift1 s)
| Lift2_cat:
   ∀ nt s,
   rules g2 nt s →
   g_cat_rules g1 g2 (Transf2_cat_nt nt) (map g_cat_sf_lift2 s).
```

Closure Properties

Concatenation

```
Definition g_cat
(non_terminal_1 non_terminal_2 terminal : Type)
(g1: cfg non_terminal_1 terminal)
(g2: cfg non_terminal_2 terminal)
: (cfg g_cat_nt terminal):=
  {| start_symbol:= Start_cat;
     rules:= g_cat_rules g1 g2;
     rules_finite:= g_cat_finite g1 g2 |}.
```

Closure Properties

Concatenation

Consider grammars G_1 and G_2:

- $G_1 = (\{S_1, X_1, a, b\}, \{a, b\}, \{S_1 \to aX_1, X_1 \to aX_1 \mid b\}, S_1)$;
- $G_2 = (\{S_2, X_2, a, b\}, \{a, b\}, \{S_2 \to aX_2, X_2 \to aX_2 \mid c\}, S_2)$.

Then, the new grammar G_3 that generates $L(G_1) \cdot L(G_2)$ can be expressed as:

$$G_3 = (\{S_3, S_1, S_2, X_1, X_2, a, b, c\}, \{a, b, c\}, P_3, S_3)$$

with P_3 containing the following rules:

$$\begin{array}{rcl} S_3 & \to & S_1 S_2 \\ S_1 & \to & aX_1 \\ X_1 & \to & aX_1 \mid b \\ S_2 & \to & aX_2 \\ X_2 & \to & aX_2 \mid c \end{array}$$

Closure Properties

Concatenation

```
Definition g3:= g_cat g1 g2.
```

Closure Properties

Kleene star

Given an arbitrary context-free grammar g_1, the following definitions are used 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).

Closure Properties

Kleene star

- For the new set of non-terminals:
  - All the non-terminals of g_1;
  - A fresh new non-terminal symbol (S_2).
- For the new set of rules:
  - All the rules of g_1;
  - Two new rules: $S_2 \rightarrow S_2 S_1$ and $S_2 \rightarrow \epsilon$.
- For the new grammar:
  - The new set of non-terminals;
  - The new set of rules;
  - The new non-terminal (S_2) as the start symbol.

Closure Properties

Kleene star

```
Inductive g_clo_nt (non_terminal : Type): Type :=
| Start_clo : g_clo_nt
| Transf_clo_nt : non_terminal → g_clo_nt.

Notation sfc:= (list (g_clo_nt + terminal)).
```

Closure Properties

Kleene star

```
Definition g_clo_sf_lift (c: non_terminal + terminal):
g_clo_nt + terminal:=
  match c with
  | inl nt ⇒ inl (Transf_clo_nt nt)
  | inr t  ⇒ inr t
  end.
```

Closure Properties

Kleene star

```
Inductive g_clo_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: g_clo_nt → sfc → Prop :=
| New1_clo:
   g_clo_rules g Start_clo ([inl Start_clo] ++
   [inl (Transf_clo_nt (start_symbol g))])
| New2_clo:
   g_clo_rules g Start_clo []
| Lift_clo:
   ∀ nt: non_terminal,
   ∀ s: sf,
   rules g nt s →
   g_clo_rules g (Transf_clo_nt nt) (map g_clo_sf_lift s).
```

Closure Properties

Kleene star

```
Definition g_clo (g: cfg non_terminal terminal):
(non_terminal terminal : Type)
(g: cfg g_clo_nt terminal):=
 {| start_symbol:= Start_clo;
    rules:= g_clo_rules g;
    rules_finite:= g_clo_finite g |}.
```

Closure Properties

Kleene star

Consider once more grammar

$$G_1 = (\{S_1, X_1, a, b\}, \{a, b\}, \{S_1 \rightarrow aX_1, X_1 \rightarrow aX_1 \mid b\}, S_1)$$

Then, the new grammar G_2 that generates $L(G_1)^*$ can be expressed as:

$$G_2 = (\{S_2, S_1, X_1, a, b, c\}, \{a, b, c\}, P_2, S_2)$$

with P_2 containing the following rules:

$$\begin{array}{lcl} S_2 & \rightarrow & \epsilon \\ S_2 & \rightarrow & S_2 S_1 \\ S_1 & \rightarrow & aX_1 \\ X_1 & \rightarrow & aX_1 \mid b \end{array}$$

Closure Properties

Kleene star

```
Definition g2:= g_clo g1.
```

Closure Properties

Correctness and Completeness

$\boxed{\text{Concatenation (correctness)}}$

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)

$$\forall g_1,\, g_2,\, s_1,\, s_2, (S_1 \Rightarrow^*_{g_1} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2) \rightarrow (S_3 \Rightarrow^*_{g_3} s_1 s_2)$$

Closure Properties

Correctness and Completeness

Concatenation (correctness)

```
Theorem g_cat_correct:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s1: sf1,
∀ s2: sf2,
generates g1 s1 ∧ generates g2 s2 →
generates (g_cat g1 g2)
        ((map g_cat_sf_lift1 s1)++(map g_cat_sf_lift2 s2)).
```

Closure Properties

Correctness and Completeness

Concatenation (completeness)

$$\forall s_3, (S_3 \Rightarrow^*_{g_3} s_3) \rightarrow \exists s_1, s_2 \,|\, (s_3 = s_1 \cdot s_2) \wedge (S_1 \Rightarrow^*_{g_1} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2)$$

```
Theorem g_cat_correct_inv:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s: sfc,
generates (g_cat g1 g2) s →
s = [inl (start_symbol (g_cat g1 g2))] ∨
∃ s1: sf1,
∃ s2: sf2,
s =(map g_cat_sf_lift1 s1)++(map g_cat_sf_lift2 s2) ∧
generates g1 s1 ∧ generates g2 s2.
```

Closure Properties

Correctness and Completeness

$\boxed{\text{Union (correctness)}}$

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):

$$\forall g_1,\ g_2,\ s_1,\ s_2, (S_1 \Rightarrow^*_{g_1} s_1 \rightarrow S_3 \Rightarrow^*_{g_3} s_1) \wedge (S_2 \Rightarrow^*_{g_2} s_2 \rightarrow S_3 \Rightarrow^*_{g_3} s_2)$$

Closure Properties

Correctness and Completeness

Union (correctness)

```
Theorem g_uni_correct:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s1: sf1,
∀ s2: sf2,
(generates g1 s1 → generates (g_uni g1 g2) (map g_uni_sf_lift1 s1))
∧
(generates g2 s2 → generates (g_uni g1 g2) (map g_uni_sf_lift2 s2)).
```

Closure Properties

Correctness and Completeness

Union (completeness)

$$\forall s_3, (S_3 \Rightarrow^*_{g_3} s_3) \rightarrow (S_1 \Rightarrow^*_{g_1} s_3) \vee (S_2 \Rightarrow^*_{g_2} s_3)$$

```
Theorem g_uni_correct_inv:
∀ g1: cfg non_terminal_1 terminal,
∀ g2: cfg non_terminal_2 terminal,
∀ s: sfu,
generates (g_uni g1 g2) s →
(s=[inl (start_symbol (g_uni g1 g2))]) ∨
(∃ s1: sf1, (s=(map g_uni_sf_lift1 s1) ∧ generates g1 s1)) ∨
(∃ s2: sf2, (s=(map g_uni_sf_lift2 s2) ∧ generates g2 s2)).
```

Closure Properties

Correctness and Completeness

Kleene star (correctness)

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):

$$\forall g_1,\ s_1,\ s_2, (S_2 \Rightarrow^*_{g_2} \epsilon) \land ((S_2 \Rightarrow^*_{g_2} s_2) \land (S_1 \Rightarrow^*_{g_1} s_1) \to S_2 \Rightarrow^*_{g_2} s_2 \cdot s_1)$$

Closure Properties

Correctness and Completeness

Kleene star (correctness)

```
Theorem g_clo_correct:
∀ g: cfg non_terminal terminal,
∀ s: sf,
∀ s': sfc,
generates (g_clo g) nil ∧ (generates (g_clo g) s' ∧ generates g s →
generates (g_clo g) (s'++ map g_clo_sf_lift s)).
```

Closure Properties

Correctness and Completeness

Kleene star (completeness)

$$\forall s_2, (S_2 \Rightarrow^*_{g_2} s_2) \rightarrow (s_2 = \epsilon) \vee$$

$$(\exists\ s_1,\ s'_2 \,|\, (s_2 = s'_2 \cdot s_1) \wedge (S_2 \Rightarrow^*_{g_2} s'_2) \wedge (S_1 \Rightarrow^*_{g_1} s_1))$$

```
Theorem g_clo_correct_inv:
∀ g: cfg non_terminal terminal,
∀ s: sfc,
generates (g_clo g) s →
(s=[]) ∨
(s=[inl (start_symbol (g_clo g))]) ∨
(∃ s': sfc,
 ∃ s'': sf,
 generates (g_clo g) s' ∧ generates g s'' ∧ s=s' ++map g_clo_sf_lift s'').
```

Closure Properties

Correctness and Completeness

Proof strategy

Induction over the predicate `derives` or one of its variants.

Closure Properties

Closure over Languages

Definitions

```
Inductive l_uni (terminal : Type) (l1 l2: lang terminal):
lang terminal:=
| l_uni_l1: ∀ s: sentence, l1 s → l_uni l1 l2 s
| l_uni_l2: ∀ s: sentence, l2 s → l_uni l1 l2 s.

Inductive l_cat (terminal : Type) (l1 l2: lang terminal):
lang terminal:=
| l_cat_app: ∀ s1 s2: sentence, l1 s1 → l2 s2 → l_cat l1 l2 (s1 ++s2).

Inductive l_clo (terminal : Type) (l: lang terminal):
lang terminal:=
| l_clo_nil: l_clo l []
| l_clo_app: ∀ s1 s2: sentence, (l_clo l) s1 → l s2 → l_clo l (s1 ++s2).
```

Closure Properties

Closure over Languages

Proof strategy

- Correctness and completeness of union, concatenation and Kleene star: trivial from definitions;
- Non-trivial for `l_uni`, `l_cat` and `l_clo` being context-free languages: use the definition of CFL, find corresponding CFGs and use previous results.

Closure Properties

Closure over Languages

```
Theorem l_uni_is_cfl:
∀ l1 l2: lang terminal,
cfl l1 → cfl l2 → cfl (l_uni l1 l2).

Theorem l_cat_is_cfl:
∀ l1 l2: lang terminal,
cfl l1 → cfl l2 → cfl (l_cat l1 l2).

Theorem l_clo_is_cfl:
∀ l: lang terminal,
cfl l → cfl (l_clo l).
```

Grammar Simplification

Construct an equivalent grammar, free of:

1. Empty rules;
2. Unit rules;
3. Useless symbols;
4. Inaccessible symbols.

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'.

Grammar Simplification

Empty rule

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 of G'.

Grammar Simplification

Empty rules elimination

Nullable symbol:

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

Grammar Simplification

Empty rules elimination

Strategy for g_1:

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

Grammar Simplification

Empty rules elimination

Step 1:

- For the new set of non-terminals:
  - All the non-terminals of g_1;
  - A fresh new non-terminal symbol (S_2).
- For the new set of rules:
  - All non-empty rules of g_1;
  - All rules of g_1 with every combination on nullable symbols in the right-hand side removed, except if empty;
  - One new rule: $S_2 \rightarrow S_1$.
- For the new grammar:
  - The new set of non-terminals;
  - The new set of rules;
  - The new non-terminal (S_2) as the start symbol.

Grammar Simplification

Empty rules elimination

```
Inductive non_terminal': Type:=
| Lift_nt: non_terminal → non_terminal'
| New_ss.

Notation sf' := (list (non_terminal' + terminal)).

Definition symbol_lift
(s: non_terminal + terminal): non_terminal' + terminal:=
match s with
| inr t ⇒ inr t
| inl n ⇒ inl (Lift_nt n)
end.
```

Grammar Simplification

Empty rules elimination

```
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)
```

Grammar Simplification

Empty rules elimination

```
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))
```

Grammar Simplification

Empty rules elimination

```
Lift_start_emp:
  g_emp_rules g New_ss [inl (Lift_nt (start_symbol g))].
```

Grammar Simplification

Empty rules elimination

```
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 |}.
```

Grammar Simplification

Empty rules elimination

Suppose, for example, 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:

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

Grammar Simplification

Empty rules elimination

Step 2:

- For the new set of non-terminals:
  - All the non-terminals of Step 1.
- For the new set of rules:
  - All the rules of Step 1;
  - One new rule: $S_2 \rightarrow \epsilon$ if $\epsilon \in L(g_1)$.
- For the new grammar:
  - The same set of non-terminals;
  - The new set of rules;
  - The same start symbol (S_2).

Grammar Simplification

Empty rules elimination

```
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)) [].
```

Grammar Simplification

Empty rules elimination

```
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 |}.
```

Grammar Simplification

Empty rules elimination

```
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).
```

Grammar Simplification

Empty rules elimination

```
Definition has_one_empty_rule (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right →
((left = start_symbol g) ∧ (right = []) ∨ right ≠ []).

Definition has_no_empty_rules (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right → right ≠ [].
```

Grammar Simplification

Empty rules elimination

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

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

For the → part, the strategy used was 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 ← part, the strategy was more complicated, and involves induction over the number of derivation steps in g.

Grammar Simplification
Unit rule

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.

Grammar Simplification

Unit rules elimination

```
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.
```

Grammar Simplification

Unit rules elimination

For g_1:

- For the new set of non-terminals:
  - All the non-terminals of g_1.
- For the new set of rules:
  - All non-unit rules of g_1;
  - New rules: one for each `a`, `b`, $right$ such that (i) `unit a b`, (ii) $b \rightarrow right$, (iii) $right$ is not a single non-terminal; the new rule becomes $a \rightarrow right$.
- For the new grammar:
  - The same set of non-terminals;
  - The new set of rules;
  - The same start symbol (S_1).

Grammar Simplification

Unit rules elimination

```
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
```

Grammar Simplification

Unit rules elimination

```
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.
```

Grammar Simplification

Unit rules elimination

```
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 |}.
```

Grammar Simplification

Unit rules elimination

As an example, consider the grammar $G = (S, X, Y, Z, a, b, c, a, b, c, P, S)$, with P containing the following rules:

$$\begin{array}{rcl} S & \rightarrow & X \mid ab \\ X & \rightarrow & Y \mid bc \\ Y & \rightarrow & Z \mid ac \\ Z & \rightarrow & abc \end{array}$$

The above definitions assert that the new grammar G' (the grammar that is equivalent to G and is free of unit rules) has the following rules:

$$\begin{array}{rcl} S & \rightarrow & abc \mid ac \mid bc \mid ab \\ X & \rightarrow & abc \mid ac \mid bc \\ Y & \rightarrow & abc \mid ac \\ Z & \rightarrow & abc \end{array}$$

Grammar Simplification

Unit rules elimination

```
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:

```
Definition has_no_unit_rules (g: cfg non_terminal terminal): Prop:=
∀ left n: non_terminal,
∀ right: sf,
rules g left right → right ≠ [inl n].
```

Grammar Simplification

Unit rules elimination

For the $\rightarrow$ part of the `g_equiv (g_unit g) g` proof, the strategy adopted was 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 was more complicated, 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.

Grammar Simplification

Useless symbol

A symbol $s \in V$ is *useful* 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 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.

Grammar Simplification

Useless symbol elimination

```
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.
```

Grammar Simplification

Useless symbol elimination

For g_1:

- For the new set of non-terminals:
  - All the non-terminals of g_1.
- For the new set of rules:
  - All rules of g_1, except those that have useless symbols.
- For the new grammar:
  - The same set of non-terminals;
  - The new set of rules;
  - The same start symbol (S_1, which must be useful).

Grammar Simplification

Useless symbol elimination

```
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.
```

Grammar Simplification

Useless symbol elimination

```
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 |}.
```

Grammar Simplification

Useless symbol elimination

As an example, consider grammar $G = (X, X, Y, Z, a, b, c, a, b, c, P, S)$, with P containing the following rules:

$$\begin{array}{rcl} S & \rightarrow & Xa \mid Ya \mid Za \\ X & \rightarrow & aX \mid bY \\ Y & \rightarrow & aY \mid bX \\ Z & \rightarrow & bZ \mid c \end{array}$$

Clearly, symbols X and Y are useless symbols and can thus be removed from G, resulting in G' with the following set of rules:

$$\begin{array}{rcl} S & \rightarrow & Za \\ Z & \rightarrow & bZ \mid c \end{array}$$

Grammar Simplification

Useless symbol elimination

```
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).

Definition non_empty (g: cfg non_terminal terminal):
Prop:=
useful g (inl (start_symbol g)).

Definition has_no_useless_symbols (g: cfg non_terminal terminal):
Prop:=
∀ n: non_terminal, appears g (inl n) → useful g (inl n).
```

Grammar Simplification

Useless symbol elimination

- Hypothesis `non_empty g` on lemma `g_use_correct` is necessary in order to assure that the new grammar will have a start symbol (the start symbol should be a useful symbol, otherwise it would not be possible to obtain a new grammar free of useless symbols).
- 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 a the derivation of `g` is useful, and thus the rules used in this derivation also appear in `g_use`.

Grammar Simplification

Inaccessible symbol

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.

Grammar Simplification

Inaccessible symbol elimination

```
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).
```

Grammar Simplification

Inaccessible symbol elimination

For g_1:

- For the new set of non-terminals:
  - All the non-terminals of g_1.
- For the new set of rules:
  - All rules of g_1, except those that have inaccessible symbols.
- For the new grammar:
  - The same set of non-terminals;
  - The new set of rules;
  - The same start symbol (S_1).

Grammar Simplification

Inaccessible symbol elimination

```
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.
```

Grammar Simplification

Inaccessible symbol elimination

```
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 |}.
```

Grammar Simplification

Inaccessible symbol elimination

As an example, consider grammar $G = (X, X, Y, Z, a, b, c, a, b, c, P, S)$, with P containing the following rules:

$$\begin{array}{rcl} S & \rightarrow & aX \mid bX \\ X & \rightarrow & aX \mid bX \mid a \mid b \\ Y & \rightarrow & cZ \mid a \\ Z & \rightarrow & cZ \mid b \end{array}$$

Clearly, symbols Y, Z and c are inaccessible symbols and can thus be removed from G, resulting in G' with the following set of rules:

$$\begin{array}{rcl} S & \rightarrow & aX \mid bX \\ X & \rightarrow & aX \mid bX \mid a \mid b \end{array}$$

Grammar Simplification

Inaccessible symbol elimination

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

Definition has_no_inaccessible_symbols (g: cfg non_terminal terminal):
∀ s: (non_terminal + terminal), appears g s → accessible g s.
```

The → 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.

Grammar Simplification

Unification

```
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' ∧
(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'.
```

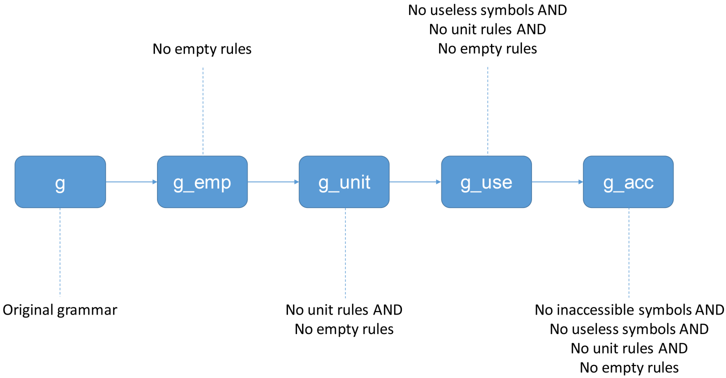
Grammar Simplification

Unification

```
Definition start_symbol_not_in_rhs (g: cfg non_terminal terminal):=
∀ left: non_terminal,
∀ right: sf,
rules g left right → ~ In (inl (start_symbol g)) right.
```

Grammar Simplification

Unification



Chomsky Normal Form

Definition

$$\forall\ G = (V, \Sigma, P, S),\ \exists\ G' = (V', \Sigma, P', S')\ |$$
$$L(G) = L(G') \wedge \forall\ (\alpha \rightarrow_{G'} \beta) \in P', (\beta \in \Sigma) \vee (\beta \in N \cdot N)$$

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 $S' \rightarrow \epsilon$, is also considered to be in the Chomsky Normal Form, and generates the original language, including the empty string.

Chomsky Normal Form

Strategy

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:

$$
\begin{array}{rcl}
\alpha & \rightarrow_{G'} & N_1[N_2 \cdots N_k], \\
[N_2 \cdots N_k] & \rightarrow_{G'} & N_2[N_3 \cdots N_k], \\
 & \cdots & \\
[N_{k-2}N_{k-1}N_k] & \rightarrow_{G'} & N_{k-2}[N_{k-1}N_k], \\
[N_{k-1}N_k] & \rightarrow_{G'} & N_{k-1}N_k
\end{array}
$$

Chomsky Normal Form

Example

As an example, consider $G = (\{S', X, Y, Z, a, b, c\}, \{a, b, c\}, P, S')$ with P equal to:

$$\begin{array}{rcl} \{S' & \rightarrow & XYZd, \\ X & \rightarrow & a, \\ Y & \rightarrow & b, \\ Z & \rightarrow & c, \} \end{array}$$

Chomsky Normal Form

Example

The CNF grammar G', equivalent to G, would then be the one with the following set of rules:

$$
\begin{array}{rcl}
\{S' & \rightarrow & X[YZd], \\
[YZd] & \rightarrow & Y[Zd], \\
[Zd] & \rightarrow & Z[d], \\
[d] & \rightarrow & d, \\
X & \rightarrow & a, \\
Y & \rightarrow & b, \\
Z & \rightarrow & c, \}
\end{array}
$$

Grammar Simplification

Chomsky Normal Form

Strategy for g_1:

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

Grammar Simplification

Chomsky Normal Form

From g_1 to g_2:

- For the new set of non-terminals:
  - One for every possibile (non-empty) sequence of terminal and non-terminal symbols of g_1: [...]
- For the new set of rules:
  - One for every terminal symbol t of g_1: $[t] \rightarrow t$;
  - One for every rule $X \rightarrow t$ of g_1: $[X] \rightarrow t$;
  - One for every rule $left \rightarrow s_1 s_2 \beta$ of g_1: $[left] \rightarrow [s1][s_2\beta]$;
  - One for every rule $[left] \rightarrow [s_1][s_2 s_3 \beta]$ of g_2: $[s_2 s_3 \beta] \rightarrow [s_2][s_3\beta]$
- For the new grammar:
  - The new set of non-terminals;
  - The new set of rules;
  - The mapped start symbol ($[S_1]$).

Chomsky Normal Form

```
Inductive non_terminal' (non_terminal terminal : Type): Type:=
| Lift_r: sf → non_terminal'.

Notation sf':= (list (non_terminal' + terminal)).
Notation term_lift:= ((terminal_lift non_terminal) terminal).

Definition symbol_lift (s: non_terminal + terminal)
: non_terminal' + terminal:=
match s with
| inr t ⇒ inr t
| inl n ⇒ inl (Lift_r [inl n])
end.
```

Chomsky Normal Form

```
Inductive g_cnf_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' → sf' → Prop:=
| Lift_cnf_t:
  ∀ t: terminal,
  ∀ left: non_terminal,
  ∀ s1 s2: sf,
  rules g left (s1++[inr t]++s2) →
  g_cnf_rules g (Lift_r [inr t]) [inr t]
```

Chomsky Normal Form

```
Lift_cnf_1:
 ∀ left: non_terminal,
 ∀ t: terminal,
 rules g left [inr t] →
 g_cnf_rules g (Lift_r [inl left]) [inr t]
```

Chomsky Normal Form

```
Lift_cnf_2:
 ∀ left: non_terminal,
 ∀ s1 s2: symbol,
 ∀ beta: sf,
 rules g left (s1 :: s2 :: beta) →
 g_cnf_rules g (Lift_r [inl left])
 [inl (Lift_r [s1]); inl (Lift_r (s2 :: beta))]
```

Chomsky Normal Form

```
Lift_cnf_3:
  ∀ left: sf,
  ∀ s1 s2 s3: symbol,
  ∀ beta: sf,
  g_cnf_rules g (Lift_r left)
  [inl (Lift_r [s1]); inl (Lift_r (s2 :: s3 :: beta))] →
  g_cnf_rules g (Lift_r (s2 :: s3 :: beta))
  [inl (Lift_r [s2]); inl (Lift_r (s3 :: beta))].
```

Chomsky Normal Form

```
Definition g_cnf
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: cfg non_terminal' terminal :=
  {| start_symbol:= Lift_r [inl (start_symbol g)];
     rules:= g_cnf_rules g;
     rules_finite:= g_cnf_finite g |}.
```

Grammar Simplification

Chomsky Normal Form

From g_1 to g_3:

- For the new set of non-terminals:
  - The same of g_2.
- For the new set of rules:
  - The same of g_2;
  - One extra rule: $[S_1] \rightarrow \epsilon$
- For the new grammar:
  - The new set of non-terminals;
  - The new set of rules;
  - The mapped start symbol ($[S_1]$).

Chomsky Normal Form

```
Inductive g_cnf'_rules
(non_terminal terminal : Type)
(g: cfg non_terminal terminal)
: non_terminal' → sf' → Prop:=
| Lift_cnf'_all:
   ∀ left: non_terminal',
   ∀ right: sf',
   g_cnf_rules g left right →
   g_cnf'_rules g left right
| Lift_cnf'_new:
   g_cnf'_rules g (start_symbol (g_cnf g)) [].
```

Chomsky Normal Form

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

Chomsky Normal Form

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

Chomsky Normal Form

```
Definition is_cnf_rule (left: non_terminal) (right: sf): Prop:=
(∃ s1 s2: non_terminal, right = [inl s1; inl s2]) ∨
(∃ t: terminal, right = [inr t]).

Definition is_cnf (g: cfg non_terminal terminal): Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right → is_cnf_rule left right.

Definition is_cnf_with_empty_rule (g: cfg non_terminal terminal):
Prop:=
∀ left: non_terminal,
∀ right: sf,
rules g left right →
(left = (start_symbol g) ∧ right = []) ∨
is_cnf_rule left right.
```

Chomsky Normal Form

- The proof of this theorem requires that the original grammar is first simplified according to the results discussed before;
- For the $\leftarrow$ part of g_equiv, the strategy adopted was to prove that for every rule $left \rightarrow right$ of (g), either $left \rightarrow right$ is a rule of g_cnf g or $left \Rightarrow^* right$ in g_cnf g.
- For the $\rightarrow$ part, that is, $(s_1 \Rightarrow^*_{g\_cnfg} s_2) \rightarrow (s_1 \Rightarrow^*_g s_2)$, it was enough to note that the sentential forms of g are embedded in the sentential forms of g_cnf g, specifically in the arguments of the constructor Lift_r of non_terminal'. Thus, a simple extraction mechanism allows the implication to be proved by induction on the structure of the sentential form s_1.

Chomsky Normal Form

Example

Using the previous example, suppose we have: $X[YZd] \Rightarrow^*_{g\_cnfg} abcd$, which would be represented in our formalization as:

```
derives (g_cnf g) [inl X] ++[inl (Lift_r ([inl Y; inl Z; inr d]))]
(map (·symbol_lift _ _) (map term_lift [inr a; inr b; inr c; inr d]))
```

The extraction mechanism, applied to this case, would yield:

```
derives g [inl X; inl Y; inl Z; inr d]
(map term_lift [inr a; inr b; inr c; inr d])
```

which is exactly the expected result $(XYZd \Rightarrow^*_g abcd)$.

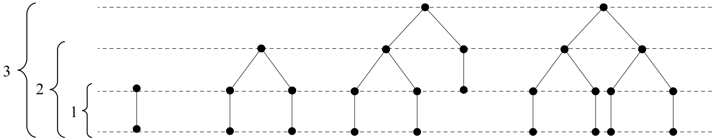
Generic Binary Trees Library

General results on binary trees and their relation to CNF grammars:

- 4,539 lines of Coq script, $\sim$18.9% of the total;
- 84 lemmas;
- Supports the formalization of the Pumping Lemma.
- Based on the definition of `btree`.

Generic Binary Trees Library

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



Generic Binary Trees Library

```
Definition broot (t: btree): non_terminal:=
match t with
| bnode_1 n t ⇒ n
| bnode_2 n t1 t2 ⇒ n
end.

Fixpoint bfrontier (t: btree): sentence:=
match t with
| bnode_1 n t ⇒ [t]
| bnode_2 n t1 t2 ⇒ bfrontier t1 ++bfrontier t2
end.

Fixpoint bheight (t: btree): nat:=
match t with
| bnode_1 n t ⇒ 1
| bnode_2 n t1 t2 ⇒ S (max (bheight t1) (bheight t2))
end.
```

Generic Binary Trees Library

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

Generic Binary Trees Library

```
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'.
```

Generic Binary Trees Library

```
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 ≠ []).
```

Generic Binary Trees Library

```
Inductive bpath (bt: btree): sf → Prop:=
| bp_1: ∀ n: non_terminal,
        ∀ t: terminal,
        bt = (bnode_1 n t) → bpath bt [inl n; inr t]
| bp_l: ∀ n: non_terminal,
        ∀ bt1 bt2: btree,
        ∀ p1: sf,
        bt = bnode_2 n bt1 bt2 → bpath bt1 p1 → bpath bt ((inl n) :: p1)
| bp_r: ∀ n: non_terminal,
        ∀ bt1 bt2: btree,
        ∀ p2: sf,
        bt = bnode_2 n bt1 bt2 → bpath bt2 p2 → bpath bt ((inl n) :: p2).
```

Generic Binary Trees Library

```
Lemma btree_ex_bpath:
∀ bt: btree,
∀ ntl: list non_terminal,
bheight bt ≥ length ntl + 1 →
bnts bt ntl →
∃ z: sf,
bpath bt z ∧
length z = bheight bt + 1 ∧
∃ u r: sf,
∃ t: terminal,
z = u ++r ++[inr t] ∧
length u ≥ 0 ∧
length r = length ntl + 1 ∧
(∀  s: symbol, In s (u ++r) → In s (map inl ntl)).
```

Generic Binary Trees Library

```
Inductive bnts (bt: btree) (ntl: list non_terminal): Prop:=
| bn_1: ∀ n: non_terminal,
        ∀ t: terminal,
        bt = (bnode_1 n t) → In n ntl → bnts bt ntl
| bn_2: ∀ n: non_terminal,
        ∀ bt1 bt2: btree,
        bt = bnode_2 n bt1 bt2 →
        In n ntl →
        bnts bt1 ntl →
        bnts bt2 ntl →
        bnts bt ntl.
```

Generic Binary Trees Library

```
Inductive bcode (bt: btree): list bool → Prop:=
| bcode_0: ∀ n: non_terminal,
           ∀ t: terminal,
           bt = (bnode_1 n t) → bcode bt []
| bcode_1: ∀ n: non_terminal,
           ∀ bt1 bt2: btree,
           ∀ c1: list bool,
           bt = bnode_2 n bt1 bt2 → bcode bt1 c1 → bcode bt (false :: c1)
| bcode_2: ∀ n: non_terminal,
           ∀ bt1 bt2: btree,
           ∀ c2: list bool,
           bt = bnode_2 n bt1 bt2 → bcode bt2 c2 → bcode bt (true :: c2).
```

Generic Binary Trees Library

```
Lemma bpath_ex_bcode:
∀ t: btree,
∀ p: sf,
bpath t p →
∃ c: list bool,
bcode t c ∧
bpath_bcode t p c.
```

Generic Binary Trees Library

```
Inductive bpath_bcode (bt: btree): sf → (list bool) → Prop:=
| bb_0: ∀ n: non_terminal, ∀ t: terminal,
        bt = (bnode_1 n t) → bpath_bcode bt [inl n; inr t] []
| bb_1: ∀ n: non_terminal, ∀ bt1 bt2: btree,
        ∀ c1: list bool, ∀ p1: sf,
        bt = (bnode_2 n bt1 bt2) →
        bpath bt1 p1 →
        bpath_bcode bt1 p1 c1 →
        bpath_bcode bt ((inl n) :: p1) (false :: c1)
| bb_2: ∀ n: non_terminal, ∀ bt1 bt2: btree,
        ∀ c2: list bool, ∀ p2: sf,
        bt = (bnode_2 n bt1 bt2) →
        bpath bt2 p2 →
        bpath_bcode bt2 p2 c2 →
        bpath_bcode bt ((inl n) :: p2) (true :: c2).
```

Generic Binary Trees Library

```
Lemma bcode_split:
∀ t: btree,
∀ p1 p2: sf,
∀ c: list bool,
bpath_bcode t (p1 ++p2) c →
length p1 > 0 →
length p2 > 1 →
bheight t = length p1 + length p2 − 1 →
∃ c1 c2: list bool,
c = c1 ++c2 ∧
length c1 = length p1 ∧
∃ t2: btree,
∃ x y: sentence,
bpath_bcode t2 p2 c2 ∧
btree_decompose t c1 = Some (x, t2, y) ∧
bheight t2 = length p2 − 1.
```

Generic Binary Trees Library

```
Fixpoint btree_decompose (bt: btree) (c: list bool):
option (sentence * btree * sentence):= ...

Fixpoint btree_subst (t1 t2: btree) (c: list bool):
option btree:= ...
```

Generic Binary Trees Library

```
Inductive btree_cnf (g: cfg non_terminal' terminal)
(bt: btree non_terminal' terminal): Prop:=
| bt_c1: ∀ n: non_terminal',
        ∀ t: terminal,
        rules g n [inr t] →
        bt = (bnode_1 n t) →
        btree_cnf g bt
| bt_c2: ∀ n n1 n2: non_terminal',
        ∀ bt1 bt2: btree _ _,
        rules g n [inl n1; inl n2] →
        btree_cnf g bt1 →
        broot bt1 = n1 →
        btree_cnf g bt2 →
        broot bt2 = n2 →
        bt = (bnode_2 n bt1 bt2) →
        btree_cnf g bt.
```

Generic Binary Trees Library

```
Lemma derives_g_cnf_equiv_btree:
∀ g: cfg non_terminal' terminal,
∀ n: non_terminal',
∀ s: sentence,
s ≠ [] →
(is_cnf g ∨ is_cnf_with_empty_rule g) →
start_symbol_not_in_rhs g →
derives g [inl n] (map term_lift' s) →
∃ t: btree non_terminal' terminal,
btree_cnf g t ∧
broot t = n ∧
bfrontier t = s.
```

Generic Binary Trees Library

```
Lemma btree_equiv_derives_g_cnf:
∀ g: cfg _ _,
∀ t: btree _ _,
btree_cnf g t →
derives g [inl (broot t)] (map inr (bfrontier t)).
```

Pumping Lemma

$$\forall\ \mathcal{L}, (\mathsf{cfl}\ \mathcal{L}) \to \exists\ n \mid$$

$$\forall\ \alpha, (\alpha \in \mathcal{L}) \land (|\alpha| \geq n) \to$$

$$\exists\ u, v, w, x, y \in \Sigma^* \mid (\alpha = uvwxy) \land (|vx| \geq 1) \land (|vwx| \leq n) \land$$

$$\forall\ i, uv^i wx^i y \in \mathcal{L}$$

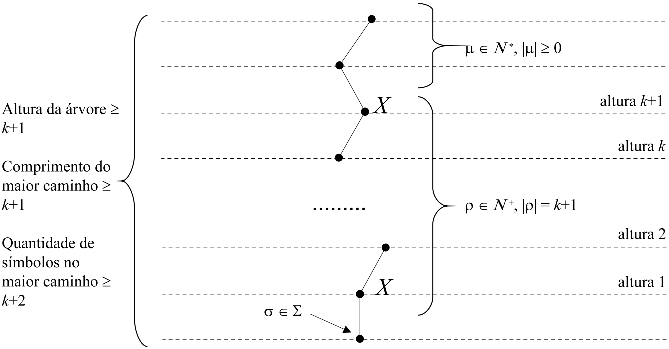
Pumping Lemma

Informal proof

1. Since $\mathcal{L}$ is declared to be a context-free language (predicate `cfl`), then there exists a context-free grammar G such that $L(G) = \mathcal{L}$;
2. Obtain G' such that G' is in Chomsky Normal Form and $L(G') = L(G)$;
3. Take n as 2^k, where k is the number of non-terminal symbols in G';
4. Consider an arbitrary sentence α such that $\alpha \in \mathcal{L}$ and $|\alpha| \geq n$;
5. Obtain a derivation tree t that represents the derivation of α in G';
6. Take a path that starts in the root of t and whose length is the height of t plus 1 (maximum length);
7. Then, the height of t should be greater or equal than $k+1$;
8. This means that the selected path has at least $k+2$ symbols, being at least $k+1$ non-terminals and one (the last) a terminal symbol;

Pumping Lemma

Informal proof



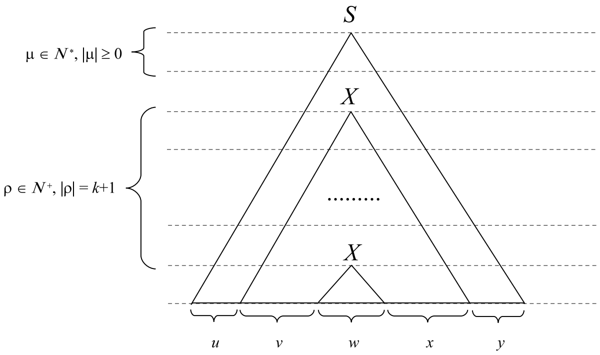
Pumping Lemma

Informal proof

9. Since G' has only k non-terminal symbols, this means that this path has at least one non-terminal symbol that appears at least two times in it;
10. Name the duplicated symbols n_1 and n_2 ($n_1 = n_2$) and the corresponding subtrees t_1 and t_2 (note that t_2 is a subtree of t_1 and t_1 is a subtree of t);
11. It is then possible to prove that the height of t_1 is greater than or equal to 2, and less than or equal to 2^k;
12. Also, that the height of t_2 is greater than or equal to 1 and less than or equal to 2^{k-1};
13. This implies that the frontier of t can be split into five parts: u, v, w, x, y, where w is the frontier of t_2 and vwx is the frontier of t_1;

Pumping Lemma

Informal proof



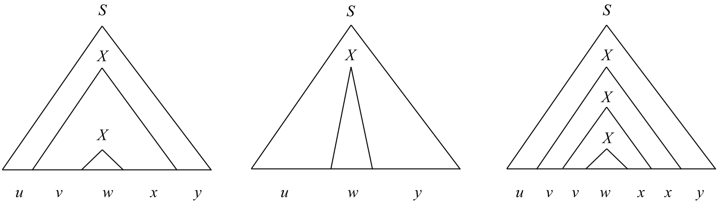
Pumping Lemma

Informal proof

14. As a consequence of the heights of the corresponding subtrees, it can be shown that $|vx| \geq 1$ and $|vwx| \leq n$;
15. If t_1 is removed from t, and t_2 is inserted in its place, then we have a new tree t^0 that represents the derivation of string $uv^0wx^0y = uwy$;
16. If, instead, t_1 is inserted in the place where t_2 lies originally, then we have a tree t^2 that represents the derivation of string uv^2wx^2y;
17. Repetition of the previous step generates all trees t^i that represent the derivation of the string uv^iwx^iy, $\forall i \geq 2$.

Pumping Lemma

Informal proof



Pumping Lemma

```
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).
```

Pumping Lemma

Formal proof

- Find a grammar G that generates the input language L (this is a direct consequence of the predicate `is_cfl` and corresponds to step 1;
- Obtain a CNF grammar G' that is equivalent to G (step 2), using previous results;
- G is substituted for G' and the value for n is defined as 2^k (step 3) where k is the length of the list of non-terminals of G' (which in turn is obtained from the predicate `rules_finite`);

Pumping Lemma

Formal proof

- An arbitrary sentence α of $L(G')$ that satisfies the required minimum length n is considered (step 4);
- Lemma `derives_g_cnf_equiv_btree` is then applied in order to obtain a `btree` t that represents the derivation of α in G' (step 5). Naturally we have to ensure that $\alpha \neq \epsilon$, which is true since by assumption $|\alpha| \geq 2^k$;
- 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 (steps 6 and 7). This is accomplished by means of the definition `bpath` and the lemma `btree_ex_bpath`.

Pumping Lemma

Formal proof

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 (steps 8, 9 and 10). 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.
```

Pumping Lemma

Formal proof

- Since a path is not unique in a tree, it is necessary to use some some other representation that can describe this path uniquely, which is done by the predicate `bcode` and the lemma `bpath_ex_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;

Pumping Lemma

Formal proof

- From this information it is then possible to extract most of the results needed to prove the goal (steps 11, 12, 13 and 14), except for the pumping condition. This has been 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 $v^i w x^i, i \geq 1$ and w, or simply $v^i w x^i, i \geq 0$.

Pumping Lemma

Formal proof

```
Lemma pumping_aux:
∀ g: cfg _ _,
∀ t1 t2: btree (non_terminal' non_terminal terminal) _,
∀ n: _, ∀ c1 c2: list bool, ∀ v x: sentence,
btree_decompose t1 c1 = Some (v, t2, x) →
btree_cnf g t1 → broot t1 = n →
bcode t1 (c1 ++c2) → c1 ≠ [] →
broot t2 = n → bcode t2 c2 →
(∀ i: nat,
 ∃ t': btree _ _,
 btree_cnf g t' ∧
 broot t' = n ∧
 btree_decompose t' (iter c1 i) = Some (iter v i, t2, iter x i) ∧
 bcode t' (iter c1 i ++c2) ∧
 get_nt_btree (iter c1 i) t' = Some n).
```

Pumping Lemma
Formal proof

- 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 (steps 15 and 16).
- 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` (step 17).

Pumping Lemma

Finite languages

If L is finite, then the PL is trivially true:

- Suppose L is finite;
- Let G in CNF such that $L = L(G)$;
- Let k be the number of non-terminals of G;
- We claim there is no $w \in L$ such that $|w| \geq 2^k$:
  - If there is, then the PL asserts that L is i nfinite, which contradicts the hypothesis.
- Since there is no $w \in L$ such that $|w| \geq 2^k$, then the PL is trivially true.

Summary

- 23,985 lines of Coq script spread in 18 libraries;
- Eight auxiliary libraries contain 11,781 lines of Coq script and correspond to almost half of the formalization (49.1%);
- Two of these auxiliary libraries (`cfg.v` and `trees.v`) sum, alone, 8,932 lines or more than one third (37.2%) of the total;
- 533 lemmas and theorems, 83 definitions and 40 inductive definitions among 1,067 declared names;
- Created and compiled with the Coq Proof Assistant, version 8.4pl4 (June 2014), using CoqIDE for Windows;
- Available for download at `https://github.com/mvmramos/v1`;
- Compiled with the following commands under Cygwin:
  - `coq_makefile *.v > _makefile`
  - `make -f _makefile`
  - `make -f _makefile html`

Summary

Main lemmas

- Library chomsky.v:
  - `g_cnf_exists`
- Library closure.v:
  - `l_clo_is_cfl`
  - `l_clo_correct`
  - `l_clo_correct_inv`
- Library concatenation.v:
  - `l_cat_is_cfl`
  - `l_cat_correct`
  - `l_cat_correct_inv`
- Library emptyrules.v:
  - `g_emp_correct`
  - `g_emp'_correct`
- Library inaccessible.v:
  - `g_acc_correct`

Summary

Main lemmas

- Library pumping:
  - pumping_lemma
  - pumping_lemma_v2
- Library simplification.v:
  - g_simpl_exists_v1
  - g_simpl_exists_v2
- Library union.v:
  - l_uni_is_cfl
  - l_uni_correct
  - l_uni_correct_inv
- Library unitrules.v:
  - g_unit_correct
- Library useless.v:
  - g_use_correct

Discussion

Lessons

One needs to have a previous hands-on experience in a real world formalization project of some complexity and size, preferably in a group willing to share its (supposedely) higher expertise and experience, before facing alone the challenges of a similar project.

Discussion

Lessons

Formalization projects (as with any other projects) should come in increasing size and complexity, allowing the person (or team) involved to be adequately prepared to cope with the new challenges.

Discussion

Lessons

Avoid formalizing a theory that you are not familiar with, unless you already master the proof assistant and have some experience with the formalization process. Otherwise, stick to a well-know theory and reduce the risks involved.

Discussion

Lessons

The formalization of any theory should start with the shortest, simpler and more independent lemmas and theorems, and proceed towards the largest and more complex ones, benefiting from previous results.

Discussion

Advices

- Make a deep review of the informal proof;
- Be sure of the statement to be proved;
- Use the cohesion and coupling principles;
- Choose a naming policy;
- Develop a writing style;
- Be prepared for lots of trial and error;
- Do not underestimate the importance of the inductive definitions;
- Get rid of useless code.

Discussion

This formalization

- Set versus Prop;
- Finiteness of the context-free grammar;
- Variants of inductive predicate definitions;
- Use of syntax trees in proofs;
- Statement and proof of the Pumping Lemma.

Discussion

Pumping Lemma

$$\forall\ \mathcal{L}, (\mathsf{cfl}\ \mathcal{L}) \rightarrow \exists\ n \mid$$
$$\forall\ \alpha, (\alpha \in \mathcal{L}) \wedge (|\alpha| \geq n) \rightarrow$$
$$\exists\ u, v, w, x, y \in \Sigma^* \mid (\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|vwx| \leq n) \wedge$$
$$\forall\ i, uv^i wx^i y \in \mathcal{L}$$

Discussion

Pumping Lemma

$$\forall\ \mathcal{L}, (\text{cfl } \mathcal{L}) \to \exists\ n \mid$$

$$\forall\ \alpha, (\alpha \in \mathcal{L}) \land (|\alpha| \geq n) \to$$

$$\exists\ u, v, w, x, y \in \Sigma^* \mid (\alpha = uvwxy) \land (|vx| \geq 1) \land (|uy| \geq 1) \land (|vwx| \leq n) \land$$

$$\forall\ i, uv^i wx^i y \in \mathcal{L}$$

Discussion

Pumping Lemma

A variant of the Pumping Lemma, using a smaller value of n, has also been proved. This result uses $n = 2^{k-1} + 1$ instead of $n = 2^k$ (k is the number of non-terminal symbols in the CNF grammar). Since the proof needs a binary tree of height at least $k + 1$ in order to proceed, and since trees of height i have as frontier strings of length maximum 2^{i-1}, it is possible to consider strings of length equal to or greater than $2^{k-1} + 1$ (and not only of length equal to or greater than 2^k) in order to have the corresponding binary tree with height equal to or higher than $k + 1$. This way, two slightly different proofs of the Pumping Lemma have been produced: one with $n = 2^k$ (`pumping_lemma`) and the other with $n = 2^{k-1} + 1$ (`pumping_lemma_v2`).

Discussion

Pumping Lemma

The statement of (pumping_lemma_v2) becomes:

$$\forall\ \mathcal{L}, (\text{cfl}\ \mathcal{L}) \rightarrow \exists\ n\,|$$

$$\forall\ \alpha, (\alpha \in \mathcal{L}) \wedge (|\alpha| \geq n) \rightarrow$$

$$\exists\ u, v, w, x, y \in \Sigma^*\,|\,(\alpha = uvwxy) \wedge (|vx| \geq 1) \wedge (|vwx| \leq (n-1) * 2) \wedge$$

$$\forall\ i, uv^iwx^iy \in \mathcal{L}$$

Discussion

Comparison

	Norrish & Barthwal	Firsov & Uustalu	Ramos
Proof assistant	HOL4	Agda	Coq
Closure	✓	×	✓
Simplification	✓	*empty and unit rules*	✓
CNF	✓	✓	✓
GNF	✓	×	×
PDA	✓	×	×
PL	✓	×	✓

Achievements

- A set of libraries that formalizes an important subset of context-free language theory;
- Expertise on interactive theorem proving.
  - Pioneering;
  - Reasoning about context-free language theory;
  - Learning and experimenting in an educational environment;
  - New projects and theories.

Contributions

Pioneering

- Bring formalization into an area which has relied so far mostly in informal arguments;
- First formalization of a coherent and complete subset of context-free language theory in the Coq proof assistant;
- Second formalization ever (in any proof assistant) of the Pumping Lemma for context-free languages;
- Second most comprehensive formalization of an important subset of the context-free language theory in any proof assistant.

Contributions

Reasoning about context-free language theory

- The present formalization can be very helpful to get insight into the nature and behaviour of the objects of context-free language theory, as well on the proofs of their properties;
- Also, when developing representations for new and similar devices, and proofs for new results of the theory;
- Finally, the formalization represents the guarantee that the proofs are correct and that the remaining errors in the informal demonstrations, if any, could finally and definitely be reviewed and corrected.

Contributions

Learning and experimenting in an educational environment

Teachers, students and professionals can use the formalization to learn and experiment with the objects and concepts of context-free language theory in a software laboratory, where further practical observations and developments could be done independently. Also, the material could be deployed as the basis for a course on the theoretical foundations of computing, exploring simultaneously or independently:

- Language theory;
- Logic;
- Proof theory;
- Type theory;
- Models of computation;
- Formal mathematics;
- Interactive theorem provers and Coq.

Contributions

Expertise and knowledge

- The essence of formalization comes into light with the accomplishment of this project;
- This enables the application of similar principles to the formalization of other theories, and allow for the multiplication of the knowledge among students and colleagues;
- Considering the growing interest in formalization in recent years, this project can be considered as a good technical preparation for dealing with the challenges of theory and computer program developments of the future.

Further Work

Various possibilities, considered in three different groups:

- New devices and results;
- Code extraction;
- General enhancements.

Further Work

New devices and results

- Pushdown automata, including: definition, equivalence of pushdown automata and context-free grammars; equivalence of empty stack and final state acceptance criteria; non-equivalence of the deterministic and the non-deterministic models;
- Elimination of left recursion in context-free grammars and Greibach Normal Form;
- Derivation trees, ambiguity and inherent ambiguity;
- Decidable problems for context-free languages (membership, emptyness and finiteness for example);
- Odgen's Lemma.

Further Work

Code extraction

- Add computational content;
- Extract certified programs for:
  - Closure properties;
  - Grammar simplification;
  - CNF.
- Certified parser generator.

Further Work

General enhancements

- Creating a naming policy that can be used rename the various objects and better identify their nature and intended use;
- Eliminating unnecessary definitions and lemmas;
- Making a better grouping of related objetcs and thus a better structuring of the whole formalization;
- Simplifying some proof scripts;

Further Work

General enhancements

- Commenting the scripts in order to provide a better understanding of their nature.
- Substitution of the classical logic proof of the pigeonhole principle for a constructive version;
- Rewriting of the contents of the trees.v library, in order to allow that all definitions and results be parametrized on any two types, one for the leafs and the other for the internal nodes of a `btree`;
- Experimenting and rewriting in SSReflect.