Formalization of Context-Free Language Theory

Marcus Vinícius Midena Ramos
(PhD student - UFPE, Recife, Brazil)

Ruy J. G. B. de Queiroz (Advisor - UFPE, Recife, Brazil)
Nelma Moreira (Supervisor - UP, Porto, Portugal)
José Carlos Bacelar Almeida (Supervisor - UM, Braga, Portugal)

Universidade do Porto
Departamento de Ciência de Computadores, Faculdade de Ciências
Porto, Portugal

July 10th, 2015

mvmr@cin.ufpe.br

(12 de setembro de 2015, 10:50)

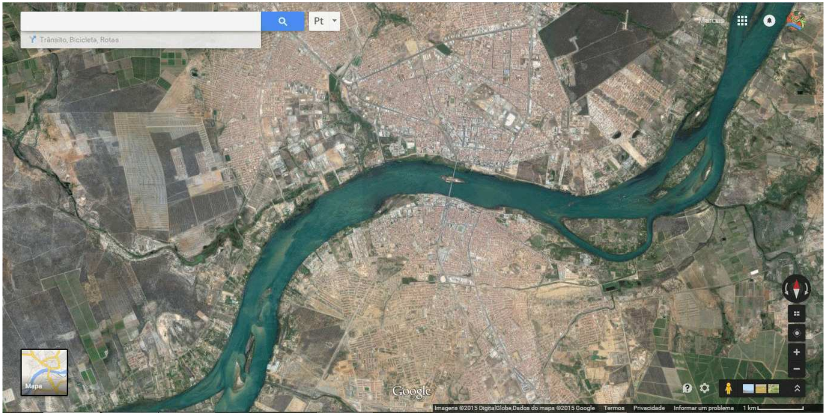
Profile

- Electronics Engineering at Universidade de São Paulo in 1982;
- M.Sc. in Digital Systems at Universidade de São Paulo in 1991;
- Teaching experience with programming languages, compilers, formal languages, automata theory and computation theory since 1991;
- Professional experience from 1983 to 1999 (software development, product management, marketing, retail, franchising, human resources, IT management);
- Current position at UNIVASF (Universidade Federal do Vale do São Francisco) in Petrolina-PE/Juazeiro-BA since April/2008;
- PhD student at UFPE (Universidade Federal de Pernambuco) since February/2011;
- Full dedication since July/2013.

Location

Fonte: IBGE, Diretoria de Geociências, Coordenação de Cartografia.
IBGE
VENEZUELA
COLÔMBIA
SURINAME
GUIANA
GUIANA FRANCESA
PERU
BOLÍVIA
CHILE
PARAGUAI
ARGENTINA
URUGUAI
OCEANO ATLÂNTICO
OCEANO PACÍFICO
BRASIL
RORAIMA
AMAPÁ
AMAZONAS
PARÁ
MARANHÃO
CEARÁ
RIO GRANDE DO NORTE
PARAÍBA
PERNAMBUCO
ALAGOAS
SERGIPE
PIAUÍ
BAHIA
TOCANTINS
ACRE
RONDÔNIA
MATO GROSSO
GOIÁS
BRASÍLIA
MINAS GERAIS
ESPÍRITO SANTO
MATO GROSSO DO SUL
SÃO PAULO
RIO DE JANEIRO
PARANÁ
SANTA CATARINA
RIO GRANDE DO SUL
Equador
Trópico de Capricórnio
Núcleos Urbanos
CAPITAL DE PAÍS
Capital de Estado
Sedes Municipais
Vias de Acesso
rodovias pavimentadas
terra
ferrovias
Limites
estadual
internacional
120 0 240 km
Projeção Policônica
Meridiano de Referência: -54° W. Gr
Paralelo de Referência: 0°
N

Location


Trânsito, Bicicleta, Rotas
Pt
Mapa
Google
Imagens ©2015 DigitalGlobe,Dados do mapa ©2015 Google
Termos
Privacidade
Informar um problema
1 km

So...

- Formalization?
- Context-Free Language Theory?
- Why?
- How?

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.

More on the next slides.

Summary

General Picture

- "Informal" mathematics:
  - Levels of abstraction may hide errors difficult to trace;
  - Non-uniform notation is also a problem.
- Formalization (*"computer encoded mathematics"*) is a clear trend towards theoretical development and theory representation;
- Computer-aided reasoning and use of proof assistants (interactive theorem provers);
- Mechanized checking of proofs (and programs), enabling:
  - Checking of every reasoning step against the inference rules of the underlying logic;
  - Uniform notation.
- Advantages:
  - Less effort and time;
  - Improved reliability.

Software Development

- Theorem proofs:
  - Informal;
  - Difficult to build;
  - Difficult to check.
- Computer programs:
  - Informal;
  - Difficult to build;
  - Difficult to test.
- Coincidence?

Software Development

- **NOT REALLY**, as theorem proving and software development have essentially the same nature;
- According to the Curry-Howard Isomorphism, to develop a program is the same as to prove a theorem, and vice-versa;
- Exploring this similarity his can be beneficial to both activities:
  - Reasoning can be brought into programming, and
  - Computational ideas can be used in theorem proving.
- How to improve both then?

Perspectives

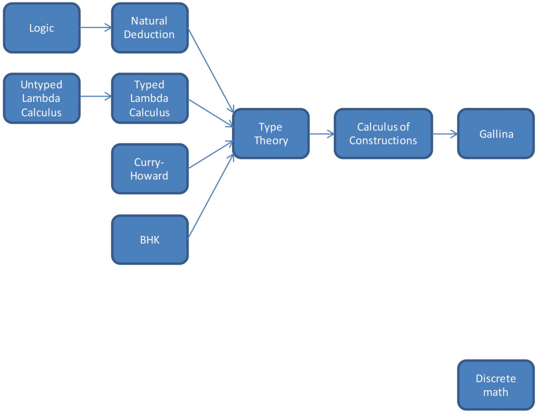
- Formalization (*"computer encoded mathematics"*) is the answer;
- Computer-aided reasoning;
- Use of proof assistants, also known as interactive theorem provers.

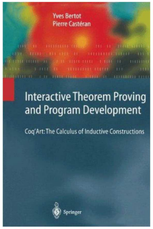
Background

Required before starting to use Coq:

- Natural Deduction;
- Untyped Lambda Calculus;
- Typed Lambda Calculus;
- Curry-Howard Isomorphism;
- Type Theory;
- Constructivism and BHK;
- Martin Löf's Intuitionistic Type Theory;
- Calculus of Constructions with Inductive Definitions.

Background

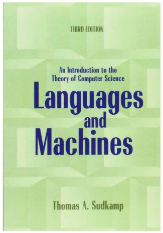



Yves Bertot
Pierre Castéran

Interactive Theorem Proving and Program Development

Coq'Art: The Calculus of Inductive Constructions

Springer


THIRD EDITION

An Introduction to the Theory of Computer Science

Languages and Machines

Thomas A. Sudkamp

Natural Deduction

- Calculus for theorem proving;
- Part of Proof Theory;
- Based in simple inference rules that resemble the rules of natural thinking;
- Each connective is associated to introduction and elimination rules;
- The proof of a theorem (proposition) is a structured sequence of inference rules that validate the conclusion, usually without depending on any hypothesis;
- The proof is represented as a tree;
- Gentzen (1935) and Prawitz (1965);
- Originally developed for propositional logic, was later extended for predicate logic.

Untyped Lambda Calculus

Formal system used for the representation of computations.

- Based on the definition and application of functions;
- Functions are treated as higher-order objects, as they can be passed as arguments and returned as values from other functions;
- Simplicity: only two constructs (“commands”);
- Allows the combination of basic functions in the creation of more complex functions;
- Even in the *pure* version (without constants), allows the representation of a broad range of datatypes, including booleans, natural numbers, integers etc, and operations on their values.
- Untyped and typed versions.

Untyped Lambda Calculus

- Alonzo Church, 1903-1995, United States;
- Invented the Lambda Calculus in the 1930s;
- Result of his investigations about the foundations of mathematics;
- Intended to formalize mathematics through the notion of functions, instead of the notion of sets;
- Although he did not succeed in this objective, his work was of great importante in other areas, specially in computer science.

Untyped Lambda Calculus

Mathematical model for:

- Theory, specification and implementation of programming languages, specially the functional ones.
- Program verification;
- Representation of computable functions;
- Computability theory;
- Proof theory.

Was used in the demonstration of the undecidability of various problems, even before the machine-based formalisms (e.g. Turing Machine).

Typed Lambda Calculus

- Created by Church to avoid the inconsistencies of the untyped version;
- Type tags are associated to lambda terms;
- Variables have base types $(x : \sigma)$;
- Abstractions and applications create new types accordingly;
- Types must match;
- Less powerful model of computation;
- Type systems for programming languages;
- Equality of terms is decidable;
- Strongly normalizing (all computations terminate);
- $(\lambda x.xx)(\lambda x.xx)$ and $(\lambda x.xxy)(\lambda x.xxy)$ are not terms of the typed lambda calculus.

Curry-Howard Isomorphism

Mathematics is all about:

- Reasoning;
- Computing.

For long time considered as separate areas; even today, ignored by many. Any relation there?

Curry-Howard Isomorphism

YES, according to the Curry-Howard Isomorphism.

- There is a direct relationship between reasoning (as expressed by first-order logic and natural deduction) and computing (as expressed by the typed lambda calculus);
- *Proofs-as-programs* or *Propositions-as-types* notions;
- First observed by (Haskell) Curry in 1934, later developed and extended by Curry in 1958 and William Howard in 1969;

Curry-Howard Isomorphism

- This has many important consequences as is the basis of modern software development and computer assisted theorem proofing:
  - Reasoning principles and techniques can be brought into software development;
  - Computing (idem) can be used in theorem proving.
- In the *simply typed lambda calculus*, the function operator $(\rightarrow)$ corresponds to the implication connective $(\Rightarrow)$; correspondences also exist for other operators.

Curry-Howard Isomorphism

General picture:

Proofs	Theorems
Programs	Types

Curry-Howard Isomorphism

Proofs & Theorems

First of all:

Proofs	$\Leftrightarrow$	Theorems
Programs		Types

Proofs & Theorems

Example

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow (b \Rightarrow c) \qquad a}{b \Rightarrow c}\,(\Rightarrow E) \qquad b}{c}\,(\Rightarrow E)}{a \Rightarrow c}\,(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

Theorem:

$$(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$$

Curry-Howard Isomorphism

Programs & Types

Also:

Proofs		Theorems
Programs	$\Leftrightarrow$	Types

Programs & Types
Example

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to (b \to c) \qquad z : a}{xz : b \to c}\,(\to E) \qquad y : b}{xzy : c}\,(\to E)}{\lambda z^a.xzy : (a \to c)}\,(\to I)}{\lambda y^b.\lambda z^a.xzy : (b \to (a \to c))}\,(\to I)}{\lambda x^{a \to (b \to c)}.\lambda y^b.\lambda z^a.xzy : (a \to (b \to c)) \to (b \to (a \to c))}\,(\to I)$$

Type:

$$(a \to (b \to c)) \to (b \to (a \to c))$$

Curry-Howard Isomorphism

Theorems & Types

Next, it is easy to observe that:

$$\begin{array}{cc} \text{Proofs} & \text{Theorems} \\ & \Updownarrow \\ \text{Programs} & \text{Types} \end{array}$$

Types (specifications) and Theorems (propositions) share the same syntactic structure.

Theorems & Types

Example

Type or theorem?

Type:

$$(a \rightarrow (b \rightarrow c)) \rightarrow (b \rightarrow (a \rightarrow c))$$

Theorem:

$$(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))$$

Curry-Howard Isomorphism

The Isomorphism

Logic	Typed lambda calculus
$\Rightarrow$ (implication)	$\rightarrow$ (function type)
$\wedge$ (and)	$\times$ (product type)
$\vee$ (or)	$+$ (sum type)
$\forall$ (forall)	Π (pi type)
$\exists$ (exists)	Σ (sigma type)
$\top$	unit type
$\bot$	bottom type

Curry-Howard Isomorphism

Proofs & Programs

Finally, the isomorphism extends to:

$$\begin{tikzcd}
\text{Proofs} \arrow[d, Leftrightarrow] & \text{Theorems} \\
\text{Programs} & \text{Types}
\end{tikzcd}$$

One can be obtained directly from the other:

- From Proof to Program: by adding the terms with the corresponding types;
- From Program to Proof: by eliminating the terms and keeping only the types.

Proofs & Programs

Example

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \Rightarrow (b \Rightarrow c) \qquad a}{b \Rightarrow c}\,(\Rightarrow E) \qquad b}{c}\,(\Rightarrow E)}{a \Rightarrow c}\,(\Rightarrow I)}{b \Rightarrow (a \Rightarrow c)}\,(\Rightarrow I)}{(a \Rightarrow (b \Rightarrow c)) \Rightarrow (b \Rightarrow (a \Rightarrow c))}\,(\Rightarrow I)$$

Program:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : a \to (b \to c) \qquad z : a}{xz : b \to c}\,(\to E) \qquad y : b}{xzy : c}\,(\to E)}{\lambda z^a.xzy : (a \to c)}\,(\to I)}{\lambda y^b.\lambda z^a.xzy : (b \to (a \to c))}\,(\to I)}{\lambda x^{a \to (b \to c)}.\lambda y^b.\lambda z^a.xzy : (a \to (b \to c)) \to (b \to (a \to c))}\,(\to I)$$

Curry-Howard Isomorphism

Consequences

- To build a program that satisfies a specification (type):
  - Interpret the specification as a theorem (proposition);
  - Build a proof tree for this theorem;
  - Add terms with the corresponding types.
- To build a proof of a theorem:
  - Interpret the theorem as a specification;
  - Build a program that meets the specification;
  - Remove the terms from the derivation tree.

Curry-Howard Isomorphism

Consequences

Summary:

- To build a program is the same as to build a proof;
- To build a proof is the same as to build a program;
- To verify a program is the same as to verify a proof;
- Both verifications can be done via simple and efficient type checking algorithms.

Type Theory

A *Type Theory* is a theory that allows one to assign types to variables and construct complex type expressions. Then, lambda expressions can be derived to meet a certain type, or the type of an existing expression can be obained by following the theory's inference rules.

- Originally developed by Bertrand Russell in the 1910s as a tentative of fixing the paradoxes of set theory ("is the set composed of all sets that are not members of themselves a member of itself?");
- The *Simply Typed Lambda Calculus* is a type theory with a single operator ($\rightarrow$) and was developed by Church in the 1940s as a tentative of fixing the inconsistencies of the untyped lambda calculus;
- Since then it has been extended with many new operators;
- Various different type theories exist nowadays;
- *Martin Löf's Intuitionistic Type Theory* is one of the most important.

Constructivism and BHK

- Every true proposition must be accompanied by a proof of the validity of the statement; the proof must explain how to build the object that validates the argument (proposition);
- Proposed by Brouwer, Heyting and Kolgomorov, the BHK interpretation leaves behind the idea of the truth values of Tarski;
- $x : \sigma$ is interpreted as x *is a proof of* σ;

Constructivism and BHK

A proof of...

- $a \Rightarrow b$ is a mapping that creates a proof of b from a proof of a (*function*);
- $a \wedge b$ is a proof of a together with a proof of b (*pair*);
- $a \vee b$ is a proof of a or a proof of b together with an indication of the source (*pair*);
- $\forall x : A.P(x)$ is a mapping that creates a proof of $P(t)$ for every t in A (*function*);
- $\exists x : A.P(x)$ is an object t in A together with a proof of $P(t)$ (*pair*).

Constructivism and BHK

- Constructivism does not use the Law of the Excluded Middle $(p \vee \neg p)$ or any of its equivalents, that belong to classic logic only:
  - Double negation $\neg(\neg p) \Rightarrow p$;
  - Proof by contradiction $(\neg a \Rightarrow b) \wedge (\neg a \Rightarrow \neg b) \Rightarrow a$;
  - Peirce's Law $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$.
- A constructive proof is said to have *computational content*, as it is possible to "construct" the object that validates the proposition (the proof is a recipe for building this object);
- A constructive proof enables (computer) code *extraction* from proofs, thus the interest for it in computer science.

Constructivism

According to Troelstra:

> *"... the insistence that mathematical objects are to be constructed (mental constructions) or computed; thus theorems asserting the existence of certain objects should by their proofs give us the means of constructing objects whose existence is being asserted."*

Martin Löf's Intuitionistic Type Theory

A constructive type teory based on:

1. First-order logic to represent types and propositions;
2. Typed lambda calculus to represent programs and theorems.

and structured around the Curry-Howard Isomorphism.

- It is a powerful theory for sotware development and interactive theorem proving;
- Also used as a theory for the foundations of mathematics.

Calculus of Constructions with Inductive Definitions

A richly typed lambda calculus extended with inductive definitions.

- *Calculus of Constructions* developed by Thierry Coquand;
- Constructive type theory;
- Later extended with inductive definitions;
- Used as the mathematical language of the Coq proof assistant

Calculus of Constructions

- All logical operators ($\rightarrow, \wedge, \vee, \neg$ and $\exists$) are defined in terms of the universal quantifier ($\forall$), using “dependent types”;
- Types and programs (terms) have the same syntactical structure;
- Types have a type themselves (called “Sort”);
- Base sorts are “$Prop$” (the type of propositions) and “Set” (the type of small sets);
- $Prop : Type(1)$, $Set : Type(1)$, $Type(i) : Type(i+1), i \geq 1$;
- $S = \{Prop, Set, Type(i) \mid i \geq 1\}$ is the set of sorts;
- Various datatypes can be defined (naturals, booleans etc);
- Set of typing and conversion rules.

Inductive Definitions

Finite definition of infinite sets.

- “Constructors” define the elements of a set;
- Constructors can be base elements of the set;
- Constructors can be a functions that takes set elements and return new set elements.
- Manipulation is done via “pattern matching” over the inductive definitions.

Inductive Definitions

Booleans

```
{false,true}

Inductive boolean:
 | false: boolean
 | true: boolean.

Variable x: boolean.
Definition f: boolean:= false.
```

Inductive Definitions

Naturals

```
{0, 1, 2, 3, ...} = {O, SO, SSO, SSSO, ...}

Inductive nat:=
 | O: nat
 | S: nat->nat.

Variable y: nat.
Definition zero: nat:= O.
Definition one: nat:= S O.
Definition two: nat:= S one.
```

Inductive Definitions

String sets

```
Inductive ss:=
 | ss_empty: ss
 | ss_item: string->ss
 | ss_build: string->ss->ss.

Variable z: ss.
Definition ss0: ss:= ss_empty.
Definition ss1: ss:= ss_item "abc".
Definition ss2: ss:= ss_build "def" (ss_item "abc").
Definition ss3: ss:= ss_build "ghi" (
                     ss_build "def" (ss_item "abc")).
```

Inductive Definitions

Pattern matching

Booleans:

```
Definition negb (x: bool): bool:=
match x with
 | false => true
 | true => false
end.
```

Inductive Definitions

Pattern matching

Naturals:

```
Definition sub (n: nat): nat :=
match n with
 | O => O
 | S m => m
end.

Fixpoint nat_equal (n1 n2: nat): bool :=
match n1, n2 with
 | O, O => true
 | S m, S n => nat_equal m n
 | O, S n => false
 | S m, O => false
end.
```

Characteristics

- Software tools that assist the user in theorem proving and program development;
- First initiative dates from 1967 (Automath, De Bruijn);
- Many provers are available today (Coq, Agda, Mizar, HOL, Isabelle, Matita, Nuprl...);
- Interactive;
- Graphical interface;
- Proof/program checking;
- Proof/program construction.

Usage

1. The user writes a statement (proposition) or a type expression (specification) in the language of the underlying logic;
2. He constructs (directly or indirectly):
   - A proof of the theorem;
   - A program (term) that complies to the specification.
3. Directly: the proof/term is written in the formal language accepted by the assistant;
4. Indirectly: the proof/term is built with the assistance of an interactive "tactics" language:
5. In either case, the assistant checks that the proof/term complies to the theorem/specification.

Check and/or construct

- Proof assistants check that proofs/terms are correctly constructed;
- This is done via simple type-checking algorithms;
- Automated proof/term construction might exist is some cases, to some extent, but this is not the main focus;
- Thus the name "proof assistant";
- Automated theorem proofing might be pursued, due to "proof irrelevance";
- Automated program development, on the other hand, is unrealistic.

Main benefits

- Proofs and programs can be mechanically checked, saving time and effort and increasing reliability;
- Checking is efficient;
- Results can be easily stored and retrieved for use in different contexts;
- Tactics help the user to construct proofs/programs;
- User gets deeper insight into the nature of his proofs/programs, allowing further improvement.

Applications

- Formalization and verification of theorems and whole theories;
- Verification of computer programs;
- Correct software development;
- Automatic review of large and complex proofs submitted to journals;
- Verification of hardware and software components.

Drawbacks

- Failures in infrastructure may decrease confidence in the results (proof assistant code, language processors, operating system, hardware etc);
- Size of formal proofs;
- Reduced numer of people using proof assistants;
- Slowly increasing learning curve;
- Resemblance of computer code keeps pure mathematicians uninterested.

Overview

- Developed by Huet/Coquand at INRIA in 1984;
- First version released in 1989, inductive types were added in 1991;
- Continuous development and increasing usage since then;
- The underlying logic is the Calculus of Constructions with Inductive Definitions;
- It is implemented by a typed functional programming with a higher order logic language called *Gallina*;
- Interaction with the user is via a command language called *Vernacular*;
- Constructive logic with large standard library and user contributions base;
- Extensible environment;
- All resources freely available from `http://coq.inria.fr/`.

User session

The proof can be constructed directly ou indirectly.
In the indirect case,

- The initial goal is the theorem/specification supplied by the user;
- The environment and the context are initially empty;
- The application of a “tactics” substitutes the current goal for zero ou more subgoals;
- The context changes and might incorporate new hypotheses;
- The process is repeated for each subgoal, until no subgoal remains;
- The proof/term is constructed from the sequence of tactics used.

Tactics usage

- Inference rules map premises to conclusions;
- *Forward reasoning* is the process of moving from premises to conclusions;
  - Example: from a proof of a and a proof of b one can prove $a \wedge b$;
- *Backward reasoning* is the process of moving from conclusions to premises;
  - Example: to prove $a \wedge b$ one has to prove a and also prove b;
- Coq uses *backward reasoning*;
- They are implemented by “tactics”;
- A tactic reduces a goal to its subgoals, if any, or simply proves the goal.

Certified software development

1. Write the specifications as type expressions;
2. Interpret them as theorems;
3. Build the proofs;
4. Let the proof assistant check them;
5. Convert them to computer programs using the code extraction facility.

Introduction

- Great and increasing interest in formal proof and program development over the recent years;
- Main areas include:
  - Programming language semantics formalization;
  - Mathematics formalization;
  - Education.
- Important projects in both academy and industry;
- Top 100 theorems (91% formalized as of July/2015);
- Check `http://www.cs.ru.nl/~freek/100/`;
- One way road.

Four Color Theorem

- Stated in 1852, proved in 1976 and again in 1995;
- The two proofs used computers to a some extent, but were not fully mechanized;
- In 2005, Georges Gonthier (Microsoft Research) and Benjamin Werner (INRIA) produced a proof script that was fully checked by a machine;
- Milestone in the history of computer assisted proofing;
- 60,000 lines of Coq script and 2,500 lemmas;
- Byproducts.

Four Color Theorem

"Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction using mathematics to help programming computers."

Georges Gonthier

Odd Order Theorem

- Also known as the Feit-Thomson Theorem;
- Important to mathematics (in the classification of finite groups) and cryptography;
- Conjectured in 1911, proved in 1963;
- Formally proved by a team led by Georges Gonthier in 2012;
- Six years with full-time dedication;
- Huge achievement in the history of computer assisted proofing;
- 150,000 lines of Coq script and 13,000 theorems;

Compiler Certification

- CompCert, a fully verified compiler for a large subset of C that generates PowerPC code;
- Object code is certified to comply with the source code in all cases;
- Applications in avionics and critical software systems;
- Not only checked, but also developed in Coq;
- Three persons-years over a five yers period;
- 42,000 lines of Coq code.

Microkernel Certification

- Critical component of operating systems, runs in privileged mode;
- Harder to test in all situations;
- seL4, written in C (10,000 lines), was fully checked in HOL/Isabelle;
- No crash, no execution of any unsafe operation in any situation;
- Proof is 200,000 lines long;
- 11 persons-years, can go down to 8, 100% overhead over a non-certified project.

Digital Security Certification

- JavaCard smart card platform;
- Personal data such as banking, credit card, health etc;
- Multiple applications by different companies;
- Confidence and integrity must be assured;
- Formalization of the behaviour and the properties of its components;
- Complete certification, highest level achieved;
- INRIA, Schlumberger and Gemalto.

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.

Origins

- Experience in teaching language and automata theory;
- Book *Linguagens Formais* published in 2009 (with J.J. Neto and I.S. Vega);
- Algorithms were used instead of demonstrations for most theorems;
- Interest in formalization after studying logic, lambda calculus, type theory and Coq;
- Desire to follow the lines of the book and formalize its contents;
- 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.

Objectives

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

1. Closure properties:
   - 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.

Six main theorems.

Current Status

- 600+ lemmas and theorems, 20+ libraries, 25.000+ lines of scripts;
- 2 year effort;
- Representation of all relevant objects of the universe of discourse using inductive definitions for types and propositions:
  - Terminal and non-terminal symbol sets;
  - Sentence and sentential forms;
  - Rules;
  - Context-free grammars;
  - Derivations;
  - Trees.
- Declarative style;
  - Closer to textbook definitions;
  - More abstract to deal with;
  - Does not allow for the extraction of certified programs.
- Currently finishing the formalization of the 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 → sf → 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: sentence,
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):= 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.

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

Generic CFG Library

General purpose lemmas:

- $\forall g, s_1, s_2, s_3,\ (s_1 \Rightarrow_g^* s_2) \rightarrow (s_2 \Rightarrow_g^* s_3) \rightarrow (s_1 \Rightarrow_g^* s_3)$
- $\forall g, s_1, s_2, s, s',\ (s_1 \Rightarrow_g^* s_2) \rightarrow (s \cdot s_1 \cdot s' \Rightarrow_g^* s \cdot s_2 \cdot s')$
- $\forall g, s_1, s_2, s_3, s_4,\ (s_1 \Rightarrow_g^* s_2) \rightarrow (s_3 \Rightarrow_g^* s_4) \rightarrow (s_1 \cdot s_3 \Rightarrow_g^* s_2 \cdot s_4)$
- $\forall g, s_1, s_2, s_3,\ (s_1 \cdot s_2 \Rightarrow_g^* s_3) \rightarrow \exists s_1', s_2' \,|\, (s_3 = s_1' \cdot s_2') \wedge (s_1 \Rightarrow_g^* s_1') \wedge (s_2 \Rightarrow_g^* s_2')$
- $\forall g, s_1, s_2, n, w,\ (s_1 \cdot n \cdot s_2 \Rightarrow_g^* w) \rightarrow \exists w' \,|\, (n \Rightarrow_g^* w')$
- $\forall g, n, w,\ (n \Rightarrow_g^* w) \rightarrow (n \rightarrow_g w) \vee (\exists right \,|\, n \rightarrow_g right \wedge right \Rightarrow_g^* w)$
- $\forall g_1, g_2, g_3,\ (g_1 \equiv g_2) \wedge (g_2 \equiv g_3) \rightarrow (g_1 \equiv g_3)$

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

Context-free languages are closed under union, concatenation and Kleene star.

- Define union, concatenation and Kleene star operations;
- Prove that the resulting languages are context-free;
- Prove that the resulting languages contain exactly the expected strings.

First with grammars, then with languages.

Union
Definitions

Construct g_3 such that $L(g_3) = L(g_1) \cup L(g_2)$:

```
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.

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 |}.
```

Union

Definitions

```
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))]
```

Union
Definitions

```
| 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).
```

Union
Correctness

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

```
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)).
```

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)).
```

Union

Proofs Outline

- The correctness proof is straightforward and was obtained directly from the definition of the corresponding grammars;
- The completeness proofs is more complicated, and was constructed by induction on the inductive definition *derives*, with extensive case analysis;
- Equivalent statements were proved using context-free languages instead of context-free grammars:

```
Inductive l_uni (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.

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

Concatenation

Definitions

Construct g_3 such that $L(g_3) = L(g_1) \cdot L(g_2)$:

```
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.

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 |}.
```

Concatenation

Definitions

```
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))])
```

Concatenation

Definitions

```
| 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).
```

Concatenation

Correctness

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

```
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)).
```

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.
```

Concatenation
Proofs Outline

- Both the correctness and the completeness proofs are constructed by induction on the inductive definition *derives*, with extensive case analysis.
- Equivalent statements were proved using context-free languages instead of context-free grammars:

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

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

Kleene Star

Definitions

Construct g_2 such that $L(g_2) = (L(g_1))^*$:

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

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 |}.
```

Kleene Star

Definitions

```
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).
```

Kleene Star

Correctness

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

```
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)).
```

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'').
```

Kleene Star
Proofs Outline

- The correctness proof is straightforward and are obtained directly from the definition of the corresponding grammars;
- The completeness proofs is more complicated, and are constructed by induction on the inductive definition *derives*, with extensive case analysis.
- Equivalent statements were proved using context-free languages instead of context-free grammars:

```
Inductive l_clo (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).

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

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': 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 $P = \{S' \to X, X \to aX, X \to Y, Y \to XbY, Y \to Z, Z \to c\}$. The previous definitions assert that P' has the following rules:

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

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

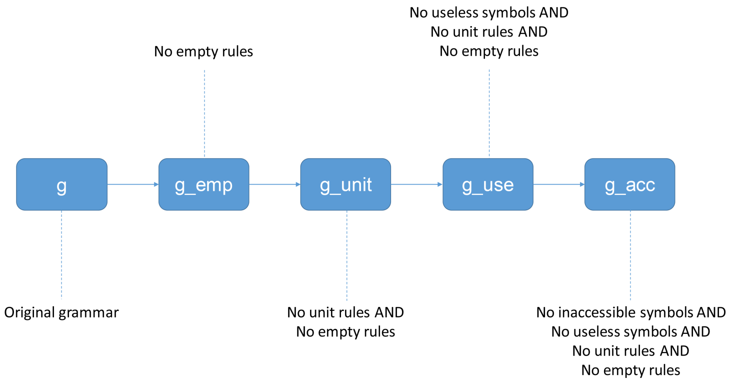
All in the Same Grammar

```
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:



Concept

$$\forall G = (V, \Sigma, P, S),$$
$$\exists G' = (V', \Sigma, P', S') \mid$$
$$L(G) = L(G') \wedge$$
$$\forall(\alpha \to \beta) \in P', (\beta \in \Sigma) \vee (\beta \in N \cdot N)$$

Important for:

- Decidability of the membership problem ;
- Some parsing algorithms (e.g. CYK);
- Pumping Lemma.

Example

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

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

Example

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

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

Definitions

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

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 |}.
```

Definitions

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

Definitions

```
| 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))]
| 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))].
```

Definitions

```
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 |}.
```

Definitions

```
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)) [].
```

Correctness

```
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') ∧
start_symbol_not_in_rhs g'.
```

Proof Outline

The proof of this theorem requires, among other things, that the original grammar is first simplified according to the results discussed in the previous section.

- For the $\leftarrow$ part of `g_equiv`, the strategy adopted is 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 is 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.

Concept

$$\forall L, \text{context-free } (L) \rightarrow$$
$$\exists n \mid \forall s,\ (s \in L) \wedge (|s| \geq n) \rightarrow$$
$$(s = uvwxy) \wedge (|vx| > 0) \wedge (|vwx| \leq n) \wedge (\forall i, uv^i wx^i y \in L)$$

- A property of all context-free languages;
- States that from certain strings of the language it is possible to generate an infinite number of other strings that also belong to the language;
- Is used to prove that certain languages are not context-free;
- Explores the finiteness of the number of non-terminals, in particular in the CNF grammar, and makes extensive use of (binary) trees.

Computers and mathematics

- Practitioners base is still small;
- Learning curve grows (very) slowly;
- Advantages of formalization are immense;
- Important industrial projects;
- Important theoretical works;
- Disadvantages are being gradually eliminated;
- The trend is clearly set.

This Formalization

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

Plans for the Future

- Obtain the degree (deadline Feb/2016);
- Promote Coq and mathematical formalization through speechs, workshops and other academic activities;
- Continue the formalization:
  - SSRreflect;
  - Code extraction and certified algorithms;
  - Pushdown automata and other results of CFLs.
- Keep learning Coq!

Computers and mathematics

- Not easy, but very rewarding;
- Hope you have enjoyed;
- Ask me if you want references;
- Write me if you have questions or suggestions;
- Let me know you if plan to work in this area.

Thank you!