LSFA 2014

Formalization of closure properties for context-free grammars

Marcus Vinícius Midena Ramos

UFPE/UNIVASF

September 09, 2014

mvmr@cin.ufpe.br
marcus.ramos@univasf.edu.br

Profile

- Electronics Engineering at Universidade de São Paulo in 1982;
- Master 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;
- Current position at Universidade Federal do Vale do São Francisco (Petrolina-PE/Juazeiro-BA) since 2008;
- PhD in Computer Science student at Universidade Federal de Pernambuco since 2011.

Background

- 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 over recent years, usually with a restricted focus.

Current work

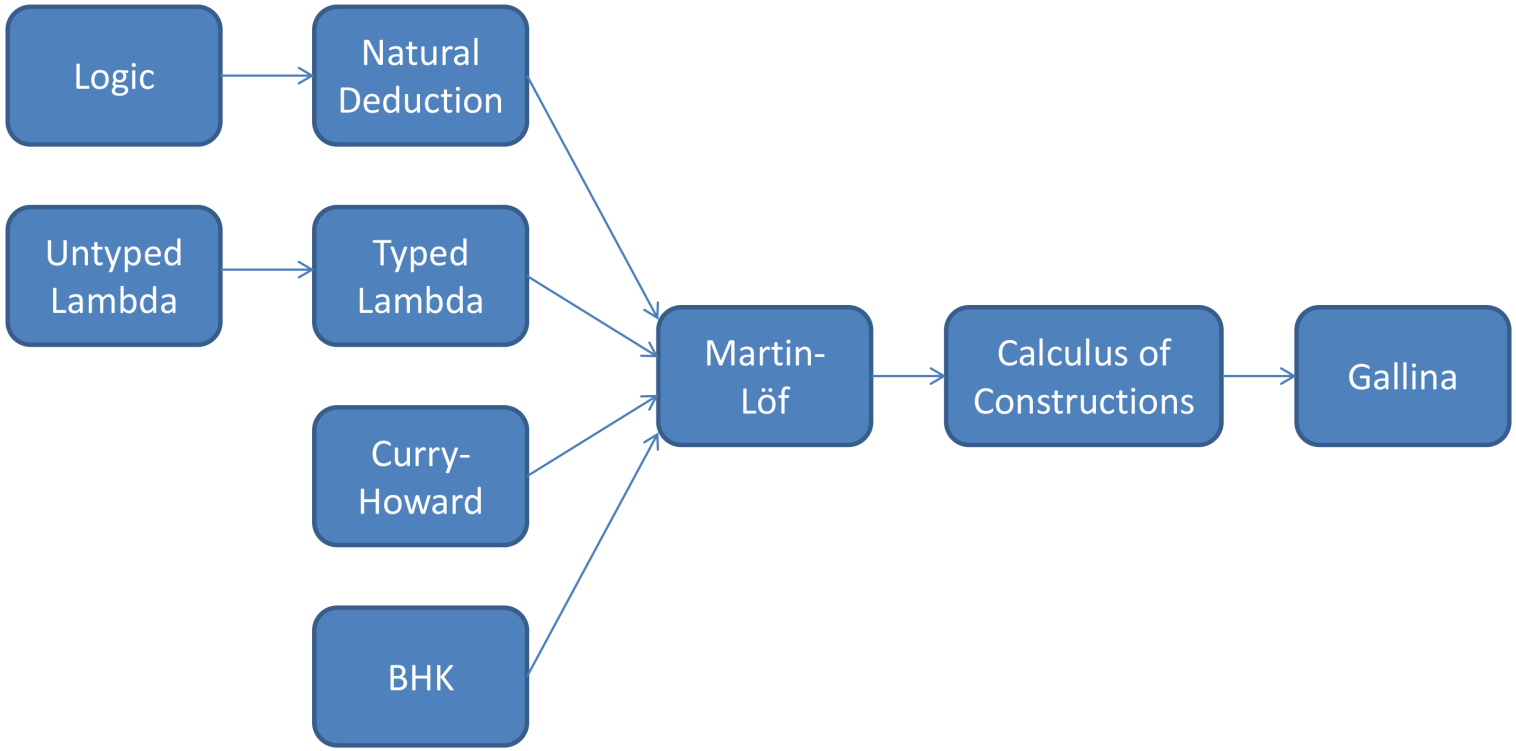
- Formal mathematics;
- Interactive theorem proving;
- Context-free language theory formalization;
- Proof assistants.
- Coq.

Summary

History and practice of theorem proving

- Theorem proofs:
  - Informal;
  - Difficult to build;
  - Difficult to check.
- Formalization (*"computer encoded mathematics"*) is an alternative;
- Computer-aided reasoning;
- Use of proof assistants, also known as interactive theorem provers.

Formal mathematics



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 theorem (proposition) in first-order logic or a type expression (specification);
2. The 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 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.

General

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);
- $Prop : Type(1)$, $Set : Type(1)$, $Type(i) : Type(i+1), i \geq 1$;
- $S = \{Prop, Set, Type(i) | i \geq 1\}$ 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.

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 one 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.

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 (88% formalized);
- The trend is clearly set.

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.

Scope

Context-free language theory:

- Relevant to computer language definition and processing;
- Context-free languages are represented by context-free grammars;
- Focus on results directly related to context-free grammars;
- Stack-automata will be considered in the future.

Motivation

- The large amount of formalization already existing for regular language theory;
- The apparent absence of a similar formalization effort for context-free language theory, at least in the Coq proof assistant and
- The high interest in context-free language theory formalization as a consequence of its practical importance in computer technology (e.g. correctness of language processing software).

Objectives

Prove theorems about context-free language theory:

1. Closure properties (union, concatenation and Kleene star);
2. Simplification of context-free grammars;
3. Normal forms for context-free grammars;
4. Pumping lemma for context-free languages.

Using Coq proof assistant + CoqIDE.

This presentation

Closure of context-free languages under union, concatenation and Kleene star:

1. Representation of context-free grammars and string derivations;
2. Representation of closure grammars;
3. Check that context-free languages are closed under the operations;
4. Proof that the proposed closure grammars generate the desired languages:
   - All inputs produce correct outputs (*direct operation*);
   - All outputs are correct results from some inputs (*inverse operation*).
5. A pair of theorems for each operation (total of six).

Grammar

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

- Σ is the set of terminal symbols;
- $N = V - \Sigma$ 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: Type:= {
non_terminal: Type;
terminal: Type;
start_symbol: non_terminal;
sf:= list (non_terminal + terminal);
rules: non_terminal -> sf -> Prop }.
```

Example

- $G = (\{S, X, Y, a, b, c\}, \{a, b, c\}, \{S \to aS, S \to b\}, S)$

```
Inductive nt1: Type:= S | X | Y.
Inductive  t1: Type:= a | b | c.
Inductive rs1: nt1 -> list (nt1+t1) -> Prop:=
  r11: rs1 S [inr a;inl S]
| r12: rs1 S [inr b].
Definition g1:= {|
non_terminal:= nt1;
terminal:= t1;
start_symbol:= S;
rules:= rs1 |}.
```

Derivation

- $s \Rightarrow^* s$
- $s_1 \Rightarrow^* s_2\, l\, s_3$ and $l \rightarrow r$ implies $s_1 \Rightarrow^* s_2\, r\, s_3$

```
Inductive derives (g: cfg): sf g -> sf g -> Prop :=
| derives_refl: forall s: sf g,
                derives g s s
| derives_step: forall s1 s2 s3: sf g,
                forall left: non_terminal g,
                forall right: sf g,
                derives g s1 (s2 ++ inl left :: s3)->
                rules g left right ->
                derives g s1 (s2 ++ right ++ s3).
```

Sentence generation

- $S \Rightarrow^* s$

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

Example

```
Lemma gen_g1_aab: generates g1 ([inr a]++[inr a]++[inr b]).
Proof.
unfold generates.
rewrite <- app_nil_l. rewrite <- app_nil_r.
repeat rewrite <- app_assoc.
rewrite <- (app_nil_l [inl (start_symbol g1)]).
rewrite <- (app_nil_r [inl (start_symbol g1)]).
apply derives_step with (left:=S)(s2:=[]++[inr a]++[inr a]).
rewrite <- app_nil_r.
repeat rewrite <- app_assoc.
apply derives_step with (left:=S)(right:=[inr a]++[inl S]).
rewrite app_nil_l. rewrite app_nil_r.
apply derives_direct.
apply r11. apply r11. apply r12.
Qed.
```

Union

Semantics

Let L_1, L_2 be context-free languages.

- $\forall w_1 \in L_1, w_1 \in L_1 \cup L_2$;
- $\forall w_2 \in L_2, w_2 \in L_1 \cup L_2$;
- $L_1 \cup L_2$ contains no other strings;
- $L_1 \cup L_2$ is context-free.

Union

General

- $G_1 = (V_1, \Sigma_1, P_1, S_1)$
- $G_2 = (V_2, \Sigma_2, P_2, S_2)$
- $G_3 = (V_1 \cup V_2 \cup \{S_3\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S_3 \rightarrow S_1, S_3 \rightarrow S_2\}, S_3)$

```
Definition g_uni (g1 g2: cfg): cfg := {|
non_terminal:= g_uni_nt g1 g2;
terminal:= g_uni_t g1 g2;
start_symbol:= Start_uni g1 g2;
rules:= g_uni_rules g1 g2 |}.
```

Union

Terminals

- $\Sigma_3 = \Sigma_1 \cup \Sigma_2$

```
Definition g_uni_t (g1 g2: cfg): Type:=
(terminal g1 + terminal g2).
```

Union

Non-terminals

- $V_3 = V_1 \cup V_2 \cup \{S_3\}$

```
Inductive g_uni_nt (g1 g2: cfg): Type :=
| Start_uni : g_uni_nt g1 g2
| Transf1_uni : non_terminal g1 -> g_uni_nt g1 g2
| Transf2_uni : non_terminal g2 -> g_uni_nt g1 g2.
```

Union

Maps

- Every symbol (string) of G_1 is a symbol (string) of G_3.

```
Definition g_uni_sf_lift_left (g1 g2: cfg)
(c: non_terminal g1 + terminal g1):
g_uni_nt g1 g2 + g_uni_t g1 g2:=
  match c with
  | inl nt => inl (Transf1_uni g1 g2 nt)
  | inr t  => inr (inl t)
  end.
```

Union

Maps

- Every symbol (string) of G_2 is a symbol (string) of G_3.

```
Definition g_uni_sf_lift_right (g1 g2: cfg)
(c: non_terminal g2 + terminal g2):
g_uni_nt g1 g2 + g_uni_t g1 g2:=
  match c with
  | inl nt => inl (Transf2_uni g1 g2 nt)
  | inr t  => inr (inr t)
  end.
```

Union

Rules

- Every rule of G_1 is a rule of G_3
- Every rule of G_2 is a rule of G_3
- $S_3 \rightarrow S_1$ is a rule of G_3
- $S_3 \rightarrow S_2$ is a rule of G_3

$N_1 \cap N_2 = \emptyset$.
$S_3 \notin N_1 \cup N_2$.

Union

Rules

```
Inductive g_uni_rules (g1 g2: cfg): g_uni_nt g1 g2 ->
list (g_uni_nt g1 g2 + g_uni_t g1 g2) -> Prop :=
| Start1_uni: g_uni_rules g1 g2 (Start_uni g1 g2)
               [inl (Transf1_uni g1 g2 (start_symbol g1))]
| Start2_uni: g_uni_rules g1 g2 (Start_uni g1 g2)
               [inl (Transf2_uni g1 g2 (start_symbol g2))]
| Lift1_uni: forall nt s,
              rules g1 nt s ->
              g_uni_rules g1 g2 (Transf1_uni g1 g2 nt)
              (map (g_uni_sf_lift_left g1 g2) s)
| Lift2_uni: forall nt s,
              rules g2 nt s ->
              g_uni_rules g1 g2 (Transf2_uni g1 g2 nt)
              (map (g_uni_sf_lift_right g1 g2) s).
```

Union

Direct operation

- $w_1 \in L(G_1)$ implies $w_1 \in L(G_3)$ and
- $w_2 \in L(G_2)$ implies $w_2 \in L(G_3)$

```
Theorem g_uni_correct (g1 g2: cfg)(s1: sf g1)(s2: sf g2):
(generates g1 s1 -> generates (g_uni g1 g2)
                    (map (g_uni_sf_lift1 g1 g2) s1))
/\
(generates g2 s2 -> generates (g_uni g1 g2)
                    (map (g_uni_sf_lift2 g1 g2) s2)).
```

Union

Inverse operation

$w \in L(G_3)$ implies:

- $w \in L(G_1)$ or
- $w \in L(G_2)$

```
Theorem g_uni_correct_inv (g1 g2: cfg)(s: sf (g_uni g1 g2)):
generates (g_uni g1 g2) s ->
(s=[inl (start_symbol (g_uni g1 g2))])
\/
(exists s1: sf g1,
(s=(map (g_uni_sf_lift_left g1 g2) s1) /\ generates g1 s1))
\/
(exists s2: sf g2,
(s=(map (g_uni_sf_lift_right g1 g2) s2) /\ generates g2 s2)).
```

Concatenation

Semantics

Let L_1, L_2 be context-free languages.

- $\forall w_1 \in L_1, \forall w_2 \in L_2, w_1 w_2 \in L_1 L_2$;
- $L_1 L_2$ contains no other strings;
- $L_1 L_2$ is context-free.

Concatenation

General

- $G_1 = (V_1, \Sigma_1, P_1, S_1)$
- $G_2 = (V_2, \Sigma_2, P_2, S_2)$
- $G_3 = (V_1 \cup V_2 \cup \{S_3\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 S_2\}, S_3)$

```
Definition g_cat (g1 g2: cfg): cfg := {|
non_terminal:= g_cat_nt g1 g2;
terminal:= g_cat_t g1 g2;
start_symbol:= Start_cat g1 g2;
rules:= g_cat_rules g1 g2 |}.
```

Concatenation

Direct operation

- $w_1 \in L(G_1)$ and
- $w_2 \in L(G_2)$

implies $w_1 w_2 \in L(G_3)$.

```
Theorem g_cat_correct (g1 g2: cfg)(s1: sf g1)(s2: sf g2):
generates g1 s1
/\
generates g2 s2 ->
generates (g_cat g1 g2)
          ((map (g_cat_sf_lift_left g1 g2) s1)++
          (map (g_cat_sf_lift_right g1 g2) s2)).
```

Concatenation

Inverse operation

$w \in L(G_3)$ implies:

- $w = w_1 w_2$ and
- $w_1 \in L(G_1)$ and
- $w_2 \in L(G_2)$.

```
Theorem g_cat_correct_inv (g1 g2: cfg)(s: sf (g_cat g1 g2)):
generates (g_cat g1 g2) s ->
exists s1: sf g1,
exists s2: sf g2,
s =(map (g_cat_sf_lift_left g1 g2) s1)++
   (map (g_cat_sf_lift_right g1 g2) s2)
/\
generates g1 s1
/\
generates g2 s2.
```

Kleene star

Semantics

Let L be a context-free language.

- $\epsilon \in L^*$;
- $\forall s' \in L^*, \forall s \in L, s's \in L^*$;
- L^* contains no other strings;
- L^* is context-free.

Kleene star

General

- $G_1 = (V_1, \Sigma_1, P_1, S_1)$
- $G_3 = (V_1 \cup \{S_3\}, \Sigma_1, P_1 \cup \{S_3 \rightarrow S_3 S_1, S_3 \rightarrow \epsilon\}, S_3)$

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

Kleene star

Direct operation

- $\epsilon \in L(G_3)$;
- $s' \in L(G_3)$ and $s \in L(G_1)$ implies $s's \in L(G_3)$.

```
Theorem g_clo_correct (g: cfg)(s: sf g)(s': sf (g_clo g)):
generates (g_clo g) nil
/\
(generates (g_clo g) s' /\ generates g s ->
generates (g_clo g) (s'++ (map (g_clo_sf_lift g)) s)).
```

Kleene star

Inverse operation

$w \in L(G_3)$ implies:

- $w = \epsilon$ or
- $w = s's$ and $s' \in L(G_3)$ and $s \in L(G_1)$.

```
Theorem g_clo_correct_inv (g: cfg)(s: sf (g_clo g)):
generates (g_clo g) s ->
(s=[])
\/
(s=[inl (start_symbol (g_clo g))])
\/
(exists s': sf (g_clo g),
 exists s'': sf g,
 generates (g_clo g) s' / generates g s'' /\
 s=s'++map (g_clo_sf_lift g) s'').
```

Development

- Approximately 1.400 lines of plain Coq script;
- Induction on predicate `derives`;
- Direct hypothesis manipulation;
- Libraries Ascii, String and List;
- Available for download at:
  `http://www.univasf.edu.br/~marcus.ramos/coq/cfg-closure.v`

Simplification

Partially completed

- Simplification = symbol elimination + rule elimination;
- Symbol n:
  - Useful: $n \Rightarrow^* s, s \in \Sigma^*$;
  - Reachable: $S \Rightarrow^* \alpha n \beta$.
- Every non-empty context-free language is generated by a context-free grammar with only useful and reachable symbols;
- Approximately 4.000 lines of Coq script;
- Dozens of additional lemmas on generic lists, context-free grammars and context-free derivations were also proved;
- 116 lemmas and theorems in total (closure + simplification) so far;
- Now working on rule elimination:
  - Empty rules;
  - Unit rules.

Applications

- Academy;
- Industry;
- Software and hardware certification;
- Software and hardware development;
- Proof checking;
- Theoretical formalization;
- Mathematics database (e.g. QED project).

Computers and mathematics

- Practitioners base is still small;
- Learning curve grows slowly;
- Advantages of formalization are immense;
- Important industrial projects;
- Disadvantages are being gradually eliminated.

Computers and mathematics

- Not easy, but very rewarding;
- Hope you have enjoyed;
- Hope you want to go further;
- 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;
- Hope to bring more next time;

Thank you!