CS420

Introduction to the Theory of Computation

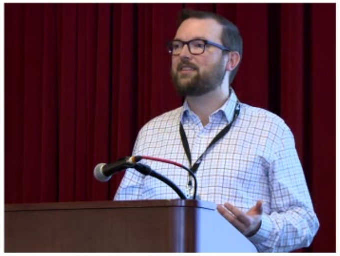
Lecture 8: Formal languages

Tiago Cogumbreiro

Today we will learn...

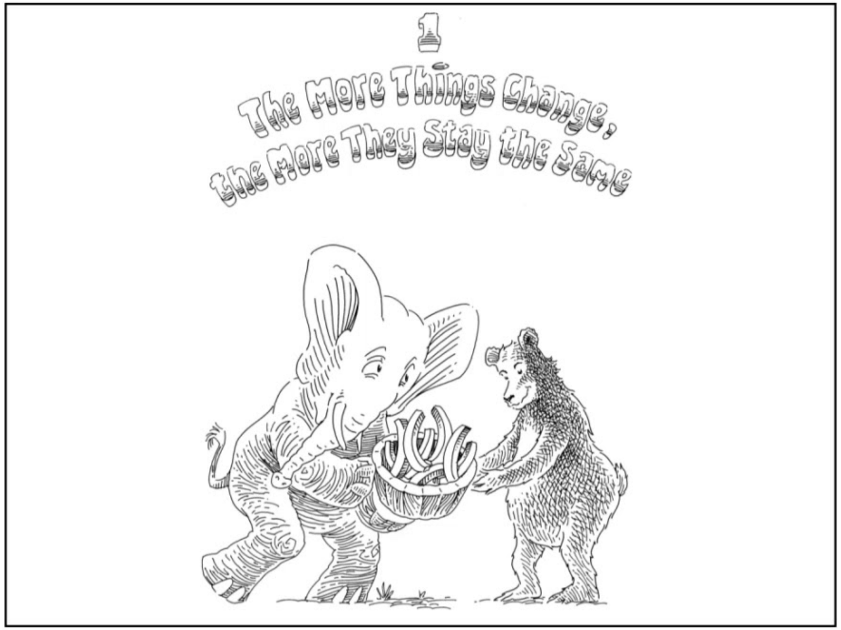
- A summary on module 1, intro do module 2
- Formal languages
- A library of languages

A little taste of dependent types



strangeloop

Sept 27-28, 2018
thestrangeloop.com



by David Christiansen. URL: `www.youtube.com/watch?v=VxINoKFm-S4`

> **Note:** Σ is `exists`, U is `Prop`, Π is `forall`

What have we learned in Module 1?

1. **A programming language to systematically prove logical facts (Coq)**
   - Dependently-typed language
   - Inductive types
   - Inductive propositions
   - Recursion and the connection to proofs by induction

What have we learned in Module 1?

1. **A programming language to systematically prove logical facts (Coq)**
   - Dependently-typed language
   - Inductive types
   - Inductive propositions
   - Recursion and the connection to proofs by induction
2. **Learn from the ground up, by assuming nothing**
   - We defined natural numbers, lists
   - We defined operations on natural numbers, lists (eg, +, -, *)
   - We proved facts about natural numbers, lists (eg, addition is commutative, associative, etc)

What have we learned in Module 1?

1. **A programming language to systematically prove logical facts (Coq)**
    - Dependently-typed language
    - Inductive types
    - Inductive propositions
    - Recursion and the connection to proofs by induction
2. **Learn from the ground up, by assuming nothing**
    - We defined natural numbers, lists
    - We defined operations on natural numbers, lists (eg, +, -, *)
    - We proved facts about natural numbers, lists (eg, addition is commutative, associative, etc)
3. **A better understanding of proofs**
    - We can look at a theorem and intuit a proof structure (case analys?, induction?)
    - We can even prove some facts like mindless robots (brute force proofs)

Where are proof assistants used?

- Industry
- Academy
- Education

Where are proof assistants used?

Industry

- CompCert is a C99 compiler written in Coq that is proved correct:
  The ***behavior*** of the output (machine code) is equivalent to that o the source code (C99).
- CompCert is used in avionics and automotive industries
- The seL4 operating system

> A formal proof of functional correctness was completed in 2009.[17] The proof provides a guarantee that the kernel's implementation is correct against its specification, and implies that it is free of implementation bugs such as deadlocks, livelocks, buffer overflows, arithmetic exceptions or use of uninitialised variables. seL4 is claimed to be the first-ever general-purpose operating-system kernel that has been verified.[17]

source: Wikipedia

Where are proof assistants used?

Academy

- Programming Language theory
- Parallel Programming theory
- Networks and distributed systems
- Cryptography
- Math (geometry)

What is programming language theory?

Programming Language theory is the cornerstone of computer science

This fields that studies:

- **abstractions of computation** (programming languages, DSLs, APIs, operating systems, distributed systems)
- **PL design & implementation**: compilers, interpreters
- **quality assurance of code** (code analyzers, linters, bug finder)
- **correctness of algorithms** (verification)

Related fields

- Logic
- Software Engineering
- DevOps (automation, DSLs)

Who hires PLT scientists?

Facebook (Automated fault-finding and fixing at Facebook) (ReasonML), Microsoft (Thinking above the code) (C#), Google (Concurrency is not parallelism) (Go, Dart), Amazon (Use of formal methods at AWS), NVidia, Intel, …

Software Verification Lab

`umb-svl.gitlab.io`

We model the behavior of intricate systems

- We identify/prove in which cases such intricate systems ***fail***
  (eg, data-races being the root causes of deadlocks)
- We build tools that help intricate systems fail less
  (eg, detecting deadlocks in distributed programs)

Why?

- To tame other people's technology — Marianne Bellotti
- To find bugs without running or even looking at the code — Jay Parlar

Where are proof assistants used?

Education

- To teach programming language theory (Benjamin Pierce, UPenn)
- To teach math (Kevin Buzzard, Imperial College)
- To teach logic
- To teach the theory of computing (here!)

What is next in Module 2?

- Formal languages
- Regular expressions
- Finite State Machines

Formal language

Formal language

> **Insight:** If we restrict what program can do, then what guarantees can we obtain from the restricted program?

- **Goal:** understanding the boundaries of computation
- **Subject:** decision procedures (a form of program)
- **Method:** introducing levels of restrictions in what programs can do

Decision procedures

- **A yes/no question:** that takes a string as input
- **A program:** that implements said question

Formal language examples

Using the mathematical notation, we simply use the **set-builder** notation to represent formal languages. **Set-membership is acceptance:** $x \in L$ reads as L accepts x.

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
    - Examples: $01 \in L_1$ $\quad 0101 \in L_1$ $\quad \texttt{foo} \notin L_1$

Formal language examples

Using the mathematical notation, we simply use the **set-builder** notation to represent formal languages. **Set-membership is acceptance:** $x \in L$ reads as L accepts x.

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
    - Examples: $01 \in L_1 \quad 0101 \in L_1 \quad \texttt{foo} \notin L_1$
- $L_2 = \{w \mid w \text{ contains character } \texttt{a}\}$
    - Examples: $000 \notin L_2 \quad \texttt{aaaaa} \in L_2$

Formal language examples

Using the mathematical notation, we simply use the **set-builder** notation to represent formal languages. **Set-membership is acceptance:** $x \in L$ reads as L accepts x.

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
  - Examples: $01 \in L_1 \quad 0101 \in L_1 \quad \texttt{foo} \notin L_1$
- $L_2 = \{w \mid w \text{ contains character } \texttt{a}\}$
  - Examples: $000 \notin L_2 \quad \texttt{aaaaa} \in L_2$
- $L_3 = \{w \mid w \text{ has 3 characters}\}$
  - Examples: $000 \in L_3 \quad \texttt{aa} \notin L_3$

Formal language examples

Using the mathematical notation, we simply use the **set-builder** notation to represent formal languages. **Set-membership is acceptance:** $x \in L$ reads as L accepts x.

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
  - Examples: $01 \in L_1 \quad 0101 \in L_1 \quad \texttt{foo} \notin L_1$
- $L_2 = \{w \mid w \text{ contains character } \texttt{a}\}$
  - Examples: $000 \notin L_2 \quad \texttt{aaaaa} \in L_2$
- $L_3 = \{w \mid w \text{ has 3 characters}\}$
  - Examples: $000 \in L_3 \quad \texttt{aa} \notin L_3$
- $L_4 = \{w \mid w \text{ is the textual representation of a prime number }\}$
  - Examples: $\texttt{aa} \notin L_4 \quad 3 \in L_4$

Formal language examples

Using the mathematical notation, we simply use the **set-builder** notation to represent formal languages. **Set-membership is acceptance:** $x \in L$ reads as L accepts x.

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
  - Examples: $01 \in L_1 \quad 0101 \in L_1 \quad \texttt{foo} \notin L_1$
- $L_2 = \{w \mid w \text{ contains character } \texttt{a}\}$
  - Examples: $000 \notin L_2 \quad \texttt{aaaaa} \in L_2$
- $L_3 = \{w \mid w \text{ has 3 characters}\}$
  - Examples: $000 \in L_3 \quad \texttt{aa} \notin L_3$
- $L_4 = \{w \mid w \text{ is the textual representation of a prime number }\}$
  - Examples: $\texttt{aa} \notin L_4 \quad 3 \in L_4$
- $L_5 = \{w \mid w \text{ is a valid C program}\}$
  - Examples: $\texttt{void main}()\{\texttt{return } 0; \} \in L_5 \qquad \texttt{aa} \notin L_5$

Formal language examples

Using the mathematical notation, we simply use the **set-builder** notation to represent formal languages. **Set-membership is acceptance:** $x \in L$ reads as L accepts x.

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
  - Examples: $01 \in L_1 \quad 0101 \in L_1 \quad \texttt{foo} \notin L_1$
- $L_2 = \{w \mid w \text{ contains character } \texttt{a}\}$
  - Examples: $000 \notin L_2 \quad \texttt{aaaaa} \in L_2$
- $L_3 = \{w \mid w \text{ has 3 characters}\}$
  - Examples: $000 \in L_3 \quad \texttt{aa} \notin L_3$
- $L_4 = \{w \mid w \text{ is the textual representation of a prime number }\}$
  - Examples: $\texttt{aa} \notin L_4 \quad 3 \in L_4$
- $L_5 = \{w \mid w \text{ is a valid C program}\}$
  - Examples: $\texttt{void main}()\{\texttt{return}\ 0;\} \in L_5 \qquad \texttt{aa} \notin L_5$
- $L_6 = \{w \mid w \text{ a valid C program and when run returns code } 0\}$

Looking ahead: formal languages

- Formal languages can be **grouped** and **ordered**
- Smaller languages represent simpler decision problems
- **Insight 1:** we can develop a restricted set of constructs to write all programs in a group
- **Insight 2:** We can know more about simpler languages

Regular $\subset$ Context-Free $\subset$ Decidable $\subset$ Turing Complete

Regular

- $L_1 = \{w \mid w \text{ starts with string } 01\}$
- $L_2 = \{w \mid w \text{ contains character } \texttt{a}\}$
- $L_3 = \{w \mid w \text{ has 3 characters}\}$

Context-free

- $L_5 = \{w \mid w \text{ is a valid C program}\}$

Decidable

- $L_4 = \{w \mid w \text{ is a prime number }\}$

Undecidable

- $L_6 = \{w \mid w \text{ a C program and returns code } 0\}$

Formal languages in Coq

How do represent a formal language in Coq?

Formal language

See `Turing.Lang`. A ***formal language*** is a predicate, of type `(list ascii) → Prop`:

- Takes a **string** (`list ascii`) and returns a **proof object** (an evidence),
- **Acceptance:** We say that the word is **accepted** by language L if, and only if `L w`.

Formal language

See `Turing.Lang`. A ***formal language*** is a predicate, of type `(list ascii) → Prop`:

- Takes a **string** (`list ascii`) and returns a **proof object** (an evidence),
- **Acceptance:** We say that the word is **accepted** by language L if, and only if `L w`.

Implementation

```
(* Boilerplate code *)
Require Import Coq.Strings.Ascii.
Require Import Coq.Lists.List.
Open Scope char_scope.
Import ListNotations.

(* Definition of a word and a language *)
Definition word := list ascii. (* Think of it as a typedef *)
Definition language := word → Prop.
Definition In w L := L w. (* A word is in the language, if we can show that [L w] holds. *)
```

Strings and their operations

A **string** is a finite sequence of characters. ϵ and $[]$ represent an empty string.

Operators

- **Length:** The length of a string, written $|w|$, is the number of characters that the string contains.
- **Substring:** String z is a substring of w if z appears consecutively within w.
- **Concatenation:** We write $x \cdot y$ for the string concatenation
- **Power:** The power operator x^n where x is a string and n is natural number, defined as x being concatenated n times (yields the empty string when $n = 0$)

$$\texttt{car}^3 = \texttt{carcarcar}$$
$$\texttt{car}^0 = \epsilon$$
$$\texttt{car}^1 = \texttt{car}$$

Strings in Coq

```
Require Import Coq.Strings.Ascii.
Require Import Coq.Lists.List.
Open Scope char_scope.
Import ListNotations.
Require Import Turing.Util.

(* Length: *)
Goal length ["c"; "a"; "r"] = 3. Proof. reflexivity. Qed.
(* Concatenation *)
Goal ["c"] ++ ["a"; "r"] = ["c"; "a"; "r"]. Proof. reflexivity. Qed.
(* Power *)
Goal pow ["c"; "a"; "r"] 3 = ["c"; "a"; "r"; "c"; "a"; "r"; "c"; "a"; "r"].
  Proof. reflexivity. Qed.
Goal pow ["c"; "a"; "r"] 1 = ["c"; "a"; "r"]. Proof. reflexivity. Qed.
Goal pow ["c"; "a"; "r"] 0 = []. Proof. reflexivity. Qed.
```

- Coq has its own string data type, but we are not using that in this course.

Preamble

```
Require Import Coq.Strings.Ascii.
Require Import Coq.Lists.List.
Require Import Turing.Util.
Require Import Turing.Lang.
Import ListNotations.
Import LangNotations.
Open Scope char_scope.
```

Example 1

> Recall that `language := word → Prop`

1. Define a language L1 that only accepts word `["c"; "a"; "r"]`
2. Show that L1 accepts `["c"; "a"; "r"]`

Example 1

- Recall that `language := word → Prop`

1. Define a language L1 that only accepts word `["c"; "a"; "r"]`
2. Show that L1 accepts `["c"; "a"; "r"]`

```
Definition L1 w := w = ["c"; "a"; "r"]. (* Define a language L1 *)

(* Show that "car" is in L1 *)
Lemma car_in_l1: In ["c"; "a"; "r"] L1.
Proof.
  unfold L1.
  reflexivity.
Qed.
```

Example 1 (continued)

3. Show that L1 rejects `["f"; "o"; "o"]`

Example 1 (continued)

3. Show that L1 rejects `["f"; "o"; "o"]`

```
(* Show that "foo" is not in L1 *)
Lemma foo_not_in_l1: ~ In ["f"; "o"; "o"] L1.
Proof.
```

Example 1 (continued)

3. Show that L1 rejects `["f"; "o"; "o"]`

```
(* Show that "foo" is not in L1 *)
Lemma foo_not_in_l1: ~ In ["f"; "o"; "o"] L1.
Proof.

  unfold not, In. (* a proof by contradiction *)
  (* Goal: L1 ["f"; "o"; "o"] → False *)
  intros N.
  (* N : L1 ["f"; "o"; "o"] *)
  (* Goal: False *)
  unfold L1 in N.
  (* N : ["f"; "o"; "o"] = ["c"; "a"; "r"] *)
  inversion N. (* Explosion principle! *)
Qed.
```

Example 2: `Vowel`

1. Language L2 accepts strings that consist of a single vowel

Example 2: `Vowel`

1. Language L2 accepts strings that consist of a single vowel

```
Definition Vowel w := w = ["a"]
                   \/ w = ["e"]
                   \/ w = ["i"]
                   \/ w = ["o"]
                   \/ w = ["u"].
```

Example 2 (continued)

2. Show that `Vowel` accepts `["a"]`

```
Lemma a_in_vowel: In ["a"] Vowel.
  unfold Vowel.
  Print or.
  (* Inductive or (A B : Prop) : Prop := | or_introl : A → A \/ B *)
  (*                                     | or_intror : B → A \/ B *)
  apply or_introl.
  reflexivity.
Qed.
```

Example 2 (continuation)

3. Show that `Vowel` rejects `["a"; "a"]`

```
Lemma aa_not_in_vowel: ~ In ["a"; "a"] Vowel.
```

Example 2 (continuation)

3. Show that `Vowel` rejects `["a"; "a"]`

```
Lemma aa_not_in_vowel: ~ In ["a"; "a"] Vowel.

  unfold Vowel.
  intros N.
  destruct N as [N|[N|[N|[N|N]]]]; inversion N.
Qed.
```

A library of language operators

A library of language operators

- Recall that our objective is to **group languages**
- We want to have a **compositional** reasoning about languages
- **Idea:** Define an algebra of languages and study how properties behave under this algebra

Language operators

1. `Nil`
2. `Char`
3. `Union`
4. `App`

Nil

A language that only accepts the empty word.

Set-builder notation: $\{w \mid w = []\}$ or $\{w \mid w = \epsilon\}$

Nil

> A language that only accepts the empty word.

Set-builder notation: $\{w \mid w = []\}$ or $\{w \mid w = \epsilon\}$

```
Definition Nil w := w = [].
```

Correction properties

1. Show that `Nil []`
2. Show that if a word is accepted by `Nil`, then that word must be `[]`

Char

- A language that accepts a single character (given as parameter).

Char

A language that accepts a single character (given as parameter).

```
Definition Char c (w:word) := w = [c].

Coercion Char: ascii >-> language. (* Allow writing "a" rather than Char "a" *)
```

Correction properties

1. Show that the word `[c]` is accepted by `Char c`: `Char c [c]`
2. Show that any word `w` accepted by `Char c` must be equal to `[c]`

Char

A language that accepts a single character (given as parameter).

```
Definition Char c (w:word) := w = [c].

Coercion Char: ascii >-> language. (* Allow writing "a" rather than Char "a" *)
```

Correction properties

1. Show that the word `[c]` is accepted by `Char c`: `Char c [c]`
2. Show that any word `w` accepted by `Char c` must be equal to `[c]` Show that any word `[c]` is in `Char c`:

Union

- A language that accepts all words of both languages.

Union

> A language that accepts all words of both languages.

```
Definition Union (L1 L2:language) w :=
  In w L1 \/ In w L2.

Infix "U" := Union. (* Define a notation for terseness *)
```

Correction properties

1. If the word is accepted by either L1 or L2, then is accepted by `L1 U L2`
2. If the word is accepted by `L1 U L2`, then is accepted by either L1 or L2.

App

- Language L1 >> L2 accepts a word from L1 concatenated with a word from L2

App

- Language `L1 >> L2` accepts a word from `L1` concatenated with a word from `L2`

```
Definition App (L1 L2:language) w :=
  exists w1 w2, w = w1 ++ w2 /\ L1 w1 /\ L2 w2.
```

Correction properties

1. If `w1` in `L1` and `w2` in `L2`, then `w1 ++ w2` in `L1 >> L2`.
2. If `w` in `L1 >> L2`, then there exists `w1` in `L1` and w2 in `L2` such `w = w1 + w2`.