Mechanization of Proof:
From 4-Color Theorem to Compiler Verification

Chung-Kil Hur (허충길)

Department of Computer Science and Engineering
Seoul National University

My passion towards rigorous math

My question when I was young:

Can we do math rigorously?

In a set theory course:

Yes! We can do math only using first-order logic with ZFC.

But, I was disappointed:

because it looked practically impossible to do math in such a way.

Formal math thanks to computers

Now I do math formally using Coq:

Computers do such tedious details for me!

I will talk about Coq today.

What Is A Proof Assistant?

- Underlying Logic for constructing Propositions & Proofs
- Set theory in the Logic for defining Sets & Elements
- Tool that implements such a logic and a set theory
- Independent Proof Checker that checks validity of given definitions and proofs

Examples of Proof Systems

➢ Conventional Mathematics

- First-order logic
- Zermelo–Fraenkel set theory with the axiom of choice (ZFC)
- No Mechaniztion

➢ Isabelle/HOL

- Higher-order logic
- Function Space + Inductive Set
- Tool and Proof Checker
- Developed at University of Cambridge & Technical University of Munich

➢ Coq

- Calculus of Construction (Logic = Set Theory = Programming Language)
- Tool and Proof Checker
- Developed at INRIA, France

Demo of Coq

Set Theory
= **Logic**
= **Programming**

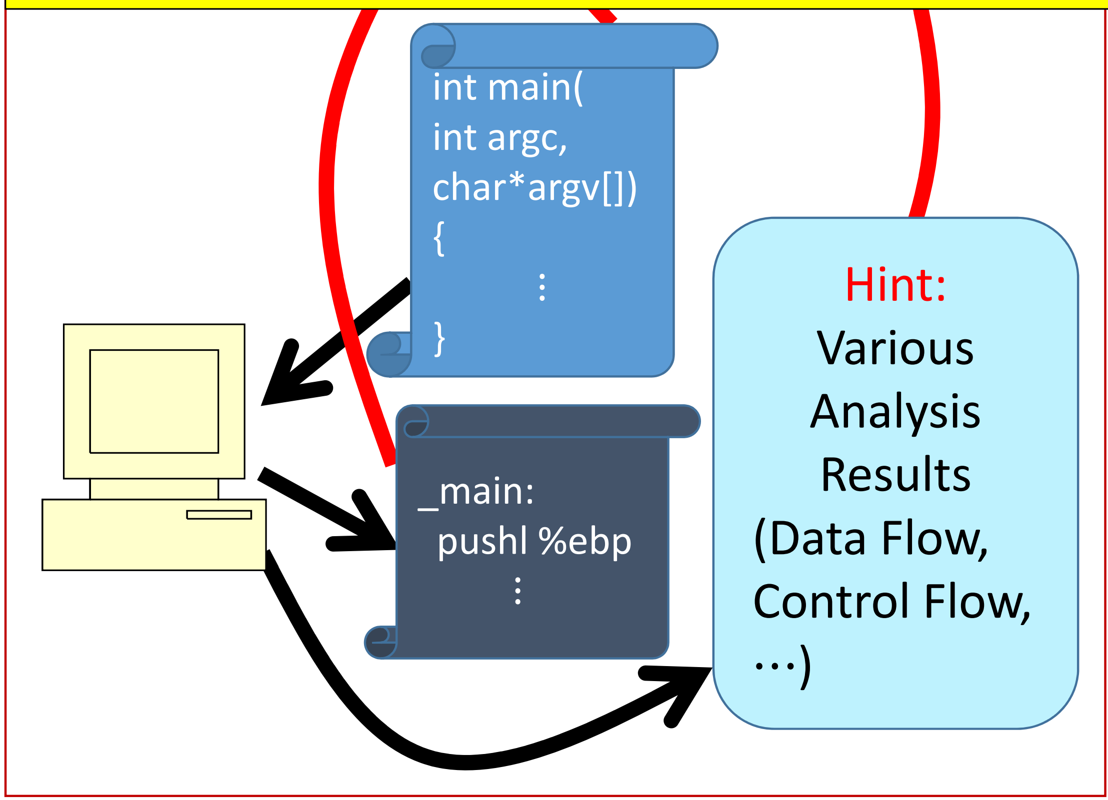
Applications of Proof Assistants

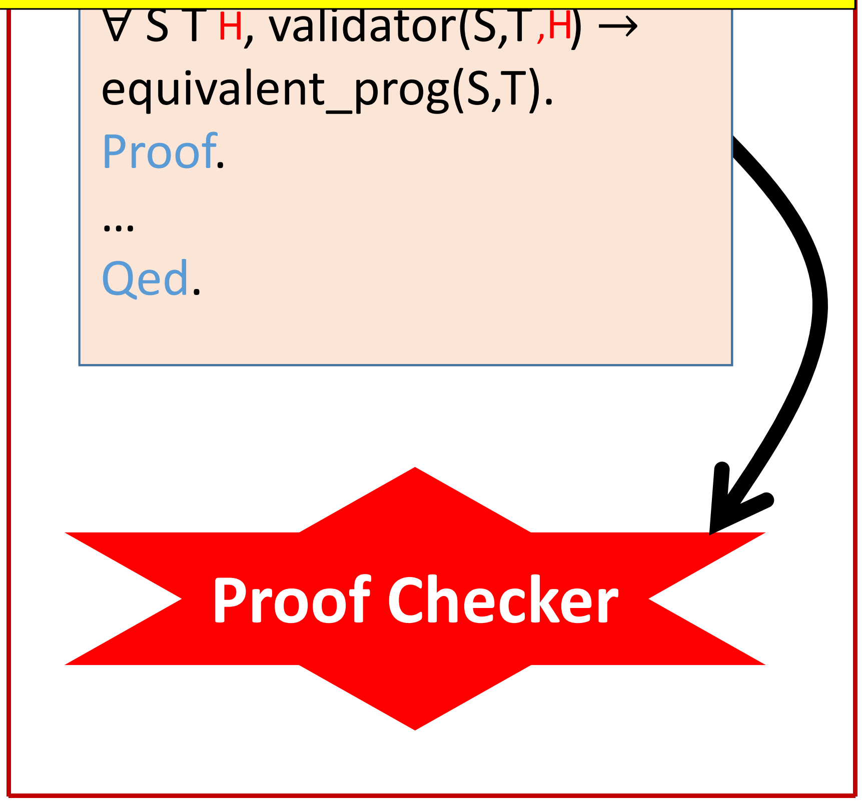
- 4-Color Theorem & Feit-Thompson theorem
  - Any map in a plane can be colored using four-colors.
  - Every finite group of odd order is solvable.
  - Mechanized in Coq by Georges Gonthier.
  - (Note) Kenneth Appel and Andrew Appel.
- seL4 (secure embedded L4)
  - Fully verified highly-optimized Microkernel based on L4
  - at NICTA, Isabelle/HOL
- CompCert
  - Fully verified optimizing C compiler
  - at INRIA, Coq (by Xavier Leroy)
  - 79 bugs in GCC, 202 bugs in LLVM, no bugs in CompCert
  - Its use in Airbus is being investigated.
- Security Protocols, Homotopy Type Theory, …

My Research: Compilation Validation

Validators can guarantee absence of bugs.

We are currently developing
a Verified Validator for LLVM Compiler.





Let me now talk about

Calculus of Construction

Calculus of Constructions (CoC)

➢ Calculus of (Inductive & Coinductive) Construction

- The type theory behind Coq
- Introduced by Thierry Coquand (1985)
- Constructive mathematics, intuitionistic logic

➢ History of Coq

- 1984: Coquand and Huet start to develop Coq
- 1989: Coquand and Paulin extend CoC to CIC
- 1994: Eduardo Giménez extends CIC to CICC
- 2014: Coq version 8.4 (still actively being developed)
- Extract to a program (only constructive part)
- Story behind the name Coq

1. Coq is the symbol of France.
2. Coq ~ COC (Calculus Of Construction)
3. Thierry COQuand

Type: The Set of All Sets

Term (program, element, proofs)

$$\boxed{\begin{array}{l} A : \mathrm{Type} \\ a : A \end{array}}$$

Type (Type, Set, Proposition)

e.g. $\mathrm{nat} : \mathrm{Type}$

$0 : \mathrm{nat}, \quad 1 : \mathrm{nat}, \quad 2 : \mathrm{nat}$

e.g. $\mathrm{Type} : \mathrm{Type}$ → ??

$\mathrm{nat} : \mathrm{Type}$

e.g. $\mathrm{nat} \to \mathrm{nat} : \mathrm{Type}$

$\mathrm{fun}\ x \Rightarrow S(x) : \mathrm{nat} \to \mathrm{nat}$

e.g. $\mathrm{Type} \to \mathrm{Type} : \mathrm{Type}$

$\mathrm{fun}\ X \Rightarrow X : \mathrm{Type} \to \mathrm{Type}$

Type Formation Rules

- Dependent function types

$$\frac{A : \text{Type} \qquad x : A \vdash B(x) : \text{Type}}{\forall x : A.\, B(x) : \text{Type}}$$

$$\boxed{A \to B \equiv \forall x : A.\, B}$$

- Inductive types

e.g. $\text{Inductive list } (A : \text{Type}) : \text{Type} :=$

$\quad | \text{ nil} : \text{list } A$

$\quad | \text{ cons} : A \to \text{list } A \to \text{list } A$

- CoInductive types ...

Term Formation Rules (1)

- Dependent function types
  - Introduction rule

$$\frac{x : A \vdash t : B(x)}{\mathrm{fun}\ x \Rightarrow t \ : \ \forall x : A.\ B(x)}$$

- recursive, corecursive functions

  - Elimination rule

$$\frac{f : \forall x : A.\ B(x) \qquad t : A}{f(t) \ : \ B(t)}$$

Term Formation Rules (2)

- Inductive types

  e.g. Inductive list (A:Type) : Type :=

  | nil : list A

  | cons : A → list A → list A .

  - Introduction rule

$$\frac{}{nil : list\ A} \qquad \frac{a : A \quad l : list\ A}{cons\ a\ l : list\ A}$$

  - Elimination rule

$$\frac{t : list\ A \quad u : B \quad x : A,\ y : list\ A \vdash v : B}{\textbf{match}\ t\ \textbf{with}\ nil \Rightarrow u \mid cons\ x\ y \Rightarrow v\ \textbf{end} : B}$$

- Coinductive types ...

Computation Rule (Equational theory)

- Dependent function types

$$\frac{x : A \vdash t : B(x) \qquad u : A}{(\mathrm{fun}\ x \Rightarrow t)\ u \equiv t[u/x] : B(u)}$$

- Inductive types

$$\frac{\cdots}{\mathrm{match\ nil\ with\ nil} \Rightarrow u \mid \mathrm{cons}\ x\ y \Rightarrow v\ \mathrm{end} \equiv u : B}$$

$$\frac{\cdots}{\mathrm{match\ cons}\ a\ l\ \mathrm{with\ nil} \Rightarrow u \mid \mathrm{cons}\ x\ y \Rightarrow v\ \mathrm{end} \equiv v[a/x][l/y] : B}$$

- Coinductive types . . .

As a Set Theory

We can define arbitrary sets using

1) inductive types
2) dependent functions
3) subset construction

+ coinductive types

e.g. $0 : nat$

As a Programming Language

We can write programs using

fun, match

+ fix, cofix

```
Definition sum : nat -> nat :=
  fix f n =>
    match n with
    | O => O
    | S m => f m + n
    end.
```

As a Logic

We can interpret types as propositions.

$$A : \text{Type}$$

A is true if A is inhabited
A is false if A is uninhabited

e.g. $\forall n : nat.\ mult\ 2\ (sum\ n) = mult\ n\ (plus\ n\ 1)$

inductive type

program = proof

$$p : \forall n : nat. \ \ldots$$

☆ $\exists$ is an iductive type

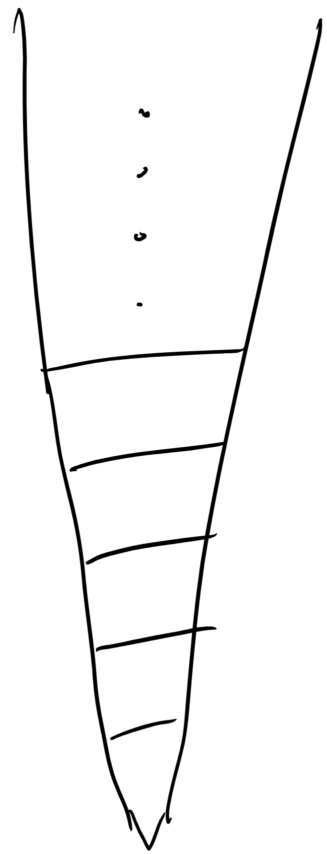
Size Problem

Type : Type leads to inconsistency!

The level of Type is implicitly determined.

Demo

Von Neumann hierarchy



Proposition

$\mathrm{Prop} \leq \mathrm{Type}$

Intuitively,

$X : \mathrm{Prop} \Rightarrow |X| = 0$ or $|X| = 1$

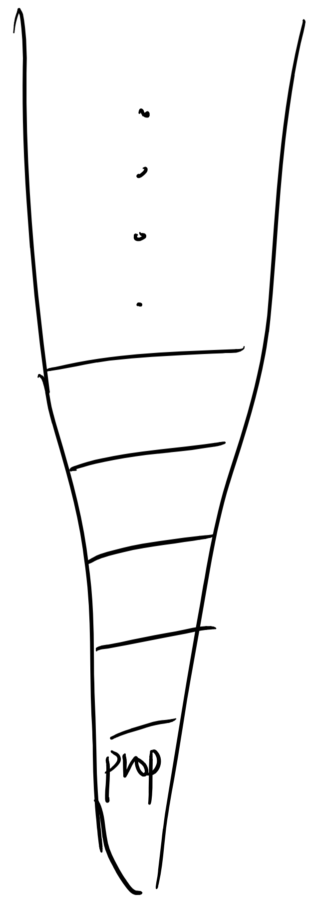
Thus, we can avoid the size problem:

e.g.

$\mathrm{Type} \to X : \mathrm{prop}$ if $X : \mathrm{prop}$

Instead, the system does not allow to distinguish elements of a proposition.

Von Neumann hierarchy



Subset Construction

$$\{x \in A \mid P(x)\}$$

$$\equiv$$

$$\prod_{x \in A} P(x)$$

Demo