ChatGPT, math-proving machine?

Tuesday, February 28, 2023 10:46 AM

Exercise 4.9 Describe explicit bijections

① $[(A \uplus B) \to C] \cong (A \to C) \times (B \to C),$

② $[A \to (B \times C)] \cong (A \to B) \times (A \to C)\ .$

① $\forall\ A \uplus B \xrightarrow{h} C$

$\exists!\ (f, g)$

$$\begin{tikzcd} & A \uplus B \arrow[dd, dashed, "h"] & \\ A \arrow[ur, "i_1"] \arrow[dr, "f"'] & & B \arrow[ul, "i_2"'] \arrow[dl, "g"] \\ & C & \end{tikzcd}$$

use lambda notation from lambda calculus to describe bijection [(A ⊔ B) \to C]= (A\to C)\times(B\to C)

To define a bijection between the functions of type `[(A ⊔ B) → C]` and `(A → C) × (B → C)` using lambda notation from lambda calculus, we need to define two lambda functions that implement the bijection in both directions.

First, let's define a function that converts a function of type `[(A ⊔ B) → C]` to a pair of functions of type `A → C` and `B → C`. We can represent this function using the lambda notation as follows:

```yaml
f : ((A ⊔ B) → C) → ((A → C) × (B → C))
f = lambda g: ((lambda a: g((a, 0))), (lambda b: g((b, 1))))
```

This function takes a function `g` of type `[(A ⊔ B) → C]` as input and returns a pair of functions, where the first function maps elements of type `A` to values of type `C`, and the second function maps elements of type `B` to values of type `C`. The functions are obtained by applying the original function `g` to pairs `(a, 0)` and `(b, 1)` that represent elements of type `A` and `B` respectively.

Now, let's define a function that converts a pair of functions of type `A → C` and `B → C` to a function of type `[(A ⊔ B) → C]`. We can represent this function using the lambda notation as follows:

```python
h : ((A → C) × (B → C)) → ((A ⊔ B) → C)
h = lambda p: (lambda x: (p[0](x[0]) if x[1] == 0 else p[1](x[0])))
```

This function takes a pair of functions `p` of type `(A → C) × (B → C)` as input and returns a function of type `[(A ⊔ B) → C]` that maps elements of type `A ⊔ B` to values of type `C`. The returned function works by taking an element `x` of type `A ⊔ B` as input and applying the corresponding function from the pair `p` based on the second component of `x`. If the second component of `x` Regenerate response ` is applied to the first component of `x`, which must be an element of type `A`. Otherwise, if the second

② $\forall\ A \xrightarrow{h} B \times C,$

$\exists!\ (f, g)$

use lambda notation from lambda calculus to describe bijection [A \to (B \times C)]= (A\to B)\times(A\to C)

To show that there is a bijection between the sets `[A → (B × C)]` and `[(A → B) × (A → C)]`, we need to construct two functions that are inverses of each other.

First, we define a function `f : [A → (B × C)] → [(A → B) × (A → C)]` as follows:

```css
f = λg. (λa. π₁(g(a)), λa. π₂(g(a)))
```

where `π₁` and `π₂` are the projection functions that extract the first and second components of a pair, respectively. That is, `π₁ : B × C → B` and `π₂ : B × C → C`.

To show that `f` is well-defined and injective, we need to show that for any two functions `g` and `h` in `[A → (B × C)]`, if `f(g) = f(h)`, then `g = h`. Suppose that `f(g) = f(h)`. Then, we have:

```css
(λa. π₁(g(a)), λa. π₂(g(a))) = (λa. π₁(h(a)), λa. π₂(h(a)))
```

By applying these functions to some argument `a ∈ A`, we obtain:

```css
π₁(g(a)) = π₁(h(a))
π₂(g(a)) = π₂(h(a))
```

Since `π₁` and `π₂` are injective, it follows that `g(a) = h(a)` for all `a ∈ A`. Therefore, `g = h`, and thus `f` is injective.

Next, we define a function `g : [(A → B) × (A → C)] → [A → (B × C)]` as follows:

```css
g = λ(f₁, f₂). λa. (f₁(a), f₂(a))
```

$$
\begin{tikzcd}
 & B\times C \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\
B & & C \\
 & A \arrow[ul, "f"] \arrow[ur, "g"'] \arrow[uu, dashed, "h"] &
\end{tikzcd}
$$

To show that `g` is well-defined and surjective, we need to show that for any function `h` in `[A → (B × C)]`, there exists a pair of functions `(f₁, f₂)` in `(A → B) × (A → C)` such that `g(f₁, f₂) = h`. Let `h` be some function in `[A → (B × C)]`. Define `f₁` and `f₂` as follows:

```css
f₁ = λa. π₁(h(a))
f₂ = λa. π₂(h(a))
```

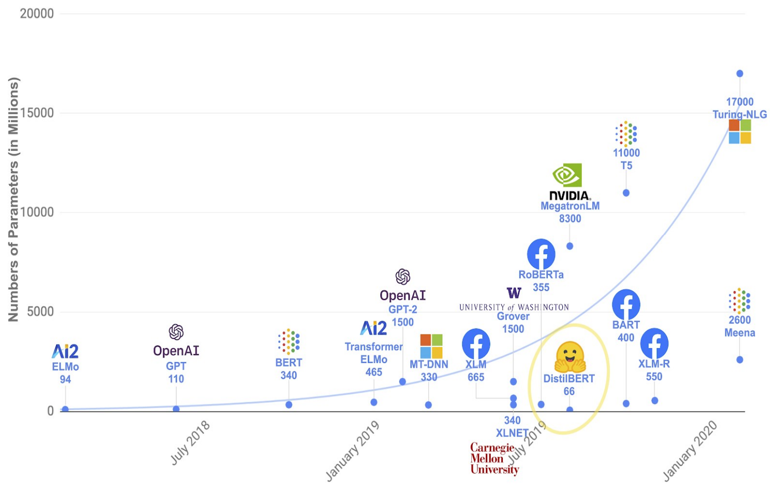
By applying `g` to `(f₁, f₂)`, we obtain:

```css
g(f₁, f₂) = λa. (f₁(a), f₂(a)) = λa. (π₁(h(a)), π₂(h(a))) = h
```

Therefore, `g` is surjective.

Since we have shown that `f` and `g` are both injective and surjective, it follows that there is a bijection between `[A → (B × C)]` and `[(A → B) × (A → C)]`.

Can AI (or in particular, large language model**) prove 'deeper' theorems?**



The Lean Theorem Prover (system description)

Leonardo de Moura[1], Soonho Kong[2], Jeremy Avigad[2],
Floris van Doorn[2] and Jakob von Raumer[2*]

[1] Microsoft Research
leonardo@microsoft.com
[2] Carnegie Mellon University
soonhok@cs.cmu.edu, {avigad, fpv, javra}@andrew.cmu.edu

Abstract. Lean is a new open source theorem prover being developed at Microsoft Research and Carnegie Mellon University, with a small trusted kernel based on dependent type theory. It aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs. Lean is an ongoing and long-term effort, but it already provides many useful components, integrated development environments, and a rich API which can be used to embed it into other systems. It is currently being used to formalize category theory, homotopy type theory, and abstract algebra. We describe the project goals, system architecture, and main features, and we discuss applications and continuing work.

Computer-formalized proofs

Fundamental Theorem of Calculus (Harrison)

Fundamental Theorem of Algebra (Milewski)

Prime Number Theorem (Avigad++ @ CMU)

Gödel's Incompleteness Theorem (Shankar)

Jordan Curve Theorem (Hales)

Brouwer Fixed Point Theorem (Harrison)

Four Color Theorem (Gonthier)

Feit-Thompson Theorem (Gonthier)

Kepler Conjecture (Hales)

The Lean Mathematical Library

The mathlib Community*

Abstract

This paper describes mathlib, a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant. Among proof assistant libraries, it is distinguished by its dependently typed foundations, focus on classical mathematics, extensive hierarchy of structures, use of large- and small-scale automation, and distributed organization. We explain the architecture and design decisions of the library and the social organization that has led to its development.

CCS Concepts • **Mathematics of computing** → *Mathematical software*; • **Security and privacy** → *Logic and verification*.

Keywords Lean, mathlib, formal library, formal proof

ACM Reference Format:
The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372885.3373824

1 Introduction

This paper describes mathlib, a formal library developed for the Lean proof assistant [20]. As a community-driven effort with dozens of contributors, there is no central organization to mathlib; it has arisen from the desires of its users to develop a repository of formal mathematical proofs. We are certainly not the first to profess this goal [1], nor is our library particularly large in comparison to others. However, its organizational structure, focus on classical mathematics, and inclusion of automation distinguish it in the space of proof assistant libraries. We aim here to explain our design decisions and the ways in which mathlib has been put to use.

In contrast to most modern proof assistant libraries, many of the contributors to mathlib have an academic background in pure mathematics. This has significantly influenced the contents and direction of the library. It is a goal of many in the community to support the formalization of modern, research-level mathematics, and various projects discussed in Section 7.2 suggest that we are approaching this point.

1.1 A History of mathlib and Lean 3

The Lean project was started by Leonardo de Moura in 2013 [20]. Its most recent version, Lean 3, was released in

$\sqrt{2}$ is irrational

```
import data.nat
open nat

theorem sqrt_two_irrational {a b : ℕ} (co : coprime a b) : a^2 ≠ 2 * b^2 :=
assume H : a^2 = 2 * b^2,
have even (a^2),
  from even_of_exists (exists.intro _ H),
have even a,
  from even_of_even_pow this,
obtain (c : ℕ) (aeq : a = 2 * c),
  from exists_of_even this,
have 2 * (2 * c^2) = 2 * b^2,
  by rewrite [-H, aeq, *pow_two, mul.assoc, mul.left_comm c],
have 2 * c^2 = b^2,
  from eq_of_mul_eq_mul_left dec_trivial this,
have even (b^2),
  from even_of_exists (exists.intro _ (eq.symm this)),
have even b,
  from even_of_even_pow this,
have 2 | gcd a b,
  from dvd_gcd (dvd_of_even `even a`) (dvd_of_even `even b`),
have 2 | (1 : ℕ),
  by rewrite [gcd_eq_one_of_coprime co at this]; exact this,
show false, from absurd `2 | 1` dec_trivial
```

Stander Symposium Abstract for Set Theory Class

Lean Theorem Prover: The Lean, Mean, Math-Proving Machine

This is an exploratory project for MTH 342 - Set Theory. Lean Theorem Prover is a computer programming language that allows for the formalization of mathematical proofs and the use of computer-readable logic. We explore the structure and syntax of Lean and show how this can be used to formalize mathematical proofs. We identify classic math proofs that have already been formalized within Lean, as well as discuss how this language can advance the writing of proofs. Finally, we investigate proofs that are still yet to be formalized, and the potential reasons why they have yet to achieve formalization in Lean.

[Submitted on 25 May 2022]

Autoformalization with Large Language Models

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, Christian Szegedy

Autoformalization is the process of automatically translating from natural language mathematics to formal specifications and proofs. A successful autoformalization system could advance the fields of formal verification, program synthesis, and artificial intelligence. While the long-term goal of autoformalization seemed elusive for a long time, we show large language models provide new prospects towards this goal. We make the surprising observation that LLMs can correctly translate a significant portion (25.3%) of mathematical competition problems perfectly to formal specifications in Isabelle/HOL. We demonstrate the usefulness of this process by improving a previously introduced neural theorem prover via training on these autoformalized theorems. Our methodology results in a new state-of-the-art result on the MiniF2F theorem proving benchmark, improving the proof rate from 29.6% to 35.2%.

Comments: 44 pages
Subjects: **Machine Learning (cs.LG)**; Artificial Intelligence (cs.AI); Logic in Computer Science (cs.LO); Software Engineering (cs.SE)
Cite as: arXiv:2205.12615 **[cs.LG]**
(or arXiv:2205.12615v1 **[cs.LG]** for this version)
https://doi.org/10.48550/arXiv.2205.12615

Submission history

From: Yuhuai(Tony) Wu [view email]