TTIC 31230, Fundamentals of Deep Learning

David McAllester, Autumn 2024

Theorem Proving

MathZero?

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last – a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO

In 2017 AlphaZero learned to play Go, chess and shogi at a superhuman level entirely from self play given only the rules of the game.

Lots of people have wondered whether something similar might be done for mathematics.

AlphaProof

The IMO (International Math Olypiad) is an annual mathemamtics contest for high school students.

Of the six problems in July 2024 IMO, DeepMind's AlphaGeometry solved the one geometry problem and AlphaProof solved three of the remaining five.

The solutions were worth a silver medal. However, AlphaProof took three days on a supercomputer rather than the 1.5 hours humans get per problem.

There is a blog post with a high level description (discussed below) but no paper has yet appeared.

Three Cultures

Mathematicians: Building an AI system that can learn to play chess, Go or shogi is different from playing chess, Go or shogi. Building a systems that can do mathematics different from doing mathematics.

AI Researchers: There is now a significant literature on using LLMs to generate formal proofs.

Logicians: The developers of formal verification systems that ensure correctness are part of the formal methods community (programming languages, type theory, and formally sound automated reasoning).

Mathematicians

Terence Tao (Fields Medalist) organized a group effort that machine verified one of his (coauthored) recent papers. (September 2023).

Peter Scholze (Fields Medalist) organized a group effort that machine verified what he considered his most important theorem (2022).

Tim Gowers (Fields Medalist) is now focusing on improving the level of automation in machine verfication systems.

Kevin Buzzard has announced a project to machine verify Fermat's last theorem.

AI Researchers

Generative Language Modeling for Automated Theorem Proving,
Polu and Sutskever, (Sept, 2020, OpenAI)

AlphaProof, Deep Mind, July 2024, unpublished

DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback
for Reinforcement Learning and Monte-Carlo Tree Search
Xin, et al., August 2024

Logicians

Logicians are concerned with the formal foundations of mathematics and formal reasoning algorithms.

The choice of foundation is in many ways analogous to a choice of formal programming languages. There are religious factions.

- **Amorphous:** Metamath
- **ZFC:** Mizar
- **HOL:** HOL light, Isabelle/HOL, HOL4
- **Dependent Type Theory:** Coq, Agda, Lean, Alfred

For programming the choice of language does matter — python is better than assembler.

Logicians

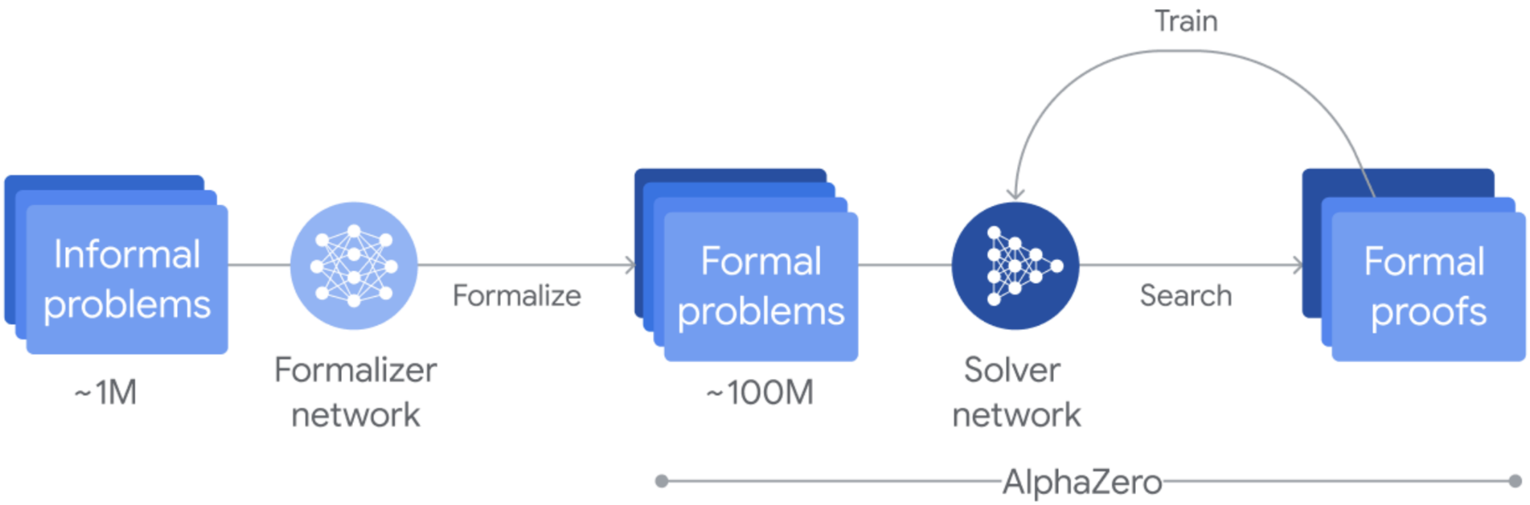
Currently the dominant system (language) is Lean.

AlphaProof and DeepSeek use Lean to verify formal statements.

I have built my own formal foundation I'm calling Alfred.

AlphaProof

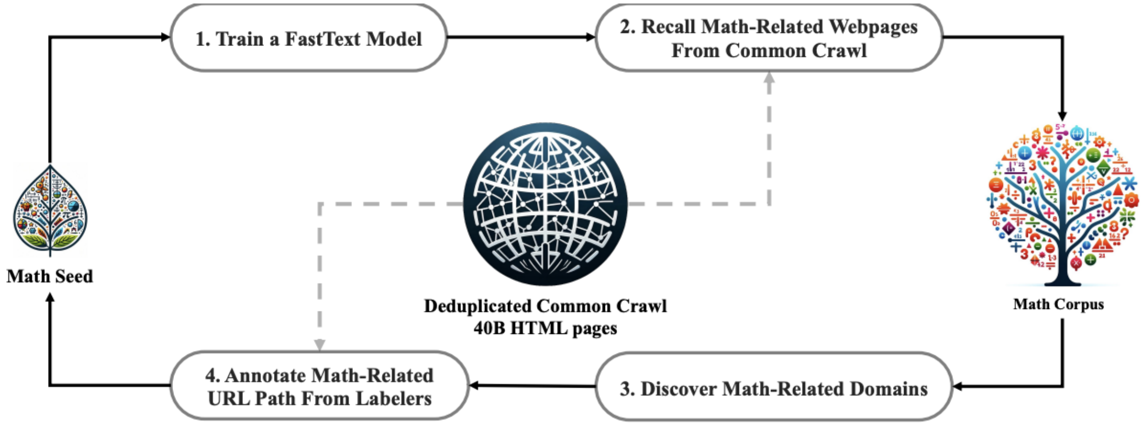
The blog post includes the following figure.



DeepSeek Pretraining Corpus

Since we don't have a paper on AlphaProof I will focus on DeepSeek Prover.

The DeepSeek Provers architecture seems to be consistent with what we know about AlphaProof.


1. Train a FastText Model
2. Recall Math-Related Webpages From Common Crawl
Math Seed
Deduplicated Common Crawl 40B HTML pages
Math Corpus
4. Annotate Math-Related URL Path From Labelers
3. Discover Math-Related Domains

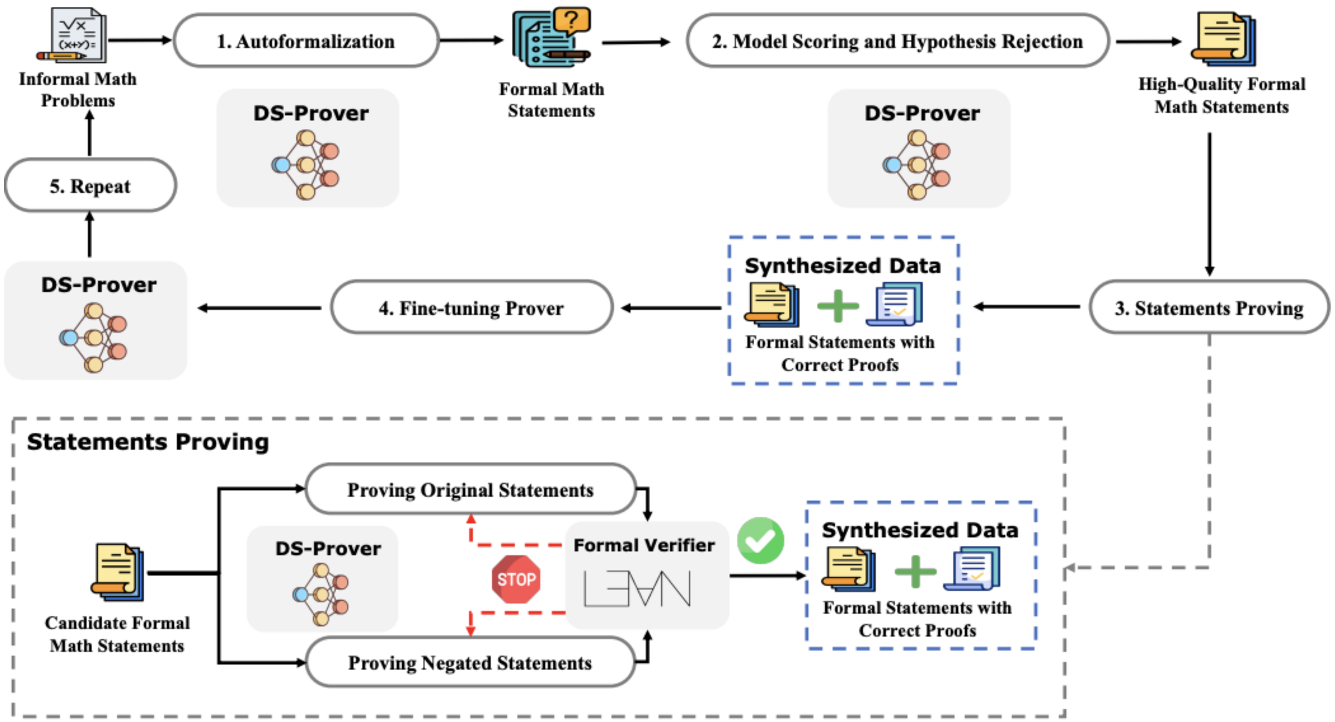
DeepSeek collected 40B tokens of mathematical text.

Autoformalization

Autoformalization with Large Language Models (Google, May 2022)

DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data (DeepSeek AI, May 2024)

Autoformalization in DeepSeek



Informal Math Problems
1. Autoformalization
DS-Prover
Formal Math Statements
2. Model Scoring and Hypothesis Rejection
DS-Prover
High-Quality Formal Math Statements
5. Repeat
DS-Prover
4. Fine-tuning Prover
Synthesized Data
Formal Statements with Correct Proofs
3. Statements Proving
Statements Proving
Candidate Formal Math Statements
Proving Original Statements
DS-Prover
STOP
Formal Verifier
LEAN
Proving Negated Statements
Synthesized Data
Formal Statements with Correct Proofs

Autoformalization is the process of converting informal (English) statements into formal (machine readable) statements.

Automatic Blueprinting

The term “blueprint” is used in software engineering and refers to specifications and dependencies for modules.

It should be possible (??) to use autoformalization to convert a mathematics paper or textbook into a blueprint for verification.

The blueprint is a partial order on formal statements where each statement is to be proved from earlier statements.

The statements in proofs can included in the blueprint.

Self Play Training of Autoformalization

Autoformalization can be error-prone so one can sample different possible formalizations.

The correct formalization is one that can be proved and that is used either as a step in a correct proof or in the proof of a later theorem.

Searching for Proofs: Goals and Tactics

For a set of goals S and tactic T we have that $T(S)$ is a new set of goals such that solving $T(S)$ solves S.

Example of tactics include:

apply: For applying a particular lemma under a certain instantiation

simplify: For simplifying an algebraic expression.

induction: For applying mathematical induction.

show: For showing a particular intermediate step that can then be added to the premises of a goal.

and-or Proof Trees

Given a set of goals we can try to solve the goals seperately by applying a tactic to each individual goal.

This results in an and-or search tree. At an and node we need solve all the chidren and at an or node we need to solve only one of the children.

HyperTree Proof Search for Neural Theorem Proving, Lample et al., May 2022 (Meta).

and-or Proof Trees

An and-or proof search is similar to chess or Go in that we can imagine two agents where one tries to prove the result and other tries to refute it.

This is equivalent to min-max search where the maximizer proves and the minimizer refutes.

We can then use α-β search, or Monte-Carlo tree search, or AlphaZero.

Whole Proof Generation

Baldur: Whole-Proof Generation and Repair with Large Language Models, First et al., March 2023, (UMass Amherst, Google, UIUC).

One can repeatedly ask the language model to generate a compete proof — a sequence of tactics that solves the given goal.

Whole proof generation is used as a subroutine in DeepSeek Prover.

Whole proof generation is typically associated with or-branching tree search where each node is a set of subgoals.

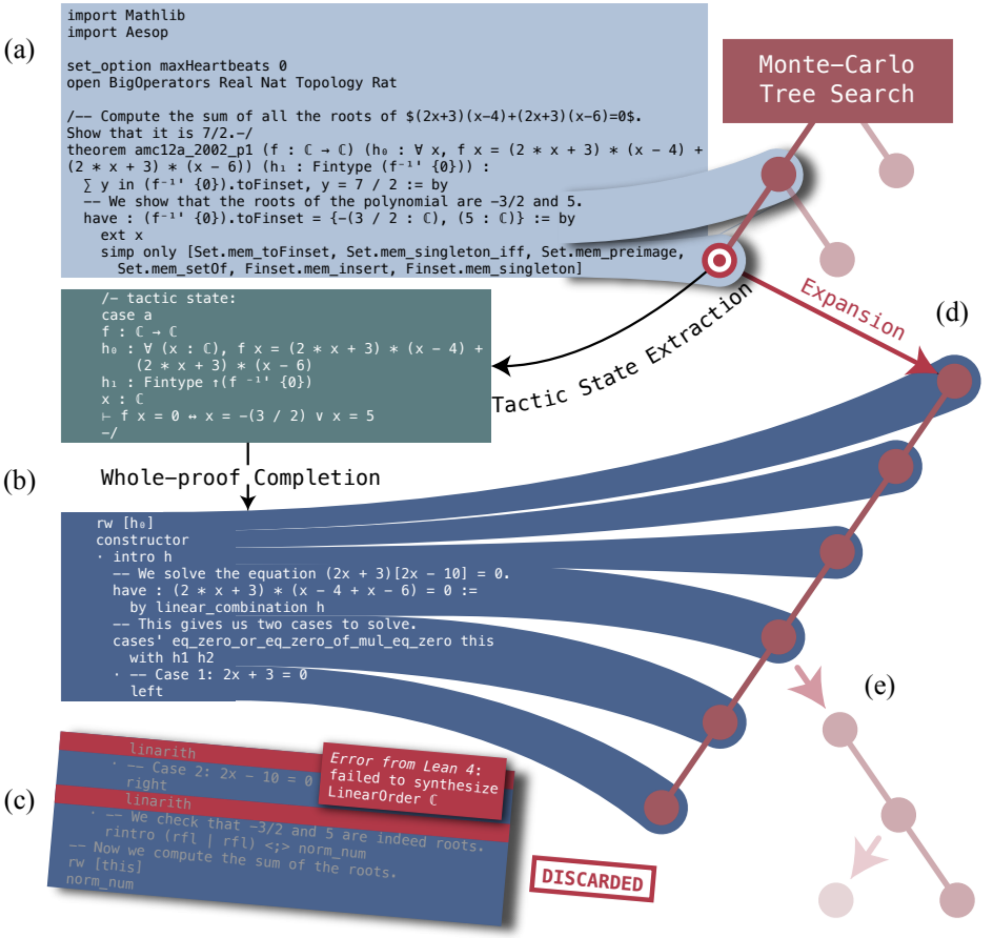
DeepSeek1.5

DeepSeek 1.5 uses whole proof generation but treats these as “rollouts” in an or-tree.

The steps in each generated whole proof become new nodes in an or-tree.

Rollouts (whole proof generation) is done from internal nodes.

DeepSeek 1.5



```
import Mathlib
import Aesop

set_option maxHeartbeats 0
open BigOperators Real Nat Topology Rat

/-- Compute the sum of all the roots of $(2x+3)(x-4)+(2x+3)(x-6)=0$.
Show that it is 7/2.-/
theorem amc12a_2002_p1 (f : ℂ → ℂ) (h₀ : ∀ x, f x = (2 * x + 3) * (x - 4) +
(2 * x + 3) * (x - 6)) (h₁ : Fintype (f⁻¹' {0})) :
  ∑ y in (f⁻¹' {0}).toFinset, y = 7 / 2 := by
  -- We show that the roots of the polynomial are -3/2 and 5.
  have : (f⁻¹' {0}).toFinset = {-(3 / 2 : ℂ), (5 : ℂ)} := by
    ext x
    simp only [Set.mem_toFinset, Set.mem_singleton_iff, Set.mem_preimage,
      Set.mem_setOf, Finset.mem_insert, Finset.mem_singleton]
```

```
    /- tactic state:
    case a
    f : ℂ → ℂ
    h₀ : ∀ (x : ℂ), f x = (2 * x + 3) * (x - 4) +
        (2 * x + 3) * (x - 6)
    h₁ : Fintype ↑(f ⁻¹' {0})
    x : ℂ
    ⊢ f x = 0 ↔ x = -(3 / 2) ∨ x = 5
    -/
```

```
    rw [h₀]
    constructor
    · intro h
      -- We solve the equation (2x + 3)[2x - 10] = 0.
      have : (2 * x + 3) * (x - 4 + x - 6) = 0 :=
        by linear_combination h
      -- This gives us two cases to solve.
      cases' eq_zero_or_eq_zero_of_mul_eq_zero this
        with h1 h2
      · -- Case 1: 2x + 3 = 0
        left
        linarith
      · -- Case 2: 2x - 10 = 0
        right
        linarith
    · -- We check that -3/2 and 5 are indeed roots.
      rintro (rfl | rfl) <;> norm_num
  -- Now we compute the sum of the roots.
  rw [this]
  norm_num
```

```
Error from Lean 4:
failed to synthesize
LinearOrder ℂ
```

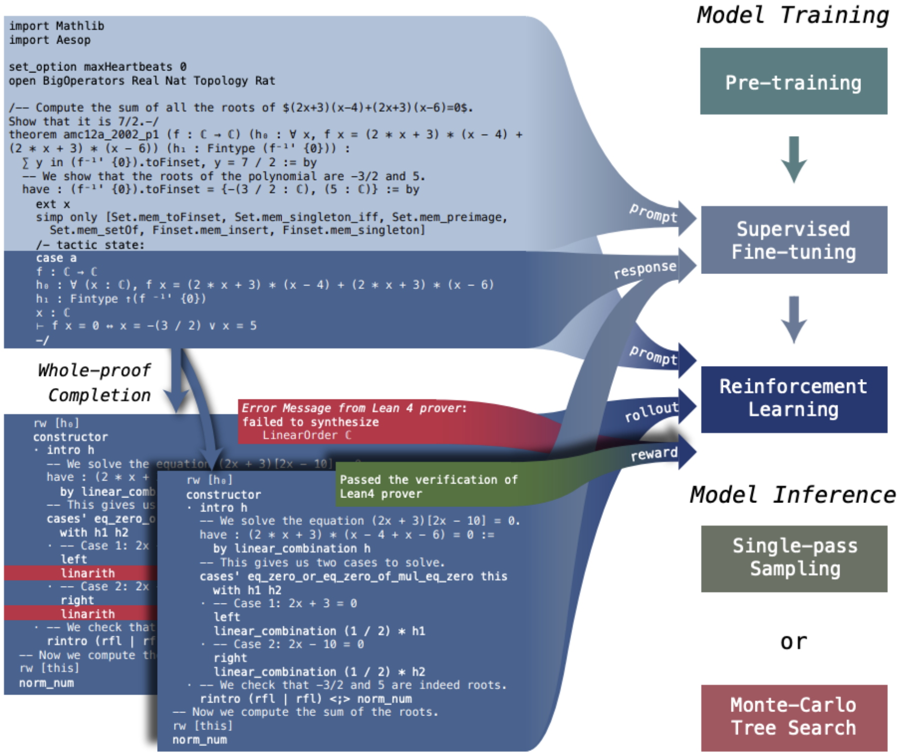
English Comments and Intrinsic Reward

DeepSeek uses monte-carlo tree search with an intrinsic reward for exploration (generating new nodes).

DeepSeek inserts natural language comments for each tactic application which might help guide the proof.

Self Play Imitation Learning

As they prove new formal theorems the new theorems are added to a database of proofs used in “imitation fine tuning”.



```
import Mathlib
import Aesop

set_option maxHeartbeats 0
open BigOperators Real Nat Topology Rat

/-- Compute the sum of all the roots of $(2x+3)(x-4)+(2x+3)(x-6)=0$.
Show that it is 7/2.-/
theorem amc12a_2002_p1 (f : ℂ → ℂ) (h₀ : ∀ x, f x = (2 * x + 3) * (x - 4) +
(2 * x + 3) * (x - 6)) (h₁ : Fintype (f⁻¹' {0})) :
  ∑ y in (f⁻¹' {0}).toFinset, y = 7 / 2 := by
  -- We show that the roots of the polynomial are -3/2 and 5.
  have : (f⁻¹' {0}).toFinset = {-(3 / 2 : ℂ), (5 : ℂ)} := by
    ext x
    simp only [Set.mem_toFinset, Set.mem_singleton_iff, Set.mem_preimage,
      Set.mem_setOf, Finset.mem_insert, Finset.mem_singleton]
    /- tactic state:
    case a
    f : ℂ → ℂ
    h₀ : ∀ (x : ℂ), f x = (2 * x + 3) * (x - 4) + (2 * x + 3) * (x - 6)
    h₁ : Fintype ↑(f ⁻¹' {0})
    x : ℂ
    ⊢ f x = 0 ↔ x = -(3 / 2) ∨ x = 5
    -/
```

```
    rw [h₀]
    constructor
    · intro h
      -- We solve the equation (2x + 3)[2x - 10]
      have : (2 * x +
        by linear_comb
      -- This gives us
      cases' eq_zero_o
        with h1 h2
      · -- Case 1: 2x
        left
        linarith
      · -- Case 2: 2x
        right
        linarith
    · -- We check that
      rintro (rfl | rf
  -- Now we compute th
  rw [this]
  norm_num
```

Error Message from Lean 4 prover:
failed to synthesize
  LinearOrder ℂ

```
    rw [h₀]
    constructor
    · intro h
      -- We solve the equation (2x + 3)[2x - 10] = 0.
      have : (2 * x + 3) * (x - 4 + x - 6) = 0 :=
        by linear_combination h
      -- This gives us two cases to solve.
      cases' eq_zero_or_eq_zero_of_mul_eq_zero this
        with h1 h2
      · -- Case 1: 2x + 3 = 0
        left
        linear_combination (1 / 2) * h1
      · -- Case 2: 2x - 10 = 0
        right
        linear_combination (1 / 2) * h2
    · -- We check that -3/2 and 5 are indeed roots.
      rintro (rfl | rfl) <;> norm_num
  -- Now we compute the sum of the roots.
  rw [this]
  norm_num
```

Passed the verification of
Lean4 prover

GRPO RL Optimization

DeepSeek uses a variant of PPO called GRPO for RL.

PPO is an advantage-actor-critic policy gradient algorithm.

$$\nabla_\Theta R(\pi_\Theta) \approx E_{s_t,a_t}\left[(\ln \pi_\Theta(a_t))\hat{A}_\Psi(s_t,a_t)\right]$$

For a fixed advantage $\hat{A}_\Psi(s_t,a_t)$ this gradient is a nonlinear function of Θ.

Rather than take a single linear step, PPO moves some distance along this nonlinear path.

In AlphaZero this happens when elements of the replay buffer are reused.

DeepSeek does not use tree-search bootstrapping.

LLM Proofs Without Formal Verification

The blog post on alphaproof also says that they are experimenting with proofs generated entirely by natural language — no formal verification. They say:

> We also tested [the language only] approach on this year's IMO problems and the results showed great promise.

OpenAI o1 as a Proof Assistant

Terence Tao has a Mastodon post evauating o1 as an assistant for research mathematicians. He says:

> The experience seemed roughly on par with trying to advise a mediocre, but not completely incompetent, (static simulation of a) graduate student.

Formal Language vs. Natural Language

In chess, Go, shogi, it is trivial to define the moves of the game in a formal language.

This is also true in the game of Diplomacy where orders to generals have a fixed grammar and even statements in negotiation are best viewed as proposals for orders.

Mathematics can also be defined in terms of a formal language.

This would suggest that English/formal translation is ultimately not difficult and the hard challenge is solving the game defined in the rules of formal mathematics.

Formal Language vs. Natural Language

An argument in favor of LLM reasoning (rather than formal reasoning) is that mathematics requires intuition and intuition is not formal.

Time will tell if intuition is best captured in natural language or in some internal language with formal grammar but imprecise semantics.

Algorithms in the Age of LLMs

What is the role of algorithms in the age of LLMs?

It seems unlikely that LLMs will ever replace compilers which, by deep model standards, are **incredibly efficient**.

Lean includes certain decision procedures such as one for deciding whether a linear equation follows from a given set of linear equations and inequalities.

AlphaGeometry (a DeepMind geometry theorem prover) relies on Wu's algorithm from the 1970s.

Can Lean be Strengthened?

There are well-known algorithms for general logic that are used to great effect in SMT (Sat Modulo a Theory) software verifiers (such as Microsoft's Z3).

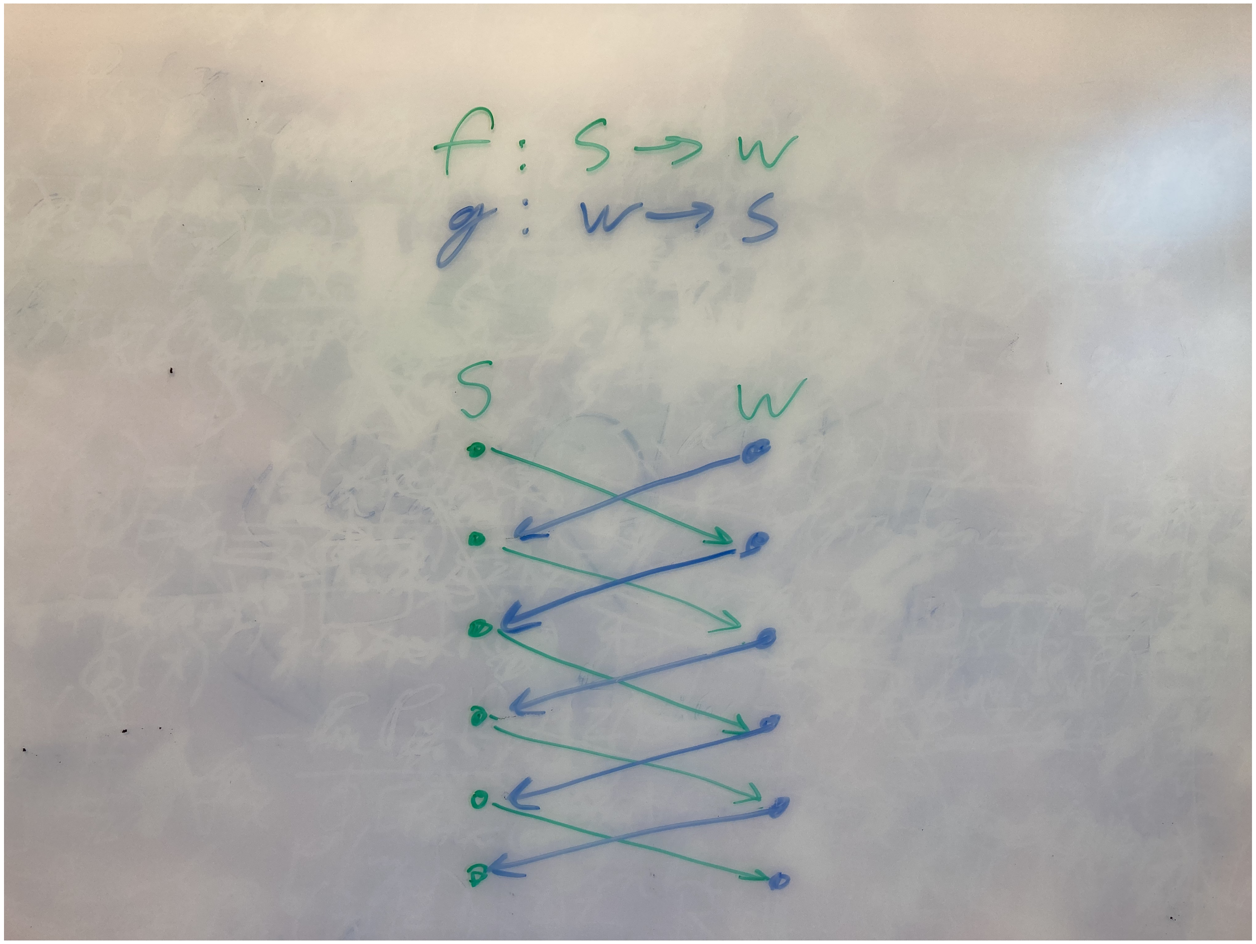
Alfred is a system that I have built based on general logic algorithms not present in Lean.

Alfred is intended as a competitor for Lean.

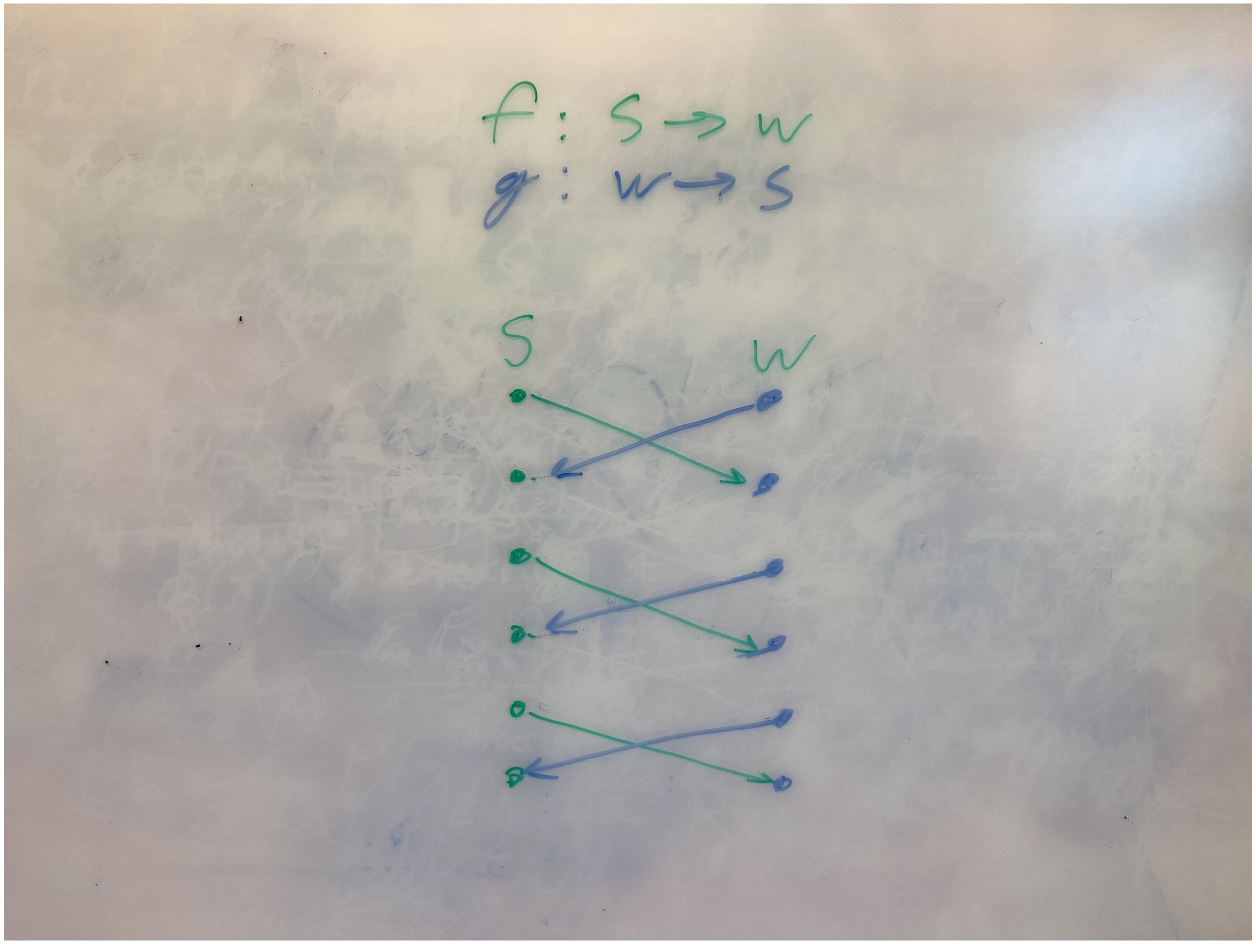
The Schroder-Berstein Theorem

For any two sets s and w, if there exists an injection of s into w and an injection of w into s then there exists a bijection between s and w.

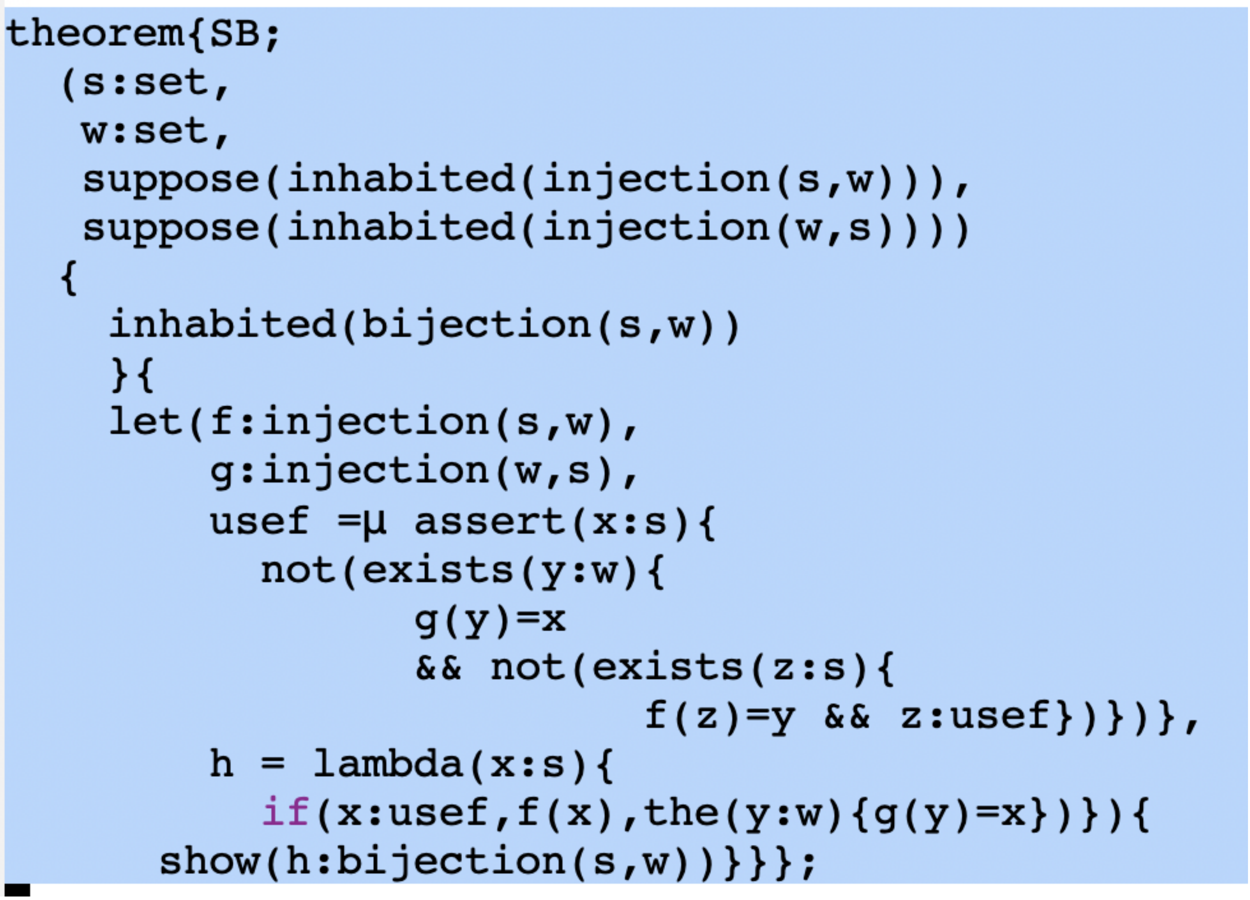
The Schroder-Berstein Theorem



The Schroder-Berstein Theorem

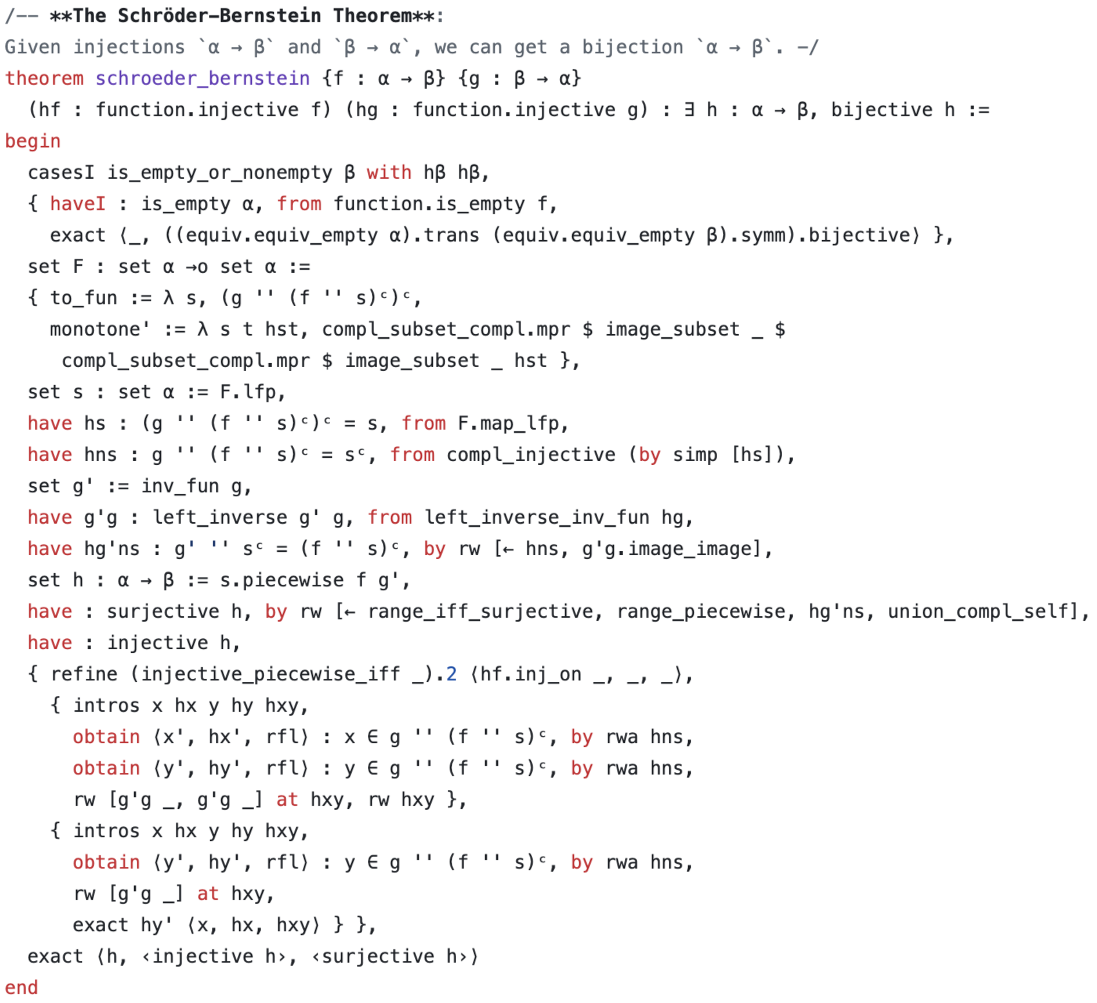


The Schroder-Berstein Theorem in Alfred



```
theorem{SB;
  (s:set,
   w:set,
   suppose(inhabited(injection(s,w))),
   suppose(inhabited(injection(w,s))))
  {
    inhabited(bijection(s,w))
    }{
    let(f:injection(s,w),
        g:injection(w,s),
        usef =μ assert(x:s){
          not(exists(y:w){
                g(y)=x
                && not(exists(z:s){
                         f(z)=y && z:usef}))})},
        h = lambda(x:s){
          if(x:usef,f(x),the(y:w){g(y)=x}))}){
      show(h:bijection(s,w))}}};
```

The Schroeder-Bernstein Theorem from Lean’s MathLib



```
/-- **The Schröder-Bernstein Theorem**:
Given injections `α → β` and `β → α`, we can get a bijection `α → β`. -/
theorem schroeder_bernstein {f : α → β} {g : β → α}
  (hf : function.injective f) (hg : function.injective g) : ∃ h : α → β, bijective h :=
begin
  casesI is_empty_or_nonempty β with hβ hβ,
  { haveI : is_empty α, from function.is_empty f,
    exact ⟨_, ((equiv.equiv_empty α).trans (equiv.equiv_empty β).symm).bijective⟩ },
  set F : set α →o set α :=
  { to_fun := λ s, (g '' (f '' s)ᶜ)ᶜ,
    monotone' := λ s t hst, compl_subset_compl.mpr $ image_subset _ $
     compl_subset_compl.mpr $ image_subset _ hst },
  set s : set α := F.lfp,
  have hs : (g '' (f '' s)ᶜ)ᶜ = s, from F.map_lfp,
  have hns : g '' (f '' s)ᶜ = sᶜ, from compl_injective (by simp [hs]),
  set g' := inv_fun g,
  have g'g : left_inverse g' g, from left_inverse_inv_fun hg,
  have hg'ns : g' '' sᶜ = (f '' s)ᶜ, by rw [← hns, g'g.image_image],
  set h : α → β := s.piecewise f g',
  have : surjective h, by rw [← range_iff_surjective, range_piecewise, hg'ns, union_compl_self],
  have : injective h,
  { refine (injective_piecewise_iff _).2 ⟨hf.inj_on _, _, _⟩,
    { intros x hx y hy hxy,
      obtain ⟨x', hx', rfl⟩ : x ∈ g '' (f '' s)ᶜ, by rwa hns,
      obtain ⟨y', hy', rfl⟩ : y ∈ g '' (f '' s)ᶜ, by rwa hns,
      rw [g'g _, g'g _] at hxy, rw hxy },
    { intros x hx y hy hxy,
      obtain ⟨y', hy', rfl⟩ : y ∈ g '' (f '' s)ᶜ, by rwa hns,
      rw [g'g _] at hxy,
      exact hy' ⟨x, hx, hxy⟩ } },
  exact ⟨h, ‹injective h›, ‹surjective h›⟩
end
```

END