TTIC 31230, Fundamentals of Deep Learning

David McAllester, Autumn 2024

Theorem Proving

MathZero?

In 2017 AlphaZero learned to play Go, chess and shogi at a superhuman level 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 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 (users): 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.

Deep Learning Researchers: There is now a significant literature on using LLMs to generate formal proofs (LLMPGs).

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).

Interest From Mathematicians

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

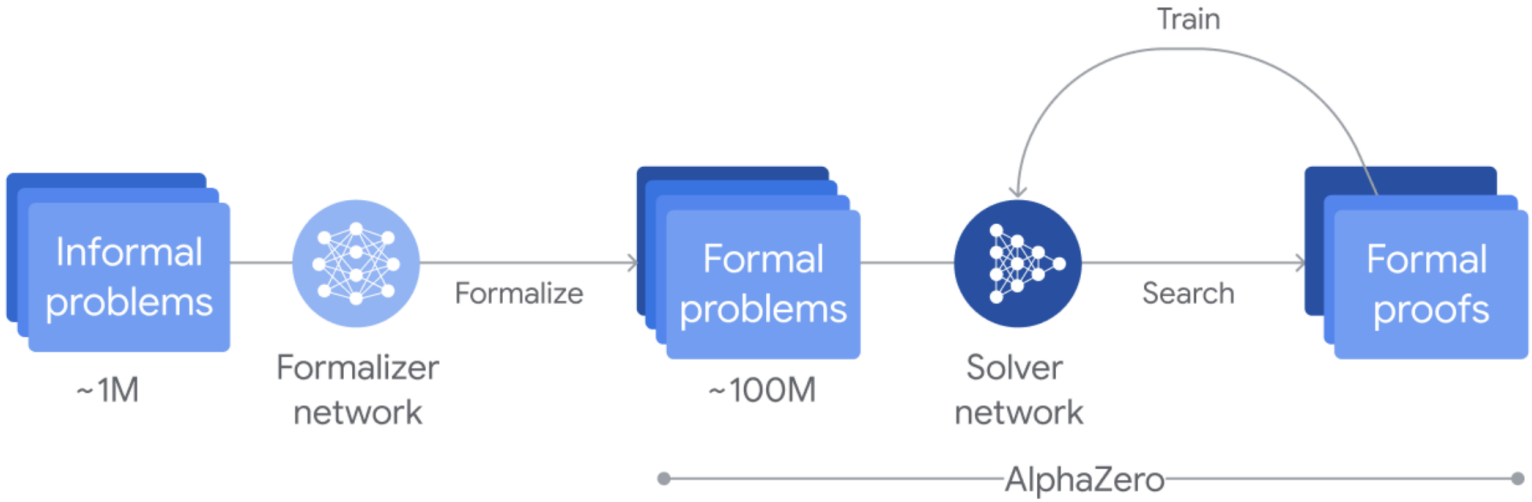
Peter Scholze (Fields Medalist) organized a group effort that machine verified what he considered his most important theorem (2022).

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.

AlphaProof

The blog post includes the following figure.



Research in LLMPGs

Generative Language Modeling for Automated Theorem Proving, Polu and Sutskever, (Sept, 2020, OpenAI, cited by 223)

The state of the art for open source LLMPGs seems to be DeepSeek 1.5 (August 15, 2024, DeepSeek).

Logicians: The LEAN System

By far, the dominant system for ensuring correctness is LEAN.

Users: Mathematician users, such as Terence Tau, Peter Scholze, and Kevin Buzzard, work directly in LEAN with possible assistance from LLMs used in the same way that programmers use LLMs – to make suggestions.

Deep Learning Researchers: Deep learning researchers are almost exclusively using LEAN to ensure correctness.

Logicians: I have built a system (Alfred) that is intended to be a compeitor for LEAN.

LLMPGs: Goals and Tactics

I will call a statement that remains to be proved a goal.

In LEAN a tactic is a function that takes a set of goals and "backward chains" to produce a new set of goals.

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

LLMPGs: Goals and Tactics

For a proof state S and tactic T we have that $T(S)$ is a new proof state such that solving $T(S)$ solves S.

For example, if a goal involves a formula such as $2x + 5x$ we might use a simplification tactic to replace this expression with $7x$. Or one might apply induction tactic on a specified well-founded order.

For each proof state there is a choice of tactic.

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, 93 citations).

A Choice in LLMPG Architectures

- **AND-OR search**

- **OR-search:** Each node is a set of goals.

- **Whole proof generation:** Repeatedly ask the languge model to generate a compete proof — a sequence of tactics that solves the given goal.

  Baldur: Whole-Proof Generation and Repair with Large Language Models, First et al, March 2023, (Google, 75 Citations).

Another Choice

Do we insert natural language "thoughts" from the language model at internal nodes of the search.

A model doing this will be called a chain-of-thought LLMPG.

More Choices

Do we use some form of AO*-like tree growth procedure or Monte-Carlo tree search as in AlphaZero.

Do we try to incorporate "intrinsic" reward into RL optimization to handle the sparse reward in mostly failing proof attempts.

DeepSeek1.5

DeepSeek 1.5 appears to be the state of the art LLMPG among those with a published paper.

It uses whole proof generation but treats these as "rollouts" in an OR-tree. The steps in each generated whole proof become new nodes in the OR-tree and rollouts (whole proof generation) can be done from internal nodes.

This is a chain-of-thought LLMPG with English thoughts at internal nodes of the tree.

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

LLMPGs 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.

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 an 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).

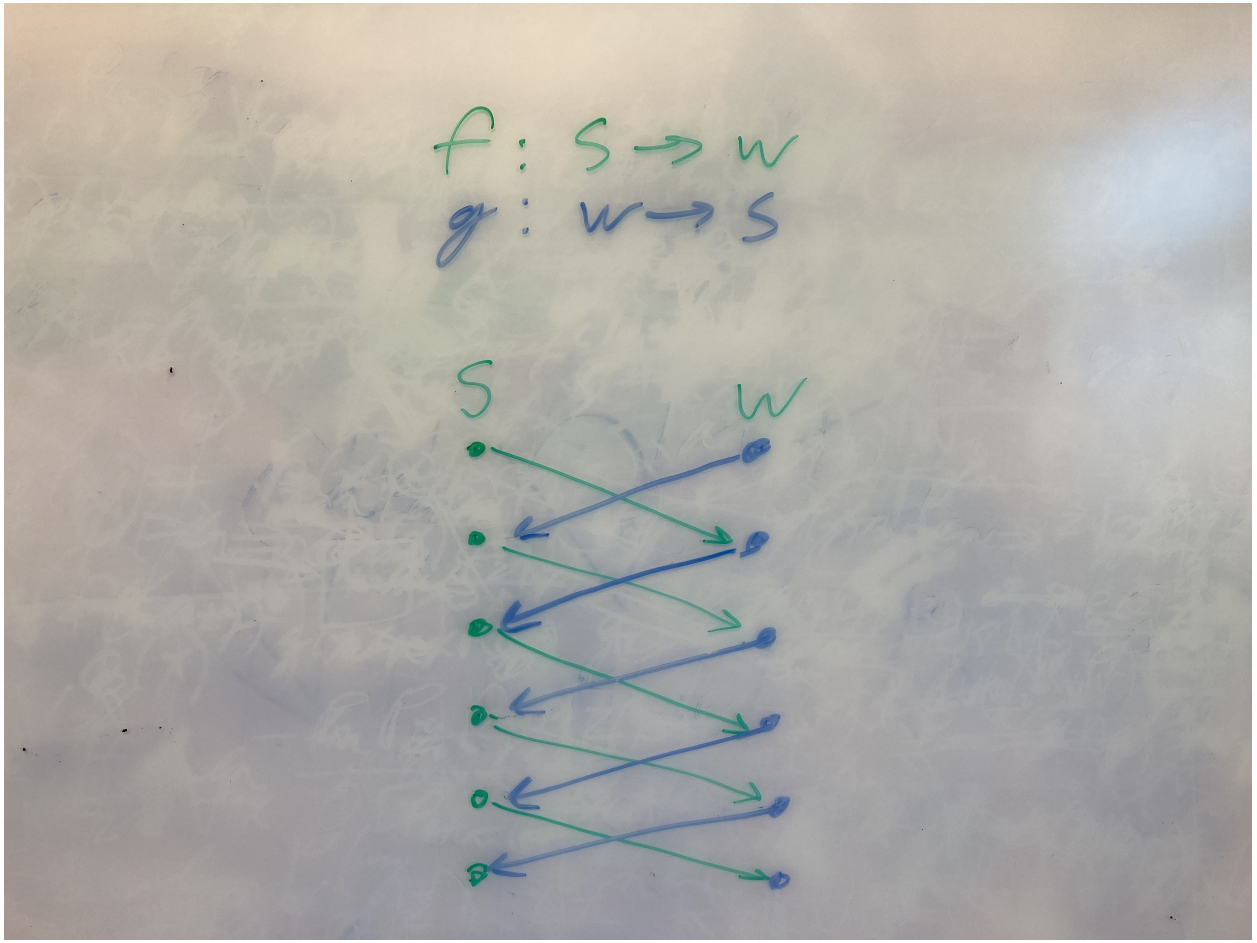
Alfred is a system that I have built based on general logic algorithms not present in LEAN.

Alfred is intended as a competitor for LEAN.

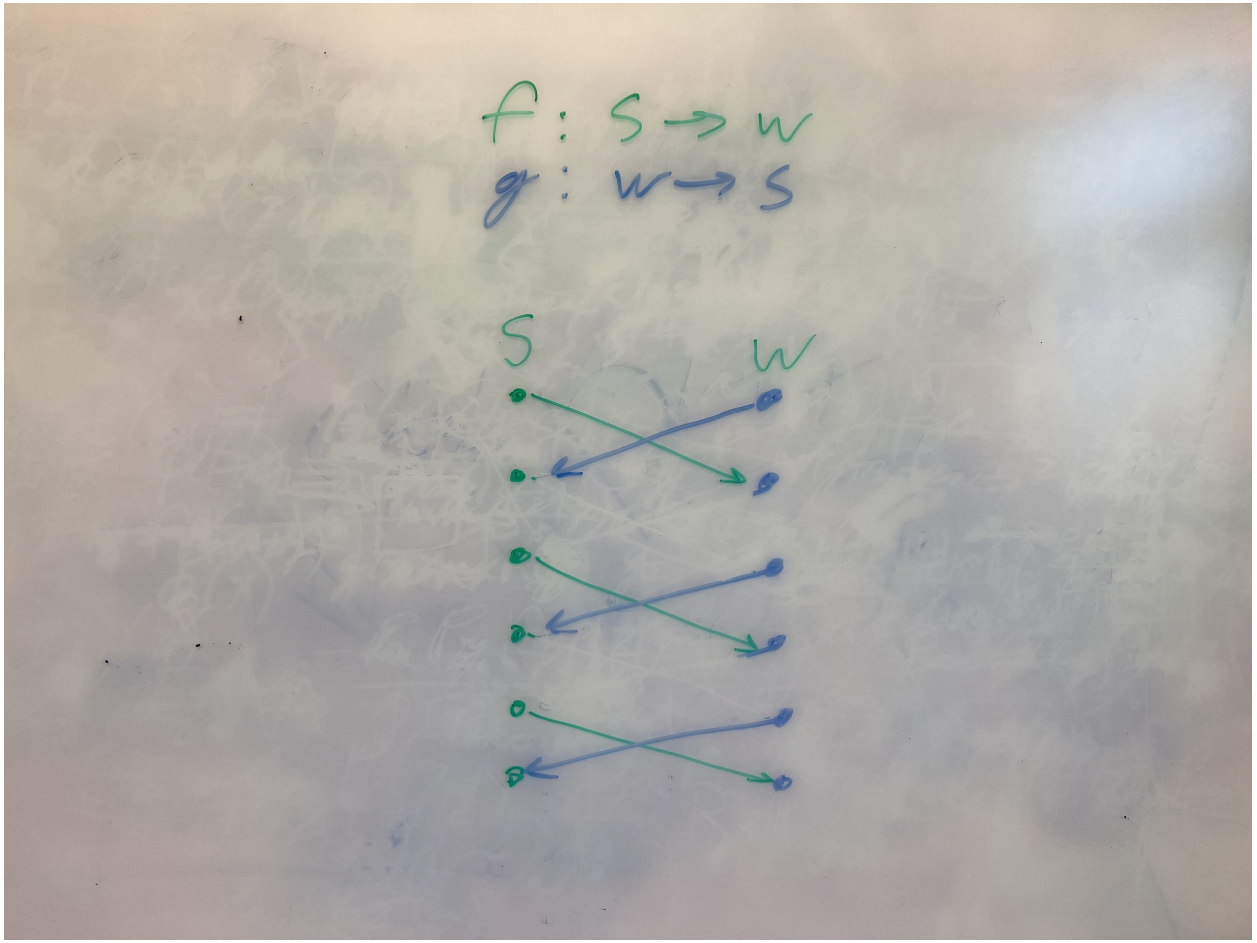
The Schroder-Berstein Theorem

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
```

Why Does LEAN Currently Dominate?

Grammatical Functor Property: Any well-typed (grammatically correct) definition of a mapping between classes denotes a **functor between groupoids**.

To explain what this means, and why it is significant, I will need to develop some background in logic.

First Order Logic

As simple example we consider the language defined by a constant **zero** and a succesor function **succ**.

In this language we can write the following axioms.

$$\neg\exists x\ \mathrm{succ}(x) = \mathrm{zero}$$
$$\forall x\ (x \neq \mathrm{zero}) \Rightarrow \exists y\ x = \mathrm{succ}(y)$$
$$\forall x, y, z\ (\mathrm{succ}(y) = x \wedge \mathrm{succ}(z) = x) \Rightarrow y = z$$

But this does not pin down the natural numbers — there are models of these axioms with elements not reachable from zero.

The Grammar of First Order Logic

A theorem proving system must manipulate data structures representing terms and formulas.

The first order language of **zero** and **succ** has terms and formulas defined by the following grammar.

$$t ::= \text{variable} \;||\; \text{zero} \;||\; \text{succ}(t)$$

$$\Phi ::= t_1 = t_2 \;||\; \forall x\ \Phi[x] \;||\; \exists x\ \Phi[x] \;||\; \Phi_1 \vee \Phi_2 \;||\; \Phi_1 \wedge \Phi_2 \;||\; \neg\Phi$$

Signatures

A first order language is defined by a set of constant, function and predicate symbols each with a specified arity (number of arguments).

The set of constant, function and predicate symbols together with their arity is called the **signature** of the language.

The signature defines a gammar specifying a set of **grammatically well-formed** terms and formulas.

Multi-Sorted Logic

A multi-sorted signature consists of a set of "sorts" and a specification $f : \tau$ of a type τ for the each symbol f of the language.

A vector space has two sorts — one for scalars and one for vectors. The multiplication-by-a-scalar operator has the type specification

$$\text{SVProd} : \text{scalar} \times \text{vector} \rightarrow \text{vector}$$

Higher Order Multi-Sorted Logic

The "simple types" over a given set of sorts consist of the expressions that can be constructed from the sorts, the constant type bool, and the type constructors $\times$ and $\rightarrow$.

$$\tau ::= \text{sort} \;||\; \text{bool} \;||\; \tau_1 \times \tau_2 \;||\; \tau_1 \rightarrow \tau_2$$

A topological space has one sort — the points — and an (second order) predicate **open** which has the type specification

$$\text{open} : (\text{point} \rightarrow \text{bool}) \rightarrow \text{bool}$$

The induction axiom for arithmetic can be written as

$$\forall P : (N \rightarrow \text{bool})\ (P(\text{zero}) \wedge \forall x : N\ P(x) \rightarrow P(s(x))) \rightarrow \forall x : N\ P(x)$$

Extending Terms with Pairs and Functions

As in programming languages, we now extend terms to include pairs, projections of pairs, functions, and applications of functions.

Pairing $\langle s, u \rangle$ and projections $\pi_1(e)$ and $\pi_2(e)$.

λ-expressions (functions) $\lambda\, x\!:\!\tau\ e[x]$ and applications $f(e)$.

It is not difficult to define the grammar of this extended set of terms.

Signature-Axiom Classes

Common mathematical concepts can be defined as models of a multi-sorted signature satifying given axioms written in the language defined by the signature.

Intuitively we have a "data type" specified by the signature. An instance of this data type is a model (particular data) specifying a value for each sort and a value for each declared symbol (consistent with the type declarations).

We also have "axioms" which are the properties that the data must satisfy. We assume the axioms to be grammaticaly well-formed (well typed).

The Signature-Axiom Groupoid

Two structures of the same signature are isomorphic if there exists a system of bijections between the sorts which carry the data of one to the data of the other.

It is straightforward to define the notion of "carry" for simply typed language constants.

It is also straightforward to prove that if two models of the same signature are isomorphic then they satisfy the same (grammatical) formulas.

We then get that every signature-axiom class defines a groupoid — a category in which every morphism is an isomorphism.

We now have Groupoid Classes.

What About Functors?

We want a formal language for expressing functions (functors) between groupoids.

We want that isomorphic graphs have the same graph Laplacian because the definition of the graph Laplacian is grammatically well-formed (well typed).

We want that isomorphic topological manifolds have isomorphic homotopy groups because the definition of the homotopy group is grammatically well-formed (well typed).

LEAN's Advantage

Grammatical Functor Property: Any well-typed (grammatically correct) definition of a mapping between classes denotes a **functor between groupoids**.

This implies that one can substitute isomorphic objects into any **well typed** context.

$$\frac{\Gamma \models f : \sigma \to \tau \qquad \Gamma \models u =_\sigma v}{\Gamma \models f(u) =_\tau f(\sigma)}$$

The Importance of Isomorphism: Classification

Classification is a central objective of mathematics. Classifying the finite groups, or topological manifolds, or differentiable manifolds, or Lie groups.

Classification is "up to isomorphism".

We can expect an autonomous AI mathematician to naturally be oriented toward classification problems.

The Importance of Isomorphism: Representation

Any two three-dimensional vector spaces over the reals are isomorphic (although there is no natural or canonical isomorphism).

$\mathbb{R}^3$, defined as the set of triples of real numbers, is a representation of a three dimentional vector space over the reals.

"Representation theory" is the study of the representation of groups as linear operators on vector spaces.

The Importance of Isomorphism:

Cryptomorphism

People immediately recognize when two different types are "the same" or "provide the same data".

A group can be defined in terms of the group operation, the identity element, and the inverse operation, or alternatively, just the group operation.

Birkhoff (1967) called the relationship between these two fomulations of group a cryptomorphism.

Two classes σ and τ are cryptomorphic if there exists well-formed functors $F : \sigma \to \tau$ and $G : \tau \to \sigma$ whose composition is the identity.

The Importance of Isomorphism:

Symmetry

Any x of type τ has a τ symmetry group — the set of τ-automorphisms of x (isomorphisms of x with itself). For example a geometric circle has rotational and reflective symmetries.

If $x : \tau$ and $y : \sigma$ are τ-σ-cryptomorphic then the τ symmetry group of x must be isomorphic (as a permutation group) to the σ symmetry group of y.

If we treat cryptomorphic objects as just different expressions of "the same data" then an object has no natural or canonical structure beyond its symmetry group.

The Role of Constructivism

The grammatical functor property was proved for Martin-Löf type theory, from which LEAN is derived, by Hofmann and Streicher in 1995.

But Martin-Löf type theory (1972) was motivated entirely by constructivism.

It turns out that there is no need for constructivism. The success of LEAN should not be viewed as a vindication of constructivism.

This talk presents a dependent type system satisfying the grammatical functor property but derived as a direct generalization of the semantics of multi-sorted higher-order logic.

Constructive Logic:
Propositions as Types

Constructive logics formulate propositions as the types of computer programs.

Proof checking is reduced to type checking.

A proof of P is a well-typed program whose type is P.

Constructive logic must be extended with the excluded middle and the axiom of choice to support working mathematicians.

Semantics

Constructive logic is specified by inference rules.

Following Tarski (1933) we have specified logics **semantically**.

We write $\Sigma \models \Phi$ to mean that Φ is true in all models of Σ.

Semantics defines soundness and completeness and is needed to formulate Gödel's incompleteness theoerms.

We will continue to work semantically and simply generalize a little further the logic developed so far.

First Class Sorts

In a programming language something is "first class" if it can be passed as an argument to a procedure and included as a value in data structues.

A group contains its sort as part of its data (the set of group elements).

To define the type "group" we need sorts to be included in objects — we need first class sorts.

Dependent Pair Types

To support first class sorts we now include set as a type so that we can declare a sort s with $s : \mathrm{set}$.

We generalize $\sigma \times \tau$ to $\Sigma_{x:\sigma}\, \tau[x]$ which denotes the set of all pairs $\langle x, y \rangle$ with $x \in \sigma$ and $y \in \tau[x]$.

$$\mathrm{magma} : \Sigma_{s:\mathrm{set}}[s \times s \to s]$$

Dependent Function Types

We generalize $\sigma \to \tau$ to $\Pi_{x:\sigma}\, \tau[x]$ which denotes the set of all functions f such that the domain of f is σ and for all $x \in \sigma$ we have $f(x) \in \tau[x]$.

$$\text{cons} : \Pi_{\alpha:\text{set}}\ (\alpha \times \text{listof}(\alpha)) \to \text{listof}(\alpha)$$

Axioms

Axioms can be incoporated into the type system with "some such that" types technically known as **restriction types**.

The some-such-that type $S_{x:\tau}\ \Phi[x]$ denotes the type of those values $x : \tau$ satisfying the "axiom" $\Phi[x]$.

$$\textbf{Group} \equiv \Sigma_{s:\mathrm{Set}}\ S_{f:s\times s\to s}\ \Phi[s, f]$$

Constructive Logic "Axioms"

It seems natural to represent a group as a signature-axiom class.

$$\textbf{Group} \equiv \Sigma_{s:\mathrm{Set}}\, S_{f:s\times s\to s}\, \Phi[s,f]$$

In constructive type theories one replaces the restriction type with a pair type.

$$\textbf{Group} \equiv \Sigma_{s:\mathrm{Set}}\, \Sigma_{f:s\times s\to s}\, \Phi[s,f]$$

Here a proof of the axioms must always be given as part of the data of the group.

The Signature-Axiom Distinction in Programming

In a typed programming language a procedure is declared by specifying types for its arguments and return value. This declaration is called the "signature" of the procedure.

Programming languages also support "assertions" — run-time checks on program invariants. For example, one might assert that at this point in the program the variable x is an even number.

Compile-time checking of assertions is undecidable. Assertions become run-time checks.

Alfred

Alfred, named for Alfred Tarski, is an under-development system intended to compete with LEAN.

Any competitive advantage over LEAN will be due to the level of automation. Time will tell ...

The Signature-Axiom Distinction in Alfred

Alfred has a decidable **signature-checking** algorithm for the type system defined here analogous to type checking in programming language with run-time assertions.

For a mathematical verification system we also want **axiom-checking**. If f is a functor taking a group as an argument we want to check that in any application $f(G)$ we have that G is a group.

This is analogous to verifying the run-time assertions in a computer program.

Handling Undecidability

Alfred has a quickly terminating but incomplete axiom-checker. We make this as strong as possible while preserving quick termination.

If the axiom-checker fails to prove that G is a group we can first provide an explicit proof.

Variants of Dependent Type Theory

I will reserve the term "dependent type theory" for type system satisfying the grammatical functor property.

In practice such a system should be given a ZFC-complete inference mechanism (rules or algorithms).

Just as with typed programming languages, among dependent type theories the choice of particular language features matters.

Of particular interest is object-oriented type systems.

Speculation:

The Grammar of Mathematical Natural Language

Just as in all human languages, human mathematical language has grammar.

Dependent type theory can be interpreted as a formal treatment of the grammar of the natural language of mathematics.

Speculation:

Object-Oriented Everything

Modern programming languages support object-oriented programming.

Mathematics is object-oriented in the sense that one deals with classes, such as the class of groups, and instances.

Class-instance structure (object orientation) underlies natural language semantics (Fillmore 1976).

Perhaps large language models will eventually make a transition from the transformer to an object-oriented archiecture.

Speculation:

What is an Electron?

If we view cryptomorphic objects as "the same data", and we view objects with the same symmetry group as cryptomorphic, then a mathematical object has no natural or canonical structure beyond its symmetry group. It then seems natural that an electron (or the value space of the electron field) has no identifiable structure beyond its symmetry group.

Summary: The Grammatical Functor Property

$$\frac{\Gamma \models f : \sigma \to \tau \qquad \Gamma \models u =_\sigma v}{\Gamma \models f(u) =_\tau f(\sigma)}$$

END