TTIC 31230, Fundamentals of Deep Learning

David McAllester, Autumn 2024

Logic

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

$$\mathrm{SVProd} : \mathrm{scalar} \times \mathrm{vector} \to \mathrm{vector}$$

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 $\rightarrow$.

$$\tau ::= \mathrm{sort} \mid\mid \mathrm{bool} \mid\mid \tau_1 \times \tau_2 \mid\mid \tau_1 \rightarrow \tau_2$$

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

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

The induction axiom for arithmetic can be written as

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

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 $f(x) \in \tau[x]$.

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

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" $\Phi[x]$.

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

Constructive Logic "Axioms"

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

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

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

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

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

$$\frac{\begin{array}{l}\Gamma \models f : \sigma \to \tau \\ \Gamma \models u =_\sigma v\end{array}}{\Gamma \models f(u) =_\tau f(\sigma)}$$

END