# TTIC 31230, Fundamentals of Deep Learning

David McAllester, Autumn 2024

# The Foundations of Mathematics

## Zermelo-Fraenkel Set Theory with the Axiom of Choice (ZFC)

ZFC is a set of nine axioms in first order logic with equality and a single relation $\in$.

ZFC plus a plus a large cardinal axiom defines the set of all possible mathematical theorems.

A theorem of mathematics is a formula in the language of set theory that is provable using the nine axioms of ZFC, a large cardinal axiom, and the inference rules of first order logic.

# First Order Logic

A first order language is defined by the Boolean operations $\vee$, $\wedge$, $\Rightarrow$ and $\neg$ (and, or, implies and not), the quantifiers $\forall$ and $\exists$ (forall and exists) the equality symbol $=$, and a certain set of function and relation symbols.

For example, the first order language of Peano arithmetic is defined by the constant 0 (zero) and function $s$ (successor) of one argument.

# First Order Logic

The first order language of Peano arithmetic is defined by the constant 0 (zero) and function $s$ (successor) of one argument.

The axioms of Peano arithmetic are:

1. $\forall x, y \;\; s(x) = s(y) \Rightarrow x = y$

2. $\forall x \;\; s(x) \neq 0$

3. For any formula $\Phi[x]$:

$$(\Phi[0] \wedge \forall x \; \Phi[x] \Rightarrow \Phi[s(x)]) \Rightarrow \forall x \; \Phi[x]$$

## Set Theory

In set theory we have only the single relation $\in$.

The axioms imply that there exists an empty set $\emptyset$.

$$\exists x \; \forall y \; y \notin x$$

For any set $x$ we have the set containing $x$.

We also have an infinite set

$$\exists x \; \emptyset \in x \wedge \forall y \;\; y \in x \Rightarrow \{y\} \in x$$

We can define the natural numbers, the integers, the rationals, the reals and all the structures of mathematics as particular sets within the universe of sets.

## The Universe $V$

The universe of all sets is typically written as $V$.

In the 1870s Cantor proposed to define $V$ by stating that any formula can be used to name a set.

$$\text{for any formula } \Phi[y]: \quad \exists x\ \forall y\ y \in x \Leftrightarrow \Phi[y]$$

This is called the axiom of **naive comprehension**.

Naive comprehension leads to Russell's paradox

$$\exists x\ \forall y\ \ y \in x \Leftrightarrow y \notin y$$

$$x \in x \Leftrightarrow x \notin x$$

## Nine ZFC Axioms (The First Four)

**Extensionality:** If two sets have the same members then they are the same set.

**Foundation:** $\in$ is a well-founded relation (every set contains a least member under the order $\in$).

**Restricted Comprehension:** For any $x \in V$, and any formula $\Phi[y]$, $V$ contains $\{y \in x : \Phi[y]\}$

**Infinity:** $V$ contains an infinite set:

$$\exists x \, \emptyset \in x \wedge \forall y \;\; y \in x \Rightarrow \{y\} \in x$$

## The Nine ZFC Axioms (The Rest)

**Power Set:** For any set $x \in V$ we have that $V$ contains the set of subsets of $x$, denote $\mathcal{P}(x)$.

**Pairing and Union:** We can form two element sets and take the union of the elements of a set.

**Replacement:** For any set $x \in V$, and any function $f$ from $x$ into $V$, the set $\{f(y) : \ y \in x\}$ is also in $V$.

**Choice:** If $(\forall x \exists y \ \Phi(x, y)) \Rightarrow \exists f \forall x \ \Phi(x, f(x))$.

## The Large Cardinal Axiom

A Grothendieck universe is a set $\tilde{V}$ with all the properties of $V$ but is a member of "the real"(larger) $V$.

The large cardinal axiom is equivalent to the statement that $V$ contains a Grothendieck universe.

This is equivalent to the statement that large cardinals exist.

Grothendieck assumed the existence of such a universe in proving certain classification theorems for topological spaces.

Large cardinal theory was a major topic in set theory for many decades.

## What is wrong with Set Theory?

The fundamental issue with set theory is that it ignores fact that mathematical statements involve a notion of grammaticality.

We can add two numbers but we cannot add a word to a number.

The representation of the natural numbers where zero is the empty set and $s(x)$ is $\{x\}$ is completely arbitrary.

The natural numbers (and all mathematical structures) should be defined in a way that is independent of any particular representation.

## Grammar and Isomorphism

The idea of isomorphic graphs seems intuitively clear.

For any two isomorphic graphs $G$ and $G'$ and for any **Grammatically Well Formed** statements $\Phi(g)$ about an arbitrary graph $g$ we have $\Phi(G) \Leftrightarrow \Phi(G')$.

## General First Order Logic Provides Grammar

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 $\tau$ 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 $\pi_2(e)$.

$\lambda$-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).

## Signature-Axiom Classes

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.

## The Set-Class Distinction

All of mathematics can be translated into set theory. A particular graph can be represented by a set.

However, any set can be taken to be a node in a graph. There are at least as many graphs as there are sets.

This implies that the collection of all graphs is not a set — it is a **proper class**, a subset of $V$ that is not a member of $V$.

## Functors: Functions Between Classes

We want a formal language with strict grammar that includes expressions for functions between classes.

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).

Neither set theory (Mizar) nor higher order logic (Isabelle/HOL) support this.

# The Substitution of Isomorphics

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

## 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 $\sigma$ and $\tau$ 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 $\tau$ has a $\tau$ symmetry group — the set of $\tau$-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 $\tau$-$\sigma$-cryptomorphic then the $\tau$ symmetry group of $x$ must be isomorphic (as a permutation group) to the $\sigma$ 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.

## Dependent Type Theory

For a type system to support the substitution of isomorphics we need types for signature-axiom classes.

While having a type "graph" seems natural in an object-oriented programming language, the type "planar graph" typically cannot be defined in the type system.

A typical object-oriented programming language supports signatures but not statically checked (compile time checked) axioms.

Dependent Type theory (Lean) supports statically checked signature-axiom types.

# Dependent Function Types

$\Pi_{x:\tau}\ \sigma[x]$ is the type of functions that maps $x$ of type $\tau$ to a value of type $\sigma[x]$.

$(\lambda\ x : \text{int}\ x + 5) : \Pi_{x:\text{int}}\ (>\ (x))$

# Dependent Function Type Inference Rules

$$\frac{\Gamma; x:\tau \vdash e[x]:\sigma[x]}{\Gamma \vdash (\lambda\ x:\tau\ e[x]):\Pi_{x:\tau}\ \sigma[x]}$$

$$\frac{\Gamma \vdash f:\Pi_{x:\tau}\ \sigma[x] \quad \Gamma \vdash e:\tau}{\Gamma \vdash f(e):\sigma[e]}$$

# Dependent Pair Types

$\Sigma_{x:\tau}\ \sigma[x]$ is the type of pairs $(x, y)$ with $x : \tau$ and $y : \sigma[x]$.

For $x : \mathrm{int}$ we have $(x, x + 5) : \Sigma_{x:\mathrm{int}}\ (> (x))$.

# Dependent Pair Type Inference Rules

$$\frac{\Gamma;\vdash e:\tau \qquad \Gamma;\vdash w:\sigma[e]}{\Gamma \vdash (e,w):\Sigma_{x:\tau}\ \sigma[x]}$$

$$\frac{\Gamma \vdash e:\Sigma_{x:\tau}\ \sigma[x]}{\Gamma \vdash \pi_1(e):\tau \qquad \Gamma \vdash \pi_2(e):\sigma[\pi_1(e)]}$$

## Propositions as Types

In Lean propositions (Boolean formulas) are represented by types.

$\forall x : \tau\ \Phi[x]$ is represented by the type $\Pi_{x:\tau}\ \sigma[x]$.

$\exists x : \tau\ \Phi[x]$ is represented by the type $\Sigma_{x:\tau}\ \sigma[x]$.

Types are classified into universes $U_0$, $U_1$, $U_2$, ... where type in $U_0$ are "propositions", types in $U_1$ are "sets", types in $U_2$ are "classes", and types $U_i$ for $i > 2$ are ever larger Grothendieck universes.

## Propositions as Types: the Good News

Reducing all of mathematics to type inference rules is extremely compact (elegant?).

Much more compact than the inference rules of first order logic plus the nine axioms of ZFC plus a large cardinal axiom.

It also has the effect of supporting signature-axiom classes as first class types where the axioms are statically checked (explained below).

The validity of the substitution of isomorphics was proved for Martin-Löf type theory, from which Lean is derived, by Hofmann and Streicher in 1995.

## The Bad News: Constructivism

A proposition is a type whose elements are the proofs of the statement represented by the type.

A proposition type is "true" if it is "inhabited" — there exists an element of the type (proof of the proposition).

The proposition $\forall\ P : \mathrm{Prop}\ P \vee \neg P$ (often called the "excluded middle") is rejected.

Proof by contradiction is not allowed.

The distinction between truth and provability is lost (Gödel's incompleteness theorems).

## Getting Around Constructivism

Mathematicians that use Lean get around constructivist limitations by using extensions of constructive logic that provide the excluded middle and the axiom of choice.

However, it turns out there is no need for propositions as types or constructivism.

There is no problem with simply using Boolean propositions from the start.

## Semantics

Constructive logic is specified by inference rules.

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

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

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.

# Recall Multi-Sorted Logic

A multi-sorted signature consists of a set of "sorts" and a specification $f : \tau$ of a type $\tau$ 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

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

## 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.

# Recall Dependent Pair Types

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

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 $y \in \tau[x]$.

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

## Recall 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 $\sigma$ 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 supporting signature-axiom classes as types and supporting the substitution of isomorphics.

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 Substitution of Isomorphics

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

# END