Primer on Proof/Type Theory

CAS CS 392: Rust, in Theory and in Practice

March 18, 2025 (Lecture 13)

Outline

What is logic?

A **logic** is a formalization of (a kind of) language. To define a logic we need:

- *syntax*: what can I write down in my logic?
- *semantics*: what do those things mean?
- *proof system*: (optional) how can I derive new things from old things?

(This is also what you need for a PL, replacing proof system with type system)

Propositions

Some facts seems contingent on the world:

- It is raining
- It is cold
- The house is on fire

Propositions

Some facts seems contingent on the world:

- It is raining
- It is cold
- The house is on fire

Other facts seems unassailable:

- If it is raining and it is cold then it is raining
- Either the house is on fire or the house is not on fire

Propositions

Some facts seems contingent on the world:

- It is raining
- It is cold
- The house is on fire

Other facts seems unassailable:

- If it is raining and it is cold then it is raining
- Either the house is on fire or the house is not on fire

We're interested in why this is the case

Boolean Connectives

$$P \wedge Q$$

Proposition logic (a.k.a. sentential logic) is the logic of **Boolean connectives**. It's motivated by questions like:

- Given a collection of true statements, when is a *compound* statement made of them them also true?
- How to connectives affect our ability to derive new facts from old ones?
- What connectives are required, which are definable in terms of others?

Conditions in PLs

We can think of propositional logic as the logic of *conditions* in PLs:

```
let p1: bool = true;
let p2: bool = true;
let p3: bool = false;
assert!(p1 && !p2 || !p3);
```

Propositional logic can help us reason about:

- when conditions will hold (evalutation)
- if it's possible for a condition to hold (satisfiability)
- what connectives are necessary (functional completeness)

Proposition Variables

$$P \wedge Q, P \vee Q, P \rightarrow Q$$

Propositional variables are "stand-ins" for "atomic" statements. We can think of them like variables from algebra or calculus

Proposition Variables

$$P \wedge Q, P \vee Q, P \rightarrow Q$$

Propositional variables are "stand-ins" for "atomic" statements. We can think of them like variables from algebra or calculus

We're interested in how propositions *interact* with respect to connectives. *We don't care what the actual propositions are*

Propositional Formulas

Fix a set of propositional variables $\mathcal{V}$. The set of **propositional formulas** $\mathcal{P}_{\mathcal{V}}$ is defined inductively:

- All propositional variables from $\mathcal{V}$ are propositional formulas
- $\bot$ is a propositional formula
- If ϕ and ψ are propositional formulas then so are $(\phi \vee \psi)$ and $(\phi \wedge \psi)$ and $(\phi \to \psi)$

We pronounce $\wedge$ as "and", $\vee$ as "or", $\to$ as "implies", and $\bot$ as "false".

Annoying Technicalities

- We assume an infinite set of "natural" propositional variables, e.g, P, Q, R, X, Y, Z...
- We elide parentheses by according to the table below (operators are given in order of increasing precedence)

operator	associativity
$\rightarrow$	right
$\vee$	left
$\wedge$	left

Induction/Recursion on Formulas

Propositional formulas are defined inductively

Induction/Recursion on Formulas

Propositional formulas are defined inductively

If we want to write a function on *all* formulas, we have to define it using *structural recursion* on formulas

Induction/Recursion on Formulas

Propositional formulas are defined inductively

If we want to write a function on *all* formulas, we have to define it using *structural recursion* on formulas

And if we want to *prove* something about all formulas, we have to use structural induction

Induction/Recursion on Formulas

Propositional formulas are defined inductively

If we want to write a function on *all* formulas, we have to define it using *structural recursion* on formulas

And if we want to *prove* something about all formulas, we have to use structural induction

Exercise. Implement a function which determines the number of connectives in a formula

Exericse. Prove by induction that all formulas have the same number of left parentheses as right parentheses

Model Theory of Classical Propositional Logic

A **valuation** of the propositional variables $\mathcal{V}$ is a function $\tau : \mathcal{V} \to \{0, 1\}$

We read 0 as "false" and 1 as "true"

A truth assignment can be lifted to a **evaluation function** of the form $\overline{\tau} : \mathcal{P}_{\mathcal{V}} \to \{0, 1\}$ inductively:

- $\overline{\tau}(P) = \tau(P)$ when $P \in \mathcal{V}$
- $\overline{\tau}(\bot) = 0$
- $\overline{\tau}(\phi \wedge \psi) = \overline{\tau}(\phi)\overline{\tau}(\psi)$
- $\overline{\tau}(\phi \vee \psi) = \max(\overline{\tau}(\phi), \overline{\tau}(\psi))$
- $\overline{\tau}(\phi \to \psi) = \max(1 - \overline{\tau}(\phi), \overline{\tau}(\psi))$

What about negation?

Negation is definable as $\neg\phi \equiv \phi \rightarrow \bot$

Entailment

A set of formulas Γ **entails** a formula ϕ, written $\Gamma \models \phi$ if $\overline{\tau}(\Gamma) = 1$ implies $\overline{\tau}(\phi) = 1$ for any valuation τ

Example: $P, Q \vdash \neg(\neg P \vee \neg Q)$

Outline

What is Proof Theory?

Proof Theory is a subfield of mathematical logic which takes *proof* as an object of formal study

There are many ways to formalize the notion of *proof*, we'll use **sequent calculi**

Proof Systems and Inference Rules

A **proof system** $\mathcal{S}$ is given in terms of a class of **judgments** $\mathcal{J}$, and consists of **inference rules** of the form:

$$\frac{J_1 \qquad J_2 \qquad \ldots \qquad J_k}{J_{k+1}}$$

where $k \geqslant 0$ and $J_1, \ldots, J_{k+1}$ are judgments from $\mathcal{J}$. We read an inference rule as "if the judgments $J_1, \ldots, J_k$ hold, then J_{k+1} follows"

Judgments above the vertical line are called **premises** and judgment below is called the **conclusion**. We call in inference rule without any premises an **axiom**

Derivations

A **derivation** $\mathcal{D}$ in a proof system $\mathcal{P}$ over the judgments $\mathcal{J}$ is a tree with the following properties:

- the nodes of $\mathcal{D}$ are judgments from $\mathcal{J}$
- A node J has the children $J_1, \ldots, J_k$ only if $(J_1, \ldots, J_k)/J_{k+1}$ is an inference rule of $\mathcal{P}$.

Trees are also inductively defined so we can do *induction on derivations*

(Contrived) Example: Binary Strings

$$\frac{}{0} \qquad \frac{}{1} \qquad \frac{S_1 \qquad S_2}{S_1 \circ S_2}$$

Consider the above simple proof system with binary strings as judgments, and where '$\circ$' is string concatenation

Sequents

Fix a language $\mathcal{L}$ of which consists of statements. A **sequent** is a judgment of the form

$$\phi_1, \ldots, \phi_k \vdash \psi_1 \ldots, \psi_l$$

where $k \geqslant 0$ and $l \geqslant 0$ and $\phi_1, \ldots, \phi_k, \psi_1, \ldots, \psi_l$ are statements in $\mathcal{L}$. We read a sequent as saying that "if $\phi_1, \ldots, \phi_k$ are hold, then one of $\psi_1, \ldots, \psi_k$ hold"

Sequents

Fix a language $\mathcal{L}$ of which consists of statements. A **sequent** is a judgment of the form

$$\phi_1, \ldots, \phi_k \vdash \psi_1 \ldots, \psi_l$$

where $k \geqslant 0$ and $l \geqslant 0$ and $\phi_1, \ldots, \phi_k, \psi_1, \ldots, \psi_l$ are statements in $\mathcal{L}$. We read a sequent as saying that "if $\phi_1, \ldots, \phi_k$ are hold, then one of $\psi_1, \ldots, \psi_k$ hold"

We call the formulas to the left of the '$\vdash$' the **context**

Sequents

Fix a language $\mathcal{L}$ of which consists of statements. A **sequent** is a judgment of the form

$$\phi_1, \ldots, \phi_k \vdash \psi_1 \ldots, \psi_l$$

where $k \geqslant 0$ and $l \geqslant 0$ and $\phi_1, \ldots, \phi_k, \psi_1, \ldots, \psi_l$ are statements in $\mathcal{L}$. We read a sequent as saying that "if $\phi_1, \ldots, \phi_k$ are hold, then one of $\psi_1, \ldots, \psi_k$ hold"

We call the formulas to the left of the '$\vdash$' the **context**

For technical reasons, we will always assume $l = 1$

(Intuitionistic) Propositional Logic

Assumptions and Falsity:

$$\frac{}{\Delta, \phi, \Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}$$

(Intuitionistic) Propositional Logic

Assumptions and Falsity:

$$\frac{}{\Delta, \phi, \Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}$$

Conjunction:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

(Intuitionistic) Propositional Logic

Assumptions and Falsity:

$$\frac{}{\Delta, \phi, \Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}$$

Conjunction:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

Disjunction:

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \phi \vee \psi \qquad \Gamma, \phi \vdash \xi \qquad \Gamma, \psi \vdash \xi}{\Gamma \vdash \xi}$$

(Intuitionistic) Propositional Logic

Assumptions and Falsity:

$$\frac{}{\Delta, \phi, \Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}$$

Conjunction:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

Disjunction:

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \phi \vee \psi \qquad \Gamma, \phi \vdash \xi \qquad \Gamma, \psi \vdash \xi}{\Gamma \vdash \xi}$$

Implication:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \qquad \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

Example Derivation

$$P \vdash (P \to \bot) \to \bot$$

Soundness and Completeness

Theorem. $\Gamma \vdash \phi$ if and only if $\Gamma \models \phi$

Soundness and Completeness

Theorem. $\Gamma \vdash \phi$ if and only if $\Gamma \models \phi$

Soundness (the "only if" part) says that everything (conditionally) provable is (conditionally) true

Soundness and Completeness

Theorem. $\Gamma \vdash \phi$ if and only if $\Gamma \models \phi$

Soundness (the "only if" part) says that everything (conditionally) provable is (conditionally) true

Completeness (the "if" part) says that everything (conditionally) true is (conditionally) provable

Soundness and Completeness

Theorem. $\Gamma \vdash \phi$ if and only if $\Gamma \models \phi$

Soundness (the "only if" part) says that everything (conditionally) provable is (conditionally) true

Completeness (the "if" part) says that everything (conditionally) true is (conditionally) provable

Completeness does not actually hold unless we include one more axiom:

$$\frac{}{\phi \vee (\phi \to \bot)}$$

Anything else?

There is quite a bit that we're not going to cover. All this is to give you a taste and to motivate our discussion on Thursday.

If you want more, there is a logic course in the math department (CAS MA 531) and in the philosophy department (CAS PH 360) and (eventually) in the CS department.

(If you're *really* interested, talk to me. I have many undergraduate project in logic I'd like to work on)

Outline

What is a Type?

```
let f : int -> int = fun x -> x + 1
let g : int -> bool = fun x -> x > 0
let compose : int -> bool = fun x -> g (f (x))
(* g (f (x)) will give use a compile-time error *)
```

A **type** is a **syntactic construct** which annotates and describes the behavior and compositionality of a program

What is a Type?

```
let f : int -> int = fun x -> x + 1
let g : int -> bool = fun x -> x > 0
let compose : int -> bool = fun x -> g (f (x))
(* g (f (x)) will give use a compile-time error *)
```

A **type** is a **syntactic construct** which annotates and describes the behavior and compositionality of a program

The "syntactic" part is important, *we write the annotation*

What is a Type?

```
let f : int -> int = fun x -> x + 1
let g : int -> bool = fun x -> x > 0
let compose : int -> bool = fun x -> g (f (x))
(* g (f (x)) will give use a compile-time error *)
```

A **type** is a **syntactic construct** which annotates and describes the behavior and compositionality of a program

The "syntactic" part is important, *we write the annotation*

This allows type-checking to happen at *compile time* (we don't need semantic information)

Simply Typed Lambda Calculus

$$f : A \to B, g : B \to C \vdash \lambda x.g(fx) : A \to C$$

The **simply typed lambda calculus (STLC)** is a type theory built on top of the untyped lambda calculus

It was created by Alonzo Church in the 1930s (after his logical system based on the lambda calculus was shown to be inconsistent)

Recall: The Untyped Lambda Calculus

Fix a set of variables $\mathcal{V}$. The collection of **lambda terms** Λ is defined inductively:

- $\mathcal{V} \subset \Lambda$ (variables are lambda terms)
- $M, N \in \Lambda$ is $(MN) \in \Lambda$ (applications are lambda terms)
- $x \in \mathcal{V}$ and $M \in \Lambda$ implies $\lambda x.M \in \Lambda$ (abstractions are lambda terms)

Recall: The Untyped Lambda Calculus

Fix a set of variables $\mathcal{V}$. The collection of **lambda terms** Λ is defined inductively:

- $\mathcal{V} \subset \Lambda$ (variables are lambda terms)
- $M, N \in \Lambda$ is $(MN) \in \Lambda$ (applications are lambda terms)
- $x \in \mathcal{V}$ and $M \in \Lambda$ implies $\lambda x.M \in \Lambda$ (abstractions are lambda terms)

The small-step semantics of the λ-calculus are given by:

$$\frac{M \longrightarrow N}{\lambda x.M \longrightarrow \lambda x.N} \qquad \frac{M \longrightarrow M'}{MN \longrightarrow M'N} \qquad \frac{N \longrightarrow N'}{MN \longrightarrow MN'}$$

β-reduction:

$$\frac{}{(\lambda x.M)N \longrightarrow M[N/x]}$$

Recall: The Untyped Lambda Calculus

Fix a set of variables $\mathcal{V}$. The collection of **lambda terms** Λ is defined inductively:

- $\mathcal{V} \subset \Lambda$ (variables are lambda terms)
- $M, N \in \Lambda$ is $(MN) \in \Lambda$ (applications are lambda terms)
- $x \in \mathcal{V}$ and $M \in \Lambda$ implies $\lambda x.M \in \Lambda$ (abstractions are lambda terms)

The small-step semantics of the λ-calculus are given by:

$$\frac{M \longrightarrow N}{\lambda x.M \longrightarrow \lambda x.N} \qquad \frac{M \longrightarrow M'}{MN \longrightarrow M'N} \qquad \frac{N \longrightarrow N'}{MN \longrightarrow MN'}$$

β-reduction:

$$\frac{}{(\lambda x.M)N \longrightarrow M[N/x]}$$

The term M is a **normal form** if there is no N such that $M \longrightarrow N$

Simple Types

Fix a set of base types $\mathcal{B}$. The set of **simple types** $\mathcal{T}_{\mathcal{B}}$ is defined inductively:

- $\mathcal{B} \subset \mathcal{T}$ (base types are simple types)
- $\bot \in \mathcal{T}$ (the empty type is a simple type)
- $A, B \in \mathcal{T}$ implies $(A \to B) \in \mathcal{T}$ (function types are simple types)

Simple Types

Fix a set of base types $\mathcal{B}$. The set of **simple types** $\mathcal{T}_{\mathcal{B}}$ is defined inductively:

- $\mathcal{B} \subset \mathcal{T}$ (base types are simple types)
- $\bot \in \mathcal{T}$ (the empty type is a simple type)
- $A, B \in \mathcal{T}$ implies $(A \to B) \in \mathcal{T}$ (function types are simple types)

This should look very familiar, it's similar to $\mathcal{P}_{\mathcal{V}}$ but without conjunction and disjunction (more on that later)

Typing Judgments

$$x_1 : A_1, \ldots, x_k : A_k \vdash M : A$$

A **typing statement** $(M : A)$ is a λ-term together with a simple type, and reads "M is of type A"

Typing Judgments

$$x_1 : A_1, \ldots, x_k : A_k \vdash M : A$$

A **typing statement** $(M : A)$ is a λ-term together with a simple type, and reads "M is of type A"

A **variable declaration** $(x : B)$ the a typing statement in which the lambda term is a variable is a variable

Typing Judgments

$$x_1 : A_1, \ldots, x_k : A_k \vdash M : A$$

A **typing statement** $(M : A)$ is a λ-term together with a simple type, and reads "M is of type A"

A **variable declaration** $(x : B)$ the a typing statement in which the lambda term is a variable is a variable

A **typing judgment** is a type sequent whose judgments are typing statements and whose context is made up of variable declarations

Simply-Types Lambda Calculus

Assumptions and Falsity:

$$\frac{}{\Delta, x : A, \Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathsf{explode}(M) : A}$$

Simply-Types Lambda Calculus

Assumptions and Falsity:

$$\frac{}{\Delta, x : A, \Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathsf{explode}(M) : A}$$

Implication:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Example Derivation

$$x : B \vdash \lambda f.fx : (B \to \bot) \to \bot$$

Meta-Theoretic Properties

Progress: If $\Gamma \vdash M : A$ and M is not a normal form, then there is as a term N such that $M \longrightarrow N$

Preservation: If $\Gamma \vdash M : A$ and $M \longrightarrow N$, then $\Gamma \vdash N : A$

Normalization: If $\Gamma \vdash M : A$, then M has a normal form

Strong normalization: If $\Gamma \vdash M : A$ then M does not appear in any infinite reductions

We will focus on the first two in the back-half of the course

Curry-Howard Isomorphism (First Glance)

Implication (STLC):

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Implication (Propositional Logic):

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \qquad \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

Type Theory is Logic + Computation

Outline

Tasks

- Write an `enum` for propositional formulas and implement a function an evaluation function on expressions
- Finish assignment 5