Propositional Logic II: Meta-Theory

Type Theory and Mechanized Reasoning

Lecture 5

CAS CS 400

Introduction

Administrivia

- Assignment 1 is officially due 11:59PM tomorrow.
- Assignment 2 will be released tomorrow as well.
- If you haven't gotten Agda set up on your machine, please do so ASAP.

Objectives

1. Use propositional logic as a setting for learning important concepts and terms in logic.

2. Look at what we can say *about* logic, not just with logic.

3. Being discussing normal forms, which are used for representing formulas in the context of CS.

Unicode cheatsheet

→ is `\->`

ℕ is `\bN`

× is `\times`

∧ is `\and`

λ is `\lambda`

∨ is `\or`

¬ is `\neg`

$\_^{p}$ is `\^p`

∷ is `\::`

⊎ is `\uplus`

Practice Problem

Write a function ***replace-implies*** *which recursively replaces* $A \to B$ *in* ***Form*** *with* $\neg A \vee B$.

(see the starter code for more details)

Agda Tutorial: Pattern Matching

Standard Pattern Matching

```
my-pred : ℕ → ℕ
my-pred zero = zero
my-pred (suc n) = n
```

We define a function in multiple lines in which the argument varies by pattern.

We can automatically split on a variable using **C-c C-c**.

Anonymous Functions

```
foo : List ℕ
foo = map (λ { x → x * 10 }) (1 :: 2 :: 3 :: [])

my-pred : ℕ → ℕ
my-pred = λ { zero → zero
            ; (suc n) → n }
```

As in most languages, we have lambdas.

We can even pattern match within lambdas.

Case Expressions

```
case_of_ : {A B : Set} → A → (A → B) → B
case x of f = f x
```

Because of this, we don't need special **case** notation.

Casing is just function application(!)

Simple Example

```
my-pred : ℕ → ℕ
my-pred n = case n of λ
  { zero → zero
  ; (suc n) → n }
```

Because of this, we can get OCaml-like function definitions.

This is nice for simple functions, but it has its drawbacks...

With Abstractions

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) with p x
filter p (x ∷ xs)    | true  = x ∷ filter p xs
filter p (x ∷ xs)    | false = filter p xs
```

The slightly more canonical way of pattern matching on intermediate values is by using **with abstractions.**

demo

Understanding Check

*Write a function **split** of type*

```
{A : Set} → {n : ℕ} → Vec A n → Fin n → List A × List A
```

which split a vector into to lists given an index.

(the function Data.Vec.toList will be useful here.)

Propositional Logic: Recap

Recall: Boolean Connectives

$$x \boxed{\wedge} y$$

conjunction (and)

Propositional logic is the study of logical (Boolean) connectives.

» *What do connectives mean? How do they affect truth?*
» *What connectives exists? How many do we need?*
» *How do connectives interact?*

Recall: Programming Conditionals

```
if is_raining and not is_warm:
    # some code
```

Propositional logic is the study of **Bools.**

- *When can I replace one conditional with another?*
- *Why is there only **and**, **or**, and **not**?*
- *Why can conditionals short-circuit?*

Recall: Propositional Variables

$$\boxed{x} \land \boxed{y}$$

propositional variables

Recall: Propositional Variables

$$\boxed{x} \land \boxed{y}$$

propositional variables

We're interested in how propositions *interact* with respect to connectives.

Recall: Propositional Variables

$$\boxed{x} \wedge \boxed{y}$$

propositional variables

We're interested in how propositions *interact* with respect to connectives.

We don't care what the actual propositions are.

Recall: Propositional Variables

$$\boxed{x} \wedge \boxed{y}$$

propositional variables

We're interested in how propositions *interact* with respect to connectives.

We don't care what the actual propositions are.

We think of these variables in the same was as we think of variables in algebra.*

Recall: Propositional Variables

$$x \land y$$

propositional variables

We're interested in how propositions *interact* with respect to connectives.

We don't care what the actual propositions are.

We think of these variables in the same was as we think of variables in algebra.*

*We will assume a countable number of variable symbols like in algebra

Syntax: In Agda

```
data Formula : Set where
  _ᵖ : String → Formula
  ¬ᵖ_ : Formula → Formula
  _∧ᵖ_ : Formula → Formula → Formula
  _∨ᵖ_ : Formula → Formula → Formula
  _→ᵖ_ : Formula → Formula → Formula
```

The tree structure of formulas is implicit in it being an ADT.

At a high level

At a high level

Given an expression $3x + (2y \times 6h^2)$ and values for x, y, and h, we can compute the value of the entire expression.

At a high level

Given an expression $3x + (2y \times 6h^2)$ and values for x, y, and h, we can compute the value of the entire expression.

Given a *formula* $\neg(x \wedge y) \vee (x \rightarrow \neg z)$ if we know the values of x, y, and z, we can compute the value of the the formula.

At a high level

Given an expression $3x + (2y \times 6h^2)$ and values for x, y, and h, we can compute the value of the entire expression.

Given a *formula* $\neg(x \wedge y) \vee (x \rightarrow \neg z)$ if we know the values of x, y, and z, we can compute the value of the the formula.

We think of a propositional variable as having the values of either **true** or **false**.

Valuations (In Mathematical English)

Valuations (In Mathematical English)

Definition. A valuation is a function from all possible propositional values to $\{\mathsf{true}, \mathsf{false}\}$.

Valuations (In Mathematical English)

Definition. A valuation is a function from all possible propositional values to $\{\mathsf{true}, \mathsf{false}\}$.

A valuation v is like a state of affairs.

Valuations (In Mathematical English)

Definition. A valuation is a function from all possible propositional values to $\{\mathsf{true}, \mathsf{false}\}$.

A valuation v is like a state of affairs.

The idea. *If we know the state of affairs, we can determine the truth or falsity of any statement.*

Partial Valuations

$$v(x) = \text{true}$$
$$v(y) = \text{false}$$
$$v(z) = \text{false}$$
$$v(\_) = \text{false}$$

$$(x \wedge y) \vee z$$

Note. We will typically only care about a small collection of variables.

We can assume all unspecified variables are assigned to be false.

Evaluation: At a High Level

Evaluation: At a High Level

A valuation function can be lifted from propositional variables to arbitrary formulas.

Evaluation: At a High Level

A valuation function can be lifted from propositional variables to arbitrary formulas.

This is where we give a formula meaning with respect to a state of the affairs.

Evaluation: At a High Level

A valuation function can be lifted from propositional variables to arbitrary formulas.

This is where we give a formula meaning with respect to a state of the affairs.

We have to say what we want the truth of a statement to be based on its constituent parts.

demo

Evaluation: In English

Evaluation: In English

The evaluation function we wrote in Agda for a valuation v is written $\overline{v}$.

Evaluation: In English

The evaluation function we wrote in Agda for a valuation v is written $\bar{v}$.

We say that a valuation v **makes** ϕ **true** if $\bar{v}(\phi) = \mathsf{true}$.

Evaluation: In English

The evaluation function we wrote in Agda for a valuation v is written $\overline{v}$.

We say that a valuation v **makes** ϕ **true** if $\overline{v}(\phi) = \mathsf{true}$.

I leave it as an exercise to write out a "mathy" definition, but the shape will be similar, e.g.,

$$\overline{v}(\neg P) = \begin{cases} \mathsf{false} & \overline{v}(P) = \mathsf{true} \\ \mathsf{true} & \texttt{otherwise} \end{cases}$$

Propositional Logic: Semantic Notions

Validity and Satisfiability

Validity and Satisfiability

Definition. A formula ϕ is **valid** (written $\models \phi$) if *every* valuation v makes ϕ true, e.g.

$$x \vee \neg x$$

Validity and Satisfiability

Definition. A formula ϕ is **valid** (written $\models \phi$) if *every* valuation v makes ϕ true, e.g.

$$x \vee \neg x$$

We say that ϕ is a **tautology**.

Validity and Satisfiability

Definition. A formula ϕ is **valid** (written $\models \phi$) if *every* valuation v makes ϕ true, e.g.

$$x \vee \neg x$$

We say that ϕ is a **tautology.**

Definition. A formula ϕ is **satisfiable** if there is *some* valuation which makes ϕ true, e.g.

$$x \vee y$$

Example

Proposition. $A \vee \neg A$ is a tautology for any formula A.

Proof. Let's try it.

Validity and Satisfiability

Theorem. ϕ is a tautology if and only if $\neg\phi$ is unsatisfiable.

Proof. Let's try it.

Entailment

Definition. A set of formulas Γ **entails** ϕ (written $\Gamma \vDash \phi$) if *every* valuation which make *every* formula in Γ true, also makes ϕ true.

Example. $\{x \to y, y \to z\} \vDash x \to z$

The Deduction Theorem

Theorem. $\Gamma \cup \{\phi\} \vDash \psi$ if and only if $\Gamma \vDash \phi \to \psi$.

Proof. Let's try it.

Understanding Check

True *or* ***False:*** *If* $\Gamma \vDash A \vee B$ *then* $\Gamma \vDash A$ *or* $\Gamma \vDash B$.

Logical Equivalence

Logical Equivalence

Definition. Formulas ϕ and ψ are **logically equivalent** (written $\phi \equiv \psi$) if $\{\phi\} \vDash \psi$ and $\{\psi\} \vDash \phi$. That is, all valuations agree on ϕ and ψ.

Logical Equivalence

Definition. Formulas ϕ and ψ are **logically equivalent** (written $\phi \equiv \psi$) if $\{\phi\} \vDash \psi$ and $\{\psi\} \vDash \phi$. That is, all valuations agree on ϕ and ψ.

Example. $x \wedge y$ is logically equivalent to $y \wedge x$.

Logical Equivalence

Definition. Formulas ϕ and ψ are **logically equivalent** (written $\phi \equiv \psi$) if $\{\phi\} \vDash \psi$ and $\{\psi\} \vDash \phi$. That is, all valuations agree on ϕ and ψ.

Example. $x \wedge y$ is logically equivalent to $y \wedge x$.

Logical equivalence captures when boolean conditionals express the same thing.*

Logical Equivalence

Definition. Formulas ϕ and ψ are **logically equivalent** (written $\phi \equiv \psi$) if $\{\phi\} \vDash \psi$ and $\{\psi\} \vDash \phi$. That is, all valuations agree on ϕ and ψ.

Example. $x \land y$ is logically equivalent to $y \land x$.

Logical equivalence captures when boolean conditionals express the same thing.*

*This does not account for short-circuiting.

Understanding Check

Show that $A \wedge B \equiv \neg(\neg A \vee \neg B)$.

Standard Logical Equivalences

Distribution:

$$A \vee (B \wedge C) = (A \vee C) \wedge (B \vee C) \qquad A \wedge (B \vee C) = (A \wedge C) \vee (B \wedge C)$$

DeMorgan's Law:

$$A \wedge B \equiv \neg(\neg A \vee \neg B) \qquad A \vee B \equiv \neg(\neg A \vee \neg B)$$

Double Negation Elimination: $\neg(\neg A) \equiv A$

Law of the Excluded Middle: $A \vee \neg A$

And many more...

Propositional Logic: Functional Completeness

Other Connectives

Other Connectives

Consider the exclusive-or operations $P \otimes Q$.

Other Connectives

Consider the exclusive-or operations $P \otimes Q$.

Why didn't we include this in our presentation of propositional logic?

Other Connectives

Consider the exclusive–or operations $P \otimes Q$.

Why didn't we include this in our presentation of propositional logic?

Theorem. For any propositions P and Q, $P \otimes Q$ is logically equivalent to $(P \wedge Q) \vee (\neg P \vee \neg Q)$.

Boolean Functions

Boolean Functions

Definition. An n–variate boolean function is a function of the form

$$f : \{\text{true}, \text{false}\}^n \to \{\text{true}, \text{false}\}$$

Boolean Functions

Definition. An n–variate boolean function is a function of the form

$$f : \{\mathsf{true}, \mathsf{false}\}^n \to \{\mathsf{true}, \mathsf{false}\}$$

f is **represented** by a formula ϕ on with propositional variables $x_1, \ldots, x_n$ if for any valuation v,

$$f(v(x_1), \ldots, v(x_n)) = \overline{v}(\phi)$$

Boolean Functions

Definition. An n–variate boolean function is a function of the form

$$f : \{\text{true}, \text{false}\}^n \to \{\text{true}, \text{false}\}$$

f is **represented** by a formula ϕ on with propositional variables $x_1, \ldots, x_n$ if for any valuation v,

$$f(v(x_1), \ldots, v(x_n)) = \overline{v}(\phi)$$

We can think of arbitrary connectives as boolean functions.

Functional Completeness

Theorem. Every n–variate boolean function is represented by a formula.

Proof. Let's try it.

Complete Sets of Connectives

A set of connectives is **complete** if they used to represent any boolean function.

Theorem. $\{\neg, \vee\}$ is complete.

Proof. Let's try it.

Incomplete Sets

Theorem. $\{\rightarrow\}$ is not complete.

Understanding Check

Show that $\{\text{NAND}\}$ *where* $A \text{ NAND } B \equiv \neg(A \wedge B)$.

Normal Forms

Motivation

We can get away with single connective, it is more useful to have a formula in a simple form.

There are many normal forms for formulas, but we will consider one primary form: **Conjunctive Normal Form (CNF)**, e.g.

$$(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_1 \vee x_4) \wedge (x_4 \wedge \neg x_5)$$

Motivation

We can get away with single connective, it is more useful to have a formula in a simple form.

There are many normal forms for formulas, but we will consider one primary form: **Conjunctive Normal Form (CNF)**, e.g.

literals $(\boxed{\neg x_1} \vee x_2) \wedge (\neg x_3 \vee \neg x_1 \vee \boxed{x_4}) \wedge (x_4 \wedge \neg x_5)$

Motivation

We can get away with single connective, it is more useful to have a formula in a simple form.

There are many normal forms for formulas, but we will consider one primary form: **Conjunctive Normal Form (CNF)**, e.g.

literals $(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_1 \vee x_4) \wedge (x_4 \wedge \neg x_5)$

clause

Definitions

A **literal** is a propositional variable or the negation of a propositional variable, e.g. x or $\neg y$.

A k–**clause** is the disjunction (or) of n literals.

A k–**CNF** formula is a conjunction (and) of k–clauses.

A **CNF** formula is a conjunction of clauses of any size.

Why CNFs?

CNFs are the basis of SAT solvers, algorithms that try to determine the satisfiability of propositional formula.

They are easier to represent in programs, and easier to design algorithms for.

Next Time

Theorem. Every formula is logically equivalent to a CNF formula.