Propositional Proofs

Type Theory and Mechanized Reasoning
Lecture 9

CAS CS 400

Introduction

Administrivia

Homework 3 is due on Thursday 11:59PM. Homework 4 will be released on Thursday (Tomorrow).

I updated the syllabus for the course to include some information about the final project.

We'll take a bit of time today to put together groups for the final project.

Objectives

Introduce resolution as one way of verifying that a CNF formula is unsatisfiable, and connect this to DPLL.

Introduce Gentzen-style proofs as a way of verifying that an arbitrary formulas is valid.

Look at soundness and completeness.

Agda Tutorial: The Empty Type

Definition

```
data Empty : Set where
```

Definition

```
data Empty : Set where
```

The Empty type has no constructors.

Definition

```
data Empty : Set where
```

The Empty type has no constructors.

Fact. There is no value we could write with the type **Empty**.

Definition

```
data Empty : Set where
```

`The Empty type has no constructors.`

Fact. `There is no value we could write with the type` **`Empty`**`.`

`This fact is surprisingly useful.`

Inheriting Emptiness

```
IsEmpty : Set -> Set
IsEmpty A = A -> Empty
```

Inheriting Emptiness

```
IsEmpty : Set -> Set
IsEmpty A = A -> Empty
```

Fact. If there is a function from **A** to **Empty**, then **A** has no values.

Inheriting Emptiness

```
IsEmpty : Set -> Set
IsEmpty A = A -> Empty
```

Fact. If there is a function from **A** to **Empty**, then **A** has no values.

We can "prove" that a type **A** has no values by construction a *function* from **A** to **Empty**.

Empty is Empty

```
empty-is-empty : IsEmpty Empty
empty-is-empty found-one = found-one
```

This is just the identity function.

But it indicates that the definition could be reasonable.

A More Interesting Example

```
data NotWell : Set where
  cannot : NotWell -> NotWell

emp : IsEmpty NotWell
emp (cannot but-I-did) = emp but-I-did
```

We can't create a value of type **NotWell.**

If we could we could use it to build an value of the empty type.

More Inheriting Emptiness

```
isEmpty-Prod :
  {A : Set} ->
  {B : Set} ->
  IsEmpty A ->
  IsEmpty B ->
  IsEmpty (Either A B)
isEmpty-Prod f g (left x) = f x
isEmpty-Prod f g (right x) = g x
```

***Either A B** is empty if **A** and **B** are.*

Verifying Truth

```
IsTrue : Bool -> Set
IsTrue true = Unit
IsTrue false = Empty
```

We can use dependent types and **Empty** to "lift" truth to the types.

Principle of Explosion

```
explode : {A : Set} -> Empty -> A
explode ()
```

If I *could* create a value of the empty type, I could create anything I want.

Note. If a type is empty, then Agda doesn't need to pattern match on it.

Principle of Explosion

```
explode : {A : Set} -> Empty -> A
explode ()
```

This indicates there are no patterns

If I *could* create a value of the empty type, I could create anything I want.

Note. If a type is empty, then Agda doesn't need to pattern match on it.

Understanding Check

Write a function of type

```
{A : Set} -> A -> IsEmpty (IsEmpty A)
```

How do we "read" this type?

Proof Systems

Recap: Satisfiability

Recap: Satisfiability

Definition. A formula ϕ is **satisfiable** if there is *some* **assignment** which makes ϕ true.

Recap: Satisfiability

Definition. A formula ϕ is **satisfiable** if there is *some* **assignment** which makes ϕ true.

Example. $(x \vee y) \wedge (x \vee \neg y)$ is satisfied by $\{x, \neg y\}$ or even just $\{x\}$.

Recap: Satisfiability

Definition. A formula ϕ is **satisfiable** if there is *some* **assignment** which makes ϕ true.

Example. $(x \vee y) \wedge (x \vee \neg y)$ is satisfied by $\{x, \neg y\}$ or even just $\{x\}$.

Question. *How do I convince someone that ϕ is satisfiable?*

Proof of Satisfiability

Proof of Satisfiability

Easy, just given them a satisfying assignment.

Proof of Satisfiability

Easy, just given them a satisfying assignment.

It is easy to check if a formula is satisfied by an assignment.

Proof of Satisfiability

Easy, just given them a satisfying assignment.

It is easy to check if a formula is satisfied by an assignment.

Question. *What about unsatisfiability?*

Brute-Force Approach

Brute-Force Approach

```
We can tell them:
```

Brute-Force Approach

We can tell them:

Look at all possible assignments on the variables of ϕ and check if they all make ϕ false.

Brute-Force Approach

We can tell them:

Look at all possible assignments on the variables of ϕ and check if they all make ϕ false.

(We did this in the previous homework assignment.)

Brute-Force Approach

We can tell them:

Look at all possible assignments on the variables of ϕ and check if they all make ϕ false.

(We did this in the previous homework assignment.)

For large formulas this is unreasonable.

Recall: Validity and Satisfiability

Recall: Validity and Satisfiability

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

Recall: Validity and Satisfiability

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

Determining satisfiability is at least as hard as determining validity.

Recall: Validity and Satisfiability

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

Determining satisfiability is at least as hard as determining validity.

There is a tight connection between satisfiability and proof.

Example

$$x_1 \land (\neg x_1 \lor x_2) \land \ldots \land (\neg x_{n-1} \lor x_n) \land \neg x_n$$

Example

$$x_1 \wedge (\neg x_1 \vee x_2) \wedge \ldots \wedge (\neg x_{n-1} \vee x_n) \wedge \neg x_n$$

Question. *Is this formula satisfiable?*

Example

$$x_1 \wedge (\neg x_1 \vee x_2) \wedge \ldots \wedge (\neg x_{n-1} \vee x_n) \wedge \neg x_n$$

Question. *Is this formula satisfiable?*

It's not. It encodes a sequence of impossible implications.

Example

$$x_1 \wedge (\neg x_1 \vee x_2) \wedge \ldots \wedge (\neg x_{n-1} \vee x_n) \wedge \neg x_n$$

Question. *Is this formula satisfiable?*

It's not. It encodes a sequence of impossible implications.

Looking at all possible assignments seems unnecessary.

We want to formalize the intuition that we can demonstrate unsatisfiability by *reasoning* instead of brute force.

Proof Systems

Proof Systems

(Informal) Definition. A **proof system** or **derivation system** $\mathscr{P}$ is a formal system for modeling proofs.

Proof Systems

(Informal) Definition. A **proof system** or **derivation system** $\mathscr{P}$ is a formal system for modeling proofs.

There are many shapes of proof systems but generally they are:

Proof Systems

(Informal) Definition. A **proof system** or **derivation system** $\mathscr{P}$ is a formal system for modeling proofs.

There are many shapes of proof systems but generally they are:

- **Directed** We prove one thing *from* another thing

Proof Systems

(Informal) Definition. A **proof system** or **derivation system** $\mathscr{P}$ is a formal system for modeling proofs.

There are many shapes of proof systems but generally they are:

- **Directed** We prove one thing *from* another thing
- **Checkable** We require that is is easy to check if a proof is correct.

Judgements (Sequents)

$$\Phi_1, \ldots, \Phi_k \vdash_{\mathscr{P}} \Psi_1, \ldots, \Psi_l$$

Judgements (Sequents)

$$\Phi_1, \ldots, \Phi_k \vdash_{\mathscr{P}} \Psi_1, \ldots, \Psi_l$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then *at least one* of $\Psi_1, \ldots, \Psi_l$ hold."

Judgements (Sequents)

$$\underbrace{\Phi_1, \ldots, \Phi_k}_{\text{Context}} \vdash_{\mathscr{P}} \Psi_1, \ldots, \Psi_l$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then *at least one* of $\Psi_1, \ldots, \Psi_l$ hold."

$\Phi_1, \ldots, \Phi_k$ are assumptions, a.k.a antecedents, a.k.a the context.

Judgements (Sequents)

$$\underbrace{\Phi_1, \ldots, \Phi_k}_{\text{Context}} \vdash_{\mathscr{P}} \Psi_1, \ldots, \Psi_l$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then *at least one* of $\Psi_1, \ldots, \Psi_l$ hold."

$\Phi_1, \ldots, \Phi_k$ are assumptions, a.k.a antecedents, a.k.a the context.

$\Psi_1, \ldots, \Psi_l$ are conclusions, a.k.a. consequents (typically there will be just one.

Judgements (Sequents)

$$\boxed{\Phi_1, \ldots, \Phi_k} \vdash_{\mathscr{P}} \Psi_1, \ldots, \Psi_l$$

Context

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then *at least one* of $\Psi_1, \ldots, \Psi_l$ hold."

$\Phi_1, \ldots, \Phi_k$ are assumptions, a.k.a antecedents, a.k.a the context.

$\Psi_1, \ldots, \Psi_l$ are conclusions, a.k.a. consequents (typically there will be just one.

Φ and Ψ are *statements*. Today that means **formulas**, later that will mean *typing statements*.

Inference Rules

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n}{J_{n+1}} \; (\texttt{condition})$$

In **inference rule** an way of describing an individual step in a proof.

It reads: "If the judgments $J_1, \ldots, J_n$ hold and the condition is met, then judgment J_{n+1} holds."

Example: Modus Ponens

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P \to Q}{\Gamma \vdash Q}$$

Example: Modus Ponens

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P \to Q}{\Gamma \vdash Q}$$

"If P holds and $P \to Q$ holds then Q must hold."

Example: Modus Ponens

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P \to Q}{\Gamma \vdash Q}$$

"If P holds and $P \to Q$ holds then Q must hold."

(There is no side condition.)

Example: Modus Ponens

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P \to Q}{\Gamma \vdash Q}$$

"If P holds and $P \to Q$ holds then Q must hold."

(There is no side condition.)

Note. P and Q are *meta-variables*, they stand for arbitrary formulas. Γ is also a meta-variable, but for a list of formulas.

Derivation Trees

Derivation Trees

A proof system $\mathscr{P}$ is defined by a collection of inference rules.

Derivation Trees

A proof system $\mathscr{P}$ is defined by a collection of inference rules.

Definition. A **derivation tree for** $\mathscr{P}$ is a tree in which every node is a judgment and a node with its parents represent inference rules.

Derivation Trees

A proof system $\mathscr{P}$ is defined by a collection of inference rules.

Definition. A **derivation tree for** $\mathscr{P}$ is a tree in which every node is a judgment and a node with its parents represent inference rules.

Leaves are called **axioms.**

Derivation Trees

A proof system $\mathscr{P}$ is defined by a collection of inference rules.

Definition. A **derivation tree for** $\mathscr{P}$ is a tree in which every node is a judgment and a node with its parents represent inference rules.

Leaves are called **axioms.**

We say that $\Gamma \vdash \Psi$ holds if there is a derivation tree which has this judgment at the root.

Resolution

The Resolution Rule

$$\mathrm{res}(\mathrm{C} \vee \mathrm{x}, \mathrm{D} \vee \neg \mathrm{x}, \mathrm{x}) = C \vee D$$

The Resolution Rule

$$\text{res}(\mathsf{C} \vee \mathsf{x}, \mathsf{D} \vee \neg \mathsf{x}, \mathsf{x}) = C \vee D$$

```
If we have two clauses which disagree on a
variable, we combine them and into a single
clause, dropping the conflicting literals.
```

The Resolution Rule

$$\mathrm{res}(\mathrm{C} \vee \mathrm{x}, \mathrm{D} \vee \neg \mathrm{x}, \mathrm{x}) = C \vee D$$

If we have two clauses which disagree on a variable, we combine them and into a single clause, dropping the conflicting literals.

Fact. If v satisfies clauses $C \vee x$ and $D \vee \neg x$ then v satisfies $C \vee D$.

Resolution

axiom $$\frac{}{\phi \vdash C} \; (C \in \phi)$$

res. rule $$\frac{\phi \vdash C \vee x \quad \phi \vdash D \vee \neg x}{\phi \vdash C \vee D}$$

The resolution proof system $\mathscr{R}$ has the following two inference rules.

Note. The antecedent never changes, so we could drop it if we remember what ϕ is.

Example

$$\dfrac{\dfrac{x_1 \land (\neg x_1 \lor x_2) \land \neg x_2 \vdash x_1 \quad x_1 \land (\neg x_1 \lor x_2) \land \neg x_2 \vdash (\neg x_1 \lor x_2)}{x_1 \land (\neg x_1 \lor x_2) \land \neg x_2 \vdash x_2} \qquad x_1 \land (x_1 \lor x_2) \land \neg x_2 \vdash \neg x_2}{x_1 \land (\neg x_1 \lor x_2) \lor \neg x_2 \vdash \varnothing}$$

Example

$$\dfrac{\dfrac{x_1 \qquad \neg x_1 \vee x_2}{x_2} \qquad \neg x_2}{\varnothing}$$

The following is a resolution derivation of $x_1 \wedge (\neg x_1 \vee x_2) \wedge \neg x_2 \vdash \varnothing$.

Soundness

Soundness

Soundness

Theorem. If v satisfies ϕ and $\phi \vdash_{\mathscr{R}} C$ holds, then v satisfies C.

Soundness

Theorem. If v satisfies ϕ and $\phi \vdash_{\mathscr{R}} C$ holds, then v satisfies C.

In particular, if $\phi \vdash \varnothing$ then ϕ is unsatisfiable.

Soundness

Theorem. If v satisfies ϕ and $\phi \vdash_{\mathscr{R}} C$ holds, then v satisfies C.

In particular, if $\phi \vdash \varnothing$ then ϕ is unsatisfiable.

Soundness says that we cannot use a resolution proof to show that a satisfiable formula is unsatisfiable.

Soundness

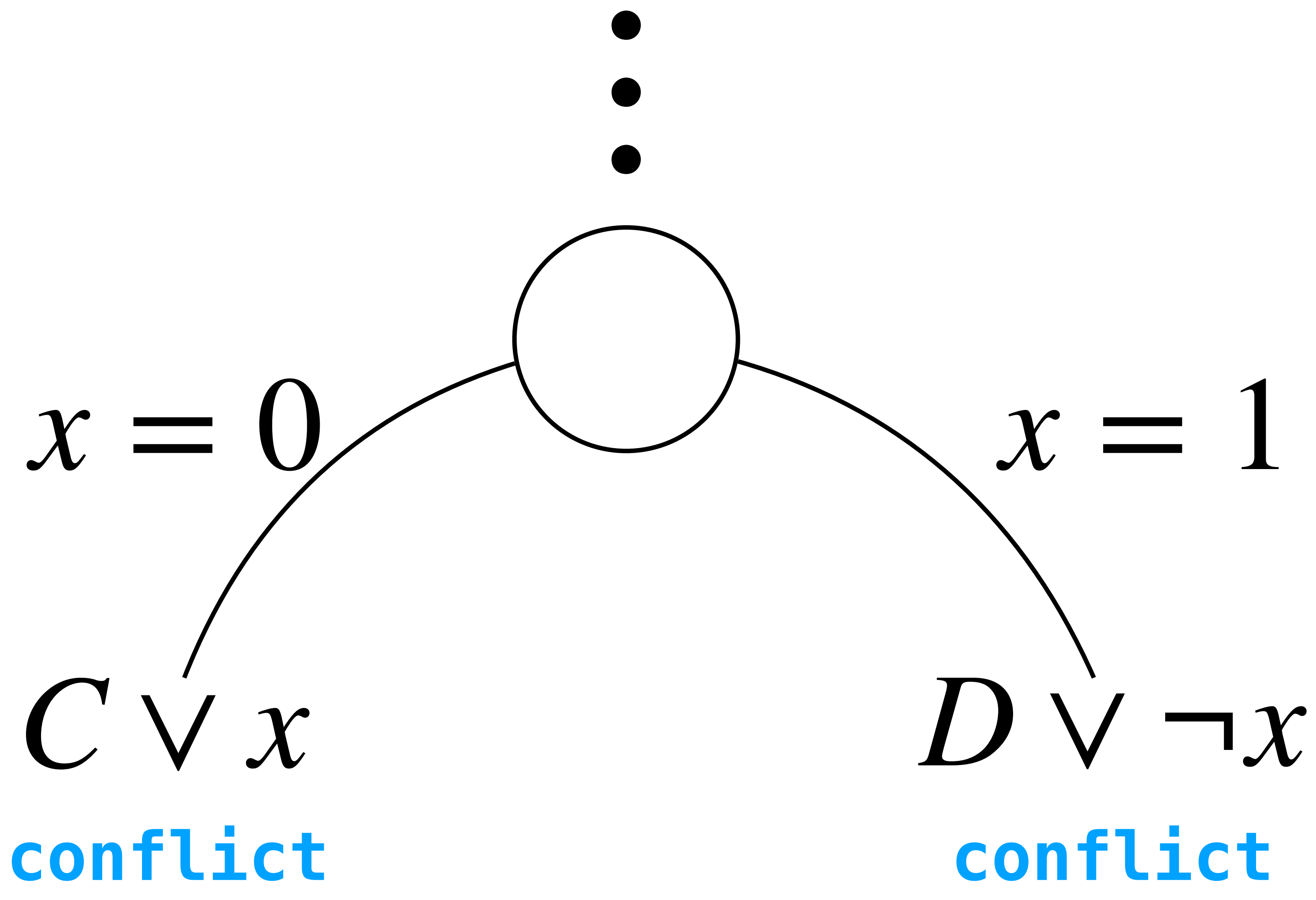
Theorem. If v satisfies ϕ and $\phi \vdash_{\mathscr{R}} C$ holds, then v satisfies C.

In particular, if $\phi \vdash \varnothing$ then ϕ is unsatisfiable.

Soundness says that we cannot use a resolution proof to show that a satisfiable formula is unsatisfiable.

We call a derivation of $\phi \vdash \varnothing$ a **refutation** of ϕ.

DPLL and Resolution



$$\frac{C \vee x \quad D \vee \neg x}{C \vee D}$$

`If` $C \vee x$ `is falsified and` $D \vee \neg x$ `is falsified, then` $C \vee D$ `is falsified by the assignment` *`without`* `assigning` x.

Completeness

Theorem. If ϕ is unsatisfiable, then $\phi \vdash_{\mathscr{R}} \varnothing$

Completeness says we don't miss any unsatisfiable assignments, that if a formula is unsatisfiable, we can always expect a resolution refutation.

Completeness

Theorem. If ϕ is unsatisfiable, then $\phi \vdash_{\mathscr{R}} \varnothing$

Completeness says we don't miss any unsatisfiable assignments, that if a formula is unsatisfiable, we can always expect a resolution refutation.