Formal Methods in Software Developement

Propositional Logic - refresher

Mădălina Erașcu

West University of Timișoara
Faculty of Mathematics and Informatics

Based on slides of the lecture Satisfiability Checking (Erika Ábrahám), RTWH Aachen

October 5, 2018

Propositional logic

The slides are partly taken from:

www.decision-procedures.org/slides/

Propositional logic - Outline

- Syntax of propositional logic
- Semantics of propositional logic
- Satisfiability and validity
- Normal forms
- Enumeration and deduction

Propositional logic - Outline

- Syntax of propositional logic
- Semantics of propositional logic
- Satisfiability and validity
- Normal forms
- Enumeration and deduction

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$.
We write *PropForm* for the set of all propositional logic formulae.

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \;:=\; a \;\mid\; (\neg\varphi) \;\mid\; (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$.
We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\bot \qquad := $$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \;:=\; a \;\mid\; (\neg\varphi) \;\mid\; (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\bot \qquad := (a \wedge \neg a)$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$.
We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{ll} \bot & := (a \wedge \neg a) \\ \top & := \end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \;:=\; a \;\mid\; (\neg\varphi) \;\mid\; (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{ll} \bot & := (a \wedge \neg a) \\ \top & := (a \vee \neg a) \end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{aligned} \bot &:= (a \wedge \neg a) \\ \top &:= (a \vee \neg a) \\ (\ \varphi_1 \ \vee \ \varphi_2\ ) &:= \end{aligned}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \;:=\; a \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$.
We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{rcl} \bot & := & (a \wedge \neg a) \\ \top & := & (a \vee \neg a) \\ (\varphi_1 \vee \varphi_2) & := & \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$.
We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{rcl}
\bot & := & (a \wedge \neg a) \\
\top & := & (a \vee \neg a) \\
(\varphi_1 \vee \varphi_2) & := & \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\
(\varphi_1 \rightarrow \varphi_2) & := &
\end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \;:=\; a \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{rcl} \bot & := & (a \wedge \neg a) \\ \top & := & (a \vee \neg a) \\ (\varphi_1 \vee \varphi_2) & := & \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\ (\varphi_1 \rightarrow \varphi_2) & := & ((\neg\varphi_1) \vee \varphi_2) \end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{rcccl}
 & & \bot & & := (a \wedge \neg a) \\
 & & \top & & := (a \vee \neg a) \\
(& \varphi_1 & \vee & \varphi_2 &) := \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\
(& \varphi_1 & \rightarrow & \varphi_2 &) := ((\neg\varphi_1) \vee \varphi_2) \\
(& \varphi_1 & \leftrightarrow & \varphi_2 &) :=
\end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$.
We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{ccccl}
 & & \bot & & := (a \wedge \neg a) \\
 & & \top & & := (a \vee \neg a) \\
(& \varphi_1 & \vee & \varphi_2 &) := \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\
(& \varphi_1 & \rightarrow & \varphi_2 &) := ((\neg\varphi_1) \vee \varphi_2) \\
(& \varphi_1 & \leftrightarrow & \varphi_2 &) := ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1))
\end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \; := \; a \;|\; (\neg\varphi) \;|\; (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{rcccl}
 & & \bot & & := (a \wedge \neg a) \\
 & & \top & & := (a \vee \neg a) \\
(& \varphi_1 & \vee & \varphi_2 &) := \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\
(& \varphi_1 & \rightarrow & \varphi_2 &) := ((\neg\varphi_1) \vee \varphi_2) \\
(& \varphi_1 & \leftrightarrow & \varphi_2 &) := ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)) \\
(& \varphi_1 & \bigoplus & \varphi_2 &) :=
\end{array}$$

Syntax of propositional logic

Abstract syntax of well-formed propositional formulae:

$$\varphi \ := \ a \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

Let AP is a set of (atomic) propositions (Boolean variables). Then $a \in AP$. We write *PropForm* for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{array}{ccccc}
 & & \bot & & := (a \wedge \neg a) \\
 & & \top & & := (a \vee \neg a) \\
(& \varphi_1 & \vee & \varphi_2 &) := \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\
(& \varphi_1 & \rightarrow & \varphi_2 &) := ((\neg\varphi_1) \vee \varphi_2) \\
(& \varphi_1 & \leftrightarrow & \varphi_2 &) := ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)) \\
(& \varphi_1 & \bigoplus & \varphi_2 &) := (\varphi_1 \leftrightarrow (\neg\varphi_2))
\end{array}$$

Formulae

- Examples of well-formed formulae:
  - $(\neg a)$
  - $(\neg(\neg a))$
  - $(a \wedge (b \wedge c))$
  - $(a \rightarrow (b \rightarrow c))$

Formulae

- Examples of well-formed formulae:
  - $(\neg a)$
  - $(\neg(\neg a))$
  - $(a \wedge (b \wedge c))$
  - $(a \to (b \to c))$
- We omit parentheses whenever we may restore them through operator precedence:

$$\xleftarrow{\text{binds stronger}}$$

$$\neg \quad \wedge \quad \vee \quad \to \quad \leftrightarrow$$

chaining the same operator: left binds stronger
e.g., $a \to b \to c$ means $((a \to b) \to c)$

Propositional logic - Outline

- Syntax of propositional logic
- Semantics of propositional logic
- Satisfiability and validity
- Normal forms
- Enumeration and deduction

Semantics: Assignments

Structures for predicate logic:

- The domain is $\mathbb{B} = \{0, 1\}$.
- The interpretation assigns Boolean values to the variables:

$$\alpha : AP \to \{0, 1\}$$

We call these special interpretations assignments and use *Assign* to denote the set of all assignments.

Semantics: Assignments

Structures for predicate logic:

- The domain is $\mathbb{B} = \{0, 1\}$.
- The interpretation assigns Boolean values to the variables:

$$\alpha : AP \rightarrow \{0, 1\}$$

We call these special interpretations assignments and use *Assign* to denote the set of all assignments.

Example: $AP = \{a, b\}, \alpha(a) = 0, \alpha(b) = 1$

Semantics I: Truth tables

Semantics I: Truth tables

- Truth tables define the semantics (=meaning) of the operators.
  They can be used to define the semantics of formulae inductively over their structure.

Semantics I: Truth tables

- Truth tables define the semantics (=meaning) of the operators.
  They can be used to define the semantics of formulae inductively over their structure.
- Convention: 0= false, 1= true

Semantics I: Truth tables

- Truth tables define the semantics (=meaning) of the operators. They can be used to define the semantics of formulae inductively over their structure.
- Convention: 0= false, 1= true

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$	$p \bigoplus q$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	0	0	1
1	1	0	1	1	1	1	0

- Truth tables define the semantics (=meaning) of the operators.
  They can be used to define the semantics of formulae inductively over their structure.
- Convention: 0= false, 1= true

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$	$p \bigoplus q$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	0	0	1
1	1	0	1	1	1	1	0

Each possible assignment is covered by a line of the truth table.
α satisfies φ iff in the line for α and the column for φ the entry is 1.

Semantics I: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.

Semantics I: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.
- Let $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

Semantics I: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.
- Let $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.
- Q: Does α satisfy φ?

Semantics I: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.
- Let $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.
- Q: Does α satisfy φ?
- A1: Replace values of α in φ.

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$\alpha \models p$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$\alpha \models p \qquad \textit{iff} \qquad \alpha(p) = \mathit{true}$$

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$\begin{array}{lll} \alpha \models p & \textit{iff} & \alpha(p) = \mathit{true} \\ \alpha \models \neg\varphi & & \end{array}$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq Assign \times PropForm$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$\begin{array}{llll} \alpha & \models p & \textit{iff} & \alpha(p) = \textit{true} \\ \alpha & \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \end{array}$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \mathit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & &
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq Assign \times PropForm$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \textit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \mathit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & &
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq Assign \times PropForm$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \textit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ or } \alpha \models \varphi_2
\end{array}$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq Assign \times PropForm$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \textit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ or } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \rightarrow \varphi_2 & &
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \mathit{iff} & \alpha(p) = \mathit{true} \\
\alpha \models \neg\varphi & \mathit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \mathit{iff} & \alpha \models \varphi_1 \text{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & \mathit{iff} & \alpha \models \varphi_1 \text{ or } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \rightarrow \varphi_2 & \mathit{iff} & \alpha \models \varphi_1 \text{ implies } \alpha \models \varphi_2
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \mathit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ or } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \rightarrow \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ implies } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \leftrightarrow \varphi_2 & &
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \textit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ or } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \rightarrow \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ implies } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \leftrightarrow \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ iff } \alpha \models \varphi_2
\end{array}
$$

Semantics II: Satisfaction relation

Satisfaction relation: $\models \subseteq \mathit{Assign} \times \mathit{PropForm}$

Instead of $(\alpha, \varphi) \in \models$ we write $\alpha \models \varphi$ and say that

- α satisfies φ or
- φ holds for α or
- α is a model of φ.

$\models$ is defined recursively:

$$
\begin{array}{lll}
\alpha \models p & \textit{iff} & \alpha(p) = \mathit{true} \\
\alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ and } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \vee \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ or } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \rightarrow \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ implies } \alpha \models \varphi_2 \\
\alpha \models \varphi_1 \leftrightarrow \varphi_2 & \textit{iff} & \alpha \models \varphi_1 \textit{ iff } \alpha \models \varphi_2
\end{array}
$$

Note: More elegant but semantically equivalent to truth tables.

Semantics II: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.

Semantics II: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.
- Let $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

Semantics II: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.
- Let $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- Q: Does α satisfy φ?

Semantics II: Example

- Let φ be defined as $(a \vee (b \rightarrow c))$.
- Let $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- Q: Does α satisfy φ?

A2: Compute with the satisfaction relation:

$$
\begin{array}{ll}
 & \alpha \models (a \vee (b \rightarrow c)) \\
\textit{iff} & \alpha \models a \textit{ or } \alpha \models (b \rightarrow c) \\
\textit{iff} & \alpha \models a \textit{ or } (\alpha \models b \textit{ implies } \alpha \models c) \\
\textit{iff} & 0 \textit{ or } (0 \textit{ implies } 1) \\
\textit{iff} & 0 \textit{ or } 1 \\
\textit{iff} & 1
\end{array}
$$

- Using the satisfaction relation we can define an algorithm for the problem to decide whether an assignment $\alpha : AP \to \{0, 1\}$ is a model of a propositional logic formula $\varphi \in \mathit{PropForm}$:

- Using the satisfaction relation we can define an algorithm for the problem to decide whether an assignment $\alpha : AP \rightarrow \{0, 1\}$ is a model of a propositional logic formula $\varphi \in PropForm$:

$$
\begin{array}{l}
Eval(\alpha, \varphi) \ \{ \\
\quad \textbf{if} \ \varphi \equiv a \ \textbf{return} \ \alpha(a); \\
\quad \textbf{if} \ \varphi \equiv (\neg\varphi_1) \ \textbf{return} \ \textbf{not} \ Eval(\alpha, \varphi_1); \\
\quad \textbf{if} \ \varphi \equiv (\varphi_1 \ \mathrm{op} \ \varphi_2) \\
\qquad\qquad \textbf{return} \ Eval(\alpha, \varphi_1) \ [\![\mathrm{op}]\!] \ Eval(\alpha, \varphi_2); \\
\}
\end{array}
$$

- Using the satisfaction relation we can define an algorithm for the problem to decide whether an assignment $\alpha : AP \to \{0, 1\}$ is a model of a propositional logic formula $\varphi \in PropForm$:

```
Eval(α, φ) {
   if φ ≡ a return α(a);
   if φ ≡ (¬φ1) return not Eval(α, φ1);
   if φ ≡ (φ1 op φ2)
        return Eval(α, φ1) [[op]] Eval(α, φ2);
}
```

- Equivalent to the $\models$ relation, but from the algorithmic view.

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \rightarrow c))$
  - $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \to c))$
  - $\alpha : \{a, b, c\} \to \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $Eval(\alpha, \varphi) \ =$

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \rightarrow c))$
  - $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $Eval(\alpha,\ \varphi) \ = \ Eval(\alpha,\ a) \ or \ Eval(\alpha,\ b \rightarrow c) =$

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \rightarrow c))$
  - $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $$\begin{aligned} \mathit{Eval}(\alpha,\ \varphi) \ &= \ \mathit{Eval}(\alpha,\ a) \ \mathit{or} \ \mathit{Eval}(\alpha,\ b \rightarrow c) = \\ &\ 0 \ \mathit{or} \ (\mathrm{Eval}(\alpha,\ b) \ \mathit{implies} \ \mathrm{Eval}(\alpha,\ c)) = \end{aligned}$$

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \rightarrow c))$
  - $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $$\begin{aligned} \mathit{Eval}(\alpha,\ \varphi) \ &= \ \mathit{Eval}(\alpha,\ a) \ \mathit{or} \ \mathit{Eval}(\alpha,\ b \rightarrow c) = \\ &\quad 0 \ \mathit{or} \ (\mathrm{Eval}(\alpha,\ b) \ \mathit{implies} \ \mathrm{Eval}(\alpha,\ c)) = \\ &\quad 0 \ \mathit{or} \ (0 \ \mathit{implies} \ 1) = \end{aligned}$$

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \to c))$
  - $\alpha : \{a, b, c\} \to \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $$\begin{aligned} Eval(\alpha, \varphi) \;=\; & Eval(\alpha,\, a) \textit{ or } Eval(\alpha,\, b \to c) = \\ & 0 \textit{ or } (\mathrm{Eval}(\alpha,\, b) \textit{ implies } \mathrm{Eval}(\alpha,\, c)) = \\ & 0 \textit{ or } (0 \textit{ implies } 1) = \\ & 0 \textit{ or } 1 = \end{aligned}$$

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \rightarrow c))$
  - $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $$\begin{aligned} \mathit{Eval}(\alpha,\ \varphi) \ = \ & \mathit{Eval}(\alpha,\ a) \ \mathit{or} \ \mathit{Eval}(\alpha,\ b \rightarrow c) = \\ & 0 \ \mathit{or} \ (\mathrm{Eval}(\alpha,\ b) \ \mathit{implies} \ \mathrm{Eval}(\alpha,\ c)) = \\ & 0 \ \mathit{or} \ (0 \ \mathit{implies} \ 1) = \\ & 0 \ \mathit{or} \ 1 = \\ & 1 \end{aligned}$$

Semantics III: Example

- Recall our example
  - $\varphi = (a \vee (b \rightarrow c))$
  - $\alpha : \{a, b, c\} \rightarrow \{0, 1\}$ with $\alpha(a) = 0$, $\alpha(b) = 0$, and $\alpha(c) = 1$.

- $$\begin{aligned} Eval(\alpha,\ \varphi) \ = \ & Eval(\alpha,\ a) \textit{ or } Eval(\alpha,\ b \rightarrow c) = \\ & 0 \textit{ or } (\mathrm{Eval}(\alpha,\ b) \textit{ implies } \mathrm{Eval}(\alpha,\ c)) = \\ & 0 \textit{ or } (0 \textit{ implies } 1) = \\ & 0 \textit{ or } 1 = \\ & 1 \end{aligned}$$

- Hence, $\alpha \models \varphi$.

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \rightarrow 2^{Assign}$

  (a formula $\rightarrow$ set of its satisfying assignments)

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \rightarrow 2^{Assign}$

  (a formula $\rightarrow$ set of its satisfying assignments)
- Recursive definition:

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$sat(a) \quad =$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \rightarrow 2^{Assign}$

  (a formula $\rightarrow$ set of its satisfying assignments)
- Recursive definition:

$$sat(a) \quad = \quad \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$\begin{array}{lcl} sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\ sat(\neg\varphi_1) & = & \end{array}$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$\begin{array}{lcl} sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\ sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \end{array}$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = &
\end{array}
$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = & sat(\varphi_1) \cap sat(\varphi_2)
\end{array}
$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \rightarrow 2^{Assign}$

  (a formula $\rightarrow$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = & sat(\varphi_1) \cap sat(\varphi_2) \\
sat(\varphi_1 \vee \varphi_2) & = &
\end{array}
$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \rightarrow 2^{Assign}$

  (a formula $\rightarrow$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = & sat(\varphi_1) \cap sat(\varphi_2) \\
sat(\varphi_1 \vee \varphi_2) & = & sat(\varphi_1) \cup sat(\varphi_2)
\end{array}
$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = & sat(\varphi_1) \cap sat(\varphi_2) \\
sat(\varphi_1 \vee \varphi_2) & = & sat(\varphi_1) \cup sat(\varphi_2) \\
sat(\varphi_1 \to \varphi_2) & = &
\end{array}
$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \to 2^{Assign}$

  (a formula $\to$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = & sat(\varphi_1) \cap sat(\varphi_2) \\
sat(\varphi_1 \vee \varphi_2) & = & sat(\varphi_1) \cup sat(\varphi_2) \\
sat(\varphi_1 \to \varphi_2) & = & (Assign \setminus sat(\varphi_1)) \cup sat(\varphi_2)
\end{array}
$$

Satisfying assignments

- Intuition: each formula specifies a set of assignments satisfying it.
- Remember: *Assign* denotes the set of all assignments.
- Function $sat : PropForm \rightarrow 2^{Assign}$

  (a formula $\rightarrow$ set of its satisfying assignments)
- Recursive definition:

$$
\begin{array}{lcl}
sat(a) & = & \{\alpha \mid \alpha(a) = 1\}, \quad a \in AP \\
sat(\neg\varphi_1) & = & Assign \setminus sat(\varphi_1) \\
sat(\varphi_1 \wedge \varphi_2) & = & sat(\varphi_1) \cap sat(\varphi_2) \\
sat(\varphi_1 \vee \varphi_2) & = & sat(\varphi_1) \cup sat(\varphi_2) \\
sat(\varphi_1 \rightarrow \varphi_2) & = & (Assign \setminus sat(\varphi_1)) \cup sat(\varphi_2)
\end{array}
$$

- For $\varphi \in PropForm$ and $\alpha \in Assign$ it holds that

$$\alpha \models \varphi \qquad \textit{iff} \qquad \alpha \in sat(\varphi)$$

Satisfying assignments: Example

$$sat(a \vee (b \rightarrow c)) \qquad =$$

Satisfying assignments: Example

$$sat(a \vee (b \rightarrow c)) \qquad =$$

$$sat(a) \cup sat(b \rightarrow c)$$

Satisfying assignments: Example

$$
\begin{aligned}
&sat(a \vee (b \rightarrow c)) &&= \\
&sat(a) \cup sat(b \rightarrow c) &&= \\
&sat(a) \cup ((Assign \setminus sat(b)) \cup sat(c))
\end{aligned}
$$

Satisfying assignments: Example

$$\begin{aligned}
&sat(a \vee (b \to c)) &&= \\
&sat(a) \cup sat(b \to c) &&= \\
&sat(a) \cup ((Assign \setminus sat(b)) \cup sat(c)) &&= \\
&\{\alpha \in Assign \mid \alpha(a) = 1\} \cup \\
&\{\alpha \in Assign \mid \alpha(b) = 0\} \cup \\
&\{\alpha \in Assign \mid \alpha(c) = 1\}
\end{aligned}$$

Satisfying assignments: Example

$$
\begin{aligned}
&sat(a \vee (b \to c)) &&= \\
&sat(a) \cup sat(b \to c) &&= \\
&sat(a) \cup ((Assign \setminus sat(b)) \cup sat(c)) &&= \\
&\{\alpha \in Assign \mid \alpha(a) = 1\} \cup \\
&\{\alpha \in Assign \mid \alpha(b) = 0\} \cup \\
&\{\alpha \in Assign \mid \alpha(c) = 1\} &&= \\
&\{\alpha \in Assign \mid \alpha(a) = 1 \textit{ or } \alpha(b) = 0 \textit{ or } \alpha(c) = 1\}
\end{aligned}
$$

Short summary for propositional logic

- Syntax of propositional formulae $\varphi \in \mathit{PropForm}$:

$$\varphi \;:=\; AP \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$$

- Semantics:
  - Assignments $\alpha \in \mathit{Assign}$:

$$\begin{array}{l} \alpha : AP \to \{0,1\} \\ \alpha \in 2^{AP} \\ \alpha \in \{0,1\}^{AP} \end{array}$$

  - Satisfaction relation:

$$\begin{array}{llll} \models \subseteq \mathit{Assign} \times \mathit{PropForm} & , & (\text{e.g., } \alpha & \models \varphi\ ) \\ \models \subseteq 2^{\mathit{Assign}} \times \mathit{PropForm} & , & (\text{e.g., } \{\alpha_1, \ldots, \alpha_n\} & \models \varphi\ ) \\ \models \subseteq \mathit{PropForm} \times \mathit{PropForm}, & & (\text{e.g., } \varphi_1 & \models \varphi_2) \\ \mathit{sat} : \ \mathit{PropForm} \to 2^{\mathit{Assign}} & , & (\text{e.g., } \ \mathit{sat}(\varphi) & \ ) \end{array}$$

Propositional logic - Outline

- Syntax of propositional logic
- Semantics of propositional logic
- Satisfiability and validity
- Normal forms
- Enumeration and deduction

Semantic classification of formulae

- A formula φ is called valid if $sat(\varphi) = Assign$.
  (Also called a tautology).

Semantic classification of formulae

- A formula φ is called valid if $sat(\varphi) = Assign$.
  (Also called a tautology).

- A formula φ is called satisfiable if $sat(\varphi) \neq \emptyset$.

Semantic classification of formulae

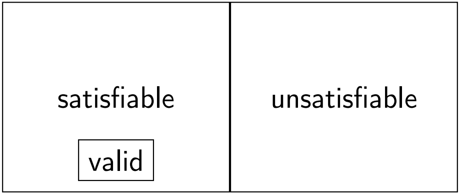
- A formula φ is called valid if $sat(\varphi) = Assign$.
  (Also called a tautology).

- A formula φ is called satisfiable if $sat(\varphi) \neq \emptyset$.

- A formula φ is called unsatisfiable if $sat(\varphi) = \emptyset$.
  (Also called a contradiction).

Semantic classification of formulae

- A formula φ is called valid if $sat(\varphi) = Assign$. (Also called a tautology).
- A formula φ is called satisfiable if $sat(\varphi) \neq \emptyset$.
- A formula φ is called unsatisfiable if $sat(\varphi) = \emptyset$. (Also called a contradiction).



Some notations

Some notations

- We can write:
  - $\models \varphi$ when φ is valid

Some notations

- We can write:
  - $\models \varphi$ when φ is valid
  - $\not\models \varphi$ when φ is not valid

Some notations

- We can write:
  - $\models \varphi$ when φ is valid
  - $\not\models \varphi$ when φ is not valid
  - $\not\models \neg\varphi$ when φ is

Some notations

- We can write:
  - $\models \varphi$ when φ is valid
  - $\not\models \varphi$ when φ is not valid
  - $\not\models \neg\varphi$ when φ is satisfiable

Some notations

- We can write:
  - $\models \varphi$ when φ is valid
  - $\not\models \varphi$ when φ is not valid
  - $\not\models \neg\varphi$ when φ is satisfiable
  - $\models \neg\varphi$ when φ is

Some notations

- We can write:
  - $\models \varphi$ when φ is valid
  - $\not\models \varphi$ when φ is not valid
  - $\not\models \neg\varphi$ when φ is satisfiable
  - $\models \neg\varphi$ when φ is unsatisfiable

Examples

- $(x_1 \wedge x_2) \rightarrow (x_1 \vee x_2)$

Examples

- $(x_1 \wedge x_2) \rightarrow (x_1 \vee x_2)$ is valid

Examples

- $(x_1 \wedge x_2) \rightarrow (x_1 \vee x_2)$ is valid
- $(x_1 \vee x_2) \rightarrow x_1$

Examples

- $(x_1 \land x_2) \to (x_1 \lor x_2)$ is valid
- $(x_1 \lor x_2) \to x_1$ is satisfiable

Examples

- $(x_1 \wedge x_2) \rightarrow (x_1 \vee x_2)$ is valid
- $(x_1 \vee x_2) \rightarrow x_1$ is satisfiable
- $(x_1 \wedge x_2) \wedge \neg x_1$

Examples

- $(x_1 \wedge x_2) \rightarrow (x_1 \vee x_2)$ is valid
- $(x_1 \vee x_2) \rightarrow x_1$ is satisfiable
- $(x_1 \wedge x_2) \wedge \neg x_1$ is unsatisfiable

Examples

- Here are some valid formulae:
  - $\models a \wedge 1 \leftrightarrow a$
  - $\models a \wedge 0 \leftrightarrow 0$

Examples

- Here are some valid formulae:
  - $\models a \wedge 1 \leftrightarrow a$
  - $\models a \wedge 0 \leftrightarrow 0$
  - $\models \neg\neg a \leftrightarrow a$ (double-negation rule)

Examples

- Here are some valid formulae:
  - $\models a \wedge 1 \leftrightarrow a$
  - $\models a \wedge 0 \leftrightarrow 0$
  - $\models \neg\neg a \leftrightarrow a$ (double-negation rule)
  - $\models a \wedge (b \vee c) \leftrightarrow (a \wedge b) \vee (a \wedge c)$

Examples

- Here are some valid formulae:
  - $\models a \wedge 1 \leftrightarrow a$
  - $\models a \wedge 0 \leftrightarrow 0$
  - $\models \neg\neg a \leftrightarrow a$ (double-negation rule)
  - $\models a \wedge (b \vee c) \leftrightarrow (a \wedge b) \vee (a \wedge c)$
- Some more (De Morgan rules):
  - $\models \neg(a \wedge b) \leftrightarrow (\neg a \vee \neg b)$
  - $\models \neg(a \vee b) \leftrightarrow (\neg a \wedge \neg b)$

The satisfiability problem for propositional logic

- The satisfiability problem for propositional logic is as follows:

  Given an input propositional formula φ, decide whether φ is satisfiable.

The satisfiability problem for propositional logic

- The satisfiability problem for propositional logic is as follows:

  Given an input propositional formula φ, decide whether φ is satisfiable.
- This problem is decidable but NP-complete.

The satisfiability problem for propositional logic

- The satisfiability problem for propositional logic is as follows:

  Given an input propositional formula φ, decide whether φ is satisfiable.
- This problem is decidable but NP-complete.
- An algorithm that always terminates for each propositional logic formula with the correct answer is called a decision procedure for propositional logic.

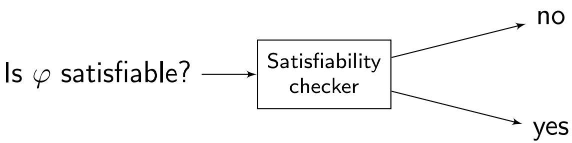
The satisfiability problem for propositional logic

- The satisfiability problem for propositional logic is as follows:

  Given an input propositional formula φ, decide whether φ is satisfiable.

- This problem is decidable but NP-complete.
- An algorithm that always terminates for each propositional logic formula with the correct answer is called a decision procedure for propositional logic.

Goal: Design and implement such a decision procedure:



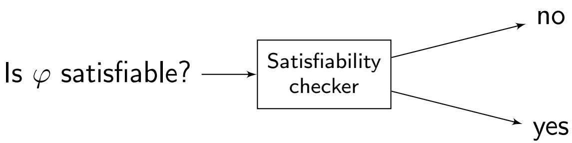
The satisfiability problem for propositional logic

- The satisfiability problem for propositional logic is as follows:

  Given an input propositional formula φ, decide whether φ is satisfiable.

- This problem is decidable but NP-complete.
- An algorithm that always terminates for each propositional logic formula with the correct answer is called a decision procedure for propositional logic.

Goal: Design and implement such a decision procedure:



Note: A formula φ is valid iff $\neg\varphi$ is unsatisfiable.

Propositional logic - Outline

- Syntax of propositional logic
- Semantics of propositional logic
- Satisfiability and validity
- Normal forms

Definitions

- Definition: A literal is either a variable or a negation of a variable.

Definitions

- Definition: A literal is either a variable or a negation of a variable.
- Example: $\varphi = \neg(a \vee \neg b)$

  Variables: $AP(\varphi) = \{a, b\}$

  Literals: $lit(\varphi) = \{a, \neg b\}$

Definitions

- Definition: A literal is either a variable or a negation of a variable.
- Example: $\varphi = \neg(a \vee \neg b)$

  Variables: $AP(\varphi) = \{a, b\}$

  Literals: $lit(\varphi) = \{a, \neg b\}$
- Note: Equivalent formulae can have different literals.

Definitions

- Definition: A literal is either a variable or a negation of a variable.
- Example: $\varphi = \neg(a \vee \neg b)$

  Variables: $AP(\varphi) = \{a, b\}$

  Literals: $lit(\varphi) = \{a, \neg b\}$
- Note: Equivalent formulae can have different literals.

  Example: $\varphi' = \neg a \wedge b$

  Literals: $lit(\varphi') = \{\neg a, b\}$

Definitions

- Definition: a term is a conjunction of literals
  - Example: $(a \wedge \neg b \wedge c)$

Definitions

- Definition: a term is a conjunction of literals
  - Example: $(a \wedge \neg b \wedge c)$
- Definition: a clause is a disjunction of literals
  - Example: $(a \vee \neg b \vee c)$

Negation Normal Form (NNF)

- Definition: A formula is in Negation Normal Form (NNF) iff
  (1) it contains only $\neg$, $\wedge$ and $\vee$ as connectives and
  (2) only variables are negated.

Negation Normal Form (NNF)

- Definition: A formula is in Negation Normal Form (NNF) iff
  (1) it contains only $\neg$, $\wedge$ and $\vee$ as connectives and
  (2) only variables are negated.

- Examples:
- $\varphi_1 = \neg(a \vee \neg b)$ is not in NNF
- $\varphi_2 = \neg a \wedge b$ is in NNF

Converting to NNF

- Every formula can be converted to NNF in linear time:
  - Eliminate all connectives other than $\wedge$, $\vee$, $\neg$
  - Use De Morgan and double-negation rules to push negations to operands

Converting to NNF

- Every formula can be converted to NNF in linear time:
  - Eliminate all connectives other than $\wedge$, $\vee$, $\neg$
  - Use De Morgan and double-negation rules to push negations to operands

- Example: $\varphi = \neg(a \to \neg b)$
  - Eliminate $'\to'$: $\varphi = \neg(\neg a \vee \neg b)$

Converting to NNF

- Every formula can be converted to NNF in linear time:
  - Eliminate all connectives other than $\wedge$, $\vee$, $\neg$
  - Use De Morgan and double-negation rules to push negations to operands

- Example: $\varphi = \neg(a \rightarrow \neg b)$
  - Eliminate $'\rightarrow'$: $\varphi = \neg(\neg a \vee \neg b)$
  - Push negation using De Morgan: $\varphi = (\neg\neg a \wedge \neg\neg b)$

Converting to NNF

- Every formula can be converted to NNF in linear time:
  - Eliminate all connectives other than $\wedge$, $\vee$, $\neg$
  - Use De Morgan and double-negation rules to push negations to operands

- Example: $\varphi = \neg(a \to \neg b)$
  - Eliminate $'\to'$: $\varphi = \neg(\neg a \vee \neg b)$
  - Push negation using De Morgan: $\varphi = (\neg\neg a \wedge \neg\neg b)$
  - Use double-negation rule: $\varphi = (a \wedge b)$

Disjunctive Normal Form (DNF)

- Definition: A formula is said to be in Disjunctive Normal Form (DNF) iff it is a disjunction of terms.

Disjunctive Normal Form (DNF)

- Definition: A formula is said to be in Disjunctive Normal Form (DNF) iff it is a disjunction of terms.
- In other words, it is a formula of the form

$$\bigvee_i \left(\bigwedge_j l_{i,j} \right)$$

where $l_{i,j}$ is the j-th literal in the i-th term.

Disjunctive Normal Form (DNF)

- Definition: A formula is said to be in Disjunctive Normal Form (DNF) iff it is a disjunction of terms.
- In other words, it is a formula of the form

$$\bigvee_i \left(\bigwedge_j l_{i,j} \right)$$

where $l_{i,j}$ is the j-th literal in the i-th term.

- Example:

$$\varphi = (a \land \neg b \land c) \lor (\neg a \land d) \lor (b) \quad \textit{is in DNF}$$

Disjunctive Normal Form (DNF)

- Definition: A formula is said to be in Disjunctive Normal Form (DNF) iff it is a disjunction of terms.
- In other words, it is a formula of the form

$$\bigvee_i \left(\bigwedge_j l_{i,j} \right)$$

where $l_{i,j}$ is the j-th literal in the i-th term.

- Example:

$$\varphi = (a \land \neg b \land c) \lor (\neg a \land d) \lor (b) \quad \textit{is in DNF}$$

- DNF is a special case of NNF.

Converting to DNF

- Every formula can be converted to DNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \wedge (\varphi_2 \vee \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

Converting to DNF

- Every formula can be converted to DNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \wedge (\varphi_2 \vee \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
- Example:

$$\begin{aligned}
\varphi \quad &= (a \vee b) \wedge (\neg c \vee d) \\
&= ((a \vee b) \wedge (\neg c)) \vee ((a \vee b) \wedge d) \\
&= (a \wedge \neg c) \vee (b \wedge \neg c) \vee (a \wedge d) \vee (b \wedge d)
\end{aligned}$$

Converting to DNF

- Every formula can be converted to DNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \wedge (\varphi_2 \vee \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
- Example:

$$\begin{aligned}
\varphi \quad & = (a \vee b) \wedge (\neg c \vee d) \\
& = ((a \vee b) \wedge (\neg c)) \vee ((a \vee b) \wedge d) \\
& = (a \wedge \neg c) \vee (b \wedge \neg c) \vee (a \wedge d) \vee (b \wedge d)
\end{aligned}$$

- Now consider $\varphi_n = (a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \ldots \wedge (a_n \vee b_n)$.
- Q: How many clauses will the DNF have?

Converting to DNF

- Every formula can be converted to DNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \wedge (\varphi_2 \vee \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
- Example:

$$
\begin{aligned}
\varphi \quad &= (a \vee b) \wedge (\neg c \vee d) \\
&= ((a \vee b) \wedge (\neg c)) \vee ((a \vee b) \wedge d) \\
&= (a \wedge \neg c) \vee (b \wedge \neg c) \vee (a \wedge d) \vee (b \wedge d)
\end{aligned}
$$

- Now consider $\varphi_n = (a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \ldots \wedge (a_n \vee b_n)$.
- Q: How many clauses will the DNF have?

  A: 2^n

Satisfiability of DNF

- Q: Is the following DNF formula satisfiable?

$$(a_1 \wedge a_2 \wedge \neg a_1) \vee (a_2 \wedge a_1) \vee (a_2 \wedge \neg a_3 \wedge a_3)$$

Satisfiability of DNF

- Q: Is the following DNF formula satisfiable?

$$(a_1 \land a_2 \land \neg a_1) \lor (a_2 \land a_1) \lor (a_2 \land \neg a_3 \land a_3)$$

A: Yes, because the term $a_2 \land a_1$ is satisfiable.

Satisfiability of DNF

- Q: Is the following DNF formula satisfiable?

  $$(a_1 \wedge a_2 \wedge \neg a_1) \vee (a_2 \wedge a_1) \vee (a_2 \wedge \neg a_3 \wedge a_3)$$

  A: Yes, because the term $a_2 \wedge a_1$ is satisfiable.
- Q: What is the complexity of the satisfiability check of DNF formulae?

Satisfiability of DNF

- Q: Is the following DNF formula satisfiable?

  $$(a_1 \land a_2 \land \neg a_1) \lor (a_2 \land a_1) \lor (a_2 \land \neg a_3 \land a_3)$$

  A: Yes, because the term $a_2 \land a_1$ is satisfiable.
- Q: What is the complexity of the satisfiability check of DNF formulae?

  A: Linear (time and space).

Satisfiability of DNF

- Q: Is the following DNF formula satisfiable?

  $$(a_1 \wedge a_2 \wedge \neg a_1) \vee (a_2 \wedge a_1) \vee (a_2 \wedge \neg a_3 \wedge a_3)$$

  A: Yes, because the term $a_2 \wedge a_1$ is satisfiable.
- Q: What is the complexity of the satisfiability check of DNF formulae?

  A: Linear (time and space).
- Q: Can there be any polynomial transformation into DNF?

Satisfiability of DNF

- Q: Is the following DNF formula satisfiable?

  $$(a_1 \land a_2 \land \neg a_1) \lor (a_2 \land a_1) \lor (a_2 \land \neg a_3 \land a_3)$$

  A: Yes, because the term $a_2 \land a_1$ is satisfiable.
- Q: What is the complexity of the satisfiability check of DNF formulae?

  A: Linear (time and space).
- Q: Can there be any polynomial transformation into DNF?
- A: No, it would violate the NP-completeness of the problem.

Conjunctive Normal Form (CNF)

- Definition: A formula is said to be in Conjunctive Normal Form (CNF) iff it is a conjunction of clauses.

Conjunctive Normal Form (CNF)

- Definition: A formula is said to be in Conjunctive Normal Form (CNF) iff it is a conjunction of clauses.
- In other words, it is a formula of the form

$$\bigwedge_i \left(\bigvee_j l_{i,j} \right)$$

where $l_{i,j}$ is the j-th literal in the i-th clause.

- Definition: A formula is said to be in Conjunctive Normal Form (CNF) iff it is a conjunction of clauses.
- In other words, it is a formula of the form

$$\bigwedge_i \left(\bigvee_j l_{i,j} \right)$$

where $l_{i,j}$ is the j-th literal in the i-th clause.

- Example:

$$\varphi = (a \vee \neg b \vee c) \wedge (\neg a \vee d) \wedge (b) \quad \textit{is in CNF}$$

Conjunctive Normal Form (CNF)

- Definition: A formula is said to be in Conjunctive Normal Form (CNF) iff it is a conjunction of clauses.
- In other words, it is a formula of the form

$$\bigwedge_i \left(\bigvee_j l_{i,j} \right)$$

  where $l_{i,j}$ is the j-th literal in the i-th clause.
- Example:

$$\varphi = (a \vee \neg b \vee c) \wedge (\neg a \vee d) \wedge (b) \quad \textit{is in CNF}$$

- Also CNF is a special case of NNF.

Converting to CNF

- Every formula can be converted to CNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \vee (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$

Converting to CNF

- Every formula can be converted to CNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \vee (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
- Consider the formula $\varphi = (a_1 \wedge b_1) \vee (a_2 \wedge b_2)$.

  Transformation: $(a_1 \vee a_2) \wedge (a_1 \vee b_2) \wedge (b_1 \vee a_2) \wedge (b_1 \vee b_2)$

Converting to CNF

- Every formula can be converted to CNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \vee (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
- Consider the formula $\varphi = (a_1 \wedge b_1) \vee (a_2 \wedge b_2)$.

  Transformation: $(a_1 \vee a_2) \wedge (a_1 \vee b_2) \wedge (b_1 \vee a_2) \wedge (b_1 \vee b_2)$
- Now consider $\varphi_n = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \ldots \vee (a_n \wedge b_n)$.

  Q: How many clauses does the resulting CNF have?

Converting to CNF

- Every formula can be converted to CNF in exponential time and space:
  1. Convert to NNF
  2. Distribute disjunctions following the rule:
     $\models \varphi_1 \vee (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
- Consider the formula $\varphi = (a_1 \wedge b_1) \vee (a_2 \wedge b_2)$.

  Transformation: $(a_1 \vee a_2) \wedge (a_1 \vee b_2) \wedge (b_1 \vee a_2) \wedge (b_1 \vee b_2)$
- Now consider $\varphi_n = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \ldots \vee (a_n \wedge b_n)$.

  Q: How many clauses does the resulting CNF have?

  A: 2^n

Converting to CNF: Tseitin's encoding

- Every formula can be converted to CNF in linear time and space if new variables are added.
- The original and the converted formulae are not equivalent but equisatisfiable.

  Two formulae are equisatisfiable if both are, or are not, satisfiable simultaneously.

Converting to CNF: Tseitin's encoding

- Every formula can be converted to CNF in linear time and space if new variables are added.
- The original and the converted formulae are not equivalent but equisatisfiable.

  Two formulae are equisatisfiable if both are, or are not, satisfiable simultaneously.

- Consider the formula

  $\varphi = (a \to (b \wedge c))$

Converting to CNF: Tseitin's encoding

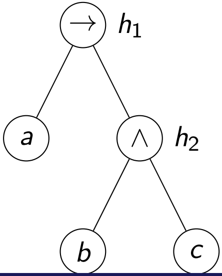
- Every formula can be converted to CNF in linear time and space if new variables are added.
- The original and the converted formulae are not equivalent but equisatisfiable.

  Two formulae are equisatisfiable if both are, or are not, satisfiable simultaneously.

- Consider the formula

  $\varphi = (a \to (b \land c))$

Parse tree:

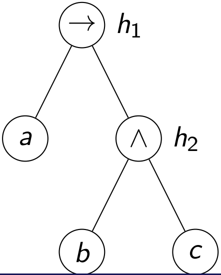


Converting to CNF: Tseitin's encoding

- Every formula can be converted to CNF in linear time and space if new variables are added.
- The original and the converted formulae are not equivalent but equisatisfiable.
  Two formulae are equisatisfiable if both are, or are not, satisfiable simultaneously.

- Consider the formula
  $\varphi = (a \rightarrow (b \wedge c))$
- Associate a new auxiliary variable with each gate.

Parse tree:



Converting to CNF: Tseitin's encoding

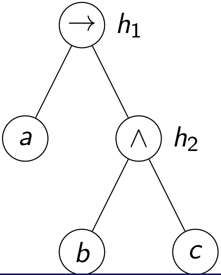
- Every formula can be converted to CNF in linear time and space if new variables are added.
- The original and the converted formulae are not equivalent but equisatisfiable.

  Two formulae are equisatisfiable if both are, or are not, satisfiable simultaneously.

- Consider the formula

  $\varphi = (a \to (b \wedge c))$
- Associate a new auxiliary variable with each gate.
- Add constraints that define these new variables.

Parse tree:



Converting to CNF: Tseitin's encoding

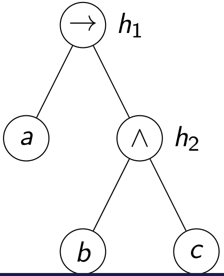
- Every formula can be converted to CNF in linear time and space if new variables are added.
- The original and the converted formulae are not equivalent but equisatisfiable.

  Two formulae are equisatisfiable if both are, or are not, satisfiable simultaneously.

- Consider the formula

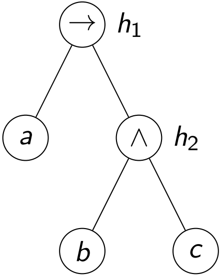
  $\varphi = (a \rightarrow (b \wedge c))$
- Associate a new auxiliary variable with each gate.
- Add constraints that define these new variables.
- Finally, enforce the root node.

Parse tree:



Converting to CNF: Tseitin's encoding

- Need to satisfy:
  $(h_1 \leftrightarrow (a \rightarrow h_2)) \wedge$
  $(h_2 \leftrightarrow (b \wedge c)) \wedge$
  (h_1)



- Each gate encoding has a CNF representation with 3 or 4 clauses.

Converting to CNF: Tseitin's encoding

- Need to satisfy:
  $(h_1 \leftrightarrow (a \rightarrow h_2)) \wedge (h_2 \leftrightarrow (b \wedge c)) \wedge (h_1)$

Converting to CNF: Tseitin's encoding

- Need to satisfy:
  $(h_1 \leftrightarrow (a \rightarrow h_2)) \wedge (h_2 \leftrightarrow (b \wedge c)) \wedge (h_1)$

- First: $(h_1 \vee a) \wedge (h_1 \vee \neg h_2) \wedge (\neg h_1 \vee \neg a \vee h_2)$

Converting to CNF: Tseitin's encoding

- Need to satisfy:

  $(h_1 \leftrightarrow (a \rightarrow h_2)) \wedge (h_2 \leftrightarrow (b \wedge c)) \wedge (h_1)$

- First: $(h_1 \vee a) \wedge (h_1 \vee \neg h_2) \wedge (\neg h_1 \vee \neg a \vee h_2)$
- Second: $(\neg h_2 \vee b) \wedge (\neg h_2 \vee c) \wedge (h_2 \vee \neg b \vee \neg c)$

Converting to CNF: Tseitin's encoding

- Let's go back to
  $\varphi_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$

Converting to CNF: Tseitin's encoding

- Let's go back to
  $\varphi_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$
- With Tseitin's encoding we need:
  - n auxiliary variables $h_1, \ldots, h_n$.
  - Each adds 3 constraints.
  - Top clause: $(h_1 \vee \cdots \vee h_n)$

Converting to CNF: Tseitin's encoding

- Let's go back to

  $\varphi_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$

- With Tseitin's encoding we need:
  - n auxiliary variables $h_1, \ldots, h_n$.
  - Each adds 3 constraints.
  - Top clause: $(h_1 \vee \cdots \vee h_n)$

- Hence, we have
  - $3n + 1$ clauses, instead of 2^n.
  - $3n$ variables rather than $2n$.

Propositional logic - Outline

- Syntax of propositional logic
- Semantics of propositional logic
- Satisfiability and validity
- Normal forms