Logic and Proof Hilary 2026

Propositional Logic Refresher

James Worrell

This lecture reviews facts about propositional logic that should be familiar to you. We also establish terminology and conventions that will be used throughout the course.

1 Syntax

Let $p_1, p_2, \ldots$ be an infinite set of *propositional variables*. The set of formulas of propositional logic is defined inductively as follows:

1. **true** and **false** are formulas;
2. All propositional variables are formulas;
3. For every formula F, $\neg F$ is a formula;
4. For all formulas F and G, $(F \wedge G)$ and $(F \vee G)$ are formulas.

We call $(F \wedge G)$ the *conjunction* of F and G, $(F \vee G)$ is the *disjunction* of F and G, and $\neg F$ the *negation* of F. We read $F \wedge G$ as "F and G", we read $F \vee G$ as "F or G", and we read $\neg F$ as "not F". The parentheses around conjunction and disjunction ensure that each string of symbols generated by the above definition can be parsed uniquely. Typically we omit the outermost parentheses when writing formulas, e.g., we write $p_1 \wedge (p_2 \vee p_1)$ instead of $(p_1 \wedge (p_2 \vee p_1))$.

We introduce three derived connectives, the *conditional* $(F \to G)$ (read "if F then G"), the *biconditional* $(F \leftrightarrow G)$ (read "F if and only if G") and *exclusive or* $(F \oplus G)$. We define these syntactically in terms of the existing connectives as follows.

$$\begin{aligned} F \to G &:= (\neg F \vee G) \\ F \leftrightarrow G &:= (F \to G) \wedge (G \to F) \\ F \oplus G &:= (F \wedge \neg G) \vee (\neg F \wedge G) \end{aligned}$$

It is also useful to have indexed versions of disjunction and conjunction, similar to indexed sums and products in arithmetic. We thus define

$$\begin{aligned} \bigvee_{i=1}^{n} F_i &:= (\ldots((F_1 \vee F_2) \vee F_3) \vee \ldots \vee F_n) \\ \bigwedge_{i=1}^{n} F_i &:= (\ldots((F_1 \wedge F_2) \wedge F_3) \wedge \ldots \wedge F_n) \end{aligned}$$

To avoid clutter we adopt the following *operator precedences*: $\leftrightarrow$ and $\to$ bind weaker than $\wedge$ and $\vee$, which in turn bind weaker than $\neg$. Indexed conjunction and disjunction bind weaker than any of the above operators. For example, we can write $\neg p \wedge q \to r$ instead of $((\neg p \wedge q) \to r)$. However well-chosen parenthesis can often help parsing formulas.

2 Semantics

2.1 Assignments and Satisfiability

We call $\{0,1\}$ the set of *truth values*. A *valuation* is a function $\mathsf{v}\colon \{p_i : i \in \mathbb{N}\} \to \{0,1\}$ from the set of propositional variables to the set of truth values. We also call v an *assignment*. We extend v to a function on the set of all propositional formulas by structural induction. The base case of the induction are $\mathsf{v}[\![\mathbf{true}]\!] := 1$ and $\mathsf{v}[\![\mathbf{false}]\!] := 0$. The inductive cases are as follows:

1. $\mathsf{v}[\![F \wedge G]\!] := \min(\mathsf{v}[\![F]\!], \mathsf{v}[\![G]\!])$;
2. $\mathsf{v}[\![F \vee G]\!] := \max(\mathsf{v}[\![F]\!], \mathsf{v}[\![G]\!])$;
3. $\mathsf{v}[\![\neg F]\!] := 1 - \mathsf{v}[\![F]\!]$.

Let F be a formula and let v be an assignment. If $\mathsf{v}[\![F]\!] = 1$ then we say that v *satisfies* F and write $\mathsf{v} \models F$. Otherwise we write $\mathsf{v} \not\models F$. A formula F is *satisfiable* if it is satisfied by some valuation; otherwise F is called *unsatisfiable*. A (finite or infinite) set of formulas $\boldsymbol{S}$ is *satisfiable* if there is a valuation that satisfies every formula in $\boldsymbol{S}$. A formula F is *valid* or a *tautology* if all assignments are models of F. Note that F is unsatisfiable if and only if $\neg F$ is valid.

2.2 Entailment and Equivalence

We say that a formula G is *entailed* by a set of formulas $\boldsymbol{S}$ if every assignment that satisfies each formula in $\boldsymbol{S}$ also satisfies G. In this case we write $\boldsymbol{S} \models G$. If $\boldsymbol{S} = \{F_1, \ldots, F_n\}$ then we write $F_1, \ldots, F_n \models G$, and if $\boldsymbol{S} = \emptyset$ then we write $\models G$. Observe that $\models G$ just asserts that G is valid.

> **Warning!** The symbol $\models$ is overloaded. Above we define $\boldsymbol{S} \models F$ for a set of formulas $\boldsymbol{S}$ and formula F. Previously we have written $\mathsf{v} \models F$ to say that an assignment v is a model of F.

Two formulas F and G are said to be *logically equivalent* if $\mathsf{v}[\![F]\!] = \mathsf{v}[\![G]\!]$ for every assignment v. We write $F \equiv G$ to denote that F and G are equivalent. (We reserve the symbol $=$ for syntactic equality, i.e., $F = G$ means that F and G are the same formula.)

2.3 Normal Forms

A *literal* is a propositional variable or the negation of a propositional variable. In the former case the literal is *positive* and in the latter case it is *negative*. A formula F is in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, where each clause is a disjunction of literals $L_{i,j}$:

$$F = \bigwedge_{i=1}^{n} \Big(\bigvee_{j=1}^{m_i} L_{i,j} \Big).$$

A formula F is in *disjunctive normal form* (DNF) if it is a disjunction of clauses, where each clause is a conjunction of literals $L_{i,j}$:

$$F = \bigvee_{i=1}^{n} \Big(\bigwedge_{j=1}^{m_i} L_{i,j} \Big).$$

Every formula is logically equivalent to one in CNF as well as to one in DNF. In particular, the CNF formula with no clauses is **true**, while the CNF formula consisting of a single clause that contains no literals is **false**.

2.4 Substitution

Given a formula F and propositional variable p we define a new formula $G[F/p]$ (read "G with F substituted for all occurrences of p") by structural induction as follows. For a propositional variable q we have

$$q[F/p] := \begin{cases} F & \text{if } q = p \\ q & \text{if } q \neq p \end{cases}$$

Otherwise we inductively define

$$\begin{aligned} (G_1 \wedge G_2)[F/p] &:= G_1[F/p] \wedge G_2[F/p] \\ (G_1 \vee G_2)[F/p] &:= G_1[F/p] \vee G_2[F/p] \\ (\neg G_1)[F/p] &:= \neg(G_1[F/p]) \end{aligned}$$

For example, $(p_1 \wedge (p_2 \vee p_1))[\neg q_1/p_1] = \neg q_1 \wedge (p_2 \vee \neg q_1)$. Note that all occurrences of p_1 are replaced with $\neg q_1$.

The above definition was purely syntactic. Next we define a semantic counterpart—the *update* operation on assignments. Given an assignment v, propositional variable p, and truth value $b \in \{0, 1\}$, define the assignment $\mathsf{v}_{[p \mapsto b]}$ by

$$\mathsf{v}_{[p \mapsto b]}[\![q]\!] = \begin{cases} b & \text{if } q = p \\ \mathsf{v}[\![q]\!] & \text{if } q \neq p \end{cases}$$

for each propositional variable q.

Substitution and update are related in the following lemma.

Lemma 1 (Substitution Lemma)**.** Given formulas F, G, propositional variable p, and assignment v, we have $\mathsf{v}[\![G[F/p]]\!] = \mathsf{v}_{[p \mapsto \mathsf{v}[\![F]\!]]}[\![G]\!]$.

Proof. The proof is by induction on G, exploiting the inductive definition of substitution.

The first base case is that $G = p$. Then we have

$$\mathsf{v}[\![p[F/p]]\!] = \mathsf{v}[\![F]\!] = \mathsf{v}_{[p \mapsto \mathsf{v}[\![F]\!]]}[\![p]\!] .$$

The second base case is that $G = q$ for some propositional variable q different from p. Then

$$\mathsf{v}[\![q[F/p]]\!] = \mathsf{v}[\![q]\!] = \mathsf{v}_{[p \mapsto \mathsf{v}[\![F]\!]]}[\![q]\!] .$$

The induction case for conjunction is as follows. If $G = G_1 \wedge G_2$ then

$$\begin{aligned} \mathsf{v} \models (G_1 \wedge G_2)[F/p] \quad &\text{iff} \quad \mathsf{v} \models G_1[F/p] \wedge G_2[F/p] \\ &\text{iff} \quad \mathsf{v} \models G_1[F/p] \ \text{ and } \ \mathsf{v} \models G_2[F/p] \\ &\text{iff} \quad \mathsf{v}_{[p \mapsto \mathsf{v}[\![F]\!]]} \models G_1 \ \text{ and } \ \mathsf{v}_{[p \mapsto \mathsf{v}[\![F]\!]]} \models G_2 \qquad \text{(induction hypothesis)} \\ &\text{iff} \quad \mathsf{v}_{[p \mapsto \mathsf{v}[\![F]\!]]} \models G_1 \wedge G_2 \end{aligned}$$

The induction cases for disjunction and negation are similar and are omitted. □

3 The SAT Problem

A decision problem is a computational problem for which the output is either "yes" or "no". Such a problem consists of a family of *inputs* or *instances*, together with a question that can be applied to each instance. In this course a central decision problem is the *SAT problem* for propositional logic. Here the instances are propositional formulas and the question is whether the given formula is satisfiable.

3.1 Complexity of SAT

The truth-table method for solving the SAT problem requires at least 2^n steps in the worst case for a formula with n variables. Nevertheless modern SAT solvers work well in practice, routinely determining (un)satisfiability of formulas with thousands of variables and clauses. These solvers are based on a proof system called *resolution*. However, as we will see later in the course, resolution has worst-case exponential time complexity.

It is an open question whether there is an algorithm for deciding SAT whose worst-case running time is polynomial in the formula size. In fact this question is a formulation of the famous $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem. It is even open whether there is a *sub-exponential* algorithm for the SAT problem. By a sub-exponential algorithm we mean that the running time $f(n)$ is $2^{o(n)}$, e.g., $f(n)$ could be n^{600}, $n^{\log(n)}$, $n^{\sqrt{n}}$, or $2^{n/\log(n)}$. In other words, it is not known whether or not we can do even marginally better than exhaustive search in the worst case.

3.2 SAT Encodings

Many "hard" combinatorial decision problems can be reduced to SAT. A reduction of a decision problem to SAT is an algorithm that inputs an instance I of the decision problem and outputs a propositional formula φ_I such that φ_I is satisfiable if and only if I is a "yes" instance.

Example 2. We consider the *3-colourability problem* for graphs. Recall that an undirected graph is a tuple $G = (V, E)$ consisting of a set of *vertices* V and an irreflexive symmetric edge relation $E \subseteq V \times V$. If $(u, v) \in E$ we say that vertices u and v are *adjacent*. A 3-*colouring* of G is an assignment of an element of the set of colours $C = \{r, b, g\}$ to each vertex so that no two adjacent vertices have the same colour. An instance of the 3-colouring problem is a graph G, and the question is whether G has a 3-colouring.

We express the requirements of a 3-colouring in a propositional formula φ_G that is derived from G. To define φ_G we first introduce a set of atomic propositions $\{p_{v,c} : v \in V, c \in C\}$. Intuitively $p_{v,c}$ represents the proposition *vertex v has colour c*. We then encode the notion of a 3-colouring by the following formulas.

- Each vertex has at least one colour:
$$F_1 := \bigwedge_{v \in V} \bigvee_{c \in C} p_{v,c} \,.$$
- Each vertex has at most one colour:
$$F_2 := \bigwedge_{v \in V} \bigwedge_{\substack{c,c' \in C \\ c \neq c'}} \neg p_{v,c} \vee \neg p_{v,c'} \,.$$
- Adjacent vertices have different colours:
$$F_3 := \bigwedge_{(u,v) \in E} \bigwedge_{c \in C} \neg p_{u,c} \vee \neg p_{v,c} \,.$$

Finally, we define $\varphi_G := F_1 \wedge F_2 \wedge F_3$. Note that it is straightforward to write a small program that takes a graph as G input and outputs the formula φ_G. It is clear that φ_G is satisfiable if and only if G has a 3-colouring and moreover a satisfying assignment of φ_G determines a 3-colouring of G.

The idea of solving a combinatorial problem by reduction to SAT is that the SAT-solver should do the hard work. The reduction itself should be computationally straightforward: at the very least it should be implementable by a polynomial-time algorithm. For example, in the case of 3-colourability, given a graph G one can produce φ_G by performing a single traversal of G.