Logic and Proof Hilary 2016

Propositional Logic

James Worrell

1 Introduction

Propositional logic analyses how the truth values of compound sentences depend on their constituents. The most basic kind of sentences are *atomic propositions*, which can either be true or false independently of each other. Sentences are combined using *logical connectives*, such as *not* ($\neg$), *or* ($\vee$), and *implies* ($\to$). A prime concern of propositional logic is, *given a compound sentence, determine which truth values of its atoms make it true.* This question is key to formulating the notions of *logical consequence* and *valid argument.*

Consider the following atomic propositions:

A	Alice is an architect
B	Bob is a builder
C	Charlie is a cook

From these one can build the following compound sentences using the connectives. For example,

$\neg C$	Charlie is not a cook
$A \vee B$	Alice is an architect or Bob is a builder
$B \to C$	If Bob is a builder then Charlie is a cook

The above three propositions entail that Alice is an architect, i.e., if the above three propositions are all true then A must also be true. We denote this fact by the *entailment*

$$\neg C,\, A \vee B,\, B \to C \models A\,.$$

The correctness of this entailment is independent of the meaning of the atomic propositions. It has nothing to do with specific facts about building or architecture, rather it is determined by the meaning of the logical connectives. As far as propositional logic is concerned, atomic propositions are just things that are true or false, independently of each other. To make this clear we have represented the atomic propositions in this example by *propositional variables* A, B and C. In our semantics for propositional logic each propositional variable takes either the value 1 or 0, standing for true and false respectively.

The above entailment was reasonably intuitive, but imagine trying to justify an entailment involving tens or hundreds of sentences and variables. Clearly one would quickly get confused. One of the main aims in this course is to introduce systematic procedures supporting the calculation of such logical consequences.

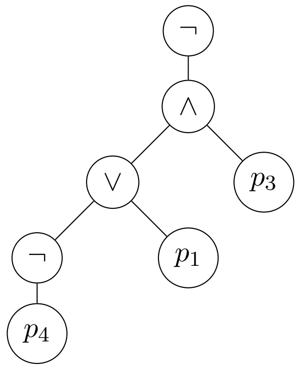

Figure 1: Syntax tree of $\neg((\neg p_4 \vee p_1) \wedge p_3)$.

2 Syntax of Propositional Logic

2.1 The Core Language

The syntax of propositional logic is given by rules for writing down *well-formed formulas* over a given collection of *propositional variables* $p_1, p_2, \ldots$. 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 in exactly one way as a formula. Typically we omit the outermost parentheses when writing formulas, e.g., we write $p_1 \wedge (p_2 \vee p_1)$ instead of $(p_1 \wedge (p_2 \vee p_1))$.

Every formula F can be represented by a *syntax tree* whose internal nodes are labelled by connectives and whose leaves are labelled by propositional variables. The size of F is defined to be the number of nodes in its syntax tree. Each node in the syntax tree determines a *subformula* of F whose syntax tree is the subtree rooted at that node.

Example 1. $\neg((\neg p_4 \vee p_1) \wedge p_3)$ is a formula. Its subformulas are p_1, p_3, p_4, $\neg p_4$, $(\neg p_4 \vee p_1)$, $((\neg p_4 \vee p_1) \wedge p_3)$, and $\neg((\neg p_4 \vee p_1) \wedge p_3)$. The syntax tree of this formula is shown in Figure 1.

The inductive structure of the set of propositional formulas allows us to define functions on propositional formulas by induction. For example, we use induction to give a more formal definition of the size of a formula as follows:

1. size(**true**) $= 1$, size(**false**) $= 1$, and size(P) $= 1$ for a propositional variable P

2. $\text{size}(\neg F) = 1 + \text{size}(F)$
3. $\text{size}(F \wedge G) = 1 + \text{size}(F) + \text{size}(G)$
4. $\text{size}(F \vee G) = 1 + \text{size}(F) + \text{size}(G)$

Exercise 2. Let $\text{sub}(F)$ denote the set of subformulas of a formula F. Given an inductive definition of $\text{sub}(F)$.

2.2 Derived Connectives

Having a small set of primitive connectives makes it easier to implement our logic and to prove properties about it. However in applications it is typically helpful to have a rich set of *derived connectives* to hand. These are not part of the official language, but can be considered as macros.

We introduce three derived connectives, the *conditional* ($F \rightarrow G$ (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.

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

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

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

We adopt the following *operator precedences*: $\leftrightarrow$ and $\rightarrow$ 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. We also typically omit the outermost parentheses. For example, we can write $\neg P \wedge Q \rightarrow R$ instead of $((\neg P \wedge Q) \rightarrow R)$. However well-chosen parenthesis can often help to parse formulas.

3 Semantics of Propositional Logic

3.1 Assignments and Satisfiability

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

1. $\mathcal{A}[\![F \wedge G]\!] = 1$ if and only if $\mathcal{A}[\![F]\!] = 1$ and $\mathcal{A}[\![G]\!] = 1$;

$\mathcal{A}[\![F]\!]$	$\mathcal{A}[\![G]\!]$	$\mathcal{A}[\![F \vee G]\!]$
0	0	0
0	1	1
1	0	1
1	1	1

$\mathcal{A}[\![F]\!]$	$\mathcal{A}[\![G]\!]$	$\mathcal{A}[\![F \wedge G]\!]$
0	0	0
0	1	0
1	0	0
1	1	1

$\mathcal{A}[\![F]\!]$	$\mathcal{A}[\![\neg F]\!]$
0	1
1	0

Figure 2: Truth tables for the Boolean connectives

$\mathcal{A}[\![F]\!]$	$\mathcal{A}[\![G]\!]$	$\mathcal{A}[\![F \rightarrow G]\!]$
0	0	1
0	1	1
1	0	0
1	1	1

$\mathcal{A}[\![F]\!]$	$\mathcal{A}[\![G]\!]$	$\mathcal{A}[\![F \leftrightarrow G]\!]$
0	0	1
0	1	0
1	0	0
1	1	1

$\mathcal{A}[\![F]\!]$	$\mathcal{A}[\![G]\!]$	$\mathcal{A}[\![F \oplus G]\!]$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3: Truth tables for the conditional, biconditional and exclusive or

2. $\mathcal{A}[\![F \vee G]\!] = 1$ if and only if $\mathcal{A}[\![F]\!] = 1$ or $\mathcal{A}[\![G]\!] = 1$;
3. $\mathcal{A}[\![\neg F]\!] = 1$ if and only if $\mathcal{A}[\![F]\!] \neq 1$.

These definitions are summarised by *truth tables* in Figure 2.

Figure 3 gives truth tables for the derived operators. Observe that $F \rightarrow G$ is true if F is false. In this case we say that the implication holds *vacuously*.

Let F be a formula and let $\mathcal{A}$ be an assignment. If $\mathcal{A}[\![F]\!] = 1$ then we write $\mathcal{A} \models F$. In this case we say that $\mathcal{A}$ *satisfies* F or that $\mathcal{A}$ is a *model* of F. Otherwise we write $\mathcal{A} \not\models F$, and say that $\mathcal{A}$ does not satisfy F.

A formula F is *satisfiable* if it has a model, otherwise F is called *unsatisfiable*. A (finite or infinite) set of formulas $\boldsymbol{S}$ is *satisfiable* if there is an assignment that is a model of every formula in $\boldsymbol{S}$. (Thus sets of formulas are treated conjunctively.) 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.

Example 3. In the following example we analyse the consistency of a sequence of assertions in natural language by translating them to a propositional formula and computing its truth table.

A device consists of a thermostat, a pump, and a warning light. Suppose we are told the following four facts about the pump:

(a) The thermostat or the pump (or both) are broken.

(b) If the thermostat is broken then the pump is also broken.

(c) If the pump is broken and the warning light is on then the thermostat is not broken.

(d) The warning light is on.

We want to find out if it is possible for the above sentences to all be true at the same time. To this end we introduce atomic propositions T (*the thermostat is broken*), P (*the pump is broken*), and W (*the warning light is on*). Then statements (a)–(d) are expressed by the following formula:

$$F := (T \vee P) \wedge (T \rightarrow P) \wedge ((P \wedge W) \rightarrow \neg T) \wedge W\,.$$

T	P	W	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 4: Truth table for the thermostat

The truth table for F in Figure 4 reveals that it is satisfied by a unique assignment. One way to think of this is that each assignment describes a possible world, and there is only possible world in which F is true.

3.2 Sudoku Example

Example 3 shows that propositional logic is an expressive *constraint language.* The following example further illustrates this point.

Example 4. We transform an instance of the Sudoku puzzle into an instance of propositional satisfiability, see Figure 5. For each choice of $i, j, k \in \{1, \ldots, 9\}$ we have a proposition $P_{i,j,k}$ expressing that *grid position i, j contains number k.* We then build a formula F as the conjunction of the following constraints:

- Each number appears in each row:

$$F_1 \stackrel{\text{def}}{=} \bigwedge_{i=1}^{9} \bigwedge_{k=1}^{9} \bigvee_{j=1}^{9} P_{i,j,k}$$

- Each number appears in each column:

$$F_2 \stackrel{\text{def}}{=} \bigwedge_{j=1}^{9} \bigwedge_{k=1}^{9} \bigvee_{i=1}^{9} P_{i,j,k}$$

- Each number appears in each 3×3 block:

$$F_3 \stackrel{\text{def}}{=} \bigwedge_{k=1}^{9} \bigwedge_{u=0}^{2} \bigwedge_{v=0}^{2} \bigvee_{i=1}^{3} \bigvee_{j=1}^{3} P_{3u+i,3v+j,k}$$

- No square contains two numbers:

$$F_4 \stackrel{\text{def}}{=} \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{1 \le k < k' \le 9} \neg(P_{i,j,k} \wedge P_{i,j,k'})\,.$$

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Figure 5: A Sudoko grid

- Certain numbers appear in certain positions: For the puzzle in Figure 5 we assert

$$F_5 \overset{\text{def}}{=} P_{2,1,2} \wedge P_{1,2,8} \wedge P_{2,3,3} \wedge \ldots \wedge P_{8,9,6} \,.$$

It might appear at first sight that some constraints are missing. For example, we might want to express that no number appears twice in the same row:

$$F_6 \overset{\text{def}}{=} \bigwedge_{i=1}^{9} \bigwedge_{k=1}^{9} \bigwedge_{1 \leq j < j' < 9} \neg(P_{i,j,k} \wedge P_{i,j',k}) \,.$$

But this is entailed by the existing formulas: adding F_6 as an extra constraint would not change the set of satisfying assignments. (It turns out that adding logically redundant constraints may be helpful to a computer that is searching for an assignment that satisfies the formula.)

The number of variables in our encoding is $9^3 = 729$. Thus a truth table for the corresponding formula would have $2^{729} > 10^{200}$ lines, which is greater than the estimated number of atoms in the universe! Nevertheless a modern SAT-solver (see, e.g., `www.minisat.se`) can typically find a satisfying assignment in milliseconds.

3.3 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 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.

Example 5. In the Sudoku puzzle in Example 4 we have $F_1, \ldots, F_4 \models F_6$.

Warning! In logic 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 $\mathcal{A} \models F$ to say that an assignment $\mathcal{A}$ is a model of F.

Two formulas F and G are said to be *logically equivalent* if $\mathcal{A}[\![F]\!] = \mathcal{A}[\![G]\!]$ for every assignment $\mathcal{A}$. 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.) For example, an implication $F \to G$ is equivalent to its so-called *contrapositive* $\neg G \to \neg F$. This fact is often used in mathematical proofs, where it may be more intuitive to prove the contrapositive than the original implication.

Exercise 6. Fix $n \in \mathbb{N}$. Argue that there are finitely many formulas on the set of propositional variables $p_1, \ldots, p_n$ up to logical equivalence. How many such formulas are there?

4 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 *instances*, together with a question that can be applied to each instance. A decision problem of prime importance is the *SAT problem* for propositional logic. Here the instances are propositional formulas and the question is whether a given formula is satisfiable.

4.1 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, that is, it runs in no better than exponential time. The proof system underlying modern SAT solvers can be seen as a subsystem of the *resolution proof procedure*, which will be introduced later on. These SAT solvers work well in practice, routinely determining (un)satisfiability of formulas with thousands of variables and clauses. However it is known that resolution is also exponential in the worst case.

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 $\mathbf{P} \overset{?}{=} \mathbf{NP}$ 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 a tiny bit better than exhaustive search in the worst case!

4.2 Reductions to SAT

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 7. 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 a colour in the set $C = \{r, b, g\}$ 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 all 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.

Exercise 8. Consider the following two decision problems concerning propositional formulas:

1. *Entailment:* Let F and G be formulas. Does $F \models G$ hold?
2. *Equivalence:* Let F and G be formulas. Does $F \equiv G$ hold?

Describe how a procedure to solve the SAT problem could be used as a *black box* to solve the Entailment and Equivalence problems.