Lecture 2

Propositional logic

syntax and semantics, the satisfiability problem, constraint problems

Print version of the lecture in *Logic and Proof*

presented on 1 May 2019

by Dr Christoph Haase

1 Propositional logic

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* ($\rightarrow$). 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*.

For an example, consider the following atomic propositions and compound sentences:

- Atomic propositions:

a	"Alice is an architect"
b	"Bob is a builder"
c	"Charlie is a cook"

- Compound sentences:

$\neg c$	"Charlie is not a cook"
$a \vee b$	"Alice is an architect or Bob is a builder"
$b \rightarrow c$	"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*

$$\neg c,\ a \vee b,\ b \rightarrow c \models a\,.$$

The correctness of this entailment is independent of the meaning of the atomic propositions. It has nothing to do with specific facts about buildings 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.

2 Syntax and semantics of propositional logic

The syntax of propositional logic is given by rules for writing down *well-formed formulas*.

Definition 1 (Syntax of propositional logic)**.** Let $X = \{x_1, x_2, x_3, \ldots\}$ be a countably infinite set of **propositional variables**. **Formulas** of propositional logic are inductively defined as follows:

1. *true* and *false* are formulas.
2. Every propositional variable x_i is a formula.
3. If F is a formula, then $\neg F$ is a formula.
4. If F and G are formulas, then $(F \wedge G)$ and $(F \vee G)$ are formulas.

Further to this definition, we introduce some additional notation:

- We often write x, y, z or p to denote propositional variables.
- We call $\neg F$ the **negation** of F.
- Given formulas F and G, $(F \wedge G)$ is the **conjunction** of F and G, and $(F \vee G)$ is the **disjunction** of F and G.
- We call $\neg, \wedge$ and $\vee$ **logical connectives**.
- We denote by $\mathcal{F}(X)$ the **set of all formulas** built from propositional variables in X.

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.

- **Implication**: $(F_1 \to F_2) := (\neg F_1 \vee F_2)$
- **Bi-implication**: $(F_1 \leftrightarrow F_2) := (F_1 \to F_2) \wedge (F_2 \to F_1)$
- **Exclusive Or**: $(F_1 \oplus F_2) := (F_1 \wedge \neg F_2) \vee (\neg F_1 \wedge F_2)$
- **Indexed Conjunction**: $\bigwedge_{i=1}^{n} F_i := (\cdots((F_1 \wedge F_2) \wedge F_3) \wedge \cdots \wedge F_n)$
- **Indexed Disjunction**: $\bigvee_{i=1}^{n} F_i := (\cdots((F_1 \vee F_2) \vee F_3) \vee \cdots \vee F_n)$

Note that implication and bi-implication are often also called **conditional** and **biconditional**, respectively.

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. We also typically omit the outermost parentheses. For example, we can write $\neg x \wedge y \to z$ instead of $((\neg x \wedge y) \to z)$. However well-chosen parenthesis can often help to parse formulas.

Every formula F can be represented by a *syntax tree*, which is 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. Figure 1 illustrates the syntax tree of the formula $\neg((\neg x_4 \vee x_1) \wedge x_3)$.

The inductive definition of formulas allows us to define functions on formulas by **structural induction**, by defining the function

- For the base cases *true*, *false* and x (propositional variables), and
- For the induction step for $\neg F$, $F \wedge G$ and $F \vee G$, i.e., all logical connectives

We give two examples of functions defined by structural induction below.

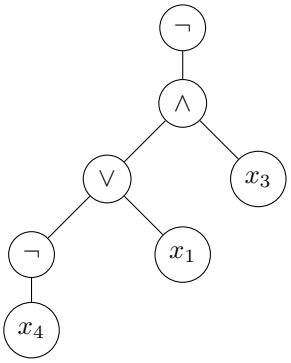

Figure 1: Example of a syntax tree.

Example 2. The function $size\colon \mathcal{F}(X) \to \mathbb{N}$ returns for a given formula the number of symbols required to write it down. Here, we assume that $true$ and $false$ account for just one symbol, and that brackets are omitted:

1. $size(true) = 1$, $size(false) = 1$
2. $size(x) = 1$ for all $x \in X$
3. $size(\neg F) = 1 + \text{size}(F)$
4. $size(F \wedge G) = 1 + \text{size}(F) + \text{size}(G)$
5. $size(F \vee G) = 1 + \text{size}(F) + \text{size}(G)$

The function $sub\colon \mathcal{F}(X) \to 2^{\mathcal{F}(X)}$ returning the set of all subformulas of a given formula can be defined by:

- $sub(true) = \{true\}$, $sub(false) = \{false\}$
- $sub(x) = \{x\}$ for all $x \in X$
- $sub(\neg F) = \{\neg F\} \cup sub(F)$
- $sub(F \wedge G) = \{F \wedge G\} \cup sub(F) \cup sub(G)$
- $sub(F \vee G) = \{F \vee G\} \cup sub(F) \cup sub(G)$

For instance,

$$\begin{aligned} &sub(\neg((\neg x_4 \vee x_1) \wedge x_3)) \\ &= \{x_1, x_3, x_4, \neg x_4, (\neg x_4 \vee x_1), ((\neg x_4 \vee x_1) \wedge x_3), \neg((\neg x_4 \vee x_1) \wedge x_3)\}. \end{aligned}$$

3 Semantics of propositional logic

So far, we have only seen how to write syntactically correct formulas of propositional logic. The syntax describes just a formal language, and does not give any meaning to formulas. The meaning, the *semantics*, of formulas will be the topic of this section, which is given in terms of **truth values** $\{0, 1\}$, where 0 indicates *false* and 1 indicates *true*, and is defined by structural induction.

Definition 3. An **assignment** is a function $\mathcal{A}\colon X \to \{0,1\}$ that induces an assignment $\hat{\mathcal{A}}\colon \mathcal{F}(X) \to \{0,1\}$ by structural induction as follows:

1. $\hat{\mathcal{A}}(false) = 0$, $\hat{\mathcal{A}}(true) = 1$
2. For every $x \in X$, $\hat{\mathcal{A}}(x) := \mathcal{A}(x)$
3. $\hat{\mathcal{A}}(\neg F) := \begin{cases} 1 & \text{if } \hat{\mathcal{A}}(F) = 0 \\ 0 & \text{otherwise} \end{cases}$
4. $\hat{\mathcal{A}}((F \wedge G)) := \begin{cases} 1 & \text{if } \hat{\mathcal{A}}(F) = 1 \text{ and } \hat{\mathcal{A}}(G) = 1 \\ 0 & \text{otherwise} \end{cases}$

5. $\hat{\mathcal{A}}((F \vee G)) := \begin{cases} 1 & \text{if } \hat{\mathcal{A}}(F) = 1 \text{ or } \hat{\mathcal{A}}(G) = 1 \\ 0 & \text{otherwise} \end{cases}$

Whenever $\hat{\mathcal{A}}(F) = 1$, we say that the formula F evaluates to true under the assignment $\mathcal{A}$. Otherwise, we say that F evaluates to false under $\mathcal{A}$.

Example 4. Let $F = (x \wedge \neg y) \vee z$ and $\mathcal{A}$ be an assignment such that $\mathcal{A}(x) = 1$ and $\mathcal{A}(y) = \mathcal{A}(z) = 0$. Then F evaluates to true under $\mathcal{A}$, since

$$\begin{aligned}
\hat{\mathcal{A}}(F) &= \begin{cases} 1 & \text{if } \hat{\mathcal{A}}((x \wedge \neg y)) = 1 \text{ or } \hat{\mathcal{A}}(z) = 1 \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } \hat{\mathcal{A}}((x \wedge \neg y)) = 1 \text{ (since } \mathcal{A}(z) = 0) \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } \hat{\mathcal{A}}(x) = 1 \text{ and } \hat{\mathcal{A}}(\neg y) = 1 \text{ (by definition of } \wedge) \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } \hat{\mathcal{A}}(y) = 0 \text{ (since } \mathcal{A}(x) = 1) \\ 0 & \text{otherwise} \end{cases} \\
&= 1 \text{ (since } \mathcal{A}(y) = 0).
\end{aligned}$$

In the following, for convenience we will omit writing the hat when dealing with the induced assignment $\hat{\mathcal{A}}$ of an assignment $\mathcal{A}$.

Since any logical connective only connects a finite number of formulas, we can alternatively give the semantics by explicitly listing the resulting truth value for all possible truth values of the subformulas. This gives rise to so-called truth tables.

Example 5. The semantics of logical connectives via **truth tables**:

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}(F \wedge G)$
0	0	0
1	0	0
0	1	0
1	1	1

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}(F \vee G)$
0	0	0
1	0	1
0	1	1
1	*1*	*1*

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}(F \to G)$
0	*0*	*1*
1	0	0
0	*1*	*1*
1	1	1

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}(F \oplus G)$
0	0	0
1	0	1
0	1	1
1	1	0

People new to logic get sometimes confused by the semantics of disjunction and implication. Observe that $x \vee y$ evaluates to true also if both x and y evaluate to true, whereas humans often use disjunction with the semantics of the exclusive or. Moreover, observe that an implication $x \to y$ evaluates to true in particular whenever x evaluates to false: from an incorrect premise we can draw any conclusion. In this case, we say that the implication holds *vacuously*.

4 Models, satisfiability and validity

We introduce a couple of further definitions of concepts that are central to propositional logic.

Definition 6. Let $F \in \mathcal{F}(X)$ and $\mathcal{A} \colon X \to \{0, 1\}$ be an assignment.

1. If $\mathcal{A}(F) = 1$ then we write $\mathcal{A} \models F$ ("F *holds under* $\mathcal{A}$", or "$\mathcal{A}$ is a **model** of F".)
2. If F has at least one model then F is **satisfiable**, otherwise F is **unsatisfiable**.

3. If F holds under any assignment $\mathcal{A}\colon X \to \{0,1\}$ then F is called **valid** or a **tautology**, written $\models F$.

In particular, the satisfiability problem plays an important role, not only in logic, but in the whole of computer science.

Definition 7. Given $F \in \mathcal{F}(X)$, the **Boolean satisfiability problem (SAT)** is to decide whether F is satisfiable.

In order to illustrate the concept of a tautology, the following example states four well-known tautologies. You can convince yourself of the fact that they actually are tautologies by constructing the corresponding truth tables.

Example 8. The subsequent first two tautologies are known as the *distributive laws*, the last two as *De Morgan's laws*:

$$\begin{aligned}
&\models (F \vee (G \wedge H)) \leftrightarrow ((F \vee G) \wedge (F \vee H))\\
&\models (F \wedge (G \vee H)) \leftrightarrow ((F \wedge G) \vee (F \wedge H))\\
&\models \neg(F \wedge G) \leftrightarrow \neg F \vee \neg G\\
&\models \neg(F \vee G) \leftrightarrow \neg F \wedge \neg G.
\end{aligned}$$

We close this section with the definitions of logical consequence and equivalence.

Definition 9 (Entailment)**.** A formula G is a **consequence** of (or is **entailed** by) a set of formulas $\mathcal{S}$ if every assignment that satisfies each formula in $\mathcal{S}$ also satisfies G. In this case we write $\mathcal{S} \models G$.

Entailment allows us to draw logical conclusions from existing facts. Using truth tables, you can convince yourself that

$$\{\neg c, a \vee b, b \to c\} \models a.$$

This is the example we saw in the previous section, and now we can even formally conclude that Alice is an architect.

> **Warning!** In logic the symbol $\models$ is overloaded. Above we define $\mathcal{S} \models F$ for a set of formulas $\mathcal{S}$ and formula F. Previously we have written $\mathcal{A} \models F$ to say that an assignment $\mathcal{A}$ is a model of F.

The final concept we want to introduce is logical equivalence.

Definition 10 (Equivalence)**.** 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.

5 Encoding constraint problems into satisfiability problems

In order to illustrate the concepts that we have seen so far, we are now going to discuss how some well-known constraint problems can be encoded in propositional logic, and how we can reason about them using our machinery.

The first problem that we encode into propositional logic is the well-known Sudoku puzzle:

	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	

For each $i, j, k \in \{1, \ldots, 9\}$ we have a proposition $x_{i,j,k}$ expressing that *grid position i, j contains number k*. Now build a formula F as the conjunction of the following *constraints*:

- Each number appears in each row and in each column:

$$F_1 := \bigwedge_{i=1}^{9} \bigwedge_{k=1}^{9} \bigvee_{j=1}^{9} x_{i,j,k} \qquad F_2 := \bigwedge_{j=1}^{9} \bigwedge_{k=1}^{9} \bigvee_{i=1}^{9} x_{i,j,k}$$

- Each number appears in each 3×3 block:

$$F_3 := \bigwedge_{k=1}^{9} \bigwedge_{u=0}^{2} \bigwedge_{v=0}^{2} \bigvee_{i=1}^{3} \bigvee_{j=1}^{3} x_{3u+i,3v+j,k}$$

- No square contains two numbers:

$$F_4 := \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{1 \le k < k' \le 9} \neg(x_{i,j,k} \wedge x_{i,j,k'}) \,.$$

- Certain numbers appear in certain positions: we assert

$$F_5 := x_{2,1,2} \wedge x_{1,2,8} \wedge x_{2,3,3} \wedge \ldots \wedge x_{8,9,6} \,.$$

The formula F thus obtained is satisfiable if and only if the given Sudoku instance has a solution. Some facts about Sudokus have not explicitly been included in F. For instance, the property that no number appears twice in the same row, i.e.,

$$F_6 := \bigwedge_{i=1}^{9} \bigwedge_{k=1}^{9} \bigwedge_{1 \le j < j' < 9} \neg(x_{i,j,k} \wedge x_{i,j',k})$$

has not explicitly been stated. However, it can be verified that $\models F \to F_6$. Though redundant, explicitly adding F_6 to F may help a satisfiability solver when searching for a satisfying assignment. Note that the number of variables $x_{i,j,k}$ is $9^3 = 729$. Thus a truth table for the corresponding formula would have $2^{729} > 10^{200}$ lines! Nevertheless a modern SAT-solver can find a satisfying assignment in milliseconds.

The second example we consider is that of finding a Hamiltonian path in an undirected graph. Given a undirected graph $G = (V, E)$ with vertices V and edges $E \subseteq V \times V$ such that E is symmetric, the Hamiltonian path problem asks whether there exists path in G that visits every vertex exactly once. Figure 2 illustrates a Hamiltonian path in a graph. We now show how the Hamiltonian path problem can be encoded into propositional logic. Given an undirected graph $G = (V, E)$ such that $E \subseteq V \times V$ is symmetric, for each vertex $i, j \in \{1, \ldots, n\}$ we have a proposition $x_{i,j}$ expressing that *vertex i is the jth vertex in the Hamiltonian path*. Now build the formula F as the conjunction of the following *constraints*:

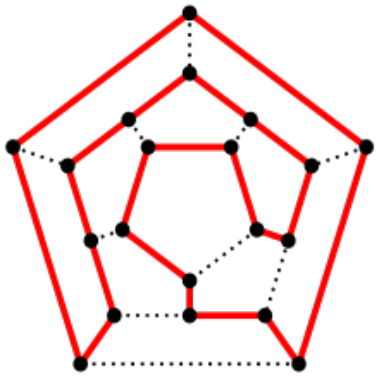

Figure 2: Example of a Hamiltonian path in an undirected graph.

- Each vertex is visited precisely once, and no two vertices are visited at the same time:

$$F_1 := \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} x_{i,j} \qquad F_2 := \bigwedge_{i=1}^{n} \bigwedge_{1 \le j \ne k \le n} \neg(x_{i,j} \wedge x_{i,k}) \wedge \neg(x_{j,i} \wedge x_{k,i})$$

- The path goes along edges:

$$F_3 := \bigwedge_{i=1}^{n} \bigwedge_{k=1}^{n} \bigwedge_{j=1}^{n-1} x_{i,j} \wedge x_{k,j+1} \rightarrow e_{i,k}$$

$$F_4 := \bigwedge_{(i,j) \in E} e_{i,j} \wedge \bigwedge_{(i,j) \notin E} \neg e_{i,j}$$

Here, the $e_{i,j}$ are additional propositional variables encoding the structure of G. The graph has a Hamiltonian path precisely when $F := F_1 \wedge F_2$ is satisfiable.

6 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.

6.1 The 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 hundreds of 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ 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 a tiny bit better than exhaustive search in the worst case!

6.2 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 11. 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 $\{x_{v,c} : v \in V, c \in C\}$. Intuitively $x_{v,c}$ 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:

$$F_1 := \bigwedge_{v \in V} \bigvee_{c \in C} x_{v,c} \,.$$

- Each vertex has at most one colour:

$$F_2 := \bigwedge_{v \in V} \bigwedge_{\substack{c,c' \in C \\ c \neq c'}} \neg(x_{v,c} \wedge x_{v,c'}) \,.$$

- Adjacent vertices have different colours:

$$F_3 := \bigwedge_{(u,v) \in E} \bigwedge_{c \in C} \neg(x_{u,c} \wedge x_{v,c}) \,.$$

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.