Lecture 3: More on Logic

In this lecture we look at some more concepts of propositional logic.

Satisfiability

Some formulas are 'always true', such as T or $x \vee \neg x$. Others are sometimes true, others never. We give some names for these situations.

Definition 1 (satisfiability)**.** We say a logical formula is

- A **tautology**, if it evaluates to true in every environment that defines all its variables.
- A **contradiction**, if it evaluates to false in every environment that defines all its variables.
- A **contingency** if it is neither a tautology nor a contradiction, that is there is at least one environment in which it evaluates to true, and at least one environment in which it evaluates to false.

So every formula is exactly one of the three cases above, but there are some more related definitions:

- A formula is **satisfiable**, if there is at least one environment in which the formula evaluates to true. Terms that are not satisfiable are also called **unsatisfiable**.

For example, the formulas T and $x \vee \neg x$ are both tautologies; the formulas F and $x \wedge \neg x$ are both contradictions, and the formula x is a contingency.

Satisfiable means the same as 'contingency or tautology', and unsatisfiable is a synonym for contradiction. Of course, we can read the definition of a tautology as 'every row in the formula's truth table column is T' if we prefer.

Further, another way of saying the same thing is that tautologies are formulas U with $U \equiv T$ and contradictions are formulas U with $U \equiv F$. All tautologies are semantically equivalent to each other, and so are all contradictions, but not all contingencies are semantically equivalent.

Entailment

Remember that if A, B are logical formulas then $A \leftrightarrow B$ is another formula that is true if and only if A, B have the same truth value in every possible assignment, whereas $A \equiv B$ is a statement (but not a formula itself) that A, B are semantically equivalent, which holds if and only if the formula $A \leftrightarrow B$ is a tautology. Entailment is a similar statement but for the one-sided arrow $\rightarrow$.

Definition 2 (entailment)**.** For logical formulas A, B we say that A *entails* B and write $A \vDash B$ for the following statement: for every row in a truth table over the variables of both A and B, if A is true in this row then B is true in this row too.

More formally, $A \vDash B$ says that every environment that makes A true also makes B true.

$A \vDash B$, like $A \equiv B$, is not a logical formula but a statement (in the 'outer' logic).

For example, we could say that $x \vDash x \vee y$ since the truth table is

x	y	x	$x \vee y$
F	F	F	F
F	T	F	T
T	F	T	T
T	T	T	T

Here, every row where x is true also makes $x \vee y$ true; it does not matter that there is a row where $x \vee y$ is true but x is not. Put another way, $x \vDash x \vee y$ is the statement that the formula $x \to (x \vee y)$ is a tautology.

For another example, $x \wedge y \vDash x$ and $x \wedge y \vDash y$, but $x \vee y \nvDash x$ since, from the fact that $x \vee y$ is true in some environment, we cannot conclude that x is true as well: the formula $x \vee y$ might be true due to y, even though x is false.

More on Implication

We define some new terms related to logical implication $A \to B$.

Definition 3 (implication terminology)**.** For a formula $A \to B$, we call the left-hand side A the *antecedent* and the right-hand side B the *consequent* of the implication. Further,

- The **converse** of $A \to B$ is $B \to A$.
- The **inverse** of $A \to B$ is $\neg A \to \neg B$.
- The **contrapositive** of $A \to B$ is $\neg B \to \neg A$.

Looking at a truth table,

			converse	inverse	contrapositive
a	b	$a \to b$	$b \to a$	$\neg a \to \neg b$	$\neg b \to \neg a$
F	F	T	T	T	T
F	T	T	F	F	T
T	F	F	T	T	F
T	T	T	T	T	T

we note that the converse and inverse of an implication are equivalent to each other, but not to the original statement, that is $a \to b \not\equiv b \to a$ and $a \to b \not\equiv \neg a \to \neg b$ but $b \to a \equiv \neg a \to \neg b$. In fact, neither $a \to b \vDash b \to a$ nor $b \to a \vDash a \to b$, so knowing that the converse of a proposition is true does not help us to decide if the preposition itself is true. The same goes for the inverse.

The other way round is more useful: if we know that the converse of a proposition is false, then

a	b	$a \to b$	$\neg(b \to a)$
F	F	T	F
F	T	T	T
T	F	F	F
T	T	T	F

The converse of $a \to b$ can only be false — and therefore the negated converse is true — if a is false and b is true. But in this case, $a \to b$ is also true. So we have $\neg(b \to a) \vDash a \to b$, but $\neg(b \to a) \not\equiv a \to b$.

This will become useful later on when we do proofs: if we want to prove that a proposition $a \to b$ is true, then one strategy is to prove that $b \to a$ is false, but this is proving a strictly stronger statement than the one we are after, and so it will not always work.

On the other hand, the *contrapositive* of an implication formula is equivalent to the formula itself:

$$a \to b \quad \equiv \quad \neg b \to \neg a$$

This means that if we are trying to prove $a \to b$, then one strategy is to try and prove $\neg b \to \neg a$ instead; this may be easier or harder to prove, but it is always equivalent to the original statement $a \to b$.

Completeness and Soundness

The operation $\vDash$ is the semantic version of $\vdash$. Remember that $A \vdash B$ means you can transfrom A into B using a list of syntax rules.

We can say that a list of syntax rules is

- *complete* for a semantics if for every two formulas A, B, if $A \vDash B$ then also $A \vdash B$: if one formula A entails another formula B then we can also transform A into B with the syntax rules alone. This also implies that if $A \equiv B$, then also $A \mathrel{\widehat{=}} B$.
- *sound* for a semantics if for every two formulas A, B, if $A \vdash B$ then also $A \vDash B$: if we can transform A to B with the rules, then A also entails B. Put another way, the rules do not let you conclude anything that is not true semantically. This also implies that if $A \mathrel{\widehat{=}} B$, then $A \equiv B$.

For integer arithmetic, there is unfortunately no complete and sound list of rules (or more precisely, we will never be able to find such a list and prove that it is complete and sound), but for propositional logic, the list of the laws of logic is complete and sound. We could prove this if we wanted to, but we don't want to in this unit, as things have got quite abstract enough already.

Functional Completeness

We can express all logical operators that we have learnt so far with just the operations $\wedge, \vee, \neg$ and constants (and brackets if necessary, but these are a feature of infix notation for formula strings, not for formulas themselves):

- $x \oplus y \equiv (x \wedge \neg y) \vee (\neg x \wedge y)$ or $x \oplus y \equiv (x \vee y) \wedge \neg(x \wedge y)$.
- $x \to y \equiv \neg x \vee y$.
- $x \leftrightarrow y \equiv (x \wedge y) \vee (\neg x \wedge \neg y)$.

This is true in a much more general sense:

Definition 4 (functional completeness)**.** A set of logical operations is functionally complete if for every logical formula A, there is an equivalent formula B that uses only operations from this set (as well as values and variables).

Theorem: the set $\{\wedge, \vee, \neg\}$ is functionally complete.

To prove this theorem, we can even give an algorithm that constructs the equivalent formula B.

Normal Forms

Imagine you have a truth table for some unknown formula. How can you create a formula from the table?

The answer is that every row of the truth table is a different assignment to the variables. If we create a formula that is the logical and ($\wedge$) of all variables involved, but with a $\neg$ in front of the ones that are false in the row, then this formula will be true in this row and in no other rows. For example, if the variables are x, y, z then the formula $x \wedge y \wedge \neg z$ will be true for the assignment $x = T, y = T, z = F$ and false for the other 7 assignments.

If we now take a formula that is the logical or ($\vee$) of exactly the rows where our unknown formula is true, then this formula will have the same truth table. An example:

x	y	z	A	clause
F	F	F	F	$(\neg x \wedge \neg y \wedge \neg z)$
F	F	T	T	$(\neg x \wedge \neg y \wedge z)$
F	T	F	F	$(\neg x \wedge y \wedge \neg z)$
F	T	T	F	$(\neg x \wedge y \wedge z)$
T	F	F	F	$(x \wedge \neg y \wedge \neg z)$
T	F	T	T	$(x \wedge \neg y \wedge z)$
T	T	F	F	$(x \wedge y \wedge \neg z)$
T	T	T	F	$(x \wedge y \wedge z)$

We are trying to give a formula for the expression in the A column. First, we make a clause for each row in the rightmost column, that is true in only this row and nowhere else. Then, taking the logical or ($\vee$) of the clauses for the rows where the A formula is true, we get

$$A \equiv (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge z)$$

And note that any formula we produce with this method will, apart from variables, only contain the symbols $\wedge, \vee, \neg$ and brackets. Therefore, for any formula, we can produce an equivalent formula with only these operations, by writing out its truth table and then applying the same method. Terms in this form are special enough that they get their own name.

Definition 5 (normal forms)**.** A logical formula A is in *disjunctive [a] normal form* (DNF) if it is in the following form:

1. A is made of clauses $A_1, \ldots, A_n$ combined with $\vee$.
2. All the clauses A_i are made with literals combined with $\wedge$.
3. All the literals are either variables, or negations of variables.

A formula in the same form but with the roles of $\wedge$ and $\vee$ swapped is called in conjunctive normal form (CNF).

For every formula, there is both an equivalent formula in CNF, and an equivalent formula in DNF.

[a] Conjunction is another name for 'and', and disjunction is another name for 'or'.

More on DNF

The truth table method above does not necessarily produce the shortest possible formula: in our example, $(\neg y \wedge z)$ is equivalent to the same formula A, and is still in DNF: this formula contains only one clause, so no $\vee$ signs are needed.

In fact, there is a general rule that if you have two clauses in a DNF formula that differ only in the 'sign' of one variable — that is, one variable is negated in one clause, not negated in the other, and the rest of the clauses are the same — then you can remove one of the two clauses, and remove this variable from the other clause to get a simpler formula that is still in DNF. Mathematically, what is going on here is that

$$\begin{array}{ll} & (x \wedge B) \vee (\neg x \wedge B) \\ \widehat{=} & (x \vee \neg x) \wedge B \qquad \text{(distributive law, 'backwards')} \\ \widehat{=} & T \wedge B \qquad \text{(law of the excluded middle)} \\ \widehat{=} & B \qquad \text{(identity law)} \end{array}$$

Or we could just write out a truth table with two variables x, b to show that $(x \wedge b) \vee (\neg x \wedge b) \equiv b$.

This lets you combine blocks of 2 rows in a truth table if you want to take the clauses for both rows, and they only differ in one position. Of course, if you now have two blocks that only differ in one variable, and you want both of them, you can repeat this operation, and so on — if you are lucky, you get one clause that covers 2^n rows for some n, which is the number of variables you are eliminating.

There is a general algorithm based on this principle that is guaranteed to find a shortest possible DNF (or CNF) formula for any truth table, and can be easily done on paper for up to four variables. This algorithm is called the *Karnaugh map* and you will probably encounter it in your Computer Architecture unit, as it is used in hardware design.

The SAT Problem

Given a logical formula in CNF or DNF, is there an algorithm to check if the formula is satisfiable?

In DNF, the problem is easy: since the clauses are combined with logical OR, we can check one clause at a time, the moment we have found an assignment to the variables that satisfies one clause, the whole formula is satisfied. To check an individual clause, we can just assign values that make literals true, that is if we see an x literal we take $x = T$ and if we see a $\neg x$ literal we take $x = F$. The only way this can go wrong is if a clause contains both, e.g. $(x \wedge y \wedge \neg x)$ which is allowed under the definition of DNF. In this case, that particular clause is unsatisfiable, so we move on to the next one. This way, we only have to read the DNF formula once from left to right.

For CNF, the problem is the clauses are combined with logical AND, so we need one single assignment to the variables that satisfies all the clauses at once. This is a hard problem — what is known in Computer Science as an *NP-hard* problem. Even just checking, for a CNF formula, whether it is satisfiable at all (without coming up with an assignment) is *NP-complete*. This means that

- There is currently no known algorithm that is guaranteed to always work and substantially faster than trying all possibilities, which for N variables takes 2^N time.
- Any improvement to this state of affairs (in a particular technical sense) would prove $P = NP$, the biggest open problem in Theoretical Computer Science.

This also implies that there is no efficient algorithm (again in a technical sense, that basically means 'works for any possible formula and is always faster than trying all 2^N possibilities') to turn a formula in CNF into an equivalent formula in DNF.

NAND and NOR

We know that the set $\{\wedge, \vee, \neg\}$ is functionally complete because for any formula, we can build an equivalent formula in DNF that uses only these operations. Are there other functionally complete sets, perhaps smaller ones?

In fact, we can express $x \vee y \equiv \neg\neg(x \vee y) \equiv \neg(\neg x \wedge \neg y)$ by introducing a double negation and then applying DeMorgan, so we could rewrite a DNF formula to use only $\{\wedge, \neg\}$, or analogously only $\{\vee, \neg\}$. So these are functionally complete sets with only two elements.

But there are two more logical operations that are functionally complete on their own, called NAND ($\uparrow$) and NOR ($\downarrow$). They are just negated $\wedge$ and $\vee$ operations:

a	b	$a \uparrow b$	$a \downarrow b$
F	F	T	T
F	T	T	F
T	F	T	F
T	T	F	F

For example, we can express $\neg x \equiv x \uparrow x$ and

$$
\begin{array}{lll}
 & x \vee y & \\
\widehat{=} & \neg\neg(x \vee y) & \text{double negation introduction} \\
\widehat{=} & \neg(\neg x \wedge \neg y) & \text{DeMorgan} \\
\widehat{=} & \neg x \uparrow \neg y & \text{using } \neg(a \wedge b) \equiv a \uparrow b \\
\widehat{=} & (x \uparrow x) \uparrow (y \uparrow y) & \text{using } \neg a \equiv a \uparrow a
\end{array}
$$

so we can express $\neg$ and $\vee$ just with $\uparrow$. From this, we can express everything with just $\uparrow$ by turning the $\wedge$ into combinations of $\vee, \neg$, or we can calculate a bit to find that $x \wedge y \equiv (x \uparrow y) \uparrow (x \uparrow y)$. So, the set $\{\uparrow\}$ on its own is functionally complete, and the same holds for $\{\downarrow\}$ by a similar argument.

This has applications in hardware design where you learn in Computer Architecture how to build and, or, not etc. gates out of NAND gates using transistors.