

CS1Bh Lecture Note 18

Propositional Logic: Equivalence Reasoning and Semantic Trees

CS1Ah Note 18 introduced propositional logic and the use of truth-tables to specify the meaning of propositional connectives and propositional formulas. It also discussed reasoning about propositions using equivalence rules. In this note we consolidate that material, continuing the study of equivalence reasoning and looking at the *semantic tree* or *tableaux* technique for analysing the truth of formulas.

Semantic trees are mainly studied in logic and used in AI applications. They are considered here in a computer science course because they reinforce one's understanding of the propositional logic connectives.

In what follows, we use the common logical notation introduced earlier. In high-to-low order of precedence, the logical operators are \neg (not), \wedge (and), \vee (or), \Rightarrow (implies) and \Leftrightarrow (iff, if and only if). The boolean constants are 't' (true) and 'f' (false). We refer frequently to the following terminology:

- A propositional formula is a *tautology* if all truth assignments to the proposition's variables make the formula true.
- A propositional formula is *satisfiable* if some truth assignments to the proposition's variables makes the formula true.
- A propositional formula is a *contradiction* if all truth assignments to the proposition's variables make the formula false.

A formula is satisfiable if and only if it is not a contradiction.

Equivalence Reasoning

Since \Leftrightarrow is an equivalence relation, one can use it in 'chain of equivalences' arguments, just as one uses equality in algebra. For example, to show that

$$(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$$

one can reason as follows:

$$\begin{aligned}
 (P \Rightarrow Q) &\Leftrightarrow (\neg P \vee Q) && \text{characterisation of } \Rightarrow \\
 &\Leftrightarrow (Q \vee \neg P) && \text{commutativity of } \vee \\
 &\Leftrightarrow (\neg\neg Q \vee \neg P) && \text{double negation rule} \\
 &\Leftrightarrow (\neg Q \Rightarrow \neg P) && \text{characterisation of } \Rightarrow .
 \end{aligned}$$

In CS1Ah Note 18 we saw the equivalence rules:

1. $P \Leftrightarrow P$ \Leftrightarrow is reflexive
2. $P \wedge Q \Leftrightarrow Q \wedge P$ \wedge is commutative
3. $P \vee Q \Leftrightarrow Q \vee P$ \vee is commutative
4. $P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$ \wedge is associative
5. $P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$ \vee is associative
6. $P \wedge P \Leftrightarrow P$ \wedge is idempotent
7. $P \vee P \Leftrightarrow P$ \vee is idempotent
8. $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$ \wedge distributes over \vee
9. $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$ \vee distributes over \wedge
10. $\neg\neg P \Leftrightarrow P$ double negation elimination
11. $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$ DeMorgan's Law for \neg over \wedge
12. $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$ DeMorgan's Law for \neg over \vee
13. $P \wedge t \Leftrightarrow P$ t is the identity for \wedge
14. $P \vee f \Leftrightarrow P$ f is the identity for \vee
15. $P \wedge f \Leftrightarrow f$ f is the annihilator for \wedge
16. $t \vee P \Leftrightarrow t$ t is the annihilator for \vee
17. $P \wedge \neg P \Leftrightarrow f$
18. $P \vee \neg P \Leftrightarrow t$
19. $P \vee (P \wedge Q) \Leftrightarrow P$
20. $P \wedge (P \vee Q) \Leftrightarrow P$
21. $P \wedge (\neg P \vee Q) \Leftrightarrow P \wedge Q$
22. $P \vee (\neg P \wedge Q) \Leftrightarrow P \vee Q$
23. $(P \Rightarrow Q) \Leftrightarrow \neg P \vee Q$ \Rightarrow elimination
24. $\neg(P \Rightarrow Q) \Leftrightarrow P \wedge \neg Q$ \neg of \Rightarrow elimination
25. $(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$
26. $(P \Leftrightarrow Q) \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge \neg Q)$ \Leftrightarrow elimination
27. $\neg(P \Leftrightarrow Q) \Leftrightarrow (P \wedge \neg Q) \vee (\neg P \wedge Q)$ \neg of \Leftrightarrow elimination
28. $\neg(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow Q)$
29. $(P \Leftrightarrow Q) \Leftrightarrow (Q \Leftrightarrow P)$ \Leftrightarrow is symmetric
30. $(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$
31. $(P \Leftrightarrow Q) \Leftrightarrow (P \Rightarrow Q) \wedge (Q \Rightarrow P)$ \Leftrightarrow is bi-implication .

To reason effectively with propositional equivalences, most of these rules need to be committed to memory, or readily figured out when needed. It helps to note the many patterns in the rules. For example, those involving \wedge and \vee come in pairs where the \wedge s are swapped with \vee s and the 't's are swapped with 'f's.

Associativity and commutativity

Explicitly giving a derivation where associativity and commutativity rules are spelled out, one by one, is exceptionally tedious. For example, to show $((P \wedge Q) \wedge R) \wedge S \Leftrightarrow$

$((S \wedge R) \wedge Q) \wedge P$ involves the derivation:

$$\begin{aligned}
 ((P \wedge Q) \wedge R) \wedge S &\Leftrightarrow S \wedge ((P \wedge Q) \wedge R) && \text{commutativity} \\
 &\Leftrightarrow S \wedge (R \wedge (P \wedge Q)) && \text{commutativity} \\
 &\Leftrightarrow S \wedge (R \wedge (Q \wedge P)) && \text{commutativity} \\
 &\Leftrightarrow (S \wedge R) \wedge (Q \wedge P) && \text{associativity} \\
 &\Leftrightarrow ((S \wedge R) \wedge Q) \wedge P && \text{associativity} .
 \end{aligned}$$

Usually in logic, since we know in general that associative and commutative laws for a binary operator permit arbitrary rearrangements of arguments to nested occurrences of the operator, we do such rearrangements all at once and don't spell them out. For similar reasons, it's common not to use parentheses to show how uses of an associative operator are associated

Of course, when programming, we are concerned with how nested occurrences of an operator are arranged. For example, consider using the conditional operators `&&` and `||`, or if propositional formulas contain method calls or operators with side effects.

Non-obvious steps

Sometimes equivalence reasoning involves non-obvious steps that seem initially to make things more complicated rather than easier. For example, to show $P \vee (P \wedge Q) \Leftrightarrow P$ (rule 19) using rules 1–18, we need to argue:

$$\begin{aligned}
 P \vee (P \wedge Q) &\Leftrightarrow (P \wedge t) \vee (P \wedge Q) \\
 &\Leftrightarrow (P \wedge (Q \vee \neg Q)) \vee (P \wedge Q) \\
 &\Leftrightarrow (P \wedge Q) \vee (P \wedge \neg Q) \vee (P \wedge Q) \\
 &\Leftrightarrow (P \wedge Q) \vee (P \wedge \neg Q) \\
 &\Leftrightarrow P \wedge (Q \vee \neg Q) \\
 &\Leftrightarrow P \wedge t \\
 &\Leftrightarrow P .
 \end{aligned}$$

If we make use of rules 19-22 in equivalence reasoning, such non-obvious steps are usually not necessary.

Using equivalence reasoning for simplification

Equivalence reasoning is very useful for simplifying propositions in both logic and programming.

Common criteria for determining if a proposition is 'simpler' are that it:

- has fewer variable occurrences,
- only uses certain logical connectives, (e.g. \wedge , \vee , and \neg),
- doesn't involve any 't's and 'f's, unless it is identical to 't' or 'f',
- has \neg s pushed in towards formula syntax tree leaves using the De Morgans laws and double negation elimination,
- has common subformulas gathered using the distributive laws.

In different circumstances, one might choose different subsets of these criteria to determine what characterises a simpler formula: there is no single universal definition of ‘simpler’.

One reason for wanting to simplify propositions is to make them easier to understand, and ensure there are no errors. When programming, we might also want to simplify propositions in order to make their evaluation more efficient. For example we could reduce the number of occurrences of some side-effect-free boolean-valued method call that initially occurs as several subformulas of some boolean formula.

Conjunctive and disjunctive normal forms

These are common simplified formats for propositions. First some definitions:

A *conjunction* of formulas C_1, C_2, \dots, C_n is the formula $C_1 \wedge C_2 \wedge \dots \wedge C_n$. The C_i are called *conjuncts*. If $n = 1$, the conjunction is simply C_1 . It is convenient to allow too the case of $n = 0$, in which case the conjunction is the truth value ‘ t ’.

Similarly, a *disjunction* of formulas D_1, D_2, \dots, D_n is the formula $D_1 \vee D_2 \vee \dots \vee D_n$ and the D_i are called *disjuncts*. If $n = 1$, the disjunction is simply D_1 , and, if $n = 0$, the disjunction is the truth value ‘ f ’.

A *literal* is a propositional variable or a negation of a propositional variable. P and $\neg Q$ are both literals.

A proposition is in *Conjunctive Normal Form*, frequently abbreviated to CNF, if it is a conjunction of disjunctions of literals. For example, the formula

$$(A \vee B) \wedge (C \vee D) \wedge (D \vee \neg A \vee \neg B) \wedge E$$

is in CNF.

A proposition is in *Disjunctive Normal Form*, or DNF, if it is a disjunction of conjunctions of literals.

While the structure of these normal forms might be simple, they are not necessarily compact. For example, the proposition

$$(A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge \dots \wedge (A_{20} \vee B_{20})$$

has 20 conjuncts, each with 2 variables. If this proposition were put in DNF, it would have over 10^6 disjuncts, each with 20 variables!

Any formula can be put in CNF or DNF by repeated use of appropriate equivalence rules.

Tautology and contradiction checking

If equivalence reasoning reduces a proposition to ‘ t ’ or ‘ f ’, we know the proposition is respectively a tautology or a contradiction. Equivalence reasoning therefore presents one alternative to using truth tables for checking these properties. For example,

$$\begin{aligned} (P \Rightarrow Q) \vee (Q \Rightarrow P) &\Leftrightarrow \neg P \vee Q \vee \neg Q \vee P \\ &\Leftrightarrow (P \vee \neg P) \vee (Q \vee \neg Q) \\ &\Leftrightarrow t \vee t \\ &\Leftrightarrow t, \end{aligned}$$

so $(P \Rightarrow Q) \vee (Q \Rightarrow P)$ is a tautology.

Semantic Trees

The simplest systematic technique for checking if a propositional formula is a tautology is to use a truth table. However, these are often very verbose and tedious to produce. The technique of *Semantic Trees* (sometimes known as *Tableaux*) is often a more efficient alternative.

With truth tables, one calculates the truth of a formula ‘bottom up’, starting with the leaves of the formula syntax tree and working up to the formula’s root. By contrast, the semantic tree technique is ‘top down’. It considers hypotheses about the truth or falsity of nodes of a formula tree, and explores what values (true or false) subformulas and variables would then have to take on.

The technique checks if a formula ϕ is a tautology by exploring the consequences of assuming that ϕ is false. If it is found that there is no way ϕ could be false, we then know if that ϕ must be a tautology. In addition, if a semantic tree does find how formula can be falsified, the assignment of values to variables that makes it false can simply be read off the tree.

An Example

Consider checking whether $(A \wedge \neg B) \vee (\neg A \wedge B)$ is a tautology. We start by asserting this is false, drawing the semantic tree with one node:

$$f: (A \wedge \neg B) \vee (\neg A \wedge B)$$

In general, nodes of semantic trees have form $f: P$ asserting P is false, and $t: P$ asserting P is true. These are called *signed formulas*. If $(A \wedge \neg B) \vee (\neg A \wedge B)$ is false, then it must be the case that both immediate subformulas $A \wedge \neg B$ and $\neg A \wedge B$ are also false. We indicate this in the tree as follows:

$$\begin{array}{c} *f: (A \wedge \neg B) \vee (\neg A \wedge B) \\ | \\ f: A \wedge \neg B \\ | \\ f: \neg A \wedge B \end{array}$$

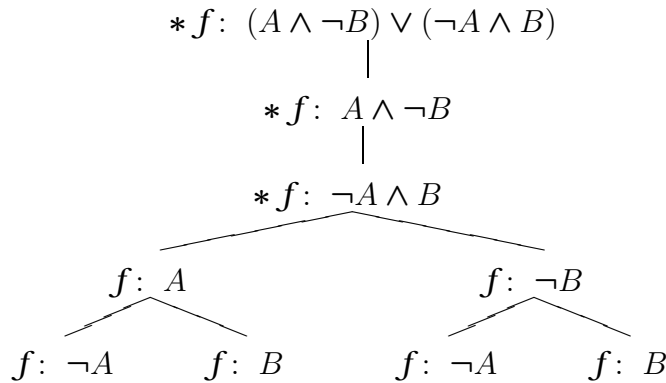
We add the $*$ to indicate that we have now already considered or *discharged* the formula $(A \wedge \neg B) \vee (\neg A \wedge B)$.

If $A \wedge \neg B$ is false, either it must be the case that A is false or it must be the case that $\neg B$ is false. (It may be the case that both are false, but this case is covered by the previous two and we don’t need to separate it out as a distinct case.) We indicate that the cases are different by adding a fork in the tree, and placing the assertions in each case in a distinct branch of the fork:

$$\begin{array}{c} *f: (A \wedge \neg B) \vee (\neg A \wedge B) \\ | \\ *f: A \wedge \neg B \\ | \\ f: \neg A \wedge B \\ \swarrow \quad \searrow \\ f: A \quad f: \neg B \end{array}$$

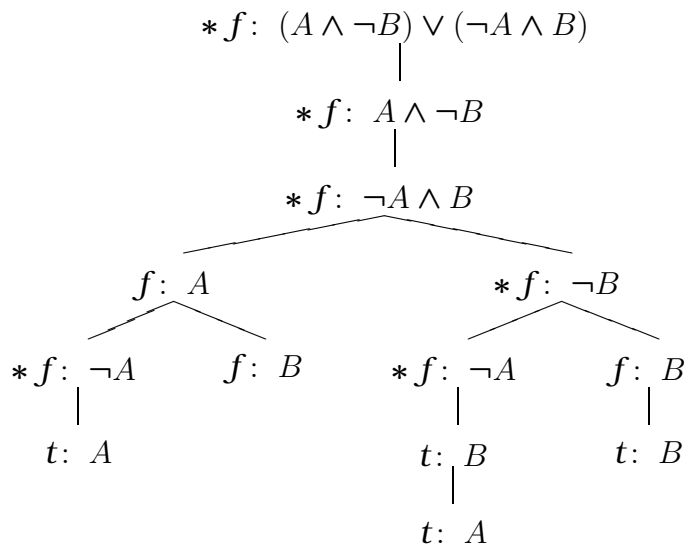
In general, each path in the semantic tree represents a distinct possibility, and we always have the case that the paths of a tree cover all possibilities. (A *path* in a tree is a sequence of nodes from the tree root to some leaf node).

We continue by considering the node $f: \neg A \wedge B$. This produces a second splitting of cases:

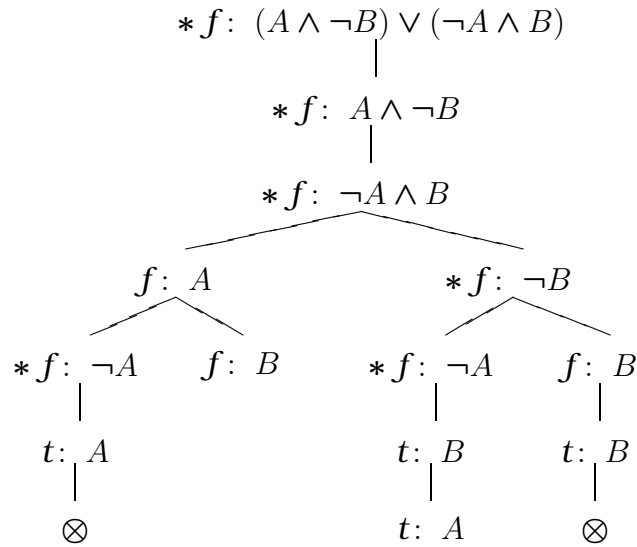


Observe how one always adds new nodes to tree below the leaf nodes of the old tree, rather than adjacent to the formula being considered.

If we continue to grow the tree by looking at the consequences of the $\neg A$ and $\neg B$ being asserted false, we get:



Look at the leftmost path. Note that, for this possibility to work, A must be both true and false. This is clearly impossible and we rule out this possibility by saying that this path is *closed*. Likewise the rightmost path is closed because it asserts that B is both true and false. We indicate closed paths by adding the symbol \otimes to the bottom of paths:



There is no further way of extending this tree: all subformulas that are not variables have been discharged. We still have two open paths remaining and these indicate ways in which the original formula can be false: namely, we assign A and B both false, or both true. The original formula is therefore not a tautology. If it so happens that we end up with a semantic tree in which all paths are closed, we know there is no way the original formula can be made false, so the formula is a tautology.

Rules for creating a semantic tree

The basic rules for growing a semantic tree are presented in Table 1.

	\wedge	\vee	\neg	\Rightarrow	\Leftrightarrow
t	$ \begin{array}{c} t: A \wedge B \\ \\ t: A \\ \\ t: B \end{array} $	$ \begin{array}{c} t: A \vee B \\ \swarrow \quad \searrow \\ t: A \quad t: B \end{array} $	$ \begin{array}{c} t: \neg A \\ \\ f: A \end{array} $	$ \begin{array}{c} t: A \Rightarrow B \\ \swarrow \quad \searrow \\ f: A \quad t: B \end{array} $	$ \begin{array}{c} t: A \Leftrightarrow B \\ \swarrow \quad \searrow \\ t: A \quad f: A \\ \quad \quad \\ t: B \quad f: B \end{array} $
f	$ \begin{array}{c} f: A \wedge B \\ \swarrow \quad \searrow \\ f: A \quad f: B \end{array} $	$ \begin{array}{c} f: A \vee B \\ \\ f: A \\ \\ f: B \end{array} $	$ \begin{array}{c} f: \neg A \\ \\ t: A \end{array} $	$ \begin{array}{c} f: A \Rightarrow B \\ \\ t: A \\ \\ f: B \end{array} $	$ \begin{array}{c} f: A \Leftrightarrow B \\ \swarrow \quad \searrow \\ t: A \quad f: A \\ \quad \quad \\ f: B \quad t: B \end{array} $

Table 1: Rules for Growing a Semantic Tree

To check whether a propositional formula P is a tautology:

1. Begin a tree with node of form $f: P$.
2. Repeat until no further progress can be made:
 - (a) Choose an undischarged node n which isn't a variable or a constant.
 - (b) Find the appropriate rule for n , and add the new nodes suggested by this rule to the leaves of the subtree below this node.
 - (c) Mark node n as discharged
 - (d) Check every path through n for feasibility. If both $t: Q$ and $f: Q$ occur on the path for some formula Q , mark the path as closed and never extend it further in subsequent steps. Also mark a path as closed if the signed formula $t: f$ or $f: t$ occurs on it.
3. If all paths are closed, P is a tautology. Otherwise, a counter-example can be read off each of the open paths by looking at the assertions of values for the propositional variables on the path.

If one instead begins a tree with a node of form $t: P$, the semantic tree algorithm can be used to check if P is a contradiction or is satisfiable. If all paths end up closed, then P is a contradiction. If one or more paths remain open, P is satisfiable and a satisfying truth assignment to the variables can be read off from the tree.

The algorithm as presented above is said to be *non-deterministic* because at step 2(a) there might be more than one choice, and different trees will be produced if different choices are made.

As a semantic tree acquires more and more forks, it becomes more work to grow it further. Therefore, a good heuristic for making the choice is, if possible, to choose a rule that doesn't cause any forking (for example, the $t\text{-}\wedge$ rule).

If we amend step 2(a) to read

2. (a) Choose an undischarged node n which isn't a variable. If there is more than one, choose the one closest to the top of the tree. If there is more than one of those, choose the leftmost one.

then the algorithm is *deterministic* and guaranteed always to produce the same result.

The above rules and algorithm for semantic trees should become sufficiently intuitive to you that you can generate semantic trees without explicit reference to the rules and algorithm. This week's exercise sheet contains several examples for you to practice on.

Paul Jackson,
14th April 2003.