Topics in propositional logic

Readings: None from the textbook.

In this module, we will look at alternate methods of demonstrating the truth or falsity of statements in the propositional calculus, and consider some implications of soundness and completeness. We will be quickly covering a number of related topics whose combined scope is considerable.

Variations on natural deduction

One of the advantages of natural deduction is that tautologies can be built up starting with nothing at all. Other related systems reduce the number of rules by introducing **axioms**, basic formulas assumed to be true (and therefore usable on the left-hand side of any sequent).

It is possible to define a proof system with a single derivation rule (MP) but with thirteen axioms. An example of an axiom would be $\phi \to \phi \vee \psi$. This is really an **axiom schema**, since substitution of any formulae ϕ and ψ produces an axiom.

Such a system may have some technical advantages (it shifts work from the inductive step to base cases in some proofs) but it does not correspond as well to mathematical reasoning.

The sequent calculus

Instead of using a proof as a method of showing validity of a sequent, we could describe transformations on sequents that preserve validity. This is the basis of Gentzen's sequent calculus, which he introduced at the same time as natural deduction. Here is a sample rule in the sequent calculus.

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \wedge L_1$$

The intuitive meaning of the label is that a formula involving $\wedge$ is introduced on the left side of the sequent.

There are also rules for introducing connectives on the right side of the sequent. Here is one:

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta} \vee R_1$$

Notice that these sequents, unlike ours, permit a set to appear on the right-hand side. Other rules deal with managing the left-hand and right-hand sets.

The sequent calculus is more complicated than our system of natural deduction, but it simplifies treating a proof as a formal object, for example to show that a given formula cannot be proved in intuitionistic logic. We will not explore this idea further.

Semantic equivalence (1.5.1)

We previously defined two formulas ϕ and ψ as provably equivalent if $\phi \dashv\vdash \psi$. We can use our notion of semantics to say that ϕ and ψ are **semantically equivalent** if $\phi \models \psi$ and $\psi \models \phi$. In this case, we write $\phi \equiv \psi$ ($\equiv$ is pronounced "is equivalent to").

Because we have proved the soundness and completeness of propositional logic, we know that these two notions coincide. Thus if we wish to show whether a formula is provable or not, we can try all valuations, though this may be inefficient.

Our method of assigning a truth value to a formula is very algebraic in nature, and this was the original source of the connectives we use (George Boole and what is now called Boolean algebra). This suggests yet another method of proof.

Transformational proof

Transformational proof uses algebraic laws that preserve semantic equivalence. For example, commutativity tells us that $\phi \wedge \psi \equiv \psi \wedge \phi$. This describes the commutativity of $\wedge$ as an operator, as verified by its truth table.

Other such laws include associativity, double negation, idempotence [$\phi \wedge \phi \equiv \phi$], absorption [$\phi \vee (\phi \wedge \psi) \equiv \phi$] and deMorgan's laws. These are covered in Math 135 and CS 251.

A transformational proof that two formulas are semantically equivalent looks like an algebraic proof that two expressions are equal. It consists of a series of steps transforming one formula into another, or transforming two formulas into the same formula.

Because a transformational proof is just algebra, which you have had experience with for many years, we will not explore this idea further here. You may have had some practice in transformational proof in Math 135, and it comes up in CS 251 (which sets it in the context of creating digital circuits to compute Boolean functions).

One advantage of transformational proof is that algebraic substitution can be applied to subformulas (as to subexpressions in classical algebra), unlike with natural deduction, whose rules apply to an entire formula.

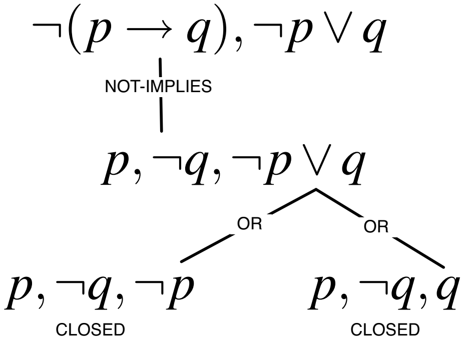
In practice, when doing an informal mathematical proof, we use a combination of natural deduction and transformational proof. If we have a proposition of the form $\phi \wedge \psi$ to prove, we would think nothing of flipping it around to yield $\psi \wedge \phi$ if necessary.

Semantic tableaux

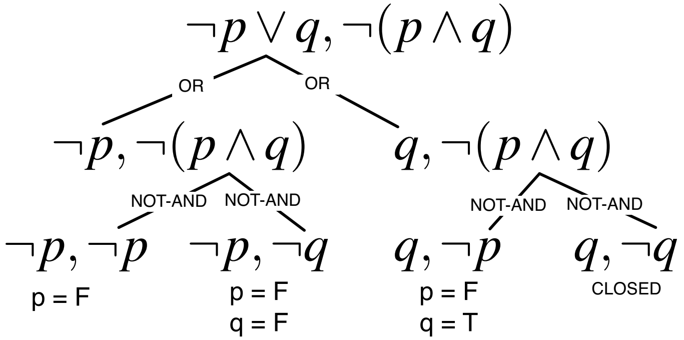
The method of semantic tableaux, invented by the Dutch logician Evert Beth in 1954, is **refutation-based**; it tries to find reasons why a formula is false. More generally, it tries to show that a set of formulas is inconsistent (cannot be made simultaneously true by any valuation).

It does this by constructing a tree (whose nodes are labelled with sets of formulas) via a set of branching rules that operate on leaves. A branch replaces one of the formulas with one or two simpler ones representing possible ways of making the formula true. A node at which there is a contradiction is considered closed. If all leaves are closed, there is no way of making all formulas true; otherwise, a valuation can be read off from the non-closed leaves.

This is best illustrated by a couple of examples. Here's a proof that a set of two formulas is inconsistent.



This proof finds a valuation proving a set of formulas consistent.



As with the other proof techniques discussed in this module, we will not give a complete description of all the rules. Some of the exercises in the textbook (e.g. 1.2.6, 1.5.6) invite you to explore these ideas further.

Some texts use other sets of connectives, most commonly introducing $\leftrightarrow$ to model the English "if and only if". Our system of natural deduction represents $p \leftrightarrow q$ by $(p \to q) \wedge (q \to p)$. Exercise 1.5.3 explores alternate sets of connectives.

An extreme example was provided by Nicod (1917), who gave a system with a single connective (written $|$, but representing "not-and" or NAND as discussed in CS 251), a single axiom schema, and a single inference rule.

We have been concentrating on tautologies, formulas ϕ for which $\models \phi$ holds; such formulas are called **valid**. But many problems that we wish to solve automatically have the property that an instance can be expressed as a formula in propositional calculus such that a solution corresponds to one valuation assigning true to the formula. Such formulas are called **satisfiable**.

There is a simple connection between these two notions.

Thm (1.45): ϕ is satisfiable if and only if $\neg\phi$ is not valid.

Thus any decision procedure for validity can be converted into one for satisfiability, and vice-versa.

But how hard is it to find a proof of a valid formula, or equivalently to decide whether a formula is satisfiable? The best bound we have so far involves trying all 2^n valuations of a formula with n propositional atoms. None of our other proof techniques yields a better worst-case running time.

CS 341 discusses a class of problems called NP-complete, for which there are no known good algorithms (and none believed to exist). They are equivalent in the sense that a good algorithm for one would yield a good algorithm for all. Satisfiability is one of these problems, in fact a central one in proofs of NP-completeness.

Validity is in the related class of co-NP-complete problems, and the existence of a good algorithm for finding proofs for tautologies would imply these classes are equal, which is believed to be false.

Since satisfiability is an important and natural way of phrasing many problems we wish to solve automatically, computer scientists persist in trying to find ways to decide satisfiability efficiently for some classes of formulas.

The concept of **normal forms** is very important in mathematics and computer science. Normal forms provide a restricted manner of describing various properties, where the restriction does not in fact exclude anything, but facilitates proofs and algorithms. Math 235 discusses normal forms for matrices; CS 360 discusses normal forms for grammars.

There are normal forms for both formulas and proofs. We will briefly discuss **conjunctive normal form** for formulas, also covered in CS 251.

Conjunctive normal form

A **literal** L is a propositional atom p or its negation $\neg p$. A **clause** C is a disjunction $L_1 \vee L_2 \vee \ldots \vee L_k$ of literals. A formula is in CNF if it is a conjunction $C_1 \wedge C_2 \wedge \ldots C_m$ of clauses.

It is easy to test a formula in CNF for validity. To make such a formula false, we only have to make one clause C false. The only way we can fail to do so is if some atom p in the clause appears as both p and $\neg p$.

A CNF formula is valid if and only for every clause C, there is an atom p such that both p and $\neg p$ appear in C. Though there is no good algorithm to convert an arbitrary formula to an equivalent CNF formula, conversion to CNF might be a good way of testing validity for some formulas.

One way to convert a formula to CNF, discussed in the text and in CS 251, is to use a truth table. Each line in a truth table corresponds to a valuation. For each valuation making the formula false, we can construct a clause containing all atoms of the formula that is made false by only that valuation. The conjunction of all such clauses is an equivalent formula in CNF.

This is guaranteed to take time exponential in the number of atoms in the formula. A method that might be faster for some formulas is to use ideas from transformational proofs, first eliminating $\rightarrow$ and then recursively applying deMorgan's rules and the distributive law to obtain the right form. The book goes into some detail on this idea in section 1.5.2, but we will not consider it further.

Deciding validity is quite easy for a formula in CNF, but in CS 341 we learn that deciding satisfiability for a formula in CNF remains NP-complete. In fact, it is NP-complete even if every clause has at most three literals in it.

There are classes of formulas for which good algorithms for satisfiability are known. One example often covered in CS 341 is the class of formulas in CNF with at most two literals per clause.

Another class not usually mentioned in CS 341 is the class of Horn formulas, named after the logician who studied them. A Horn clause is a disjunction in which at most one literal is positive (does not contain $\neg$). A Horn formula is a conjunction of Horn clauses. The textbook, in section 1.5.3, gives a different but equivalent definition.

The textbook defines a Horn clause as being of the form $p_{i_1} \wedge p_{i_2} \wedge \ldots \wedge p_{i_k} \rightarrow q$. By transformational proof, it is not hard to show this equivalent to a disjunction with at most one positive literal.

By a process of marking atoms forced to be true in any satisfying valuation, the satisfiability of Horn formulas can be decided in linear time.

The logic programming language Prolog allows programmers to define goals in terms of subgoals. In the statement above, think of the p_{i_j}'s as being subgoals and q as being a goal. Essentially, Prolog statements are Horn clauses. This is part of a more general style of programming known as **declarative programming**, which we will explore using Alloy in the last module of the course.

SAT solvers

The design of algorithms for satisfiability is the subject of active research in the artificial intelligence, hardware verification, and software engineering communities. There are conferences, competitions, and journals devoted to them.

The Alloy Analyzer, a model-checking tool using the declarative language Alloy, does its work with the help of a SAT solver (in fact, you can choose which SAT solver it uses). We will discuss why and how the SAT solver is useful to the Analyzer in module 10.

The text, in section 1.6, covers the design and analysis of two SAT solvers with guaranteed good running time but which cannot guarantee an answer. This is interesting reading for students with some exposure to graph-theoretic algorithms.

Summary

We have focussed on natural deduction in this course, but there are many other proof systems for propositional logic (each requiring their own proofs of soundness and completeness). There are also many systems for deciding validity or satisfiability by focussing more on the semantic side.

Even though propositional logic is not expressive enough to capture many aspects of mathematical proof, it is very important in many subfields of computer science, ranging over hardware, software, theory, and application areas.

Goals of this module

We have not concentrated on technique in this module. On an assignment, we may guide you through working out more of the details of the methods briefly touched upon in lecture, but mostly we just wanted to demonstrate some of the diversity in this area.

We would like you to be familiar with the terminology we have introduced and the connections among different areas of computer science, both for the sake of later modules in this course and for connections to specific topics in other courses in CS.

If you are interested, the course Web page may contain links to further readings on these topics.