Logic and Computation

CS245

Jaiden Ratti

Prof. Stephen Watt

Contents

1 Logic01: Introduction

Logic is the science of reasoning.

Aristotelian Logic: Correctness of an argument depends on form, not content.

All x are y. B is an x. $\therefore B$ is a y.

Logic is fundamental to Computer Science and improves one's general powers of analytical thinking. CS245 will not directly improve your coding skills, it will make you a more effective thinker (which will then improve coding skills).

Propositional Logic

An argument is a set of statements, one or several premises, and a conclusion. A valid (correct, sound) argument is one in which, whenever the premises are true, the conclusion is also true.

For example,

No pure water is burnable.

Some Cuyahoga River water is burnable.

—

Therefore, some Cuyahoga River water is not pure.

This argument is valid.

Note, the conclusion being false does not necessarily prove that an argument is invalid.

To see which arguments are correct, and which are not, we abbreviate essential statements by using letters (p, q, r).

p = "demand rises", q = "companies expand", r = "companies hire workers".

If p then q.

If q then r.

—

If p then r.

This argument is a hypothetical syllogism.

More important logical arguments:

p or q

Not q

—

p

This argument is a disjunctive syllogism.

If p then q

p

—

q

This argument is called modus ponens.

If p then q

Not q

—

Not p

This argument is called modus tollens.

A proposition is a declarative sentence that is either true (1) or false (0), in some context.

Propositional variables are atomic variables. An atomic proposition is a proposition that cannot be broken down into smaller propositions. A proposition that is not atomic is called compound.

Or, and, not, if-then are referred to as logical connectives.

Let p be a proposition, the compound proposition $\neg p$ (not p) is true when p is false.

p	$\neg p$
1	0
0	1

Let p and q be two propositions. The proposition $p \wedge q$ (p and q) is true when both p and q are true, and false otherwise. Referred to as the conjunction of p and q.

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

When writing truth tables, use the convention in decreasing lexicographic ordering.

Let p and q be two propositions. The proposition $p \vee q$ is true when either p, or q, or both p and q are true, and is false when both p and q are false. Referred to as the disjunction of p and q.

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

The English "or" has two different meanings.

Exclusive or: "You can either have soup or salad" (can have one or the other, but not both).

Inclusive or: "The computer has a bug, or the input is erroneous".

To avoid ambiguity, $p \vee q$ translates to the inclusive or.

Let p and q be two propositions. Then $p \implies q$ (if p, then q) is false when p is true and q is false, and true otherwise. Referred to as the implication of p and q.

Means that, whenever p is correct, so is q. p is the antecedent, q is the consequent.

If p is false, then $p \implies q$ is vacuously true. This is consistent with everyday speech.

The following are logically equivalent.

$p \implies q \equiv$ If p then $q \equiv p$ is sufficient for $q \equiv p$ only if $q \equiv p$ implies $q \equiv q$ if p.

Let p and q be two propositions. Then $p \iff q$ (p if and only if q) is true whenever p and q have the same truth values. Referred to as equivalence (or biconditional). We often use *iff* as an abbreviation for if and only if.

An ambiguous sentence usually has multiple interpretations.

Imprecision arises from the use of qualitative descriptions.

$\neg$ is the only unary connective. All other connectives are binary connectives (require two propositions). They are also symmetric (except for $\implies$).

Translations Between English and Propositional Logic

p	q	$p \implies q$
1	1	1
1	0	0
0	1	1
0	0	1

p	q	$p \iff q$
1	1	1
1	0	0
0	1	0
0	0	1

If I feed my fish, and I change my fish's tank filter, then my fish will be healthy.

p : I feed my fish, q : I change my fish's tank filter, r : My fish will be healthy.

$((p \wedge q) \implies r)$

2 Logic02: Syntax

With connectives we can combine propositions. To prevent ambiguity we introduce fully parenthesized expressions that can be parsed uniquely.

We construct the propositional language $\mathcal{L}^p$ which is the formal language of propositional logic.

The set of formulas in $\mathcal{L}^p$, denoted by Form($\mathcal{L}^p$), will then be defined by a set of formation rules which produce expressions in $\mathcal{L}^p$ belonging to Form($\mathcal{L}^p$).

Strings in $\mathcal{L}^p$ comprise three classes of symbols. Propositional symbols, connective symbols, punctuation symbols (,).

Two expressions U and V are equal if and only if they are of the same length and have the same symbols in the same order.

Note $\epsilon U = U\epsilon = U$ for any expression U.

Set of Formulas of $\mathcal{L}^p$

Definition: Atom($\mathcal{L}^p$) - is the set of expressions of $\mathcal{L}^p$ that consist of a proposition symbol only.

Definition: The set Form($\mathcal{L}^p$), of formulas of $\mathcal{L}^p$, is defined recursively as:

Base: Every atom in Atom($\mathcal{L}^p$) is a formula in Form($\mathcal{L}^p$).

Recursion: If A and B are formulas in Form($\mathcal{L}^p$), then:

1. $(\neg A)$ is a formula in Form($\mathcal{L}^p$)
2. $(A \wedge B)$ is a formula in Form($\mathcal{L}^p$)
3. $(A \vee B)$ is a formula in Form($\mathcal{L}^p$)
4. $(A \implies B)$ is a formula in Form($\mathcal{L}^p$)
5. $(A \iff B)$ is a formula in Form($\mathcal{L}^p$)

Restriction: No other expressions in $\mathcal{L}^p$ are formulas in Form($\mathcal{L}^p$).

Examples:

p, q, r are atomic formulas in Atom($\mathcal{L}^p$), and thus formulas in Form($\mathcal{L}^p$).

$((p \wedge q) \implies r)$ and $((\neg q) \iff (p \vee s))$ are formulas in Form($\mathcal{L}^p$), but not atomic formulas in Atom($\mathcal{L}^p$).

$p \wedge \wedge \wedge (((r \implies$ is an expression in $\mathcal{L}^p$, but it is neither an atomic formula in Atom($\mathcal{L}^p$), nor a formula in Form($\mathcal{L}^p$).

Example (Generating Formulas).

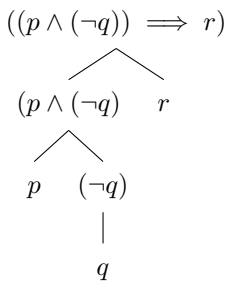
The expression,

$$((p \wedge q) \implies ((\neg p) \iff (q \wedge r)))$$

is a formula. Formulation rules are as follows.

- p, q, r are in Form($\mathcal{L}^p$) by Definition of Form($\mathcal{L}^p$) (BASE).
- $(\neg p)$ is in Form($\mathcal{L}^p$) (RECURSION).
- $(q \wedge r)$ and $(p \vee q)$ are in Form($\mathcal{L}^p$) (RECURSION).
- $((\neg p) \iff (q \wedge r))$ is in Form($\mathcal{L}^p$) (RECURSION applied to $(\neg p)$ and $(q \wedge r)$).
- $((p \vee q) \implies ((\neg p) \iff (q \wedge r)))$ is in Form($\mathcal{L}^p$) (RECURSION).

We can use parse trees to analyze formulas.



Every formula in $\mathcal{L}^p$ has the same number of occurrences of left and right parentheses.

Any non-empty proper initial segment of a formula in $\mathcal{L}^p$ has more occurrences of left than right parentheses. Any non-empty proper terminal segment of a formula in $\mathcal{L}^p$ has fewer occurrences of left than right parentheses.

Neither a non-empty proper initial segment nor a non-empty proper terminal segment of a formula can itself be a formula of $\mathcal{L}^p$.

Unique Readability Theorem: Every formula of $\mathcal{L}^p$ is of exactly one of the six forms: An atom, $(\neg A), (A \wedge B), (A \vee B), (A \implies B)$, or $(A \iff B)$ and in each case, it is of that form in exactly one way.

To prove these claims, we will use mathematical induction.

The statement "every natural number has property P" corresponds to a sequence of statements.

$P(0), P(1), P(2), P(3), P(4), \ldots$ where $P(2)$ means P holds for 2.

Principle of mathematical induction:

If we establish two things:

1. 0 has property P, and
2. whenever a natural number has property P, then the next natural number also has property P.

Then we may conclude that every natural number has property P.

Observations:

To talk about something, give it a name (e.g. property P, number k).

A formula is a textual object. In this text, we can substitute one symbol or expression for another. For example, we often put $k + 1$ in place of k.

The induction principle gives a template for a proof:

- The proof has two parts: Base Case and Inductive Step.
- In the Inductive Step, hypothesize $P(k)$ and prove $P(k + 1)$ from it.

How do we prove properties of formulas?

How to prove a statement along the lines of "Every formula in $\mathcal{L}^p$ has property P".

A formula is not a natural number, but it suffices to prove any one of the following.

For every natural number n, every formula with n or fewer symbols has property P.

OR

For every natural number n, every formula with n or fewer connectives has property P.

OR

For every natural number n, every formula whose parse tree has height less than or equal to n has property P.

OR

For every natural number n, every formula producible with n or fewer uses of the formation rules has property P.

Alternatively, we can use the fact that $\text{Form}(\mathcal{L}^p)$ is a recursively defined set, and use structural induction to prove properties about formulas in $\text{Form}(\mathcal{L}^p)$.

Recursively Defined Sets

Inductive definition of sets consist of a universe, core set, and operations (functions).

Given any subset $Y \subseteq X$ and any set F of operations (functions $f : X^k \to X$ for any $k \geq 1$), Y is closed under F if, for every $f \in F$, (say f is a $k-$ary function) and every $y_1, \ldots, y_k \in Y, f(y_1, \ldots, y_k) \in Y$.

Y is a minimal set with respect to a property R if

1. Y satisfies R, and
2. for every set Z that satisfies $R, Y \subseteq Z$.

We can formally define $I(X, A, F) =$ The minimal subset of X that contains A, and is closed under the operations in F.

Example: The set of Natural numbers:

$$\mathbb{N} = I\left(\mathbb{R}, \{0\}, \{ \underbrace{f(x) = x + 1}_{\text{successor function}} \}\right).$$

Structural Induction

The strategy to prove a property R holds for every element of a set $I(X, A, F)$ is as follows.

1. Prove that $R(a)$ holds for every a in the core set A (the base case).
2. Prove that, for every $k-$ary $f \in F$ (for any $k \geq 1$), and any $y_1, \ldots, y_k \in X$ such that $R(y_1), \ldots, R(y_k)$ all hold, we also have that $R(f(y_1, \ldots, y_k))$ holds (the inductive case).

The core objects are the base, recursion (collection of rules indicating how to form new set objects from those already known to be in the set), restriction (a statement that no objects belong to the set other than those coming from base and recursion).

Examples

The set of natural numbers $\mathbb{N}$ is a recursively defined set with one formation rule ("add 1").

1. Base: 0 is a natural number in $\mathbb{N}$
2. Recursion: If k is a natural number in $\mathbb{N}$, then $k + 1$ is a natural number in $\mathbb{N}$.
3. Restriction: No other numbers are in $\mathbb{N}$.

Structural Induction applied to $\text{Form}(\mathcal{L}^p)$

Suppose R is a property. If

- Every atomic formula $p \in \text{Atom}(\mathcal{L}^p)$ satisfies property R, and
- If formulas A and B in $\text{Form}(\mathcal{L}^p)$ satisfy property R, then:
  1. $(\neg A)$ satisfies property R,
  2. $(A \wedge B)$ satisfies property R,
  3. $(A \vee B)$ satisfies property R,
  4. $(A \implies B)$ satisfies property R,
  5. $(A \iff B)$ satisfies property R,

it follows that every formula in $\text{Form}(\mathcal{L}^p)$ satisfies property R.

We shall prove the following.

Lemma

Every formula in $\text{Form}(\mathcal{L}^p)$ has an equal number of left and right parentheses.

Proof:

We use structural induction. The property to prove is $R(A)$*: A has an equal number of left and right parentheses* for every formula A in $\text{Form}(\mathcal{L}^p)$.

Base Case:

A is an atom.

A has zero left and right parentheses, as it is only a proposition symbol. Thus $R(A)$ holds. This completes the proof of the Base Case.

Inductive Step:

Define the notation

- $\ell(A)$ denotes the number of '(' symbols in A.
- $r(A)$ denotes the number of ')' symbols in A.

Subcase of $\neg$:

Assume A is $(\neg B)$.

Inductive Hypothesis: Formula B has property R (i.e. $\ell(B) = r(B)$).

Then we have

$$\begin{aligned} \ell((\neg B)) &= 1 + \ell(B) \quad \text{(inspection)} \\ &= 1 + r(B) \quad \text{(induction hypothesis: } R(B)\text{)} \\ &= r((\neg B)) \quad \text{(inspection)} \end{aligned}$$

Subcases $(\wedge, \vee, \implies, \iff)$

Inductive Hypothesis: Formulas B and C both have property R.

To prove: Each of the formulas $(B \wedge C), (B \vee C), (B \implies C)$, and $(B \iff C)$ has property R.

Without loss of generality, we consider $(B \bullet C)$.

We calculate $\ell((B \bullet C))$:

$$\begin{aligned} \ell(B \bullet C)) &= 1 + \ell(B) + \ell(C) \quad \text{(inspection)} \\ &= 1 + r(B) + r(C) \quad \text{(I.H) } R(B) \text{ and } R(C) \\ &= r((B \bullet C)) \quad \text{(inspection)} \end{aligned}$$

This concludes the proof of the composite inductive step, the inductive proof and thus the example.

Unique Readability Theorem

Theorem: Every formulas is exactly one of: an atom, $(\neg B), (B \wedge C), (B \vee C), (B \implies C), (B \iff C)$ and, in each case, it is of that form in exactly one way.

Prove this using structural induction

Base Case: Trivial, as every proposition symbol is an atom.

Inductive Step Idea: We will have to consider e.g., formulas of the form $(B \implies C)$ (one of the five subcases of the Inductive Step).

An example of an "implication" formula (a formula of the type $(B \implies C)$, where B and C are formulas) which we have to consider is $(p \wedge q) \implies r)$, which has $B = (p \wedge q)$, and $C = r$.

Question: Is this the only way to "parse" the formula $((p \wedge q) \implies r)$? What about parsing the same formulas as a conjunction of two formulas, that is,

$$((p \wedge q) \implies r) = (B' \wedge C')$$

where $B' = \underline{(p}$ and $C' = \underline{q) \implies r}$.

Fortunately, neither B' nor C' is a formula.

Does this proof idea always work?

How can we make sure that such a proof for the Inductive Step works for every formula $(B \implies C)$?

That is, if we have a formula $(B \implies C)$ where B and C are both formulas, and $(B \implies C) = (B' \wedge C')$, how can we argue that neither B' nor C' can be a formula?

Hint: Can B' or C' have an equal number of left and right parentheses.

If not, why not?

To do the proof, we actually need to know more about formulas. This illustrates a common feature of inductive proofs: they often prove more than just the statement given in the theorem.

Proof:

Property $P(n)$:

Every formula A containing at most n connectives satisfies all three of the following properties:

(a) The first symbol of A is either '(' or a proposition symbol.

(b) A has an equal number of '(' and ')', and each non-empty proper initial segment of A has more '(' then ')'.

(c) A has a unique construction as a formula.

We will prove that the property $P(n)$ holds for all n, by induction on n (the number of connectives).

Base Case: The statement holds for $n = 0$ (a formula with 0 connectives is a proposition symbol, it has 0 left and right parentheses, and has no non-empty proper initial/terminal segments).

Inductive Step:

Inductive Hypothesis: $P(k)$ holds for some natural number k.

To show that $P(k + 1)$ holds, let formula A have $k + 1$ connectives.

The proof of the Inductive Step has five subcases, one for each of the formation rules (connectives) in the recursive definition of Form($\mathcal{L}^p$).

First subcase: $A = (\neg B)$, where $\neg$ is the $(k + 1)$st connective, and the inductive hypothesis is that B has properties (a), (b), and (c).

(a): By construction, $(\neg B)$ has Property (a), since it begins with '('.

(b): Since B has an equal number of left and right parentheses, so does $(\neg B)$. For the second part of Property (b), we check the following subcases of every possible non-empty proper initial segment, x, of $(\neg B)$:

1. x is "(": Then x has one "(" symbol, and no ")" symbols.
2. x is "($\neg$": Then x has one "(" symbol, and no ")" symbols.
3. x is "($\neg z$", for some non-empty proper initial segment z of B: Since by Inductive Hypothesis, z has more "(" than ")" symbols, so does x.
4. x is "($\neg B$": Since B has equally many "(" and ")" symbols, x has more ")" than "(" symbols.

In every case, x has more "(" than ")" symbols. Hence $(\neg B)$ has Property (b).

(c): Because B has Property (c), by construction so does $(\neg B)$.

The other four subcases: Assume that $A = (B \bullet C)$, for some formulas B and C, where $(k+1)$st connective is the binary connective $\bullet \in \{\wedge, \vee, \implies, \iff\}$.

Inductive Hypothesis: Both B and C have properties (a), (b), (c). Verifying properties (a), (b) for $(B \bullet C)$ is analogous to the case of $\neg$.

We prove only (c). First, we show that formula A cannot be decomposed in two different ways, with two binary connectives, as $A = (B \bullet C) = (B' \bullet' C')$, for formulas B, C, B', C'. Equivalently:

If the same formula A can be decomposed as $A = (B \bullet C) = (B' \bullet' C')$, for formulas B' and C' and binary connective $\bullet'$, then $B = B', \bullet = \bullet'$, and $C = C'$.

Note: $A = (B \bullet C) = (B' \bullet' C')$ means that $(B \bullet C)$ and $(B' \bullet' C')$ are two different compositions of the same formula (same length and the same sequence of symbols, in the same order).

Recall that $A = (B \bullet C) = (B' \bullet' C')$.

Case (1): If B' has the same length as B, then they must be the same string (both start at the second symbol of A).

Case (2): B' is a non-empty proper prefix of B. Since B and B' are formulas with at most k connectives, the inductive hypothesis applies to them. In particular, they have property (b).

Since B' has the first half of property (b), B' should have an equal number of left and right parentheses.

Since B has the second half of property (b), and since B' is a non-empty proper prefix of the formula B, it follows that B' should have strictly more left than right parentheses.

We reached a contradiction, so Case (2) cannot hold.

Case (3): B is a non-empty proper prefix of B' - impossible, using a similar reasoning as in Case (2).

Since Case (2) and Case (3) are impossible, the only case that can hold is Case (1), whereby the two decompositions of the formula A must coincide. Thus, A has a unique construction, as required by (c).

Second, we show that formula A cannot be decomposed in two different ways, once with a binary and once with unary connective, as $A = (B \bullet C) = (\neg D)$, for formulas B, C, D.

Assume that $A = (B \bullet C) = (\neg D)$. If we delete the first symbol from both $(B \bullet C)$ and $(\neg D)$ we obtain $B \bullet C) = \neg D)$. Then, the formula B starts with $\neg$, a contradiction with part (a) of the inductive hypothesis. Hence, this second situation cannot hold.

Since these are the only two possibilities for $A = (B \bullet C)$, this proves the unique construction of A, as required by Property (c).

We will define the semantics (meaning) of a formula from its syntax (its structure, as determined by the formation rules).

Unique readability ensures unambiguous formulas. How?

Given a formula, determine its subformulas by counting parentheses.

Precedence rules:

$\neg$ has precedence over $\wedge$

$\wedge$ has precedence over $\vee$

$\vee$ has precedence over $\implies$

$\implies$ has precedence over $\iff$

Examples:

$\neg p \vee q$ is to be understood as $((\neg p) \vee q)$.

$p \wedge q \vee r$ is to be understood as $((p \wedge q) \vee r)$.

And so on.

Suppose $A = (\neg((p \wedge q) \vee ((\neg p) \implies r)))$.

The scope of the first $\neg$ is $((p \wedge q) \vee ((\neg p) \implies r))$ and of the second $\neg$ is p.

The left and right scopes of $\wedge$ are p and q.

And so on.

3 Logic03: Semantics

Syntax is concerned with the rules used for constructing the formulas in Form($\mathcal{L}^p$). This is similar to computer science, where syntax refers to the rules governing the composition of well-formed expressions in a programming language.

Semantics is concerned with meaning. Atoms are intended to express simple propositions (sentences). The connectives take their intended meanings $\neg, \wedge, \vee, \implies, \iff$ express "not", "and", "or", "if, then", and "iff". The meaning of a non-atomic formula, that is, its truth value is derived from the truth values of its constituent atomic formulas, and the meanings of the connectives.

Example:

If you take a class in computers and if you do not understand recursion, you will not pass.

We want to know exactly when this statement is true and when it is false.

Define:

p : "You take a class in computers."

q : "You understand recursion."

r : "You pass."

The statement becomes $(p \wedge \neg q) \implies \neg r$.

The truth table for $(p \wedge \neg q) \implies \neg r$ is

p	q	r	$\neg q$	$p \wedge \neg q$	$\neg r$	$(p \wedge \neg q) \implies \neg r$
1	1	1	0	0	0	1
1	1	0	0	0	1	1
1	0	1	1	1	0	0
1	0	0	1	1	1	1
0	1	1	0	0	0	1
0	1	0	0	0	1	1
0	0	1	1	0	0	1
0	0	0	1	0	1	1

Two propositional formulas A and B in Form($\mathcal{L}^p$) are called (logically) equivalent (denoted $A \vDash B$) if $A^t = B^t$ for every truth valuation, t. (Equivalently, if A and B have the same truth table.)

A truth table list the values of a formula under all possible truth valuations.

Fix a set $\{0, 1\}$ of truth values. We interpret 0 as false and 1 as true.

Definition. A truth valuation is a function t

$$t : \text{Atom}(\mathcal{L}^p) \longrightarrow \{0, 1\}$$

with the set of all proposition symbols as domain and $\{0, 1\}$ as range.

Convention: For $A \in \text{Atom}(\mathcal{L}^p)$ we denote A^t the value $t(A) \in \{0, 1\}$ that A takes under truth valuation t.

In practice, we restrict the truth valuation to the set of proposition symbols in the formulas under consideration.

Then, a truth valuation corresponds to a single row in the truth table.

Definition: Let t be a truth valuation. The value of a formula in $\text{Form}(\mathcal{L}^p)$ with respect to the given truth valuation t is defined recursively as follows:

1. If the formula is a proposition symbol p, then $p^t \in \{0, 1\}$ given by the definition of t.
2. $(\neg A)^t = \begin{cases} 1 & \text{if } A^t = 0 \\ 0 & \text{if } A^t = 1 \end{cases}$
3. $(A \wedge B)^t = \begin{cases} 1 & \text{if } A^t = B^t = 1 \\ 0 & \text{otherwise} \end{cases}$
4. $(A \vee B)^t = \begin{cases} 1 & \text{if } A^t = 1 \text{ or } B^t = 1 \text{ or both} \\ 0 & \text{otherwise} \end{cases}$
5. $(A \implies B)^t = \begin{cases} 1 & \text{if } A^t = 0 \text{ or } B^t = 1 \text{ or both} \\ 0 & \text{otherwise} \end{cases}$
6. $(A \iff B)^t = \begin{cases} 1 & \text{if } A^t = B^t \\ 0 & \text{otherwise} \end{cases}$

Suppose A is the formula $p \vee q \implies q \wedge r$, and t is a truth valuation such that $p^t = q^t = r^t = 1$.

Then we have $(p \wedge q)^t = 1, (q \wedge r)^t = 1$ and therefore $A^t = 1$.

Suppose t_1 is another truth valuation, $p^{t_1} = q^{t_1} = r^{t_1} = 0$. Then we have $(p \vee q)^{t_1} = 0, (q \wedge r)^{t_1} = 0$ and therefore $A^{t_1} = 1$.

If t_2 is yet another truth valuation, with $p^{t_2} = 1$ and $r^{t_2} = q^{t_2} = 0$, then $A^{t_2} = 0$.

The above example illustrates that, for a particular formula, its value under one truth valuation may (or may not) differ from its value under a different truth valuation.

Definition: We say that a truth valuation t satisfies a formula A in $\text{Form}(\mathcal{L}^p)$ iff $A^t = 1$.

We use the capital Greek letter Σ to denote any set of formulas.

Definition: The value of a set of formulas Σ under truth valuation t is defined as:

$$\Sigma^t = \begin{cases} 1 & \text{if for each formula } B \in \Sigma, B^t = 1, \\ 0 & \text{otherwise} \end{cases}$$

Definition: A set of formulas $\Sigma \subseteq \text{Form}(\mathcal{L}^p)$ is satisfiable if and only if there exists a truth valuation t such that $\Sigma^t = 1$. If, in the other hand, there is no truth valuation t such that $\Sigma^t = 1$ (or, equivalently, if $\Sigma^t = 0$ for all truth valuations t), then the set Σ is called unsatisfiable.

Observations:

1. If for a truth valuation t we have that $\Sigma^t = 1$, then t is said to satisfy Σ, and Σ is said to be satisfied by (under) t.
2. Note that $\Sigma^t = 1$ means that under the truth valuation t, all the formulas of Σ are true.
3. On the other hand, $\Sigma^t = 0$ means that for at least one formula $B \in \Sigma$, we have that $B^t = 0$.
4. In particular, $\Sigma^t = 0$ does not necessarily mean that $C^t = 0$ for every formula C in Σ.

Definition: A formula A is a tautology if and only if it is true under all possible truth valuations, i.e. iff for any truth valuation t, we have that $A^t = 1$.

Definition: A formula A is a contradiction if and only if it is false under all possible truth valuations, i.e. iff for every truth valuation t, we have that $A^t = 0$.

Definition: A formula that is neither a tautology nor a contradiction is called contingent.

The law of excluded middle ("tertium non datur") states that $p \vee \neg p$ is a tautology.

If A is a tautology that contains the proposition symbol p, one can determine a new expression by replacing all instances of p by an arbitrary formula. The resulting formula A' is also a tautology.

For example, $p \vee \neg p$ is a tautology.

Replace all instances of p by any formula we like, say by $p \wedge q$. The resulting formula $A' = (p \wedge q) \vee \neg(p \wedge q)$ is again a tautology.

Theorem: Let A be a tautology and let $p_1, p_2, \ldots, p_n$ be the proposition symbols of A. Suppose that $B_1, B_2, \ldots, B_n$ are arbitrary formulas. Then, the formula obtained by replacing p_1 by B_1, p_2 by $B_2, \ldots, p_n$ by B_n, is a tautology.

Important Contradiction - Law of contradiction: "Nothing can both be, and not be", that is, $\neg(p \wedge \neg p)$ is a tautology, equivalently, $(p \wedge \neg p)$ is a contradiction.

Contradictions and tautologies are related. A is a tautology if and only if $\neg A$ is a contradiction. Being satisfiable is the negation of being a contradiction.

Logical arguments consist of Premises followed by a Conclusion. Arguments can be Correct (valid, sound) or Incorrect (invalid, unsound).

Definition:

Suppose $\Sigma \subseteq \text{Form}(\mathcal{L}^p)$ and $A \in \text{Form}(\mathcal{L}^p)$.

A is a tautological consequence of Σ (that is, of the formulas in Σ), written as $\Sigma \vDash A$, $\iff$ for any truth valuation t, we have that $\Sigma^t = 1$ implies $A^t = 1$.

Observations:

- $\vDash$ is not a symbol of the formal propositional language and $\Sigma \vDash A$ is not a formula.
- $\Sigma \vDash A$ is a statement (in the metalanguage) about Σ and A.
- We write $\Sigma \nvDash A$ for "not $\Sigma \vDash A$".
- If $\Sigma \vDash A$, we say that the formulas in Σ (tauto)logically imply formula A.

When Σ is the empty set, we obtain the important special case of tautological consequence, $\emptyset \vDash A$.

By definition, $\emptyset \vDash A$ means that the following holds: "For any truth valuation t, if $\emptyset^t = 1$ then $A^t = 1$." where $\emptyset^t = 1$ means "For any B, if $B \in \emptyset$ then $B^t = 1$".

Because $B \in \emptyset$ is false, "$\emptyset^t = 1$" is always (vacuously) true. Consequently, $\emptyset \vDash A$ means that A is always true (is a tautology).

Intuitively speaking, $\Sigma \vDash A$ means that the truth of the formulas in Σ is a sufficient condition for the truth of A.

Since $\emptyset$ has no formulas, $\emptyset \vDash A$ means that the truth of A is unconditional, hence A is a tautology.

Let $\Sigma = \{A_1, A_2, \ldots, A_n\} \subseteq \text{Form}(\mathcal{L}^p)$ be a set of formulas (premises) and $C \in \text{Form}(\mathcal{L}^p)$ be a formula (conclusion). The following are equivalent.

- The argument with premises $A_1, A_2, \ldots, A_n$ and conclusion C is valid.
- $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \implies C$ is a tautology.
- $(A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge \neg C)$ is a contradiction.
- The formula $(A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge \neg C)$ is not satisfiable.
- The set $\{A_1, A_2, \cdots, A_n, \neg C\}$ is not satisfiable.

- C is a tautological consequence of Σ, i.e. $\{A_1, A_2, \ldots, A_n\} \vDash C$.

Consider an argument with premises $A_1, A_2, \ldots, A_n$ and conclusion C.

The conclusion C is true, if the following two conditions hold:

- The argument with premises $A_1, A_2, \ldots, A_n$ and conclusion C is valid (sound, correct),
- The premises $A_1, A_2, \ldots, A_n$ are all true.

The validity of an argument does not guarantee the truth of the conclusion. Only when the argument is valid and the premises are all true, is the conclusion guaranteed to be true.

Definition:

For two formulas, we write $A \dashv\vDash B$ to denote "$A \vDash B$" and "$B \vDash A$".

A and B are said to be tautologically equivalent (or simply equivalent) if and only if $A \dashv\vDash B$ holds.

Tautological equivalence is weaker than equality of formulas. For example, if $A = \neg(p \wedge q)$ and $B = (\neg p \vee \neg q)$ then $A \dashv\vDash B$, as can be proved by a truth table, but $A \neq B$.

Note that,

$A \vDash B$ if and only if $A \implies B$ is a tautology.

$A \implies B$ is a formula, which can be true or false.

$\emptyset \vDash A \implies B$ means that $A \implies B$ is a tautology.

$A \dashv\vDash$ if and only if $A \iff B$ is a tautology.

$A \iff B$ is a formula, which can be true or false.

$\emptyset \vDash A \iff B$ means that $A \iff B$ is a tautology.

To prove that the tautological consequence $\Sigma \vDash A$ (prove the validity of the argument with premises Σ and conclusion A) we must show that any truth valuation t satisfying Σ also satisfies A. One way to show this is by using truth tables.

Example: Show that $\{p \implies q, q \implies r\} \vDash (p \implies r)$

The premises are $A_1 = p \implies q$ and $A_2 = q \implies r$; the conclusion is $p \implies r$.

p	q	r	$p \implies q$	$q \implies r$	$A_1 \wedge A_2$	concl: $p \implies r$
1	1	1	1	1	1*	1
1	1	0	1	0	0	0
1	0	1	0	1	0	1
1	0	0	0	1	0	0
0	1	1	1	1	1*	1
0	1	0	1	0	0	1
0	0	1	1	1	1*	1
0	0	0	1	1	1*	1

The truth valuations in rows 1,5,7,8 (with *) are all the truth valuations which make all premises true, that is, which satisfy $\Sigma = \{p \implies q, q \implies r\}$. For each of these four truth valuations, the conclusion $p \implies r$ is also true (is satisfied).

This shows that

$$\{p \implies q, q \implies r\} \vDash (p \implies r)$$

This further means that the argument

Premise 1: $p \implies q$

Premise 2: $q \implies r$

Conclusion: $p \implies r$

is a valid argument.

Proving that an argument is not valid

Example: Prove that $(p \implies q) \vee (p \implies r) \nvDash p \implies (q \wedge r)$

Solution: Find at least one row in the truth table in which the premises are true but the conclusion is false.

The row in the truth table that corresponds to the truth valuation t which assigns $p^t = 1, q^t = 1, r^t = 0$ is one such counterexample.

- Note that several such truth valuations may exist
- We only need one such truth valuation (that makes all premises true but the conclusion false), in order to prove that an argument is not valid.

Truth tables get large fast. If a formula has n proposition symbols and m occurrences of connectives, we get 2^n rows and $\leq n + m$ columns. We need another method for proving argument validity.

We can prove by contradiction (different than the word "contradiction" in propositional logic)

Example: Show that $\{A \implies B, B \implies C\} \vDash (A \implies C)$.

Proof: Assume the contrary, that is, $\{A \implies B, B \implies C\} \nvDash (A \implies C)$

This means that there is a truth valuation t that makes all premises true but the conclusion false, that is,

1. $(A \implies B)^t = 1$,
2. $(B \implies C)^t = 1$,
3. $(A \implies C)^t = 0$.

(4) By (3), we have that $A^t = 1$ and $C^t = 0$

(5) By (1) and the fact that $A^t = 1$, we have $B^t = 1$

From $B^t = 1$ and (2), we deduce $C^t = 1$, which contradicts (4).

Since we have reached a contradiction, our assumption that the argument was invalid was false, hence the opposite is true: The argument is valid.

To prove $\Sigma \nvDash A$ we must construct a counterexample. A truth valuation t satisfying Σ but not satisfying A.

Example: Show that $\{(p \implies \neg q) \vee r, q \wedge \neg r, p \iff r\} \nvDash (\neg p \wedge (q \implies r))$

Let t be the truth valuation $p^t = 0, q^t = 1, r^t = 0$.

Then we have

$$\begin{aligned} ((p \implies \neg q) \vee r)^t &= 1 \\ (q \wedge \neg r)^t &= 1 \\ (p \iff r)^t &= 1 \\ (\neg p \wedge (q \implies r))^t &= 0 \end{aligned}$$

We have found a counterexample (truth valuation that makes all premises true but the conclusion false), hence the argument is invalid.

De Morgan's Law

Consider the following two statements:

It is not true that he is informed and honest.

He is either not informed, or he is not honest.

Intuitively, these two statements are logically equivalent.

The first statement translates to $\neg(p \wedge q)$, whereas the second into $\neg p \vee \neg q$.

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$	$\neg(p \wedge q) \iff (\neg p \vee \neg q)$
1	1	1	0	0	1
1	0	0	1	1	1
0	1	0	1	1	1
0	0	0	1	1	1

De Morgan's Law: $\neg(p \wedge q) \dashv\vdash (\neg p \vee \neg q)$

Dual De Morgan's Law: $\neg(p \vee q) \dashv\vdash (\neg p \wedge \neg q)$

De Morgan's Laws are used to negate conjunctions and disjunctions, and show how to distribute $\neg$ over $\wedge$, and over $\vee$.

To negate a conjunction, take the disjunction of the negations of the conjuncts.

To negate a disjunction, take the conjunction of the negations of the disjuncts.

Definition: Given an implication of the form $(p \implies q)$, the formula $(\neg q \implies \neg p)$ is called the contrapositive of $(p \implies q)$, and the formula $(q \implies p)$ is called the converse of $(p \implies q)$.

Via truth table, it is obvious that $p \implies q \dashv\vdash \neg q \implies \neg p$.

We can use this fact in our proofs. Sometimes, it is easier to prove the contrapositive instead of a direct proof.

Note that the converse of an implication is not equivalent to it.

It is also obvious that $(p \iff q) \dashv\vdash ((p \implies q) \wedge (q \implies p))$.

Tautological Equivalences

Lemma: If $A \dashv\vdash A'$ and $B \dashv\vdash B'$, then

1. $\neg A \dashv\vdash \neg A'$
2. $A \wedge B \dashv\vdash A' \wedge B'$
3. $A \vee B \dashv\vdash A' \vee B'$
4. $A \implies B \dashv\vdash A' \implies B'$
5. $A \iff B \dashv\vdash A' \iff B'$

Theorem: (Replaceability of tautologically equivalent formulas) Let A be a formula which contains a subformula B. Assume that $B \dashv\vdash C$, and let A' be the formula obtained by simultaneously replacing in A some (but not necessarily all) occurrences of the formula B by formula C, then $A' \dashv\vdash A$.

Theorem: (Duality) Suppose A is a formula composed only of atoms and the connectives $\neg, \vee, \wedge$, by the formation rules concerned these three connectives. Suppose $\Delta(A)$ results from simultaneously replacing in A all occurrences of $\wedge$ with $\vee$, all occurrences of $\vee$ with $\wedge$, and each atom with its negation. Then $\neg A \dashv\vdash \Delta(A)$.

Both of these proofs are by structural induction.

This table is handy.

Σ	C	$\Sigma \vDash C$?
Not Satisfiable	Contradiction	Yes
Not Satisfiable	Satisfiable, not a tautology	Yes
Not Satisfiable	Tautology	Yes
Satisfiable	Contradiction	No
Satisfiable	Satisfiable, not a tautology	Maybe
Satisfiable	Tautology	Yes

4 Logic04: Propositional Calculus: Essential Laws, Normal Forms

In standard algebra, expressions in which the variables and constants represent numbers are manipulated. Consider for instance the expression $(a + b) - b$.

This expression yields a.

In fact, we are so accustomed to these manipulations we are not aware of what is behind each step.

Here we use the identities:

$$\begin{aligned}(x + y) - z &= x + (y - z)\\ y - y &= 0\\ x + 0 &= x\end{aligned}$$

Consider the formula $(p \wedge q) \wedge \neg q$.

This formula can be simplified in a similar way, except that (tauto)logical equivalences take the place of algebraic identities.

$$\begin{aligned}(A \wedge B) \wedge C &\Leftrightarrows A \wedge (B \wedge C)\\ (A \wedge \neg A) &\Leftrightarrows 0\\ A \wedge 0 &\Leftrightarrows 0\end{aligned}$$

We can now apply these tautological equivalences to conclude $(p \wedge q) \wedge \neg q \Leftrightarrows p \wedge (q \wedge \neg q) \Leftrightarrows p \wedge 0 \Leftrightarrows 0$

Since the symbolic treatment of $\Longrightarrow$ and $\Longleftrightarrow$ is relatively cumbersome, one usually removes them before performing further formula manipulations.

To remove the connective $\Longrightarrow$ one uses the logical equivalence

$$A \Longrightarrow B \Leftrightarrows \neg A \vee B$$

There are two ways to remove the connective $\Longleftrightarrow$:

$$\begin{aligned}&A \Longleftrightarrow B \Leftrightarrows (A \wedge B) \vee (\neg A \wedge \neg B), \text{and}\\ &A \Longleftrightarrow B \Leftrightarrows (A \Longrightarrow B) \wedge (B \Longrightarrow A) \Leftrightarrows (\neg A \vee B) \wedge (\neg B \vee A)\end{aligned}$$

Example: Remove $\Longrightarrow$ and $\Longleftrightarrow$ from the following formula:

$$(p \Longrightarrow q \wedge r) \vee ((r \Longleftrightarrow s) \wedge (q \vee s))$$

Solution:

$$(\neg p \vee q \wedge r) \vee (((\neg r \vee s) \wedge (\neg s \vee r)) \wedge (q \vee s))$$

Essential Laws for Propositional Calculus

These laws allow us to simplify formulas, and it is a good idea to apply them whenever possible.

All of these laws can be proved by the truth table method.

With the exception of of the double-negation law, all laws come in pairs (called dual pairs).

The commutativity, associativity and distributivity laws have their equivalents in standard algebra.

$$A \wedge (B \vee C) \Leftrightarrows (A \wedge B) \vee (A \wedge C) \approx a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

We can derive further laws, for example, the absorption laws

$$\begin{aligned}A \vee (A \wedge B) &\Leftrightarrows A\\ A \wedge (A \vee B) &\Leftrightarrows A\end{aligned}$$

Law	Name
$A \vee \neg A \dashv\vdash 1$	Excluded Middle Law
$A \wedge \neg A \dashv\vdash 0$	Contradiction Law
$A \vee 0 \dashv\vdash A, A \wedge 1 \dashv\vdash A$	Identity Laws
$A \vee 1 \dashv\vdash 1, A \wedge 0 \dashv\vdash 0$	Domination Laws
$A \vee A \dashv\vdash A, A \wedge A \dashv\vdash A$	Idempotent Laws
$\neg(\neg A) \dashv\vdash A$	Double-Negation Law
$A \vee B \dashv\vdash B \vee A, A \wedge B \dashv\vdash B \wedge A$	Commutativity Laws
$(A \vee B) \vee C \dashv\vdash A \vee (B \vee C)$ $(A \wedge B) \wedge C \dashv\vdash A \wedge (B \wedge C)$	Associativity Laws
$A \vee (B \wedge C) \dashv\vdash (A \vee B) \wedge (A \vee C)$ $A \wedge (B \vee C) \dashv\vdash (A \wedge B) \vee (A \wedge C)$	Distributivity Laws
$\neg(A \wedge B) \dashv\vdash \neg A \vee \neg B$ $\neg(A \vee B) \dashv\vdash \neg A \wedge \neg B$	De Morgan's Laws

Another important law (and its dual):

$$(A \wedge B) \vee (\neg A \wedge B) \dashv\vdash B$$
$$(A \vee B) \wedge (\neg A \vee B) \dashv\vdash B$$

Definition: A formula is called literal if it is of the form p or $\neg p$, where p is a proposition symbol. The two formulas p and $\neg p$ are called complementary literals.

We can simplify conjunctions and disjunctions using certain rules.

If a conjunction contains complementary literals, it is a contradiction. If a disjunction contains complementary literals, or of it contains a 1, it is a tautology.

Example: Simplify the formula

$(p_3 \wedge \neg p_2 \wedge p_3 \wedge \neg p_1) \vee (p_1 \wedge p_3 \wedge \neg p_1)$

Solution: $\neg p_1 \wedge \neg p_2 \wedge p_3$

Normal Forms

Formulas can be transformed into standard forms so that they can become more convenient for symbolic manipulations and make identification and comparison of two formulas easier.

There are two types of normal forms in propositional calculus: the Disjunctive Normal Form and the Conjunctive Normal Form.

Definition: A disjunction with literals as disjuncts is called a disjunctive clause. A conjunction with literals as conjuncts is called a conjunctive clause.

Examples:

- $(p \vee q \vee \neg r)$ is a disjunctive clause
- $(\neg p \wedge s \wedge \neg q)$ is a conjunctive clause
- p or $\neg p$ is a (degenerate) disjunctive clause with one disjunct, and a (degenerate) conjunctive clause with one conjunct.

Disjunctive and conjunctive clauses are simply called clauses.

Definition: A disjunction with conjunctive clauses as its disjuncts is said to be in Disjunctive Normal Form (DNF). A conjunction with disjunctive clauses as its conjuncts is said to be in Conjunctive Normal Form (CNF).

Examples:

- $(p \wedge q) \vee (p \wedge \neg q), p \vee (q \wedge r)$, and $\neg p \vee t$ is in disjunctive normal form.
- The formula $\neg(p \wedge q) \vee r$ is not in disjunctive normal form.

- Each of $p \wedge (q \vee r) \wedge (\neg q \vee r)$ and $p \wedge q$ is in conjunctive normal form.
- The formula $p \wedge (r \vee (p \wedge q))$ is not in conjunctive normal form.

A formula in Disjunctive Normal Form (DNF) is of the form $(A_{11} \wedge \cdots \wedge A_{1n_1}) \vee \cdots \vee (A_{k1} \wedge \cdots \wedge A_{kn_k})$ where $k \geq 1, n_1, \ldots, n_k \geq 1$ and A_{ij} are literals for $1 \leq i \leq k$ and $1 \leq j \leq n_i$. The formulas $(A_{i1} \wedge \cdots \wedge A_{in_i})$ are the conjunctive clauses of the formula in DNF.

A formula in Conjunctive Normal Form (CNF) is of the form $(A_{11} \vee \cdot \vee A_{1n_1}) \wedge \cdot \wedge (A_{k1} \vee \cdot \vee A_{kn_k})$, where $k \geq 1, n_1, \ldots, n_k \geq 1$ and A_{ij} are literals for $1 \leq i \leq k$ and $1 \leq j \leq n_i$. The formulas $(A_{i1} \vee \cdots \vee A_{in_i}$ are the disjunctive clauses of the formula in CNF.

Examples

p is an atom, and therefore a literal.

It is a disjunction with only one disjunct.

It is also a conjunction with only one conjunct.

Hence it is a disjunctive or conjunctive clause with one literal.

It is a formula in disjunctive normal form with one conjunctive clause p.

It is also a formula in conjunctive normal form with one disjunctive clause p.

$\neg p \wedge q \wedge \neg r$ is a conjunction of three literals, and a formula in conjunctive normal form with three clauses. It is also a conjunctive clause, and a formula in disjunctive normal form, with one conjunctive clause, $(\neg p \wedge q \wedge \neg r)$.

$\neg p \wedge (q \vee \neg r) \wedge (\neg q \vee r)$ is a formula in conjunctive normal form with three disjunctive clauses, $\neg p, (q \vee \neg r), (\neg q \vee r)$. It is not a formula in disjunctive normal form.

How do we obtain normal forms?

Use the following tautological equivalences:

$$A \implies B \dashv\vdash \neg A \vee B \tag{1}$$
$$A \iff B \dashv\vdash (\neg A \vee B) \wedge (A \vee \neg B) \tag{2}$$
$$A \iff B \dashv\vdash (A \wedge B) \vee (\neg A \wedge \neg B) \tag{3}$$
$$\neg\neg A \dashv\vdash A \tag{4}$$
$$\neg(A_1 \wedge \cdots \wedge A_n) \dashv\vdash \neg A_1 \vee \cdots \vee \neg A_n \tag{5}$$
$$\neg(A_1 \vee \cdots \vee A_n) \dashv\vdash \neg A_1 \wedge \cdots \wedge \neg A_n \tag{6}$$
$$A \wedge (B_1 \vee \cdots \vee B_n) \dashv\vdash (A \wedge B_1) \vee \cdots \vee (A \wedge B_n) \tag{7}$$
$$A \vee (B_1 \wedge \cdots \wedge B_n) \dashv\vdash (A \vee B_1) \wedge \cdots \wedge (A \vee B_n) \tag{8}$$

By the Theorem of Replaceability of Tautologically Equivalent Formulas, we can use the equivalences above to convert any formula into a tautologically equivalent formula in normal form.

Example

Convert the following formula into a conjunctive normal form $\neg((p \vee \neg q) \wedge \neg r)$.

The conjunctive normal form can be found by the following derivations:

$$\begin{aligned}
\neg((p \vee \neg q) \wedge \neg r) &\dashv\vdash \neg(p \vee \neg q) \vee \neg\neg r && \text{De Morgan} \\
&\dashv\vdash \neg(p \vee \neg q) \vee r && \text{Double-Negation} \\
&\dashv\vdash \neg(p \vee \neg q) \vee r && \text{Double-negation} \\
&\dashv\vdash (\neg p \wedge \neg\neg q) \vee r && \text{De Morgan} \\
&\dashv\vdash (\neg p \wedge q) \vee r && \text{Double-negation} \\
&\dashv\vdash (\neg p \vee r) \wedge (q \vee r) && \text{Distributivity}
\end{aligned}$$

Algorithm for Conjunctive Normal Form

1. Eliminate equivalence and implication, using $A \implies B \dashv\vdash \neg A \vee B$ and $A \iff B \dashv\vdash (\neg A \vee B) \wedge (A \vee \neg B)$.
2. Use De Morgan and double-negation to obtain an equivalent formula, where each $\neg$ symbol has only an atom as its scope.
3. Recursive procedure $CNF(A)$:
   (a) If A is a literal then return A
   (b) If A is $B \wedge C$ then return $CNF(B) \wedge CNF(C)$
   (c) If A is $B \vee C$ then
   - Call $CNF(B)$ and $CNF(C)$
   - Suppose $CNF(B) = B_1 \wedge B_2 \wedge \cdots \wedge B_n$
   - Suppose $CNF(C) = C_1 \wedge C_2 \wedge \cdots \wedge C_m$
   - Return $\wedge_{i=1\ldots n, j=1\ldots m}(B_i \vee C_j)$
   - Note: The last step is similar to using distributivity to expand $(x_1 + x_2 + \cdots + x_n) \cdot (y_1 + y_2 + \cdots + y_m)$

Example of Step 3.3 in converting to CNF

$((a \vee b) \wedge (c \vee \neg a \vee d)) \vee ((\neg a) \wedge (c \vee d) \wedge (\neg b \vee \neg c \vee \neg d))$

- $n = 2$ clauses
- $m = 3$ clauses
- The resulting CNF will have $2 \times 3 = 6$ clauses
- It can be further simplified

$(a \vee b \vee \neg a) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee \neg b \vee \neg c \vee \neg d) \wedge$
$(c \vee \neg a \vee d \vee \neg a) \wedge (c \vee \neg a \vee d \vee c \vee d) \wedge$
$(c \vee \neg a \vee d \vee \neg b \vee \neg c \vee \neg d) \cdots \dashv\vdash (a \vee b \vee c \vee d) \wedge (\neg a \vee c \vee d)$

Existence of Normal Forms

Theorem: Any formula $A \in \text{Form}(\mathcal{L}^p)$ is tautologically equivalent to some formula in disjunctive normal form.

Proof

(i) If A is a contradiction, then A is tautologically equivalent to the DNF $p \wedge \neg p, p$ being any atom occurring in A.

(ii) If A is not a contradiction, we employ the following method (this is the idea of the proof worked out on an example, not the full proof).

Suppose A has three atoms, p, q, r occurring in A, and the value of A is 1 if and only if $1, 1, 0$, or $1, 0, 1$, or $0, 0, 1$, are assigned to p, q, r respectively.

For each of these truth valuations, we form a conjunctive clause with three literals, each being one of the atoms or its negation, according to whether this atom is assigned 1 or 0:

$(p \wedge q \wedge \neg r), (p \wedge \neg q \wedge r)$, and $(\neg p \wedge \neg q \wedge r)$

Due to the definition of the connective $\wedge$, we have that:

- $(p \wedge q \wedge \neg r)$ has value 1 $\iff$ $1, 1, 0$ are assigned to p, q, r
- $(p \wedge \neg q \wedge r)$ has value 1 $\iff$ $1, 0, 1$ are assigned to p, q, r
- $(\neg p \wedge \neg q \wedge r)$ has value 1 $\iff$ $0, 0, 1$ are assigned to p, q, r

Therefore, the following DNF is tautologically equivalent to A:

$$(p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge r)$$

Note: If A is a tautology, the required DNF may simply be $p \vee \neg p$ where p is any atom occurring in A.

Similarly, we have:

Theorem: Any formula $A \in \text{Form}(\mathcal{L}^p)$ is tautologically equivalent to some formula in conjunctive normal form.

Disjunctive Normal Forms from Truth Tables

Obtaining the DNF from truth tables is straight-forward.

p	q	r	f
1	1	1	1^*
1	1	0	0
1	0	1	1^*
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	1^*
0	0	0	0

$$f \dashv\vdash (p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge r)$$

Conjunctive Normal Form from Truth Tables

Duality can be used to obtain conjunctive normal forms from truth tables. Recall that, if A is a formula containing only the connectives $\neg, \vee$, and $\wedge$, then its dual, $\Delta(A)$, is formed by replacing all $\vee$ by $\wedge$, all $\wedge$ by $\vee$, and all atoms by their negations.

Example: The dual of the formula $A = (p \wedge q) \vee \neg r$ is $\Delta(A) = (\neg p \vee \neg q) \wedge \neg\neg r \dashv\vdash (\neg p \vee \neg q) \wedge r$.

Recall that by the Duality Theorem, $\Delta(A) \dashv\vdash \neg A$. Also note that, if a formula A is in DNF, then its dual can easily be transformed into an equivalent formula in CNF, using double-negation if necessary.

This idea can be used to find the conjunctive normal form from the truth table of a formula f.

CNF of f obtained from the truth table of f.

- Determine the disjunctive normal form of $\neg f$.
- If the resulting DNF formula is A, then $A \dashv\vdash \neg f$.
- Compute $\Delta(A) \dashv\vdash \neg A$, by the Duality Theorem.
- $\Delta(A) \dashv\vdash \neg A \dashv\vdash \neg(\neg f) \dashv\vdash f$.
- $\Delta(A)$ is in CNF, or can be changed into CNF by using $\neg\neg p \dashv\vdash p$.

Example

p	q	r	f_1	$\neg f_1$
1	1	1	1	0
1	1	0	1	0
1	0	1	0	1
1	0	0	0	1
0	1	1	1	0
0	1	0	1	0
0	0	1	0	1
0	0	0	1	0

The DNF of $\neg f_1$, based on the truth table for $\neg f_1$, is the formula:

$$A = (p \wedge \neg q \wedge r) \vee (p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \dashv\vdash \neg f_1$$

The CNF for f_1 is equivalent to the dual of formula A, namely

$$\Delta(A) \dashv\vdash \neg A \dashv\vdash (\neg p \vee q \vee \neg r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee \neg r) \dashv\vdash \neg(\neg f_1) \dashv\vdash f_1$$

5 Logic05: Adequate Set of Connectives, Logic Gates, Circuit Design, Code Simplification

Connectives

Formulas $A \implies B$ and $\neg A \vee B$ are tautologically equivalent. Then $\implies$ is said to be definable in terms of (or reducible to) $\neg$ and $\vee$. $\vee$ is definable in terms of $\neg$ and $\implies$, as $A \vee B \dashv\vdash \neg A \implies B$.

We have mentioned so far one unary, and four binary connectives.

There are many more unary and binary connectives, and also $n-$ary connectives, for $n > 2$.

We shall use letters f, g, etc., (with or without subscripts) to denote any connectives. We shall write

$$f(A_1, \ldots, A_n)$$

for the formula formed by an $n-$ary connective f connecting formulas $A_1, \ldots, A_n$.

A connective is defined by its truth table. Two $n-$ary connectives, $n \geq 1$ are the same if and only if they have the same truth tables.

How many distinct unary connectives are there?

p	$f_1(p)$	$f_2(p)$	$f_3(p)$	$f_4(p)$
1	1	1	0	0
0	1	0	1	0

Note that $f_3(p)$ is the negation of p, that is $\neg p$.

How many distinct $n-$ary connectives are there?

For an $n-$ary connective, the truth table has 2^n rows, and the number of possible distinct $n-$ary connectives equals the number of possible distinct columns of a truth table with 2^n rows, which in turn equals the number of possible binary numbers of length 2^n (length of binary number = height of truth table column). Thus the answer is $2^{(2^n)}$.

Adequate Set of Connectives

Definition: Any set of connectives with the capability to express any truth table is said to be adequate.

Emil Post observed in 1921 that the set of five standard connectives $\{\neg, \wedge, \vee, \implies, \iff\}$, is adequate.

Definition (equivalent): A set of S connectives is called adequate if and only if any $n-$ary connective can be defined in terms of the connectives in S.

Theorem: The set $S_0 = \{\neg, \wedge, \vee\}$ is an adequate set of connectives.

Proof: Let f be an arbitrary $n-$ary connective.

We want to find a formula A_{s_0}, using only connectives in $S_0 = \{\neg, \wedge, \vee\}$, such that $f(p_1, \ldots, p_n) \dashv\vdash A_{S_0}$.

- Construct the truth table for the connective $f(p_1, \ldots, p_n)$
- Use the theorem about the existence of Disjunctive Normal Forms to obtain a formula A_{S_0}, in DNF, with $f(p_1, \ldots, p_n) \dashv\vdash A_{S_0}$.
- By construction, A_{S_0} uses only connectives in $S_0 = \{\neg, \wedge, \vee\}$.
- We Can show that a new set $\mathcal{S}$ of connectives is adequate by showing that all connectives in $S_0 = \{\neg, \wedge, \vee\}$ (which we already proved adequate) are definable in terms of the new connectives in $\mathcal{S}$.
- More precisely, given any $n-$ary connective f, the previous theorem states that $f \dashv\vdash A_{S_0}$ is a formula that uses only connectives in $S_0 = \{\neg, \wedge, \vee\}$.
- If we can show that there exists a formula A_S, using only connectives in S, such that $A_{S_0} \dashv\vdash A_S$, then we have $f \dashv\vdash A_{S_0} \dashv\vdash A_S$.

- The existence of the formula A_S is proved by showing that each of the connectives in $S_0 = \{\neg, \wedge, \vee\}$ is definable in terms of the connectives $\mathcal{S}$, and by invoking the Replaceability Theorem.
- This proves the adequacy of $\mathcal{S}$.

Corollary: Show that $\{\neg, \wedge\}, \{\neg, \vee\}$, and $\{\neg, \implies\}$ are adequate.

Proof: We show that $S_1 = \{\neg, \wedge\}$ is an adequate set of connectives. By the previous theorem, for any $n-$ary connective f there exists a formula A_{S_0}, using only connectives in $S_0 = \{\neg, \wedge, \vee\}$ with $f \Leftrightarrows A_{S_0}$.

- Consider the formula $A_{S_0} \Leftrightarrows f$, using the three connectives in S_0.
- Goal: A formula equivalent to A_{S_0}, using only connectives in S_1.
  - $\neg$ in A_{S_0} is also a connective in S_1, no change needed to A_{S_0}.
  - $\wedge$ in A_{S_0} is also a connective in S_1, no change needed to A_{S_0}.
  - $\vee$ in A_{S_0} is not a connective in S_1. Remove all occurrences of $\vee$ in A_{S_0}, by using the equivalence $B \vee C \Leftrightarrows \neg(\neg B \wedge \neg C)$. The resulting formula, A_{S_1}, uses only connectives in S_1.
- By the Replaceability Theorem, $A_{S_1} \Leftrightarrows A_{S_0}$
- Thus, A_{S_1} contains only connectives in S_1, and we have that $f \Leftrightarrows A_{S_0} \Leftrightarrows A_{S_1}$.

Peirce Arrow

The binary connective $g_{15}(p, q)$, is also called Peirce arrow, (after C.S. Peirce, 1839-1914), or NOR, and denoted by $\downarrow$, is defined as:

p	q	$p \downarrow q$
1	1	0
1	0	0
0	1	0
0	0	1

Proof that Peirce arrow is adequate.

Since we showed that the set $S_0 = \{\neg, \wedge, \vee\}$ is adequate, to show that $S = \{\downarrow\}$ is adequate it suffices to prove that one can define each of the three connectives in S_0 in terms of the Peirce $\downarrow$, as follows:

$$\neg p \Leftrightarrows p \downarrow p$$
$$p \wedge q \Leftrightarrows (p \downarrow p) \downarrow (q \downarrow q)$$
$$p \vee q \Leftrightarrows (p \downarrow q) \downarrow (p \downarrow q)$$

Thus it follows that the set $S = \{\downarrow\}$, consisting of a single binary connective NOR, is adequate.

Note: To express a standard connective in terms of new connectives, we can write the truth table of the standard connective, and try writing formulas using various combinations of the new connectives, until we find a formula that gives the same truth values as the standard connective.

Proof that Sheffer stroke is adequate.

The binary connective $g_5(p, q)$, also called Sheffer stroke, "|", (after H.M Sheffer, 1882-1964), or NAND, is defined by:

p	q	$p \mid q$
1	1	0
1	0	1
0	1	1
0	0	1

One can express the standard connectives in S_0 in terms of "|", by:

$$\neg p \Leftrightarrow p \mid p$$
$$p \wedge q \Leftrightarrow (p \mid q) \mid (p \mid q)$$
$$p \vee q \Leftrightarrow (p \mid p) \mid (q \mid q)$$

Thus, the set $S' = \{|\}$ consisting of a single connective, NAND, is also adequate.

Proving Inadequacy

How do we show that a set $\mathcal{S}$ of connectives is not adequate?

We show that one of the connectives in the adequate set $S_0 = \{\neg, \vee, \wedge\}$ cannot be defined by using the connectives in $\mathcal{S}$.

Example

Prove that the set $\mathcal{S} = \{\wedge\}$ is not adequate.

Proof:

Claim: A formula depending on only one atom p, and using only the connective $\wedge$, has the property that its truth value under a truth valuation t with $p^t = 0$ is always 0 (proof by induction).

Assume now that $\mathcal{S} = \{\wedge\}$ were adequate. This implies that we could define the negation $\neg p$ in terms of $\wedge$, which implies that we could find a formula $A_\wedge(p)$ depending only on p, and using only the connective $\wedge$, such that $\neg p \Leftrightarrow A_\wedge(p)$.

However, due to the Claim, for a truth valuation t such that $p^t = 0$, we have that $(A_\wedge(p))^t = 0$. This implies that $\neg p$ and $A_\wedge(p)$ cannot be tautologically equivalent (since $(\neg p)^t = 1$) - a contradiction.

Let us use the symbol τ for the ternary connective whose truth table is given by

p	q	r	$\tau(p,q,r)$
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	0

Note that, for any truth valuation t, we have $\tau(p,q,r)^t$ equals q^t if $p^t = 1$, and equals r^t if $p^t = 0$.

This is the familiar if-then-else connective from computer science, namely if p then q else r.

This is one of the $2^{(2^3)} = 256$ distinct ternary connectives.

George Boole, author of "An Investigation of the Laws of Thoughts", has been fundamental in the development of digital electronics.

Definition: A Boolean algebra is a set B, together with two binary operations $+$ and $\cdot$, and a unary operation $^-$. The set B contains elements 0 and 1, is closed under the application of $+, \cdot$ and $^-$, and the following properties hold for all x, y, z in B.

- Identity Laws: $x + 0 = x$ and $x \cdot 1 = x$
- Complement Laws: $x + \overline{x} = 1, x \cdot \overline{x} = 0$
- Associativity Laws: $(x + y) + z = x + (y + z), (x \cdot y) \cdot z = x \cdot (y \cdot z)$.
- Commutativity Laws: $x + y = y + x, x \cdot y = y \cdot x$.
- Distributivity Laws: $x + (y \cdot z) = (x + y) \cdot (x + z)$ and $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.

The set of formulas in Form($\mathcal{L}^p$), with the $\vee$ and $\wedge$ operators, the $\neg$ operator, 0 and 1, and where $=$ is $\dashv\vdash$, is a Boolean algebra.

The set of subsets of a universal set U, with the union operator $\cup$, the intersection operator $\cap$, the set complementation operator $^{\complement}$, the empty set $\emptyset$, and the universal set U, is a Boolean algebra.

Note that, using the laws given in the definition of a Boolean algebra, it is possible to prove many other laws that hold for every Boolean algebra.

Thus to establish results about propositional logic, or about sets, we need only prove results about Boolean algebra.

Logical Equivalences	Set Properties
$\neg(\neg p) \dashv\vdash p$	$(A^{\complement})^{\complement} = A$
$p \vee p \dashv\vdash p, p \wedge p \dashv\vdash p$	$A \cup A = A, A \cap A = A$
$p \vee 0 \dashv\vdash p, p \wedge 1 \dashv\vdash p$	$A \cup \emptyset = A, A \cap U = A$
$p \wedge 0 \dashv\vdash 0, p \vee 1 \dashv\vdash 1$	$A \cap \emptyset = \emptyset, A \cup U = U$
$p \vee \neg p \dashv\vdash 1, p \wedge \neg p \dashv\vdash 0$	$A \cup A^{\complement} = U, A \cap A^{\complement} = \emptyset$
$\neg(p \wedge q) \dashv\vdash (\neg p \vee \neg q)$	$(A \cap B)^{\complement} = (A^{\complement} \cup B^{\complement})$
$\neg(p \vee q) \dashv\vdash (\neg p \wedge \neg q)$	$(A \cup B)^{\complement} = A^{\complement} \cap B^{\complement}$

Boolean algebra is used to model the circuitry of electronic devices, including electronic computers.

Such a device has inputs and outputs from the set $\{0, 1\}$.

A Boolean variable is a variable that can take values in the set $\{0, 1\}$ (1/true and 0/false are also called Boolean constants).

An $n-$variable Boolean function is a function $f : \{0, 1\}^n \to \{0, 1\}$

An electronic computer is made up of a number of circuits, each of which implements a Boolean function.

The basic elements of circuits are called logic gates, and they implement the three Boolean operators $+, \cdot, \overline{}$.

A logic gate is an electronic device that operates on a collection of binary digits (bits, in $\{0, 1\}$) and produces on binary output.

Each circuit can be designed using the laws of Boolean algebra.

Logic gates are physically implemented by transistors.

A transistor is simply a switch, it can be in an off state, which does not allow electricity to flow, or in an on state, in which electricity can pass unimpeded.

Each transistor contains three lines: two inputs lines and one output line. The first input line, called the control line, is used to open or close the switch inside the transistor.

ON states is used to represent the binary 1, and the OFF state can be used to represent the binary 0.

This solid-state switching device, the transistor, forms the basis of construction of virtually all computers built today, and it is thus the fundamental building block for all high-level computers.

However, there is no theoretical reason why we must use transistors as our elementary devices when designing computer systems.

In fact, binary computers can be built out of any bistable device.

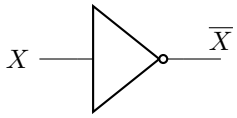
In principle, it is possible to construct a binary computer using any bistable device that meets the following four conditions:

- It has two stable energy states.
- These two states are separated by a large energy barrier.
- It is possible to sense what state the device is in without permanently destroying the stored value.
- It is possible to switch from a 0 to a 1 and viceversa by applying a sufficient amount of energy.

Basic Logic Gates

NOT

An inverter, or a NOT gate, is a logic gate that implements negation ($\neg$). It accepts the value of a Boolean variable as input, and produces the negation of its value as its output.



NOR

To construct the negation of OR, we use two transistors connected in parallel.

If either or both of the liens Input-1 and Input-2 are set to 1, then the corresponding transistor is in the ON state, and the output is connected to the ground, producing an output value of 0.

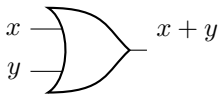
Only if both input lines are 0, effectively shutting off both transistors, will the output line contain a 1.

This is the definition of the negation of OR, and this gate is called NOR gate.

OR

The OR gate can be implemented using a NOR gate and a NOT gate.

The inputs to this gate are the values of two Boolean variables. The output is the Boolean sum $+$ (denoting $\vee$) of their values.

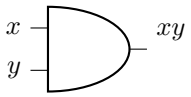


NAND

Negation of AND (which we already know).

AND

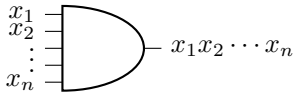
AND gate can be implemented using a NAND gate and a NOT gate.

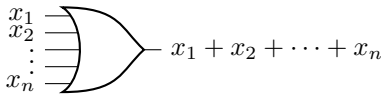


In circuit design, we use the following notations:

- $x + y$ denotes $x \vee y$
- $x \cdot y$ and xy both denote $x \wedge y$
- $\overline{x}$ denotes $\neg x$
- $=$ denotes tautological equivalence $\Eqcirc$

We sometime permit multiple inputs to AND gates (top) and OR gates (bottom), as illustrated below.





Non-standard gates: Toffoli gate

It has a 3-bit input and 3-bit output: If the first two bits are both 1, it inverts the 3rd bit, otherwise all bits stay the same.

Toffoli gates and Quantum Computing

x_1	x_2	x_3	y_1	y_2	y_3
1	1	1	1	1	0
1	1	0	1	1	1
1	0	1	1	0	1
1	0	0	1	0	0
0	1	1	0	1	1
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	0	0	0

- The Toffoli gate is a universal, reversible logic gate. It is:
- (1) Universal: All truth tables are implementable by Toffoli gates
- (2) Reversible: Given the output, we can uniquely reconstruct the input (e.g., $\neg$ is reversible, but $\wedge$ is not)
- The Toffoli gate can be realized by five 2-qubit quantum gates.
- This implies that a quantum computer using Toffoli gates can implement all possible classical computations.
- A quantum-mechanics-based Toffoli gate has been successfully realized in January 2009 at the University of Innsbruck, Austria.

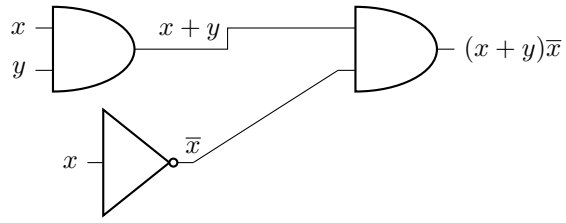
Combinational circuits

- Combinational logic circuits (sometimes called combinatorial circuits) are memoryless digital logic circuits whose output is a function of the present value of the inputs only.
- A combinational circuit is implemented as a combination of NOT gates, OR gates, and AND gates. In general such a circuit has n inputs and m outputs in $\{0, 1\}$.
- In contrast, sequential logic circuits - not described in this course - are basically combinational circuits with the additional properties of storage (to remember past inputs) and feedback.

Example:

Design a circuit that produces the following output.

(1) $(x + y)\overline{x}$



Design a circuit that accomplishes a task

Example 1: A committee of three individuals decides issues for an organization. Each individual votes either "yes" or "no" for each proposal that arises. A proposal is passed if and only if it receives at least two "yes" votes. Design a circuit that determines whether a proposal passes.

Solution: Let $x = 1$ if the first individual votes "yes", and $x = 0$ if this individual votes "no", and similarly for y and z.

Then a circuit must be designed that produces output 1 (proposal passes) from the inputs x, y, z if and only if two or more of x, y, z are 1.

Note that a Boolean function that has these output values is $f(x, y, z) = xy + xz + yz$.

Example 2: Sometimes light fixtures are controlled by more than one switch. Circuits need to be designed so that flipping any one of the switches for the fixture turns the light on when it is off, and turns the light off when it is on. Design a circuit that accomplishes this task, when there are three switches.

x	y	z	$F(x,y,z)$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

Solution: The inputs are three Boolean variables x, y, z, one for each switch. Let $x = 1$ if the first switch is closed, and $x = 0$ if it is open, and similarly for y and z.

The output function is $F(x, y, z)$ defined as $F(x, y, z) = 1$ if the light is on, and $F(x, y, z) = 0$ if the light is off.

We can choose to specify that the light be on when all three switches are closed, so that $F(1, 1, 1) = 1$.

This determines all the other values of F.

The formula in DNF corresponding to this truth table is

$$F(x, y, z) \Leftrightarrow xyz + x\overline{y}\overline{z} + \overline{x}y\overline{z} + \overline{x}\overline{y}z$$

Adders:

- Logic circuits can be used to carry out addition of two positive integers from their binary expansions.
- Recall that, e.g., the binary (base 2) expansion/representation of the integer 2 is $(10)_2$, of 8 is $(1000)_2$, of 9 is $(1001)_2$, etc.
- We will build up the circuitry to do addition of two positive integers in binary representation, from some component circuits.
- First we build a circuit that can be used to find $x + y$ when x and y are each a single bit (0 or 1).
- The input to our circuit will be two bits, x and y.
- The output will consist of two bits, namely s and c, where s is the sum bit and c is the carry bit.
- This circuit is a multiple output circuit.
- It has two input bits x, y, and it adds them up (in binary), producing two outputs: s (the sum bit) and c (the carry bit).
- The circuit we are designing is called the half-adder since it adds two its, without considering a carry from the previous addition

x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

- From the truth table we see that $c = xy$ and $s = x\overline{y} + \overline{x}y$
- If we use this fact that $x\overline{y} + \overline{x}y \Leftrightarrow (x + y)\overline{(xy)}$ we obtain a circuit with fewer gates (4 instead of 6).

The full-adder is used to add two numbers $(x_n x_{n-1} \dots x_0)_2$, and $(y_n y_{n-1} \dots y_0)_2$, in their binary representation, $x_i, y_i \in \{0, 1\}$ for all $0 \leq i \leq n$.

$$\begin{array}{ccccccc} x_n & x_{n-1} & \dots & x & \dots & x_0 & + \\ y_n & y_{n-1} & \dots & y & \dots & y_0 & \\ & & \dots & s & \dots & & \end{array}$$

The addition proceeds from right to left. To add x_0 to y_0 one uses a half-adder. Subsequently, at each step, a full-adder takes three bits as input (x, y, and the carry bit c_i from the previous addition), and it adds them up (in binary) producing two outputs, the sum bit s, and the next carry bit c_{i+1} (not shown in figure).

Truth table for the full-adder:

Input: Bits x and y and the carry bit c_i.

Output: The sum bit s and the carry bit c_{i+1}

Input			Output	
x	y	c_i	s	c_{i+1}
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

Formulas for outputs of the full-adder

From the truth table we obtain the following formulas in DNF, equivalent to s and c_{i+1}:

$$s = xyc_i + x\overline{yc_i} + \overline{x}y\overline{c_i} + \overline{xy}c_i$$
$$c_{i+1} = xyc_i + xy\overline{c_1} + x\overline{y}c_i + \overline{x}yc_i$$

Circuit Minimization Through Formula Simplification

Consider the circuit that has output 1 if and only if $x = y = z = 1$ or $x = z = 1$ and $y = 0$.

The formula corresponding to its truth table is $xyz + x\overline{y}z$. Simplify: $xyz + x\overline{y}z = (y + \overline{y})(xz) = 1 \cdot (xz) = xz$.

xz is a Boolean expression with fewer operators that represents the circuit, thus the corresponding simplified circuit will have fewer logic gates.

Thus, one can use the essential laws for propositional logic to minimize circuits.

Analyzing and simplifying code through logic formula simplification.

Consider the code fragment:

```
if (C1 ∨ ¬C2) then
|  if (¬(C2 ∧ C3)) then
|  |  P1
|  else
|  |  if (C2 ∧ ¬C3) then
|  |  |  P2
|  |  else
|  |  |  P3
|  |  end
|  end
else
|  P4
end
```

where C_1, C_2, C_3 are true/false conditions (formulas in propositional logic), and P_1, P_2, P_3, P_4 are sub-fragments of code.

We will prove that P_2 is dead code without a truth table.

Dead code is code that is never executed. The condition for P_2 to be executed is

$$(C_1 \vee \neg C_2) \wedge \neg\neg(C_2 \wedge C_3) \wedge (C_2 \wedge \neg C_3) \Leftrightarrow (C_1 \vee \neg C_2) \wedge C_2 \wedge C_3 \wedge C_2 \wedge \neg C_3) \Leftrightarrow 0$$

Since this condition can never be true, this means that P_2 can never be executed. Thus it is dead code.

We can simplify this code (via truth tables or inspection) to get

if *($C_1 \wedge C_2 \wedge C_3$)* **then**
| P_3
else
| **if** $\neg C_1 \wedge C_2$ **then**
| | P_4
| **else**
| | P_1
| **end**
end

6 Logic06: Formal Deduction in Propositional Logic

Formal Deducibility

We have seen how to prove arguments valid by using truth tables and other semantic methods (tautological consequence, "$\vDash$").

We now want to replace this approach by a purely syntactic one, that is, we give formal rules for deduction which are purely syntactic.

We want to define a relation called formal deducibility (denoted by "$\vdash$") that will allow us to mechanically/syntactically check the correctness of a proof that an argument is valid.

The intuitive meaning of "$\vdash$" is similar to the meaning of "$\vDash$", in that it signifies argument validity. However, the method of proving validity is different.

The word "formal" signifies that we will be concerned only with the syntactic form of formulas. The proofs themselves will not refer to any semantic properties. The correctness of the proof can be checked mechanically.

Formal deducibility is a relation between a set of formulas Σ (called the premises) and a formula A (called the conclusion).

We use the symbol "$\vdash$" to denote the relation of formal deducibility and write

$$\Sigma \vdash A$$

to mean that A is formally deducible (or provable) from Σ. Note that $\vDash$ is semantics; and $\vdash$ is syntactic.

For convenience, we will write sets as sequences.

- If $\Sigma = \{A_1, A_2, A_3, \ldots\}$ is a set of formulas, then Σ may be written as a sequence, $A_1, A_2, \ldots, A_n$.
- Since the premises are elements of a set, the order in which premises in Σ are written does not matter.
- The set $\Sigma \cup \{A\}$, where A is a formula, may be written as Σ, A.
- If Σ and Σ' are sets of formulas, $\Sigma \cup \Sigma'$ may be written as Σ, Σ'.

For any formulas, A, B, and C, and any set Σ of formulas:

Note: Each of the above rules is really a template, or scheme, for infinitely many rules. Each of A, B, C may be any formula; Σ may be any set of formulas.

We can use the 11 rules to prove new theorems.

(1)	(Ref)	$A \vdash A$ is a theorem	(Reflexivity)
(2)	(+)	If $\Sigma \vdash A$ is a theorem then $\Sigma, \Sigma' \vdash A$ is a theorem.	(Addition of premises)
(3)	$(\neg -)$	If $\Sigma, \neg A \vdash B$ is a theorem and $\Sigma, \neg A \vdash \neg B$ is a theorem then $\Sigma \vdash A$ is a theorem.	($\neg$ elimination)
(4)	$(\implies -)$	If $\Sigma \vdash A \implies B$ is a theorem and $\Sigma \vdash A$ is a theorem then $\Sigma \vdash B$ is a theorem.	($\implies$ elimination)
(5)	$(\implies +)$	If $\Sigma, A \vdash B$ is a theorem then $\Sigma \vdash A \implies B$ is a theorem.	($\implies$ introduction)
(6)	$(\wedge -)$	If $\Sigma \vdash A \wedge B$ is a theorem then $\Sigma \vdash A$ is a theorem and $\Sigma \vdash B$ is a theorem.	($\wedge$ elimination)
(7)	$(\wedge +)$	If $\Sigma \vdash A$ is a theorem and $\Sigma \vdash B$ is a theorem then $\Sigma \vdash A \wedge B$ is a theorem.	($\wedge$ introduction)
(8)	$(\vee -)$	If $\Sigma, A \vdash C$ is a theorem and $\Sigma, B \vdash C$ is a theorem then $\Sigma, A \vee B \vdash C$ is a theorem.	($\vee$ elimination)
(9)	$(\vee +)$	If $\Sigma \vdash A$ is a theorem then $\Sigma \vdash A \vee B$ is a theorem and $\Sigma \vdash B \vee A$ is a theorem	($\vee$ introduction)
(10)	$(\leftrightarrow -)$	If $\Sigma \vdash A \leftrightarrow B$ is a theorem and $\Sigma \vdash A$ is a theorem, then $\Sigma \vdash B$ is a theorem.	($\leftrightarrow$ elimination)
		If $\Sigma \vdash A \leftrightarrow B$ is a theorem and $\Sigma \vdash B$ is a theorem then $\Sigma \vdash A$ is a theorem.	
(11)	$(\leftrightarrow +)$	If $\Sigma, A \vdash B$ is a theorem and $\Sigma, B \vdash A$ is a theorem then $\Sigma \vdash A \leftrightarrow B$ is a theorem.	($\leftrightarrow$ introduction)

Example 1: Prove the following theorem, called "membership rule":

$$(\in) \quad \text{If } A \in \Sigma \text{ then } \Sigma \vdash A.$$

Proof: Suppose $A \in \Sigma$ and $\Sigma' = \Sigma - \{A\}$ (thus, Σ is A, Σ').

(1)	$A \vdash A$	(by (Ref))
(2)	$A, \Sigma' \vdash A$	(by $(+), (1)$)

- Step (1) is generated directly by the rule (Ref).
- Step (2) is generated by the rule (+), which is applied to Step (1).
- At each step, the rule applied, and the preceding steps cited (if any), form a justification for this step, and are written on the right.
- These steps constitute a formal proof of the last step, $\Sigma \vdash A$.
- Having been formally proven, $(\in)$ is now a theorem.

Hypothetical Syllogism by Formal Deduction

Example 2: Prove that $A \implies B, B \implies C \vdash A \implies C$

The following sequence of 6 steps is a proof.

(1)	$A \implies B, B \implies C, A$	$\vdash A \implies B$	(by $(\in)$)
(2)	$A \implies B, B \implies C, A$	$\vdash A$	(by $(\in)$)
(3)	$A \implies B, B \implies C, A$	$\vdash B$	(by $(\implies -), (1), (2)$)
(4)	$A \implies B, B \implies C, A$	$\vdash B \implies C$	(by $(\in)$)
(5)	$A \implies B, B \implies C, A$	$\vdash C$	(by $(\implies -), (4), (3)$)
(6)	$A \implies B, B \implies C$	$\vdash A \implies C$	(by $(\implies +), (5)$)

Each step applies either one of the rules of formal deduction, or a theorem which we have already proved, e.g., $(\in)$.

On the right are written justifications for the steps.

These six steps form a formal proof of $A \implies B, B \implies C \vdash A \implies C$, which is generated at the last step.

The formal rules of deduction do not specify the use of "proved theorems". Why is this legitimate?

Instead of invoking a proved theorem, we could insert its proof.

For example, in the previous proof, instead of Step (1)

$$(1) \quad A \implies B, B \implies C, A \vdash A \implies B \quad (\text{by } (\in)$$

we could write an instance of the proof of $(\in)$:

$$(1a) \quad A \implies B \vdash A \implies B \quad (\text{by (Ref)})$$
$$(1b) \quad A \implies B, B \implies C, A \vdash A \implies B \quad (\text{by } (+), (1a))$$

A demonstrated $\Sigma \vdash A$ (that is, for which we have a formal proof) is called a scheme of formal deducibility, or a theorem.

Rules of formal deduction are purely syntactic. For instance, from two (not necessarily consecutive) "lines" in a proof

$$(i) \quad \Sigma, \neg A \vdash B$$
$$(ii) \quad \Sigma, \neg A \vdash \neg B$$

we can generate the new line $(iii) \Sigma \vdash A$, by applying $(\neg -)$.

Therefore it can be checked mechanically whether the rules of formal deduction are used correctly.

Intuitive meaning of rules:

$(\neg -)$ expresses the method of proof by contradiction. Say we want to prove theorem $\Sigma \vdash A$, that is, prove that from the set of premises Σ we can formally deduce the conclusion A.

A "proof by contradiction" would start by assuming that the conclusion does not hold. Formally, this amounts to adding its negation, $\neg A$, to the set of premises.

If the premises in Σ together with this new assumption lead to a contradiction (two formulas B and $\neg B$), that is, if we prove that $\Sigma, \neg A \vdash B$ and $\Sigma, \neg A \vdash \neg B$, then we can conclude that our assumption was wrong, and that the proposition A is deducible from the premises, that is, $\Sigma \vdash A$.

$(\implies +)$ expresses that to prove "If A then B" from certain premises Σ, that is, if we want to prove $\Sigma \vdash A \implies B$, it is sufficient to prove B from the premises together with A (that is, it suffices to prove $\Sigma, A \vdash B$).

In other words, if the conclusion is an implication, $A \implies B$, then the antecedent of the implication, A, can be considered to be an additional premise that we can use to prove B (both Σ and A are assumptions that we make when trying to prove B).

If, together with this additional premise, we can prove the consequent of the implication (that is, if we can prove $\Sigma, A \vdash B$), then we can conclude that $\Sigma \vdash A \implies B$.

Essentially, ($\implies$ +) states that an assumption (premise) may be converted into the antecedent of a conditional.

Definition of Formal Deducibility ($\vdash$)

A formal deduction system is specified by a set of deduction rules.

A formula A is formally deducible from Σ, written as $\Sigma \vdash A$, iff $\Sigma \vdash A$ is generated by (a finite number of applications of) the rules of formal deduction.

By the above definition, $\Sigma \vdash A$ holds iff there is a finite sequence

$$\begin{aligned} (1) \quad & \Sigma_1 \vdash A_1 \\ & \dots \\ (n) \quad & \Sigma_n \vdash A_n \end{aligned}$$

such that each term $\Sigma_k \vdash A_k (k = 1, \dots, n)$ is generated by one rule of formal deduction, and $\Sigma_n \vdash A_n$ is $\Sigma \vdash A$ (that is, $\Sigma_n = \Sigma$ and $A_n = A$).

To check whether a sequence of steps is indeed a formal proof of a "scheme of formal deducibility" (theorem), we:

- Check whether the rules of formal deduction are correctly applied at each step, and
- Check whether the last term of the formal proof is identical with the desired scheme of formal deducibility (theorem).

In this sense, rules of formal deduction and formal proofs serve to clarify the concepts of inference and proofs from informal reasoning.

The sequence of rules generating $\Sigma \vdash A$ is called a formal proof.

A scheme of formal deducibility may have various formal proofs. Perhaps one may not know how to construct a formal proof for it.

It is significant however that any proposed formal proof for a theorem can be checked mechanically to decide whether it is indeed a formal proof of this theorem.

How do we find a proof?

A useful idea is to work in reverse.

If $A \implies B, B \implies C \vdash A \implies C$ is what we want to prove (hence the last line of its proof), what rule of formal deduction could produce this line, from previous lines?

The rule ($\implies$ +) provides a way to produce an implication such as "$A \implies C$".

Recall ($\implies$ +): If $\Sigma, A \vdash B$ then $\Sigma \vdash A \implies B$. That is, to produce an implication in the conclusion, $\Sigma A \implies B$, we can first prove $\Sigma, A \vdash B$, and then apply ($\implies$ +) to it.

Here, take B to be C, and Σ to be $\{A \implies B, A \implies C\}$.

Thus, if we could prove $A \implies B, B \implies C, A \vdash C$, as the 2nd last step of the proof, then one application of ($\implies$ +), would finish the proof.

Tautological Consequence vs Deducibility

Tautological consequence ($\Sigma \vDash A$) and formal deducibility ($\Sigma \vdash A$) are different matters. Former belongs to semantics, latter belongs to syntax.

The connection between $\vDash$ and $\implies$ is that $A \vDash B \iff A \implies B$ is a tautology.

The connection between $\vdash$ and $\implies$ is that $A \vdash B \iff \emptyset \vdash A \implies B$.

The definition of formal deducibility is a recursive definition of the set of the proved schemes of formal deducibility (theorems):

- Rule (REF) is the BASE (similar to atoms being formulas in the recursive definition of $\mathrm{Form}(\mathcal{L}^p)$);
- The other ten rules of formal deduction are the RECURSION (similar to the five formation rules for formulas).

Statements concerning formal deducibility can be proved by structural induction on its structure (of generation).

The BASE CASE of structural induction is to prove that $A \vdash A$, generated directly by rule (Ref), has a certain property.

The (COMPOSITE) Inductive Step is to prove that the other ten rules preserve the property.

Theorem: Finiteness of premise set

If $\Sigma \vdash A$, then there exists a finite $\Sigma^0 \subseteq \Sigma$ such that $\Sigma^0 \vdash A$.

Proof: By induction on the structure of $\Sigma \vdash A$.

Base Case: The set of premises in $A \vdash A$, generated by (Ref), is a set of cardinality one, hence finite.

Inductive Step: We distinguish ten cases. For each case, assume that the cited theorems have the property, and prove that the derived theorem has the property.

Case of $(\implies -)$: "If $\Sigma \vdash A \implies B$, and $\Sigma \vdash A$, then $\Sigma \vdash B$". By the Inductive Hypothesis, the cited theorems have the property, that is, there exist finite sets $\Sigma_1, \Sigma_2 \subseteq \Sigma$ such that $\Sigma_1 \vdash A \implies B$ and $\Sigma_2 \vdash A$. By $(+)$ we have $\Sigma_1, \Sigma_2 \vdash A \implies B$, as well as $\Sigma_1, \Sigma_2 \vdash A$.

Then by $(\implies -)$, we have $\Sigma_1, \Sigma_2 \vdash B$, where $\Sigma_1 \cup \Sigma_2$ is a finite subset of Σ.

Theorem: Transitivity of Deducibility

Let $\Sigma, \Sigma' \subseteq \mathrm{Form}(\mathcal{L}^p)$. If $\Sigma \vdash \Sigma'$ and $\Sigma' \vdash A$, then $\Sigma \vdash A$.

Proof:

(1)	$A_1, \ldots, A_n \vdash A$	$A_i \in \Sigma', (\text{Th.Fin.Prem.})$
(2)	$A_1, \ldots, A_{n-1} \vdash A_n \implies A$	$(\implies +), (1)$
$\vdots$		
$(n+1)$	$\emptyset \vdash A_1 \implies (\ldots (A_n \implies A) \ldots)$	$(\implies +), (n)$
$(n+2)$	$\Sigma \vdash A_1 \implies (\ldots (A_n \implies A) \ldots)$	$(+), (n+1)$
$(n+3)$	$\Sigma \vdash A_1$	given
$(n+4)$	$\Sigma \vdash A_2 \implies (\ldots (A_n \implies A) \ldots)$	$(\implies -), (n+2), (n+3)$
$\vdots$		
$(3n+1)$	$\Sigma \vdash A_n \implies$	$(\implies -), (3n), (3n-1)$
$(3n+2)$	$\Sigma \vdash A_n$	given
$(3n+3)$	$\Sigma \vdash A$	$(\implies -), (3n+1), (3n+2)$

A useful theorem: Double-negation

Theorem: $\neg\neg A \vdash A$.

Proof:

(1)	$\neg\neg A, \neg A \vdash \neg A$	by $(\in)$
(2)	$\neg\neg A, \neg A \vdash \neg\neg A$	by $(\in)$
(3)	$\neg\neg A \vdash A$	by $(\neg -), (1), (2)$.

Note: When applying $(\neg -)$ in step (3), we take:

- $\Sigma \coloneqq \{\neg\neg A\}$
- $A \coloneqq A$
- $B \coloneqq \neg A$

Theorem: Reductio ad absurdum, $(\neg+)$

If $\Sigma, A \vdash B$ and $\Sigma, A \vdash \neg B$, then $\Sigma \vdash \neg A$.

Proof: We will only prove the theorem for the case when Σ is finite.

(1)	$\Sigma, A \vdash B$	given
(2)	$\Sigma, \neg\neg A \vdash \Sigma$	by $(\in)$
(3)	$\neg\neg A, \vdash A$	by the previous proved theorem
(4)	$\Sigma, \neg\neg A \vdash A$	by $(+), (3)$
(5)	$\Sigma, \neg\neg A \vdash B$	by (Tr) $, (2), (4), (1)$
(6)	$\Sigma, \neg\neg A \vdash \neg B$	analogous to the proof for (5)
(7)	$\Sigma \vdash \neg A$	by $(\neg-), (5), (6)$

In case Σ is infinite, the proof is similar, but one has to invoke the Finiteness of Premise Set theorem, similar to the way it is done in the proof of (Tr).

The theorem of reductio ad absurdum is denoted by $(\neg+)$. $(\neg+)$ and $(\neg-)$ both formalize the idea of "proof by contradiction", and are similar in shape but different in strength.

$(\neg-)$ is stronger than $(\neg+)$ in the following sense.

Definition: For two formulas A and B we write

$$A \dashv\vdash B$$

to mean $A \vdash B$ and $B \vdash A$.

A and B are said to be syntactically equivalent iff $A \dashv\vdash B$ holds.

We write $\dashv$ to denote the converse of $\vdash$.

Lemma. If $A \dashv\vdash A'$ and $B \dashv\vdash B'$ then

1. $\neg \dashv\vdash \neg A'$
2. $A \wedge B \dashv\vdash A' \wedge B'$
3. $A \vee B \dashv\vdash A' \vee B'$
4. $A \implies B \dashv\vdash A' \implies B'$
5. $A \iff B \dashv\vdash A' \iff B'$

Note the resemblance to analogous results about tautological equivalences $\dashv\vdash$, which are semantic.

Theorem: Replaceability of syntactically equivalent formulas (Repl).

Let $B \dashv\vdash C$. For any A, let A' be constructed from A by replacing some (not necessarily all) occurrences of B by C. Then $A \dashv\vdash A'$.

Theorem: $A_1, A_2, \ldots, A_n \vdash A \iff \emptyset \vdash A_1 \wedge \ldots \wedge A_n \implies A$

Theorem: $A_1, \ldots, A_n \vdash A \iff \emptyset A_1 \implies (\ldots (A_n \implies A) \ldots)$.

When the set of premises is empty we have the special case $\emptyset \vdash A$ of formal deducibility.

Obviously, $\emptyset \vdash A \iff \Sigma \vdash A$ for any Σ.

It has been mentioned before that A is said to be formally provable from Σ when $\Sigma \vdash A$ holds.

Definition: If $\emptyset \vdash A$ holds, then formula A is called formally provable.

The laws of non-contradiction $\neg(A \wedge \neg A)$ and excluded middle $A \vee \neg A$ are instances of formally provable formulas, that is, $\emptyset \vdash \neg(A \wedge \neg A)$ and $\emptyset \vdash A \vee \neg A$.

Why do we need formal deduction?

One of the things that sets mathematics/computer science apart from poetry, biology, engineer, etc., is the insistence upon proof.

Our goal with tautological consequence ($\vDash$) and formal deducibility ($\vdash$) was to define a proof system called formal deduction with which we could prove formally everything that is correct semantically.

This approach is similar to axiomatic geometry, in the sense that we accept as correct only those theorems that have a formal proof, based on the 11 rules.

Consider a system of formal deducibility, defined by a certain number of formal deduction rules.

For this system of formal deduction to be "good", it has to be connected to informal reasoning in the following sense:

- It should not be able to formally prove incorrect statements (soundness)
- It should be able to formally prove every correct statement (completeness)

A system of formal deducibility, denoted by $\vdash_*$, is defined by listing its formal deduction rules.

Suppose that statement "If $\Sigma \vdash_* A$ then $\Sigma \vDash A$" is true for any Σ and A.

This means that what can be proved formally, by using the system of formal deducibility $\vdash_*$, also holds in informal reasoning.

In other words, it means that in the system $\vdash_*$, we cannot prove incorrect statements.

If this property holds for a given system of formal deducibility $\vdash_*$, then that system is called **sound**.

The next theorem will prove that the system of formal deduction denoted by $\vdash$, based on the 11 given rules of formal deduction, is **sound**.

Theorem (Soundness Theorem): If $\Sigma \vdash A$ then $A \vDash A$, where $\vdash$ means the formal deduction based on the 11 given rules.

Proof: Structural induction, on the structure of "$\Sigma \vdash A$".

We only prove the cases of (Ref), $(\neg-)$ and $(\vee-)$.

Base Case (Ref). If $A \vdash A$, then $A \vDash A$. Obvious.

Inductive Step, subcase $(\neg-)$.

Assume that the statement of the theorem holds for $\Sigma, \neg A \vdash B$, and $\Sigma, \neg A \vdash \neg B$ (the IH). We want to prove that

$$\begin{aligned} \text{If } \Sigma, \neg A \vdash B \quad &\text{and} \\ \Sigma, \neg A \vdash \neg B, \\ \text{then } \Sigma \vDash A \end{aligned}$$

By the IH we have that $\Sigma, \neg A \vdash B$ implies $\Sigma, \neg A \vDash B$, and $\Sigma, \neg A \vdash \neg B$ implies $\Sigma, \neg A \vDash \neg B$.

Use "proof by contradiction". Assume that $\Sigma\neg \vDash A$ that is, there is a truth valuation t such that $\Sigma^t = 1$ and $A^t = 0$. Then $(\neg A)^t = 1$.

Since $\Sigma, \neg A \vDash B$ and $\Sigma, \neg A \vDash \neg B$, this implies $B^t = 1$ and $(\neg B)^t = 1$, which is a contradiction.

Hence $\Sigma \vDash A$, and the proof of subcase $(\neg-)$ is complete.

Subcase $(\vee-)$

Assume that the statement of the theorem holds for $\Sigma, A \vdash C$ and $\Sigma, B \vdash C$ (the IH). We want to prove that

$$\begin{aligned} \Sigma, A \vdash C \quad &\text{and} \\ \Sigma, B \vdash C \quad & \\ \text{then } \Sigma, A \vee B \vDash C & \end{aligned}$$

By the IH, we have that $\Sigma, A \vdash C$ implies $\Sigma, A \vDash C$, and $\Sigma, B \vdash C$ implies $\Sigma, B \vDash C$.

Let t be an arbitrary truth valuation such that $\Sigma^t = 1$ and $(A \vee B)^t = 1$. Then $A^t = 1$ or $B^t = 1$. Use "proof by cases".

Case (a): If $A^t = 1$, then, by $\Sigma, A \vDash C$, we have that $C^t = 1$.

Case (b): If $B^t = 1$, then, by $\Sigma, B \vDash C$, we have that $C^t = 1$.

Hence $C^t = 1$, implying $\Sigma, A \vee B \vDash C$. This proves subcase $(\vee -)$.

The other subcases are similar, and this completes the proof of the Soundness Theorem.

Completeness of a formal deduction system

Consider a system of formal deducibility, denoted by $\vdash_*$, defined by certain formal deduction rules.

Suppose that the statement "If $\Sigma \vDash A$ then $\Sigma \vdash_* A$" is true for any set of formulas Σ and formula A.

This means that anything that holds by informal reasoning can be proved using the system of formal deducibility $\vdash_*$.

In other words, it means that whatever is correct, can be formally proved using the system $\vdash_*$.

If this property holds for a system of formal deducibility $\vdash_*$, then that system is called **complete**.

The next theorem will prove that the system of formal deduction denoted by $\vdash$, based on the 11 given rules for formal deduction is **complete**.

Theorem (Completeness Theorem).

If $\Sigma \vDash A$ then $\Sigma \vdash A$, where $\vdash$ means the formal deduction based on the 11 given rules.

Proof in three steps:

1. If $A_1, A_2, \ldots, A_n \vDash A$ then $\emptyset \vDash (A_1 \implies (A_2 \implies \ldots (A_n \implies A) \ldots))$
2. If $\emptyset \vDash A$ then $\emptyset \vdash A$ (every tautology has a formal proof).
3. If $\emptyset \vdash (A_1 \implies (A_2 \implies \ldots (A_n \implies A) \ldots))$ then $A_1, A_2, \ldots, A_n \vdash A$.

The idea is to prove the required statement for the case $\Sigma = \emptyset$ (prove that every tautology is formally provable, step (2)), then "convert" from general set of premises Σ to the empty set of premises $\emptyset$ (step (1)), and "convert back" from $\emptyset$ to the set of premises Σ (step (3)).

We first prove that the "conversion" works, i.e., prove (1) and (3).

(1) Proof:

By contradiction. Assume there exists a truth valuation t such that $((A_1 \implies (A_2 \implies \ldots (A_n \implies A) \ldots)))^t = 0$.

This formula being structured as a series of nested implications, this implies that $A_1^t = A_2^t = \cdots = A_n^t = 1$ and $A^t = 0$.

This contradicts our hypothesis that $A_1, A_2, \ldots, A_n \vDash A$.

(2) Proof:

Assume that A is a tautology, and A has n atoms.

Construct 2^n subproofs of A - one for each truth valuation, and then use the Law of Excluded Middle, $\emptyset \vdash p \vee \neg p$, the rule $(\vee -)$, and (Tr.), to put them together.

More precisely:

Let the n atoms in A be $p_1, p_2, \ldots, p_n$, and let t be a truth valuation. Define (relative to t):

$$p'_i = \begin{cases} p_i & \text{if } p_i^t = 1 \\ \neg p_i & \text{if } p_i^t = 0 \end{cases}$$

Lemma: Let A be a formula with atoms $p_1, p_2, \ldots, p_n$ and let t be a truth valuation. Then

- if $A^t = 1$ then $p'_1, p'_2, \ldots, p'_n \vdash A$, and
- if $A^t = 0$ then $p'_1, p'_2, \ldots, p'_n \vdash \neg A$

Claim: Every tautology A is formally provable, that is, $\emptyset \vDash A$ implies $\emptyset \vdash A$.

Proof: Since all 2^n truth valuations make a tautology A true, the 1st statement of the Lemma guarantees that, for every possible choice for $p'_1, p'_2, \ldots, p'_n$, we can find a formal proof for A, that is, we can prove $p'_1, p'_2, \ldots, p'_n \vdash A$.

This implies that, for each row of the truth table for A, if we can choose $p_1 = p_i$ when $p_i^t = 1$, and $p'_i = \neg p_i$ when $p_i^t = 0$, we can find a proof for A, that is, we can prove that $p'_1, p'_2, \ldots, p'_n \vdash A$.

We then use the rule $(\vee -)$ to combine all the 2^n proofs for $p'_1, p'_2, \ldots, p'_n \vdash A$, into one big proof for A, that has as premises $(p_i \vee \neg p_i)$, for all $1 \leq i \leq n$.

Lastly, we use the Law of Excluded Middle, $\emptyset \vdash p_i \vee \neg p_i$ for all atoms p_i, together with the "big proof", and (Tr.), to obtain $\emptyset \vdash A$. This proves the Claim, and the Completeness Theorem.

The Soundness and Completeness Theorems associate the syntactic notion of formal deduction, based on the 11 rules, with the semantic notion of (tauto)logical consequence, and establish the equivalence between them.

The Soundness and Completeness Theorems say that with formal deduction (as defined by the 11 rules) we can prove

The Truth,
the whole truth, (completeness)
and nothing but the truth. (soundness)

Formal deduction cannot be used to prove that an argument is invalid!

Descartes famously said "I think, therefore I am". Joke: Descartes goes into a bar and the bartender asks him if he wants another drink. "I think not," says Descartes, and he vanishes.

The argument roughly translates to:

$$\begin{array}{l} T \implies A \\ \neg T \\ \hline \neg A \end{array}$$

This is an invalid argument, called the logical fallacy of denying the antecedent.

We can prove that $T \implies A, \neg T \neg \vDash \neg A$. However we cannot prove that the argument is invalid using formal deduction, $\vdash$.

Another logical fallacy.

"Why are you standing on this street corner, waiving your hands?"

"I am keeping away the elephants."

"But there aren't any elephants here."

"That's because I'm here."

This argument is roughly

$$\begin{array}{l} q \implies \neg q \\ \neg q \\ \hline p \end{array}$$

This is the fallacy of affirming the consequent.

Formal deduction proof strategies

If the conclusion is an implication, that is, we have to prove $\Sigma \vdash A \implies B$, then try using $(\implies +)$, as follows. Add A to the set of premises and try to prove B. In other words, prove $\Sigma, A \vdash B$ first. If this is proved, then one application of $(\implies +)$ will result in $\Sigma \vdash A \implies B$.

If one of the premises is a disjunction, that is, if we have to prove $\Sigma, A \vee B \vdash C$, then try to use "proof by cases" $(\vee -)$. In other words, prove separately $\Sigma, A \vdash C$ (Case 1), then $\Sigma, B \vdash C$ (Case 2), and then put these two proofs together with one application of $(\vee -)$ to obtain $\Sigma, A \vee B \vdash C$.

If we have to prove $C \vdash D$ and the direct proof does not work, try "proving the contrapositive", that is, try to prove $\neg D \vdash \neg C$. Then use the "flip-flop" theorem "If $A \vdash B$ then $\neg B \vdash \neg A$".

If everything else fails, try "proof by contradiction", $(\neg -)$:

- If we want to prove $\Sigma \vdash B$ and we do not know how, start with modified premises $\Sigma, \neg B$ (add a new premise, $\neg B$, to the premise set), and try to reach a contradiction:
- Prove that $\Sigma, \neg B \vdash C$, for some formula C
- Prove that $\Sigma, \neg B \vdash \neg C$, for the same C
- If we succeed in proving both, we reached a contradiction (we proved both C and $\neg C$)
- This means that our assumption, $\neg B$, was incorrect, and its opposite (that is B) holds.
- Formally, from $\Sigma, \neg B \vdash C$ and $\Sigma, \neg B \vdash \neg C$, one application of $(\neg -)$ yields $\Sigma \vdash B$.

Note that to prove $\Sigma \vdash A$, sometimes we have to start our proof with a premise set which is (somewhat) different from Σ.

For example, if we want to use $(\neg -)$ to prove $\Sigma \vdash A$, the proof starts with premises $\Sigma, \neg A$ (from which we try to prove a contradiction). Or, in the proof for hypothetical syllogism, the premises in the first line of the proof have an extra A for the proof to work.

However, while the premises in intermediate lines of the proof for $\Sigma \vdash A$ can be different from Σ, one must have a strategy of how to "undo" any such modifications of Σ by the end of the proof. This is because the last line of the proof must coincide exactly with $\Sigma \vdash A$ (otherwise, we proved a different theorem).

Note that these are only general formal deduction proof strategies (not algorithms).

A "line" in a proof can be used several times during the proof.

Example:

$$\{((s \wedge h) \implies p), s, (\neg p)\} \quad \vdash \quad (\neg h)$$

Here is a formal proof of the above result, in the proof system of Formal Deduction.

(1)	$((s \wedge h) \implies p), s, (\neg p), h$	$\vdash$	s	(by $(\in)$)
(2)	$((s \wedge h) \implies p), s, (\neg p), h$	$\vdash$	h	(by $(\in)$)
(3)	$((s \wedge h) \implies p), s, (\neg p), h$	$\vdash$	$(s \wedge h)$	(by $(\wedge+), (1), (2)$)
(4)	$((s \wedge h) \implies p), s, (\neg p), h$	$\vdash$	$((s \wedge h) \implies p)$	(by $(\in)$
(5)	$((s \wedge h) \implies p), s, (\neg p), h$	$\vdash$	p	(by $(\implies -), (3), (4)$)
(6)	$((s \wedge h) \implies p), s, (\neg p), h$	$\vdash$	$(\neg p)$	(by $(\in)$)
(7)	$((s \wedge h) \implies p), s, (\neg p)$	$\vdash$	$(\neg h)$	(by $(\neg+), (5), (6)$)

Definition: A set of formulas Σ is consistent (w.r.t a system of formal deduction, herein $\vdash$) if there is no formula F, such that $\Sigma \vdash F$ and $\Sigma \vdash \neg F$). Otherwise Σ is called inconsistent.

Lemma: A set Σ of formulas is satisfiable iff Σ is consistent.

Proof:

" $\implies$ " Σ is satisfiable, so there exists a truth valuation t with $\Sigma^t = 1$. Assume Σ is inconsistent. Then there exists a formula F such that $\Sigma \vdash F$ and $\Sigma \vdash \neg F$. By the Soundness of formal deduction, this implies $\Sigma \vDash F$ and $\Sigma \vDash \neg F$ which, since $\Sigma^t = 1$, implies $F^t = (\neg F)^t = 1$ - a contradiction. Thus, Σ is consistent.

" $\impliedby$ " Conversely, let Σ be a consistent set of formulas. Assume Σ is not satisfiable. Then, for all truth valuations t we have $\Sigma^t = 0$. For any logic formulas F, we have consequently $\Sigma \vDash F$ and $\Sigma \vDash \neg F$ (vacuously, since there is no valuation t' with $\Sigma^{t'} = 1$). By Completeness, this implies $\Sigma \vdash F$ and $\Sigma \vdash \neg F$, which means that Σ is inconsistent - a contradiction. Thus, Σ is satisfiable.

7 Logic07: Resolution for Propositional Logic

In the field of Artificial Intelligence, there have been many attempts to construct programs that could prove theorems (or verify their proofs) automatically.

Given a set of axioms and a technique for deriving new theorems from old theorems and axioms, would such a program be able to prove a particular theorem?

Early attempts faltered. J.A. Robinson at Syracuse University discovered the technique called resolution.

Resolution theorem proving is a method of formal derivation (formal deduction) that has the following features:

The only formulas allowed in resolution theorem proving are disjunctions of literals, such as $(p \vee q \vee \neg r)$.

Recall that such a disjunction of literals is called a (disjunctive) clause. Hence, all formulas involved in resolution theorem proving must be (disjunctive) clauses.

There is only one rule of formal deduction, called resolution.

<u>How does resolution work?</u>

Recall: A set of formulas $\Sigma \subseteq \text{Form}(\mathcal{L}^p)$ is consistent iff there is no formula $F \in \text{Form}(\mathcal{L}^p)$ such that $\Sigma \vdash F$ and $\Sigma \vdash \neg F$ (one cannot derive a contradiction). A set of formulas that is not consistent is called inconsistent.

For the system of formal deduction based on the 11 rules $(\vdash)$ we proved that a set is satisfiable iff it is consistent. A similar result holds for the proof system based on resolution.

To prove that an argument $A_1, A_2, \ldots, A_n \vDash C$ is valid, we show that the set $\{A_1, A_2, \ldots, A_n, \neg C\}$ is not satisfiable, by proving that it is inconsistent.

To prove the latter, we show that from $\{A_1, A_2, \ldots, A_n, \neg C\}$ we can formally derive both F and $\neg F$, for some formula F.

In general, one can convert any formula into one or more disjunctive clauses.

To do this, one first converts the formula into a conjunction of disjunctions; that is, one converts the formula into conjunctive normal form.

Each term of the conjunction is then made into a clause of its own.

Example: Convert $p \implies (q \wedge r)$ into clauses.

Solution:

We first eliminate the $\implies$ by writing $\neg p \vee (q \wedge r)$.

We then apply the distributivity law to obtain

$$p \implies (q \wedge r) \Leftrightarrow (\neg p \vee q) \wedge (\neg p \vee r)$$

This yields the two clauses $\neg p \vee q$ and $\neg p \vee r$.

Definition: Resolution is the formal deduction rule

$$C \vee p, D \vee \neg p \vDash_r C \vee D$$

where C and D are disjunctive clauses, and p is a literal.

$C \vee p$ and $D \vee \neg p$ are parent clauses, and $C \vee D$ is the resolvent clause. We say that we resolve the two parent clauses over p.

Let $\perp$ denote a clause that is always false (a contradiction), hereafter called empty clause. ($\perp$ is not a formula, but a notation for a contradiction, e.g., $p \wedge \neg p$)

The resolvent of p and $\neg p$ is the empty clause, i.e., $p, \neg p \vdash_r \perp$.

Removal of duplicates of literals in disjunctive clauses is allowed, e.g., $p \vee q \vee r, p \vee \neg r \vdash_r p \vee q$.

Commutativity of disjunction is allowed within clauses.

A (resolution) derivation from a set of clauses S is a finite sequence of clauses such that each clause is either in S or results from previous clauses in the sequence by resolution.

Two comments can be resolved if and only if they contain two complementary literals, say p (a positive literal) and $\neg p$ (a negative literal).

If the complementary literals are p and $\neg p$, one says that we resolve over p, or that resolution is on p.

The result of resolution on p is the resolvent, which is the disjunction of all literals of the parent clauses, except that p and $\neg p$ are omitted.

In the particular case when the two parent clauses are p and $\neg p$, their resolvent is called the empty clause, denoted by $\perp$.

In the context of resolution, the empty clause $\perp$ is a notation signifying that the contradiction $p \wedge \neg p$ was reached.

By definition, the empty clause is not satisfiable.

Find the resolvent of $p \vee \neg q \vee r$ and $\neg s \vee q$.

Solution: The two parent clauses $p \vee \neg q \vee r$ and $\neg s \vee q$ can be resolved over q, because q is negative in the first clause and positive in the second.

The resolvent is the disjunction of $p \vee r$ with $\neg s$, which yields $p \vee r \vee \neg s$.

To prove that the argument with premises $A_1, A_2, \ldots, A_n$ and conclusion C is valid, we show that from the set

$$\{A_1, A_2, \ldots, A_n, \neg C\}$$

we can derive, by $\vdash_r$, the empty clause $\perp$ (a contradiction), as follows:

- Pre-process the input by transforming each of the formulas in $\{A_1, A_2, \ldots, A_n, \neg C\}$ into conjunctive normal form.

- Make each disjunctive clause a distinct clause. These clauses are the input of the resolution procedure.
- If the resolution procedure outputs the empty clause, $\bot$, this implies that the set $\{A_1, A_2, \ldots, A_n, \neg C\}$ is inconsistent, hence not satisfiable, and thus the argument is valid.

Resolution Procedure

Input: Set of disjunctive clauses $S + \{D_1, D_2, \ldots, D_m\}$

Repeat, trying to get the empty clause, $\bot$:

- Choose two parent clauses, one with p and one with $\neg p$
- Resolve the two parent clauses, and call the resolvent D.
- If $D = \bot$ then output "empty clause"
- Else add D to S

Parent clauses can be reused.

Example: Modus ponens by resolution

Prove that

$$p, p \implies q \vdash_r q$$

Proof:

1.	p	Premise
2.	$\neg p \lor q$	Premise
3.	$\neg q$	Negation of conclusion
4.	q	Resolvent of 1,2 (over p)
5.	$\bot$	Resolvent of 3,4 (over q)

Soundness of resolution formal deduction

Theorem: The resolvent is tautologically implied by its parent clauses, which makes resolution a sound rule of formal deduction.

Proof: Let p be a propositional variable, and let A and B be clauses.

Assume that $p \lor A, \neg p \lor B \vdash_r A \lor B$.

We want to prove that $p \lor A, \neg p \lor B \vDash A \lor B$.

(i) If at least one of A or B is not empty, then we prove:

Claim: $p \lor A, \neg p \lor B \vDash A \lor B$ for any clauses A, B, both not empty.

Consider a truth valuation t such that $(p \lor A)^t = (\neg p \lor B)^t = 1$.

- If $p^t = 0$, then $A^t = 1$, because otherwise $(p \lor A)^t = 0$.
- Similarly, if $p^t = 1$, then $B^t = 1$, because otherwise $(\neg p \lor B)^t = 0$

In either situation, $(A \lor B)^t = 1$, therefore $p \lor A, \neg p \lor B \vDash A \lor B$. This proves the Claim.

(ii) If both A and B are empty then the resolvent of p and $\neg p$ is the empty clause $\bot$, which is short for $p \land \neg p$, and always false.

In this case $p, \neg p \vDash \bot$ because the premises are contradictory.

In both cases, (i) and (ii), the required tautological consequence holds, and this proves soundness of resolution.

Prove that $p \implies q, q \implies r \vdash_r p \implies r$

Proof: The CNF for $p \implies q$ is $\neg p \vee q$. The CNF for $q \implies r$ is $\neg q \vee r$. The CNF for the negation of the conclusion is $\neg(\neg p \vee r) \dashv\vdash p \wedge \neg r$.

1.	$\neg p \vee q$	Premise
2.	$\neg q \vee r$	Premise
3.	p	Derived from the negation of conclusion
4.	$\neg r$	Derived from the negation of conclusion
5.	q	Resolvent of 1, 3 (over p)
6.	$\neg q$	Resolvent of 2, 4 (over r)
7.	$\bot$	Resolvent of 5, 6 (over q)

A common mistake in using resolution is to apply it to more than one literal. This is not correct.

For example, the following is an incorrect use of resolution:

- $p \vee \neg q$
- $\neg p \vee q$
- $\bot$ (from 1,2 resolving over p and q)

This disagrees with the Soundness of Resolution since

$$p \vee \neg q, \neg p \vee q\neg \vDash \bot$$

We can prove the invalidity of the argument by noticing that we can satisfy the premises by setting p and q equal to 1, but cannot satisfy the conclusion $\bot$ (which is short for $p \wedge \neg p$, hence always false, and thus not satisfiable).

This is not resolution!

When doing resolution automatically, one has to decide in which order to resolve the clauses.

This order can greatly affect the time needed to find a contradiction.

Strategies include: The "Set-of-Support Strategy" and The Davis-Putnam Procedure (DPP)

Set-of-Support Strategy

One partitions all clauses into two sets, the set of support and the auxiliary set.

The auxiliary set is formed in such a way that the formulas in it are not contradictory.

For instance, the premises are usually not contradictory. The contradiction will only arise after one adds the negation of the conclusion.

One often uses the set of premises as the "auxiliary set", and the negation of the conclusion as the initial "set of support".

Since one cannot derive any contradiction by resolving clause within the auxiliary set, one avoids such resolutions.

Stated positively, when using the Set-of-Support Strategy, each resolution takes at least one clause from the set of support.

The resolvent is then added to the set of support.

Theorem: Resolution with the set-of-support strategy is complete.

Example

Prove p_4 from $p_1 \implies p_2, \neg p_2, \neg p_1 \implies p_3 \vee p_4, p_3 \implies p_5, p_6 \implies \neg p_5$ and p_6 by using the set-of-support strategy.

The auxiliary set consists of the clauses obtained from $p_1 \implies p_2, \neg p_2, \neg p_1 \implies p_3 \vee p_4, p_3 \implies p_5, p_6 \implies \neg p_5$ and p_6.

The initial set of support Σ is given by $\Sigma = \{\neg p_4\}$, the negation of the conclusion.

One then performs all the possible resolutions involving $\neg p_4$, then all possible resolutions involving the resulting resolvents, and so on.

At each step, a resolvent (which has at least one parent in the set of support) gets added to the set of support.

Prove p_4 from $p_1 \implies p_2, \neg p_2, \neg p_1 \implies p_3 \vee p_4, p_3 \implies p_5, p_6 \implies \neg p_5$ and p_6, by using the set-of-support strategy.

1.	$\neg p_1 \vee p_2$	Premise	
2.	$\neg p_2$	Premise	
3	$p_1 \vee p_3 \vee p_4$	Premise	
4.	$\neg p_3 \vee p_5$	Premise	
5.	$\neg p_6 \vee \neg p_5$	Premise	
6.	p_6	Premise	
7.	$\neg p_4$	Negation of conclusion	$\Sigma = \{7\}$
8.	$p_1 \vee p_3$	Resolvent of 7,3	$\Sigma = \{7, 8\}$
9.	$p_2 \vee p_3$	Resolvent of 1,8	$\Sigma = \{7, 8, 9\}$
10.	p_3	Resolvent of 2,9	$\Sigma = \{7, 8, 9, 10\}$
11.	p_5	Resolvent of 4,10	$\Sigma = \{7, 8, 9, 10, 11\}$
12.	$\neg p_6$	Resolvent of 5,11	$\Sigma = \{7, 8, 9, 10, 11, 12\}$
13.	$\bot$	Resolvent of 6,12	

The Pigeonhole Principle $\mathcal{P}_n$ says that one cannot put $n + 1$ objects into n slots, with distinct objects going into distinct slots.

Example: In any group of 367 people there must be at least two with the same birthday.

Formulate the Pigeonhole Principle as a conjunction of formulas.

(a) Choose propositional variables p_{ij} for $1 \leq i \leq n + 1, 1 \leq j \leq n$.

(b) Define p_{ij} as true iff the ith pigeon goes into the jth slot.

(c) Construct clauses for:

1. Each pigeon $i, 1 \leq i \leq n+1$, goes into some slot $k, 1 \leq k \leq n : p_{i1} \vee p_{i2} \ldots \vee p_{in}$ for $1 \leq i \leq n+1$.
2. Distinct pigeons $i \neq j, 1 \leq i, j \leq n+1$ cannot go into the same slot $k : p_{ik} \implies \neg p_{jk} \uplus \neg p_{ik} \vee \neg p_{jk}$ for $1 \leq i < j \leq n + 1, 1 \leq k \leq n$

Observe now that any truth valuation that satisfies the conjunction of all the above clauses would map $n + 1$ pigeons one-to-one into n slots.

Of course, by the Pigeonhole Principle, this cannot be done, so this set of clauses must be unsatisfiable.

What is the Pigeonhole Principle $\mathcal{P}_2$ (3 pigeons and 2 slots) states as a resolution problem?

Every pigeon in at least one slot: $p_{11} \vee p_{12}, p_{21} \vee p_{22}, p_{31} \vee p_{32}$.

No two pigeons per slot:

Slot 1: $\neg p_{11} \vee \neg p_{21}, \quad \neg p_{11} \vee \neg p_{31} \quad \neg p_{21} \vee \neg p_{31}$

Slot 2: $\neg p_{12} \vee \neg p_{22}, \quad \neg p_{12} \vee \neg p_{32}, \quad \neg p_{22} \vee \neg p_{32}$

Note: We do not need all possible pairs (i, j) for every slot k because, e.g., $(p_{31} \implies \neg p_{11}) \barwedge (p_{11} \implies \neg p_{31}) \barwedge (\neg p_{11} \vee \neg p_{31})$

Since the set of the 9 clauses is not satisfiable (due to the Pigeonhole Principle), one should be able to derive the empty clause from it.

Davis-Putnam Procedure (DPP)

Any clause corresponds to a set of literals, that is, the literals contained within the clause.

For instance, the clause $p \vee \neg q \vee r$ corresponds to the set $\{p, \neg q, r\}$ and $\neg s \vee q$ corresponds to the set $\{\neg s, q\}$.

Since the order of the literals in a disjunction is irrelevant, and since the same is true for the multiplicity of the terms (duplicates do not matter), the set associated with the clause completely determines the clause.

For this reason, one frequently treats clauses as sets, which allows one to speak of the union of two clauses.

If clauses are represented as sets, one can write the resolvent, on p, of two clauses $C \cup \{p\}$ and $D \cup \{\neg p\}$, when neither C nor D is empty, as

$$[(C \cup \{p\}) \cup (D \cup \{\neg p\})] \setminus \{p, \neg p\}$$

In words, the resolvent is the union of all literals in the parent clauses except that the two literals involving p are omitted.

In the particular case when C and D are both empty, the resolvent of $\{p\}$ and $\{\neg p\}$ is the empty clause, denoted by $\{\bot\}$ (not satisfiable by definition).

Given an input as a nonempty set of clauses in the propositional variables $p_1, p_2, \ldots, p_n$, the Davis-Putnam Procedure (DPP) repeats the following steps until there are no variables left:

- Remove all clauses that have both a literal q and its complement $\neg q$ in them (a disjunctive clause in which both q and $\neg q$ appear is a tautology, and will never lead to a contradiction)
- Choose a variable p appearing in one of the clauses
- Add to the set of clauses all possible resolvents using resolution on p (parent clauses containing p can be re-used)
- Discard all (parent) clauses with p or $\neg p$ in them
- Discard any duplicate clauses

We refer to this sequence of steps as eliminating the variable p

If in some step one resolves $\{p\}$ and $\{\neg p\}$ then one obtains the empty clause, $\{\bot\}$, and it will be the only clause at the end of the procedure.

If one never has a pair $\{p\}$ and $\{\neg p\}$ to resolve, then all the clauses will be discarded and the output will be no clauses.

Thus, the output of DPP is either the empty clause $\{\bot\}$, or the empty set (no clauses).

DPP

Input: A set S of disjunctive clauses, in DPP format, with propositional variables $p_1, p_2, \ldots, p_n, \quad n \geq 1$.

- Let $S_1 = S$
- Let $i = 1$
- LOOP until $i = n + 1$
- Discard members of S_i in which a literal and its complement appear, to obtain S_i'.
- Let T_i be the set of parent clauses in S_i' in which p_i or $\neg p_i$ appears

- Let U_i be the set of resolvent clauses obtained by resolving (over p_i) every pair of clauses $C \cup \{p_i\}$ and $D \cup \{\neg p_i\}$ in T_i
- Set S_{i+1} equal to $(S_i' \setminus T_i) \cup U_i$
- Let i be increased by 1
- ENDLOOP
- Output S_{n+1}

Example:

Apply the Davis-Putnam Procedure to the set of clauses

$$\{\neg p, q\}, \{\neg q, \neg r, s\}, \{p\}, \{r\}, \{\neg s\}$$

Eliminating p gives $\{q\}, \{\neg q, \neg r, s\}, \{r\}, \{\neg s\}$ (This is S_2 and S_2')

Eliminating q gives $\{\neg r, s\}, \{r\}, \{\neg s\}$. (This is S_3 and S_3')

Eliminating r gives $\{s\}, \{\neg s\}$. (This is S_4 and S_4')

Eliminating s gives $\{\bot\}$. (This is S_5)

The output is the empty clause $\{\bot\}$.

If the set of clauses is more complex, before each iteration (elimination of a variable) we give each clause in T_i a numerical identifier.

Then, in the next step (which produces the resolvents in U_i from parent clauses in T_i) we provide, for each resolvent, the identifiers of the two parent clauses that produced it.

If the output of DPP is the empty clause $\{\bot\}$, then this indicates that both p and $\neg p$ were produced. This implies that the set of clauses that was obtained by pre-processing the premises and negation of the conclusion of the argument is inconsistent, hence not satisfiable, that is, the argument (theorem) is valid.

If, on the other hand, the output of DPP is not clause, $\emptyset$, this means that no contradiction can be found, and the original argument (theorem) is not valid.

Soundness and Completeness of DPP

Theorem: Let S be a finite set of clauses. Then S is not satisfiable iff the output of DPP on input S is the empty clause $\{\bot\}$.

Proof idea:

Resolution propagates satisfiability "forwards", from parent clauses to resolvent (this follows by the Soundness of Resolution)

Resolution propagates satisfiability "backwards", from a resolvent to its parent clauses, as follows:

Saw we have a resolution $p \vee B, \neg p \vee C \vdash_r B \vee C$. If $B \vee C$ is satisfiable, there exists a truth valuation t with $(B \vee C)^t = 1$.

- If $B^t = 1$, then extend t (define t for p, which did not occur in B or C) to $p^t = 0$. Then both parent clauses are satisfied by t.
- If $C^t = 1$, then extend t to $p^t = 1$. Then both parent clauses are satisfied by t.
- Hence, the parent clauses are satisfiable by some extension of t.

Proof: "$S \vdash_r \{\bot\}$ by DPP" implies "S not satisfiable"

Sketch:

We can use induction on i to show that if C is any clause in S_i then there is a resolution derivation of C from the initial set S.

Since the output of DPP is the empty clause, that is, $\{\bot\} \in S_{n+1}$, it would follow that there is a resolution derivation from S to $\{\bot\}$.

Since $\{\bot\}$ is not satisfiable and resolution preserves satisfiability (by Soundness of Resolution) this implies that S is not satisfiable.

This concludes the proof of this implication

Proof of the other implication

"S not satisfiable" implies "$S \vdash_r \{\bot\}$ by DPP"

Proof by contradiction:

Assume that the output of the DPP is not the empty clause $\{\bot\}$, but the empty set $\emptyset$ (the only other possibility).

We want to show that this would imply that S was satisfiable.

If $S_{n+1} = \emptyset$, then S_{n+1} is (vacuously) satisfiable.

We will prove that if S_{i+1} is satisfiable then S_i is satisfiable.

In other words, satisfiability also propagates "backwards".

If proved, this would lead to a contradiction with out assumption that $S = S_1$ was not satisfiable, and complete the proof of this implication.

S_{i+1} satisfiable implies S_i satisfiable

S_{i+1} has variables $p_{i+1}, \ldots, p_n$.

S_i has variables $p_i, p_{i+1}, \ldots, p_n$ (one extra variable p_i, which is eliminated in iteration i of DPP that constructs S_{i+1} from S_i)

Recall that $S_{i+1} = (S'_i \setminus T_i) \cup U_i$

Assume S_{i+1} is satisfied by some truth valuation t_{i+1}. Then t_{i+1} satisfies both U_i and $(S'_i \setminus T_i)$.

Since $S'_i = (S'_i \setminus T_i) \cup T_i$, to show that S'_i is satisfiable, it suffices to show that T_i is satisfiable, as follows.

T_i is satisfied by a truth valuation obtained by extending t_{i+1} to a truth valuation that coincides with t_{i+1} on variables $p_{i+1}, \ldots, p_n$, and assigns a suitable value to p_i (p_i does not occur in S_{i+1})

Note: Clearly, S_i is satisfiable iff S'_i is satisfiable (all clauses deleted from S_i to obtain S'_i contain complementary literals, and are thus tautologies, hence satisfiable).

Show that T_i (parents with p_i) is satisfiable

Assume $S_{i+1} = (S'_1 \setminus T_i) \cup U_i$ is satisfied by some truth valuation t_{i+1} that assigns some truth values to $p_{i+1}, \ldots, p_n$.

Claim: One of the following two truth valuations satisfy T_i

- t_0: agrees with t_{i+1} on variables $p_{i+1}, \ldots, p_n$ and $(p_i)^{t_0} = 0$,
- t_1: agrees with t_{i+1} on variables $p_{i+1}, \ldots, p_n$ and $(p_i)^{t_1} = 1$.

Assume neither t_0 nor t_1 satisfies T_i.

Since t_0 satisfies all formulas in T_i that contain $\neg p_i$, it must falsify some clause $D \cup \{p_i\}$ in T_i. As $D \cup \{p_i\}$ is not satisfied by t_0, we have that D is not satisfied by $t_{i+1} = t_0 \mid_{\{p_{i+1}, \ldots, p_n\}}$.

Since t_1 satisfies all formulas in T_i that contain p_i, it must falsify some clause $E \cup \{\neg p_i\}$. As $E \cup \{\neg p_i\}$ is not satisfied by t_1, we have that E is not satisfied by $t_{i+1} = t_1 \mid_{\{p_{i+1}, \ldots, p_n\}}$.

As t_{i+1} satisfies neither D nor E, it follows that it does not satisfy $D \cup E$ - a contradiction, since $D \cup E \subseteq S_{i+1}$ and $(S_{i+1})^{t_{i+1}} = 1$. This concludes the proof of the Claim.

Concluding the proof of the other implication.

Since we assumed that

$$S_{i+1} = (S'_i \setminus T_i) \cup U_i$$

was satisfiable, and we proved that T_i is satisfied by an extension of one of the truth valuations that satisfies S_{i+1}, we have that S'_i (and thus S_i) is also satisfiable by that truth valuation.

Recall that we had assumed (for the sake of contradiction) that $S_{n+1} = \emptyset$, which is vacuously satisfiable.

Working backwards, this implies that $S_1 = S$ is satisfiable, which contradicts the hypothesis of the implication that we have to prove, namely

$$\text{"}S \text{ not satisfiable" implies "} S \vdash_r \{\bot\} \text{ by DPP"}$$

Since we reached a contradiction, our assumption that "$S \vdash_r \emptyset$" by DPP was incorrect, and we have "$S \vdash_r \{\bot\}$ by DPP".

The theorem proved that "$S \vdash_r \{\bot\}$ by DPP" implies "S not satisfiable". How does this show the soundness of DPP?

Say we are given an argument to prove, with set of premises Σ, and conclusion A.

Taking $S = \Sigma \cup \{\neg A\}$, if we prove "$S \vdash_r \{\bot\}$ by DPP", then the theorem (implication cited above) implies "$\Sigma \cup \{\neg A\}$ not satisfiable."

$\Sigma \cup \{\neg A\}$ not satisfiable further implies $\Sigma \vDash A$.

Thus, if we prove the validity of an argument formally, by using DPP to obtain the empty clause, then the argument is indeed valid, that is, DPP is sound.

The theorem proved that "S is not satisfiable" implies "$S \vdash_r \{\bot\}$ by DPP." How does this show completeness of DP?

- Assume we have a valid argument $\Sigma \vDash A$
- This implies "$\Sigma \cup \{\neg A\}$ not satisfiable"
- Taking $S = \Sigma \cup \{\neg A\}$, the theorem (implication cited above) implies "$\Sigma, \neg A \vdash_r \{\bot\}$ by DPP".
- This means that every valid argument can be formally proved to be correct by the method based on DPP, that is, DPP is complete.

Exercise:

Use the DPP to show that the set of 12 clauses below is not satisfiable. (If the set of clauses would originate from pre-processing the premises and the negation of the conclusion of an argument in propositional logic, the unsatisfiability of the set of clauses would lead us to conclude that the argument was valid.)

Eliminate the variables in the order p, q, r, s, t.

$\{p, q\}, \{\neg p, \neg q\}, \{\neg q, r, t\}, \{q, \neg r, t\}, \{q, r, \neg t\}, \{\neg q, \neg r, \neg t\}, \{\neg r, s\},$
$\{r, \neg s\}, \{\neg p, s, t\}, \{p, \neg s, t\}, \{p, s, \neg t\}, \{\neg p, \neg s, \neg t\}$

Note: We remove the underlined clauses dynamically on the fly (they are tautologies).

$S_1 = S'_1 = \{p, q\}, \{\neg p, \neg q\}, \{\neg q, r, t\}, \{q, \neg r, t\}, \{q, r, \neg t\}, \{\neg q, \neg r, \neg t\}, \{\neg r, s\},$
$\{r, \neg s\}, \{\neg p, s, t\}, \{p, \neg s, t\}, \{p, s, \neg t\}, \{\neg p, \neg s, \neg t\}$

Eliminate p:

$T_1 = \{p, q\}^1, \{\neg p, \neg q\}^2, \{\neg p, s, t\}^3, \{p, \neg s, t\}^4, \{p, s, \neg t\}^5, \{\neg p, \neg s, \neg t\}^6$

$U_1 = \underline{\{q, \neg\}^{1,2}}, \{q, s, t\}^{1,3}, \{q, \neg s, \neg t\}^{1,6}, \{\neg q, \neg s, t\}^{2,4}, \{\neg q, s, \neg t\}^{2,5},$
$\underline{\{s, t, \neg s\}^{3,4}}, \underline{\{s, t, \neg t\}^{3,5}}, \underline{\{\neg s, t, \neg t\}^{4,6}}, \underline{\{s, \neg t, \neg s\}^{5,6}}$

$S'_{i+1} = (S'_i \setminus T_i) \cup U_i$

$S'_2 = \{\neg q, r, t\}, \{q, \neg r, t\}, \{q, r, \neg t\}, \{\neg q, \neg r, \neg t\}, \{\neg r, s\}, \{r, \neg s\}, \{q, s, t\}, \{q, \neg s, \neg t\},$
$\{\neg q, \neg s, t\}, \{\neg q, s, \neg t\}$

Eliminate q:

$T_2 = \{\neg q, r, t\}^1, \{q, \neg r, t\}^2, \{q, r, \neg t\}^3, \{\neg q, \neg r, \neg t\}^4, \{q, s, t\}^5, \{q, \neg s, \neg t\}^6,$
$\{\neg q, \neg s, t\}^7, \{\neg q, s, \neg t\}^8$

$U_2 = \underline{\{r, t, \neg r\}^{\{1,2\}}}, \underline{\{r, t, \neg t\}^{1,3}}, \{r, t, s\}^{1,5}, \underline{\{r, t, \neg s, \neg t\}^{1,6}}, \underline{\{\neg r, t, \neg t\}^{2,4}}, \{\neg r, t, \neg s\}^{2,7}, \underline{\{\neg r, t, s, \neg t\}^{2,8}}, \underline{\{r, \neg t, \neg r\}^{3,4}}, \underline{\{r, \neg t, \neg s, t\}^{3,7}}, \{r, \neg t, s\}^{3,8}, \underline{\{\neg r, \neg t, s, t\}^{4,5}}, \{\neg r, \neg t, \neg s\}^{4,6}, \underline{\{s, t, \neg s\}^{5,7}}, \underline{\{s, t, \neg t\}^{5,8}}, \underline{\{\neg s, \neg t, t\}^{6,7}}, \underline{\{\neg s, \neg t, s\}^{6,8}}$

$S'_3 = \{\neg r, s\}^1, \{r, \neg s\}^2, \{r, t, s\}^3, \{\neg r, t, \neg s\}^4, \{r, \neg t, s\}^5, \{\neg r, \neg t, \neg s\}^6$

Eliminate r:

$T_3 = \{\neg r, s\}^1, \{r, \neg s\}^2, \{r, t, s\}^3, \{\neg r, t, \neg s\}^4, \{r, \neg t, s\}^5, \{\neg r, \neg t, \neg s\}^6$

$U_3 = \underline{\{s, \neg s\}^{1,2}}, \{s, t\}^{1,3}, \{s, \neg t\}^{1,5}, \{\neg s, t\}^{2,4}, \{\neg s, \neg t\}^{2,6}, \underline{\{t, s, \neg s\}^{3,4}}, \underline{\{t, s, \neg t, \neg s\}^{3,6}}, \underline{\{t, \neg s, \neg t, s\}^{4,5}}, \underline{\{\neg t, s, \neg s\}^{5,6}}$

$S'_4 = \{s, t\}, \{s, \neg t\}, \{\neg s, t\}, \{\neg s, \neg t\}$

Eliminate s:

$T_4 = \{s, t\}^1, \{s, \neg t\}^2, \{\neg s, t\}^3, \{\neg s, \neg t\}^4$

$U_4 = \{t\}^{1,3}, \underline{\{t, \neg t\}^{1,4}}, \underline{\{\neg t, t\}^{2,3}}, \{\neg t\}^{2,4}$

$S'_5 = \{t\}, \{\neg t\}$

Eliminate t:

$T_5 = \{t\}^1, \{\neg t\}^2$

$U_5 = \{\bot\}^{1,2}$

$S_6 = \{\bot\}$ (the empty clause)

The outcome of DPP is $\{\bot\}$, the empty clause (indicating that a contradiction was reached, by resolving two complimentary literals).

By DPP Soundness and Completeness, this implies that S_1 is not satisfiable.

(If S_1 would have originated from pre-processing the set of premises, and the negation of the conclusion, of an argument, this outcome would further imply that the argument was valid.)

8 Logic10: First-Order Logic

In propositional logic, a simple proposition is an unanalyzed whole which is either true or false.

There are certain arguments that seem to be logical, yet they cannot be expressed using propositional logic.

For analyzing these we introduce first-order logic.

Alternate names: Predicate logic, predicate calculus, elementary logic, restricted predicate calculus, relational calculus, theory of quantification with equality, etc.

Example

1. All humans are mortal
2. Socrates is human
3. Socrates is mortal

This is clearly not a valid argument in propositional logic.

To show that arguments such as the previous one are valid (sound), we must be able to identify individuals together with their properties and relations.

This is the objective of first-order logic.

First-order logic (FoL) is an extension of propositional logic.

Applications in CS:

- Prolog

- Automated theorem provers, proof assistants
- Proving program correctness

First-order logic is used to describe, e.g., mathematical theories. Such a theory comprises certain concepts specific to the structure/theory:

- A domain of objects (called individuals). Designated individuals in the domain. Variables ranging over the domain.
- Functions on the domain
- Relations

In addition, in making statements about individuals in the domain we use first-order logic concepts:

- Logical connectives
- Quantifiers
- Punctuation

To prevent ambiguities we introduce the concept of a domain or universe of discourse.

Definition: The domain (or universe of discourse) is the collection of all persons, ideas, symbols, data structures, and so on, that affect the logical argument under consideration.

Many arguments involve numbers and, in this case, one must stipulate whether the domain is, for example, the set of natural numbers, of integers, of real numbers, or of complex numbers.

The truth of a statement may depend on the domain selected.

Definition: The elements of the domain are called individuals.

An individual can be a person, number, data structure, or anything else one wants to reason about.

To avoid trivial cases, we stipulate that every domain must contain at least one individual.

Instead of the word individual, one sometimes uses the word object.

Another important concept is that of functions whose inputs and outputs are both in the domain (universe of discourse).

For example, $f(1, 2)$ may stand for the sum of 1 and 2.

Each function has an arity, defined as the number of arguments the function takes as input.

The arity of a function is fixed. We can think of individuals as functions of arity 0.

Relations make true/false statements about individuals in the domain.

Mary and Paul are siblings.
Joan is the mother of Mary.
Socrates is human.
The sum of 2 and 3 is 5.

In each of these statements, there is a list of individuals, called argument list, together with phrases that describe certain relations among the individuals in the argument list.

Definition: The number of elements in the argument list of a relation is called the arity of the relation.

For instance, the relation "Is the mother of" has arity 2.

The arity of a relation is fixed: A relation cannot have to arguments in one case and three in another.

A one-place relation is called a property.

Often we do not want to associate the arguments of a function or relation with a particular individual. To avoid this, we use variables, that range over the domain.

Variables are frequently chosen from the end of the alphabet; x, y, and z, with or without subscripts, suggest (bound) variable names, and u, v, and w, with or without subscripts, suggest (free) variable names. The distinction will be explained later.

Examples:

Human(u) may denote "u is human",

Mother(u, w) may denote "u is the mother of w".

If all arguments of a relation are individuals in the domain, then the resulting formula must be either true or false.

This is part of the definition of the relation.

For instance, if the domain consists of Joan, Doug, Mary and Paul, we must know, for each ordered pair of individuals whether or not the relation is the mother of is true.

In a finite domain, one can represent the assignments of relations with arity n by $n-$dimensional arrays.

Note that the usual mathematical symbols $>$, or $<$, $\geq$, or $\leq$, or $=$ are all relations, namely of arity 2 (binary relations).

These relations are normally used in infix notation.

By infix notation, the binary relation symbol is placed between its two arguments.

Let $A(u)$ represent a formula, and let u be a variable.

If we want to indicate that $A(u)$ is true for all possible values of u in the domain, we write $\forall x A(x)$.

Here, $\forall x$ is called universal quantifier, and $A(x)$ is called the scope of the quantifier.

The variable x is said to be bound by the quantifier.

Statements containing words like all, for all, for every etc., usually indicate universal quantification.

Translate "Everyone needs a break" into first-order logic.

Let D be the set of all people.

We define $B(u)$ to mean u needs a break. In other words,

$$B(u) = \begin{cases} 1, & \text{if } u \text{ needs a break} \\ 0, & \text{otherwise} \end{cases}$$

Then the translation in first-order logic is: $\forall x B(x)$.

If we want to indicate that $A(u)$ is true for at least one value u in the domain (possibly more than one, but not necessarily) we write $\exists x A(x)$.

Here, $\exists x$ is called the existential quantifier, and $A(x)$ is called the scope of the quantifier.

The variable x is said to be bound by the quantifier.

Statements containing phrases as there exists, there is etc., are rephrased as "there is an x (in the domain) such that".

Translate "Some people like their tea iced" in first-order logic.

Let D be the set of all people.

Let $P(u)$ mean u likes their tea iced. In other words, P is defined as

$$P(u) = \begin{cases} 1, & \text{if } u \text{ likes their tea iced} \\ 0, & \text{otherwise} \end{cases}$$

Then the translation in first-order logic is $\exists x P(x)$.

The variable appearing in the quantifier is said to be bound.

For instance, in the formula $\forall x(P(x) \implies Q(x))$, the variable x appears three times and each time x is a bound variable.

Any variable that is not bound is said to be free.

We can consider the bound variables to be local to the scope of the quantifier just as parameters and locally declared variables in procedural programming languages are local to the procedure in which they are declared.

The analogy to procedural programming languages can be extended further if we consider the variable name in the quantifier as a declaration.

This analogy also suggests that, if several quantifiers use the same bound variable for quantification, then all these variables are local to their scope and they are therefore distinct.

$\forall x$ and $\exists x$ have to be treated like unary connectives.

The quantifiers are given a higher precedence than all binary connectives.

For instance, to translate "Everything is either living or dead", where the domain is all creatures, $P(u)$ means "u is living", and $Q(u)$ means "u is dead", we write

$$\forall x(P(x) \vee Q(x))$$

$\forall x P(x) \vee Q(x)$ means "Everything is living, or x is dead" (the first x is a bound variable, and the second x is a free variable).

The variable x in a quantifier is just a placeholder, and it can be replaced by any other variable symbol not appearing elsewhere in the formula.

For instance $\forall x P(x)$ and $\forall y P(y)$ mean the same thing (they are logically equivalent), so $\forall x P(x) \vee Q(x) \Eqv \forall y P(y) \vee Q(x)$.

Quantifying over a subset of the domain.

Sometimes, quantification is over a subset of the domain.

Suppose, for instance, that the domain is the set of all animals.

Consider the first statement "All dogs are mammals".

Since the quantifier should be restricted to dogs, one rephrases the statement as "If u is a dog, then u is a mammal", which leads to the formula

$$\forall x(\text{dog}(x) \implies \text{mammal}(x))$$

Conversely, the formula

$$\forall x(P(x) \implies Q(x))$$

can be interpreted as "All individuals (in the domain) with property P, also have property Q".

Consider now the statement "Some dogs are brown".

This means that there are some animals that are dogs and that are brown.

The statement "u is dog and u is brown" can be translated as $\text{dog}(u) \wedge \text{brown}(u)$.

These are some brown dogs can be translated as

$$\exists x(\text{dog}(x) \wedge \text{brown}(x))$$

Conversely, the formula

$$\exists x(P(x) \wedge Q(x))$$

can in general be interpreted as "Some individuals (in the domain) with property P, have also property Q".

If the universal quantifier applies only to a subset of the domain:

- First we have to define a property (relation) that describes that subset of the domain, and
- We then use the implication, $\implies$, to restrict the quantification to the subset of the domain consisting of the individuals with this property.

If we want to restrict application of the existential quantifier to a subset of the domain:

- First we have to define a property (relation) that describes that subset of the domain, and
- We then use the conjunction, $\wedge$, to restrict the quantification to the subset of the domain consisting of the individuals with this property.

What not to do!

The domain is the set of all animals.

"All dogs are mammals" cannot be translated using " $\wedge$ ", as in

$$\forall x(\text{dog}(x) \wedge \text{mammal}(x))$$

The above is a stronger statement, which translates as "Every animal is both a mammal and a dog" (a false statement).

"Some dogs are brown" cannot be translated using " $\implies$ ", as in

$$\exists x(\text{dog}(x) \implies \text{brown}(x))$$

The above is a weaker statement, which is vacuously true (even if no brown dogs would exist), by the definition of $\implies$, since there exists at least one animal which is not a dog.

Consider statements such as "Only dogs bark", where the domain is the set of all animals.

This must be first reworded as "It barks only if it is a dog", its equivalent "If it is not a dog, it does not bark", or its contrapositive equivalent "If it barks, then it is a dog".

The translation is thus: $\forall x(\text{barks}(x) \implies \text{dog}(x))$.

Negating formulas with $\forall$ quantifiers.

We will often want to consider the negation of a quantified formula.

Consider the negation of the statement:

Every student in the class has taken a course in calculus.

If the domain is the set of all students in this class, this statement can be translated as $\forall x P(x)$ where $P(u)$ is the statement "u has taken a course in calculus".

The negation of this statement is "It is not the case that every student in this class has taken a course in calculus".

This is equivalent to "There is a student in the class who has not taken a course in calculus, that is $\exists x \neg P(x)$".

In other words, $\neg \forall x P(x) \nVdash \exists x \neg P(x)$.

Consider the proposition.

There is a student in this class who has taken a course in calculus.

Which if the domain is the set of all students in this class, can be translated as $\exists x P(x)$ where $P(u)$ is the statement "u has taken a course in calculus".

The negation of this statement is "It is not the case that there is a student in this class who has taken a course in calculus".

This is equivalent to "Every student in this class has not taken calculus", that is, $\forall x \neg P(x)$.

In other words, $\neg \exists x P(x) \nVdash \forall x \neg P(x)$.

Assume the domain D is finite, $D = \{\alpha_1, \ldots, \alpha_n\}$

In this case, the universal quantifier is the same as conjunction: $\forall x R(x) = 1$ iff $R(\alpha_1) \wedge R(\alpha_2) \wedge \ldots \wedge R(\alpha_n) = 1$.

In this case, the existential quantifier is the same as disjunction: $\exists x R(x) = 1$ iff $R(\alpha_1) \vee R(\alpha_2) \vee \ldots \vee R(\alpha_n) = 1$.

The English statement Nobody is perfect also includes a quantifier, "nobody" which is the absence of an individual with a certain property.

In first-order logic, the fact that nobody has a property cannot be expressed directly.

If the domain is the set of all people, and if $P(u)$ means "u is perfect", then:

$\neg\exists x P(x)$ expresses "It is not the case that there is somebody who is perfect",

$\forall x \neg P(x)$ expresses "For everyone, it is not the case that they are perfect".

Thus, both $\neg\exists x P(x)$ and $\forall x \neg P(x)$ are correct translations for "Nobody is perfect".

Nested quantifiers

Example: Translate "There is somebody who knows everyone" into first-order logic, where the domain is the set of all people.

Use $K(u, v)$ to express "u knows v".

$\exists x \forall y K(x, y)$.

Let $Q(u, v)$ denote "$u + v = 0$". If the domain is the set of real numbers, what are the truth values of $\exists y \forall x Q(x, y)$ and $\forall x \exists y Q(x, y)$?

Solution: The first formula $\exists y \forall x Q(x, y)$ means "There is a real number v_0 such that for every real number u, we have $Q(u, v_0) = 1$". Since there is no real number v_0 such that $u + v_0 = 0$ for all real numbers u, this statement, and thus the first formula, is false.

The second formula $\forall x \exists y Q(x, y)$ means "For every real number u, there is a real number v_u such that $Q(u, v_u) = 1$".

Given real number u there is indeed a real number v_u such that $u + v_u = 0$, namely $u_u = -u$. Hence, this statement, and thus the second formula, is true.

The order in which quantifiers $\forall$ and $\exists$ appear matters!

In working with qualifications of more than one variable, it is sometimes helpful to think of them in terms of nested loops.

For example, to see whether $\forall x \forall y P(x, y)$ is true:

We consider $P(u, v)$ and loop through all the values for u and, for each u, we loop through all the values for v:

If we find that $P(u, v)$ is true for all values of u and v, then we have determined that $\forall x \forall y P(x, y)$ is true.

If we find that we ever hit a value u for which we hit a value v for which $P(u, v)$ is false, then we have shown that $\forall x \forall y P(x, y)$ is false.

9 Logic11: First-Order Logic Syntax

In propositional logic, formulas are recursively built starting from atoms (proposition symbols), by the five formation rules that describe the use of connectives.

In first-order logic, we add the capacity to refer to individuals, and their properties and relationships (rather than only to true/false propositions). This requires that formulas be more fine-grained, with:

- A specification of the basic individual, given by a domain.
- Terms, which are expressions referring to individuals in the domain.

- Atomic formulas (atoms) which are relations to combine terms, and are the simplest true/false formulas. Atoms play the same role here as proposition symbols do in propositional logic.
- Formulas which are recursively built starting from atomic formulas, by formation rules that describe the use of connectives and quantifiers.

There is no single "first-order language." Instead, there is a framework that combines logical elements with non-logical elements that are specific to the mathematical theory/structure we want to describe. In particular, we consider two different kinds of symbols:

Logical symbols: They have a fixed syntactic use and a fixed semantic meaning.

Non-logical symbols: These have a designated syntax, but their semantic meaning is not pre-defined.

$\mathcal{L}$ is the formal language of first-order logic. $\mathcal{L}$ may or may not be associated to a mathematical theory/structure.

The word term is used to refer to either an individual or a variable. More generally, a term is anything that can be used in place of an individual. Formally, we have:

Definition. Term($\mathcal{L}$) is the smallest class of expressions of $\mathcal{L}$ closed under the following formation rules:

1. Every individual symbol a is a term in Term($\mathcal{L}$)
2. Every free variable symbol u is a term in Term($\mathcal{L}$)
3. If $t_1, t_2, \ldots, t_n, n \geq 1$, are terms in Term($\mathcal{L}$), and f is an $n-$ary function symbol, then $f(t_1, \ldots, t_n)$ is a term in Term($\mathcal{L}$)

Terms containing no free variable symbols are called closed terms.

An "atomic formula", or "atom", of $\mathcal{L}$ is the simplest formula expressing a proposition, that is, a statement for which we can determine whether it is true or false. Formally, we have:

Definition: An expression of $\mathcal{L}$ is an atom or atomic formula in Atom($\mathcal{L}$) iff it is of one of the following two forms:

1. $F(t_1, t_2, \ldots, t_n), n \geq 1$, where F is an $n-$ary relation symbol and $t_1, t_2, \ldots, t_n$ are terms in Term($\mathcal{L}$)
2. $\approx (t_1, t_2)$, where t_1, t_2 are terms in Term($\mathcal{L}$).

For example, $\approx (+(s(0), s(0)), s(s(0)))$ are atoms.

Formulas of $\mathcal{L}$ are built recursively, starting from atoms, by defining formation rules that describe the use of connectives and quantifiers.

Definition: Form($\mathcal{L}$), the set of formulas of $\mathcal{L}$, is the smallest class of expressions of $\mathcal{L}$ closed under the following formation rules.

1. Every atom in Atom($\mathcal{L}$) is a formula in Form($\mathcal{L}$)
2. If A is a formula in Form($\mathcal{L}$), then $(\neg A)$ is a formula in Form($\mathcal{L}$)
3. If A, B are formulas in Form($\mathcal{L}$), then $(A \wedge B), (A \vee B), (A \implies B)$, and $(A \iff B)$ are formulas in Form($\mathcal{L}$)
4. If $A(u)$ is a formula in Form($\mathcal{L}$), where u is a free variable, and x is a variable not occurring in $A(u)$, then $\forall x A(x)$ and $\exists x A(x)$ are formulas in Form($\mathcal{L}$), where "$A(x)$" denotes the expression formed from $A(u)$ by replacing every occurrence of u by x.

Terms play a similar role in first-order logic as nouns and pronouns do in the English language: They are the expressions which can be interpreted as naming an object in the domain.

Terms are individual symbols, are free variable symbols, or function symbols having as arguments other terms.

Atoms (atomic formulas) are the simplest formulas in Form($\mathcal{L}$), and are built by using exactly one relation symbol applied to terms. They contain neither connectives nor quantifiers.

Formulas are expressions which can be built up from atoms, by using connective symbols and quantifier symbols.

A term or formula is said to be closed if it contains no free variables. A closed formula is also called a sentence.

Example:

1. For every integer x, there is an integer which is greater than x.
2. 500 is an integer.

–

3. There is an integer which is greater than 500.

Symbol	Meaning
$N(u)$	u is an integer
$G(u,v)$	u is greater than v
$\forall x$	for all x
$\exists y$	there exists y

The preceding logical argument can be formalized as:

1. $\forall x(N(x) \implies \exists y(N(y) \wedge G(y,x)))$
2. $N(500)$

–

3. $\exists y(N(y) \wedge G(y,500))$

Theorem: Any term in Term($\mathcal{L}$) is of exactly one of three forms: an individual symbol, a free variable symbol, or $f(t_1,\ldots,t_n)$ where $n \geq 1$, f is an $n-$ary function symbol and t_i are terms, $1 \leq i \leq n$; and in each case it is of that form in exactly one way.

Theorem: Any formula in Form($\mathcal{L}$) is of exactly one of eight forms: an atom (a single relation symbol applied to terms), $(\neg A), (A \wedge B), (A \vee B), (A \implies B), (A \iff B), \forall x A(x)$ or $\exists x A(x)$; and in each case it is of that form in exactly one way.

Definition: A sentence or a closed formula in Form($\mathcal{L}$) is a formula in Form($\mathcal{L}$) in which no free variable symbols occur. The set of sentences of $\mathcal{L}$ is denoted by Sent($\mathcal{L}$).

10 Logic12: First-Order Logic: Semantics

The language of first-order logic $\mathcal{L}$ is a purely syntactic object. The formulas in Form($\mathcal{L}$), however, are intended to express statements. This is the subject of semantics.

Semantics for propositional logic formulas in Form($\mathcal{L}^p$) is simple: A truth valuation assigns truth values to proposition symbols, and the truth value of a formula is based on the values of its proposition symbols and the "meaning" of connectives.

The language $\mathcal{L}$ includes more classes of symbols and therefore the "valuations" are more complex.

A valuation for $\mathcal{L}$ consists of an interpretation of its non-logical symbols, together with an assignment of values to its free variables.

Informally, a valuation for the first-order language $\mathcal{L}$ must contain sufficient information to determine whether each formula in Form($\mathcal{L}$) is true or false.

The logical symbols of $\mathcal{L}$ have a fixed semantics (meaning):

The connectives will be interpreted as in propositional logic.

The meaning of quantifiers has been explained intuitively (we will define them precisely).

The equality symbol $\approx$ denotes the relation "equal to".

The variable symbols will be interpreted as variables ranging over the domain.

Punctuation symbols serve just like ordinary punctuation.

A valuation consists of an interpretation plus an assignment.

An interpretation consists of a non-empty set of individuals (objects) called the domain. A specification, for each individual symbol, relation symbol, and function symbol, of the actual individuals, relations, and functions that each will denote.

An assignment assigns to each free variable a value in the domain.

We need this because, for formulas that contain free variables, in addition to an interpretation we must have an assignment of "values" (individuals in the domain) to the free variables in the formula, in order to determine if the formula is true or false.

Notation: We denote the meaning given by a valuation v to a symbol s by s^v.

One final step before the formal definitions: sometimes it is convenient to describe relations and functions in terms of sets.

Recall than an $n-$ary relation R on a set D an be thought of as a subset R of $D^n = D \times D \times \ldots \times D(n$ times), defined as

$$R = \{(a_1, a_2, \ldots, a_n) | a_i \in D, \text{ and } R(a_1, a_2, \ldots, a_n) = 1\}$$

For example, the equality relation on D is the subset of $D^2\{(x, y)|x, y \in D$ and $x = y\}$ or alternatively $\{(x, x)|x \in D\}$.

Recall that an $m-$ary function $f : D^m \longrightarrow D$ can be represented by the $(m + 1)-$ary relation

$$R_f = \{(x_1, x_2, \ldots, x_m, x_{m+1}) | f(x_1, x_2, \ldots, x_m) = x_{m+1}\}$$

Definition: A valuation for the first-order language $\mathcal{L}$ consists of:

A domain, often called D, which is a non-empty set, and a function, denoted by v, with the properties:

- For each individual symbol a, and free variable symbol u, we have that $a^v, u^v \in D$.
- For each $n-$ary relation symbol F, we have that F^v is an $n-$ary relation on D, that is, $F^v \subseteq D^n$. In particular, $\approx^v = \{(x, x)|x \in D\} \subseteq D^2$
- For each $m-$ary function symbol f, we have that f^v is a total $m-$ary function of D into D, that is, $f^v : D^m \longrightarrow D$.

A function is "total" if it is never undefined.

The definition of a domain requires it to be a non-empty set.

Consider the sentence (closed formula, no free variables)

$$\forall x(F(x) \vee H(x) \implies G(x))$$

Consider the valuation v_1 (since there are no free variables, the valuation coincides with the interpretation) defined as:

- The domain is D_1 is the set of all ships
- The symbol F is interpreted as the unary relation (over D_1) defined by $F^{v_1} = \{u|u$ is on fire $\}$, that is, $F^{v_1}(u)$ is 1 if u is on fire, and is 0 if u is not on fire.
- The symbol H is interpreted as the unary relation defined by $H^{v_1} = \{u|u$ has a hole $\}$.
- The symbol G is interpreted as the unary relation defined by $G^{v_1} = \{u|u$ sinks $\}$.

Under this interpretation, the formula says:

"Every ship that is on fire or has a hole sinks".

Individuals (constants) vs. free variables

Example: Let A_1 be $F(c)$ (where c is an individual symbol), and let A_2 be $F(u)$ (where u is a free variable).

Consider a valuation with domain $\mathbb{N}$ that interprets c as the number 4, and F as "is even".

Under this valuation, A_1 is evaluated as true, but A_2 remains undefined.

To give A_2 a value, we must also specify an assignment to the free variable u. For example, if we assign u value 3, then A_2 becomes false, but if we assign u value 4, then A_2 becomes true.

Thus, valuation = interpretation (of the individual symbols, relation symbols, function symbols) + assignment (to the free variable symbols).

Value of a term

Consider a valuation v. This fixes a domain, and the identities of a^v, F^v, and f^v for each non-logical symbol; it also fixes a value u^v for each free variable symbol.

Definition (Value of a term) The value of a term t under valuation v over a domain D, denoted by t^v, is defined recursively as follows:

1. If $t = a$ is an individual symbol a, then its value is $a^v \in D$. If $t = u$ is a free variable symbol u, then its value is $u^v \in D$.
2. If $t = f(t_1, t_2, \ldots, t_m), m \geq 1$, where f is an $m-$ary function symbol, and $t_i \in \text{Term}\ (\mathcal{L}), 1 \leq i \leq m$, then

$$f(t_1, t_2, \ldots, t_m)^v = f^v(t_1^v, t_2^v, \ldots, t_m^v)$$

Theorem: If v is a valuation over D, and $t \in \text{Term}(\mathcal{L})$ then $t^v \in D$.

To evaluate the truth value of a formula $\forall x A(x)$ (resp. $\exists x A(x)$), we should check whether $A(u)$ holds for all (resp. for some) values u in the domain.

How do we express this precisely?

Notation: For any valuation v, free variable u, and individual $d \in D$, we write

$$v(u/d)$$

to denote a valuation which is exactly the same as v except that $u^{v(u/d)} = d$.

That is, for each free variable w,

$$w^{v(u/d)} = \begin{cases} d \text{ if } w = u, \\ w^v \text{ otherwise} \end{cases}$$

Definition: (Value of a quantified formula) Let $\forall x A(x)$ and $\exists x A(x)$, and let u be a free variable not occurring in $A(x)$. The values of $\forall x A(x)$ and $\exists x A(x)$ under a valuation v with domain D are given by:

$$\forall x A(x)^v = \begin{cases} 1 \text{ if } A(u)^{v(u/d)} = 1 \text{ for every } d \in D \\ 0 \text{ otherwise} \end{cases} \quad \exists x A(x)^v = \begin{cases} 1 \text{ if } A(u)^{v(u/d)} = 1 \text{ for some} d \in D \\ 0 \text{ otherwise} \end{cases}$$

Definition: Let v be a valuation with domain D. The value of a formula in $\text{Form}(\mathcal{L})$ under v is defined recursively as:

1. $R(t_1, \ldots, t_n)^v = 1, n \geq 1 \iff (t_1^v, \ldots, t_n^v) \in R^v \subseteq D^n$,
2. $(\neg A)^v = 1 \iff A^v = 0$,
3. $(B \wedge C)^v = 1$ if both $B^v = 1$ and $C^v = 1$,
4. $(B \vee C)^v = 1$ if either $B^v = 1$ or $C^v = 1$ (or both),
5. $(B \implies C)^v = 1$ if either $B^v = 0$ or $C^v = 1$ (or both),

6. $(B \iff C)^v = 1$ $B^v = C^v$, Otherwise, in each case 1-6, the formula takes value 0.

Theorem: If v is a valuation over D and $A \in \text{Form}(\mathcal{L})$, then $A^v \in \{0, 1\}$.

Definition: A formula $A \in \text{Form}(\mathcal{L})$ is:

- Satisfiable if there exists a valuation v such that $A^v = 1$.
- (Universally) valid if for all valuations v we have $A^v = 1$.
- Unsatisfiable if it is not satisfiable, that is, if $A^v = 0$ for all valuations v.

Let Σ be a set of formulas in $\text{Form}(\mathcal{L})$, and v be a valuation over D. Define

$$\Sigma^v = \begin{cases} 1 & \text{if for every } B \in \Sigma, B^v = 1, \\ 0 & \text{otherwise} \end{cases}$$

Definition: A set $\Sigma \subseteq \text{Form}(\mathcal{L})$ is satisfiable if and only if there is some valuation v such that $\Sigma^v = 1$.

When $\Sigma^v = 1$ we say that v satisfies Σ, or that Σ is true under v.

Universally valid formulas in $\text{Form}(\mathcal{L})$ are the counterpart of tautologies in $\text{Form}(\mathcal{L}^p)$.

The similarities between them are obvious, but there is one important difference.

To decide whether or not a formula of $\text{Form}(\mathcal{L}^p)$ is a tautology, algorithms can be used (for instance the truth table method).

However, in order to know whether a formula of $\text{Form}(\mathcal{L})$ is universally valid, we have to consider all possible valuations over all possible domains, of all different sizes.

In case of an infinite domain, the procedure is in general not finite.

Given a valuation over an infinite domain, we do not have a method for evaluating the value of $\forall x B(x)$ or $\exists x B(x)$ in a finite number of steps, because it presuposes the values of $B(u)^{v(u/d)}$ for infinitely many d in D.

It is sometimes possible to decide for certain formulas in $\text{Form}(\mathcal{L})$ whether they are universally valid or not.

However, in the general case we have the following result.

Theorem (Church, 1936): There is no algorithm for deciding the (universal) validity or satisfiability of formulas in first-order logic.

In first-order logic, the variables range over individuals from the domain. The quantifiers are interpreted in the familiar way as "for all individuals of the domain", respectively "there exist some individual of the domain".

In second-order logic, we also allow as variables subsets of the domain and relations on the domain, as in:

- Every non-empty subset of natural number has a smallest element.

Here we have to take all subsets of the domain into consideration, and require variables and quantifiers for sets (not only for individuals in the domain). In second-order logic, quantifications over sets, relations, functions are allowed.

In higher-order logic, variables and quantifiers for sets of sets, sets of sets of sets, etc. are also allowed.

11 Logic13: Logical Consequence

Logical consequence in first-order logic, which are the counterparts of tautological consequences in propositional logic, involve semantics.

The notation $\vDash$ for tautological consequences is also used for logical consequences.

Definition: Suppose Σ is a set of formulas in $\text{Form}(\mathcal{L})$ and A is a formula in $\text{Form}(\mathcal{L})$. A is a logical consequence of Σ, written as $\Sigma \vDash A$, iff for any valuation v with $\Sigma^v = 1$, we have $A^v = 1$.

The notations $\nvDash$ and $\nvDash$ are used in the same sense as in propositional logic.

Two formulas are called logically equivalent (or equivalent for short, if no confusion will arise) iff $A \nvDash B$ holds.

Prove that

$$\forall x \neg A(x) \vDash \neg \exists x A(x)$$

Proof (by contradiction): Suppose the contrary, that is, suppose that there is some valuation v over a domain D such that:

$$(1) \quad (\forall x \neg A(x))^v = 1$$
$$(2) \quad (\neg \exists x A(x))^v = 0$$

By "negating equation (2)", it follows (from the semantics of first-order logic) that

$$(3) \quad (\exists x A(x))^v = 1$$

This implies that (using the simplified notation),

$$(4) \quad A(d)^v = 1 \text{ for some } d \in D$$

"Negating equation (4)" (namely, $A(d)^v = 1$) yields

$$(5) \quad (\neg A(d))^v = 0$$

On the other hand, recall (1), which states $(\forall x \neg A(x))^v = 1$. This implies that, in particular, for the individual $d \in D$ that we identified in (4), we have

$$(6) \quad (\neg A(d))^v = 1$$

We have reached a contradiction ((6) contradicts (5)), therefore our assumption was false.

Since our assumption (that the argument was invalid) was false, its opposite is true, that is, the argument is valid (or sound, correct).

Note: Similarly, we can prove $\neg \exists x A(x) \vDash \forall x \neg A(x)$, and therefore we have $\neg \exists x A(x) \nvDash \forall x \neg A(x)$.

Prove that $\forall x (A(x) \implies B(x)) \vDash \forall x A(x) \implies \forall x B(x)$.

Proof: Assume that $\forall x (A(x) \implies B(x)) \nvDash \forall x A(x) \implies \forall x B(x)$. This implies that there exists a valuation v over a domain D such that:

$$(1) \quad (\forall x (A(x) \implies B(x)))^v = 1$$
$$(2) \quad (\forall x A(x) \implies \forall x B(x))^v = 0$$

(2) implies (3): $(\forall x A(x))^v = 1$, and (4): $(\forall x B(x))^v = 0$.

If we negate equation (4), we obtain (5): $(\exists x \neg B(x))^v = 1$.

(5) implies the existence of an individual $d \in D$ such that $(\neg B(d))^v = 1$, that is $B(d)^v = 0$.

Since $(\forall x A(x))^v = 1$, we have that in particular, $A(d)^v = 1$.

From $B(d)^v = 0$ and $A(d)^v = 1$, we have $(A(d) \implies B(d))^v = 0$, which implies $(\forall x (A(x) \implies B(x)))^v = 0$. This contradicts (1), hence the argument is valid.

Prove that $\forall x A(x) \implies \forall x B(x) \nvDash \forall x (A(x) \implies B(x))$.

Proof: It suffices to find a single counter-example (a valuation v over a domain D that makes the premise true but the conclusion false). Consider $D = \{a, b\}$ and the relations A and B defined as

Under this valuation, we have

	$A(u)^v$	$B(u)^v$
a	1	0
b	0	1

- $(\forall x A(x))^v = 0$ since $A(b)^v = 0$
- $(\forall x B(x))^b = 0$ since $B(a)^v = 0$

Thus, we have (1): $(\forall x A(x) \implies \forall x B(x))^v = 1$. On the other hand, (2): $(\forall x(A(x) \implies B(x)))^v = 0$, because $(A(a) \implies B(a))^v = 0$. From (1) and (2), we see that the above valuation v over $D = \{a, b\}$ makes the premise true but the conclusion false. This implies that the argument is invalid.

Recall that a universally valid formula of Form($\mathcal{L}$) is a formula that is satisfied by every possible valuation.

For any formula A in Form($\mathcal{L}$), one has $\emptyset \vDash A$ if and only if A is universally valid.

To demonstrate that a formula A is universally valid, we have to show that $\emptyset \vDash A$.

Since $\emptyset$ is vacuously satisfied by any valuation, to prove that a formula A is universally valid we have to show that there is no valuation under which A is false.

Show that $\forall x F(x) \vee \forall x G(x) \implies \forall x(F(x) \vee G(x))$ is universally valid, that is, prove $\emptyset \vDash \forall x F(x) \vee \forall x G(x) \implies \forall x(F(x) \vee G(x))$.

Proof: Assume that $\emptyset \nvDash \forall x F(x) \vee \forall x G(x) \implies \forall x(F(x) \vee G(x))$.

This implies that there exists a valuation v over a domain D such that

$$(\forall x F(x) \vee \forall x G(x) \implies \forall x(F(x) \vee G(x)))^v = 0$$

This further implies

$$\begin{aligned} &(1) \quad (\forall x F(x) \vee \forall x G(x))^v = 1 \\ &(2) \quad (\forall x(F(x) \vee G(x)))^v = 0 \end{aligned}$$

Negating equation (2) results in $(\exists x(\neg F(x) \wedge \neg G(x)))^v = 1$. This implies that there exists an individual $d \in D$ such that $(\neg F(d) \wedge \neg G(d))^v = 1$, further yielding $F(d)^v = G(d)^v = 0$.

$F(d)^v = 0$ implies (3): $(\forall x F(x))^v = 0$, and $G(d)^v = 0$ implies (4): $(\forall x G(x))^v = 0$; (3) and (4) imply $(\forall x F(x) \vee \forall x G(x))^v = 0$, which contradicts (1). Hence, the formula is universally valid.

Prove that the formula $\exists x P(x) \implies \forall x P(x)$ is not universally valid,

$$\emptyset \nvDash \exists x P(x) \implies \forall x P(x)$$

Proof: To prove that the formula is not universally valid, it suffices to find a valuation that makes the antecedent $\exists x P(x)$ true and the consequent $\forall x P(x)$ false.

Construct the valuation v over domain $D = \{a, b\}$ defined by $P(a)^v = 1$ and $P(b)^v = 0$.

Then $(\exists x P(x))^v = 1$ (since $P(a)^v = 1$), while $(\forall x P(x))^v = 0$ (since $P(b)^v = 0$).

This implies that $(\exists x P(x) \implies \forall x P(x))^v = 0$, which further implies that the formula is not universally valid.

Can we always determine whether a formula is universally valid?

No. The problem of proving whether or not a formula in Form($\mathcal{L}$) is universally valid is undecidable (Church); that is, there is no generally applicable algorithm that, given an arbitrary formula in Form($\mathcal{L}$) as input, can always determine whether or not the formula is universally valid.

This does not mean that we can never determine that a particular formula is universally valid. In fact, there are methods that work in many particular cases.

For instance, first-order logic formulas that arise from propositional logic tautologies, such as $\forall x P(x) \vee \neg(\forall x P(x))$ (arising from $p \vee \neg p$), can be proved to be universally valid.

For other formulas, such as the ones in the previous examples, we are able to prove whether or not they are universally valid.

However there is no general-purpose algorithm that provides an answer in all cases.

Theorem: (Replaceability of equivalent formulas in first-order logic). Let A be a formula in $\text{Form}(\mathcal{L})$ which contains a subformula $B \in \text{Form}(\mathcal{L})$. Assume that $B \Bumpeq C$, and let A' be the formula obtained by simultaneously replacing in A some (but not necessarily all) occurrences of the formula B by formula C. Then $A' \Bumpeq A$.

Theorem: (Duality in first-order logic). Suppose A is a formula in $\text{Form}(\mathcal{L})$ composed only of atoms in $\text{Atom}(\mathcal{L})$, the connectives $\neg, \vee, \wedge$ and the quantifiers $\forall$ and $\exists$, by the formation rules concerned. Suppose $\Delta(A)$ results from A by simultaneously exchanging connectives $\wedge$ for $\vee$, quantifiers $\forall$ for $\exists$, and each atom for its negation. Then $\neg A \Bumpeq \Delta(A)$.

12 Logic14: First-order-Logic Formal Deduction

The goal of formal deducibility was to define a calculus of reasoning.

We defined a self-contained formal system of reasoning based on 11 rules of formal deduction.

The system of formal deduction gives syntactic procedures to construct new correct theorems from already proven ones. In such a formal deduction system, the correctness of the formal proof of a theorem can be checked mechanically/automatically.

The ultimate goal for a formal deduction system is to be able to prove formally, everything that is correct semantically.

The formal deducibility in first-order logic is an extension of that in propositional logic.

The 11 rules of formal deduction for propositional logic are included in formal deduction for first-order logic, but the formulas occurring in them are now formulas in $\text{Form}(\mathcal{L})$.

We also include 6 additional rules of formal deduction concerning the quantifiers, and the equality symbol.

(12)	$(\forall -)$	If $\Sigma \vdash \forall x A(x)$ is a theorem then $\Sigma \vdash A(t)$ where t is any term, is a theorem
(13)	$(\forall +)$	If $\Sigma \vdash A(u)$ is a theorem and u does not occur in Σ then $\Sigma \vdash \forall x A(x)$ is a theorem
(14)	$(\exists -)$	If $\Sigma, A(u) \vdash B$ is a theorem and u does not occur in Σ or in B then $\Sigma, \exists x A(x) \vdash B$ is a theorem
(15)	$(\exists +)$	If $\Sigma \vdash A(t)$ is a theorem then $\Sigma \vdash \exists x A(x)$ is a theorem, where $A(x)$ results from $A(t)$ by replacing some (not necessarily all) occurrences of t by x.
(16)	$(\approx -)$	If $\Sigma \vdash A(t_1)$ is a theorem and $\Sigma \vdash t_1 \approx t_2$ is a theorem then $\Sigma \vdash A(t_2)$ is a theorem, where $A(t_2)$ results from $A(t_1)$ by replacing some (not necessarily all) occurrences of t_1 by t_2
(17)	$(\approx +)$	$\emptyset \vdash u \approx u$ is a theorem

The additional rules of formal deduction for first-order logic are called:

- $\forall-$elimination for $(\forall -)$; $\forall-$introduction for $(\forall +)$;

- $\exists-$elimination for $(\exists-)$; $\exists-$introduction for $(\exists+)$;
- $\approx -$elimination for $(\approx -)$; $\approx -$introduction for $(\approx +)$;

Note: In these rules, u is a free variable, and t, t_1, t_2 are terms. $\approx$ is an alternative notation for the usual equality relation "=".

In $(\forall-)$, the formula $A(t)$ results from $A(x)$ by substituting all occurrences of x by t, and similarly for $(\forall+)$ and $(\exists-)$.

In $(\exists+)$, another kind of replacement is employed, which should be distinguished from substitution. This kind of replacement allows us to either substitute all occurrences of t by x (as usual), or replace only some of the occurrences of t by x (and leave the rest as t), as needed. The case of $(\approx -)$ is similar.

The u in $(\forall-)$ and $(\exists-)$may be replaced by t (any term). This extends the range of application of these two rules, as the set of terms strictly contains the set of free variables. However, since the formulations - as defined - are sufficient, the replacement of u by t in the definition of these two rules is not necessary.

The conditions u not occurring in Σ in $(\forall+)$, and u not occurring in Σ or B in $(\exists-)$ are essential.

Explanation for $(\forall+)$

$(\forall+)$ If $\Sigma \vdash A(u)$ and u does not occur in Σ then $\Sigma \vdash \forall x A(x)$.

The rule $(\forall+)$ means intuitively that from

"Any element of a set has a certain property." we can deduce that "Every element of the set has this property".

Example: (Perpendicular Bisector Theorem) Every point on the perpendicular bisector L has a segment AB has the property that it is equidistant from A and B.

Proof: It is sufficient to prove the theorem for any point P on the perpendicular bisector L. In other words, the proof would start with "Let P be an (arbitrary) point on L. We have to show that $|PA| = |PB|$." Etc.

At the end of the proof, from the statement

"Any point P on L is equidistant from A and B." we deduce the statement "Every point P on L is equidistant from A and B."

The reasoning above is only sensible if the "any" means an arbitrary element, with no restriction whatsoever.

If "any" means a particular element, such a reasoning would be nonsense.

Here, the arbitrariness of P means that the choice of P is independent of the premises (hypotheses) of the theorem.

This is expressed syntactically in $(\forall+)$ by "u not occurring in Σ" (where u expresses P, and Σ the premises of the theorem).

The explanation is similar for the case of $(\exists-)$. The value u that satisfies $A(u)$ is fixed but unknown, and thus u as a symbol must be completely independent of all other variables in all formulas.

Comments on $\forall$- elimination.

Given $\forall x A(x)$ we should be able to derive $A(t)$ for any term t.

For instance, if the domain is all people in a given house, and $A(u)$ stands for u is sleeping, then $\forall x A(x)$ means "Everyone in the house is sleeping".

If Dan is in the house, from this statement we can derive that Dan is sleeping.

This is the type of valid reasoning that the $\forall$-elimination rule is intended to formalize.

Note: t can be an individual symbol, a free variable symbol, or a function symbol applied to terms.

Comments on $\forall$-introduction

If u does not appear as a free variable in any premise, one can "generalize" over u.

If u would appear free in any premise, then u would always refer to the same individual, and would be "fixed" in this sense. For example, if u would appear in a premise, e.g., in $P(u)$, then u would only refer to the particular individual that makes $P(u)$ true, and would not be "arbitrary".

If u is fixed, one cannot generalize over u. Generalizations from one particular individual towards the entire population are unsound.

If, on the other hand, u does not appear in any premise as a free variable, then u is assumed to stand for anyone, and the generalization $(\forall+)$ may be applied without restriction.

Comments on $\exists$-introduction.

If Aunt Cordelia is 100 years old, then there is obviously someone who is 100 years old.

If there is any term t for which $A(t)$ holds, then one can conclude that some x satisfies $A(x)$.

This is the type of valid reasoning that the $\exists$-introduction rule is intended to formalize.

Definition: Suppose Σ is a set of formulas in $\text{Form}(\mathcal{L})$ and A is a formula in $\text{Form}(\mathcal{L})$. We say that A is formally deducible from Σ in first-order logic iff

$$\Sigma \vdash A$$

can be generated by the 17 rules of formal deduction.

Example

Prove that

$$\neg\forall x A(x) \dashv\vdash \exists x \neg A(x)$$

Solution: Prove the direct implication, $\neg\forall x A(x) \vdash \exists x \neg A(x)$.

1	$\neg A(u) \vdash \neg A(u)$	Ref
2	$\neg A(u) \vdash \exists x \neg A(x)$	$1, (\exists+)$
3	$\neg\exists x \neg A(x) \vdash A(u)$	2, Flip-flop rule, $\neg\neg A \dashv\vdash A$, Repl
4	$\neg\exists x \neg A(x) \vdash \forall x A(x)$	$3, (\forall+), u$ does not occur in premises
5	$\neg\forall x A(x) \vdash \exists x \neg A(x)$	4, Flip-flop rule,$\neg\neg A \dashv\vdash A$, Repl

We can now prove the converse, that is, $\exists x \neg A(x) \vdash \neg\forall x A(x)$.

1	$\forall x A(x) \vdash \forall x A(x)$	Ref
2	$\forall x A(x) \vdash A(u)$	$1, (\forall-)$
3	$\neg A(u) \vdash \neg\forall x A(x)$	2, Flip-flop rule
4	$\exists x \neg A(x) \vdash \neg\forall x A(x)$	$3, (\exists-), u$ does not occur elsewhere

Theorem: (Replaceability of equivalent formulas in formal deduction for first-order logic) Let $A, B, C \in \text{Form}(\mathcal{L})$ with $B \dashv\vdash C$. Let A' result from A by substituting some (not necessarily all) occurrences of B by C. Then $A' \dashv\vdash A$.

Theorem: (Duality in formal deduction for first-order logic) Suppose A is a formula composed of atoms in $\text{Atom}(\mathcal{L})$, the connectives $\neg, \vee, \wedge$, and the two quantifiers $\exists$ and $\forall$, by the formation rules concerned. Let $\Delta(A)$ be the formula obtained from A by exchanging $\vee$ and $\wedge$, $\exists$ and $\forall$, and negating all atoms. Then $\Delta(A) \dashv\vdash \neg A$.

Theorem: (Soundness and Completeness)

Let $\Sigma \subseteq \text{Form}(\mathcal{L})$ and $A \in \text{Form}(\mathcal{L})$. Then $\Sigma \vDash A$ if and only if $\Sigma \vdash A$.

The theorem states that the formal deduction system for first-order logic defined by the 17 rules of formal deduction is:

- Sound ($\Sigma \vdash A$ implies $\Sigma \vDash A$), and
- Complete ($\Sigma \vDash$ implies $\Sigma \vdash A$)

Proof strategies for formal deduction

One strategy for figuring out the high-level idea for a proof is to "ignore" the quantifiers, and imagine what the proof would look like if all formulas were in propositional logic.

After we have an idea of the general shape of the proof, we:

- Remove quantifiers (e.g., by using $(\forall-)$ or $(\exists-)$)
- Carry on with the proof with the formulas in propositional logic
- Introduce quantifiers, as needed (using $(\forall+)$, or $(\exists+)$)

If one of the premises in Σ is an existentially quantified formula, e.g., $\exists x A(x)$, the way to remove this $\exists$, is to:

- Replace the premise $\exists x A(x)$ in Σ by $A(u)$, resulting in Σ',
- Carry on with the proof, with the modified set of premises Σ',
- Use $(\exists-)$ to reintroduce this $\exists$ back, at the very end, when u does not appear anymore in the other premises or the conclusion (this step returns the premise set from the modified Σ' to the original premise set Σ).

13 Logic15: First-order-Logic Resolution

For resolution in first-order logic, and for other purposes, it is often more convenient to deal with formulas in which all quantifiers have been moved to the front of the formula. These types of formulas are said to be in prenex normal form.

Definition: A formula is in prenex normal form if it is of the form

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n B$$

where $n \geq 1, Q_i$ is $\forall$ or $\exists$, for $1 \leq i \leq n$, and the expression B is quantifier free.

The string $Q_1 x_1 Q_2 x_2 \dots Q_n x_n$ is called the prefix and B is called the matrix.

Convention: A formula with no quantifiers ($n = 0$) is regarded as a trivial case of a prenex normal form.

Algorithm for converting a formula in Form($\mathcal{L}$) into prenex normal form.

Any formula in Form($\mathcal{L}$) is logically equivalent to (and can be converted into) a formula in prenex normal form (PNF). To find its logically equivalent formula in PNF, the following steps are needed:

- Eliminate all occurrences of $\implies$ and $\iff$ from the formula.
- "Move all negations inward" such that, in the end, negations only appear as part of literals
- Standardize the variables apart, when necessary.
- The prenex normal form can now be obtained by "moving" all quantifiers to the front of the formula.

In the following, we will describe the logical equivalences that can be used to accomplish the steps above.

To accomplish Step 1 (eliminate $\implies$, $\iff$), make use of the following logical equivalences:

- $A \implies B \dashv\vdash \neg A \vee B$
- $A \iff B \dashv\vdash (\neg A \vee B) \wedge (A \vee \neg B)$
- $A \iff B \dashv\vdash (A \wedge B) \vee (\neg A \wedge \neg B)$

To accomplish step 2 (move all negations inward, such that negations only appear as parts of literals), use the logical equivalences:

- De Morgan's Laws

- Double negation: $\neg\neg \vDash A$
- $\neg\exists x A(x) \dashv\vDash \forall x \neg A(x)$
- $\neg\forall x A(x) \dashv\vDash \exists x \neg A(x)$

Step 3 (Standardize)

Recall that the symbol denoting a bound variable is just a place holder, and two occurrences of a symbol x in a formula do not necessarily refer to the same bound variable. For example, in $\forall x(A(x) \vee B(x)) \vee \exists x C(x)$, the first two occurrences of x in the formula refer to the variable in the scope of $\forall x$, while the last occurrence of x refers to a distinct variable, in the scope of $\exists x$.

Renaming the variables in a formula such that distinct bound variables (variables bound by distinct quantifiers) have distinct names called standardizing the variables apart.

To accomplish Step 3, we use the following theorem, which allows us to rename bound variables.

Theorem (Replaceability of bound variable symbols) Let A be a formula in $\text{Form}(\mathcal{L})$. Suppose that A' results from A by replacing in A some (not necessarily all) occurrences of $QxB(x)$ by $QyB(y)$, where $Q \in \{\forall, \exists\}$. Then $A \dashv\vDash A'$ and $A \vdash A'$.

Example:

$$\forall x(P(x) \implies Q(x)) \wedge \exists x Q(x) \wedge \exists z P(z) \wedge \exists z(Q(z) \implies R(z))$$

becomes

$$\forall y(P(y) \implies Q(y)) \wedge \exists x_1 Q(x_1) \wedge \exists x_2 P(x_2) \wedge \exists x_3(Q(x_3) \implies R(x_3))$$

Step 4 (move all quantifiers to the front)

To accomplish Step 4, make use of the following logical equivalences:

- $A \wedge \exists x B(x) \dashv\vDash \exists x(A \wedge B(x)), x$ not occurring in A.
- $A \wedge \forall x B(x) \dashv\vDash \forall x(A \wedge B(x)), x$ not occurring in A.
- $A \vee \exists x B(x) \dashv\vDash \exists x(A \vee B(x)), x$ not occurring in A.
- $A \vee \forall x B(x) \dashv\vDash \forall x(A \vee B(x)), x$ not occurring in A.

These equivalences essentially show that if a formula A has a truth value that does not depend on x, then one is allowed to quantify it over x, using any quantifier.

Example: Find the prenex normal form of

$$\forall x(\exists y R(x,y) \wedge \forall y \neg S(x,y) \implies \neg\exists y \neg Q(x,y))$$

Solution:

$$\begin{aligned}
&\forall x(\neg(\exists y R(x,y) \wedge \forall y \neg S(x,y)) \vee \neg\exists y \neg Q(x,y)) \\
&\forall x(\forall y \neg R(x,y) \vee \exists y S(x,y) \vee \forall y Q(x,y)) \\
&\forall x(\forall y_1 \neg R(x,y_1) \vee \exists y_2 S(x,y_2) \vee \forall y_3 Q(x,y_3)) \\
&\forall x \forall y_1 \exists y_2 \forall y_3(\neg R(x,y_1) \vee S(x,y_2) \vee Q(x,y_3))
\end{aligned}$$

Definition: A sentence (formula without free variables) $A \in \text{Sent}(\mathcal{L})$ is said to be in $\exists-$free prenex normal form if it is in prenex normal form and does not contain existential quantifier symbols.

Consider a sentence of the form $\forall x_1 \forall x_2 \ldots \forall x_n \exists y A$ where $n \geq 0$, and A is an expression, possible involving other quantifiers.

- Note that $\exists y A$ generates at least one individual for each $n-$tuple $(a_1, a_2, \ldots, a_n)$ in the domain.
- In other words, the individual generated by $\exists y A$ is a function of $x_1, \ldots x_n$, which can be expressed by using $f(x_1, x_2, \ldots, x_n)$

- The function f is called a Skolem function.
- The function symbol for a Skolem function is a new function symbol, which must not occur anywhere in A.

Skolem functions allow one to remove all existential quantifiers. The skolemized version of $\forall x_1 \forall x_2 \dots \forall x_n \exists y A$ is $\forall x_1 \forall x_2 \dots \forall x_n A'$ where $n \geq 0$, and A' is the expression obtained from A by substituting each occurrence of y be $f(x_1, x_2, \dots, x_n)$.

Example: Let the domain be $\mathbb{Z}$, and consider $\forall x \exists y (x + y = 0)$. Each instance of x, say $x = d, d \in \mathbb{Z}$, generates a corresponding $y = -d$ that makes the formula true. If we define $f(x) = -x$, we have that the skolemized version of the formula is $\forall x (x + f(x) = 0)$.

More generally, in $\forall x \exists y P(x, y)$, one has a different value of y generated, for each value of x. The skolemized version of $\forall x \exists y P(x, y)$ is $\forall x P(x, g(x))$. Here, $g(x)$ is the Skolem function "generating" a value $y = g(x)$, for each value of x.

Note that the sentence obtained by using Skolem functions is not, in general, logically equivalent to the original sentence. This happens because it is possible that there is more than one individual arising from the existential quantifier. However, for our purposes, it is irrelevant how many individuals satisfy A in $\exists y A$, as long as there exists at least one individual.

It is convenient to consider individual symbols as functions of zero arguments. With this convention, the skolemized sentence $(*)$ remains valid even if an existential quantifier is not preceded by any universal quantifier $(n = 0)$.

For any sentence in $\text{Sent}(\mathcal{L})$ we can generate a sentence in $\exists-$free prenex normal form by using the following algorithm.

Step 1: Transform the input sentence $A_0 \in \text{Sent}(\mathcal{L})$ into a logically equivalent sentence A_1 in prenex normal form. Set $i = 1$.

Step 2: Repeat until all existential quantifiers are removed.

- Assume A_i is of the form $A_i = \forall x_1 \forall x_2 \dots \forall x_n \exists y A$ where A is an expression, possibly involving quantifiers.
- If $n = 0$, then A_i is of the form $\exists y A$. Then $A_{i+1} = A'$, where A' is obtained from A by replacing all occurrences of y by the individual symbol c, where c is a symbol not occurring in A_i.
- If $n > 0, A_{i+1} = \forall x_1 \forall x_2 \dots \forall x_n A'$, where A' is the expression obtained from A by replacing all occurrences of y by $f(x_1, x_2, \dots, x_n)$, where f is a new function symbol.
- Increase i by 1.

Example: Transform the following sentence into $\exists-$free prenex normal form:

$$\exists x \forall y \forall z \exists s P(x, y, z, s)$$

Becomes

$$A = \forall y \forall z P(a, y, z, g(y, z))$$

which is a formula in $\exists-$free prenex normal form.

Theorem: Given a sentence F in $\text{Sent}(\mathcal{L})$, there is an effective procedure for finding an $\exists-$free prenex normal form formula F' such that F is satisfiable iff F' is satisfiable.

Notational convention:

- After all the existential quantifiers have been eliminated through Skolem functions, and the formula is in $\exists-$free prenex normal form, it is customary to "drop" the universal quantifiers.
- For instance, $\forall y \forall z P(a, y, z, g(y, z))$ becomes $P(a, y, z, g(y, z))$.
- The above conventional notation means that, when working with formulas in $\exists-$free prenex normal form (e.g., in resolution for first-order logic), all variables are implicitly considered to be universally quantified.

From formulas in first-order logic to clauses.

Theorem: Given a sentence F in $\exists-$free prenex normal form, one can effectively construct a finite set C_F of disjunctive clauses such that F is satisfiable iff the set C_F of clauses is satisfiable.

Example: Construct the set of clauses C_F for

$$F = \forall x \forall y \forall z (R(x,y) \implies (R(x,z) \land R(z,y)))$$

First we put the matrix of F in Conjunctive Normal Form

$$\begin{aligned} R(x,y) &\implies (R(x,z) \land R(z,y)) \\ \equiv \neg R(x,y) &\lor (R(x,z) \land R(z,y)) \\ \equiv (\neg R(x,y) &\lor R(x,z)) \land (\neg R(x,y) \lor R(z,y)) \end{aligned}$$

Now we can read off the clauses from the conjuncts, that is, $C_F = \{\neg R(x,y) \lor R(x,z), \neg R(x,y) \lor R(z,y)\}$

Valid argument & satisfiability of clause set

Theorem: Let Σ be a set of sentences, and A be a sentence. The argument $\Sigma \vDash A$ is valid iff the set

$$\left(\bigcup_{F \in \Sigma} C_F \right) \cup C_{\neg A}$$

is not satisfiable.

In other words, an argument in first-order logic is valid (the conclusion is a logical consequence of the premises) iff the set of clauses consisting of the union of:

- $\bigcup_{F \in \Sigma} C_F$: The sets of clauses obtained from each premise F in Σ, and
- $C_{\neg A}$: The set of clauses generated by the negation of the conclusion A

is not satisfiable.

Let the set of premises of an argument in first-order logic be

$$\Sigma = \{\forall x R(x,x), \forall x \forall y (R(x,y) \implies R(y,x)), \forall x \forall y \forall z ((R(x,y) \land R(y,z)) \implies R(x,z))\}$$

and the conclusion of the argument be

$$A = \forall x \forall y (\neg R(x,y) \implies \forall s (R(x,s) \implies \neg R(y,s)))$$

Find the set of clauses $C_{\Sigma, \neg A}$ that is not satisfiable iff the argument $\Sigma \vDash A$ is valid.

Solution:

The negation of the conclusion is

$$\begin{aligned} \neg A = \neg \forall x \forall y (\neg R(x,y) &\implies \forall s (R(x,s) \implies \neg R(y,s))) \\ \equiv \neg \forall x \forall y (R(x,y) &\lor \forall s (\neg R(x,s) \lor \neg R(y,s))) \\ \equiv \exists x \exists y (\neg R(x,y) &\land \exists s (R(x,s) \land R(y,s))) \end{aligned}$$

Putting $\neg A$ in prenex normal form gives

$$\exists x \exists y \exists s (\neg R(x,y) \land R(x,s) \land R(y,s))$$

For $\neg A$ obtain the formula in $\exists-$free prenex normal form

$$\neg R(a,b) \land R(a,c) \land R(b,c)$$

The premises are already formulas in $\exists-$free prenex normal form.

Thus, the set of clauses $C_{\Sigma,\neg A}$ consists of

$$
\begin{array}{ll}
R(x,x) & (\text{from } \forall x R(x,x)) \\
\neg R(x,y) \vee R(y,x) & (\text{from } \forall x \forall y (R(x,y) \implies R(y,x))) \\
\neg R(x,y) \vee \neg R(y,z) \vee R(x,z) & (\text{from } \forall x \forall y \forall z ((R(x,y) \wedge R(y,z)) \implies R(x,z))) \\
\neg R(a,b) & \text{from negation of conclusion} \\
R(a,c) & \text{from negation of conclusion} \\
R(b,c) & \text{from negation of conclusion}
\end{array}
$$

The set of six clauses $C_{\Sigma,\neg A}$ is not satisfiable iff the argument $\Sigma \vDash A$ is valid.

Thus, if we would want to prove that the original argument were valid, we would have to show that the set of clauses $C_{\Sigma,\neg A}$ is not satisfiable.

For this, we will use resolution for first-order logic.

A last ingredient for resolution: Unification

In resolution we aim to reach the empty clause $\perp$ (symbolizing a contradiction, that is, a formula that is not satisfiable).

In propositional logic, it is impossible to derive a contradiction from a set of formulas, unless the same variable occurs more than once.

For instance, there is no way to derive a contradiction from the two formulas $p \wedge q \vee r$ and $\neg s$. The two formulas do not share variables, and the truth of the first has no bearing on the truth of the second.

Similarly, in first-order logic, one cannot derive a contradiction from two formulas A and B share complementary literals.

To obtain complementary literals, we may have to use a procedure called unification.

Definition: An instantiation is an assignment to a variable x_i of a quasi-term t_i' (defined as either an individual symbol, or a variable symbol, or a function symbol applied to individual symbols or variable symbols). We write $x_i \coloneqq t_i'$.

Definition: Two formulas in first-order logic are said to unify if there are instantiations that make the formulas in equation identical. The act of unifying is called unification. The instantiation that unified the formulas in question is called a unifier.

Note: Unification works because in resolution all variables are implicitly universally quantified. Thus, the steps that lead to unification are either variable renamings, or applications of universal instantiation, $(\forall -)$.

Example: Assume that $Q(a,y,z)$ and $Q(y,b,c)$ are expressions appearing on different lines in a resolution proof. Show that the two expressions unify and give a unifier.

Solution: Since y in $Q(a,y,z)$ is a different variable than y in $Q(y,b,c)$, rename y in the second formula to become y_1. This means that one must unify $Q(a,y,z)$ with $Q(y_1,b,c)$. An instance of $Q(a,y,z)$ is $Q(a,b,c)$ (given by $\coloneqq b, z \coloneqq c$), and an instance of $Q(y_1,b,c)$ is $Q(a,b,c)$ (given by $y_1 \coloneqq a$). Since these two instances are identical, $Q(a,y,z)$ and $Q(y,b,c)$ unify, with unifier is $y_1 \coloneqq a, y \coloneqq b, z \coloneqq c$.

Recall that resolution can only be applied to expressions that contain complementary literals.

The idea is now to create complementary literals by means of unification, and then to determine the resolvent.

Example: Find the resolvent of the following two clauses:

$$
\begin{array}{r}
G(a,x,y) \vee H(y,x) \vee D(z) \\
\neg G(x,c,y) \vee H(f(x),b) \vee E(a)
\end{array}
$$

Here a, b, c are individual symbols and x, y, z are variable symbols.

Solution: To obtain two complementary literals, we unify $G(a, x, y)$ in the first clause with $G(x, c, y)$ in the second clause.

Since x, y, z in the 1st clause are (implicitly) universally quantified, we can instantiate these variables by any quasi-term.

In particular, we can set $x \coloneqq c$, which yields

$$(*) \quad G(a, c, y) \lor H(y, c) \lor D(z)$$

Similarly, one can instantiate the variables in the 2nd clause by any quasi-term. We set $x \coloneqq a$ and obtain

$$(**) \quad \neg G(a, c, y) \lor H(f(a), b) \lor E(a)$$

Once the unification is done, the resolvent of the two new clauses

$$\begin{aligned} (*) \quad & G(a, c, y) \lor H(y, c) \lor D(z) \\ (**) \quad & \neg G(a, c, y) \lor H(f(a), b) \lor E(a) \end{aligned}$$

can be found, as

$$H(y, c) \lor D(z) \lor H(f(a), b) \lor E(a)$$

Note that not all expressions can be unified. For example, $Q(a, b, y)$ and $Q(c, b, y)$ cannot be unified, because there is no instantiation that makes individual a become individual c.

In general, the decision of which expressions to unify is nontrivial. To make a good choice as to which expressions to unify next, one must think about what is to be accomplished.

Automated Theorem Proving

A theorem is a logical argument, in the sense that it has several premises and a conclusion.

To automatically prove that a theorem is correct (that is, prove it is a valid logical argument), we first transform the premises and negation of the conclusion into a set of clauses, as follows:

- Each formula is converted into Prenex Normal Form.
- The existential quantifiers are replaced by Skolem functions.
- The universal quantifiers are dropped (by convention).
- The resulting quantifier-free sentences are converted into clauses, i.e., their matrices are transformed into Conjunctive Normal Form, with each disjunctive clause becoming a separate clause on its own.

If the set of clauses thus obtained is not satisfiable, then the theorem is correct (it is a valid logical argument).

Theorem: A set S of clauses in first-order logic is not satisfiable iff there is a resolution derivation of the empty clause, $\bot$ (a contradiction) from S.

Soundness: If resolution with input S outputs the empty clause, then the set S is not satisfiable.

Completeness: If the set S is not satisfiable, then resolution with input S outputs the empty clause.

By the Soundness and Completeness Theorem, a set of clauses is not satisfiable iff a contradiction (the empty clause, $\bot$) can be derived by resolution.

Resolution can only be applied to formulas that contain complementary literals.

To create complementary literals, we use unification, and then we determine the resolvent.

Any search for a contradiction in a set of clauses can be restricted to formulas that can be unified.

Thus, automated resolution theorem proving uses unification combined with resolution to obtain an efficient refutation method (method for obtaining a contradiction, $\bot$).

Example: Prove that everybody has a grandparent, provided everybody has a parent.

Solution: Let the domain be the set of all people, and $P(x, y)$ represent x is a parent of y. The premise can now be stated as $\forall x \exists y P(y, x)$.

From this, we must be able to conclude that there exists a parent of a parent, which can be expressed as $\forall x \exists y \exists z (P(z, y) \land P(y, x))$.

We must thus prove that

$$\forall x \exists y P(y, x) \vDash \forall x \exists y \exists z (P(z, y) \land P(y, x))$$

We add the negation of the conclusion to the set of premises, which yields the formulas:

$$\forall x \exists y P(y, x), \quad \exists x \forall y \forall z (\neg P(z, y) \lor \neg P(y, x))$$

Eliminate the existential quantifiers (obtain the $\exists-$free prenex normal form of the formulas) to obtain:

$$\forall x P(f(x), x), \forall y \forall z (\neg P(z, y) \lor \neg P(y, a))$$

After dropping the universal quantifiers, this yields the set of clauses

$$\{P(f(x), x), \neg P(z, y) \lor \neg P(y, a)\}$$

Resolution can now be used to find the empty clause $\bot$ (a contradiction) as follows:

1.	$P(f(x), x)$	(from premise)
2.	$\neg P(z, y) \lor \neg P(y, a)$	(from negation of conclusion)
3.	$P(f(a), a)$	1 with $x \coloneqq a$
4.	$\neg P(z, f(a)) \lor \neg P(f(a), a)$	2 with $y \coloneqq f(a)$
5.	$\neg P(z, f(a))$	Resolve 3 and 4
6.	$P(f(f(a)), f(a))$	1 with $x \coloneqq f(a)$
7.	$\neg P(f(f(a)), f(a))$	5 with $z \coloneqq f(f(a))$
8.	$\bot$	Resolve 6 and 7

By the Soundness of Resolution, the fact that we obtained the empty clause $\bot$ implies that the original argument is valid.

Comments on resolution.

Any clause can be used multiple times as parent clause.

Any clause with variables can be instantiated multiple times:

- For example, if $R(x, y) \lor Q(x, y)$ is a clause, it can produce the new clause $R(a, b) \lor Q(a, b)$ via the instantiation $x \coloneqq a$ and $y \coloneqq b$, as well as the new clause $R(f(x), x) \lor Q(f(x), x)$ via the instantiation $x \coloneqq f(x)$ and $y \coloneqq x$, if so needed.
- The intuition behind instantiation is that in resolution for first-order logic, after obtaining formulas in $\exists-$free prenex normal form, all variables are assumed to be implicitly universally quantified and can thus be instantiated by any quasi-terms.

In any given clause, we can remove duplicate literals. For instance, $P(y) \lor Q(x) \lor P(y)$ is written as $P(y) \lor Q(x)$.

Resolutions that result in formulas that are (universally) valid should be avoided, since a formula that is always true can never lead to a contradiction. For example, one should avoid a resolution whose resolvent is $P(a) \lor \neg P(a) \lor Q(b)$.

14 Logic16a: Logic and Computation

Recall, an argument in first-order logic is valid iff the set S of clauses obtained from all premises and the negation of conclusion is not satisfiable.

The Soundness and Completeness for resolution in first-order logic states that a set of clauses S is not satisfiable iff there is a derivation of the empty clause $\bot$, from S, by resolution.

Informally, an algorithm is a finite sequence of well-defined computer-implementable instructions, typically to solve a class of problems or perform a computation.

We say that an algorithm solves a problem if, for every input, the algorithm produces the correct output.

There are problems that cannot be solved by computer programs (algorithms), even assuming unlimited time and space.

Halting Problem: Does there exist an algorithm (program) that operates as follows:

> Input: A program P, and an input I to the program.
> Output: "Yes" if the program P halts on input$I$, and "No" otherwise

We will describe Turing's proof that no such algorithm exists.

Halting problem examples.

The "$3x + 1$" problem

Input: Positive integer x

While x is not equal to 1

- If x is even, then $x \coloneqq x/2$
- Else $x \coloneqq 3x + 1$

Does this halt on all inputs? No one knows.

Theorem: The Halting Problem is unsolvable.

Proof: By contradiction.

Assume that there is a solution to the Halting Problem, namely a program called $H(P, I)$ that can determine whether or not a program halts, as follows.

$H(P, I)$ takes two inputs, the first being a program P, and the second being I (an input to the program P).

$H(P, I)$ outputs:

- The string "halt" (Yes) if the program P halts on input I, and
- The string "loops forever" (No) if the program P never stops on input I

We will now derive a contradiction.

When an algorithm is coded, it is expressed as a string of characters; this string can be interpreted as a sequence of bits.

This means that the program itself can be used as data.

Therefore, a program can be thought of as input for another program, or even for itself.

Hence our hypothetical program H can take a program P as both its inputs, that is, we could call $H(P, P)$.

H should be able to determine whether P will halt when it is given a copy of itself as an input.

Construct a program $K(P)$ such that:

- If $H(P, P)$ outputs "halt", then $K(P)$ goes into an infinite loop, e.g., printing "ha" at each iteration.
- If $H(P, P)$ outputs "loops forever", then $K(P)$ halts.

In other words, $K(P)$ does the opposite of what the output $H(P, P)$ specifies.

Case 1: If $K(K)$ halts then, by definition of $H(K, K)$, it follows that $H(K, K)$ outputs "halt" then, by construction of $K(K)$ (which calls $H(K, K)$, and does the opposite of what H specifies), we have that $K(K)$ loops forever - contradiction.

Case 2: If $K(K)$ loops forever then, by definition of $H(K, K)$, it follows that $H(K, K)$ outputs "loops forever". But, if $H(K, K)$ outputs "loops forever" then, by construction of $K(K)$ (which calls $H(K, K)$, and does the opposite of what H specifies, we have that $K(K)$ halts - contradiction.

Since in both cases we reached a contradiction, our assumption of the existence of a "halting program" $H(P, I)$ was incorrect.

Thus, no algorithm $H(P, I)$ exists, that solves the Halting Problem (i.e., for all inputs P and I, it terminates and answers "Yes" if program P stops on input I, and "No" otherwise).

A Turing Machine is a simple mathematical model of the notion of algorithm/computation. It consists of:

- A two-way infinite tape, divided into cells
- A finite control unit with a read-write head, which can move along the tape, and can be in any state from a finite set of states
- Read/Write capabilities on the tape, as the finite control unit moves back and forth on the tape, changing states depending on: (a) the tape symbol currently being read, and (b) its current state

A Turing Machine $T = (S, I, f, s_0)$ consists of:

- S - a finite set of states,
- I - an input alphabet (finite set of symbols/letters) containing the blank symbol B,
- $s_0 \in S$ - the start state,
- $f : S \times I \longrightarrow S \times I \times \{L, R\}$ - a partial function called the transition function, where L and R stand for the direction "left" and "right."

To interpret this definition in terms of a machine, consider a control unit and a tape divided into cells, infinite in both directions, having only a finite number of non-blank symbols on it at any given time.

Given a string, to write it on the tape means that we write the consecutive symbols of this string in consecutive cells.

The action of the Turing machine at each step of its operation depends on the value of the transition function f for the current state and current tape symbol being read by the control unit.

At each step, the control unit reads the current tape symbol x. If the control unit is in state s and the partial function f is defined for the pair (s, x), by $f(s, x) = (s', x', d)$, then the control unit:

1. Enters the state s',
2. Writes the symbol x' in the current cell, erasing x, and
3. Moves right (left) by one cell if $d = R$ (respectively $d = L$)

We write this step as the five-tuple (s, x, s', x', d), and call it a transition rule of the TM.

If the transition function f is undefined for the pair (s, x), then the Turing machine T will halt.

At the beginning of its operation a TM is assumed to be in the initial state s_0 and to be positioned over the leftmost non-blank symbol on the tape. If the tape is all blank, the control head can be positioned over any cell.

This positioning of the control head in state s_0 over the leftmost non-blank tape symbol is the initial position of the machine.

An alphabet Σ is a finite non-empty set of symbols, also called letters. For example $\Sigma = \{0, 1\}$ is an alphabet.

By Σ^* we denote the set of all possible strings written with letters from Σ, including the empty string λ. For example, if $\Sigma = \{0, 1\}$ then $0, 011, 001100$ are strings in Σ^*.

A language L over Σ is a subset of Σ^*. If $\Sigma = \{0, 1\}$, then $L = \{w \in \{0, 1\}^* | w$ is a string with equally many 0s and 1s $\}$ is a language over $\Sigma = \{0, 1\}$.

TMs can be used to accept / recognize languages.

To do so requires that we define the concept of final state.

Definition: A final state of a Turing machine $T = (S, I, f, s_0)$ is any state $s_f \in S$ that is not the first state in any five-tuple in the description of T using five-tuples.

Definition: Let V be a subset of I. A Turing machine $T = (S, I, f, s_0)$ accepts (recognizes) a string $x \in V^*$ if and only if T, starting in the initial position when x is written on the tape, halts in a final state. T is said to accept (recognize) a language $L \subseteq V^*$, if x is accepted (recognized) by T if and only if x belongs to L.

Note that to accept a subset L of V^* we can use symbols not in V. This means that the input alphabet I may include symbols not in V. These extra symbols are often used as markers.

Question: When does a Turing Machine T not accept a string x in V^*?

Answer: A string $x \in V^* \subseteq I^*$ is not accepted if, when started in the initial position with x written on the tape, either

- T does not halt, or
- T halts in a non-final state.

A common way to define a TM is to specify its transition rules as a set of five-tuples of the form (s, x, s', x', d). Another way to define a TM is by a transition diagram, where

- Each state is represented by a node.
- The start and final states are specified
- A transition rule (s, x, s', x', d) is symbolized by an arrow between node s and the node s'. This arrow is labelled by the triplet $x/x', d$ (current-symbol, new-symbol, move).

We saw that Turing machines can be used to accept languages.

A TM can also be thought of as computing a partial function.

- Suppose that the Turing machine T, when given the string x as input, halts with the string y on its tape.
- We can then define $T(x) = y$.
- The domain of T is the set of strings for which T halts.
- $T(x)$ is undefined if T does not halt when given x as input.

Using appropriate encoding of integers in unary notation, the idea above can be used to define TMs that compute (partial or total) functions from integers to integers.

Definition: A Turing Machine that always halts, on every input is called a decider or a total Turing Machine.

TMs are relatively simple, but they are extremely powerful.

Turing also showed that one can construct a single TM, called Universal Turing Machine (UTM) that can simulate the computations of every TM, when given as input: (a) an encoding of the TM, together with (b) an input for the TM.

The Turing machine is the currently accepted formalization of the informal notion of algorithm/computation, and is the most general model of computation; the total Turing machine is the formalization of the notion of terminating algorithm.

A Universal Turing Machine is the formalization of the notion of a computer (A UTM can do whatever a computer can do).

Clearly we cannot prove that the Turing Machine model is equivalent to our intuitive idea of an algorithm/computation, but there are compelling arguments for this equivalence, which has become known as the Church-Turing Thesis, which states:

"Any problem that can be solved with an algorithm can be solved by a Turing Machine"

It was proved that a Turing Machine is equivalent in computing power to all other, most general, mathematical models of computation. Thus, the Church-Turing Thesis is used as a basis to prove if a given problem is solvable by a computer or not.

Definition: A decision problem is a yes-or-no question on an infinite set of inputs. Each input is an instance of the problem,

Example: Satisfiability for First-Order Logic:

- Input: A formula A in first-order logic.
- Output: Yes, if A is satisfiable, and no otherwise

We can think of a decision problem as the language L of all problem instances for which the answer to the corresponding decision problem is "yes".

Definition: A decision problem for which there exists a terminating algorithm that solves it (a total TM that accepts those and only those problem instances that lead to a "yes" answer), is called decidable (solvable). If no such algorithm exists, the decision problem is called undecidable (unsolvable).

Definition: A (total) function that can be computed by a (total) TM is called computable, otherwise it is called uncomputable.

Turing machines were introduced by Alan Turing. The Halting Problem was proved undecidable by Turing in 1936.

Given a formula A in propositional logic, is A:

1. Unsatisfiable?
2. Satisfiable?
3. A tautology?
4. A contradiction?

All the above problems are decidable and we have described several algorithms to solve them during this course.

To show that a problem is undecidable we often use reductions.

A mathematician and an engineer are on a desert island. They find two palm trees with one coconut each.

The engineer climbs up the first tree, gets the coconut, eats.

The mathematician climbs up the second palm tree, gets the coconut, climbs the first palm tree and puts the coconut there:

"Now we've reduced it to a previously solved problem."

Say we know that problem P_1 is solvable, and want to solve new problem P_2. If we reduce problem P_2 to problem P_1, this implies "If P_1 was solvable, then P_2 is solvable."

Conversely, say we know that P_1 is unsolvable and want to prove that P_2 is also unsolvable. Then we have to use the opposite reduction, that is, reduce the old unsolvable problem P_1 to the new problem P_2.

Assume we already proved that another problem P_1 is undecidable.

If we have a (terminating) algorithm to convert any instance of the problem P_1 into an instance of the problem P_2 with the same yes/no answer, we say that "we reduced P_1 to P_2".

Such an algorithm is called a "reduction from P_1 to P_2".

Theorem: If Problem P_1 is reducible to problem P_2, then "If P_1 is undecidable then P_2 is undecidable".

It is a common mistake to try to prove a new problem P_2 undecidable by reducing P_2 to some old known (undecidable) problem P_1, thus proving the statement "If P_1 is decidable, then P_2 is decidable"

That statement, although true, is useless, since its hypothesis "P_1 is decidable" is false.

The correct way to prove a new problem P_2 undecidable is to reduce another known undecidable problem P_1 to our P_2.

This reduction proves that "If P_2 were decidable, then P_1 would be decidable," with contrapositive "If P_1 is undecidable, then P_2 is undecidable."

Thus, since we know that P_1 is undecidable, the antecedent of the latter implication is true and we can deduce that our P_2 is also undecidable.

The blank-tape halting problem.

Input: Turing Machine M.

Question: Does M halt when started with a blank tape?

Theorem: The blank-tape halting problem is undecidable.

Proof: Reduce the halting problem to the blank-tape halting problem (use nested deciders).

A decision problem for which there exists a terminating algorithm that solves it, is called decidable (solvable). If no such algorithm exists, the decision problem is called undecidable (unsolvable).

To show that a decision problem is solvable/decidable, we only need to construct a terminating algorithm that solves it.

To show that a decision problem is unsolvable/undecidable, we need to prove that no such algorithm exists. The fact that we tried to find such an algorithm but failed, does not prove that the problem is unsolvable.

By studying only decision problems, it may seem that we are studying only a small set of problems. However, most problems can be recast as decision problems.

15 Logic16b: Turing Machines

Example 1: Consider a Turing machine $T_1 = (S, I, f, s_0)$, with set of states $S = \{s_0, s_1, s_2, s_3\}$, alphabet $I = \{0, 1, B(\text{blank})\}$, start state s_0 and (partial) transition function f defined by the transition rules:

1. $(s_0, 0, s_0, 0, R)$
2. $(s_0, 1, s_1, 1, R)$
3. (s_0, B, s_3, B, R)
4. $(s_1, 0, s_0, 0, R)$
5. $(s_1, 1, s_2, 0, L)$
6. (s_1, B, s_3, B, R)
7. $(s_2, 1, s_3, 0, R)$

Recall the five-tuple transition rule notation used to define the values of the transition function: for example, $f(s_1, 1) = (s_2, 0, L)$ is denoted by the five-tuple $(s_1, 1, s_2, 0, L)$, etc.

A Turing machine computation consists of successive applications of transition rules to the tape content.

One computation step (transition step) = the application of one transition rule.

Question: What is the output of the computation of the Turing machine T_1, given the input 010110?

Answer: The output is 010000.

A configuration of a TM $T = (S, I, f, s_0)$ is denoted by $\alpha_1 s \alpha_2$.

Here $s \in S$ is the current state of T, and $\alpha_1\alpha_2$ is the string in I^* that consists of the current contents of the tape, starting with the leftmost non-blank symbol and up to the rightmost non-blank symbol, with respect to the read/write head (observe that the blank B may occur in $\alpha_1\alpha_2$).

The read/write head is assumed to be scanning the leftmost symbol of α_2 or, if $\alpha_2 = \lambda$, it is scanning a blank.

A computation of the TM consists of a succession of configurations, each obtained by applying one transition rule to the previous configuration ($\implies$ denotes the application of a transition rule, and $\implies^*$ denotes zero or more rule applications).

With this notation, the computation of T_1 with input 010110 is $s_0010110 \implies 0s_010110 \implies 01s_10110 \implies 010s_0110 \implies 0101s_110 \implies 010s_2100 \implies 0100s_300$.

States are represented by nodes in a transition diagram (directed graph). The start state s_0 is singled out by an incoming arrow.

A transition rule (s_i, a, s_j, b, R) is represented by an arrow (directed edge) from s_i to s_j, labelled by $a; b; R$.

A computation corresponds to a path in the graph.

At the beginning, the TM is in the start state, with the read/write head over the leftmost symbol of the input string.

Example: Construct a Turing machine T_2 that recognizes (accepts) the set of bit strings in $\{0, 1\}^*$ that have a 1 as their second bit.

We want a TM that, starting at the leftmost non-blank tape cell, moves right and determines whether the 2nd symbol is a 1.

If the 2nd symbol is a 1, the TM should move into a final state. If the 2nd symbol is not a 1, the TM should not accept (that is, it should not halt, or it should halt in a non-final state).

We include five-tuples $(s_0, 0, s_1, 0, R)$ and $(s_0, 1, s_1, 1, R)$ to read the 1st symbol and put the TM in state s_1.

We include five-tuples $(s_1, 0, s_2, 0, R)$ and $(s_1, 1, s_3, 1, R)$ to read the 2nd symbol and either move to state s_2 if this symbol is a 0, or to state s_3 if this symbol is a 1 (s_3 should be a final state).

We do not want to recognize strings with 0 as their 2nd symbol, so s_2 should not be a final state (final states have no outgoing transitions). Thus, we include the five-tuple $(s_2, 0, s_2, 0, R)$.

We do not want to recognize the empty string, respectively a string with one bit only. Thus, we include the five-tuples $(s_0, B, s_2, 0, R)$, respectively $(s_1, B, s_2, 0, R)$.

The TM is $T_2 = (S, I, f, s_0)$, where $S = \{s_0, s_1, s_2, s_3\}$, the alphabet is $I = \{0, 1, B\}$, the initial state is s_0, the final state is s_3, and the transition function is defined by the seven five-tuples we described.

This TM will terminate in the final state s_3 if and only if the input bit string has at least two bits, and the 2nd bit of the input string is a 1.

If the bit string contains fewer than two bits, or if the 2nd bit is not a 1, the TM will terminate in the non-final state s_2.

Final states are double-circled nodes (final states = states with no outgoing arrows).

TM computation on input string 01100 (accept, as 2nd bit is 1) $s_001100 \implies 0s_11100 \implies 01s_3100$.

TM computation on input string 000 (not accept) $s_0000 \implies 0s_100 \implies 00s_20 \implies 000s_2$

TM computation on input string 1, and blank tape (not accept) $s_01 \implies 1s_1 \implies 10s_2 \quad s_0B \implies 0s_2$.

Note: This TM is not minimal.

Example:

Construct a TM that recognizes the set $\{0^n1^n | n \geq 1\}$.

We will use an auxiliary tape symbol M as a marker.

We have $V = \{0, 1\}$, and $I = \{0, 1, M, B\}$.

We wish to recognize only a subset of strings in V^*.

We will have one final state, s_6.

The TM successively replaces a 0 at the leftmost position of the string with an M, and a 1 at the rightmost position of the string with an M, sweeping back and forth, terminating in a final state if and only if the string consists of a block of 0s followed by a block of the same number of 1s.

Although this is easy to describe and is easily carried out by a Turing Machine, the machine is somewhat complicated.

We use the marker M to keep track of the leftmost and rightmost symbols we have already examined.

$(s_0, 0, s_1, M, R), (s_1, 0, s_1, 0, R), (s_1, 1, s_1, 1, R), (s_1, M, s_2, M, L), (s_1, B, s_2, B, L),$
$(s_2, 1, s_3, M, L), (s_3, 1, s_3, 1, L), (s_3, 0, s_4, 0, L), (s_3, M, s_5, M, R),$
$(s_4, 0, s_4, 0, L)(s_4, M, s_0, M, R), (s_5, M, s_6, M, R)$

To consider a TM as computing number-theoretic functions (functions from the set of k-tuples of natural numbers to the set of natural numbers), we need a way to represent $k-$tuples of natural numbers on tape.

We use unary representations, whereby the natural number n is represented by a string of $(n + 1)$ 1s.

For instance, 0 is represented by the string 1, and 5 is represented by the string 111111.

To represent the k-tuple $(n_1, n_2, \ldots, n_k)$ we use a string of $(n_1 + 1)$ 1s, followed by an asterisk, followed by a string of $(n_2 + 1)$ 1s, followed by an asterisk, and so on, ending with a string of $(n_k + 1)$ 1s.

For example, to represent the four-tuple $(2, 0, 1, 3)$ we use the string $111^*1^*11^*1111$.

Example: Construct a TM that adds two natural numbers.

We need a TM computing function $f(n_1, n_2) = n_1 + n_2$.

The pair (n_1, n_2) is represented by a string of $(n_1 + 1)$ 1s, followed by an asterisk, followed by a string of $(n_2 + 1)$ 1s.

The TM should take this as an input and produce as output a tape with $(n_1 + n_2 + 1)$ 1s.

One way to do this is as follows (the alphabet is $\{0, 1, *\}$).

The TM first erases the leftmost 1 of n_1. If $n_1 = 0$, then it erases the asterisk and it halts.

Otherwise, it reads the leftmost remaining 1 in n_1 (and it deletes it, but remembers this by changing to state s_2), then traverses all remaining 1s in n_1 until it reaches the asterisk $*$, which it replaces by the "remembered" 1. Then it halts, in final state s_3.

The transition function is: $(s_0, 1, s_1, B, R), (s_1, *, s_3, B, R), (s_1, 1, s_2, B, R), (s_2, 1, s_2, 1, R), (s_2, *, s_3, 1, R)$.

A total function that can be computed by a total Turing machine is called computable, otherwise it is called uncomputable.

Uncomputable function example: the Busy Beaver function.

Let $B(n)$ be the maximum number of 1s that a Turing machine with n states and alphabet $\{1, B\}$ may print on a tape before halting, when started with a blank tape. The problem of determining $B(n)$ for particular values of n is known as the Busy Beaver Problem.

Currently it is known that $B(2) = 4, B(3) = 6$, and $B(4) = 13$, but $B(n)$ is not known for $n \geq 5$. It is known that $B(5) \geq 4098$ and $B(6) \geq 10^{18267}$

Constructing TMs to compute relatively simple functions can be extremely tedious. For example, a TM for multiplying two non-negative integers, which is found in many books, has 31 five-tuples, and 11 states.

If it is challenging to construct TMs to compute even relatively simple functions, what hope do we have of building TMs for more complicated functions?

One way to simplify this problem is to use a multi-tape TM that uses more than one tape simultaneously, and to build up multi-tape TMs for the composition of functions.

It can be shown that TMs and multi-tape TMs have the same computational power, that is, for any multi-tape TM there is as one-tape TM that can compute the same thing.

Assuming that the Church-Turing thesis holds, "If there exists an algorithm that solves a problem, then there exists a TM that solves it."

16 Logic17: Peano Arithmetic

We gave seen two rules of formal deduction concerning equality, $\approx$, where $A(u)$ is a relation, and t_1, t_2 are terms.

$(\approx -)$ If $\Sigma \vdash A(t_1), \Sigma \vdash t_1 \approx t_2$, then $\Sigma \vdash A'(t_2)$, where $A'(t_2)$ results from $A(t_1)$ by replacing some occurrences of t_1 by t_2.

$(\approx +)$ $\emptyset \vdash u \approx u$

We can use these rules to prove the usual properties of equality.

- (Reflexitivity of Equality) $\forall x(x = x)$
- (Symmetry of Equality) $\forall x \forall y((x = y) \implies (y = x))$
- (Transitivity of Equality) $\forall x \forall y \forall z((x = y) \wedge (y = z) \implies (x = z))$

Proof that $\emptyset \vdash \forall x(x = x)$

1. $\emptyset \vdash u = u \quad (\approx +)$
2. $\emptyset \vdash \forall x(x = x) \quad (1, \forall +)$ [u not elsewhere]

Notation conventions:

We employ $=$ and $\approx$ interchangeably, usually using $\approx$ when citing a formal deduction rule/theorem involving equality, and using $=$ when equality occurs inside a formula.

Proof that $\emptyset \vdash \forall x \forall y((x = y) \implies (y = x))$

1. $(u = v) \vdash (u = v) \quad (\in)$
2. $\emptyset \vdash (u = u) \quad (\approx +)$
3. $(u = v) \vdash (u = u) \quad (2, +)$
4. $(u = v) \vdash (v = u) \quad (1, 3, \approx -)$
5. $\emptyset \vdash (u = v) \implies (v = u) \quad (4, \implies +)$
6. $\emptyset \vdash \forall y((u = y) \implies (y = u)) \quad (5, \forall +)$, [$u$ not elsewhere]
7. $\emptyset \vdash \forall x \forall y((x = y) \implies (y = x)) \quad (6, \forall +)$, [$u$ not elsewhere]

Proof that $\emptyset \vdash \forall x \forall y \forall z((x = y) \wedge (y = z) \implies (x = z))$

1. $(u = v) \wedge (v = w) \vdash (u = v) \wedge (v = w) \quad (\in)$
2. $(u = v) \wedge (v = w) \vdash (u = v) \quad (1, \wedge -)$
3. $(u = v) \wedge (v = w) \vdash (v = w) \quad (1, \wedge -)$
4. $(u = v) \wedge (v = w) \vdash (u = w) \quad (2, 3, \approx -)$
5. $\emptyset \vdash (u = v) \wedge (b = w) \implies (u = w) \quad (4, \implies +)$
6. $\emptyset \vdash \forall z((u = v) \wedge (v = z) \implies (u = z)) \quad (5, \forall +)$, [$w$ not elsewhere]
7. $\emptyset \vdash \forall y \forall z((u = y) \wedge (y = z) \implies (u = z)) \quad (6, \forall +)$, [$v$ not elsewhere]
8. $\emptyset \vdash \forall x \forall y \forall z((x = y) \wedge (y = z) \implies (x = z)) \quad (7, \forall +)$, [$u$ not elsewhere]

First-order logic is often used to describe specialized domains, starting from a small number of relation and function symbols. In each case, we use some "domain axioms", which are first-order logic formulas assumed to be true in that domain, and that specify properties of the relation and function symbols.

A set of domain axioms, together with a system of formal deduction, and all theorems that can be formally proved from the domain axioms, is called a theory. Examples:

- Number theory
- Set theory
- Group theory
- Graph theory

The set A of domain axioms is a set of first-order logic formulas which we accept (assume to be always true in that domain/theory).

The set A should be decidable: There should exist a terminating algorithm to decide if a given formula is a domain axiom.

The set A should be consistent (with respect to $\vdash$ for first-order logic).

The set A should be syntactically complete, in the sense that for any formula F describable in the language of the system, either F or its negation, $\neg F$, should be provable from A.

Note: The notion of syntactic completeness of a set of domain axioms (and its corresponding theory) is different from that of (semantic) completeness of a system of formal deduction (the latter means that $\Sigma \vDash F$ implies $\Sigma \vdash F$).

Oldest example of a "theory" - Euclidean Geometry.

Euclid's Postulates ("Geometry Axioms")

1. A straight line may be drawn between any two points.
2. Any straight line can be extended infinitely.
3. A circle may be drawn with any given point as the centre, and any given radius.
4. All right angles are equal.
5. Parallel Postulate: For any given point not on a given line, there is exactly one line passing through the point, that is parallel to the given line.

There were many failed attempts over the centuries to prove that Parallel Postulate from the others.

Finally, it was proved that there exist interpretations in which all the other 4 "geometry axioms" hold true, but the Parallel Postulate fails.

Thus, by the Soundness and Completeness Theorem, the Parallel Postulate is not provable from the other 4 geometry axioms.

Note: In non-euclidean geometry the Parallel Postulate is replaced with other possibilities: no such line (spherical or elliptic geometry), infinitely many lines (hperbolic geometry), or no assumption (absolute geometry).

Another example is Number Theory.

Intended interpretation:

- Domain: Natural numbers $0, 1, 2, 3 \ldots$
- Addition $+$
- Multiplication $\cdot$
- Ordering via $<$

Number Theory Axioms should be a small set of true statements (formulas) from which all theorems about natural numbers can be derived. We want induction.

Peano's axioms are the basis for the version of number theory known as Peano Arithmetic (PA).

Non-logical symbols:

- Individual (constant): 0
- Functions: Successor s, $+$ (addition), $\cdot$ (multiplication)
- Relation: Equality (this is already part of first-order logic)

Axioms defining the unary function successor, and the binary functions addition, and multiplication and axiom for induction.

Axioms defining the unary function successor, denoted by s

PA1 $\forall x \neg(s(x) = 0)$

PA2 $\forall x \forall y((s(x) = s(y)) \implies (x = y))$

We want the successor to give us

- Numbers $0, s(0), s(s(0)), s(s(s(0))), \ldots$ (which are $0, 1, 2, 3, \ldots$)
- $\neg(s(x) = x)$ (we will prove it later), etc.

Note: We often use $A \neq B$ to denote $\neg(A = B)$

Axioms defining the binary function addition, denoted by $+$

PA3 $\forall x(x + 0 = x)$

PA4 $\forall x \forall y(x + s(y) = s(x + y))$

Axioms defining the binary function multiplication, denoted by $\cdot$

PA5 $\forall x(x \cdot 0 = 0)$

PA6 $\forall x \forall y(x \cdot s(y) = x \cdot y + x)$

Principle of Mathematical Induction

Let $P(n)$ be a statement that depends on $n \in N$.

If

1. (Base Case) $P(0)$ is true and
2. (Inductive Step) For all $k \in N$ we have: "$P(k)$ is true implies that $P(k + 1)$ is true"

then $P(n)$ is true for all $n \in N$.

Recall that "$P(k)$ is true" is called the "Inductive Hypothesis")

Express the Principle of Mathematical Induction in first-order logic:

$$[P(0) \wedge \forall x(P(x) \implies P(s(x))] \implies \forall x P(x)$$

Let PA be the set {PA1, PA2, PA3, PA4, PA5, PA6, PA7}

In a Peano Arithmetic proof, the set PA is implicitly considered to be part of the set of premises of any theorem we wish to prove.

To account for this, we introduce the following;

Notation: Given a theory T, with an associated set of axioms A_T, we use $\Sigma \vdash_{A_T} C$ to denote the fact that $\Sigma, A_T \vdash C$

In particular, for Peano Arithmetic, $\Sigma \vdash_{PA} C$ denotes $\Sigma \cup PA \vdash C$

Proofs in Peano arithmetic.

Example 1: Prove that $\forall x(s(x) \neq x)$.

Solution: Formally, we want to prove that $\emptyset \vdash_{PA} \forall x(s(x) \neq x)$

Proof idea: Apply induction to $A(u) : s(u) \neq u$.

Use PA7: $A(0) \land \forall x(A(x) \implies A(s(x)) \implies \forall x A(x)$

Proof structure:

1. Prove $A(0)$, the Base Case.
2. Prove $\forall x(A(x) \implies A(s(x)))$, the Inductive Step
3. Obtain $A(0) \land \forall x(A(x) \implies A(s(x)))$ from (1) and (2), by $(\land+)$
4. Obtain $\forall x A(x)$, using (3) and PA7, by $(\implies -)$

Informal Proof

Apply induction to $A(u) : s(u) \neq u$

Base Case: $A(0) : (s(0) \neq 0)$

By PA1: $\forall x(s(x) \neq 0)$, with $x \coloneqq 0$ proves the Base Case.

Inductive Step:

Assume $s(k) \neq k$ (IH).

Prove $s(s(k)) \neq s(k)$

Prove the inductive step by contradiction.

Suppose $s(s(k)) = s(k)$.

By PA2: $\forall x \forall y((s(x) = s(y)) \implies (x = y))$, with $x \coloneqq s(k), y \coloneqq k$, we have $s(k) = k$. This contradicts the I.H

Formal proof of $\emptyset \vdash_{PA} \forall x(s(x) \neq x)$

$$
\begin{aligned}
1.&\emptyset \vdash_{PA} \forall x(s(x) \neq 0)(PA1)\\
2.&\emptyset \vdash_{PA} s(0) \neq 0(1, \forall -)\\
3.&s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(s(k)) = s(k)(\in)\\
4.&\emptyset \vdash_{PA} \forall x \forall y(s(x) = s(y) \implies x = y)(PA2)\\
5.&\emptyset \vdash_{PA} s(s(k)) = s(k) \implies s(k) = k(4, \forall -, x \coloneqq s(k); \forall -, y \coloneqq k)\\
6.&s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(s(k)) = s(k) \implies s(k) = k(5, +)\\
7.&s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(k) = k(3, 6, \implies -)\\
8.&s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(k) \neq k(\in)\\
9.&s(k) \neq k \vdash_{PA} s(s(k)) \neq s(k)(7, 8, \neg+)\\
10.&\emptyset \vdash_{PA} s(k) \neq k \implies s(s(k)) \neq s(k)(9, \implies +)\\
11.&\emptyset \vdash_{PA} \forall x(s(x) \neq x \implies s(s(x)) \neq s(x))(10, \forall +), [k \text{ not elsewhere}]\\
12.&\emptyset \vdash_{PA} s(0) \neq 0 \land \forall x(s(x) \neq x \implies s(s(x)) \neq s(x))(2, 11, \land +)\\
13.&\emptyset \vdash_{PA} s(0) \neq 0 \land \forall x(s(x) \neq x \implies s(s(x)) \neq s(x)) \implies \forall x(s(x) \neq x)(PA7)\\
14.&\emptyset \vdash_{PA} \forall x(s(x) \neq x)(12, 13, \implies -)
\end{aligned}
$$

Example 2: Prove that $\forall x(x = 0 \lor \exists y(s(y) = x))$.

(A natural number is either zero, or it has a predecessor.)

Formally, we have to prove that $\emptyset \vdash_{PA} \forall x(x = 0 \lor \exists y(s(y) = x))$

Take $P(u)$ to be $(u = 0) \lor \exists y(s(y) = u)$. We have to prove $\emptyset \vdash_{PA} \forall x P(x)$.

We will use induction on x, as formalized by:

PA7: $P(0) \land \forall x(P(x) \implies P(s(x)) \implies \forall x P(x)$

(Base Case) We first need to prove the first operand of $\land$, that is, prove $P(0)$ which is $(0 = 0) \lor \exists y(s(y) = 0)$

(Inductive Step) For the second operand of $\wedge$, we will assume the Inductive Hypothesis $P(k)$ holds: $(k = 0) \vee \exists y(s(y) = k)$.

Under this assumption, we will have to prove $P(s(k))$ holds: $(s(k) = 0) \vee \exists y(s(y) = s(k))$

Then we use $(\implies +)$ to obtain $P(k) \implies P(s(k))$, and use $(\forall +)$ to generalize over k, to conclude the proof of the Inductive Step.

Finally, we will use PA7 and $(\implies -)$ to obtain $\forall x P(x)$.

To prove $\emptyset \vdash_{PA} \forall x(x = 0 \vee \exists y(s(y) = x))$

Base Case: $\emptyset \vdash_{PA} (0 = 0) \vee \exists y(s(y) = 0)$ - the proof is easy, with $(\approx +)$ and $(\vee +)$.

Inductive Step: Prove $P(k) \vdash_{PA} P(s(k))$. Since $P(k)$ is the disjunction $(k = 0) \vee \exists y(s(y) = k)$, we seem to need proof by cases, $(\vee -)$. However, this is not needed. Note that $P(s(k))$ is $(s(k) = 0) \vee \exists y(s(y) = s(k))$, which we can prove without using $P(k)$.

We will use "$\emptyset \vdash t = t$, where t is any term," proved below:

1. $\emptyset \vdash \forall x(x = x)$ (Reflexitivity of $\approx$)
2. $\emptyset \vdash t = t$ $(1, \forall -)$

We use this theorem under the same name, "Reflexitivity of $\approx$".

Proof of $\emptyset \vdash_{PA} \forall x((x = 0) \vee (\exists y(s(y) = x))$

Other theorems we can prove.

- $\forall x((x \neq 0) \implies \exists y(s(y) = x))$
- $\forall x \forall y(x + y = x \implies y = 0)$
- Commutativity of addition (requires double induction) $\forall x \forall y(x + y = y + x)$
- Associativity of addition $\forall x \forall y \forall z((x + y) + z + x + (y + z))$
- Commutativity of multiplication
- All the things you expect ...

We can define new arithmetic relations be using logic formulas to describe their behaviour. For example, we can define:

- $u \leq v$ to be true iff $\exists z(u + z = v)$
- $u < v$ to be true iff $\exists z(u + s(z) = v)$
- $Even(u)$ to be true iff $\exists y(u = y + y)$
- $Prime(u)$ to be true iff $(1 < y) \wedge \neg(\exists z \exists y((u = y \cdot z) \wedge (1 < y) \wedge (1 < z)))$

We can use axioms of Peano Arithmetic to prove properties of relations.

Example 3: Prove that $\leq$ is transitive: $\forall x \forall y \forall z((x \leq y) \wedge (y \leq z) \implies (x \leq z))$

We do not need induction. We only need the properties of equality, and associativity of addition (in this example, we assume that we proved associativity of addition).

Proof idea (informal):

- From $u \leq v$ and $v \leq w$ we want to prove $u \leq w$
- $u \leq v$ means $\exists z(u + z = v)$, implying $u + \alpha_1 = v$ for some α_1
- $v \leq w$ means $\exists z(v + z = w)$, implying $v + \alpha_2 = w$ for some α_2.
- Start with $u + \alpha_1 = v$ and $v + \alpha_2 = w$, and use $(\approx -)'$ to substitute $v = u + \alpha_1$ in $v + \alpha_2 = w$, resulting in $(u + \alpha_1) + \alpha_2 = w$
- Use associativity of addition to obtain $u + (\alpha_1 + \alpha_2) = w$
- Introduce $\exists$ to obtain $\exists y(u + y = w)$ which means $u \leq w$.

In this way we can obtain all the theorems we have ever seen in number theory, starting with just 7 Peano axioms, and using the 17 rules of formal deduction for first-order logic to deduce new theorems.

Gödel's Incompleteness Theorem: In any consistent formal theory T with a decidable set of axioms, that is capable of expressing elementary arithmetic (e.g., Peano Arithmetic), there exists a statement/formula that can neither be proved nor disproved in the theory.

Gödel's original proof constructs a particular statement G_T indirectly stating "G_T is unprovable in T" (G_T is referred to as "the Gödel sentence" for the system T).

Gödel specifically cites the Liar Paradox, namely the sentence stating "This sentence is false"

A Gödel sentence G_T for a theory T makes an assertion similar to the Liar Paradox, but with "truth" replaced by "provability".

The analysis of the provability of G_T is a formalized version of the analysis of the truth of the Liar Paradox.

A CS proof of Gödel's Incompleteness Theorem.

Proof (by contradiction): Assume that any statement can be formally proved or disproved in the theory T. We will use this assumption to solve the Halting Problem.

Write a program (Turing Machine) that takes two inputs, a program P and an input I for the program P, and:

1. Generates all strings s, of all lengths (in increasing length-order), over the Latin alphabet and the set of math symbols.

- Most of them will be nonsense, some will be English, some will be "Hamlet", some will be proofs.
- By definition, any proof is of finite length.
- Among the strings s, there will be attempted proofs that P halts on I, and attempted proofs that P does not halt on I.

2. For each string s, the program checks whether s is a correct formal proof of the statement "P halts on input I" (output "yes and stop), or a proof of its negation (output "no" and stop).

Since either the statement "P halts on input I" or its negation can be formally proved in T, our program terminates and gives the correct yes/no answer.

This means we solved the Halting Problem, which is unsolvable.

Since we reached a contradiction, our assumption was incorrect, and there exist statements, expressible in T, that can neither be proved or disproved from the axioms of T.

The CS proof of Gödel's Incompleteness Theorem requires the ability to express a program (Turing Machine) in the theory T. Peano Arithmetic is such a theory.

The proof contains the consistency of T as hidden assumption: If T were inconsistent, the program could find a formal proof that P halts on I, even if P did not halt on I, and viceversa (since both proofs could exist). Thus, the program would not solve the Halting Problem since it could output an incorrect answer.

Theory = domain axioms + a system of formal deduction, + all theorems thus provable from domain axioms.

Gödel's Incompleteness Theorem establishes inherent limitations of all but the most trivial theories, i.e., inherent limitations of theories capable of doing at least basic arithmetic.

Gödel's Incompleteness Theorem is important both in mathematical logic and in the philosophy of mathematics.

This result is widely - but not universally - interpreted as showing that Hilbert's program to find a (decidable) syntactically complete and consistent set of axioms for all mathematics is impossible.

Gödel proved the Completeness Theorem for First-Order Logic in 1929.

Gödel published his Incompleteness Theorem (syntactic incompleteness of consistent theories with a decidable set of axioms, capable of expressing basic arithmetic) in 1931.

Paris-Harrington Theorem is a theorem, expressible in Peano Arithmetic and not self-referring, that is unprovable in Peano Arithmetic (it can be proved in another system).

Another such theorem G was found by Putnam and Kripke.

Does this contradict the "Completeness of $\vdash$"? (since if $PA \vDash G$ then $PA \vdash G$ by Completeness of FoL $\vdash$?) NO.

- Theorem G does not say that $PA \vDash G$ (this would mean that all interpretations that make PA true, also make G true).
- What happens is that, while G is true in the standard interpretation of arithmetic, (domain is $\mathbb{N}, 0$ is "zero", etc.), there are "nonstandard interpretations" of natural numbers that satisfy the Peano axioms PA, but do not satisfy G.
- In other words, $PA \nvDash G$, because there exist two interpretations that make PA true, one (standard) making G true, the other (nonstandard) making G false.
- Thus, Completeness of first-order logic ($\vdash$) is not contradicted.

The book "Godel, Escher, Back: an Eternal Golden Braid" includes a complete and entertaining proof of Gödel's Incompleteness Theorem.

17 Logic18: Program Verification

Program correctness: does a program satisfy its specification-does it do what it is supposed to do?

Techniques for showing program correctness:

Inspection, code walk-throughs

Testing

- Black-box testing: Tests designed independent of code
- White-box testing: Tests designed based on code

Formal program verification

- Formal verification is a formal proof system for proving programs correct.
- The motivation being formal verification is similar to that of previous modules: A proof can provide confidence of correctness in a situation where exhaustive semantic checking is time-consuming or impossible.

Testing is analogous to checking that a propositional formula is a theorem by trying a few truth valuations, or to checking that a first-order formula is a theorem by constructing a few valuations.

Testing is not proof.

A proof calculus for program correctness was first proposed by Robert Floyd and Tony Hoare.

Formal program verification:

- Formally state the specification of a problem (using the formalism of first-order logic), and
- Prove that the program satisfies the specification for all inputs.

Why formally specify and verify programs?

- Reduce bugs
- Safety-critical software or important components
- Documentation

The steps of formal (program) verification:

1. Convert the informal description R of requirements for an application into an "equivalent" formula Φ_R of some symbolic logic,
2. Write a program P which is meant to realize Φ_R in some given programming environment, and
3. Prove that the program P satisfies the formula Φ_R

We consider only Step 3 in this course and use a subset of C/C++ and Java, with their core features:

- Integer and Boolean expressions
- Assignment
- Sequence
- Conditionals
- While-loops

A program specification is an informal or formal definition of what the program is expected to do.

Hoare Triples

Our assertions about programs will have the form

$(\!|P|\!)$ − precondition

C − program or code

$(\!|Q|\!)$ − postcondition

The meaning of the triple $(\!|P|\!)\ C\ (\!|Q|\!)$:

If program C is run starting in a state that satisfies the logic formula P, then the resulting state after the execution of C will satisfy the logic formula Q.

An assertion $(\!|P|\!)\ C\ (\!|Q|\!)$ is called a Hoare triple.

Conditions P and Q are written in the first-order logic of integers. Use relations $<, =$, functions, $+, -, *$ and other derivable from these.

Definition: A specification of a program C is a Hoare triple $(\!|P|\!)C(\!|Q|\!)$ with the program C as the second component.

Example:

"If the input x is a positive number, compute a number whose square is less than x" can be expressed as the Hoare triple $(\!|x > 0|\!)C(\!|y \cdot y < x|\!)$.

Often we do not want to put any constraints on the initial state. In that case, the precondition can be set to true, which is a formula which is true in any state.

We want to develop a notion of program verification "formal proof" that will allow us to prove that a program C satisfies the specification given by the precondition P and the postcondition Q.

This kind of proof calculus is different from the (formal) proof calculus in first-order logic, since reasoning about Hoare triples has two additional features besides the logic formulas P and Q:

- Program instructions, and
- A sense of time: Before execution, versus after execution

Definition: A Hoare triple $(\!|P|\!)C(\!|Q|\!)$ is satisfied under partial correctness, denoted

$$\vDash_{\text{par}} (\!|P|\!)C(\!|Q|\!)$$

if and only if for every state s that satisfies condition P, if the execution of the program C starting from state s terminates in state s', then the state s' satisfies condition Q.

The program

```
while true {x = 0; }
```

satisfies all specifications under partial correctness.

It is an endless loop and never terminates, but partial correctness only says what must happen if the program terminates.

Definition: A Hoare triple $(\!|P|\!)C(\!|Q|\!)$ is satisfied under total correctness, denoted

$$\vDash_{\text{tot}} (\!|P|\!)C(\!|Q|\!)$$

if and only if for every state s that satisfies P, execution of program C starting from state s terminates, and the resulting state s' satisfies Q.

Total Correctness = Partial Correctness + Termination

Example 1:

$(\!|x = 1|\!)$

```
y = x;
```

$(\!|y = 1|\!)$

This Hoare triple is satisfied under both partial and total correctness.

Example 2:

$(\!|x = 1|\!)$

```
y = x;
```

$(\!|y = 2|\!)$

This Hoare triple is satisfied under neither total nor partial correctness.

Example 3:

$(\!|x \geq 0|\!)$

```
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

$(\!|y = x!|\!)$

This Hoare triple is satisfied under both partial and total correctness.

Partial correctness is a weak notion.

Example: Give a program that is satisfied under partial correctness for any pre- and postconditions.

Answer:

$(\!|P|\!)$

```
while (true) {
    x = 0;
}
```

$(\!|Q|\!)$

This program never terminates so partial correctness is vacuously satisfied.

Example;: Give pre- and postconditions that are satisfied by any program under partial correctness.

Answer:

(| true |)

```
C
```

$(\!|\ \text{true}\ |\!)$

Suppose

- C never terminates $\implies$ C satisfies the specification under partial correctness, but not under total correctness
- C sometimes terminates $\implies$ C satisfies the specification under partial correctness, but not under total correctness
- C always terminates $\implies$ C satisfies the specification under both partial and total correctness

Total correctness is our goal.

We usually prove partial correctness and termination separately.

- For proving partial correctness, we will introduce sound inference rules.
- For proving termination, we will use ad hoc reasoning, which suffices for our examples. (In general, program termination is undecidable)

There are different techniques for proving partial and total correctness.

We introduce a formal proof system for proving partial correctness.

Recall the definition of partial correctness: For every starting state that satisfies P and for which C terminates, the final state satisfies Q.

Question: How do we show this, if there are large or infinite number of possible states?

Answer: We define sound inference rules (like formal deduction rules)

A partial correctness proof will be an annotated program, with one or more conditions before and after each program statement.

Each program statement (instruction), together with the preceding and following condition, form a Hoare triple.

Each Hoare triple has a justification that explains its correctness.

Sometimes the pre- and postconditions require additional variables that do not appear in the program.

These are called logical variables (or auxiliary variables).

Inference rule for assignment

$$\frac{}{(\!|Q[E/x]|\!)\,x=E;\,(\!|Q|\!)}$$

How to read program verification inference rules: "If the condition(s)/Hoare triples above the horizontal line are proved, then the Hoare triples above the horizontal line are proved, then the Hoare triple under the horizontal line holds."

Intuition for the assignment rule: Normally, Q is a relation depending on the variable x. If we denote this by writing $Q(x)$, then the assignment rule informally means that the following statement holds, with no assumptions: "$Q(x)$ will hold after assigning (the value of) E to x, if $Q(E)$ was true beforehand."

We read the stroke "/" as "in place of". Thus, $Q[E/x]$ is read as "Q with E in place of x," and it denotes the result of substituting in Q all occurrences of x by E. Here x is a free variable.

Example:

Prove that the following Hoare triple is satisfied under partial correctness

$(\!|y+1=7|\!)$ `x = y + 1` $(\!|x=7|\!)$

Solution:

The partial correctness is formally proved by one application of the (sound) assignment inference rule, with $Q(x)$ being "$x=7$", and E being "$y+1$".

The assignment rule is applied backwards: The right way to understand it is to think about what we would have to prove about the initial state, in order to prove that Q holds in the resulting state.

Since Q will be in general depending on x, whatever it says about x must have been true for E, since in the resulting state the value of x is E.

Thus, "Q with E in place of x" must be true of the initial state.

Example 1:

$(\!|y = 2|\!)$ $(\!|Q[E/x]|\!)$

`x = y;` `x = E;`

$(\!|x = 2|\!)$ $(\!|Q|\!)$

If we want to prove that $x = 2$ after the assignment whereby x takes value y, then we must have proved $y = 2$ before it.

Here $Q(x)$ is "$x = 2$", E is y, $Q[y/x]$ is "$y = 2$".

Example 2:

$(\!|0 < 2|\!)$ $(\!|Q[E/x]|\!)$

`x = 2;` `x = E;`

$(\!|0 < x|\!)$ $(\!|Q|\!)$

If we want to prove that $0 < x$ after the assignment whereby x takes value 2, we must have proved $0 < 2$ before it.

Here $Q(x)$ is "$0 < x$", E is 2, $Q[2/x]$ is "$0 < 2$".

Implied Rule of "precondition strengthening":

$\frac{P \to P' \quad (\!|P'|\!)C(\!|Q|\!)}{(\!|P|\!)C(\!|Q|\!)}$ implied

Implied Rule of "postcondition weakening":

$\frac{(\!|P|\!)C(\!|Q'|\!) \quad Q' \to Q}{(\!|P|\!)C(\!|Q|\!)}$ (implied)

The implied rules allow us to import formal deduction proofs from first-order logic, $\emptyset \vdash P \vdash P', \emptyset \vdash Q' \implies Q$, (enhanced with basic facts of arithmetic) into proofs in formal program verification.

Note that the first implied rule allows us the precondition to be strengthened (thus, we assume more than we need to), while the second implied rule allows the postcondition to be weakened (i.e., we conclude less than we are entitled to).

Example: Show that the program "$x = y + 1$" satisfies the specification $(\!|y = 6|\!)x = y + 1(\!|x = 7|\!)$ under partial correctness.

$(\!|y = 6|\!)$

$(\!|y + 1 = 7|\!)$ implied

`x = y + 1;`

$(\!|x = y|\!)$ assignment

Here the strengthened precondition is P is $y = 6$, the precondition P' is $y + 1 = 7$, the program C is $x = y + 1$, and the postcondition Q is $x = 7$.

Note that here we have $\emptyset \vdash P \iff P'$.

Example: Show that the program "$x = y + 1$" satisfies the specification $(\!|y + 1 = 7|\!)x = y + 1(\!|x \leq 7|\!)$ under partial correctness.

$(\!|y + 1 = 7|\!)$

`x = y + 1;`

$(\!|x = 7|\!)$ assignment

$(\!|x \leq 7|\!)$ implied

Here the precondition P is $y + 1 = 7$, the program C is $x = y + 1;$, the postcondition Q' is $x = 7$, and the weakened postcondition Q is $x \leq 7$.

In this case, $\emptyset \vdash Q' \to Q$, but the converse $\emptyset \vdash Q \to Q'$ does not hold.

Inference rule for instruction composition

$\frac{(\!|P|\!)C_1(\!|Q|\!), \quad (\!|Q|\!)C_2(\!|R|\!)}{(\!|P|\!)C_1;C_2(\!|R|\!)}$ (composition)

In order to prove $(\!|P|\!)C_1; C_2(\!|R|\!)$, whereby the program consists of a sequence (composition) of two instructions C_1 and C_2, we need to:

- Find a midcondition Q for which
- We can prove $(\!|P|\!)C_1(\!|Q|\!)$, and
- We can prove $(\!|Q|\!)C_2(\!|R|\!)$

Inference rules applied to a composition/sequence of instructions allows us to "string together" pre/post-conditions and lines of code/

Each condition is the postcondition of the previous line of code and the precondition of the next line of code.

Interleave program statements with assertions (= conditions), each justified by an inference rule.

The composition rule is implicit.

Each assertion should hold whenever the program reaches that point in its execution.

Each assertion (condition) is justified by an inference rule.

If the implied reference rule is used, we also need to prove a (first-order logic) formal proof of the implication $\emptyset \vdash P \to P'$ or $\emptyset \vdash Q' \to Q$. Usually, we do these proofs separately, after annotating the program.

Example 1: Show that the program "$x = y + 1$" satisfies the specification $(\!|y = 5|\!)x = y + 1(\!|x = 6|\!)$ under partial correctness.

$(\!|y = 5|\!)$

$(\!|y + 1 = 6|\!)$ implied

```
x = y + 1;
```

$(\!|x = 6|\!)$ assignment

The proof is constructed from the bottom upwards.

We start with $x = 6$ and, using the assignment rule, we "push it upwards", "through" the assignment that gives x value $y + 1$.

This means substituting $y + 1$ for all occurrences of x, resulting in $y + 1 = 6$.

Now compare this with the given precondition $y = 5$.

The given precondition $y = 5$ and the arithmetic fact that $5+ = 6$ imply $y + 1 = 6$, so we have finished the proof.

Although constructed bottom-up, its justifications make sense when read top-down.

The second line is implied by the first line.

The fourth line followed from the second, by the intervening assignment which gives x value $y + 1$.

Note that implied always refers to the immediately preceding line.

Programs with Conditional Statements:

if-then-else:

$\frac{(\!|P \wedge B|\!)C_1(\!|Q|\!) \quad (\!|P \wedge \neg B|\!)C_2(\!|Q|\!)}{(\!|P|\!) \text{ if } (B) C_1 \text{ else } C_2(\!|Q|\!)}$ (if then else)

if-then (without else):

$\frac{(\!|P \wedge B|\!)\, C\, (\!|Q|\!) \quad (P \wedge \neg B) \to Q}{(\!|P|\!) \text{ if } (B) C (\!|Q|\!)}$ (if-then)

Annotated program template for if-then-else:

$(\!|P|\!)$

```
if ( B ) {
```

$(\!|P \wedge B|\!)$

```
    C_1
```

$(\!|Q|\!)$

```
} else {
```

$(\!|P \wedge \neg B|\!)$

```
    C_2
```

$(\!|Q|\!)$

```
}
```

$(\!|Q|\!)$

Example: Prove that the program below satisfies the specifications under partial correctness.

$(\!|true|\!)$

```
if ( max < x ) {
    max = x;
}
```

$(\!|\max \geq x|\!)$

Let's recall our proof method.

The three steps of a proof of partial correctness:

1. First annotate the program using the appropriate inference rules.
2. Then "back up" in the proof: Add an assertion/condition before each assignment statement, based on the assertion/condition following the assignment.
3. Finally prove any "implieds"

Proofs here can use first-order logic, basic arithmetic, or any other appropriate reasoning.

$(\!|true|\!)$

```
if ( max < x ) {
```

$(\!|true \wedge \max < x|\!)$ if-then

$(\!|x \geq x|\!)$ Implied (a)

```
    max = x;
```

$(\!|\max \geq x|\!)$ assignment

```
}
```

$(\!|\max \geq x|\!)$ if-then

Implied (b) $(true \wedge \neg(\max < x)) \to \max \geq x$

The auxiliary "implied" proofs can be done using formal deduction in first-order logic (and assuming the necessary arithmetic properties). We will write them formally, or informally but clearly.

Proof of Implied (a):

$$\emptyset \vdash ((true \wedge (\max < x)) \to x \geq x$$

Clearly, $x \geq x$ holds (basic arithmetic), and thus the required implication holds.

Proof of Implied (b): Show $\emptyset \vdash (P \wedge \neg B) \rightarrow Q$, which in this case is

$$\emptyset \vdash (true \wedge \neg(\max < x)) \rightarrow (\max \geq x)$$
$$1.(true \wedge \neg(\max < x)) \vdash (true \wedge \neg(\max\, x))(\in)$$
$$2.(true \wedge \neg(\max\, x)) \vdash \neg(\max < x))(1, \wedge -1)$$
$$3.(true \wedge \neg(\max < x)) \vdash (\max \geq x)$$
$$4.\emptyset \vdash (true \wedge \neg(\max < x)) \rightarrow (\max \geq x)(3, \rightarrow +)$$

"Partial while" (does not require termination)

$\frac{(\!|I \wedge B|\!)\, C\, (\!|I|\!)}{(\!|I|\!) \text{ while } (B) C (\!|I \wedge \neg B|\!)}$ (partial-while)

Intuitively: If the code C satisfies the tripe under partial correctness then, no matter how many times C is executed, if I was true initially and the while-statement terminates, then I will be true at the end.

Condition I is called a loop invariant.

Annotations for partial-while:

$(\!|P|\!)$

$(\!|I|\!)$ Implied (a)

```
while ( B ) {
```

$(\!|I \wedge B|\!)$ partial-while

```
  C
```

$(\!|I|\!)$

```
  }
```

$(\!|I \wedge \neg B|\!)$ partial-while

$(\!|Q|\!)$ Implied (b)

(a) Prove $P \rightarrow I$ (precondition P implies the loop invariant)

(b) Prove $(I \wedge \neg B) \rightarrow Q$ (exit condition implies postcondition)

We need to determine/find the loop invariant I.

A loop invariant is an assertion (condition) that is true both before and after each execution of the body of a loop.

- True before the while-loop begins
- True after the while-loop ends
- It expresses a relationship among the variables used within the body of the loop. Some of these variables will have their values changed within the loop.
- An invariant may or may not be useful in proving termination.

$(\!|x \geq 0|\!)$

```
y = 1;
z = 0;
-> while (z != x) {
    z = z + 1;
    y = y * z;
}
```

$(\!|y = x!|\!)$

From the trace of the loop and the postcondition, a candidate loop invariant is $y = z!$

$(\!|x \geq 0|\!)$

$(\!|1 = 0!|\!)$ implied (a)

```
y = 1;
```
$(\!|y = 0!|\!)$
```
z = 0;
```
$(\!|y = z!|\!)$
```
while (z != x) {
```
$(\!|(y = z!) \wedge \neg(z = x)|\!)$ partial-while $((\!|I \wedge B|\!))$

$(\!|y(z+1) = (z+1)!|\!)$ implied (b)
```
    z = z + 1;
```
$(\!|yz = z!|\!)$
```
    y = y * z;
```
$(\!|y = z!|\!)$
```
}
```
$(\!|y = z! \wedge z = x)|\!)$ partial-while $((\!|I \wedge \neg B|\!))$

$(\!|y = x!|\!)$ implied (c)

Proof of implied (a): $(x \geq 0) \vdash (1 = 0!)$. By definition of factorial.

Proof of implied (c): $((y = z!) \wedge (z = x)) \vdash (y = x!)$

$$
\begin{aligned}
&1.(y = z!) \wedge (z = x) \vdash (y = z!) \wedge (z = x)(\in)\\
&2.(y = z!) \wedge (z = x) \vdash (y = z!)(1, \wedge -)\\
&3.(y = z!) \wedge (z = x) \vdash (z = x)(1, \wedge -)\\
&4.(y = z!) \wedge (z = x) \vdash (y = x!)(2, 3, \approx -)
\end{aligned}
$$

Proof of implied (b): $((y = z!) \wedge \neg(z = x)) \vdash (z+1)y = (z+1)!$

$$
\begin{aligned}
&1.y = z! \wedge z \neq x \vdash y = z! \wedge z \neq x(\in)\\
&2.y = z! \wedge z \neq x \vdash y = z!(1, \wedge -))\\
&3.y = z! \wedge z \neq x \vdash (z+1)y = (z+1)z!(2, \text{ algebra})\\
&4.y = z! \wedge z \neq x \vdash (z+1)z! = (z+1)! \text{ (def. of factorial}, +)\\
&5.y = z! \wedge z \neq x \vdash (z+1)y = (z+1)!(3, 4, \text{ transitivity of equality})
\end{aligned}
$$

Total Correctness = Partial Correctness + Termination

Only while-loops can be responsible for non-termination in our programming language.

Proving termination: For each while-loop in the program: Identify an integer expression which is always non-negative and whose values decreases every time through the while-loop.

Total Correctness Problem: Is a given Hoare triple $(\!|P|\!)C(\!|Q|\!)$ satisfied under total correctness?

Theorem: The Total Correctness Problem is undecidable.

Proof: Reduce the Blank-Tape Halting Problem to our problem:

- Suppose we have a terminating algorithm A to solve the Total Correctness Problem
- We can use it to solve the Blank-Tape Halting Problem
- Given program C as input, construct a program C' that erases any input x to C, and then runs C on the blank tape.
- We can now use our algorithm A to test if $(\!|true|\!)C'(\!|true|\!)$ is totally correct.

- Claim: The program C' halts on a blank take iff the Hoare triple $(\!|true|\!)C'(\!|true|\!)$ is totally correct.
- Contradiction, since the Blank-Tape Halting Problem is undecidable.

Partial Correctness Problem: Is a given Hoare triple $(\!|P|\!)C(\!|Q|\!)$ satisfied under partial correctness? Theorem: The Partial Correctness Problem is undecidable.

Proof: Reduce the Blank-Tape Halting Problem to our problem.

- Suppose we have a terminating algorithm A to solve the Partial Correctness Problem. We can use it to solve the Blank-Tape Halting Problem for any program C as follows.
- Given program C as input, make a new program C' by adding the new line "$x = 1$;" to the end of C (here x is a new variable).
- Claim: The program C does not halt on a blank take iff the Hoare Triple $(\!|true|\!)C'(\!|x = 0|\!)$ is partially correct.
- Contradiction, since the Blank-Tape Halting Problem is undecidable.