CS 245

Notes

Author
Steven Cao
University of Waterloo
Fall 2023

Contents

1 Introduction

1.1 Aristotelian logic

Definition 1.1 (Syllogism)**.** A **syllogism** is a logical argument in which one position (conclusion) is inferred from two or more others (premises) of a specific form.

Example 1.1. Example of a syllogism:
All humans are mortal
Socrates is human

Socrates is mortal

Syllogism is good reasoning because it is *truth-perserving*.

Correctness of an argument depends on form (structure), not content.

1.2 Propositional logic

Definition 1.2 (Proposition)**.** A **proposition** is a statement that is either true or false.

We can assign propositions to propositional variables such as p, q, r.

1 (true), 0 (false) are propositional constants. Any propositional variable can be assigned the value of either 1 or 0.

Propositional variables are **atomic propositions**, meaning they cannot be futher subdivided.

Compound propositions are obtained by combining several atomic propositions.

Definition 1.3 (Logical connective)**.** A **logical connective** is a word that combines propositions.

"or", "and", etc. are examples of logical connectives.

Statements formulated in natural languages are often ambiguous, so we will use mathematical symbols instead.

Definition 1.4 (Propositional symbols)**.** Here are some common propositional symbols.
$\neg p$ is the **negation** of p, meaning "not q".
$p \wedge q$ is the **conjunction** of p and q, meaning "p and q".
$p \vee q$ is the **disjunction** of p and q, meaning "p or q inclusive".

Definition 1.5 (Implication)**.** $p \to q$ is false when p is true and q is false, and true otherwise. $p \to q$ is the **implication** of p and q, meaning "if p then q".

If p is false, then $p \to q$ is *vacuously true*.

$\neg$ is an *unary connective*, while other connectives are *binary connectives*.

The binary connectives $\wedge, \neg, \leftrightarrow$ are *symmetric*, meaning the two proposition joined by the connective can be swapped without affecting the truth value. However, $\to$ is not symmetric.

2 Propositional language

By using connectives, we can combine propositions.

To prevent ambiguity, we use **fully parenthesized expressions**, which can be parsed in a unique way.

Definition 2.1. $\mathcal{L}^p$ is the formal language of propositional logic.
Strings in $\mathcal{L}^p$ are made of three classes of symbols:

1. Proposition symbols: $p, q, r, \ldots$
2. Connective symbols: $\neg, \wedge, \vee, \to, \leftrightarrow$
3. Punctuation symbols: ()

Definition 2.2. $\text{Atom}(\mathcal{L}^p)$ is the set of atoms, or atomic formulas in $\mathcal{L}^p$.

Definition 2.3. $\text{Form}(\mathcal{L}^p)$ is the set of formulas in $\mathcal{L}^p$. It is defined recursively:
Base case: Let $A \in \text{Atom}(\mathcal{L}^p)$. Then, $a \in \text{Form}(\mathcal{L}^p)$.
Recursive: Let $A, B \in \text{Form}(\mathcal{L}^p)$. Then,

1. $(\neg A) \in \text{Form}(\mathcal{L}^p)$.
2. $(A \wedge B) \in \text{Form}(\mathcal{L}^p)$.
3. $(A \vee B) \in \text{Form}(\mathcal{L}^p)$.
4. $(A \to B) \in \text{Form}(\mathcal{L}^p)$.
5. $(A \leftrightarrow B) \in \text{Form}(\mathcal{L}^p)$.

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

2.1 Parsing

We can create a parse tree of any formula.

Example 2.1. $((p \wedge (\neg q)) \to r)$ can be expressed as the following parse tree:

Theorem 2.1 (Unique readability theorem)**.** Every formula in $\mathcal{L}^p$ is of exactly one of these forms:

1. An atom
2. $(\neg A)$
3. (AcB), where c is one of $\wedge, \vee, \rightarrow, \leftrightarrow$

Proof. We will prove this using **structural induction**.

Let $P(n)$ be the property: every formula A containing at most n connectives satisfies:

1. The first symbol of A is either a left parenthesis or a proposition symbol.
2. A has an equal number of left and right parentheses, and each non-empty proper initial segment of A has more left parentheses than right parentheses.
3. A has a unique construction as a formula.

We will prove $P(n)$ holds for all n.

Base case: Let $n = 0$. Then, a formula with 0 connectives must be a propositional symbol, have 0 parentheses, and has non non-empty proper initial/terminal segments.

Inductive step: □

2.2 Semantics

While **syntax** is concerned with the rules used for constructing a formula, **semantics** is concerned with the meaning of the formula.

We can use a **truth table** to represent the meaning of a formula.

Definition 2.4 (Truth valuation)**.** A **truth valuation** is a function $t : \text{Atom}(\mathcal{L}^p) \to \{0, 1\}$, which corresponds to a single row in the truth table.

Definition 2.5 (Satisfiability)**.** A set of formulas Σ is **satisfiable** iff there exists a truth valuation t such that $\Sigma^t = 1$.
Otherwise, Σ is **unsatisfiable**.

Definition 2.6 (Tautology, contradiction, contingent)**.** Let A be a formula.
A is a **tautology** iff for all truth valuation t, $A^t = 1$.
A is a **contradiction** iff for all truth valuation t, $A^t = 0$.
A is **contingent** if it is neither a tautology nor a contradiction.

Plato has three essential laws of thought regarding tautologies.

1. Law of identity: $p = p$
2. Law of contradiction: $\neg(p \wedge \neg p)$

3. Law of excluded middle: $(p \vee \neg p)$

Definition 2.7 (Tautological consequence)**.** Let Σ, A be formulas. A is a **tautological consequence** of Σ ($\Sigma \models A$) iff for any truth valuation t, $\Sigma^t = 1$ implies $A^t = 1$.

In the special case $\emptyset \models A$, this means the truth of A is unconditional. This means A is a tautology.

Definition 2.8 (Tautological equivalence)**.** Let A, B be formulas.
A, B are **tautologically equivalent** ($A \models\!\models B$) iff $A \models B \wedge B \models A$.

Note tautological equivalence is weaker than formula equality, as different formulas can be tautologically equivalent if their truth tables are the same.

Theorem 2.2 (Replaceability of tautologically equivalent formulas)**.** Let A be a formula which contains a subformula B.
Let C be a formula such that $B \models\!\models C$. Let A' be the formula obtained by replacing some occurences of formula B by C. Then, $A' \models\!\models A$.

Theorem 2.3 (Duality)**.** Let A be a formula composed only of atoms and connectives $\neg, \wedge, \vee$.
Let $\Delta(A)$ be the formula made from replacing all occurences of $\wedge$ with $\vee$, $\vee$ with $\wedge$, and each atom with its negation. Then, $\neg(A) \models\!\models \Delta(A)$.

3 Propositional calculus

In standard algebra, we rely on many algebraic identities to manipulate expressions, similar to using tautological equivalences to manipulate formulas.

Using $\rightarrow$ and $\leftrightarrow$ can be cumbersome, so we may seek to simplify formulas with these connectors.

To remove $\rightarrow$, use the equivalence

$$A \rightarrow B \models\!\models \neg A \vee B$$

To remove $\leftrightarrow$, there are two ways depending on if we desire and disjunction or a conjunction.

$$A \leftrightarrow B \models\!\models (A \wedge B) \vee (\neg A \wedge \neg B) \models\!\models (\neg A \vee B) \wedge (\neg B \vee A)$$

3.1 Laws of propositional calculus

The laws allow us to simply formulas. All laws can be proven by comparing truth tables.

All laws come in pairs, called **dual pairs**: for each formula depending only on connectives $\neg, \wedge, \vee$, a dual can be created by swapping $0, 1$ and $\wedge, \vee$.

3.2 Normal forms (DNF, CNF)

Definition 3.1 (Normal forms)**.** A disjuncton with conjunctive clauses as its disjuncts is in **disjunctive normal form** (DNF). A conjunction with disjunctive clauses as its conjuncts is in **conjunctive normal form** (CNF).

To convert a formula into CNF, we can use this algorithm.

1. Eliminate equivalence and implication.
2. Remove instances of $\neg$ that does not have an atom as its scope.
3. Perform recursive procedure $\text{CNF}(A)$:
   (a) If A is a literal, return A.
   (b) If $A = B \wedge C$, return $\text{CNF}(B) \wedge \text{CNF}(C)$.
   (c) If $A = B \vee C$, then let $\text{CNF}(B) = \bigwedge_{i=1}^{n} B_i, \text{CNF}(C) = \bigwedge_{j=1}^{m}$. Return $\bigwedge_{(i,j)=(1,1)}^{(n,m)} (B_i \vee C_j)$.

Theorem 3.1 (Existance of normal forms)**.** Any formula A is tautologically equivalent to some formula in DNF and some formula in CNF.

We can also obtain a formula in DNF using a truth table. To do this, find all truth valuations t that produces true. Then, for each t, create a conjunction using the truth values of each atom that would produce true. Finally, create a disjunction using all conjunctions created in this way as disjuncts.

3.3 Connectives

Definition 3.2 (Definability)**.** A connector f is **definable** in terms of a set of connectors S if there a formula with only connectors in S that is tautologically equivalent a formula with only connector f.

Formulas $A \to B$ and $\neg A \wedge B$ are tautologically equivalent., therfore $\to$ is definable (reducible) in terms of $\neg, \vee$.

Theorem 3.2 (n-ary connectives)**.** There are $2^{(2^n)}$ distinct n-ary connectives.

Definition 3.3 (Adequate set of connectives)**.** A set of connectives S is **adequete** iff it is able to express any truth table. Or, any n-ary connective can be defined in terms of connectives in S.

The set of five standard connectives $\{\neg, \wedge, \vee, \to, \leftrightarrow\}$ is adequate.

$\{\neg, \wedge, \vee\}$ is also adequate.

The NOR operator alone $\{\downarrow\}$ is adequete. So is the NAND operator $\{|\}$.

To prove adequacy, we can show that the set is definable in terms of one of the sets that are already adequate.

To prove inadequacy, we can find a truth valuation that is impossible to be expressed with the connectors in the set.

3.4 Boolean algebra

Definition 3.4 (Boolean algebra)**.** A **Boolean algebra** is a set $B = \{0, 1\}$ closed under operations $+, \cdot, \bar{}$.

Form($\mathcal{L}^p$) with operators $\vee, \wedge, \neg$ is a Boolean algebra. Sets with operators $\cup, \cap, ^c$, Boolean algebra.

Boolean algebra is used to modle the circuitry of electronic computers, which are devices with inputs and outputs from the set $\{0, 1\}$.

Definition 3.5 (Boolean variable)**.** A **boolean variable** is a variable in $\{0, 1\}$.

Definition 3.6 (Boolean function)**.** An n-var **boolean function** is a function $f : \{0, 1\}^n \to \{0, 1\}$.

The basic elements of circuits are called **logic gates**, which implement the three Boolean operators. A logic gate is a device that acts like a Boolean function.

3.5 Transistor

Logic gates are physically implemented by transistors. A **transistor** has three lines: control (in), collector (in), emitter (out).

The input value on the control line represents the state of the transistor: 1 allows current to flow from collector to emitter, and 0 prevents the current flow.

Transistors are the building blocks of high-level computers today, but it is not the only possible device to accomplish this. Any **bistable device** can be used to build binary computers.

3.6 Basic logic gates

Definition 3.7 (NOT gate)**.** An **inverter**, or a **NOT gate**, is a logic gate that implements negation.
It is implemented using one transistor between output and ground.

Definition 3.8 (NOR gate)**.** A **NOR gate** is implemented with two transistors connected in parallel.

Definition 3.9 (OR gate)**.** An **OR gate** is implemented using a NOT gate connected at the output of a NOR gate.

Definition 3.10 (NAND gate)**.** A **NAND gate** is implemented with two transistors connected in series.

Definition 3.11 (AND gate)**.** An **AND gate** is implemented using a NOT gate connected at the output of a NAND gate.

3.7 Adders

Logic circuits can be used to carry out addition of two positive integers from their binary expansions.

Definition 3.12 (Half-adder)**.** A **half-adder** is used to add two bits.
It has two inputs x, y and two outputs s, c.
$s = \overline{x}y + x\overline{y} = (x + y)\overline{(xy)}$ (xor) represents the sum of x and y.
$c = xy$ (and) represents the carry from the addition.

We can make a half-adder using a NOT gate, two AND gates, and an OR gate.

Definition 3.13 (Full-adder)**.** A **full-adder** is used to add two bits and the carry from the previous adder.
It has three inputs x, y, c_i and two outputs s, c_{i+1}.

We can make a full-adder by chaining two half adders and an OR gate.

4 Formal deduction

4.1 Formal deducibility

Up until now, we have used semantic methods, such as truth tables and tautological consequence, to prove arguments.

We will now replace it with a purely syntactic method.

Definition 4.1 (Formal deducibility)**.** Formula A is formally deducible from Σ (written $\Sigma \vdash A$) iff $\Sigma \vdash A$ is generated by the rules of formal deduction.

Definition 4.2 (11 rules of formal deduction)**.** For all formulas A, B, C and any set Σ of formulas:

1. (Ref) $A \vdash A$.
2. $(+)$ If $\Sigma \vdash A$ then $\Sigma, \Sigma' \vdash A$.
3. $(\neg -)$ If $\Sigma, \neg A \vdash B$ and $\Sigma, \neg A \vdash \neg B$ then $\Sigma \vdash A$.
4. $(\to -)$ If $\Sigma \vdash A \to B$ and $\Sigma \vdash A$ then $\Sigma \vdash B$.
5. $(\to +)$ If $\Sigma, A \vdash B$ then $\Sigma, A \to B$
6. $(\wedge -)$ If $\Sigma \vdash A \wedge B$ then $\Sigma \vdash A$ and $\Sigma \vdash B$
7. $(\wedge +)$ If $\Sigma \vdash A$ and $\Sigma \vdash B$ then $\Sigma \vdash A \wedge B$
8. $(\vee -)$ If $\Sigma, A \vdash C$ and $\Sigma, B \vdash C$ then $\Sigma, A \vee B \vdash C$
9. $(\vee +)$ If $\Sigma \vdash A$ then $\Sigma \vdash A \vee B$ and $\Sigma \vdash B \vee A$
10. $(\leftrightarrow -)$ If $\Sigma \vdash A \leftrightarrow B$ and $\Sigma \vdash A$ then $\Sigma \vdash B$
11. $(\leftrightarrow +)$ If $\Sigma, A \vdash B$ and $\Sigma, B \vdash A$ then $\Sigma \vdash A \leftrightarrow B$

We can use these rules to prove new theorems.

Example 4.1. Prove the following theorem (membership rule):
$(\in)$ If $A \in \Sigma$ then $\Sigma \vdash A$.

Proof. Assume $A \in \Sigma$. Let $\Sigma' = \Sigma - A$.

Then,

$$\begin{aligned} A &\vdash A && \text{(Ref)} \\ A, \Sigma' &\vdash A && (+) \\ \Sigma &\vdash A \end{aligned}$$

□

4.2 Proofs

To check if a sequence of steps is a formal proof, check:

1. Rules of formal deduction are correctly applied at each step.
2. Last term of the formal proof is identical with the desired theorem.

Theorem 4.1 (Finiteness of premise set)**.** If $\Sigma \vdash A$, then there exists a finite $\Sigma^0 \subseteq \Sigma$ such that $\Sigma^0 \vdash A$.

This shows that any proof involves only finitely many steps and finitely many formulas in the premise set Σ.

Theorem 4.2 (Transitivity of deducibility (Tr.))**.** Let $\Sigma, \Sigma' \in \text{Form}(\mathcal{L}^p)$. If $\Sigma \vdash \Sigma'$ and $\Sigma' \vdash A$, then $\Sigma \vdash A$.

Theorem 4.3 (Reductio ad absurdum ($\neg+$))**.** If $\Sigma, A \vdash B$ and $\Sigma, A \vdash \neg B$, then $\Sigma \vdash \neg A$.

Note ($\neg-$) is stronger than ($\neg+$).

4.3 Syntactic equivalence

Definition 4.3. For formulas A, B,

$$A \dashv\vdash B$$

means $A \vdash B$ and $B \vdash A$. That is, A and B are **syntactically equivalent**.

Theorem 4.4 (Replaceability of syntactically equivalent formulas (Repl.))**.** Let $B \dashv\vdash C$. For any A, let A' be constructed from A by replacing some occurences of B by C. Then,

$$A \dashv\vdash A'$$

Similar to tautological equivalence, $\emptyset \vdash A$ iff $\Sigma \vdash A$ for any Σ. If $\emptyset \vdash A$, then A is **formally provable**.

For example, the law of non-contadiction is formally provable. That is, $\emptyset \vdash \neg(A \wedge \neg A)$.

4.4 Soundness and completeness of formal deduction

Why do we need formal deduction? To define a proof system for which we can prove formally (syntactically) everything that is correct semantically.

Consider a *system of formal deducibility*. For this system to be "good", it has to be connected to informal reasoning in the following sense:

1. It should not be able to formally prove incorrect statements (**soundness**)
2. It should be able to formally prove every correct statement (**completeness**)

Theorem 4.5 (Soundness theorem)**.** If $\Sigma \vdash A$, then $\Sigma \models A$, where $\vdash$ means the formal deduction based on the 11 given rules.

Theorem 4.6 (Completeness theorem)**.** If $\Sigma \models A$, then $\Sigma \vdash A$, where $\vdash$ means the formal deduction based on the 11 given rules.

5 Formal deduction systems

Definition 5.1 (Resolution theorem proving)**. Resolution theorem proving** is a method of **formal derivation** (formal deduction) that has the following features:

- Only formulas allowed are disjunction of literals.
- All formulas must be disjunctive clauses.
- There is only one rule of formula deduction: **resolution**.

To prove an argument $A_1, \ldots, A_n \models C$ is valid, show the set $\{A_1, \ldots, A_n, \neg C\}$ is not satisfiable.

We can convert any formula into one or more disjunctive clauses. That is, convert the formula into conjunctive normal form. Each term of the conjunction is then made into a clause of its own.

Definition 5.2 (Resolution)**. Resolution** is the formal deduction rule

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

where C, 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**. We say that we resolve two parent clauses over p.

The $\bot$ (contradiction) clause is always false (**empty clause**). The resolvent of $p, \neg p$ is the empty clause ($p, \neg p \vdash_r \bot$).

Example 5.1. Find the resolvent of $p \vee \neg q \vee r$ and $\neg s \vee q$.

The two parent clauses can be resolved over q, as q is negative in the first clause and positive in the second.

Then, the resolvent is $p \vee r \vee \neg s$.

5.1 Proving argument validity with resolution

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

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

we can derive, by $\vdash_r$, the empty clause $\bot$, by:

- Transform each formula in the set into conjunctive normal form.
- Make each disjunctive clause a distinct clause, used for input of **resolution procedure**.
- If the resolution procedure outputs the empty clause, this implies the set is inconsistent, hence not satisfiable, and thus the argument is valid.

The **resolution procedure** takes input a set of disjunctive clauses $S = \{D_1, \ldots, D_m\}$. Then, repeat to get $\bot$:

- Choose two parent clauses in S, 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.
- Otherwise, add D to S.

Example 5.2. Prove modus ponens:

$$p, p \to q \vdash_r q$$

Proof. By resolution,

(1) p (Premise)

(1) $\neg p \vee q$ (Premise)

(1) $\neg q$ (Negation of conclusion)

(1) q (Resolvent of $1, 2$ over p)

(1) $\bot$ (Resolvent of $3, 4$ over q)

□

Theorem 5.1 (Soundness of resolution formal deduction)**.** The resolvent is tautologically implied by its parent. Thus, resolution is a sound rule of formal deduction.

5.2 Set of support strategy

When doing resolution automatically, we need to decide the order to resolve the clauses. Depending on the order, it can greatly affect the time to find a contradiction.

Strategies include:

- Set-and-support strategy
- Davis-Putnam procedure (DPP)

In set-of-support strategy, we partition all clauses into two sets: the **set of support** and the **auxiliary set**.

The auxiliary set is a set of non-contradictory formulas.

At the start, use the set of premises as the auxiliary set, and the negation of the conclusion as the set of support.

We then perform all possible resolutions involving the set of support. The resolvent is added to the set of support.

Example 5.3. Prove p_4 from

$$p_1 \to p_2, \neg p_2, \neg p_1 \to p_3 \vee p_4, p_3 \to p_5, p_6 \to \neg p_5, p_6$$

using set-of-support.

Proof. Let the auxiliary set be $p_1 \to p_2, \neg p_2, \neg p_1 \to p_3 \vee p_4, p_3 \to p_5, p_6 \to \neg p_5, p_6$. Let the initial set of support be $\Sigma = \neg p_4$.

(1) $\neg p_1 \vee p_2$ (Premise)

(1) $\neg p_2$ (Premise)

(1) $p_1 \vee p_3 \vee p_4$ (Premise)

(1) $\neg p_3 \vee p_5$ (Premise)

(1) $\neg p_6 \vee \neg p_5$ (Premise)

(1) p_6 (Premise)

(1) $\neg p_4$ (Negation of conclusion) $\Sigma = \{7\}$

(1) $p_1 \vee p_3$ (Resolvent of $7, 3$) $\Sigma = \{7, 8\}$

(1) $p_2 \vee p_3$ (Resolvent of $1, 8$) $\Sigma = \{7, 8, 9\}$

(1) p_3 (Resolvent of $2, 9$) $\Sigma = \{7, 8, 9, 10\}$

(1) p_5 (Resolvent of $4, 11$) $\Sigma = \{7, 8, 9, 10, 11\}$

(1) $\neg p_6$ (Resolvent of $5, 11$) $\Sigma = \{7, 8, 9, 10, 11, 12\}$

(1) $\bot$ (Resolvent of $6, 12$)

□

Theorem 5.2 (Pigeonhole principle $\mathcal{P}_n$)**.** One cannot put $n + 1$ objects into n slots.

We can formulate the pigeonhole principle as a conjunction of formulas.

1. Choose propositional variables $p_{ij}, i \leq n \leq n + 1, i \leq j \leq n$.
2. Define p_{ij} as true if the i-th pigeon goes into the j-th slot.
3. Construct clauses for:
   (a) Each pigeon $i, 1 \leq i \leq n + 1$ goes into some slot $k, 1 \leq k \leq n$:

   $$p_{i1} \vee p_{i2} \vee \cdots \vee p_{in}$$

   for $1 \leq i \leq n + 1$.

(b) Distinct pigeons $i \leq j, 1 \leq i, j \leq n+1$ cannot go in the same slot k:

$$\neg p_{ik} \vee \neg p_{jk}$$

for $1 \leq i < j \leq n+1, 1 \leq k \leq n$.

Any truth valuation that satisfies the conjunction of all clauses would map $n+1$ pigeons one-to-one into n slots. By pigeonhole principle, this cannot be done, so the set of clauses is unsatisfiable.

5.3 Davis-Putnam procedure (DPP)

Any clause corresponds to a set of literals contained within the clause. Since the order and multiplicity (num of duplicates) of the literals is irrelevant, the set associated with the clauses completely determines the clause.

If clauses are represented as sets, we can write the resolvent on p of two clauses $C \cup \{p\}, D \cup \{\neg p\}$ where neither C, D is empty as

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

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

If C, D are both empty, then the resolvent of $\{p\}, \{\neg p\}$ is the empty clause $\perp$.

In DPP, given a non-empty set of clauses in the propositional variables $p_1, \ldots, p_n$, we repeat the following steps until there are no variables left:

1. Remove all clauses that have both both $q, \neg q$.
2. Choose a variable p appearing in one of the clauses.
3. Add to the set of clauses all possible resolvents using resolution on p.
4. Discard all (parent) clauses with p or $\neg p$ in them.
5. Discard any duplicate clauses.

This is referred as eliminating the variable p.

If in some step we resolve $\{p\}, \{\neg p\}$, then we obtain the empty clause $\{\perp\}$.

If there is no pair $\{p\}, \{\neg p\}$ to resolve, then all the clauses are discarded and the output will be no clauses.

The output of DPP is either $\{\neg\}$ or $\emptyset$.

Example 5.4. Apply DPP to the set of clauses

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

1. Eliminating p gives $\{q\}, \{\neg q, \neg r, s\}, \{r\}, \{\neg s\}$.
2. Eliminating q gives $\{\neg r, s\}, \{r\}, \{\neg s\}$.

3. Eliminating r gives $\{s\}, \{\neg s\}$.
4. Eliminating s gives $\{\bot\}$.

The output is an empty clause.

If the output of DPP is the empty clause $\{\bot\}$, then both $p, \neg p$ are produced. Therefore, the set of clauses obtained by preprocessing premises and negation of conclusion is inconsistent. Thus, the theorem is valid.

If the output of DPP is the no clause $\emptyset$, then no contradictions can be found. In this case, the theorem is invalid.

Theorem 5.3. 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 $\{\bot\}$.

6 First-order logic

In propositional logic, a simple proposition is an unanalyzed whole which is either true or false. There are certain arguments that cannot be expressed using propositional logic.

6.1 Elements of first-order logic

First-order logic is used to describe theories that comprise certain concepts specific to the structure or category.

For example:

- Domain of objects (e.g. set of natural numbers)
- Designated individuals (e.g. 0)
- Relations (e.g. equality)
- Functions (e.g. succession, addition, multiplication)

Definition 6.1 (Domain)**.** The **domain** (universe of discourse) is the collection of all things that affect the logical argument under consideration.

The elements of the domain are called **individuals**, which can be anything.

A domain must be non-empty.

Relations make statements about individuals. Each relation is given a name, which is then followed by an **argument list**.

Example 6.1. The argument "the sum of 2 and 3 is 5" can be written as $\text{sum2}(2, 3, 5)$. It has arity 3.

A one-place relation is a **property**.

Variables are used as a placeholder for an individual in relations.

Definition 6.2 (Atomic formula)**.** A relation name, followed by an argument list in parentheses is an **atomic formula**.

Atomic formulas take on true/false values.

Atomic formulas can be combined by logical connectives similar to propositions.

Example 6.2. Define atomic formulas $\text{Human}(\text{Socrates}), \text{Mortal}(\text{Socrates})$. The statement "if Socrates is human, then Socrates is a mortal" can be written as

$$\text{Human}(\text{Socrates}) \to \text{Mortal}(\text{Socrates})$$

We can write relations in a table/array. Relations of arity n can be represented in an n-dimensional array.

As in propositional logic, formulas can be given names.

6.2 Quantifiers

In statements, we often need to indicate how frequent certain things are true. In first-order logic, **quantifiers** are used in this context.

Definition 6.3 (Universal quantifier)**.** $\forall x A(x)$ indicates $A(u)$ is true for all possible values of u in the domain.

Example 6.3. Translate "everyone needs a break" into first-order logic.

Let the domain D be the set of all people. Define $B(u)$ to mean "u needs a break".

The translation is

$$\forall x B(x)$$

Definition 6.4 (Existential quantifier)**.** $\exists x A(x)$ indicates $A(u)$ is true for at least one u in the domain.

Example 6.4. Translate "some people like their tea iced" into first-order logic.

Let the domain D be the set of all people. Define $P(u)$ to mean "u likes their tea iced".

The translation is

$$\exists x P(x)$$

Variables that appear in a qualifier is **bound**. Otherwise, the variable is **free**.

Bound variables are local to the scope of the quantifier. Theefore, if several quantifiers use the same bound variable for quantification, then all these variables are local to their scope and are distinct.

$\forall x, \exists x$ are treated like unary connectives, and have higher precedence than all binary connectives.

We can quantify over a subset of the domain. For universal quantifiers, use implication ($\rightarrow$). And for existential quantifiers, use and ($\wedge$)

Quantifiers can be nested. However, multiple quantifiers are not commutative.

We can represent a universal quantifier in terms of an existential quantifier (or vice versa) by negating one.

$$\neg \forall x A(x) \models\!\models \exists x \neg A(x)$$

$$\neg \exists x A(x) \models\!\models \forall x \neg A(x)$$

This way, the universal quantifier is similar to a conjunction, and the existential quantifier is similar to a disjuncton.

6.3 Syntax

In first-order logic, we add the capacity to refer to individuals, and their properties and relationships. This means formulas are more fine-grained, with:

- Specification of basic individuals given by domain
- Terms, expresions referring to individuals
- Atomic formulas which use relations to combine terms
- Formulas which are recursively built starting from atomic formulas

We use two kinds of symbols:

- Logical symbols with fixed syntactic use and fixed semantic meaning (e.g. $\neg, \wedge, \forall$)
- Non-logical symbols (parameters) with designated syntax but not pre-defined semantic meaning (e.g. $<, +$)

We use $\mathcal{L}$ as the formal language for first-order logic.

It has logical symbols:

- Connectives: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
- Free variable symbols: $u, v, w, u_1, \ldots$
- Bound variable symbols: $x, y, z, x_1, \ldots$
- Quantifiers: $\forall, \exists$
- Puctuation symbols: (), “,”

and non-logcal symbols:

- Individual symbols: $a, b, c, a_1, \ldots$
- Relation symbols: $F, G, H, \ldots$
- Function symbols: $f, g, h, f_1, \ldots$

There is a special binary relation symbols called the **equality symbol** ($\approx$). $\mathcal{L}$ may or may not contain $\approx$. If $\mathcal{L}$ contains $\approx$, then it is called the **first-order language with equality**.

6.4 Terms

Definition 6.5 (Term)**.** 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 $t_1, \ldots, t_n, n \geq 1$ 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 variables are called **closed terms**.

Example 6.5. Consider the first-order language of elementary number theory.

- Equality relation: yes
- Relation symbols: $<$ (less than)
- Individual (constant) symbols: 0
- 1-place function symbols: s (successor)
- 2-place function symbols: $+, \times$

The expressions $\times(u, 0), s(s(0)), +(u, \times(v, w))$ are terms. They are usually written as $u{+}0, 2, u{+}vw$.

6.5 Formulas

Definition 6.6 (Atom)**.** An expression of $\mathcal{L}$ is an **atom** or **atomic formula** in Atom($\mathcal{L}$) iff it is of one of the following forms:

1. $F(t_1, \ldots, t_n), n \geq 1$ where F is an n-ary relation symbol and $t_1, \ldots, t_n$ are terms in Term($\mathcal{L}$).
2. $\approx (t_1, t_2)$ where t_1, t_2 are terms in Term($\mathcal{L}$), also written as $t_1 \approx t_2$.

In first-order language of elementary number theory, expressions $\approx (+(s(0), s(0)), s(s(0)))$ and $< (+(u, +(v, w)), +(u, w))$ are atoms.

Definition 6.7 (Formulas in $\mathcal{L}$)**.** A formula in Form($\mathcal{L}$) is one of:

1. An atom in Atom($\mathcal{L}$).
2. $\neg A$ for some $A \in \text{Form}(\mathcal{L})$.
3. $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, or $(A \leftrightarrow B)$ for some $A, B \in \text{Form}(\mathcal{L})$.
4. $\forall x A(x)$ or $\exists x A(x)$ for some $A(u) \in \text{Form}(\mathcal{L})$ in which every occurence of u in $A(u)$ is replaced with x.

Example 6.6. Translate this into first-order logic:
(1) For every integer x there is an integer which is greater than x.
(1) 500 is an integer.
(1) There is an integer which is greater than 500.
where

- $N(u)$: u is an integer
- $G(u, v)$: u is greater than v

The argument can be formalized as:

(1) $\forall x(N(x) \to \exists y(N(y) \wedge G(y, x)))$

(1) $N(500)$

(1) $\exists y(N(y) \wedge G(y, 500))$

Definition 6.8 (Sentence)**.** A **sentence** or **closed formula** in Form(L) is a formula in which no free variables occur.
The set of sentences of $\mathcal{L}$ is denoted by Sent($\mathcal{L}$).

We can define some other first-order languages.

Definition 6.9 (First-order language of group theory)**.** The first-order language of group theory has the following properties:

- Equality relation: yes
- Relation symbols: none
- Individual symbols: e (identity element)
- Function symbols: $^{-1}$ (inverse), $\circ$

Definition 6.10 (First-order language of set theory)**.** The first-order language of set theory has the following properties:

- Equality relation: yes
- Relation symbols: $\in$
- Individual symbols: $\emptyset$
- Function symbols: none

6.6 Semantics

A valuation for $\mathcal{L}$ consists of an interpretation of its non-logical symbols and with an assignment of values to its free variables. The valuation must contain enough information to determine if each formula in Form($\mathcal{L}$) is true or false.

Logical symbols in $\mathcal{L}$ have a fixed semantics:

- Connectives are interpreted as in propositional logic.
- Meaning of quantifiers are explained intuitively.
- Equality symbol $\approx$ denotes the relation "equal to".
- Variable symbols are interpreted as variables ranging over the domain.
- Punctuation symbols serve just like ordinary punctuation.

For non-logical symbols, a valuation consists of an **interpretation** plus an **assignment**.

An interpretation consists of:

- Non-empty set of individuals (domain).
- Specification for each individual symbol, relation symbol, and function symbol, of the actual individuals, relations and functions that each will denote.

An assignment assigns each free variable a value in the domain.

We denote the meaning given by valuation v to a symbol s by s^v.

Definition 6.11 (Valuation, FoL)**.** A **valuation** for first-order logic language $\mathcal{L}$ consists of:
- A domain $D \neq \emptyset$.
- A function v with properties:
  - For each individual symbol a and free variable symbol u, we have that $a^u, u^v \in D$.
  - For each n-ary relation symbol F, we have that $F^v \subseteq D^n$. In particular, $\approx^v = \{(x, x) | x \in D\} \subseteq D^2$.
  - For each m-ary function symbol f, we have that $f^v : D^m \to D$ is a total (never undefined) function.

It is very important that domain $D \neq \emptyset$. For example, "an existing unicorn exists" can be modelled as $\forall x E(x)$ where $E(x)$ means x exists. This is a vacuously true formula.

The purpose of a valuation is to provide semantic meaning to $\mathcal{L}$.

Definition 6.12 (Value of a term)**.** The **value of a term** t under valuation v over domain D, denoted by t^v, is defined as:
- If $t = a$ is an individual/free variable symbol a, then its value is $a^v \in D$.
- If $t = f(t_1, \ldots, t_m), m \geq 1, t_1, \ldots, t_m \in \text{Term}(\mathcal{L})$, then

$$f(t_1, \ldots, t_m)^v = f^v(t_1^v, \ldots, t_m^v)$$

Theorem 6.1. If v is a valuation over D and $t \in \text{Term}(D)$, then $t^v \in D$.

We can precisely express the truth value of a formula in $\text{Form}(\mathcal{L})$.

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 $u^{v(u/d)} = d$ (substituting u with d).

That is, for each free variable w,

$$w^{v(u/d)} = \begin{cases} d, & w = u \\ w^v, & \text{otherwise} \end{cases}$$

Definition 6.13 (Value of a quantified formula)**.** Let $A(x) \in \text{Form}(\mathcal{L})$. Let u be a free variable not occuring in $A(x)$.
The values of $\forall x A(x)$ under valuation v with domain D are given by:

$$\forall x A(X)^v = \begin{cases} 1, & A(u)^{v(u/d)} = 1 \text{ for all } d \in D \\ 0, & \text{otherwise} \end{cases}$$

$$\exists x A(X)^v = \begin{cases} 1, & A(u)^{v(u/d)} = 1 \text{ for some } d \in D \\ 0, & \text{otherwise} \end{cases}$$

$v(u/d)$ is needed because u is a free variable and $A(x)$ may also contain other free variables w. When we evaluate $A(u)$, unless we specify u is assigned value d (ranging over domain D), the default assignment of u by the valuation v would be the individual u^v.

Definition 6.14 (Value of a formula)**.** Let v be a valuation with domain D. The value of a formula in $\text{Form}(\mathcal{L})$ under v is defined as:

- $R(t_1, \ldots, t_n)^v = 1, n \geq 1$ iff $(t_1^v, \ldots, t_n^v) \in R \subseteq D^n$.
- $(\neg A)^v = 1$ iff $A^v = 0$.
- $(B \wedge C)^v = 1$ iff $B^v = 1$ and $C^v = 1$ (etc. for other logical connectives).
- $\forall x A(x)^v, \exists x A(x)^v$ as defined earlier.

Theorem 6.2. If v is a valuation over D and $A \in \text{Form}(\mathcal{L})$, then $A^v \in \{0, 1\}$.

Definition 6.15 (Satisfiability and (universal) validity)**.** 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 that $A^v = 1$.
- **Unsatisfiable** if $A^v = 0$ for all valuations v.

Definition 6.16 (Satisfiability of set of formulas)**.** A set $\Sigma \subseteq \text{Form}(\mathcal{L})$ is **satisfiable** iff there is some valuation v such that $\Sigma^v = 1$.

Universally valid formulas in $\text{Form}(\mathcal{L})$ is similar to tautologies in $\text{Form}(\mathcal{L}^p)$. However, verifying if a formula in $\text{Form}(\mathcal{L})$ requires all possible valuations over all possible domains to be checked, including infinite domains.

Theorem 6.3 (Church, 1936)**.** There is no algorithm for deciding the (universal) validity of formulas in first-order logic.

In **second-order logic**, we can also range over subsets of the domain and relations to the domain.

6.7 Logical consequence in FoL

Logical consequences in first-order logic involves semantics.

Same as before, we use ($\models$) to notate tautological consequence.

Definition 6.17 (Logical consequence)**.** Let $\Sigma \subseteq \text{Form}(\mathcal{L}), A \in \text{Form}(\mathcal{L})$. A is a logical consequence of Σ iff for any valuation v with $\Sigma^v = 1$, we have $A^v = 1$.

Example 6.7. Prove

$$\forall x \neg A(x) \models \neg \exists x A(x)$$

Proof. By contradiction, suppose for some valuation v over domain D,

$$(\forall x \neg A(x))^v = 1 \tag{1}$$

$$(\neg \exists x A(x))^v = 0 \tag{2}$$

Negating (2) gives

$$(\exists x A(x))^v = 1 \tag{3}$$

Then, for some $d \in D$,

$$A(d)^v = 1 \tag{4}$$

Negating (4) gives

$$(\neg A(d))^v = 0 \tag{5}$$

However, from (1), it means that

$$(\neg A(d))^v = 1 \tag{6}$$

(5), (6) creates a contradiction. □

To prove that a formula F is valid, show that $\emptyset \models A$.

The problem of proving if a formula is valid or not is **undecidable**. That is, there is no general algorithm that can determine if a formula is valid in all cases.

7 Formal deduction in first-order logic

We add 6 additional rules to formal deduction.

Definition 7.1 (Rules of formal deduction in FoL)**.** Let t be a term. In addition to the 11 rules of formal deduction, we have:

12. $(\forall -)$ If $\Sigma \vdash \forall x A(x)$ then $\Sigma \vdash A(t)$
13. $(\forall +)$ If $\Sigma \vdash A(u)$ and u does not occur in Σ then $\Sigma \vdash \forall x A(x)$
14. $(\exists -)$ If $\Sigma, A(u) \vdash B$ and u does not occur in Σ or in B then $\Sigma, \exists x A(x) \vdash B$
15. $(\exists +)$ If $\Sigma \vdash A(t)$ then $\Sigma \vdash \exists x A(x)$ (only some t needs to be replaced with x in $A(x)$)
16. $(\approx -)$ If $\Sigma \vdash A(t_1)$ and $\Sigma \vdash t_1 \approx t_2$ then $\Sigma \vdash A(t_2)$
17. $(\approx +)$ $\emptyset \vdash u \approx u$

For $(\exists +)$, substitution allows any number of t to be replaced by x.

For $(\forall +)$ and $(\exists -)$, substitution requires all u to be replaced by x.

7.1 Resolution in first-order logic

It's generally convenient to deal with formulas in which all quantifiers are moved to the front.

Definition 7.2 (Prenex normal form)**.** A formula is in **prenex normal form (PNF)** if it is of form

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n B$$

where $n \geq 1$, Q_i is a quantifier, B is quantifier-free.

$Q_1 x_1 Q_2 x_2 \dots Q_n x_n$ is the **prefix**, and B is the **matrix**.

We can algorithmically convert any formula into a logical equivalent formula in PNF:

1. Eliminate all occurences of $\rightarrow$ and $\leftrightarrow$.
2. Move all negations inward such negations only appear as part of literals.
3. Ensure distinct bound variables have different names (standard variables apart)
4. Move all quantifiers to the front

If formula A does not depend on x, then it can be quanitifed over x using any quantifier. Use this fact to move quantifiers to the front.

Additionally,

- $\forall x A(x) \wedge \forall x B(x) \models\!\!\!\models \forall x (A(x) \wedge B(x))$

- $\exists x A(x) \vee \exists x B(x) \models\!\models \exists x(A(x) \vee B(x))$
- $QxQyA(x,y) \models\!\models QyQxA(x,y)$
- $Q_1 xA(x) \wedge Q_2 yB(y) \models\!\models Q_1 xQ_2 y(A(x) \wedge B(y))$, where x, y is not in $B(y), A(x)$
- $Q_1 xA(x) \vee Q_2 yB(y) \models\!\models Q_1 xQ_2 y(A(x) \vee B(y))$, where x, y is not in $B(y), A(x)$

Example 7.1. Find the prenex normal form of

$$\forall x(\exists y R(x,y) \wedge \forall y \neg S(x,y) \to \neg \exists y \neg Q(x,y))$$

Eliminate $\to$:

$$\forall x(\neg(\exists y R(x,y) \wedge \forall y \neg S(x,y)) \vee \neg \exists y \neg Q(x,y))$$

Move negation inwards:

$$\forall x(\forall y \neg R(x,y) \vee \exists y S(x,y) \vee \forall y Q(x,y))$$

Standardize variables apart:

$$\forall x(\forall y_1 \neg R(x,y_1) \vee \exists y_2 S(x,y_2) \vee \forall y_3 Q(x,y_3))$$

Move quantifiers to front:

$$\forall x \forall y_1 \exists y_2 \forall y_3(\neg R(x,y_1) \vee S(x,y_2) \vee Q(x,y_3))$$

Definition 7.3 ($\exists$-free prenex normal form)**.** A sentence is in **$\exists$-free prenex normal form** if it's in PNF and does not contain and existential quantifier symbols.

Definition 7.4 (Skolem function)**.** Consider a sentence of form

$$\forall x_1 \ldots \forall x_n \exists y A$$

where $n \geq 0$ and A is an expression possibly involving other quantifiers.
$\exists y A$ generates at least one individual for each n-tuple $(a_1, \ldots, a_n)$ in the domain, which can be expressed using $f(x_1, \ldots, x_n)$.
f is a **Skolem function**.

Skolem functions can be used to eliminate all existential quantifiers.

The skolemized version of $\forall x_1 \ldots \forall x_n \exists y A$ is

$$\forall x_1 \ldots \forall x_n A'$$

where A' is the expression obtained from A by substituting each occurence of y by $f(x_1, \ldots, x_n)$.

Example 7.2. Skolemize $\forall x \exists y(x + y = 0)$ with domain $\mathbb{Z}$.

Each instance of $x = d$ generates a corresponding $y = -d$ that makes the formula true.

Define $f(x) = -x$. Then, we have the skolemized version of the formula:

$$\forall x(x + f(x) = 0)$$

More generally, the skolemized version of $\forall x \exists y P(x, y)$ is

$$\forall x P(x, g(x))$$

where $g(x)$ is the Skolem function generating a value $y = g(x)$ for each x.

Note that the sentence obtained using Skolem functions is not, in general, logically equivalent to the original sentence. This is because there can be more than one individual arising from the existential quantifier.

Now, we can convert a sentence into $\exists$-free PNF algorithmically:

1. Transform the sentence A_0 into a logically equivalent sentence A_1 in PNF.
2. Repeat until all $\exists$ are removed:
   (a) Assume A_i is of form $A_i = \forall x_1 \dots \forall x_n \exists y A$.
   (b) If $n = 0$, then A_i is of form $\exists y A$. Then, $A_{i+1} = A'$ where A' is obtained from replacing all occurences of y by some unused individual symbol c in A_i (Skolem function with no arguments).
   (c) If $n > 0$, then $A_i = \forall x_1 \dots x_n A'$ where A' is obtained from replacing all occurences of y by $f(x_1, \dots, x_n)$ for some unused function symbol f.

Example 7.3. Find the $\exists$-free PNF of the sentence

$$\exists x \forall y \forall z \exists s P(x, y, z, s)$$

Consider x. Replace x with individual symbol a:

$$\forall y \forall z \exists s P(a, y, z, s)$$

Consider s. Replace s with Skolem function $g(y, z)$:

$$\forall y \forall z P(a, y, z, g(y, z))$$

Theorem 7.1. Given sentence F, there is an effective procedure for finding an $\exists$-free prenex normal form formula F' such that F is satisfiable iff F' is satisfiable.

After all existential quantifiers have been eliminated through Skolem functions, we can "drop" the universal quantifiers. For example, $\forall y \forall z P(a, y, z, g(y, z))$ becomes $P(a, y, z, g(y, z))$.

Working under this convention, all variables are implicitly universally quantified.

Theorem 7.2. Given sentence F in $\exists$-free prenex normal form, one can construct a finite set C_F of disjunctive clauses such that F is satisfiable iff C_F is satisfiable.

Theorem 7.3. Let Σ be a set of clauses and A be a sentence. The argument $\Sigma \models A$ is valid iff the set

$$\left(\bigcup_{F \in \Sigma} C_F \right) \cup C_{\neg A}$$

is not satisfiable.

In resolution, we aim to reach the empty clause.

For propositional logic, it is impossible to derive a contradiction from a set of formulas unless the same variable occurs more than once. Similarly, in first-order logic, we cannot derive a contradiction from two formulas A, B unless A, B share complementary literals.

This cane be done by **unification**.

Definition 7.5 (Instantiation)**.** An instantiation $x_i := t_i'$ is an assignment to a variable x_i of a quasi-term t_i'.

Definition 7.6 (Unification)**.** Two formulas in FoL **unify** if there are instantiations that make the formulas identical.
The instantiation that unifies the formulas are call a **unifier**.

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 7.4. Show that $Q(a, y, z), Q(y, b, c)$ unify and give a unifier.

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 y_1. So we have $Q(a, y, z), Q(y_1, b, c)$.

Consider the instantiation $y := b, z := c$ for $Q(a, y, z)$ and $y_1 := a$ for $Q(y_1, b, c)$.

Theorem 7.4. A set S of clauses in first-order logic is not satisfiable iff there is a resolution derivation of the empty clause from S.

8 Computation and logic

8.1 Automatic theorem proving and verification

A theorem is a logical statement, with several premises and a conclusion.

To automatically prove that a theorem is valid, preprocess the formulas and use resolution.

Because statements of a formal theory (e.g. mathematics) are written in symbolic form, it is possible to verify mechanically that a formal proof of a statement, from a finite set of hypotheses, is valid.

This task is known as **automatic proof verification**, and is closely related to automatic theorem proving.

Automatic/manual theorem provers include: **Isabelle**, **Coq**, **E**, **SPASS**, **Vampire**.

CASC is a yearly competition of fully automatic theorem provers for classical first-order logic.

8.2 Algorithm

The basic resolution procedure is *not* an algorithm. We have not specified how to make choices (what clauses to unify/resolve), nor have a point we can conclude "satisfiable".

Definition 8.1 (Algorithm)**.** Informally, an **algorithm** is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or perform a computation.

All of these are equivalent:

- Programs written in C, Java, Python, etc.
- High level pseudo-code
- Turing machines

An algorithm solves a problem if, for every input, the algorithm produces the correct output.

8.3 Halting Problem

There are problems that cannot be solved by computer programs, even assuming infinite time and space.

Does there exist an algorithm that operates as:

- Input: program P, input I
- Output: "yes" if program P halts on input I, and "no" otherwise.

This is described as the Halting Problem.

Theorem 8.1. The Halting Problem is unsolvable.

Proof. By contradiction, assume there is a solution to the Halting Problem $H(P, I)$ that can determine whether or not a program halts, defined as

- Input: program P, input I
- Output: "halt" (yes) if P halts on input I, and "loops forever" (no) if P never stops on input I.

Note that we can represent a program P itself as an input.

Construct program $K(P)$ such that:

- If $H(P, P)$ outputs "halt", then $K(P)$ loops forever.
- If $H(P, P)$ outputs "loops forever", then $K(P)$ halts.

Consider $K(K)$. If $K(K)$ halts, then by definition $H(K, K)$ output "halt". So, $K(K)$ loops forever. This is a contradiction.

Otherwise, if $K(K)$ loops forever, then by definition, $H(K, K)$ loops forever. So, $K(K)$ halts. This is a contradiction.

In al cases, there is a contradiction. Thus, the halting algorithm $H(P, I)$ does not exist. □

8.4 Turing machine

A Turing machine is a simple mathematical model for algorithms and computation.

Definition 8.2 (Turing machine)**.** 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
- $f : S \times I \to S \times I \times \{L, R\}$, a partial function called the transition function, where L, R standard for direction "left", "right"

Consider a control unit and an infinite tape divided into cells with a finite number of non-blank symbols at any time.

At the beginning, T is in the initial state s_0, and is positioned at the initial position, the leftmost non-blank symbol on the tape.

For each step, the Turing machine reads to current tape symbol x.

If the control unit is in state s and the function f is defined by $f(s, x) = (s', x', d)$, then the control unit enters state s', writes x', then moves left/right depending on d.

The tuple (s, x, s', x', d) is the **transition rule** of the Turing machine.

If f is undefined for some pair (s, x), then T **halts**.

8.5 Languages

Definition 8.3 (Alphabet)**.** An **alphabet** Σ is a finite non-empty set of symbols (letters).

For example, $\Sigma = \{0, 1\}$ is an alphabet.

Σ^* denotes the set of all possible strings written with letters in Σ, including empty string (λ).

Definition 8.4 (Language)**.** A **language** L over Σ is a subset of L^*.

For example, we can define a language L over $\{0, 1\}$:

$$L = \left\{w \in \{0, 1\}^* : w \text{ contains an equal number of } 0, 1\right\}$$

Turing machines can be used to accept or recognize languages. To do this, we need to have a final state.

Definition 8.5 (Final state)**.** 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 8.6 (Accepts)**.** Let $V \subseteq I$. A Turing machine $T = (S, I, f, s_0)$ **accepts** a string $x \in V^*$ iff T, starting in the initial position when x is written on the tape, halts in a final state.
T accepts a language $L \subseteq V^*$ if x is accepted by T iff $x \in L$.

To accept a subset $L \subseteq V^*$, we can use symbols not in V. This means I may include symbols not in V. These symbols are often used as markers.

A string $s \in V^* \subseteq I^*$ is not accepted if, when started in the initial condition with x written on the tape, either:

- T does not halt
- T halts in a non-final state

There are also other ways to recognize langauges using a Turing machine; this is just one possible way.

A Turing machine can be represented by a **transition diagram**, where:

- Each state is represented by a node.
- The initial state is specified with an incoming arrow.
- The final state is specified with a double circle.
- A transition rule (s, x, s', x', d) is symbolized by an arrow between nodes s, s', and labelled with triplet $x/x', d$.

8.6 Decision problems

A Turing machine can be compute a partial function.

Suppose T, when given string x, halts at string y. We can 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 with x as input.

Turing machines can compute functions $f : \mathbb{Z} \to \mathbb{Z}$ if we specify an encoding for integers in unary notation.

Definition 8.7 (Decider)**.** A **decider** or **total Turing machine** is a Turing machine that never halts.

The total Turing machine is the formalization of a terminating algorithm.

It is possible to construct an **universal Turing machine** that can simulate the computations of every Turing machine when given an encoding of the Turing machine and its input. The universal Turing machine is the formalization of a computer: it can do whatever a computer can do.

The **Church-Turing thesis** states that: "any problem that can be solved with an algorithm can be solved by a Turing machine".

A Turing machine is equivalent in computing power to all the most general mathematical notions of computation (e.g. lambda calculus).

The Church-Turing thesis can be used to prove if a problem is solvable by a computer or not.

Definition 8.8 (Decision problem)**.** A **decision problem** is a yes/no question on an infinite set of inputs. Each input is an instance of the problem.

Satisfiability in first-order logic is a decision problem, where:

- Input: a formula $A \in \text{Form}(\mathcal{L})$
- Output: "yes" if A is satisfiable, and "no" otherwise.

Definition 8.9 (Decidable)**.** A decision problem is **decidable (solvable)** if there exists a terminating algorithm that solves it. Otherwise, the decision problem is **undecidable (unsolvable)**.

Definition 8.10 (Computable)**.** A function is **computable** if it can be computed by a Turing machine. Otherwise, the function is **uncomputable**.

Many decision problems can be stated in terms of if a natural number is in a subset of natural numbers.

Definition 8.11 (S-membership)**.** Let $S \subseteq \mathbb{N}$. The **S-membership problem** asks: "for any $x \in \mathbb{N}$, is $x \in S$?"

Definition 8.12 (S-membership decidability)**.** A set $S \in \mathbb{N}$ is decidable if the S-membership problem is decidable.

Example 8.1. Suppose $S \subseteq \mathbb{N}$ is decidable. Prove that its complement $S^c = \mathbb{N} \setminus S$ is decidable.

Proof. Define an algorithm such that:

Let x be given. If $x \in S$, then return "no". If $x \notin S$, then return "yes". □

There is no algorithm for:

- Checking if a set of first-order logic clauses is satisfiable.
- Checking if a formula in first-order logic is valid.

hencce these problems are undecidable.

To show that a problem is undecidable, we can use **reductions**.

Assume P_1 is undecidable. If there is a terminating algorithm that converts any instance of P_1 to an instance of P_2 with the same answer, then P_1 is reducible to P_2.

Theorem 8.2. If P_1 is reducible to P_2 and P_1 is undecidable, then P_2 is undecidable.

8.7 Computation

A Turing machine computation consists of successive applications of **transition rules** to the tape content.

One computation step (transition) is the application of one transition rule.

Example 8.2. Define a Turing machine that changes the first pair of consecutive 1s on the tape to 0s and the halts.

Define $T_1 = (S, I, f, s_0)$ with states $S = \{s_0, s_1, s_2, s_3\}$, alphabet $I = \{0, 1, B\}$, and transition function f with 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. $(s_2, 1, s_3, 0, R)$

We can write a Turing machine computation in another way.

Definition 8.13 (Configuration)**.** A **configuration** of a Turing machine $T = (S, I, f, s_0)$ is denoted as $\alpha_1 s \alpha_2$, where $s \in S$ is the current state, and α_1, α_2 is the string on I^* that consists of the current contents of the tape from the leftmost non-blank symbol to the rightmost non-blank symbol.

A computation of the Turing machine consistents of a succession of configurations, each obtained by one transition rule to the previous configuration.

$\to$ denotes the application of one transition rule, and $\to^*$ denotes zero or more applications.

Example 8.3. Write a computation of T_1 with input 010110.

$$s_0010110 \to 0s_010110 \to 01s_10110 \to 010s_0110 \to 0101s_110 \to 010s_2100 \to 0100s_300$$

We can also represent a Turing machine graphically.

State are represented by nodes in a transition diagram.

The starting state s_0 is depicted with an incoming arrow.

The final state is depicted with a double circle.

A transition rule (s_i, a, s_j, b, D) is represented by an arrow from s_i to s_j, labelled by $a; b; D$.

A computation corresponds to a path in the graph, beginning at the starting state.

Example 8.4. Construct a Turing machine T_2 that recognizes the set of strings in $\{0, 1\}^*$ that have a 1 as their second bit.

We want a Turing machine that determines if the 2nd symbol is a 1.

For the first symbol, we can define five-tuples that move right, increment the state, and leave the cell as is:

If the 2nd symbol is a 1, T_2 should move into a final state and halt. Otherwise, the Turing machine should halt in a non-final state.

So, we have

1. $(s_0, 0, s_1, 0, R)$

2. $(s_0, 1, s_1, 1, R)$
3. $(s_0, B, s_2, 0, R)$
4. $(s_1, 0, s_2, 0, R)$
5. $(s_1, 1, s_3, 1, R)$
6. $(s_1, B, s_2, 0, R)$

where s_3 is the final state.

T_2 will terminal in the finite state iff the input string has at least 2 bits, and the 2nd bit is a 1.

Example 8.5. Construct a Turing machine that recognizes the set $\{0^n 1^n : n \geq 1\}$

Let $V = \{0, 1\}$, $I = \{0, 1, M, B\}$ where M is used as a marker.

T successively replaces a 0 at the leftmost position with an M, then replaces a 1 at the rightmost position with an M. T terminates in a final state iff the string consists of a block of 0s followed by a block of the same number of 1s.

1. $(s_0, 0, s_1, M, R)$
2. $(s_1, 0, s_1, 0, R)$
3. $(s_1, 1, s_1, 1, R)$
4. (s_1, M, s_2, M, L)
5. (s_1, B, s_2, B, L)
6. $(s_2, 1, s_3, M, L)$
7. $(s_3, 1, s_3, 1, L)$
8. $(s_3, 0, s_4, 0, L)$
9. (s_3, M, s_5, M, R)
10. $(s_4, 0, s_4, 0, L)$
11. (s_4, M, s_0, M, R)
12. (s_5, M, s_6, M, R)

where s_6 is the final state.

To use a Turing machine to compute number-theoretic functions, we can represent the number with unary representation.

Definition 8.14 (Unary representation)**.** A natural number n in unary representation is represented as a string of $(n + 1)$ 1s.

To represent multiple numbers, we can separate them using an asterisk.

Definition 8.15. Construct a Turing machine that adds two natural numbers.

The Turing machine computes the function $f(n_1, n_2) = n_1 + n_2$.

The pair (n_1, n_2) is represented in unary represntation separated by an asterisk. So, the Turing machine takes this as input and produces an output of $(n_1 + n_2 + 1)$ 1s.

1. $(s_0, 1, s_1, B, R)$
2. $(s_1, *, s_3, B, R)$
3. $(s_1, 1, s_2, B, R)$
4. $(s_2, 1, s_2, 1, R)$
5. $(s_2, *, s_3, 1, R)$

where s_3 is the final state.

For more complex functions, we can use **multi-tape Turing machines** that uses more than one tape simulataneously.

Multi-tape Turing machines and other variations has the same computation power as a regular Turing machine.

The **Busy Beaver problem** is the problem of determining $B(n)$, the maximum number of 1s a non-halting Turing machine with n states with alphabet $\{1, B\}$ can print on a tape that is initially blank.

The Busy Beaver function $B(n)$ is uncomputable.

8.8 Complexity

Among decidable problems, we can rank the problems by "difficulty" in terms of how efficiently they can be solved.

Definition 8.16 (Computational complexity)**.** The **computational complexity** (time complexity) of an algorithm is the number of operations used by the algorithm, in terms of input size.

The computational complexity of a problem is the computational complexity of the most efficient algorithm that solves it.

Polynomial time is reasonably fast/tractable, and exponential time is too slow/intractable.

Definition 8.17 (Non-deterministic Turing machine)**.** In a **non-deterministic Turing machine** (NDTM), the allowed steps are defined as a **relation** consisting of five-tuples, rather than functions.

At each step of computation, a NDTM can pick any of the different choices of the transition rules that match the current state and tape symbol.

Definition 8.18 (Polynomial time problems)**.** A decision problem is in P, the class of **polynomial time problems**, if it can be solved (halt in final state) by a DTM in no more than $p(n)$ transitions, where p is a polynomial and n is the input length.

Definition 8.19 (Non-deterministic polynomial time problems)**.** A decision problem is in NP, the class of **non-deterministic polynomial time problems**, if it can be solved by a NDTM in no more than $p(n)$ transitions, where p is a polynomial and n is the input length.

Alternatively, NP is the set of decision problems for which the problem instances where the answer is "yes" have proofs **verifiable in polynomial time** by a DTM.

Clearly, $P \subseteq NP$ as a problem solvable in polynomial time can be verified in polynomial time with the solution.

Problems in P are **tractable** and problems not in P are intractable.

Definition 8.20 (NP-complete)**.** A problem in NP is **NP-complete** if every other problem in NP can be reduced to it in polynomial time.

Finding a polynomial time algorithm for any one of the NP-complete problems would prove that $P = NP$.

For SAT (Boolean satisfiability problem), the current best algorithm for solving it has a 2^n time complexity. However, verifying a solution takes only polynomial time.

SAT is NP-complete.

9 Peano Arithmetic and Gödel's Incompleteness Theorem

We first prove the basic properties of equality: reflexivity, symmetry, transitivity.

Example 9.1. Prove the reflexivity of equality,

$$\emptyset \vdash \forall x(x = x)$$

Proof.

$$\begin{array}{ll} \emptyset \vdash u = u & (=+) \\ \emptyset \vdash \forall x(x = x) & (\forall+) \end{array}$$

□

Example 9.2. Prove the symmetry of equality,

$$\emptyset \vdash \forall x \forall y((x = y) \to (y = x))$$

Proof.

$$\begin{aligned}
u = v &\vdash u = v && (\in)\\
\emptyset &\vdash u = u && (=+)\\
u = v &\vdash u = u && (+)\\
u = v &\vdash v = u && (=-)\\
\emptyset &\vdash (u = v) \to (v = u) && (\to+)\\
\emptyset &\vdash \forall y((u = y) \to (y = u)) && (\forall+)\\
\emptyset &\vdash \forall x \forall y((x = y) \to (y = x)) && (\forall+)
\end{aligned}$$

□

Example 9.3. Prove the transitivity of equality,

$$\emptyset \vdash \forall x \forall y \forall z((x = y) \wedge (y = z) \to (x = z))$$

Proof.

$$\begin{aligned}
u = v \wedge v = w &\vdash u = v \wedge v = w && (+)\\
u = v \wedge v = w &\vdash u = v && (\wedge-)\\
u = v \wedge v = w &\vdash v = w && (\wedge-)\\
u = v \wedge v = w &\vdash v = w && (\wedge-)\\
\emptyset &\vdash (u = v \wedge v = w) \to (v = w) && (=-)\\
\emptyset &\vdash \forall x \forall y \forall z(x = y \wedge y = z) \to (x = z) && (\forall+)
\end{aligned}$$

□

First-order logic is often used to describe **specialized domains**. In each case, we use a set of **domain axioms**, which are formulas assumed to be true in that domain.

A set of domain axioms, a system of formal deduction, and all theorems that can be formally proved from the domain axioms is a **theory**.

There are some requirements for the domain axioms. Let A be the set of domain axioms. Then,

- A should be **decidable**. There should exist a terminating algorith mto decide if a given formula is a domain axiom.
- A should be **consistent** (wrt $\vdash$).
- A should be **syntactically complete**. For any formula F describable in the language of the system, either F or $\neg F$ should be provable from A.

9.1 Peano Arithmetic

Definition 9.1 (Peano Arithmetic)**. Peano Arithmetic** is defined with:
- Individuals: 0
- Functions: s (successor), $+$ (addition), $\cdot$ (multiplication)
- Relations: equality
- Axioms: definitions for the functions, induction

The set PA of axiom in Peano Arithmetic contains:

$$\forall x \neg (s(x) = 0) \tag{PA1}$$
$$\forall x((s(x) = s(y)) \to (x = y)) \tag{PA2}$$
$$\forall x(x + 0 = x) \tag{PA3}$$
$$\forall x \forall y(x + s(y) = s(x + y)) \tag{PA4}$$
$$\forall x(x \cdot 0 = 0) \tag{PA5}$$
$$\forall x \forall y(x \cdot s(y) = x \cdot y + x) \tag{PA6}$$
$$(A(0) \wedge \forall x(A(x) \to A(s(x)))) \to \forall x A(x) \tag{PA7}$$

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. We can show this with the notation

$$\Sigma \vdash_{\text{PA}} C$$

9.2 Gödel's Incompleteness Theorem

10 Program verification

We want to verify a program for its **correctness**: does a given program satisfy its specification?

Techniques for showing program correctness:

- **Inspection**: walk through code
- **Testing**: develop tests for code
- **Formal program verification**: use a formal proof system to prove correctness

Testing is similar to checking that a propositional formula is a theorem by trying some truth valuations. Testing is **not enough** to prove correctness.

Steps for formal verification:

1. Convert the informal description R of requirements for an application to an "equivalent" formula ϕ_R of some symbolic logic.
2. Write a program P that realizes ϕ_R in some environment.
3. Prove that P satisfies the formula ϕ_R.

Example 10.1. List the states of the following code that computes the factorial of input x and stores in y:

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

State at the `while` test:

- s_0: $z = 0, y = 1$
- s_1: $z = 1, y = 1$
- s_2: $z = 2, y = 2$
- s_3: $z = 3, y = 6$
- s_4: $z = 4, y = 24$
- ...

Definition 10.1 (Hoare triple)**.** A **Hoare triple** is an assertion of form $(|P|)C(|Q|)$, where P is the precondition, C is the program, Q is the postcondition.

Definition 10.2 (Partial correctness)**.** A Hoare triple $(|P|)C(|Q|)$ is satisfied under **partial correctness**, denoted as

$$\models_{\text{par}} (|P|)C(|Q|)$$

iff for every state s that satisfies condition P, if the execution of program C starting from state s terminates in state s', then state s' satisfies condition Q.

Definition 10.3 (Specification)**.** A **specification** of program C is a Hoare triple $(|P|)C(|Q|)$ with the program C as the second component.

Example 10.2. Write the specification of the requirement: "if the input x is a positive number, compute a number whose square is less than x".

This can be expressed as the Hoare triple

$$(|x > 0|)C(|y \cdot y < x|)$$

Note that **specification is not behavior**.

The program

```
while true { x = 0; }
```

satisfies all specifications under partial correctness. This is because the program never terminates, so partial correctness is vacuously satisfied.

Definition 10.4 (Total correctness)**.** A Hoare triple $(|P|)C(|Q|)$ is satisfied under **total correctness**, denoted as

$$\models_{\text{tot}} (|P|)C(|Q|)$$

iff for every state s that satisfies condition P, execution of program C from state s terminates and the resulting state s' satisfies Q.

Total correctness is partial correctness with termination.

We can write preconditions and postconditions for partial and total correctness in first-order logic:

- Domain: programs, program states, conditions
- Relation State(s): s is a program state
- Relation Condit(P): P is a condition
- Relation Code(C): C is a program
- Relation Satisfies(s, P): state s satisfies condition P
- Relation Terminates(C, s): program C terminates when execution begins in state s
- Function result(C, s): the state that results from executing code C beginning in state s, if C terminates.

Both partial and total correctness is undecidable (halting problem), so proofs will need to be done on a case by case basis.

We can apply **inference rules** to each statement to prove partial correctness for an entire program.

The result is an **annotated program**, with one or more conditions before and after each program statement. Each statement, together with the preceding and following condition, forms a Hoare triple that justifies the correctness of each statement.

Auxiliary variables can be introduced if the conditions require additional variables that do not appear in the program.

11 Review

All rules of formal deduction:

1. (Ref) $A \vdash A$.
2. (+) If $\Sigma \vdash A$ then $\Sigma, \Sigma' \vdash A$.
3. ($\neg-$) If $\Sigma, \neg A \vdash B$ and $\Sigma, \neg A \vdash \neg B$ then $\Sigma \vdash A$.

4. $(\to -)$ If $\Sigma \vdash A \to B$ and $\Sigma \vdash A$ then $\Sigma \vdash B$.
5. $(\to +)$ If $\Sigma, A \vdash B$ then $\Sigma, A \to B$
6. $(\wedge -)$ If $\Sigma \vdash A \wedge B$ then $\Sigma \vdash A$ and $\Sigma \vdash B$
7. $(\wedge +)$ If $\Sigma \vdash A$ and $\Sigma \vdash B$ then $\Sigma \vdash A \wedge B$
8. $(\vee -)$ If $\Sigma, A \vdash C$ and $\Sigma, B \vdash C$ then $\Sigma, A \vee B \vdash C$
9. $(\vee +)$ If $\Sigma \vdash A$ then $\Sigma \vdash A \vee B$ and $\Sigma \vdash B \vee A$
10. $(\leftrightarrow -)$ If $\Sigma \vdash A \leftrightarrow B$ and $\Sigma \vdash A$ then $\Sigma \vdash B$
11. $(\leftrightarrow +)$ If $\Sigma, A \vdash B$ and $\Sigma, B \vdash A$ then $\Sigma \vdash A \leftrightarrow B$
12. $(\forall -)$ If $\Sigma \vdash \forall x A(x)$ then $\Sigma \vdash A(t)$
13. $(\forall +)$ If $\Sigma \vdash A(u)$ and u does not occur in Σ then $\Sigma \vdash \forall x A(x)$
14. $(\exists -)$ If $\Sigma, A(u) \vdash B$ and u does not occur in Σ or in B then $\Sigma, \exists x A(x) \vdash B$
15. $(\exists +)$ If $\Sigma \vdash A(t)$ then $\Sigma \vdash \exists x A(x)$ (only some t needs to be replaced with x in $A(x)$)
16. $(\approx -)$ If $\Sigma \vdash A(t_1)$ and $\Sigma \vdash t_1 \approx t_2$ then $\Sigma \vdash A(t_2)$
17. $(\approx +)$ $\emptyset \vdash u \approx u$