Logic and Computation

Frank Jin

Contents

Chapter 1

Propositional Logic

1.1 Syntax of Propositional Language

To begin, we introduce fully parenthesized expressions that can be parsed in an unique way. We construct $\mathcal{L}^p$, the formal language of propositional logic. The set of formulas in $\mathcal{L}^p$ is denoted by $Form(\mathcal{L}^p)$ and defines the formulation rules for expressions in $\mathcal{L}^p$. $\mathcal{L}^p$ contains 3 classes of symbols:

- Proposition symbols: $p, q, r, ...$
- Connective symbols: $\neg, \wedge, \vee, ...$
- Punctuation symbols: $(,)$

We denote expressions as finite strings of allowed symbols. Lastly, we denote the empty expression as ϵ, where $U\epsilon = \epsilon U = U$ for all expressions U. If $U = W_1VW_2$ where U, W_1, V, and W_2 are expressions, then V is a segment of U, and if $U \neq V$, then V is a proper segment of U. In this situation, we may denote W_1 as the initial segment or prefix when $VW_2 \neq \epsilon$ and W_2 as the terminal segment or suffix when $W_1V \neq \epsilon$.

For the next definition, we use $Atom(\mathcal{L}^p)$ to denote atomic formulas of $\mathcal{L}^p$.

Definition 1.1.1: $Form(\mathcal{L}^p)$

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

- BASE: Every element of $Atom(\mathcal{L}^p)$ is a formula of $Form(\mathcal{L}^p)$
- RECURSION: If $A, B \in Form(\mathcal{L}^p)$:
  - $(\neg A) \in Form(\mathcal{L}^p)$
  - $(A \wedge B) \in Form(\mathcal{L}^p)$
  - $(A \vee B) \in Form(\mathcal{L}^p)$
  - $(A \rightarrow B) \in Form(\mathcal{L}^p)$
  - $(A \leftrightarrow B) \in Form(\mathcal{L}^p)$
- RESTRICTION: No other expressions in $\mathcal{L}^p$ are in $Form(\mathcal{L}^p)$

Lemma 1.1.1

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

Proof. We use structural induction on $R(A)$: A has equal left and right parenthesis for all formulae $A \in Form(\mathcal{L}^p)$.

Base case: A is an atom; thus vacuously has equal left and right parentheses.

Inductive step: Let $l(A)$ be the number of left parentheses in A and $r(A)$ be the number of right parentheses in A. Assume $A = (\neg B)$ has property R. Then,

$$l((\neg B)) = 1 + l(B) = 1 + r(B) = r((\neg B))$$

For formulae B and C, assume B and C hold property R. Consider $\psi \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

$$l((B\psi C)) = 1 + l(B) + l(C) = 1 + r(B) + r(C) = r((B\psi C))$$

This concludes the composite inductive step and thus our proof by structural induction. □

Theorem 1.1.1: Unique Readability Theorem

Every formula of $\mathcal{L}^p$ is exactly one of six forms: $A, (\neg A), (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$.

Proof. Let $P(n)$ be the property: Every formula A with up to n connectives satisfies:

1. The first symbol of A is either (or a propositional symbol
2. A has equal left and right parentheses and non-empty proper prefixes have more left parentheses than right parentheses
3. A has a unique construction

We wish to prove $P(n)$ for all $\mathbb{N}$.

Base case: For $n = 0$, A is a propositional symbol: this case is trivial.

Inductive step:

Case 1 (negation) Let B be a formula satisfying P. By inspection, $(\neg B)$ satisfies (1). For property (2), we show by inspection using every non-empty subcase of $(\neg B)$: $(, (\neg$, and $(\neg B$. For property (3), $(\neg B)$ is unique because B is unique.

Case 2 Let $A = (B\psi C)$ where $\psi \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ and B, C hold P. We show properties (1) and (2) similarly to the negation case. To show property (3), we will show that if $(A = (B\psi C) = (B'\psi' C')$, then $B = B'$, $C = C'$, and $\psi = \psi'$. If $len(B) = len(B')$, then $B = B'$. If B' is a non-empty prefix of B, B' has the property of equal left and right parentheses by the inductive hypothesis. However, B' is a non-empty prefix of B, so B' must have more left parentheses than right. This case is a contradiction. Furthermore, this logic also shows that B cannot be a non-empty proper prefix of B'.

Our compositive inductive step is complete. □

Definition 1.1.2: Scope

In $(\neg A)$, A is called the scope of the negation. In $(A\psi B)$ for $\psi \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, A is called the left scope of ψ and B is the right scope of ψ.

1.2 Semantics of Propositional Language

Semantics is concerned with the meaning or truth value of formulae.

Definition 1.2.1: Truth Valuation

A truth valuation is a function t,

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

where 0 indicates false and 1 indicates true. We denote A^t as $t(A)$.

We can say that a truth valuation corresponds to a single row in a truth table.

Definition 1.2.2: Satisfies

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

We use Σ to define any set of formulas. The value of a truth valuation t under a set 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 1.2.3: Satisfiable

A set of formulas $\Sigma \subseteq Form(\mathcal{L}^p)$ is satisfiable iff there exists a truth valuation t such that $\Sigma^t = 1$. Otherwise, Σ is unsatisfiable.

A formula is a tautology if it is true under all possible truth valuations. The opposite of a tautology is a contradiction, where it is false under all truth valuations. If a formula is neither a tautology or a contradiction, then it is contingent.

Axiom 1.2.1: Law of the Excluded Middle

$p \vee \neg p$ is a tautology.

Axiom 1.2.2: Law of Contradiction

$\neg(p \wedge \neg p)$ is a tautology: equivalently, $p \wedge \neg p$ is a contradiction.

Definition 1.2.4: Tautological Consequence

Suppose $\Sigma \subseteq Form(\mathcal{L}^p)$ and $A \in Form(\mathcal{L}^p)$. A is a tautological consequence of Σ, denoted $\Sigma \models A$ iff for any truth valuation t, we have $\Sigma^t = 1$ implies $A^t = 1$.

Definition 1.2.5: Tautological Equivalence

For two formulas, we write $A \models\!\!\models B$ to denote $A \models B$ and $B \models A$. A and B are tautologically equivalent if $A \models\!\!\models B$ holds.

Theorem 1.2.1: De Morgan's Laws

$$\neg(p \vee q) \models\!\!\models \neg p \wedge \neg q$$
$$\neg(p \wedge q) \models\!\!\models \neg p \vee \neg q$$

Lemma 1.2.1

If $A \models\!\!\models A'$ and $B \models\!\!\models B'$, then

1. $\neg A \models\!\!\models \neg A'$
2. $A \wedge B \models\!\!\models A' \wedge B'$
3. $A \vee B \models\!\!\models A' \vee B'$
4. $A \to B \models\!\!\models A' \to B'$
5. $A \leftrightarrow B \models\!\!\models A' \leftrightarrow B'$

By structural induction, we can prove the following two theorems:

Theorem 1.2.2: Replaceability of Tautologically Equivalent Formulas

Let A be a formula which contains subformula B. Assume $B \models\!\!\models C$, and let A' be the formula obtained by simultaneously replacing in A some occurrences of B with C. Then, $A \models\!\!\models A'$.

Theorem 1.2.3: Duality

Suppose A is a formula composed only of atoms and connectives $\neg, \wedge$, and $\vee$ following formation rules of $Form(\mathcal{L}^p)$. Suppose $\Delta(A)$ results from simultaneously replacing in A all occurrences of $\wedge$ with $\vee$, and all atoms with their negation. Then, $\neg A \models\!\!\models \Delta(A)$.

1.3 Propositional Calculus

In formula manipulations, it is common to remove instances of $\rightarrow$ and $\leftrightarrow$ with logical equivalences. These equivalences are as follows:

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

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

Axiom 1.3.1: Essential Laws for Propositional Calculus

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

Using these laws, one can derive further laws such as absorption:

$$A \vee (A \wedge B) \models\!\dashv A \models\!\dashv A \wedge (A \vee B)$$

and

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

Definition 1.3.1: Literal

A formula is literal if in the form p or $\neg p$. p and $\neg p$ are complementary literals.

In propositional calculus, there are two *normal* forms used for symbolic manipulation, identification, and comparison. These forms are the disjunctive and conjunctive normal forms.

Definition 1.3.2: Clauses

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

Definition 1.3.3: CNF and DNF

We call disjunctive normal form a disjunction with conjunctive clauses as its disjuncts. We call conjunctive normal form a conjunction with disjunctive clauses as its conjuncts.

Algorithm 1.3.1: Algorithm for CNF

1. Eliminate equivalences and implications
2. Use De Morgan's and Double Negation laws so that each $\neg$ only has an atom in scope
3. RECURSION: on $CNF(A)$:
   (a) If A is a literal, return A
   (b) If A is $B \wedge C$, return $CNF(B) \wedge CNF(C)$
   (c) If A is $B \vee C$, call $CNF(B) = B_1 \wedge B_2 \wedge ... \wedge B_n$ and $CNF(C) = C_1 \wedge C_2 \wedge ... \wedge C_m$. Return $\wedge_{i=1,...,n,j=1,...,m}(B_i \vee C_j)$

Theorem 1.3.1

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

When A is a contradiction, for any atom p in A, we can see that $p \wedge \neg p$ is equivalent to A. If A is not a contradiction, we take the following example: Suppose A has 3 atoms p, q, and r and is true iff p, q, r are assigned $1, 1, 0$ or $1, 0, 1$ or $0, 0, 1$. Then, the following DNF satisfies equivalence:

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

Theorem 1.3.2

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

Proof. We see this by Theorem 1.3.2 and the Duality Theorem. □

For any n-ary connective for $n > 2$, we denote the formula formed by connective f as $f(A_1, ..., A_n)$. Two connectives of $n \geq 1$ are the same if their truth tables are the same. For any n, there are $2^{(2^n)}$ distinctive connectives.

Definition 1.3.4: Adequate

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

$\{\neg, \rightarrow, \leftrightarrow, \wedge, \vee\}$ is adequate. A set S is adequate if any n-ary connective can be defined in terms of S.

Theorem 1.3.3

The set $S_0 = \{\neg, \wedge, \vee\}$ is adequate.

Proof. Let f be an arbitrary n-ary connective. To find formula A_{S_0} only using connectives of S_0, where $f(p, ..., p_n) \models\!\models A_{s_0}$, construct f's truth table. The proof follows directly from the algorithm for CNF and existence of DNF. □

Theorem 1.3.4

$\{\neg, \wedge\}$, $\{\neg, \vee\}$, and $\{\neg, \rightarrow\}$ are adequate.

Proof. We will prove that $\{\neg, \wedge\}$ is adequate. Since $\{\neg, \wedge\} \cap \{\neg, \wedge, \vee\} = \{\neg, \wedge\}$, we just have to show that $\vee$ can be defined in terms of $\{\neg, \wedge\}$. For any A and B, note that $A \vee B \models\!\models \neg(\neg A \wedge \neg B)$ by De Morgan's law and Double Negation, thus by the Replaceability Theorem, we are done, where $f \models\!\models A_{S_0} \models\!\models \{\neg, \wedge\}$. □

Similarly, we can show that a set is not adequate. For example, $S = \{\wedge\}$ is not adequate: for any atom p with $p^t = 0$, with only $\wedge$ we cannot have any resultant formula $A_\wedge(p) \models\!\models \neg p$ by the properties of $\wedge$'s truth table.

1.4 Boolean Algebra

Definition 1.4.1: Boolean Algebra

A Boolean algebra is a set B with binary operators $+$ and $\cdot$, and unary operator $-$. B contains 0 and 1, and is closed under its operators.

The laws of Boolean algebras are as follows:

Axiom 1.4.1: Laws of Boolean Algebras

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

The set $Form(\mathcal{L}^p)$ with $\neg, \wedge, \vee$ as operators and $0, 1$ where $=$ is $\models\!\!\models$, is a Boolean algebra. The set of subsets of a universal set $\mathcal{U}$, with operators $\cap, \cup, ^c$ and $\emptyset$ is a Boolean algebra.

The circuit design section is omitted from my notes.

1.5 Formal Deduction for Propositional Logic

Formal deduction is a mechanical/syntactical method of checking proof correctness. We write $\Sigma \vdash A$ to denote that A is formally deducible from Σ. For two sets Σ and Σ', we use the notation Σ, Σ' to show $\Sigma \cup \Sigma'$. The 11 rules of formal deduction are as follows:

Axiom 1.5.1: 11 Rules of Formal Deduction

For any Σ and any A, $\Sigma \vdash A$ implies that $\Sigma \vdash A$ is a theorem.

$$
\begin{aligned}
A \vdash A & \qquad (Ref) \\
\text{If } \Sigma \vdash A, \text{ then } \Sigma, \Sigma' \vdash A & \qquad (+) \\
\text{If } \Sigma, \neg A \vdash B \text{ and } \Sigma, \neg A \vdash \neg B, \text{ then } \Sigma \vdash A & \qquad (\neg -) \\
\text{If } \Sigma \vdash A \to B \text{ and } \Sigma \vdash A, \text{ then } \Sigma \vdash B & \qquad (\to -) \\
\text{If } \Sigma, A \vdash B, \text{ then } \Sigma \vdash A \to B & \qquad (\to +) \\
\text{If } \Sigma \vdash A \wedge B, \text{ then } \Sigma \vdash A \text{ and } \Sigma \vdash B & \qquad (\wedge -) \\
\text{If } \Sigma \vdash A \text{ and } \Sigma \vdash B, \text{ then } \Sigma \vdash A \wedge B & \qquad (\wedge +) \\
\text{If } \Sigma, A \vdash C \text{ and } \Sigma, B \vdash C, \text{ then } \Sigma, A \vee B \vdash C & \qquad (\vee -) \\
\text{If } \Sigma \vdash A, \text{ then } \Sigma \vdash A \vee B \text{ and } \Sigma \vdash B \vee A & \qquad (\vee +) \\
\text{If } \Sigma \vdash A \leftrightarrow B \text{ and } \Sigma \vdash A, \text{ then } \Sigma \vdash B & \qquad (\leftrightarrow -) \\
\text{If } \Sigma, A \vdash B \text{ and } \Sigma, B \vdash A, \text{ then } \Sigma \vdash A \leftrightarrow B & \qquad (\leftrightarrow +)
\end{aligned}
$$

We can prove additional theorems, known as schemes of formal deducibility, with these 11 rules.

Theorem 1.5.1: Membership

($\in$) If $A \in \Sigma$, then $\Sigma \vdash A$.

Proof. Let $A \in \Sigma$ and $\Sigma' = \Sigma - \{A\}$, so $\Sigma = A, \Sigma'$. Then,

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

□

Definition 1.5.1: Formal Deducibility

A formula is formally deducible from Σ iff $\Sigma \vdash A$ is generated by finite applications of the rules of formal deduction. The sequence of rules generating $\Sigma \vdash A$ is called a formal proof.

Theorem 1.5.2: Finiteness of Premise Set

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

Proof. The full proof by induction will be omitted. The base case $A \vdash A$ holds by reflexivity. We have 10 cases in our inductive step: one of them being $(\to -)$. Here, assume for $\Sigma_1, \Sigma_2 \subseteq \Sigma$, that $\Sigma_1 \vdash A \to B$ and $\Sigma_2 \vdash A$. By $(+)$, $\Sigma_1, \Sigma_2 \vdash A \to B$ and $\Sigma_1, \Sigma_2 \vdash A$. Then, by $(\to -)$, $\Sigma_1, \Sigma_2 \vdash B$. □

Theorem 1.5.3: Transitivity of Deducibility

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

Proof. By formal deduction:

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

□

Theorem 1.5.4: Double Negation

$\neg\neg A \vdash A$

Proof. By formal deduction:

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

□

Theorem 1.5.5: Reductio ad absurdum

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

Proof. By formal deduction:

1.	$\Sigma, A \vdash B$	given
2.	$\Sigma, \neg\neg A \vdash \Sigma$	$(\in)$
3.	$\neg\neg A \vdash A$	D. Neg.
4.	$\Sigma, \neg\neg A \vdash A$	$(+, 3)$
5.	$\Sigma, \neg\neg A \vdash B$	$(\text{Tr.}, 2, 4, 1)$
6.	$\Sigma, \neg\neg A \vdash \neg B$	analogous to proof of 5
7.	$\Sigma \vdash \neg A$	$(\neg -, 5, 6)$

□

Definition 1.5.2: Syntactic Equivalence

For two formulas A and B, we write $A \dashv\vdash B$ to mean $A \vdash B$ and $B \vdash A$.

Lemma 1.5.1

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. $A \to B \dashv\vdash A' \to B'$
5. $A \leftrightarrow B \dashv\vdash A' \leftrightarrow B'$

Theorem 1.5.6: Replaceability of Syntactically Equivalent Formulas

(Repl.) Let $B \dashv\vdash C$. For any A, let A' be constructed from A by replacing some occurrences of B by C. Then $A \dashv\vdash A'$.

Theorem 1.5.7

$$A_1, A_2, ..., A_n \vdash A \text{ iff } \emptyset \vdash A_1 \wedge ... \wedge A_n \to A$$

$$A_1, A_2, ..., A_n \vdash A \text{ iff } \emptyset \vdash A_1 \to (...(A_n \to A)...)$$

Definition 1.5.3: Formally Provable

A is formally provable iff $\emptyset \vdash A$

A system $\vdash_*$ of formal deducibility defined by a number of formal deduction rules should meet two requirements:

- Soundness: it does not prove incorrect statements (If $\Sigma \vdash_* A$, then $\Sigma \models A$)
- Completeness: it proves every correct statement (If $\Sigma \models A$, then $\Sigma \vdash_* A$)

Theorem 1.5.8: Soundness

If $\Sigma \vdash A$, then $\Sigma \models A$.

Proof. This is a proof by structural induction on the 11 rules of formal deduction. By (Ref), we have our base case: If $A \vdash A$, then $A \models A$. The remaining subcases for the inductive step are similar, so only $(\neg -)$ will be shown.

Suppose $\Sigma, \neg A \models B$ and $\Sigma, \neg A \models \neg B$. Assume that $\Sigma \not\models B$ for the sake of contradiction: this means there exists a truth valuation t such that $\Sigma^t = 1$ and $A^t = 0$. Then, by the definition of negation, $(\neg A)^t = 1$. Since $\Sigma, \neg A \models B$ and $\Sigma, \neg A \models \neg B$, then $B^t = 1$ and $(\neg B)^t = 1$ as well. This is a contradiction, so $\Sigma \models A$ must be the case. □

Theorem 1.5.9: Completeness

If $\Sigma \models A$, then $\Sigma \vdash A$.

Proof. This proof has 3 steps:

1. If $A_1, ..., A_n \models A$, then $\emptyset \models (A_1 \to (A_2 \to ...(A_n \to A)...))$

   Assume, for contradiction, there exists truth valuation t so $(A_1 \to ...(A_n \to A)...)^t = 0$. Thus, by the truth table of $\to$, $A_1^t = ... = A_n^t = 1$ and $A^t = 0$. This contradicts our hypothesis.

2. If $\emptyset \models A$, then $\emptyset \vdash A$

   Let the n atoms in A be $p_1, ..., p_n$ with truth valuation t. Define

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

   Aside, we prove the following lemma:

   $$\text{If } A^t = 1, \text{ then } p_1', p_2', ..., p_n' \vdash A$$

   $$\text{If } A^t = 0, \text{ then } p_1', p_2', ..., p_n' \vdash \neg A$$

   By structural induction on $A \in Form(\mathcal{L}^p)$, our base case for atom $A = p$ where $p^t = 1$ is such that $p' = p$. We have $p \vdash A$ by (Ref) in this case. In the case that $p^t = 0$, we have $p' = \neg p$. We also have by (Ref) that $\neg p \vdash A$ as required.

For the inductive step, we will show one of 5 similar cases. This case is such that $A = B \to C$ where B and C satisfy the inductive hypothesis. If $(B \to C)^t = 0$, then $B^t = 1$ and $C^t = 0$. This implies that $b'_1, ..., b'_k \vdash B$ and $c'_1, ..., c'_k \vdash \neg C$. By $(+)$, we have $p'_1, ..., p'_n \vdash B$ and $p'_1, ..., p'_n \vdash \neg C$. We must now show that $B \wedge \neg C \vdash \neg(B \to C)$:

1.	$B \wedge \neg C, B \to C \vdash B \wedge \neg C$	$(\in)$
2.	$B \wedge \neg C, B \to C \vdash B$	$(\wedge -, 1)$
3.	$B \wedge \neg C, B \to C \vdash B \to C$	$(\in)$
4.	$B \wedge \neg C, B \to C \vdash C$	$(\to -, 2, 3)$
5.	$B \wedge \neg C, B \to C \vdash \neg C$	$(\wedge -, 1)$
6.	$B \wedge \neg C \vdash \neg(B \to C)$	$(4, 5, \neg +)$

We follow a similar structure for case $(B \to C)^t = 1$ and the remaining 4 cases.

With this lemma, every choice of $p'_1, ..., p'_n$ guarantees a formal proof for A. Then, for all $1 \leq i \leq n$, we use $(\vee -)$ to combine into a general proof with premise $(p_i \vee \neg p_i)$. The law of excluded middle gives $\emptyset \vdash p_i \vee \neg p_i$ and (Tr.) gives $\emptyset \vdash A$.

3. If $\emptyset \vdash (A_1 \to ...(A_n \to A)...)$, then $A_1, ..., A_n \to A$

   $\emptyset \vdash (A_1 \to ...(A_n \to A)...)$ implies $A_1, ..., A_n \vdash (A_1 \to ...(A_n \to A)...)$. After n applications of $(\in)$ and $(\to -)$, we have the desired result.

□

Definition 1.5.4: Consistent

Σ is consistent iff there is no formula F such that $\Sigma \vdash F$ and $\Sigma \vdash \neg F$.

Lemma 1.5.2

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

Proof. $(\Rightarrow)$ For the sake of contradiction, assume that Σ is inconsistent, so there exists F where $\Sigma \vdash F$ and $\Sigma \vdash \neg F$. By soundness, $\Sigma \models F$ and $\Sigma \models \neg F$ where there exists a truth valuation t so that $\Sigma^t = 1$ and $F^t = (\neg F)^t = 1$. This is impossible, so Σ must be consistent.

$(\Leftarrow)$ Assume Σ is not satisfiable for the sake of contradiction. That is, $\Sigma^t = 0$ for all t. Vacuously, all formulas F give $\Sigma \models F$ and $\Sigma \models \neg F$, implying inconsistency. However, this contradicts our hypothesis, so Σ must be satisfiable. □

1.6 Resolution for Propositional Logic

Resolution is a theorem proving method of formal derivation. Formulas provable by resolution must be disjunctions of literals. To show that

$$A_1, ..., A_n \models C$$

we show that $\{A_1, ..., A_n, \neg C\}$ is not satisfiable.

Definition 1.6.1: Resolution

Resolution is a 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.

We call $C \vee p$ and $D \vee \neg p$ parent clauses. We resolve the two parent clauses over p, where $C \vee D$ is called the resolvent. The resolvent of p and $\neg p$ is the empty clause where $p, \neg p \vdash \{\}$. $\{\}$ is considered unsatisfiable.

Algorithm 1.6.1: Resolution Procedure

For input $\mathcal{S} = \{D_1, D_2, ..., D_n\}$ of disjunctive clauses:

- Repeat until output $\{\}$:
  - Choose 2 clauses, one with p and one with $\neg p$
  - Resolve and call the resolvent D
  - If $D = \{\}$, end the procedure
  - Otherwise, add D to $\mathcal{S}$ and repeat

Theorem 1.6.1

The resolvent is tautologically implied by its parent clause, making resolution a sound rule of formal deduction.

Proof. Let p be a propositional variable and A, B be clauses. Assume that $p \vee A, \neg p \vee B \vdash_r A \vee B$. We will show that $p \vee A, \neg p \vee B \models A \vee B$.

Case 1: at least 1 of A, B are non-empty. If truth valuation t gives $(p \vee A)^t = (\neg p \vee B)^t = 1$, then $p^t = 0$ implies $A^t = 1$ and $p^t = 0$ implies $B^t = 1$. So, $(A \vee B)^t = 1$ in either case.

Case 2: A, B are both empty. By definition, the resolvent of p and $\neg p$ is $\{\}$, denoting $p \wedge \neg p$. In this case, $p, \neg p \not\models \{\}$ because the premises are contradictory. □

We will cover two strategies for resolution:

- Set-of-support strategy
- David-Putnam procedure

Algorithm 1.6.2: Set-of-Support Strategy

- Partition all starting clauses into 2 sets: the set of support and the auxiliary set
- Repeat until output {}:
  - Each resolution takes at least one clause from the set of support
  - Upon resolution, add the resolvent to the set of support and repeat

The auxiliary set must not be contradictory: often. Usually, the premises are used as the auxiliary set, and the negation of the conclusion is used as the set of support.

Theorem 1.6.2

Resolution by set-of-support is complete.

Axiom 1.6.1: Pigeonhole Principle

($\mathcal{P}_n$): One cannot put $n+1$ objects into n slots with distinct objects going into distinct slots.

We can show the pigeonhole principle in terms of resolution. Let propositional variable p_{ij} for $1 \leq i \leq n+1$ and $1 \leq j \leq n$ be defined as true iff the ith pigeon goes into the jth slot. We construct clauses for each pigeon i. Each pigeon must go into some slot k where $1 \leq k \leq n$:

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

for $1 \leq i \leq n+1$. Distinct pigeons $i \neq j$ for $1 \leq i, j \leq n+1$ cannot go into the same slot k.

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

Any truth valuation t that satisfies the conjunction of all of the above disjunctive clauses would map $n+1$ pigeons into n slots. However, this contradicts the pigeonhole principle, so this set of clauses is unsatisfiable.

The David-Putnam procedure requires pre-processing of its input clause. Firstly, we convert clauses into sets: for example, $p \vee q \vee \neg r$ corresponds to the set $\{p, q, \neg r\}$. We now define the David-Putnam procedure as follows:

Algorithm 1.6.3: David-Putnam Procedure

For input $\mathcal{S}$ of disjunctive clauses in preprocessed format with propositional variables $p_1, ..., p_n$ for $n \geq 1$:

- Let $\mathcal{S}_1 = \mathcal{S}$
- Loop from $i = 1$ until $i = n + 1$:
  - Discard members of $\mathcal{S}_i$ in which a literal and its complement appear, to obtain $\mathcal{S}_i'$
  - Let $\mathcal{T}_i$ be the set of parent clauses in $\mathcal{S}_i'$ where p_i or $\neg p_i$ appear
  - Let $\mathcal{U}_i$ 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 $\mathcal{T}_i$
  - Let $\mathcal{S}_{i+1} = (\mathcal{S}_i' - \mathcal{T}_i) \cup \mathcal{U}_i$
- Return $\mathcal{S}_{n+1}$

If the output of DPP is the empty clause $\{\}$, then the argument is valid. If the output of DPP is the empty set $\emptyset$, then the argument is not valid. This is because no contradiction was found.

Theorem 1.6.3

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

Proof. ($\Leftarrow$): $\mathcal{S} \vdash_r \{\}$ by DPP implies $\mathcal{S}$ is not satisfiable.

We use induction on i to show that if C is a clause in $\mathcal{S}_i$, then there is a resolution derivation of C from the initial set $\mathcal{S}$. Since the empty clause is the output, that is, $\{\} \in S_{n+1}$, it follows that there is a resolution derivation from $\mathcal{S}$ to $\{\}$. Since $\{\}$ is not satisfiable and by the soundness of resolution, $\mathcal{S}$ is not satisfiable.

($\Rightarrow$): $\mathcal{S}$ is not satisfiable implies that $\mathcal{S} \vdash_r \{\}$ by DPP.

Suppose, for contradiction, that DPP on $\mathcal{S}$ produces $\emptyset$. If $\mathcal{S}_{n+1} = \emptyset$, then vacuously, $\mathcal{S}_{n+1}$ is satisfiable. We want to show that $\mathcal{S}_{i+1}$'s satisfiability implies $\mathcal{S}_i$'s satisfiability: that is, satisfiability propagates backwards. Assume that truth valuation t_{i+1} satisfies $\mathcal{S}_{i+1}$. It is implied that t_{i+1} also satisfies $\mathcal{U}_i$ and $(\mathcal{S}_i' - \mathcal{T}_i)$. Since $\mathcal{S}_i = (\mathcal{S}_i' - \mathcal{T}_i) \cup \mathcal{T}_i$, it suffices to show that $\mathcal{T}_i$ is satisfiable.

Since $\mathcal{S}_i$ has one additional variable p_i as opposed to $\mathcal{S}_{i+1}$, we extend a truth valuation coinciding with t_{i+1} such that $p_{i+1}, ..., p_n$ are evaluated identically to t_{i+1} and p_i is assigned appropriately. We claim that one of the two following truth valuations must satisfy $\mathcal{T}_i$:

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

Assume for the sake of contradiction that neither t_0 or t_1 satisfy $\mathcal{T}$. Since t_0 satisfies all formulae in $\mathcal{T}_i$ containing $\neg p_i$, it must falsify some clause $D \cup \{p_i\}$ in $\mathcal{T}_i$. As $D \cup \{p_i\}$ is not satisfied by t_0, then D is not satisfied by t_{i+1}. Since t_1 satisfies all formulae in $\mathcal{T}_i$ containing 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}. As t_{i+1} does not satisfy either D or E, it does not satisfy $D \cup E$: a contradiction, since $D \cup E \subseteq S_{i+1}$ and $(S_{i+1})^t = 1$.

Since our initial claim that DPP on $\mathcal{S}$ produces $\emptyset$ lead to a contradiction, it is the case $\mathcal{S} \vdash_r \{\}$ by DPP as required. □

In a DPP exercise, we use an underlining notation when clauses are to be eliminated. For example, if $\mathcal{T}_1 = \{p, q\}^1, \{\neg p, \neg q\}^2, \{\neg p, s, t\}^3, \{p, \neg s, t\}^4, \{p, s, \neg t\}^5, \{\neg p, \neg s, \neg t\}^6$:

$$\begin{aligned}\mathcal{U}_1 &= \underline{\{q, \neg q\}^{1,2}}, \{q, s, t\}^{1,3}, \{q, \neg s, \neg t\}^{1,6}, \{\neg q, \neg s, t\}^{2,4}, \{\neg q, s, \neg t\}^{2,5}, \underline{\{s, t, \neg s\}^{3,4}}, \underline{\{s, t, \neg t\}^{3,5}} \\ &= \underline{\{\neg s, t, \neg t\}^{4,6}}, \underline{\{s, \neg t, \neg s\}^{5,6}}\end{aligned}$$

Chapter 2

First-Order Logic

2.1 Syntax of First-Order Language

Unlike propositional logic, in first-order logic we are concerned with *domain* of arguments. Introducing domain allows us to make arguments that are specific and preventative of ambiguities found in propositional logic. For example, the statement *Joan is Paul's mother* can only be assessed to be true or false under certain contexts: there are many people named Joan and Paul.

Definition 2.1.1: Universe of Discourse

The universe of discourse, or domain, is the collection of persons, ideas, symbols, data structures, and so on, that affect the logical argument under consideration.

We call elements of the domain *individuals*. Domains are always non-empty. Assuming a domain is finite, where $D = \{\alpha_1, ..., \alpha_n\}$, we have

$$\forall x R(x) = 1 \text{ iff } R(\alpha_1) \wedge ... \wedge R(\alpha_n)$$

and

$$\exists x R(x) = 1 \text{ iff } R(\alpha_1) \vee ... \vee R(\alpha_n)$$

Thus, negating quantifiers, where

$$\neg \forall x P(x) \nvDash \exists x \neg P(x) \text{ and } \neg \exists x P(x) \nvDash \forall x \neg P(x)$$

can be viewed as generalizations of De Morgan's laws.

$\mathcal{L}$ is the language of first-order logic. It contains the following logical symbols:

- connectives
- free variable symbols
- bound variable symbols
- quantifiers
- punctuation symbols

It also contains non-logical symbols:

- individual symbols, or constants
- relation symbols
- function symbols

$\mathcal{L}$ is not a single language, but first-order languages are derived from some, but not necessarily all, of the logical and non-logical symbols of $\mathcal{L}$. The binary relation symbol called the equality symbol, denoted as $\approx$, which may or may not be contained in $\mathcal{L}$.

Definition 2.1.2: $Term(\mathcal{L})$

$Term(\mathcal{L})$ is the smallest class of expressions of $\mathcal{L}$ closed under the following rules:

- Every individual symbol is a term in $Term(\mathcal{L})$
- Every free variable is a term in $Term(\mathcal{L})$
- If $t_1, ..., t_n \in Term(\mathcal{L})$, and f is an n-ary function symbol, $f(t_1, ..., t_n) \in Term(\mathcal{L})$.

Definition 2.1.3: $Atom(\mathcal{L})$

An expression of $\mathcal{L}$ is in $Atom(\mathcal{L})$ iff it is one of the following:

- $F(t_1, ..., t_n)$ for $n \geq 1$ where F is an n-ary relation symbol and $t_1, ..., t_n \in Term(\mathcal{L})$
- $\approx (t_1, t_2)$ where $t_1, t_2 \in Term(\mathcal{L})$

Definition 2.1.4: $Form(\mathcal{L})$

The set of formulas in $\mathcal{L}$, $Form(\mathcal{L})$, is the smallest class of expressions of $\mathcal{L}$ closed under the following rules:

- Every atom in $Atom(\mathcal{L})$ is a formula in $Form(\mathcal{L})$
- If A is in $Form(\mathcal{L})$, then $(\neg A)$ is too
- If A, B are in $Form(\mathcal{L})$, then $(A \wedge B), (A \vee B), (A \to B), (A \leftrightarrow B)$ are too
- If $A(u)$ is in $Form(\mathcal{L})$ where u is a free variable and x is not occurring in $A(u)$, then $\forall x A(x)$ and $\exists x A(x)$ are too

For example, the expression $< (u, v) \to < (+(u, u), +(v, v))$ is a formula in first-order language of number theory. However, it is more commonly written as $u < v \to u + u < v + v$.

Theorem 2.1.1

Any term in $Term(\mathcal{L})$ is exactly one of three forms: an individual symbol, a free variable symbol, or $f(t_1, ..., t_n)$ where $n \geq 1$, f is an n-ary symbol, and t_i is a term for $1 \leq i \leq n$. In each case, the term is of that form in exactly one way.

Theorem 2.1.2

Any formula in $Form(\mathcal{L})$ is exactly one of eight forms: an atom (a single relation symbol applied to terms), $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, $(A \leftrightarrow B)$, $\forall x A(x)$, or $\exists x A(x)$. In each case, the formula is of that form in exactly one way.

Definition 2.1.5: $Sent(\mathcal{L})$

A sentence, or closed formula, of $Sent(\mathcal{L})$ is a formula in $Form(\mathcal{L})$ in which no free variables occur.

We can attribute $\mathcal{L}$ to different theories X with notation $\mathcal{L}(X)$. For example, if $X = \mathcal{N}$, we have $\mathcal{L}(\mathcal{N})$ denoting the first-order language of number theory. Here, we have a binary relation symbol $<$, individual symbol 0, the unary function for successor, and binary functions $+$ and $\times$.

2.2 Semantics of First-Order Language

Previously, we studied truth valuations of $Form(\mathcal{L}^p)$ which were purely based on values of propositional symbols and the truth tables of their connectives. However, valuations in $\mathcal{L}$ are more complex: they consist interpretations of non-logical symbols and assignments to free variables. Consider the following set definitions for relations and functions:

- An n-ary relation R on set D can be thought of as a subset R of D^n:

$$R = \{(a_1, ..., a_n) : a_i \in D \text{ and } R(a_1, ..., a_n) = 1\}$$

- An m-ary function $f : D^m \to D$ can be expressed by the $(m+1)$-ary relation:

$$R_f = \{(x_1, ..., x_m, x_{m+1}) : f(x_1, ..., x_m) = x_{m+1}\}$$

Definition 2.2.1: Valuation

A valuation for the first-order language $\mathcal{L}$ consists of:

- A non-empty domain D
- A function v with the properties:
  - For each individual symbol a and free variable symbol u, $a^v, u^v \in D$
  - 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, $f^v : D^m \to D$

Definition 2.2.2: Valuation of a Term

The value of a term t under valuation v over domain D, denoted by t^v, is defined recursively as:

- 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 $u^v \in D$
- If $t = f(t_1, ..., t_n)$ for $m \geq 1$ where f is a m-ary function symbol, and $t_i \in Term(\mathcal{L})$ for $1 \leq i \leq m$, then

$$f(t_1, ..., t_n)^v = f^v(t_1^v, ..., v_n^v)$$

Theorem 2.2.1

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

To determine the truth value for formulae of the form $\forall x A(x)$ and $\exists x A(x)$, we must check whether $A(u)$ hold for all, or some, values u in the domain. For any valuation v, free variable u, and individual $d \in D$, we write

$$v(u/d)$$

to denote a valuation such that $u^{v(u/d)} = d$ and is identical to v otherwise. For each free variable w,

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

Definition 2.2.3: Valuation of Quantified Formula

Let $\forall x A(x)$ and $\exists x A(x)$, and u be a free variable not occurring in $A(x)$. The values of $\forall x A(x)$ and $\exists x A(x)$ under valuation v with domain D are given by:

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

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

Definition 2.2.4: Value of a Formula

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

- $R(t_1, ..., t_n)^v = 1$, $n \geq 1$, iff $(t_1^v, ..., t_n^v) \in R^v \subseteq D^n$
- $(\neg A)^v = 1$ iff $A^v = 0$
- $(B \wedge C)^v = 1$ iff $B^v = 1$ and $C^v = 1$
- $(B \vee C)^v = 1$ iff either $B^v = 1$ or $C^v = 1$
- $(B \to C)^v = 1$ iff either $B^v = 0$ or $C^v = 1$
- $(B \leftrightarrow C)^v = 1$ iff $B^v = C^v$

Theorem 2.2.2

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

Definition 2.2.5

A formula $A \in Form(\mathcal{L})$ is:

- satisfiable if there exists valuation v such that $A^v = 1$
- universally valid if for all valuations v, we have $A^v = 1$
- unsatisfiable if for all valuations v, we have $A^v = 0$

Let Σ be a set of formulae in $Form(\mathcal{L})$ and v be a valuation over D. We define

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

Definition 2.2.6

A set $\Sigma \subseteq Form(\mathcal{L})$ is satisfiable iff there exists valuation v such that $\Sigma^v = 1$.

Unlike in $Form(\mathcal{L}^p)$, it becomes very difficult very quickly to prove universal validity of formulae in $Form(\mathcal{L})$ especially when considering infinite domains.

Theorem 2.2.3

(Church, 1936). There is no algorithm for deciding the (universal) validity or satisfiability of formulae in first-order logic.

2.3 Argument Validity in First-Order Logic

In the same sense as in propositional logic with tautological consequence, we use the notation $\models$ for logical consequence. We also use the following equivalent notations:

- $(\forall x A(x))^v = 1$ iff $A(d)^v = 1$ for all $d \in D$
- $(\exists x A(x))^v = 1$ iff $A(d)^v = 1$ for some $d \in D$

For proofs in form $\Sigma \models A$, we suppose that there exists a valuation v such that $\Sigma^v = 1$ and $A^v = 0$. For proofs in form $\Sigma \not\models A$, a single counter-example (a valuation v over domain D) such that $\Sigma \not\models A$ holds suffices. For any formula $A \in Form(\mathcal{L})$, one has $\emptyset \models A$ iff A is universally valid.

Theorem 2.3.1

$\forall x F(x) \vee \forall x G(x) \to \forall x(F(x) \vee G(x))$ is universally valid.

Proof. Assume that $\emptyset \not\models \forall x F(x) \vee \forall x G(x) \to \forall x(F(x) \vee G(x))$. This implies that there exists a valuation v such that $(\forall x F(x) \vee \forall x G(x) \to \forall x(F(x) \vee G(x)))^v = 0$. By the truth table of $\to$, we must have $(\forall x F(x) \vee \forall x G(x))^v = 1$ and $(\forall x(F(x) \vee G(x)))^v = 0$. Negating $(\forall x(F(x) \vee G(x)))^v = 0$ gives $(\exists x(\neg F(x) \wedge \neg G(x)))^v = 1$. This implies that there exists an individual $d \in D$ such that $(\neg F(d) \wedge \neg G(d))^v = 1$, or $((\neg F(d))^v = 1$ and $(\neg G(d))^v = 1$, so $(F(d))^v = (G(d))^v = 0$. This implies that $(\forall x F(x) \vee \forall x G(x))^v = 0$: contradiction! □

Lemma 2.3.1

Let $C(u) \models\!\!\models C'(u)$. Then,

- $\forall x C(x) \models\!\!\models \forall x C'(x)$
- $\exists x C(x) \models\!\!\models \exists x C'(x)$

For any $A \models\!\!\models A'$ and $B \models\!\!\models B'$, we have the identical logical equivalences as the tautological equivalences in terms of $\wedge, \vee, \to, \leftarrow$, and $\neg$.

Theorem 2.3.2: Replacability of Equivalent Formulas in First-Order Logic

Let $A \in Form(\mathcal{L})$ have subformula $B \in Form(\mathcal{L})$. Suppose that $B \models\!\!\models C$ and let A' be obtained by replacing some occurences of B by C. Then, $A \models\!\!\models A'$.

Theorem 2.3.3: Duality in First-Order Logic

Suppose $A \in Form(\mathcal{L})$ is composed of only atoms in $Atom(\mathcal{L})$, connectives $\neg, \wedge, \vee$, quantifiers $\forall, \exists$, and the formation rules concerned. Suppose $\Delta(A)$ results from exchanging $\vee$ and $\wedge$, $\forall$ and $\exists$, and each atom for its negation. Then, $\neg A \models\!\!\models \Delta(A)$.

2.4 Formal Deduction for First-Order Logic

The goal for formal deduction is to define a calculus for reasoning. The 11 rules of formal deduction for propositional logic are included in the formal deduction for first-order logic, but 6 additional rules are introduced.

Axiom 2.4.1: 6 Additional Rules for Formal Deduction of First-Order Logic

$$\begin{aligned}
\text{If } \Sigma \vdash \forall x A(x), \text{ then } \Sigma \vdash A(t) \text{ for any term } t & \quad (\forall -) \\
\text{If } \Sigma \vdash A(u) \text{ and } u \notin \Sigma, \text{ then } \Sigma \vdash \forall x A(x) & \quad (\forall +) \\
\text{If } \Sigma, A(u) \vdash B \text{ and } u \notin \Sigma, B \text{ then } \Sigma, \exists x A(x) \vdash B & \quad (\exists -) \\
\text{If } \Sigma \vdash A(t), \text{ then } \Sigma \vdash \exists x A(x) & \quad (\exists +) \\
\text{If } \Sigma \vdash A(t_1) \text{ and } \Sigma \vdash t_1 \approx t_2, \text{ then } \Sigma \vdash A(t_2) & \quad (\approx -) \\
\emptyset \vdash u \approx u & \quad (\approx +)
\end{aligned}$$

Lemma 2.4.1

If $C(u) \vdash\!\dashv C'(u)$, then

1. $\forall x C(x) \vdash\!\dashv \forall x C'(x)$
2. $\exists x C(x) \vdash\!\dashv \exists x C'(x)$

In formal deduction of first-order logic, replacability of equivalent formulas and duality also hold.

Theorem 2.4.1: Soundness and Completeness

Let $\Sigma \subseteq Form(\mathcal{L})$ and $A \in Form(\mathcal{L})$. Then $\Sigma \models A$ iff $\Sigma \vdash A$.

Theorem 2.4.2

$\forall x \forall y P(x, y) \vdash \forall y \forall x P(x, y)$

Proof. By formal deduction:

$$\begin{aligned}
&1. \quad \forall x \forall y P(x, y) \vdash \forall x \forall y P(x, y) && \text{(Ref)} \\
&2. \quad \forall x \forall y P(x, y) \vdash \forall y P(u, y) && (1, \forall -) \\
&3. \quad \forall x \forall y P(x, y) \vdash P(u, v) && (2, \forall -) \\
&4. \quad \forall x \forall y P(x, y) \vdash \forall x P(x, v) && (3, \forall +) \\
&5. \quad \forall x \forall y P(x, y) \vdash \forall y \forall x P(x, y) && (4, \forall +)
\end{aligned}$$

□

2.5 Resolution for First-Order Logic

Definition 2.5.1: Prenex Normal Form

A formula is in prenex normal form if it is of the form

$$Q_1x_1Q_2x_2...Q_nx_nB$$

where $n \geq 1$, Q_i is $\forall$ or $\exists$ for $1 \leq i \leq n$, and the expression B is quantifier free.

We call $Q_1x_1Q_2x_2...Q_nx_n$ the prefix and B the matrix. A formula with no quantifiers is called a trivial case of prenex normal form. Any formula in $Form(\mathcal{L})$ can be converted into PNF: one may accomplish this using logical equivalences.

Theorem 2.5.1: Replacability of Bound Variable Symbols

Let $A \in Form(\mathcal{L})$. Suppose A' results from A by replacing some occurrences of $QxB(x)$ with $QyB(y)$ where $Q \in \{\forall, \exists\}$. Then, $A \models\!\models A'$ and $A \vdash\!\dashv A'$.

For example,

$$\begin{aligned}\forall x(\exists yR(x,y) \wedge \forall y\neg S(x,y) \rightarrow \neg\exists y\neg Q(x,y)) &\models\!\models \forall x(\neg(\exists yR(x,y) \wedge \forall y\neg S(x,y)) \vee \neg\exists y\neg Q(x,y)) \\ &\models\!\models \forall x(\forall y\neg R(x,y) \vee \exists yS(x,y) \vee \forall yQ(x,y)) \\ &\models\!\models \forall x(\forall y_1\neg R(x,y_1) \vee \exists y_2S(x,y_2) \vee \forall y_3Q(x,y_3)) \\ &\models\!\models \forall x\forall y_1\exists y_2\forall y_3(\neg R(x,y_1) \vee S(x,y_2) \vee Q(x,y_3))\end{aligned}$$

Definition 2.5.2: $\exists$-free Prenex Normal Form

$A \in Sent(\mathcal{L})$ is said to be in $\exists$-free prenex normal form if it is in prenex normal form and does not contain existential quantifier symbols.

Let $\forall x_1...\forall x_n\exists yA$ be a sentence with $n \geq 0$. Note that for each n-tuple generated by the domain, $\exists yA$ generates at least one individual. We can express this as a function $f(x_1, ..., x_n)$. f is called a *Skolem* function. Note that the Skolem equivalent of a sentence is not logically equivalent to the original sentence unless each n-tuple generates exactly one individual.

Algorithm 2.5.1: Algorithm for ∃-free Prenex Normal Form

- Transform the input sentence $A_0 \in Sent(\mathcal{L})$ into logically equivalent PNF A_1. Let $i = 1$.
- Repeat until all existential quantifier are removed:
  - We have $A_i = \forall x_1 ... \forall x_n \exists y A$
  - If $n = 0$, then $A_{i+1} = A'$, where A' is obtained from A by replacing all occurrences of A with an individual symbol c not occuring in A_i
  - If $n > 0$, $A_{i+1} = \forall x_1 ... \forall x_n A'$ where A' is obtained from A by replacing occurrences of y by $f(x_1, ..., x_n)$ where f is a new function symbol
  - Increase i by 1

Theorem 2.5.2

Given $F \in Sent(\mathcal{L})$, there is an effective procedure for finding an ∃-free prenex normal form formula F' such that F is satisfiable iff F' is satisfiable.

After existential quantifiers have been eliminated through Skolem functions in ∃-free prenex normal form, we can drop universal quantifiers such that all variables are implicitly considered to be universally quantified.

Theorem 2.5.3

Given a sentence F in ∃-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.

For sentence F, we put the matrix of F into conjunctive normal form, where the set C_F is the set of disjunctive clauses from the conjuncts.

Theorem 2.5.4

Let Σ be a set of sentences, 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.

This theorem is intuitive in relation to resolution of propositional logic: the set containing the union of clauses obtained from each premise F in Σ and the set of clauses generated by the negation of the conclusion A is not satisfiable iff the argument in first-order logic is valid.

Definition 2.5.3: Instantiation

An instantiation is an assignment to a variable x_i of a quasi-term t'_i (individual symbol, variable symbol, or function symbol applied to individual symbols or variable symbols). We write $x_i := t'_i$.

Definition 2.5.4: Unification

Two formulas in first-order logic are said to unify if there are instantiations that make the formulas in question identical. The act of unifying is called unification. The instantiation that unifies the formulas in question is called a unifier.

With the definitions provided above, one can now effectively apply resolution in first-order logic in the same manner as in propositional logic.

Theorem 2.5.5

A set $\mathcal{S}$ of clauses in first-order logic is not satisfiable iff there is a resolution derivation of the empty clause, $\{\}$ from S. Resolution is sound and complete.

Theorem 2.5.6

A relation $R \subseteq D \times D$ is reflexive if it is transitive and symmetric, assuming that every individual of D is related, via R, to at least one other individual in D.

Proof. By resolution, we define $R(x, y)$ to mean that x is related to y. Since D is the domain, we have $\forall x \exists y R(x, y)$. We want to prove that if $\forall x \forall y \forall z (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$ and $\forall x \forall y (R(x, y) \rightarrow R(y, x))$, then $\forall x R(x, x)$. For the sake of resolution, we negate the conclusion to obtain $\exists x \neg R(x, x)$. In prenex normal form, we have:

$$\forall x R(x, f(x)) \tag{1}$$
$$\forall x \forall y \forall x (\neg R(x, y) \vee \neg R(y, z) \vee R(x, z)) \tag{2}$$
$$\forall x \forall y (\neg R(x, y) \vee R(y, x)) \tag{3}$$
$$\exists x \neg R(x, x) \tag{4}$$

In $\exists$-free prenex normal form, we have:

$$R(x, f(x)) \tag{1}$$
$$\neg R(x, y) \vee \neg R(y, z) \vee R(x, z) \tag{2}$$
$$\neg R(x, y) \vee R(y, x) \tag{3}$$
$$\neg R(a, a) \tag{4}$$

We perform resolution as follows:

1.	$\neg R(a, a)$	Conclusion
2.	$R(x, f(x))$	Premise 1
3.	$\neg R(x, y) \vee \neg R(y, z) \vee R(x, z)$	Premise 2
4.	$\neg R(x, y) \vee R(y, x)$	Premise 3
5.	$\neg R(a, y) \vee \neg R(y, a) \vee R(a, a)$	$(3, x := a, z := a)$
6.	$\neg R(a, y) \vee \neg R(y, a)$	Resolve $(1, 5)$
7.	$R(a, f(a))$	$(2, x := a)$
8.	$\neg R(a, f(a)) \vee \neg R(f(a), a)$	$(6, y := f(a))$
9.	$\neg R(f(a), a)$	Resolve $(7, 8)$
10.	$\neg R(a, f(a)) \vee R(f(a), a)$	$(4, x := a, y := f(a))$
11.	$\neg R(a, f(a))$	Resolve $(9, 10)$
12.	$\{\}$	Resolve $(7, 11)$

By soundness of resolution, the empty clause $\{\}$ implies that the original argument is valid. □

Chapter 3

Logic and Computation

3.1 The Halting Problem

An algorithm is a finite sequence of well-defined, computer-implementable instructions. We say that an algorithm solves a problem if, for every input, the algorithm produces the correct output. However, assuming unlimited time and space, there are problems that cannot be solved by an algorithm. Consider the *Halting Problem*: does there exist an algorithm that takes in a program P and an input I, and outputs *yes* if P halts on I and *no* otherwise?

Theorem 3.1.1

The Halting Problem is Unsolvable.

Proof. Assume a solution to the Halting Problem exists, defined as $H(P, I)$ for program P and input I. Construct a program $K(P)$ such that:

- If $H(P, P)$ halts, then $K(P)$ loops infinitely
- If $H(P, P)$ loops forever, $K(P)$ halts

Call $K(K)$. We have two cases now, the first being that $K(K)$ halts. This implies that $H(K, K)$ halts. However, by construction of H, this implies that $K(K)$ loops infinitely, which is a contradiction. In the other case, if $K(K)$ loops forever, then $H(K, K)$ loops forever. However, by construction of $K(K)$, it is implied that $K(K)$ halts. This is also a contradiction.

Since both cases led to contradiction, the existence of a halting program was invalid. □

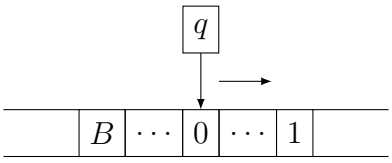
We now introduce the concept of a Turing machine: a mathematical model of the notion of algorithm. Informally, it consists of:

- A finite control unit (with a read-write head), which can be in any of a finite set of states
- A two-way infinite tape, divided into cells
- Read/write capabilities on the tape, as the finite control unit moves back and forth along the tape, changing states depending on the tape symbol read

Definition 3.1.1: Turing Machine

A Turing Machine $T = (S, I, f, s_0)$ consists of finite set S of states, finite input alphabet I containing the blank symbol B, the start state $s_0 \in S$, and a transition function

$$f : S \times I \to S \times I \times \{L, R\}$$



In the above figure, the control unit q contains the set S. Based on the infinite tape divided into cells, the action of the Turing Machine at each step of its operation depends on the value of the transition function for its current state and the current tape symbol being read in by q. More precisely, for current tape symbol x and state s, we define f for pair (s, x) as

$$f(s, x) = (s', x', d)$$

where $d = R$ or $d = L$. The Turing Machine enters state s' and writes symbol x' into the current cell, erasing x. We write the five-tuple

$$(s, x, s', x', d)$$

to describe this transition rule. Upon an undefined value for f on pair (s, x), the Turing Machine halts. Upon the beginning of operation, T is assumed to be in state s_0 and positioned on the left-most non-blank symbol on the tape.

- An alphabet Σ is a finite non-empty set of symbols, for example: $\Sigma = \{0, 1\}$ is an alphabet
- $\Sigma*$ denotes the set of all possible strings written by Σ
- A language L over Σ is a subset of $\Sigma*$

Turing Machines can be used to accept or recognize languages.

Definition 3.1.2: Final State

A final state of 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 of T

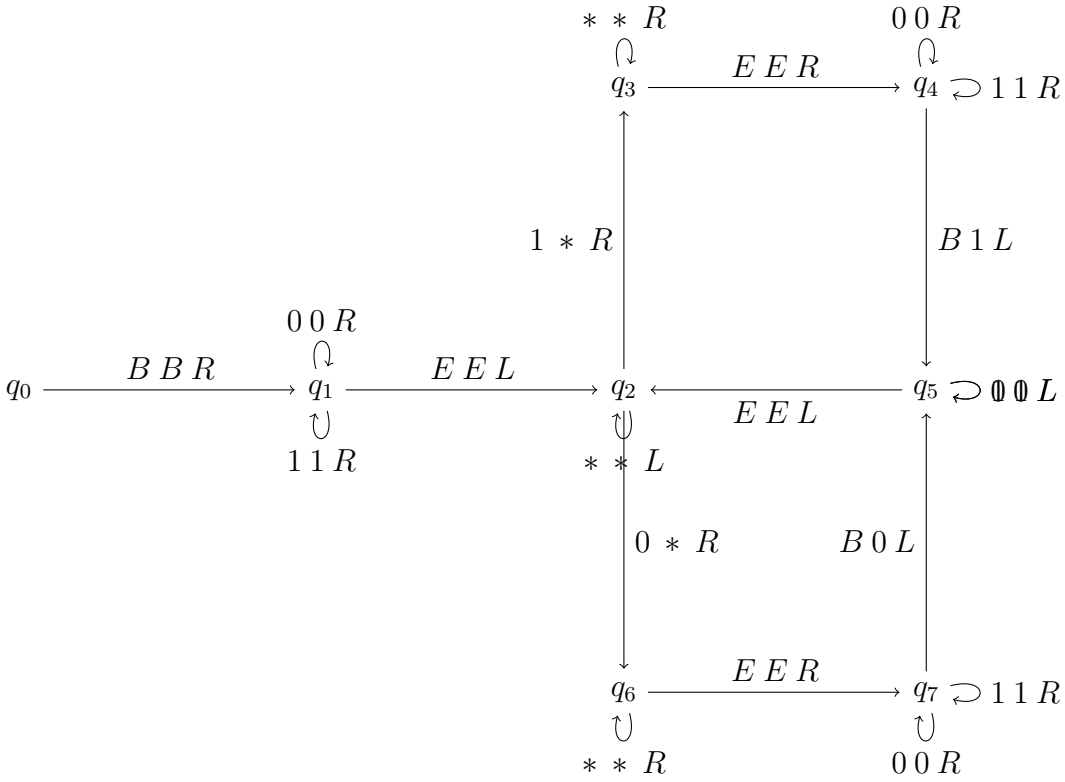
Definition 3.1.3: Acceptance

Let V be a subset of 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 is said to accept a language $L \subseteq V*$ if x is accepted by T iff $x \in L$.

A string $x \in V* \subseteq I*$ is not accepted if, when starting in the initial position with x written on the tape, either does not halt T or halts T in a non-final state.

We can define a Turing Machine by a transition diagram, where

- Each state is represented by a node
- Start state is a node with an incoming arrow, and final state is a double circle node
- A transition rule (s, x, x', x', d) is symbolized by an arrow between node s and node s' labelled by triplet x, x', d



Definition 3.1.4: Decider

A Turing Machine that always halts, on every input, is called a decider or a total Turing Machine.

The Turing machine is the accepted formalization of the notion of algorithm or computation. A universal Turing Machine, a single Turing Machine that can simulate the computations of every Turing Machine when given an encoding of the Turing Machine and its input, is the formalization of the notion of computer. A total Turing Machine is the formalization of the notion of terminating algorithm.

Theorem 3.1.2: Church-Turing Thesis

Any problem that can be solved with an algorithm can be solved by a Turing Machine.

It has been proved that a Turing Machine is equivalent in computing power to all the most general mathematical notions of computation. Thus, the Church-Turing Thesis can be used to prove if a given problem is solvable by a computer or not.

3.2 Decidability and Undecidability

Definition 3.2.1: Decision Problem

A decision problem is a yes-or-no question on an infinite set of inputs. Each input is an instance of the problem.

Definition 3.2.2: Decidability and Undecidability

A decision problem for which there exists a terminating algorithm that solves it is called decidable. If no such algorithm exists, the decision problem is called undecidable.

Definition 3.2.3: Computability and Uncomputability

A function that can be computed by a Turing Machine is called computable, otherwise it is called uncomputable.

Definition 3.2.4: S-membership

Let $S \subseteq \mathbb{N}$ be any subset. The S-membership problem asks: for an arbitrary $x \in \mathbb{N}$, is $x \in S$?

A set $S \subseteq \mathbb{N}$ is called decidable if the S-membership problem is decidable. Otherwise, it is undecidable. Some sets S are decidable and some are not. Finite $S \subset \mathbb{N}$ are decidable, but infinite $S \subset \mathbb{N}$ are not always undecidable. For example, $\{y \in \mathbb{N} : y \equiv 0 \pmod 2\}$ is infinite but decidable.

Theorem 3.2.1: Decidability of Complement

Suppose $S \subseteq \mathbb{N}$ is decidable. Then, $S^c = \mathbb{N} - S$ is decidable.

Proof. Let $x \in \mathbb{N}$ be given. If $x \in S$, then $x \notin S^c$. If $x \notin S$, then $x \in S^c$. □

If we want to solve a problem P_2 assuming we know how to solve P_1, we use the method of reduction.

- Reduce the known unsolvable problem P_1 to the problem P_2
- This will prove that: If P_2 is solvable, then P_1 is solvable
- The contrapositive of this implication is: If P_1 is unsolvable, then P_2 is unsolvable
- Since we know P_1 is unsolvable, then it follows that P_2 is unsolvable

Theorem 3.2.2

If problem P_1 is reducible to problem P_2, then if P_1 is undecidable, then P_2 is undecidable.

Theorem 3.2.3: Blank-Tape Halting Problem

The question: *Does Turing Machine M halt when started with a blank tape?* is undecidable.

Proof. We show this by reducing the halting problem to the blank-tape halting problem. Suppose we have a decider for the blank-tape halting problem. To reduce the halting problem to this decider, we must convert one problem instance of the halting problem decider with input M and w into machine M_w such that:

- If started on a blank tape, writes w
- Then continues execution like M

We now have that M halts on input string w iff M_w halts when started with blank tape. Since the halting problem is undecidable, then the blank-tape halting problem is undecidable based on this reduction. □

Theorem 3.2.4: State-Entry Problem

The question: Does Turing Machine M enter state q on input w? is undecidable.

Proof. We show this by reducing the halting problem to the state-entry problem. Suppose we have a decider for the state-entry algorithm on inputs M, w, and q. We want to build a decider for the halting problem with inputs M and w. To reduce the halting problem to the state-entry problem, we convert input (M, w) to (M', q, w) by:

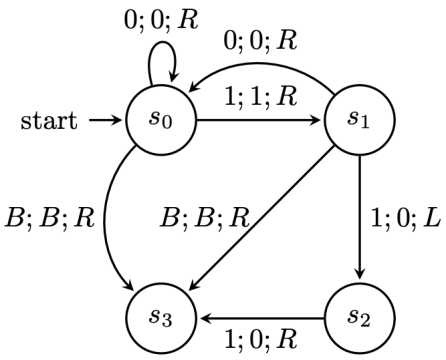
- Adding a new state q
- For any halting state of M, add transitions to q

Since M' has a single halt state q, M halts on input w iff M' halts on state q on input w. Since the halting problem is undecidable, then the state-entry problem is undecidable based on this reduction. □

3.3 Turing Machines

A configuration of Turing Machine $T = (S, I, f, s_0)$ is denoted by $\alpha_1 s \alpha_2$. $s \in S$ 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 up to the rightmost non-blank symbol. If $\alpha_2 = \lambda$, it is scanning a blank. For example, if T is defined by $S = \{s_0, s_1, s_2, s_3\}$ with $I = \{0, 1, B\}$ and f defined as $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, (s_0, B, s_3, B, R), $(s_1, 0, s_0, 0, R)$, $(s_1, 1, s_2, 0, L)$, (s_1, B, s_3, B, R), and $(s_2, 1, s_3, 0, R)$, we compute input 010110 as

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



When constructing Turing Machines as computing number-theoretic funtions, we use unary representations where the natural number n is represented by a string of $(n + 1)$ 1s.

A Wang tile is a square painted with a colour on each side. Two adjacent tiles are said to match if they have the same colour at adjacent sides. Tiles cannot be rotated. A tile system T is a finite set of tiles, and a tiling is valid if all adjacent sides of neighbouring tiles have matching colours.

Definition 3.3.1: Tiling Problem

Given a tile system T, can any square, of any size, be tiled using only the tile types in T?

Definition 3.3.2: Seeded Tiling Problem

Given a tile system T and a specified seed tile $t_0 \in T$, does there exist a valid tiling of the plane with tiles from T that contains the seed tile t_0?

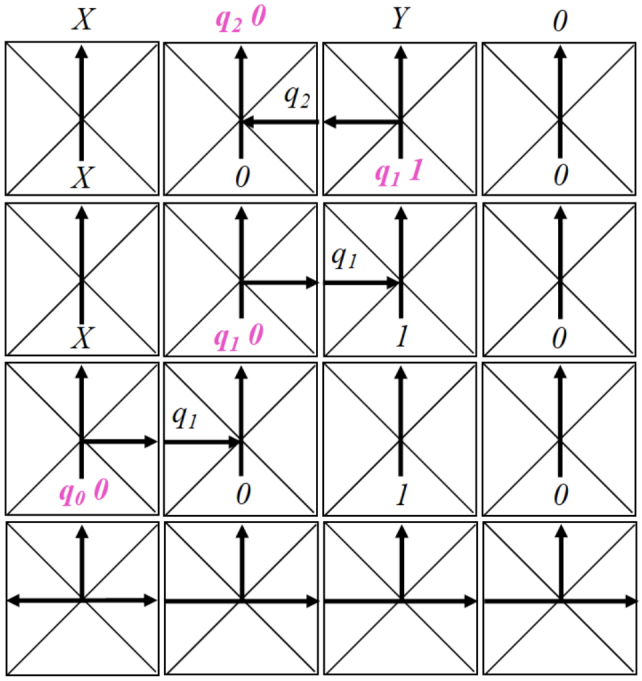
Theorem 3.3.1

The (seeded) Tiling Problem is undecidable.

Proof. We prove this through reducing the Halting Problem for Turing Machines to the Tiling Problem. Consider $T = (S, I, f, q_0)$ with state set $S = \{q_0, q_1, ..., q_n\}$ with start state $q_0 \in S$, alphabet $I = \{s_0, s_1, ..., s_n\}$ with s_0 denoting the blank, and transition function $f : S \times I \to S \times I \times \{L, R\}$. We take a set of colours with the notion of colour matching generalized as relation $R \subseteq C \times C$ where colours $c_1, c_2 \in C$ match iff $(c_1, c_2) \in R$.

In this case, our colours will be an alphabet symbol in I, a state symbol S, or a state symbol followed by an alphabet symbol. We define other colours as arrow heads and arrow tails. An infinite horizontal row of tiles simulates T's configuration flanked by blanks. On a row of tiles, the concatenation of all north side labels is the configuration of T. A transition step update of T is simulated by placing on top of the current row representing the current configuration, a row representing the next configuration. Rows are aligned so that the row on top of a given tile is the same tile in its next configuration. T starts from the initial configuration with a blank tape and proceeds upwards. Tile system $Tile(T)$ has five different types of tiles:

- Alphabet tiles (simulating alphabet symbols) - copies current configuration to the next configuration unchanged, upwards
- Action tiles (simulating transition rules) and Merge tiles (simulating left/right move of the read/write head) - together, transition rules are implemented for left and right movements
- Starting tiles (simulating a blank tape, including the seed tile) - coloured with s_0 or $q_0 s_0$ on their north side, specific white label on south side
- Filler tile (tiles the bottom half-plane) - white on all sides, top half-plane tiled by Turing Machine computations



Given T, the tile system $Tile(T)$ has a valid tiling of the plane containing its seed tile iff the computation of T starting with the blank tape never halts. Starting with the seed tile of $Tile(T)$, the only valid tiling of the plane is a tiling that simulates a computation of T, row by row, starting from the seed tile and proceeding upwards, to cover the top half-plane. Each row simulates a configuration of T, with consecutive rows representing successive configurations (obtained by one application of a transition rule). The bottom half-plane is tiled with filler tiles. Since the Halting Problem on a Blank Tape for Turing Machines is undecidable, the (seeded) Tiling Problem is also undecidable □

A deterministic Turing Machine DTM is such that transition rules are defined by the partial function $f : S \times I \to S \times I \times \{L, R\}$. Since f is a function, no two five-tuples begin with the same pair (s, x). For each state s and symbol x, there is at most one transition rule applicable.

In a non-deterministic Turing Machine $NDTM$, transition rules are defined as a relation rather than a function, so more than one transition rule may be applicable for each state s and symbol x.

Definition 3.3.3: Class of Polynomial Time

A decision problem is in P, the class of polynomial time problems, if it can be solved by a deterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in P if there is a deterministic Turing Machine T that solves the decision problem, and a polynomial $p(n)$ such that for all integers n, T halts in a final state after no more than $p(n)$ transitions whenever the input to T is a string of length n.

Definition 3.3.4: Class of Non-Deterministic Polynomial Time

A decision problem is in NP, the class of non-deterministic polynomial time problems, if it can be solved by a non-deterministic Turing machine in polynomial time in terms of the size of the input. That is, a decision problem is in NP if there is a non-deterministic Turing machine T that solves the problem, and a polynomial $p(n)$ such that for all integers n, T halts for every choice of transitions after no more than $p(n)$ transitions, whenever the input to T is a string of length n.

$P \subseteq NP$ because problems solvable in polynomial time are also verifiable in polynomial time. Problems in P are tractable if they have efficient algorithms, and intractable otherwise. For a problem not to be in P, we prove that no polynomial time algorithm to solve it exists. For a problem to be in NP, there must exist a deterministic Turing machine that, when given a problem instance and its answer, can verify in polynomial time that it is correct.

Alternatively, a decision problem is in NP if problem instances where the answer is *yes* can be verifiable in polynomial time by a deterministic Turing machine. It is widely believed that $P \neq NP$ because some problems in NP are harder to solve in polynomial time than to verify in polynomial time.

Definition 3.3.5: NP-complete

A problem in NP is called NP-complete if every other problem in NP can be reduced to it in polynomial time.

The problem: *Given a formula in propositional logic, using n propositional symbols, is the formula satisfiable?* is NP-complete. This consists of an exhaustive search of all possible 2^n truth valuations.

Chapter 4

Peano Arithmetic

4.1 Theories in First-Order Language

Recall the rules of formal deduction for first-order logic. With these rules, we can prove the following properties of equality:

Theorem 4.1.1: Properties of Equality

$$\begin{aligned} \forall x(x = x) &\quad \text{Reflexivity} \\ \forall x \forall y((x = y) \to (y = x)) &\quad \text{Symmetry} \\ \forall x \forall y \forall z((x = y) \wedge (y = z) \to (x = z)) &\quad \text{Transitivity} \end{aligned}$$

Proof. Proof of reflexivity by formal deduction:

1. $\emptyset \vdash u = u \quad (\approx +)$
2. $\emptyset \vdash \forall x(x = x) \quad (1, \forall +)$

Proof of symmetry by formal deduction:

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. $\emptyset \vdash (u = v) \to (v = u) \quad (4, \to +)$
6. $\emptyset \vdash \forall y((u = y) \to (y = u)) \quad (5, \forall +)$
7. $\emptyset \vdash \forall x \forall y((x = y) \to (y = x)) \quad (6, \forall +)$

Proof of transitivity by formal deduction:

1.	$(u = v) \wedge (v = w) \vdash (u = v) \wedge (v = w)$	$(\in)$
2.	$(u = v) \wedge (v = w) \vdash (u = v)$	$(1, \wedge -)$
3.	$(u = v) \wedge (v = w) \vdash (v = w)$	$(1, \wedge -)$
4.	$(u = v) \wedge (v = w) \vdash (u = w)$	$(2, 3, \approx -)$
5.	$\emptyset \vdash (u = v) \wedge (v = w) \rightarrow (u = w)$	$(4, \rightarrow +)$
6.	$\emptyset \vdash \forall z((u = v) \wedge (v = z) \rightarrow (u = z))$	$(5, \forall +)$
7.	$\emptyset \vdash \forall y \forall z((u = y) \wedge (y = z) \rightarrow (u = z))$	$(6, \forall +)$
8.	$\emptyset \vdash \forall x \forall y \forall z((x = y) \wedge (y = z) \rightarrow (x = z))$	$(7, \forall +)$

□

First-order logic is often useful for proofs in specialized domains. Assuming a number of domain axioms, first-order logic can be used to prove additional theorems in that domain. A set of domain axioms, together with a system of formal deduction, and all theorems that can be proved from the domain axioms, is called a theory. Examples: number theory, set theory, graph theory, etc. The set A of domain axioms should be:

- Decidable: a terminating algorithm must exist to decide whether a given formula is a domain axiom
- Consistent (recall satisfiability $\iff$ consistency)
- Syntactically complete: for any formula F describable in the language, either F or $\neg F$ must be provable

4.2 Axioms and Proofs in Peano Arithmetic

Peano's axioms are the basis of the Peano arithmetic branch of number theory. It contains the equality relation, the individual symbol 0, functions for the successor, addition, and multiplication, and an axiom for induction.

Axiom 4.2.1: Peano Arithmetic Axioms

$$\begin{aligned}
\forall x(\neg(s(x) = 0)) &\quad \text{PA1}\\
\forall x\forall y(s(x) = s(y) \to x = y) &\quad \text{PA2}\\
\forall x(x + 0 = x) &\quad \text{PA3}\\
\forall x\forall y(x + s(y) = s(x + y)) &\quad \text{PA4}\\
\forall x(x \cdot 0 = 0) &\quad \text{PA5}\\
\forall x\forall y(x \cdot s(y) = x \cdot y + x) &\quad \text{PA6}\\
(A(0) \wedge \forall x(A(x) \to A(s(x)))) \to \forall x A(x) &\quad \text{PA7}
\end{aligned}$$

In a Peano arithmetic proof, the set PA is implicitly considered part of the premises. Given a theory T, with axioms A_T, we use $\Sigma \vdash_{A_T} C$ to denote $\Sigma, A_T \vdash C$. Thus, in Peano arithmetic, we have $\Sigma \vdash_{PA} C$ to denote $\Sigma, PA \vdash C$.

Theorem 4.2.1

$\forall x(s(x) \neq x)$

Proof. By Peano arithmetic and formal deduction:

1.	$\emptyset \vdash_{PA} \forall x(s(x) \neq 0)$	$(PA1)$
2.	$\emptyset \vdash_{PA} s(0) \neq 0$	$(\forall -, 1)$
3.	$s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(s(k)) = s(k)$	$(\in)$
4.	$\forall x\forall y(s(x) = s(y) \to x = y)$	$(PA2)$
5.	$\emptyset \vdash_{PA} s(s(k)) = s(k) \to s(k) = k$	$(\forall -, \forall -, 4)$
6.	$s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(s(k)) = s(k) \to s(k) = k$	$(5, +)$
7.	$s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(k) = k$	$(\to -, 3, 5)$
8.	$s(k) \neq k, s(s(k)) = s(k) \vdash_{PA} s(k) \neq k$	$(\in)$
9.	$s(k) \neq k \vdash_{PA} s(s(k)) \neq s(k)$	$(\neg +, 7, 8)$
10.	$\emptyset \vdash_{PA} s(k) \neq k \to s(s(k)) \neq s(k)$	$(\to +, 9)$
11.	$\emptyset \vdash_{PA} \forall x(s(x) \neq x \to s(s(x)) \neq s(x))$	$(\forall +, 10)$
12.	$\emptyset \vdash_{PA} s(0) \neq 0 \wedge \forall x(s(x) \neq x \to s(s(x)) \neq s(x))$	$(\wedge +, 2, 11)$
13.	$\emptyset \vdash_{PA} s(0) \neq 0 \wedge \forall x(s(x) \neq x \to s(s(x)) \neq s(x)) \to \forall x(s(x) \neq x)$	$(PA7)$
14.	$\forall x(s(x) \neq x)$	$(\to -, 12, 13)$

□

Other theorems provable by Peano arithmetic include associativity, distributivity, and commutativity of addition and multiplication. We can also prove relations such as the transitivity of $\leq$.

Theorem 4.2.2

$\leq$ is transitive.

Proof. By Peano arithmetic and formal deduction:

1.	$u + \alpha_1 = v, v + \alpha_2 = w \vdash_{PA} u + \alpha_1 = v$	$(\in)$
2.	$u + \alpha_1 = v, v + \alpha_2 = w \vdash_{PA} v + \alpha_2 = w$	$(\in)$
3.	$u + \alpha_1 = v, v + \alpha_2 = w \vdash_{PA} (u + \alpha_1) + \alpha_2 = w$	$(1, 2, (\approx -)')$
4.	$u + \alpha_1 = v, v + \alpha_2 = w \vdash_{PA} u + (\alpha_1 + \alpha_2) = w$	$(3, \text{assoc. })$
5.	$u + \alpha_1 = v, v + \alpha_2 = w \vdash_{PA} \exists y(u + y = w)$	$(4, \exists +)$
6.	$u + \alpha_1 = v, v + \alpha_2 = w \vdash_{PA} u \leq w$	$(5, \leq)$
7.	$\exists y(u + y = v), \exists y(v + y = w) \vdash_{PA} u \leq w$	$(6, \exists -, \exists -)$
8.	$u \leq v, v \leq w \vdash_{PA} u \leq w$	$(7, \leq)$
9.	$(u \leq v) \wedge (v \leq w) \vdash_{PA} (u \leq v) \wedge (v \leq w)$	$(\in)$
10.	$(u \leq v) \wedge (v \leq w) \vdash_{PA} u \leq v$	$(9, \wedge -)$
11.	$(u \leq v) \wedge (v \leq w) \vdash_{PA} v \leq w$	$(9, \wedge -)$
12.	$(u \leq v) \wedge (v \leq w) \vdash_{PA} u \leq w$	$(10, 11, 8, Tr.)$
13.	$\emptyset \vdash_{PA} (u \leq v) \wedge (v \leq w) \to u \leq w$	$(12, \to +)$
14.	$\emptyset \vdash_{PA} \forall x \forall y \forall z((x \leq y) \wedge (y \leq z) \to x \leq z)$	$(13, \forall +, \forall +, \forall +)$

□

In fact, we can prove or disprove any formula describable in the language. However, if a formula is not describable in terms of Peano's axioms, then it is not provable or disprovable.

4.3 Gödel's Incompleteness Theorem

Theorem 4.3.1: Gödel's Incompleteness Theorem

In any consistent formula theory T with a decidable set of axioms, that is capable of expressing elementary arithmetic, there exists a formula that can neither be proved or disproved in the theory.

Proof. For the sake of contradiction, assume that any statement can be proved or disproved by theory T. Construct a Turing machine that takes program P and input I and:

- Generate all strings s of all lengths
- For each string s, the program checks whether s is a correct formal proof for P halts on I

However, this would imply that the Halting problem is solvable, which is incorrect. We have reached a contradiction. □

The Paris-Harrington theorem is one expressible in Peano arithmetic but is unprovable. This does not contradict the completeness of $\vdash$. The completeness of $\vdash$ requires that if $PA \models G$ for theorem G, then $P \vdash G$. However, $PA \models G$ does not hold, despite G's truth value on a standard interpretation of $\mathbb{N}$. There are non-standard interpretations of $\mathbb{N}$ that make PA true, but G false. (This is not covered in this course)

Chapter 5

Program Verification

5.1 Program States and Correctness

The proof calculus of formal program verification is one that formally states the specification of a problem and proves that the program satisfies the specification for all units. In this course, we learn to prove that a program satisfies a formula. These program assertions will be of the form of the Hoare triple.

Definition 5.1.1: Hoare Triple

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. This is represented as

$$(\!|P|\!)C(\!|Q|\!)$$

where P is the precondition, C is the program, and Q is the postcondition.

We call

$$(\!|P|\!)C(\!|Q|\!)$$

a specification of C.

5.2 Proving Correctness

Definition 5.2.1: Partial Correctness

A Hoare triple $(\!|P|\!)C(\!|Q|\!)$ is satisfied under partial correctness, denoted

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

iff for each state s satisfying P, if the program C starting in state s terminates in state s', then s' satisfies the condition Q.

Definition 5.2.2: Total Correctness

A Hoare triple $(\!|P|\!)C(\!|Q|\!)$ is satisfied under total correctness, denoted

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

iff for each state s satisfying P, execution of program C starting in state s terminates, and the resulting state s' satisfies the condition Q.

Simply put, total correctness is partial correctness plus the guarantee of termination.

Axiom 5.2.1: Program Inference Rules

$$\frac{}{(\!|Q[E/x]|\!)\, x = E\, (\!|Q|\!)} \quad \text{Assignment}$$

$$\frac{P \to P' \;\; (\!|P'|\!)C(\!|Q|\!)}{(\!|P|\!)C(\!|Q|\!)} \quad \text{Precondition Strength}$$

$$\frac{(\!|P|\!)C(\!|Q'|\!) \;\; Q' \to Q}{(\!|P|\!)C(\!|Q|\!)} \quad \text{Postcondition Strength}$$

$$\frac{(\!|P|\!)C_1(\!|Q|\!) \;\; (\!|Q|\!)C_2(\!|R|\!)}{(\!|P|\!)C_1; C_2(\!|R|\!)} \quad \text{Composition}$$

$$\frac{(\!|P \wedge B|\!)C_1(\!|Q|\!) \;\; (\!|P \wedge \neg B|\!)C_2(\!|Q|\!)}{(\!|P|\!) \text{ if (B) } C_1 \text{ else } C_2 \; (\!|Q|\!)} \quad \text{If-then-else}$$

$$\frac{(\!|P \wedge B|\!)C_1(\!|Q|\!) \;\; (\!|P \wedge \neg B|\!)C_2(\!|Q|\!)}{(\!|P|\!) \text{ if (B) } C \; (\!|Q|\!)} \quad \text{If-then}$$

$$\frac{(\!|I \wedge B|\!)C(\!|I|\!)}{(\!|I|\!) \text{ while (B) } C \; (\!|I \wedge \neg B|\!)} \quad \text{Partial while}$$

Definition 5.2.3: Loop Invariant

A loop invariant is an assertion that is both true before and after each execution of the body of a loop.

5.3 Undecidability of Program Verification

The Total Correctness problem is: *Is a given Hoare triple* $(\!|P|\!)C(\!|Q|\!)$ *satisfied under total correctness?*

Theorem 5.3.1

The Total Correctness problem is undecidable.

Proof. Suppose we have a terminating algorithm A to solve the Total Correctness problem. Given program C as input, construct program C' that erases any input x to C, and runs C on a blank tape. We should now be able to test whether $(\!|true|\!)C'(\!|true|\!)$ is totally correct. However, this is only true iff C' halts on the blank tape. Since the Blank Tape halting problem is undecidable, then the Total correctness problem is undecidable by reduction. □

The Partial Correctness problem is: *Is a given Hoare triple* $(\!|P|\!)C(\!|Q|\!)$ *satisfied under partial correctness?*

Theorem 5.3.2

The Partial correctness problem is undecidable.

Proof. Suppose we have a terminating algorithm A to solve the Partial Correctness problem. Given program C as input, construct program C' that adds line $x = 1$ to the end of C. We should now be able to test whether $(\!|true|\!)C'(\!|x = 0|\!)$ is partially correct. However, this is only true iff C does not halt on the blank tape. Since the Blank Tape halting problem is undecidable, then the Partial correctness problem is undecidable by reduction. □