Notes for Logic (COMP0009)

Raphael Li

Sep 2025

Contents

1 Revision: The syntax and semantics of propositional and first-order logic

Formally, a *logic* consists of three components:

Component	Describes...
Syntax	The language and grammar for writing formulas
Semantics	How formulas are interpreted
Inference system (or proof system)	A syntactic device for proving true statements

Table 1: The three key components of a logic.

This module concerns algorithms that automatically parse and determine the validity of a formula.

1.1 Propositional logic

1.1.1 Syntax

Formulas are constructed by applying negation, conjunction and disjunction to propositions.

$$\begin{aligned} \text{proposition} &\coloneqq p \mid q \mid r \mid \cdots \\ \text{formula} &\coloneqq \text{proposition} \mid \neg\text{formula} \mid (\text{formula} \circ \text{formula}) \qquad (\text{where } \circ \text{ is } \wedge, \vee \text{ or } \rightarrow) \end{aligned}$$

A proposition or its negation is called a *literal*[1].

For any formula that isn't a proposition, the *main connective* is the one with the largest scope. In other words, it is not in the scope of any other connective.

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

This is the connective with which evaluation begins. This is especially important when building parsers for algorithmically evaluating formulas.

Note that parsers working according to the above definition will recognise $(p \wedge q)$, but not $p \wedge q$, as a formula. Regardless, throughout this document we will use a looser definition where brackets may be ommitted in unambiguous cases.

1.1.2 Semantics

A valuation is a function v that maps each proposition to a truth value in $\{\top, \bot\}$.

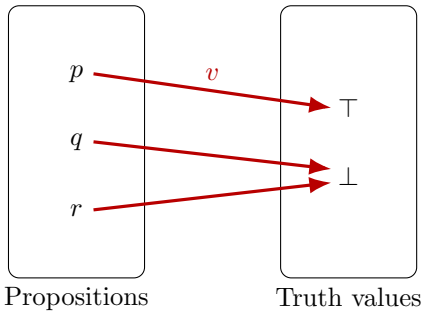

Figure 1: A valuation maps propositions to truth values.

[1]For example, p and $\neg p$ are both literals, but $\neg\neg q$ is not.

A valuation v can be extended to a unique *truth function* defined on all possible formulas. A truth function v' must satisfy

$$\begin{aligned} v'(\neg\phi) = \top &\iff v'(\phi) = \bot \\ v'(\phi \vee \psi) = \top &\iff v'(\phi) = \top \text{ or } v'(\psi) = \top \\ v'(\phi \wedge \psi) = \top &\iff v'(\phi) = \top \text{ and } v'(\psi) = \top \\ v'(\phi \to \psi) = \top &\iff v'(\phi) = \bot \text{ or } v'(\psi) = \top \\ v'(\phi \leftrightarrow \psi) = \top &\iff v'(\phi) = v'(\psi) \end{aligned}$$

for all formulas ϕ and ψ. From now on we use v to denote the more general truth function.

The result of applying a valuation v to a formula ϕ depends only on the propositional letters that occur in ϕ.

A formula ϕ is *valid* if $v(\phi) = \top$ for all valuations v, which we denote as $\models \phi$. A formula ϕ is *satisfiable* if $v(\phi) = \top$ for at least one valuation v. All valid formulas are satisfiable, but *not* vice versa.

Two formulas ϕ and ψ are *logically equivalent*, written as $\phi \equiv \psi$, if and only if for every valuation v we have $v(\phi) = v(\psi)$.

1.1.3 Truth tables

Consider the propositional formula $((p \vee \neg q) \wedge \neg(q \wedge r))$. We can check its validity and satisfiability by constructing its truth table.

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

Table 2: The truth table for the formula $((p \vee \neg q) \wedge \neg(q \wedge r))$.

In this case, the formula is satisfiable but not valid.

1.1.4 Parse trees

A parser interprets the semantics of a formula by breaking down its symbols into a *parse tree*, which shows the syntactic relation between symbols. For example, the formula $((p \vee \neg q) \wedge \neg(q \wedge r))$ can be broken down into the following parse tree.

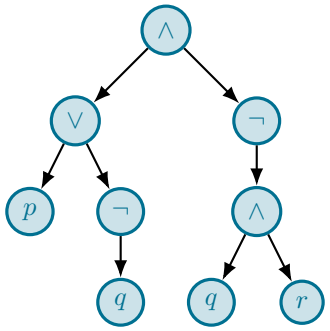

Figure 2: The parse tree for the formula $((p \vee \neg q) \wedge \neg(q \wedge r))$.

1.1.5 Disjunctive normal form (DNF)

A formula is said to be in *disjunctive normal form* (DNF) if it is a disjunction of one or more conjunctions of one or more literals.

$$\begin{aligned} \text{proposition} &:= p \mid q \mid r \mid \cdots \\ \text{literal} &:= \text{proposition} \mid \neg\text{proposition} \\ \text{conjunctiveClause} &:= \text{literal} \mid \text{literal} \ \wedge \ \text{conjunctiveClause} \\ \text{DNF} &:= \text{conjunctiveClause} \mid \text{conjunctiveClause} \ \vee \ \text{DNF} \end{aligned}$$

Below is an example of a formula in DNF.

$$\underbrace{(p \wedge \neg q \wedge \neg r)}_{\substack{\text{conjunctive} \\ \text{clause}}} \vee \underbrace{(\neg p \wedge \neg q \wedge r)}_{\substack{\text{conjunctive} \\ \text{clause}}} \vee \underbrace{(q \wedge \neg r)}_{\substack{\text{conjunctive} \\ \text{clause}}}$$

Any propositional formula has a DNF equivalent. For instance, the formula $(p \vee \neg q) \wedge \neg(q \wedge r)$ can be rewritten as follows.

$$\begin{aligned} & (p \vee \neg q) \wedge \neg(q \wedge r) \\ \iff & (p \vee \neg q) \wedge (\neg q \vee \neg r) && \text{(De Morgan's law, to remove outer negation)} \\ \iff & ((p \vee \neg q) \wedge \neg q) \vee ((p \vee \neg q) \wedge \neg r) && \text{(distributing conjunctions over disjunctions)} \\ \iff & (p \wedge \neg q) \vee (\neg q \wedge \neg q) \vee (p \wedge \neg r) \vee (\neg q \wedge \neg r) && \text{(distributing conjunctions over disjunctions)} \\ \iff & (p \wedge \neg q) \vee \neg q \vee (p \wedge \neg r) \vee (\neg q \wedge \neg r) \end{aligned}$$

Alternatively, this can also be achieved by referring to the truth table. From Table 2, we see that the formula can be written in DNF as

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

1.1.6 Conjunctive normal form (CNF)

A formula is said to be *conjunctive normal form* (CNF) if it is a conjunction of one or more disjunctions of one or more literals.

$$\begin{aligned} \text{disjunctiveClause} &:= \text{literal} \mid \text{literal} \ \vee \ \text{disjunctiveClause} \\ \text{CNF} &:= \text{disjunctiveClause} \mid \text{disjunctiveClause} \ \wedge \ \text{CNF} \end{aligned}$$

Below is a formula in CNF.

$$\underbrace{(p \vee \neg q \vee \neg r)}_{\substack{\text{conjunctive} \\ \text{clause}}} \wedge \underbrace{(\neg p \vee q \vee r)}_{\substack{\text{conjunctive} \\ \text{clause}}}$$

To find the CNF equivalent of a formula ϕ, we first express its negation $\neg\phi$ in DNF. Then, we negate it again to get $\neg\neg\phi$. Using De Morgan's law, the resultant formula will be in CNF.

For example, let ϕ be the formula $(p \vee \neg q) \wedge \neg(q \wedge r)$. To rewrite it in CNF, we start by constructing the truth table of its negation $\neg\phi$. This allows us to express $\neg\phi$ in DNF.

p	q	r	$((p \vee \neg q) \wedge \neg(q \wedge r))$	Negation of $((p \vee \neg q) \wedge \neg(q \wedge r))$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Table 3: The truth table for the negation of $((p \vee \neg q) \wedge \neg(q \wedge r))$. This is obtained by flipping the results of Table 2.

Hence we have

$$\begin{aligned} \neg\phi &= (\neg p \wedge q) \vee (p \wedge q \wedge r) && \text{(DNF of } \neg\phi) \\ \neg\neg\phi &= \neg((\neg p \wedge q) \vee (p \wedge q \wedge r)) && \text{(negating both sides)} \\ \phi &= (p \vee \neg q) \wedge (\neg p \vee \neg q \vee \neg r) && \text{(double negation; De Morgan's laws)} \end{aligned}$$

which gives us ϕ in CNF.

1.2 First-order logic

1.2.1 Syntax

A first-order language $L(C, F, P)$ is determined by a set C of constant symbols, a set F of function symbols and a non-empty set P of predicate symbols. Each function symbol and predicate symbol has an associated *arity* $n \in \mathbb{N}$. We write f^n and p^n to represent an n-ary function symbol and an n-ary predicate symbol respectively. Moreover, let V be a countably infinite set of variable symbols.

$$\begin{aligned} \text{term} &:= c \mid v \mid f^n(\text{term}_0, \text{term}_1, \cdots, \text{term}_{n-1}) && \text{(where } c \in C, v \in V \text{ and } f^n \in F) \\ \text{atom} &:= p^n(\text{term}_0, \text{term}_1, \cdots, \text{term}_{n-1}) && \text{(where } p^n \in P) \\ \text{formula} &:= \text{atom} \mid \neg\text{formula} \mid (\text{formula}_0 \vee \text{formula}_1) \mid \exists v\ \text{formula} && \text{(where } v \in V) \end{aligned}$$

This definition is functionally complete. Formulas involving universal quantifiers, implications and equivalence symbols can always be rewritten using only symbols defined above.

A *closed term* is a term with no variable symbols. A *sentence* is a formula with no free variables.

1.2.2 Semantics

For a first-order language $L(C, F, P)$, we may construct a corresponding first-order structure[2] $S = (D, I)$ where $I = (I_c, I_f, I_p)$.

$$S = (\underbrace{D}_{\substack{\text{non-empty} \\ \text{domain}}}, \overbrace{(I_c, I_f, I_p)}^{\text{interpretation } I})$$

Here,

- I_c maps each constant symbol in C to an element of D.
- I_f maps each n-ary function symbol in F to an n-ary function over D.
- I_p maps each n-ary predicate symbol $p \in P$ to an n-ary relation over D (i.e. a subset of D^n).

[2] Also known as an L-structure.

- We may occasionally use I to denote a general interpretation function where

$$\begin{aligned} I(c) &= I_c(c) && \text{(for all } c \in C\text{)} \\ I(f) &= I_f(f) && \text{(for all } f \in F\text{)} \\ I(p) &= I_p(p) && \text{(for all } p \in P\text{)} \end{aligned}$$

If P includes the equality symbol $=$, then it is always interpreted as the binary relation of true equality.

$$I_p(=) = \{(d, d) : d \in D\}$$

Given a structure $S = (D, I)$, a variable assignment A is a map from V to D. For any variable $v \in V$, two variable assignments A and A^* are said to be v-equivalent if $A(x) = A^*(x)$ for all $x \in V \setminus \{v\}$. In other words, two variable assignments are said to be v-equivalent if they are completely identical except possibly for the element in D assigned to v. This is written as $A \equiv_v A^*$.

Given a structure S and a variable assignment A, we may interpret any term as follows.

$$\begin{aligned} c^{S,A} &= I_c(c) \\ v^{S,A} &= A(v) \\ f^n(t_0, t_1, \cdots, t_{n-1})^{S,A} &= \underbrace{(I_f(f^n))}_{\substack{\text{interpreted} \\ \text{function}}}(t_0^{S,A}, t_1^{S,A}, \cdots, t_{n-1}^{S,A}) \end{aligned}$$

Formulas are evaluated as follows.

$$\begin{aligned} S \models_A p^n(t_0, t_1, \cdots, t_{n-1}) &\iff (t_0^{S,A}, t_1^{S,A}, \cdots, t_{n-1}^{S,A}) \in I_p(p^n) \\ S \models_A \neg\text{formula} &\iff S \not\models_A \text{formula} \\ S \models_A (\text{formula}_0 \vee \text{formula}_1) &\iff S \models_A \text{formula}_0 \text{ or } S \models_A \text{formula}_1 \\ S \models_A \exists v \text{ formula} &\iff S \models_{A[x \mapsto d]} \text{formula for some } d \in D \end{aligned}$$

Given a structure S and a formula ϕ, we say that

- ϕ is "valid in S" if $S \models_A \phi$ for every variable assignment A. This is written as $S \models \phi$.
- ϕ is "satisfiable in S" if $S \models_A \phi$ for some variable assignment A.
- ϕ is "valid" if ϕ is valid in all possible structures. This is written as $\models \phi$.
- ϕ is "satisfiable" if there exists some structure in which ϕ is satisfiable.

A formula ϕ is valid if and only if $\neg\phi$ is not satisfiable.

Proof. Let $\neg\phi$ be a formula that is not satisfiable. Hence we have

$$\begin{aligned} \neg\exists S\ \exists A \quad S \models_A \neg\phi &\iff \neg\exists S\ \exists A \quad S \not\models_A \phi \\ &\iff \forall S\ \neg\exists A \quad S \not\models_A \phi \\ &\iff \forall S\ \forall A \quad \neg S \not\models_A \phi \\ &\iff \forall S\ \forall A \quad S \models_A \phi \end{aligned}$$

which means S is valid.

If ϕ is a sentence, then ϕ is valid in S if and only if it is also satisfiable in S.

1.2.3 Example: Arithmetic in the set of natural numbers

Consider the first-order language $L(C, F, P)$ defined as follows. Also assume a countably infinite set V of variable symbols.

$$\begin{aligned} C &= 1, 2, 3, \cdots && \text{(constant symbols)} \\ F &= \{+, \times\} && \text{(function symbols, both binary)} \\ P &= \{=, <\} && \text{(predicate symbols, both binary)} \\ V &= \{x, y, z, \cdots\} && \text{(variable symbols)} \end{aligned}$$

A term is a string of symbols that represents a "thing" or an "object" — this can be a constant, a variable, or a function output.

- x
- $1 + 3$
- $2 \times x + 1$

Of the terms shown above, only the second one is a closed terms because it has no variable symbols.

An atom is a string of symbols that represents the output of a predicate, which is a truth value.

- $1 = 2$
- $y < 3$
- $x + 1 < 2 \times z + 3$

Finally, a formula is constructed by applying negations, disjunctions, and existential quantifiers to atoms.

- $1 = 2 \ \wedge \ y < 3$
- $\neg \exists z \ \ x + 1 < 2 \times z + 3$

The latter example is a sentence because all of its variable symbols are bounded.

For this particular first-order language, we may use the structure of ordinary arithmetic[3], defined as $N = \{\mathbb{N}, \{I_c, I_f, I_p\}\}$ where

- I_c is a function that maps numerical symbols to the corresponding natural number.

$$\begin{aligned} I_c(1) &= 1 \\ I_c(2) &= 2 \\ I_c(3) &= 3 \\ &\vdots \end{aligned}$$

- I_f maps $+$ and $\times$ to the addition and multiplication operations in arithmetic respectively.
- I_p maps $=$ and $<$ to the following relations.

$$\begin{aligned} I_p(=) &= \{(n, n) : n \in \mathbb{N}\} \\ I_p(<) &= \{(m, n) \in \mathbb{N}^2 : m < n\} \end{aligned}$$

[3]There is also a similar structure $R = (\mathbb{R}, I)$ where the domain is the set of real numbers.

1.2.4 First-order structures and directed graphs

Consider a first-order language with only one binary predicate symbol p.

$$L(C, F, \{p\})$$

Any first-order structure $S = \{D, \{I_c, I_f, I_p\}\}$ for this language can be represented as a directed graph, where each vertex is an element of D and each directed edge represents an element of the relation $I_p(p)$.

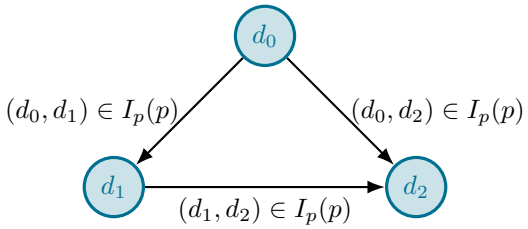

Figure 3: The first-order structure S can be visualised as a directed graph.

2 Axiomatic Proofs for Propositional Logic

A *proof system* is a system for determining the validity of formulas.

An obvious system would be to construct a truth table and check that all rows give a true result. However, this naive approach has an exponential time complexity[4], meaning that it will become increasingly impractical as more and more propositions are introduced.

To alleviate this issue, we shall introduce a different approach called a *Hilbert-style proof system*. This is an *axiomatic proof system* in which theorems are generated using axioms and inference rules.

2.1 Hilbert-style proof system

Firstly, we limit our propositional language to only use the connectives $\neg$ and $\rightarrow$. Double negations are prohibited.

Moreover, we will note some *axioms* that are known to be valid, and then try to derive other valid formulas from the axioms. Below we list three examples of *schemas*, from which axioms may be obtained by substituting any formulas in place of p, q and r.

I. $p \rightarrow (q \rightarrow p)$ (implication is true if consequent is true)

II. $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$ (implication chain as hypothetical syllogism)

III. $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$ (contrapositive)

Axioms on their own are insufficient in establishing a proof system. We also need *inference rules*, which stipulate how conclusions can be derived from premises. One of the main inference rules is *modus ponens*, which states that if you have proved both the formula ϕ and the implication $(\phi \rightarrow \psi)$, then you may deduce the conclusion ψ.

$$\frac{\phi \quad (\phi \rightarrow \psi)}{\psi} \qquad \text{(modus ponens)}$$

In this system, a *proof* is a sequence of formulas

$$\phi_0,\ \phi_1,\ \phi_2,\ \cdots \phi_n$$

such that for each $i \leq n$, the formula ϕ_i is either

- an axiom; or
- obtained from two previous formulas ϕ_j and ϕ_k in the sequence via modus ponens (for some $j, k < i$).

If such a proof exists, then the final formula ϕ_n is called a *theorem* and we may write $\vdash \phi_n$.

[4]Using this system, checking the validity of a formula with n proposition symbols requires 2^n computations.

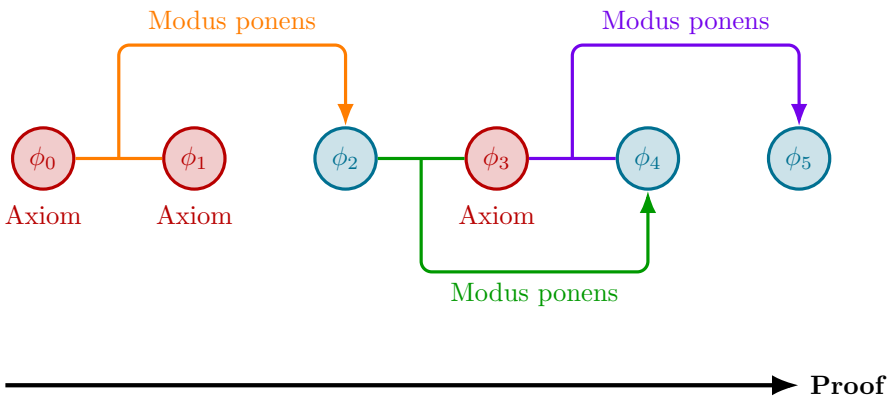

Figure 4: In a proof, every formula must be either an axiom, or derived from previous formulas via modus ponens.

For example, the theorem

$$\vdash (p \to p)$$

may be proved using the above proof system as follows.

1. $(p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))$ (Axiom I, replacing p, q, r by $p, (p \to p), p$)
2. $p \to ((p \to p) \to p)$ (Axiom II, replacing p, q by $p, (p \to p)$)
3. $(p \to (p \to p)) \to (p \to p)$ (modus ponens, via 1 and 2)
4. $p \to (p \to p)$ (Axiom I, replacing p, q by p, p)
5. $p \to p$ (modus ponens, via 3 and 4)

To include double negations and other connectives like $\wedge$ and $\vee$, we may add more axioms to our proof system.

IV. $p \to \neg\neg p$ and $\neg\neg p \to p$ (double negation)

V. $(p \vee q) \to (\neg p \to q)$ and $(\neg p \to q) \to (p \vee q)$ (implication as disjunction)

VI. $(p \wedge q) \to \neg(p \to \neg q)$ and $\neg(p \to \neg q) \to (p \wedge q)$ (implication as conjunction)

2.2 Proofs with assumptions and the principle of explosion

Let Γ be a set of *assumptions*, i.e. formulas that are assumed to be true. Under these assumptions, a proof is defined as a sequence of formulas

$$\phi_0,\ \phi_1,\ \phi_2,\ \cdots \phi_n$$

such that for each $i \leq n$, the formula ϕ_i is either

- an axiom;
- an assumption $\phi_i \in \Gamma$; or
- obtained from two previous formulas ϕ_j and ϕ_k in the sequence via modus ponens (for some $j, k < i$).

If such a proof exists, then we may write $\Gamma \vdash \phi_n$.

For example, given the set of assumptions $\Gamma = \{p\}$, we may prove that $q \to p$ using the Hilbert-style proof system, as demonstrated below.

1. $p \to (q \to p)$ (Axiom I)
2. p (Assumption)
3. $q \to p$ (modus ponens, via 1 and 2)

Proving with assumptions can be quite tricky due to the *principle of explosion*[5], which states that any statement can be proven from a contradiction. In other words, it is possible to prove any given statement, true or false, using a proof system as long as at least one of the assumptions in Γ is false.

We shall illustrate this principle as follows. Let Γ be the set containing the invalid assumption $\neg(q \to q)$. We will use the Hilbert-style proof system to prove an arbitrary formula p under this assumption.

5. $q \to q$ (proven previously)
6. $(q \to q) \to \neg\neg(q \to q)$ (Axiom IV, replacing p by q)
7. $\neg\neg(q \to q)$ (modus ponens, via 5 and 6)
8. $\neg\neg(q \to q) \to (\neg p \to \neg\neg(q \to q))$ (Axiom I, replacing p, q by $\neg\neg(q \to q), \neg p$)
9. $\neg p \to \neg\neg(q \to q)$ (modus ponens, via 7 and 8)
10. $(\neg p \to \neg\neg(q \to q)) \to (\neg(q \to q) \to p)$ (Axiom III, replacing p, q by $p, \neg\neg(q \to q)$)
11. $\neg(q \to q) \to p$ (modus ponens, via 9 and 10)
12. $\neg(q \to q)$ (assumption)
13. p (modus ponens, via 11 and 12)

2.3 Soundness, completeness and termination

A proof system is said to be *sound* if it can only prove valid theorems. In other words, anything proven using a sound system must be valid.

$$\underbrace{\vdash \phi}_{\text{proven}} \implies \underbrace{\models \phi}_{\text{valid}} \qquad \text{(soundness)}$$

Conversely, a proof system is said to be *complete* if it can prove any given valid theorem. In other words, if a formula is valid, it must be possible to prove it under a complete system.

$$\underbrace{\models \phi}_{\text{valid}} \implies \underbrace{\vdash \phi}_{\text{proven}} \qquad \text{(completeness)}$$

The main problem with the Hilbert-style proof system is that although it is relartively easy to check that a proof of a formula is correct, there is no systematic way for efficiently constructing proofs.

Moreover, even if a system is sound and complete, we don't know how long the proof for a given formula might be. Since it is impossible for us to check all the possibilities to see if a proof exists, testing the validity of a formula remains undecidable — there is no effective method for determining validity that terminates in finite time.

[5]This principle is sometimes referred to in Latin as *ex falso quodlibet*, which literally translates to "from falsehood, anything [follows]".

3 Propositional tableau

In view of the impracticality of Hilbert-style proof systems, we introduce below an easier and more implementable method for determining a formula's validity — *tableaus*.

Here is a brief overview of how a tableau works. Suppose we want to check the satisfiability of a formula ϕ. This formula will be placed at the root of a binary tree, called a tableau. We use a variety of expansion rules to grow the tree until it is complete. An *open* tableau indicates that ϕ is satisfiable, while a *closed* tableau indicates that ϕ is unsatisfiable.

To determine the validity of a formula, simply construct a tableau for $\neg\phi$. If the resultant tableau is open, then $\neg\phi$ is satisfiable, so ϕ is invalid. On the contrary, if the resultant tableau is closed, then $\neg\phi$ must be unsatisfiable, so ϕ is valid.

3.1 Constructing a tableau

In a tableau, every node is marked with a formula. To build a tableau for a formula ϕ, begin by placing ϕ at the root of a binary tree. Then, we repeat the following process:

1. Select a formula in the tree that has not been selected before. The formula must not be a literal.
2. Choose the expansion rule (see below) that applies to the selected formula.
3. For each leaf node, add new children nodes in accordance to the chosen expansion rule.
4. Place a tick beside the selected formula to make sure we don't expand it again.

There are two types of expansion rules:

- α-rules, which create one new child per leaf node; and
- β-rules, which create two new children per leaf node.

Figures 5 and 6 depict the α- and β rules respectively. Nodes that are newly created by each rule are highlighted in blue.

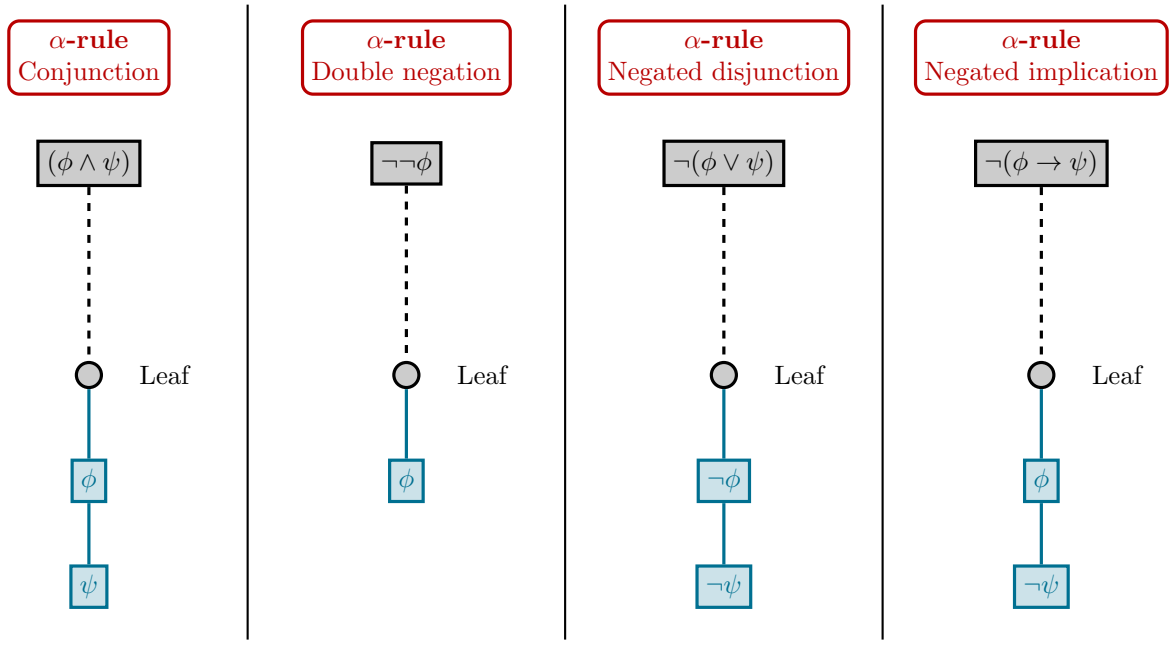

Figure 5: The four α-rules for constructing propositional tableaus.

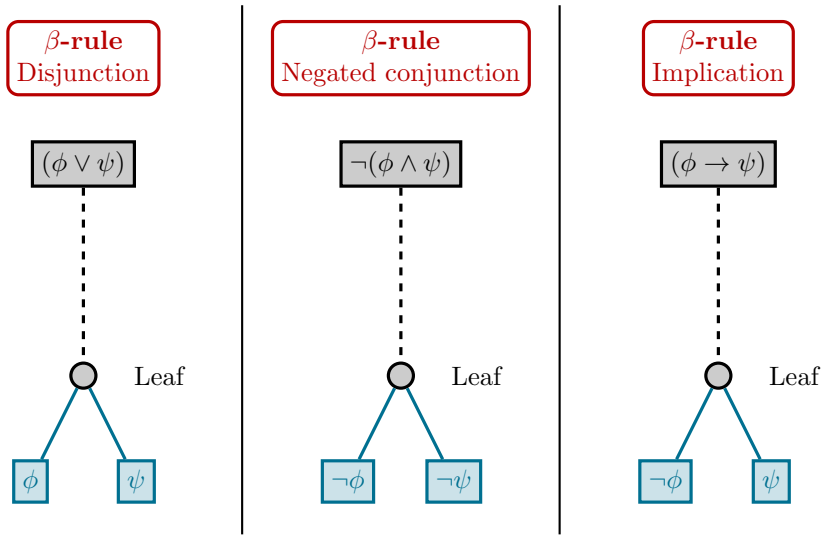

Figure 6: The three β-rules for constructing propositional tableaus.

In general, nodes located in the same branch[6] are considered in conjunction while the different branches are considered to be disjuncted. As a result, a tableau is a tree-like representation of a formula that is a disjunction of conjunctions, à la disjunctive normal form (DNF).

A tableau is considered *complete* if every node is either ticked (already expanded) or a literal. When a tableau is complete, we can determine the original formula's satisfiability as follows.

- A branch containing both a propositional letter and its negation (p and $\neg p$) is said to be *closed*, which we denote as $\oplus$. Otherwise, it is *open*.
- A tableau where all branches are closed is said to be *closed*, meaning that the formula at its root is unsatisfiable. Contrarily, a tableau with at least one open branch is said to be *open*, indicating that the formula is satisfiable.

3.2 Example of constructing a tableau and converting to DNF

To check if the formula

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

is satisfiable, we construct its tableau, as shown in figure 7.

Since only one of the four branches is closed, this formula is satisfiable. In fact, the literals in each open branch give a possible valuation that satisfies the given formula. For instance, the second branch from the left contains the literals p and $\neg q$. This indicates that the formula is true when p is true and q is false.

[6] A *branch* is defined as a path from the root of the tableau to one of its leaves.

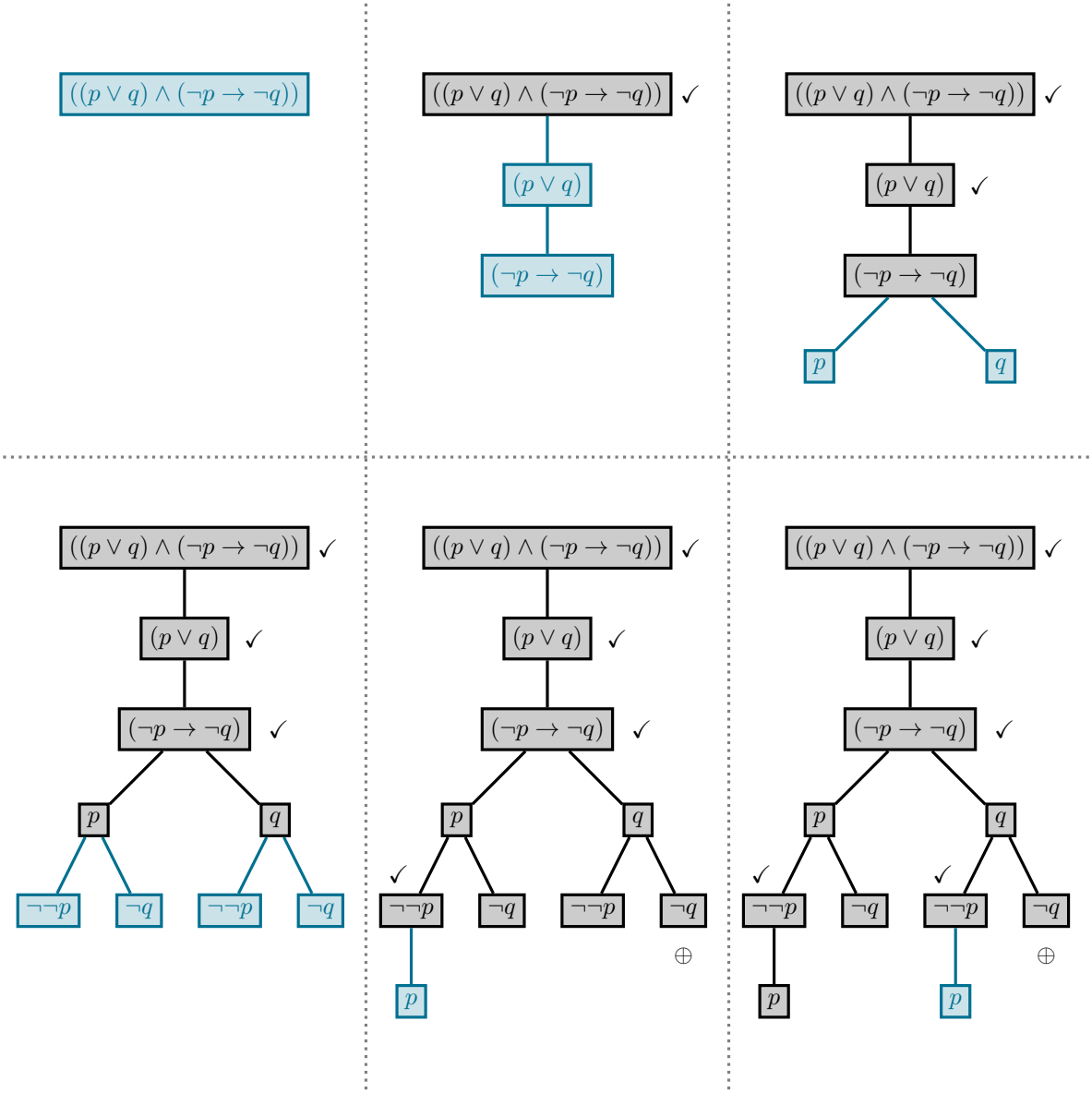

Figure 7: Constructing the tableau of $((p \vee q) \wedge (\neg p \to \neg q))$. Read from left to right and from top to bottom.

It follows that given the tableau of a formula, its DNF equivalent can be expressed as

$$\bigvee_{\text{open branch } \Theta} \left(\bigwedge \{\text{literals in } \Theta\} \right).$$

As always, the CNF of a formula can be obtained by negating the DNF form of its negation.

4 Predicate tableau

In first-order logic, a *literal* is an atom or its negation, i.e.

$$r^n(t_1, t_2, \cdots, t_n)$$

or

$$\neg r^n(t_1, t_2, \cdots, t_n)$$

where r^n is an n-ary predicate and t_i is a term.

The method for tableau construction in first-order logic is identical to that in propositional logic, but with a few extra expansion rules for dealing with quantifiers.

4.1 Expansion rules

In addition to α- and β-rules, we also require δ- and γ-rules, as depicted in Figures 8 and 9.

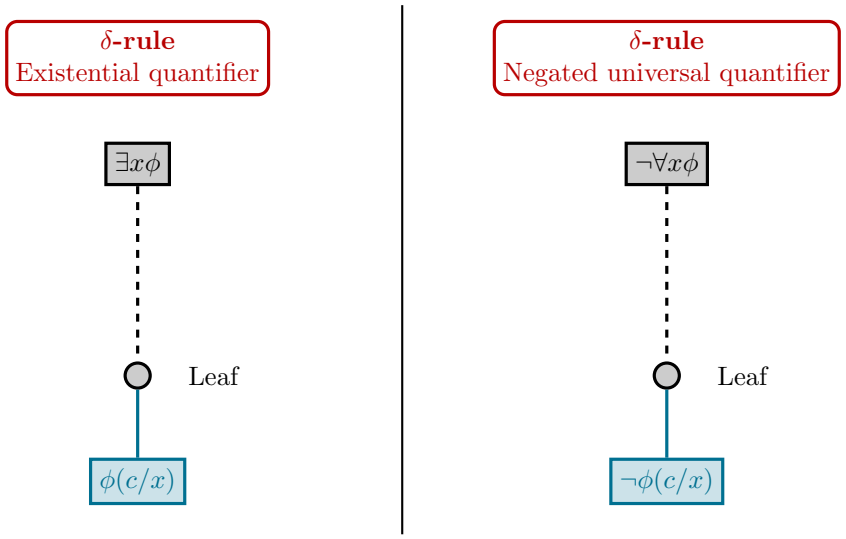

Figure 8: The two δ-rules for constructing predicate tableaus. In both rules, c should be a new constant that has not been used in the tableau before.

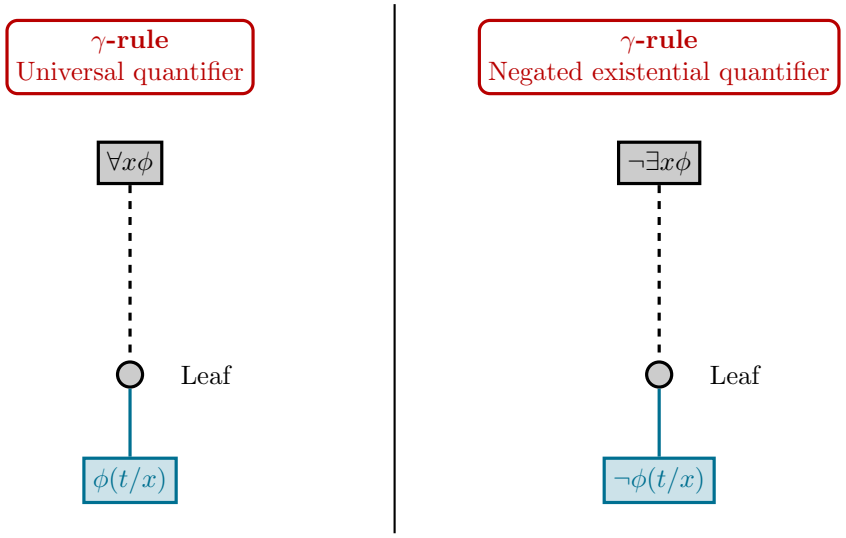

Figure 9: The two γ-rules for constructing predicate tableaus. In both rules, t is a closed term. Formulas should **not** be ticked following a γ-rule expansion.

When applying a δ-rule, make sure to introduce a new constant symbol that is not used anywhere before in the tableau. This new constant acts as a *witness*[7] for the existential statement.

Compared to the other rules, γ-rules are usually applied last. When applying a γ-rule, instantiate x with a closed term that appeared earlier in the current branch[8]. Formulas expanded via a γ-rule should **not** be ticked.

4.2 Termination

Similar to propositional tableaus, a predicate tableau's branch is closed if it contains both a literal $P(t_1, t_2, \cdots, t_n)$ and its negation $\neg P(t_1, t_2, \cdots, t_n)$. Otherwise, it is open.

The tableau terminates when:

- Every branch is closed. This shows that the root formula is unsatisfiable.
- All formulas are fully expanded and no further rules can be applied. If at least one branch remains open and cannot be further expanded, the tableau is open, indicating the root formula's satisfiability.

4.3 Example

Suppose we want to check whether the formula

$$(\forall x\ \neg p(x) \rightarrow \neg \exists y\ p(y))$$

is valid. To do this, we place its negation at the root of our tableau.

[7] Or: *Skolem witness*.
[8] This closed term should **not** be new.

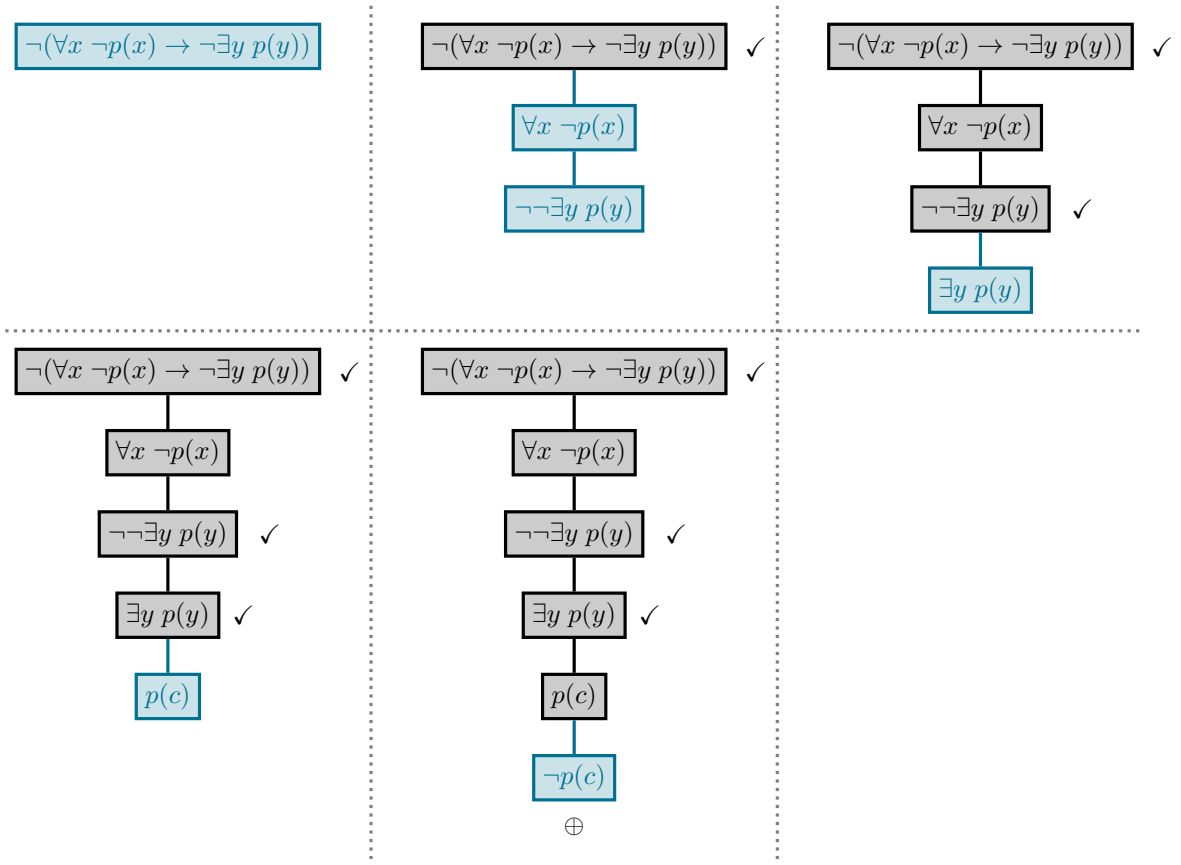

Figure 10: Constructing the tableau of $\neg(\forall x\ \neg p(x) \to \neg\exists y\ p(y))$. Read from left to right and from top to bottom. In the fourth step (bottom left), the existential formula $\exists y\ p(y)$ is expanded via a δ-rule by introducing a new constant c. In the last step (bottom middle), the universal formula $\forall x\ \neg p(x)$ is expanded via a γ-rule by replacing all bounded instances of x with the closed term c from earlier in the current branch, thereby producing both $p(c)$ and $\neg p(c)$ in the same branch. This results in a closed branch and hence a closed tableau, indicating that the formula at the root is unsatisfiable.

As shown, the negation $\neg(\forall x\ \neg p(x) \to \neg\exists y\ p(y))$ is unsatisfiable. This means that our original formula must be valid.

4.4 Non-termination

Predicate tableaus may not always terminate.

For instance, if a tableau unendingly generates nodes that require expansion via δ-rules, more and more constants would be introduced, and the number of γ-rule applications required would increase dramatically. This may result in non-termination.

Before we elaborate on this non-terminating scenario, we must note that in order to systematically handle possible infinite expansions, we should adopt a fair application strategy. A tableau construction is *fair* if

- Every formula that can still be expanded eventually will be, and
- Every formula that falls under a γ-rule will eventually be instantiated via that rule using all closed terms that appear in its branch.

This ensures that if the tableau can close, it will close after finitely many steps. We won't miss a contradiction because we ignored a rule.

Now, assuming a fair application strategy,

- If a branch keeps repeating the same configuration of formulas over and over with no new information, it is effectively saturated. This branch is then considered open, meaning that the root formula is satisfiable. This is because we may construct an infinite model for the root formula by reading off literals in the limit of the infinitely "looping" branch in the same way as we did for propositional tableaus. See Figure 11 for an example.

- If a branch runs indefinitely without closure, the satisfiability of the root formula is **undecided** and **inconclusive**.

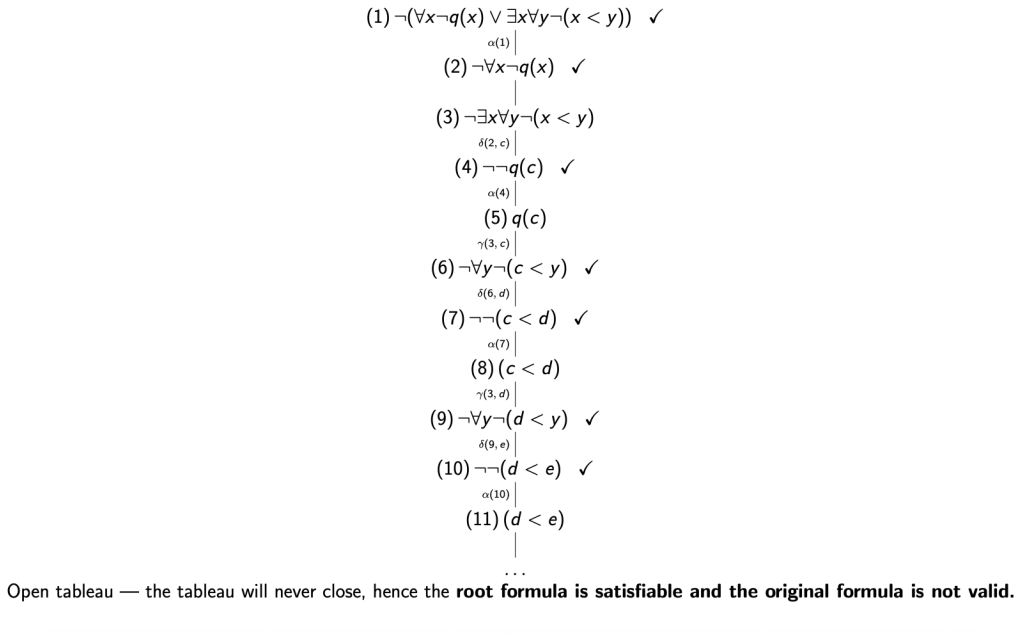

Figure 11: A non-terminating tableau where the root formula is satisfiable.