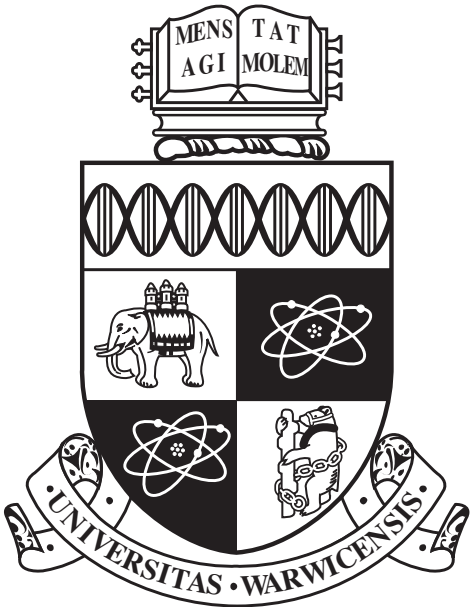
University of Warwick
Department of Computer Science

# CS262

## Logic and Verification




Cem Yilmaz
August 4, 2023

# Contents

# 1 Prolog

Prolog is programming in logic. A programming paradigm that is imperative/procedural, functional and objected-oriented. Its basic principles are recursion and induction.

## 1.1 Arithmetic

Prolog supports basic arithmetic operators, i.e.,

$$+, -, *, / \tag{1}$$

There is also basic support for functions such as

$$\sin(x), \cos(x), (x), sqrt(x), \max(X, Y) \tag{2}$$

There are also numerical operators such as

$$=:=, = / =, >, <, >=, =< \tag{3}$$

The $is/2$ predicate which assigns numerical value of right hand side to left hand side, and lastly the unification operator

$$= \tag{4}$$

bind free variables to make the match other members.

# 2 Propositional Formulas

**Definition 2.1.** Propositional formulas

Propositional Formulas are exactly those that can be constructed by a finite number of applications of the following rules:

1. Propositional variables $p, q, r, \ldots$ and $\top$ and $\bot$ are atomic formulas
2. If $X$ is a formula then so is $\neg X$
3. if $X$ and $Y$ are formulas then so is $(X \circ Y)$, where $\circ$ is a binary connective (e.g., $\wedge, \neg, \rightarrow, \ldots$ )

**Definition 2.2.** Parse Tree

We can conductively construct a parse tree for any propositional formula:

- For any atomic formula $X$, the parse tree has a single node labeled $X$.
- The parse tree for $\neg X$ has a node labeled $\neg$ as the root, and the parse tree for $X$ as the unique subtree of the root.
- The parse tree for $(X \circ Y)$ has a node labeled $\circ$ as the root, and the parse trees for $X$ and $Y$ as its left and right subtree.

**Example 2.1.** Parse Tree

Consider the statement and draw it as a parse tree

$$((p \wedge \neg q) \rightarrow (r \vee (q \rightarrow \neg p))) \tag{5}$$

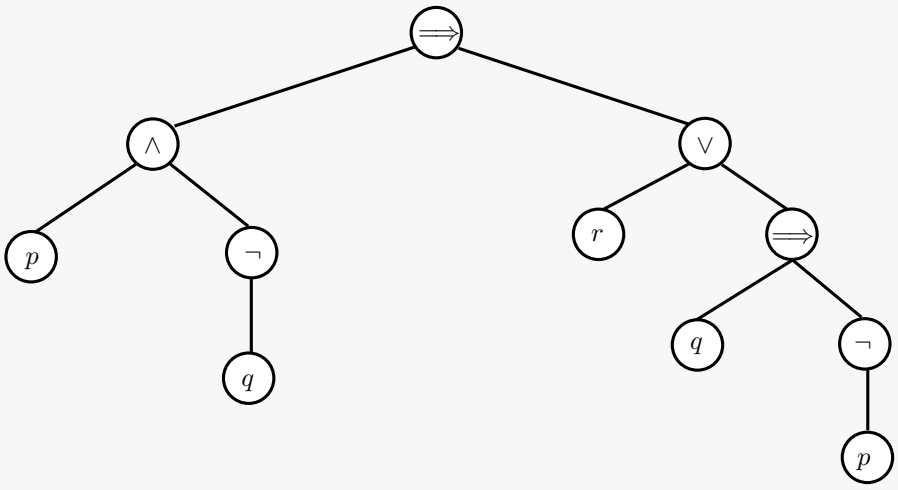



Figure 1: Parse Tree for the above statement

**Definition 2.3.** Recursion and Degrees

Based on our inductive definition, we can define recursive functions on formulas. They are defined as

$$deg(A) = 0 \qquad \text{if } A \text{ is an atomic formula} \tag{6}$$

$$deg(\neg X) = 1 + deg(X) \qquad \text{if } X \text{ is a formula} \tag{7}$$

$$deg(X \circ Y) = deg(X) + deg(Y) + 1 \qquad \text{if } X \text{ and } Y \text{ are formulas} \tag{8}$$

**Theorem 2.1.** *Degree and Inner Nodes*

The degree of a formula equals the number of inner nodes of the parse tree

*Proof.* Consider the base case of atomic formula $A$, for which $deg(A) = 0$. Its corresponding parse tree would be a single node of $A$. It has no children therefore it is a leaf and is not an inner node. Indeed, the fact that it is note an inner node and $deg(A) = 0$ we obtain that they're equal.
Consider the inductive step of two cases:

1. $\neg X$
2. $(X \circ Y)$

For the first case, we have that

$$deg(\neg X) = 1 + deg(X) \tag{9}$$

The corresponding parse tree would result in a parent node of $\neg$ and a child node of $X$. Therefore, by definition, we create a new inner node that corresponds to $+1$. The child would represent the $deg(X)$ as required.
For the second case, we have that

$$deg(X \circ Y) = 1 + deg(X) + deg(Y) \tag{10}$$

The corresponding parse tree would result in a parent node of $\circ$ and two child nodes of $X$ and $Y$. Therefore, by definition, we create a new inner node which corresponds to the $+1$. The two children would correspond to the $deg(X)$ and $deg(Y)$ respectively. □

**Definition 2.4.** Valuation

A valuation is a mapping $v$ from the set of propositional formulas to the set of truth values $\{T, F\}$ satisfying the following conditions:

$$v(\top) = T;\ v(\bot) = F \tag{11}$$
$$v(\neg X) = \neg v(X) \tag{12}$$
$$v(X \circ Y) = v(X) \circ v(Y) \tag{13}$$

To examine all interesting valuations for a given formula, it is enough to specify $v$ for each variable. The value of all subformulas can then be deduced bottom-up (in the parse tree)

**Definition 2.5.** Tautology

A formula that evaluates to $T$ under all valuations is called a tautology

**Definition 2.6.** Contradiction

A formula that evaluates to $F$ under all valuations is called a contradiction

**Definition 2.7.** Satisfiable

A formula that evaluates to $T$ for some valuation is called satisfiable

**Definition 2.8.** Propositional Consequence

We say that a formula $X$ is a consequence of a set $S$ of formulas, denoted by $S \vDash X$, provided that $X$ maps to $T$ under every valuation that maps every member of $S$ to $T$. For example, $X$ is a tautology if and only if $\varnothing \vDash X$

**Example 2.2.** Propositional Consequence

Let the formula $A = (p \vee v) \wedge (\neg q \vee \neg r)$. Let the formula $U = \{p, \neg q\}$. Consider the question $U \vDash A$. It follows that it is a logical consequence. We have two possibilities:

Table 1: Possibilities

| | | |
|---|---|---|
| $p$ | $T$ | $T$ |
| $q$ | $F$ | $F$ |
| $r$ | $T$ | $F$ |

‘ Therefore it is true. Other examples include

$$p \vee q \vDash q$$

is not true as for $q = F$ and $P = T$ we get $T \vDash F$. Another example is

$$\{p, p \to q\} \vDash q$$

is also true as we only care when both $p$ and $p \to q$ are true. We find that $q$ is also true when these are true.

**Theorem 2.2.** *Connection of consequence and implication*

$$U \vDash A \text{ iff } \bigwedge_{i \in U} A_i \to A \tag{14}$$

Where $U = \{A_1, A_2, \ldots, A_i\}$

**Definition 2.9.** Logical Equivalence

Formulas are truth functions: each valuation maps the formula to $T$ or $F$. Formulas representing the same truth function are called logically equivalent. For example,

$$p \to q = \neg p \vee q \tag{15}$$

**Definition 2.10.** Completeness

A set of connectives is said to be complete if we can represent every truth function $\{T, F\}^n \to \{T, F\}$ using only these connectives.

**Definition 2.11.** Proof By Construction

To create a formula that is logically equivalent to any function $f$ on variables $x_1, \ldots x_n$ given by its truth table, we do the following:

1. For every valuation which maps to $T$ construct the conjunction $L_1 \wedge \ldots \wedge L_n$ where $L_j$ is $x_j$ if $x_j$ is assigned $T$ under the valuation and $L_j$ is $\neg x_j$ if $x_j$ is assigned $F$ under the valuation
2. Take the disjunction of all conjunctions from the previous step
3. For the function that is everywhere $F$, by convention take $\perp$

This shows that $\{\neg \wedge, \vee\}$ is complete. The resulting formula is called the disjunctive normal form (DNF) of $f$.

**Example 2.3.** DNF Example

Consider the table

Table 2: DNF Example

| $p$ | $q$ | $r$ | $f$ |
|---|---|---|---|
| $T$ | $T$ | $T$ | $T$ |
| $T$ | $T$ | $F$ | $T$ |
| $T$ | $F$ | $T$ | $F$ |
| $T$ | $F$ | $F$ | $F$ |
| $F$ | $T$ | $T$ | $F$ |
| $F$ | $T$ | $F$ | $F$ |
| $F$ | $F$ | $T$ | $F$ |
| $F$ | $F$ | $F$ | $T$ |

We now consider only rows that have $f$ as $T$, that is, the first 2 rows and the last row. We express it as

$$L_1 = p \wedge q \wedge r$$
$$L_2 = p \wedge q \wedge \neg r$$
$$L_8 = \neg p \wedge \neg q \wedge \neg r$$

We then take the disjunction of our $L$, that is

$$L_1 \vee L_2 \vee L_8$$

**Definition 2.12.** Normal Forms

A formula in disjunctive normal form (DNF) is a disjunction of conjunction of literals.
A formula in conjunctive norma form (CNF) is a conjunction of disjunctions of literals.

**Theorem 2.3.** *Normal Form Presence*

Every boolean function has a CNF and a DNF. Furthermore, we can write the generalised disjunction as

$$[X_1, X_2, \ldots X_n] := X_1 \vee X_2 \vee \ldots \vee X_n \tag{16}$$
$$\langle X_1, X_2, \ldots X_n \rangle := X_1 \wedge X_2 \wedge \ldots \wedge X_n \tag{17}$$

**Definition 2.13.** alpha and beta formulas

We group all propositional formulas of the forms $(X \circ Y)$ and $\neg (X \circ Y)$ where $\circ$ is a binary operator into two categories, those that act conjunctively, and those that act disjunctively:

Table 3: alpha and beta formulas

| $\alpha$ | $\alpha_1$ | $\alpha_2$ | $\beta$ | $\beta_1$ | $\beta_2$ |
|---|---|---|---|---|---|
| $X \wedge Y$ | $X$ | $Y$ | $\neg (X \wedge Y)$ | $\neg X$ | $\neg Y$ |
| $\neg (X \vee Y)$ | $\neg X$ | $\neg Y$ | $X \vee Y$ | $X$ | $Y$ |
| $\neg (X \rightarrow Y)$ | $X$ | $\neg Y$ | $X \rightarrow Y$ | $\neg X$ | $\neg Y$ |
| $\neg (X \leftarrow Y)$ | $\neg X$ | $Y$ | $X \leftarrow Y$ | $X$ | $\neg Y$ |
| $\neg (X \uparrow Y)$ | $X$ | $Y$ | $X \uparrow Y$ | $\neg X$ | $\neg Y$ |
| $X \downarrow Y$ | $\neg X$ | $\neg Y$ | $\neg (X \downarrow Y)$ | $X$ | $Y$ |
| $X \not\rightarrow Y$ | $X$ | $\neg Y$ | $\neg (X \not\rightarrow Y)$ | $\neg X$ | $Y$ |
| $X \not\leftarrow Y$ | $\neg X$ | $Y$ | $\neg (X \not\leftarrow Y)$ | $X$ | $\neg Y$ |

Note that $\alpha$ formulas are used during a conjunctive $(\wedge)$ and $\beta$ formulas are used during a disjunctive $(\vee)$.

**Theorem 2.4.** *Conjunctive Normal Form Algorithm*

Given any propositional formula $X$, start with $\langle [X] \rangle$. Given the current expansion $\langle D_1, \ldots, D_n \rangle$ select one of the disjunctions $D_i$ that is not a disjunction of literals. Select a non-literal $N$ from $D_i$.

- If $N$ is a $\beta-$formula, then replace $N$ by the two formula sequence $\beta_1$ and $\beta_2$.
- If $N$ is an $\alpha-$formula, then replace the disjunction $D_i$ with two disjunctions, one in which $N$ is replaced by $\alpha_1$, and one in which $N$ is replaced by $\alpha_2$.

**Example 2.4.** CNF expansion algorithm

Consider the statement

$$\neg (p \wedge \neg \bot) \vee \neg (\top \uparrow q) \tag{18}$$

We now follow the algorithm. We have

$$\begin{aligned} \langle [\neg (p \wedge \neg \bot) \vee \neg (\top \uparrow q)] \rangle = & \quad (19) \\ = \langle [\neg (p \wedge \neg \bot)), \neg (\top \uparrow q)] \rangle & \quad (20) \\ = \langle [\neg p, \bot, \neg (\top \uparrow q) \rangle] & \quad (21) \\ = \langle [\neg p, \bot, \top], [\neg p, \bot, q] \rangle & \quad (22) \end{aligned}$$

The algorithm stops here, however, this form could be greatly simplified.

**Theorem 2.5.** *Terminal of Alorithm*

The algorithm terminates, regardless of which choices are made during the algorithm.
Consider a rooted tree. A branch is a sequence of nodes starting at the root, descending towards one of the children in each step (unless no children are present). We say that the tree is finitely branching if every node has only finitely many children.

**Lemma 2.6.** *Konig's Lemma*

A tree that is finitely branching but infinite must have an infinite branch

*Proof.* Define the rank of a propositional formula as follows:
Recursion anchor: $r(p) = r(\neg p) = 0$ for variables $p$ ; $r(\top) = r(\bot) = 0$ and $r(\neg \top) = r(\neg \bot) = 1$. The recursive step is as follows:

$$\begin{aligned} r(\neg \neg Z) &= r(Z) + 1 & (23) \\ r(\alpha) &= r(\alpha_1) + r(\alpha_2) + 1 & (24) \\ r(\beta) &= r(\beta_1) + r(\beta_2) + 1 & (25) \end{aligned}$$

For a generalised disjunction we have

$$r([X_1, \ldots, X_n]) = \sum_{i=1}^{n} r(X_i) \tag{26}$$

We assign the current conjunction of disjunctions $\langle D_1, \ldots, D_n \rangle$ a sequence of $n$ balls, by placing one ball labelled $r(D_i)$ for each disjunction $D_i$ into a box. We argue that each step of the algorithm corresponds to removing one ball from the box, and replacing it either by two balls with a lower number ($\alpha-$expansion) or by one ball with a lower number (in all other cases). □

**Theorem 2.7.** *Disjunctive Normal Form Algorithm*

Given any propositional formula $X$, start with $[\langle X \rangle]$. Given the current expansion $[D_1, \ldots, D_n]$ select one of the disjunctions $D_i$ that is not a disjunction of literals. Select a non-literal $N$ from $D_i$.

- If $N$ is a $\alpha-$formula, then replace $N$ by the two formula sequence $\alpha_1$ and $\alpha_2$.
- If $N$ is an $\beta-$formula, then replace the disjunction $D_i$ with two disjunctions, one in which $N$ is replaced by $\beta_1$, and one in which $N$ is replaced by $\beta_2$.

We will omit the example as it is similar to the CNF.

# 3 Proof Systems

**Definition 3.1.** Semantic Tableau

The proof system takes the form of a tree, with nodes labelled by propositional formulas. We think of each branch as a conjunction of the formulas on that branch and think of the tree as the disjunction of all of its branches, in other words, a DNF. The expansion is as follows:

- If $N = \neg\top$, then extend the branch by a node labelled $\bot$ at its end
- If $N = \neg\top$, then extend the branch by a node labelled $\top$ at its end
- If $N = \neg\neg Z$, then extend the branch by a node labelled $Z$ at its end
- If $N$ is an $\alpha-$formula, then extend the branch by two nodes labelled $\alpha_1$,$\alpha_2$ ($\alpha-$expansion) at its end.
- If $N$ is a $\beta-$formula, then add a left and right child to the final node of the branch, and label one of them $\beta_1$ and the other one $\beta_2$ ($\beta-$expansion)

A branch of tableau is closed if both $X$ and $\neg X$ occur on the branch for some formula $X$. A branch can also be closed if $\bot$ appears. A tableau is closed if every branch is also closed. If you want to prove that $X$ is a tautology, you want to prove that there is a closed tableau for $\neg X$. We write

$$\vdash_t X \tag{27}$$

if $X$ has a tableau proof. Lastly, a tableau is strict, if no formula has had an expansion rule applied to it twice on the same branch.

**Example 3.1.** Semantic Tableau Example

Consider

$$\neg((p \to (q \to r)) \to ((p \vee s) \to ((q \to r) \vee s))) \tag{28}$$

We initially have $\alpha$ rule where

$$\alpha_1 = p \to (q \to r) \tag{29}$$
$$\alpha_2 = \neg((p \vee s) \to ((q \to r) \vee s)) \tag{30}$$

We now find that there is a $\beta$ rule in $\alpha_1$.

$$\beta_1 = \neg p \tag{31}$$
$$\beta_2 = q \to r \tag{32}$$

Followed by another $\beta$ rule on $\beta_2$ :

$$\beta_3 = \neg q \tag{33}$$
$$\beta_4 = r \tag{34}$$

We now require to expand $\alpha_2$. However, we have lots of branches. Specifically, we have branches $\beta_1, \beta_3, \beta_4$. We will blindly decide to do the $\alpha_2$ expansion on $\beta_1$ branch.

$$\alpha_1 = p \vee s \tag{35}$$
$$\alpha_2 = \neg((q \to r) \vee s) \tag{36}$$

The rest of the example is left as exercise.

**Theorem 3.1.** *Soundness and Completeness*

The tableau system is sound, i.e., if $X$ has a tableau proof, then $X$ is a tautology. Furthermore, the tableau proof system is complete, i.e., if $X$ is a tautology, then the strict tableau system will terminate with a proof for it. Equivalent,

$$\vdash_t X \text{ iff } \vDash X \tag{37}$$

First theorem follows from correctness proof of our DNF expansion algorithm given before.

**Definition 3.2.** Resolution Proof

If the semantic tableau proof is the one that mirrors a DNF, then the resolution proof mirrors a conjunctive normal form. In particular, we have the following rules

- If $N = \neg\top$, then append a new disjunction where $N$ is replaced by $\bot$
- If $N = \neg\bot$, then append a new disjunction where $N$ is replaced by $\top$
- If $N = \neg\neg Z$, then append a new disjunction where $N$ is replaced by $Z$
- If $N$ is an $\alpha$ formula, then append two new disjunctions, one in which $N$ is replaced by $\alpha_1$, and one in which it is replaced by $\alpha_2$
- If $N$ is a $\beta$ formula, then append a new disjunction where $N$ is replaced by $\beta_1, \beta_2$

A sequence of resolution expansion applications is strict, if every disjunction has at most one resolution expansion rule applied to it. We now need a resolution rule stated as follows:
Suppose $D_1$ and $D_2$ are two disjunctions, with $X$ occurring in $D_1$ and $\neg X$ in $D_2$. Let $D$ be the result of the following:

- Delete all occurrences of $X$ from $D_1$
- Delete all occurrences of $\neg X$ from $D_2$
- Combine the resulting disjunctions

There is a special case that contains $\bot$. In such case, delete $\bot$ and call the resulting disjunction the trivial resolvent. I.e.,

$$\langle [X, Y], [\neg X, Z] \rangle = \langle [X, Y], [\neg X, Z], [Y, Z] \rangle \tag{38}$$

A resolution expansion is closed if it contains the empty clause $[]$. A resolution proof for $X$ is a closed resolution expansion for $\neg X$. We write

$$\vdash_r X \tag{39}$$

If $X$ has a resolution proof.

**Example 3.2.** Resolution Proof Example

Resolution proof for $((p \wedge q) \vee (r \to s)) \to ((p \vee (r \to s)) \wedge (q \vee (r \to s)))$:

1. $[\neg(((p \wedge q) \vee (r \to s)) \to ((p \vee (r \to s)) \wedge (q \vee (r \to s))))]$
2. $[(p \wedge q) \vee (r \to s)]$
3. $[\neg((p \vee (r \to s)) \wedge (q \vee (r \to s)))]$
4. $[p \wedge q, r \to s]$
5. $[\neg(p \vee (r \to s)), \neg(q \vee (r \to s))]$
6. $[p, r \to s]$
7. $[q, r \to s]$
8. $[\neg p, \neg(q \vee (r \to s))]$
9. $[\neg(r \to s), \neg(q \vee (r \to s))]$
10. $[\neg p, \neg q]$
11. $[\neg p, \neg(r \to s)]$
12. $[\neg(r \to s), \neg q]$
13. $[\neg(r \to s), \neg(r \to s)]$
14. $[r \to s, \neg p]$
15. $[r \to s, r \to s]$
16. $[]$

Figure 2: Resolution Proof Example

**Theorem 3.2.** *Soundness and Completeness*

The resolution proof system is sound, i.e., if $X$ has a resolution proof, then $X$ is a tautology.
The resolution proof system is also complete, i.e., if $X$ is a tautology, then the resolution system will terminate with a proof for it, even if all resolution rule applications are atomic or trivial, and come after resolution expansion steps. Equivalently

$$\vdash_r X \text{ iff } \vDash X \tag{40}$$

Similarly, our first theorem is an implication from the correctness proof of our CNF expansion algorithm given before. The resolution rule produces a semantically equivalent formula.

**Definition 3.3.** S-introduction rule

The $S$-introduction rule for tableau says that for any formula $Y \in S$ can be added to the end of any tableau branch. We write $\vdash_t X$ if there is a closed tableau for $\neg X$ allowing the $S$-introduction rule for tableau.

The $S$-introduction rule for resolution states that for any formula $Y \in S$, the line $[Y]$ can be added as a line to a resolution expansion. We write $S \vdash_r X$ if there is a closed resolution expansion for $\neg X$, allowing the $S$-introduction rule for resolution.

**Theorem 3.3.** *Strong soundness and completeness*

For any set $S$ of propositional formulas and any formula $X$, we have

$$S \vDash X \iff S \vdash_t X \iff S \vdash_r X \tag{41}$$

**Example 3.3.** S introduction Example

Prove $\{p \to q, q \to r\} \vDash \neg(\wedge p)$ via tableau and resolution.

*Proof.* We begin our proof with tableau first.

$$\neg\neg(\neg r \wedge p) \tag{42}$$
$$p \to q \text{(note that we can add the } S \text{ rules whenever we want)} \tag{43}$$
$$q \to r \tag{44}$$
$$\neg r \wedge p \tag{45}$$
$$\neg r \tag{46}$$
$$p \tag{47}$$
$$\tag{48}$$

We now create a new branch below $p$ by expanding $p \to q$

$$\neg p, q \text{Note that } p \text{ is now closed.} \tag{49}$$

] And finally we expand $q \to r$ under $q$

$$\neg q, r \tag{50}$$

And both branches close since we have $q$ and $\neg r$ above them. We now proof using the resolution proof.

$$[\neg\neg(\neg r \wedge p)] \tag{51}$$
$$[p \to q] \tag{52}$$
$$[q \to r] \tag{53}$$
$$[\neg r \wedge p] \tag{54}$$
$$[\neg p, q] \tag{55}$$
$$[\neg q, r] \tag{56}$$
$$[\neg r] \tag{57}$$
$$[p] \tag{58}$$

Let us combine 55 and 58. We obtain

$$[q] \tag{59}$$

Then we resolution 56 and 57:

$$[\neg q] \tag{60}$$

With the final resolution of 59 and 60

$$[] \tag{61}$$

□

**Definition 3.4.** Natural Deduction

Natural deduction formalizes the kind of reasoning people do in informal arguments. Unlike tableau and resolution, natural deduction is not well-suited for computer automation. It is also not based on CNF or DNF.
Implication rule. If one can derive $Y$ from $X$ as an assumption, then one can discharge the assumption and conclude that $X \to Y$ holds unconditionally. I.e.,

$$\boxed{\begin{array}{c} X \\ \vdots \\ Y \end{array}}$$

means $X \to Y$. Subordinate proofs/lemmas are contained in boxes. The first formula $X$ is an assumption.

- Modus Ponens Rule: From $X$ and $X \to Y$, we can conclude $Y$.
- Modus Tollens Rule: From $\neg Y$ and $X \to Y$, you can conclude $\neg Y$.
- Constant Rule: $\bot$ we can conclude any formula $X$, a contradiction. Similarly, we can always add $\top$ as it is always true.
- Negation Rules: If you have $X$ and $\neg X$, you can conclude $\bot$. Furthermore, if you have $X$ and $\bot$ in the same box, you can conclude $\neg X$.
- $\alpha E$: if you have an $\alpha$ rule, you can conclude $\alpha_1$ and/or $\alpha_2$ as you wish.
- $\alpha I$ : if you have $\alpha_1$ and $\alpha_2$, you can conclude $\alpha$
- $\beta E$ : if you have $\neg\beta_1$ and $\beta$, you can conclude $\beta_2$. This can also be interchanged with $\beta$ numbers.
- $\beta I$: If you have $\neg\beta_1$ and $\beta 2$ in the same box, you can conclude $\beta$ out of the box. This rule can be interchanged with $\beta$ numbers.

A formula is called active at some stage if it does not occur in a closed box. The $E$ and $I$ are $Elimination$ and $Introduction$ respectively. There are also a set of derived rules, as follows:

- Double negation: If you have $\neg\neg X$, you can conclude $X$. Similarly, if you have $X$ you can conclude $\neg\neg X$
- Copy rule: If you have $X$, you can conclude $X$ later down.

A good strategy for this proof system is to think backwards. What rules could be applied in the last step, and based on that come up with assumptions that should be made to apply those rules. Furthermore, in proving $X$ you can assume $\neg X$ and produce $\bot$, then use negation rule. Lecture 14 has a good example.

**Definition 3.5.** S Introduction

$S-$ introduction rule for natural deduction is that at any stage, any member of $S$ may be used as a line. We write

$$S \vdash_d X \tag{62}$$

If there is a natural deduction derivation of $X$ from $S$.

**Theorem 3.4.** *Soundness and Completeness*

We have

$$S \vDash X \iff S \vdash_d X \tag{63}$$

**Example 3.4.** Natural Deduction Example

Find the natural deduction proof of

$$(p \to (q \to r)) \to (q \to (p \to r)) \tag{64}$$

To begin this proof, we must create an assumption

$$p \to (q \to r) \text{ ass.} \tag{65}$$

$$q \text{ ass.} \tag{66}$$

$$p \text{ ass.} \tag{67}$$
$$q \to r \text{modus pon.} \tag{68}$$
$$r \tag{69}$$

$$p \to r \text{ impl.} \tag{70}$$

$$q \to (p \to r) \text{ impl.} \tag{71}$$

# 4 Boolean Satisfiability Problem

**Definition 4.1.** Problem SAT

Given a propositional formula in conjunctive normal form, is there a satisfying assignment for it?

**Definition 4.2.** Literal

A literal is a variable or a negated variable

**Definition 4.3.** CNF Formula

A formula in conjunctive normal form

**Definition 4.4.** k-CNF Formula

every clause has at most $k$ literals.

**Definition 4.5.** k-SAT Problem

Input of SAT is $k$-CNF

**Theorem 4.1.** *Computational Complexity*

We can solve the problem SAT in time $2^n \times L$ by computing the entire truth table, where $L$ is the total number of literals of the input formula, and $n$ is the number of variables. This problem is the mother of all NP-complete problems.

**Theorem 4.2.** *Satisfiability Iff*

There is a polynomial time algorithm that, given an instance $\mathcal{F}$ of SAT, produces a 3-CNF $\mathcal{G}(\mathcal{F})$ such that $\mathcal{F}$ is satisfiable if and only if $\mathcal{G}(\mathcal{F})$ is satisfiable

*Proof.* Consider a clause containing two literals $X$ and $Y$. Replace $X \vee Y$ in the clause by a new variable $Z$ (so the clause gets shorter by one literal) and express $X \vee Y \equiv Z$ by a 3-CNF $H(X, Y, Z) : [\ldots]$ □

**Definition 4.6.** Reduction

Problem Colouring. Given a graph $G = (V, E)$ and integer $k$, can its vertices be coloured properly with $k$ colours?
A colouring of $G$ is proper if the end vertices of every edge receive distinct colours.

**Theorem 4.3.** *Coloring to SAT*

There is a polynomial time algorithm that gives an instance $(G, k)$ of COLOURING, produces a CNF formula $\mathcal{F}(G, k)$ such that $G$ has a proper colouring with $k$ colours if and only if $\mathcal{F}(G, k)$ is satisfiable.
In other words, if we can solve SAT efficiently, we can solve COLOURING efficiently. If we can count the number of satisfying assignments of a CNF formula efficiently, then we can count proper $k$-colourings efficiently.

*Proof.* For each vertex $v$ and each colour $k$, introduces a variable

$$x_{v,k} = \begin{cases} 1 & \text{if vertex } v \text{ receives colour } k \\ 0 & \text{else} \end{cases} \tag{72}$$

We add constraints that every vertex receives exactly one colour. We then add constrains that the end vertices of every edge receive two distinct colours. We have $|V| \cdot k$ variables and the size of the formula we have polynomial in $|V|$ and $|E|$. □

**Definition 4.7.** Resolution

We will solve 2-SAT in polynomial time. Recall the resolution proof method, we maintain a conjunction of disjunctions. Our first step is resolution expansion into a CNF and the second step is the resolution rule. The key is observation is that the resolution clause of size 2 produces another clause of size $\leq 2$. At most $1 + 2n + 4\binom{n}{2} = 2n^2 + 1$ clauses ever occur in the process. If formula is satisfiable, then the empty clause [] will occur, otherwise it is satisfiable.

**Definition 4.8.** Linear-time algorithm for 2 SAT

Recall that

$$u \vee v \equiv \neg u \to v \equiv \neg v \to u \tag{73}$$

The idea is to capture this implication information of a 2-CNF $\mathcal{F}$ in a directed graph $D = D(\mathcal{F})$ :

- Vertex set $V(D) := V \cup \overline{V}$, i.e., all variables $V = V(\mathcal{F})$ and their negation
- Edge set $E(D) := \{(\neg u, v), (\neg v, u) | [u \vee v] \in \mathcal{F}\} \cup \{(\neg u, u) | [u] \in \mathcal{F}\}$

2-clause lead to two directed edges, whereas a unit clause leads to a single directed edge. The graph $D$ has $2n$ vertices, where $n := |V|$, and at most $2m$ edges, where $m := |\mathcal{F}|$

**Example 4.1.** Linear-time algorithm for 2 SAT

Consider

$$\mathcal{F} = \langle [\neg x_1, x_2], [\neg x_2, x_3], [\neg x_3, x_4], [\neg x_4, x_1], [\neg x_5, x_3], [x_5, x_1] \rangle \tag{74}$$

Using algorithm we create the edges for each clause:

- $[\neg x_1, x_2] = x_1 \to x_2, \neg x_2 \to \neg x_1$
- $[\neg x_2, x_3] = x_2 \to x_3, \neg x_3 \to \neg x_2$
- $[\neg x_3, x_4] = x_3 \to x_4, \neg x_4 \to \neg x_3$
- $[\neg x_4, x_1] = x_4 \to x_1, \neg x_1 \to \neg x_4$
- $[\neg x_5, x_3] = x_5 \to x_3, \neg x_3 \to \neg x_5$
- $[x_5, x_1] = \neg x_5 \to x_1, \neg x_1 \to x_5$

**Definition 4.9.** Directed Path

Write

$$x \rightsquigarrow y \tag{75}$$

If there is a directed path from $x$ to $y$ in the graph $D(\mathcal{F})$.

**Lemma 4.4.** *Satisfiability for linear time*

$\mathcal{F}$ is not satisfiable if and only if there is a variable $x \in V$ such that $x \rightsquigarrow \neg x \rightsquigarrow x$

**Definition 4.10.** Strongly Connected

Two vertices $u$ and $v$ in a directed graph are strongly connected if $u \rightsquigarrow v$ and $v \rightsquigarrow u$. The strongly connected components are the maximal subsets of vertices with this property.

*Proof.* Let $\mathcal{F}'$ denote the CNF obtained from $\mathcal{F}$ by exhaustively applying resolution rule. Consider the resolvent $[u, v]$ of $[x, u]$ and $[\neg x, v]$. This resolvent would add two new edges to the graph, namely $\neg u \to v$ and $\neg v \to u$. But the edges $\neg u \to x \to v$ and $\neg v \to \neg x \to u$ were already present. So adding these edges does not alter the relation $\rightsquigarrow$. Therefore, $\mathcal{F}$ is not satisfiable if and only if $[]$ appears in $\mathcal{F}'$ if and only if $[x], [\neg x]$ appears in $\mathcal{F}'$ for some variable $x \in V$ if and only if $x \to \neg x \to x$ in $D(\mathcal{F}')$ if and only if $x \rightsquigarrow \neg x \rightsquigarrow x$. □

**Definition 4.11.** Applications to 3-SAT

We want to use our polynomial-time algorithms for solving 2-SAT for solving 3-SAT more efficiently, i.e., less than $2^n$ steps ($n$ is the number of variables). We consider the subset $\mathcal{G}$ of the independent definition of maximal size i.e., no further clauses can be obtained without violating independence.

**Definition 4.12.** Independency

Given a 3-CNF $\mathcal{F}$ over $n$ variables. A subset $\mathcal{G}$ of clauses of $\mathcal{F}$ is called independent if no two clases share any variables.

**Lemma 4.5.** *Maximal Set and 3-SAT*

Consider a maximal set $\mathcal{G}$ of independent 3-clause in $\mathcal{F}$. Then we have

- $|\mathcal{G}| \leq \frac{n}{3}$
- For any truth assignment $\alpha$ to the variables in $\mathcal{G}$, $\mathcal{F}^{[\alpha]}$ is a 2-CNF. Here, $\mathcal{F}^{[\alpha]}$ is the formula obtained from $\mathcal{F}$ by setting all variables defined by $\alpha$ to $T$ or $F$ (remove clauses with $T$ literals and remove $F$ literals from clauses)
- The number of truth assignments satisfying $\mathcal{G}$ is $7^{|\mathcal{G}|} \leq 7^{\frac{n}{3}}$

**Definition 4.13.** 3-SAT Algorithm

Go through all truth assignments $\alpha$ satisfying $\mathcal{G}$, and check satisfiability of the 2-CNF $\mathcal{F}^{[\alpha]}$ in polynomial time (quadratic or linear)

**Theorem 4.6.** *3-CNF Runtime*

Satisfiability of a 3-CNF formula can be decided in time

$$O(7^{\frac{n}{3}}\text{poly}(n)) = O(1.913^n) \tag{76}$$

**Definition 4.14.** Horn Satisfiability

We sat that 2-SAT is a special case of SAT that can be solved efficiently. We now consider another special case, where there is no restriction on the size of clauses, but on their structure.

**Definition 4.15.** Horn Clause

A horn clause is a clause in which there is at most one positive literal (non-negated value)

**Definition 4.16.** Horn CNF

A horn CNF is a CNF that only has Horn clauses

**Definition 4.17.** Satisfiability of Horn CNF

If every clause has size $\geq 2$, then the formula can be satisfied by setting all variables to $F$. If the formula has a clause of size 1, then we have to set the literal in this clause to $T$ to satisfy the formula. If the formula contains the empty clause $[]$, then the formula is not satisfiable. If $\mathcal{F}$ is a Horn CNF, then $\mathcal{F}^{[\alpha]}$ is also a Horn CNF for any assignment $\alpha$. These observations give the algorithm

```
Input: Horn CNF 𝓕
Output: 'Yes' if 𝓕 is satisfiable, 'No' otherwise
while ([] ∉ 𝓕) {
  If 𝓕 = ⟨⟩ or every clause in 𝓕 has size ≥ 2, return 'Yes'
  Pick a clause [u] ∈ 𝓕 of size = 1
  Remove all clauses containing u from 𝓕, and remove all literal u from all clauses
      containing it
}
return 'No'
```

Listing 1: Horn CNF Linear Time

**Definition 4.18.** Naive Backtracking Algorithm

```
Input: CNF formula F
Output: Satisfying assignment if one exists, ⊥ otherwise
if F = ⟨⟩ then
  return ∅
else if [] ∈ 𝓕 then
  return ⊥
else
  Let l be a literal in 𝓕 and set L :=Back(𝓕|l)
  if L ≠ ⊥ return L ∪ {l} else set L :=Back(𝓕|¬l)
  if L ≠ ⊥ return L ∪ {¬l} else return ⊥
```

Listing 2: Naive Backtracking Algorithm

$\mathcal{F}|l$ denotes the formula obtained by setting $l := T$, i.e., we remove all clauses containing $l$ and all occurrences of $\neg l$. In the computed assignment $L$, membership $l \in L$ means that literal $l$ is set to $T$.
However, we can improve this algorithm with two observations. Unit clauses $[l]$ force the assignment $l := T$; we call it unit clause propagation. Furthermore, pure literals, i.e., literals $l$ for which $\neg l$ does not appear in the current formula, can be set to $l := T$. In other words,

```
Input: CNF formula 𝓕
Output: Partial assignment
L := ∅; 𝓕' := 𝓕
while 𝓕' contains a unit clause [l] or a pure literal l do
  L := L ∪ {l}
  𝓕' := 𝓕'|l
return L
```

Listing 3: UnitPure

**Definition 4.19.** DPLL Algorithm

DPLL is the Davis-Putnam-Logemann-Loveland algorithm

```
Input: CNF formula 𝒢
Output: Satisfying assignment if one exists, ⊥ otherwise
U :=UnitPure(𝒢)
ℱ := 𝒢|U
If F = ⟨⟩ then
  return ∅
else if [] ∈ ℱ then
  return ⊥
else
  Let l be a literal in ℱ and set L :=DPLL(ℱ|l)
  if L ≠ ⊥ return U ∪ L ∪ {l} else set L :=DPLL(ℱ| ≠ l)
  if L ≠ ⊥ return U ∪ L ∪ {≠ l} else return ⊥
```

Listing 4: DPLL(G)

However, which literal $l$ to choose in the next recursion step and which branch $l$ or $\neg l$ first?
Static heuristics linear ordering of variables fixed before start, usually very fast to compute, and thus also use more expensive algorithms. Or we could try dynamic heuristics, where we determine ordering based on current formula $\mathcal{F}$, typically from number of occurrences of literals.

**Definition 4.20.** DLIS

Dynamic Largest Individual Sum is when we choose literal which occurs most frequently

**Definition 4.21.** MOMS

Maximum occurrence in clauses of minimum size, i.e., we choose literal which occurs most frequently in clauses of minimum size

**Definition 4.22.** Implication Graph

The implication graph is a directed graph associated with any particular stage of the algorithm

- For each literal $l$ set to true, called a decision literal, the graph contains a node labelled $l$
- For any clause $C = [l_1, \ldots, l_k, l]$, where $\neg l_1, \ldots, \neg l_k$ are nodes, add a new node $l$ and edges from $\neg l_i \to l$ for all $i = 1, \ldots, k$. These edges correspond to the clause $C$.

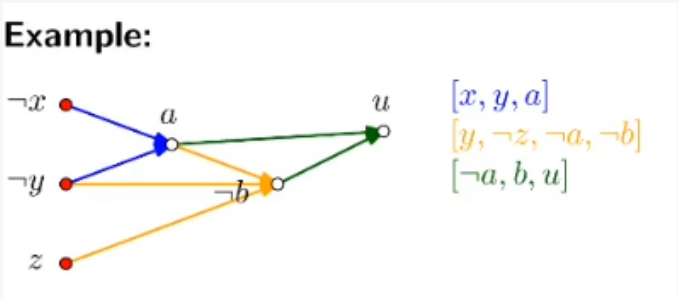



Figure 3: Implication Graph Example

A conflict literal $l$ is one for which both $l$ and $\neg l$ appear as nodes in the graph. A conflict graph is a sub graph of the implication graph that contains exactly one conflict literal $l$, only nodes that have a path to $l$ or $\neg l$, and for every node only the incoming edges corresponding to one particular clause

**Definition 4.23.** CDCL Algorithm

CDCL, the conflict-driven clause learning works as follows:

1. Select a variable and assign $T$ or $F$
2. Apply unit clause propagation
3. Build the implication graph
4. If there is any conflict
   (a) Derive a corresponding conflict clause and add it to the formula
   (b) Non-chronologically backtrack ("back jump") to the decision level where the first-assigned variable involved in the conflict was assigned
5. Continue from step 1 until all variable values are assigned

# 5 First Order Logic

**Definition 5.1.** First-Order Language

A first-order language is determined by specifying:

- A finite or countable set $\mathbf{R}$ of relation symbols, or predicate symbols. Each has some number of arguments
- A finite or countable set $\mathbf{F}$ of function symbols. Each has some number of arguments
- A finite or countable set $\mathbf{C}$ of constant symbols

We use $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ for the first-order language determined by $\mathbf{R}$, $\mathbf{F}$, $\mathbf{C}$. Sometimes it is useful to think of constant symbols as function symbols with 0 arguments.

**Definition 5.2.** Model

A model for the first order language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is a pair $\mathbf{M} = (\mathbf{D}, \mathbf{I})$ where

- $\mathbf{D}$ is a nonempty set, called the domain of $\mathbf{M}$.
- $\mathbf{I}$ is a mapping, called an interpretation that associates:
  - To every constant symbol $c \in \mathbf{C}$, some member $c^I \in \mathbf{D}$
  - To every function symbol $f \in \mathbf{F}$ with $n$ arguments, some $n$-ary function $f^I : \mathbf{D}^n \to \mathbf{D}$
  - To every function symbol $R \in \mathbf{R}$ with $n$ arguments, some $n$-ary relation $R^I \subseteq \mathbf{D}^n$

An assignment in a model $\mathbf{M} = (\mathbf{D}, \mathbf{I})$ is a mapping $\mathbf{A}$ from the set of variables to the set $\mathbf{D}$. We write $x^{\mathbf{A}}$ for the image of $x$ under $\mathbf{A}$.

**Definition 5.3.** Values of terms

Let $\mathbf{M} = (\mathbf{D}, \mathbf{I})$ be a model for the language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, and let $\mathbf{A}$ be an assignment in this model. To each term $t$ of the language, we associate a value $t^{\mathbf{I},\mathbf{A}}$ as follows:

- For a constant symbol $c$, $c^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I}}$
- For a variable $x$, $x^{\mathbf{I},\mathbf{A}} = x^{A}$
- For a function symbol $f$ with $n$ arguments, $(f(t_1, \ldots, t_n))^{\mathbf{I},\mathbf{A}} = f^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}})$

**Definition 5.4.** Example

Suppose $L$ has a constant symbol 0, a 1-argument function symbol $s$, and a 2-argument function symbol $+$. Consider the terms $t_1 := s(s(0) + s(x))$ and $t_2 := s(x + s(x + s(0)))$.
$\mathbf{D} = \{0, 1, 2, \ldots\}$, $0^I = 0$, $s^I$ is the successor function, and $+^I$ is the addition operation. If $\mathbf{A}$ is an assignment such that $x^{\mathbf{A}} = 3$, then we have $t_1^{\mathbf{I},\mathbf{A}} = 6$ and $t_2^{\mathbf{I},\mathbf{A}} = 9$. More generally, $t_1^{\mathbf{I},\mathbf{A}} = x^{\mathbf{A}} + 3$ and $t_2^{\mathbf{I},\mathbf{A}} = 2x^{\mathbf{A}} + 3$
$\mathbf{D}$ is the set of all words over the alphabet $\{a, b\}$, $0^I = a$, $s^{\mathbf{I}}$ appends $a$ to the end of a word, and $+^{\mathbf{I}}$ is the concatenation. If $\mathbf{A}$ is an assignment such that $x^{\mathbf{A}} = aba$ , then $t_1^{\mathbf{I},\mathbf{A}} = aaabaaaa$ and $t_2^{\mathbf{I},\mathbf{A}} = abaabaaaaa$

**Definition 5.5.** Truth of formulas

Let $\mathbf{M} = (\mathbf{D}, \mathbf{I})$ be a model for the language $L$ and let $\mathbf{A}$ be an assignment to this model. To each $\phi$ of $L$, we associate a truth value $\phi^{\mathbf{I},\mathbf{A}}$ as follows:

- $R(t_1, \ldots, t_n)^{\mathbf{I},\mathbf{A}} = T \iff (t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}}) \in R^I$

**Definition 5.6.** Gamma and Delta Formulas

We extend the notation of $\alpha$ and $\beta$ formulas for our first order logic formulas. Namely, we have

Table 4: Gamma and Delta

| $\gamma$ | $\gamma(t)$ | $\delta$ | $\delta(t)$ |
|---|---|---|---|
| $\forall x\phi$ | $\phi(\frac{x}{t})$ | $\exists\phi$ | $\phi(\frac{x}{t})$ |
| $\neg\exists x\phi$ | $\neg\phi(\frac{x}{t})$ | $\neg\forall x\phi$ | $\neg\phi(\frac{x}{t})$ |

Here, $\phi(\frac{x}{t})$ denotes the formula obtained from $\phi$ by substituting the free occurrences of the variable $x$ by the term $t$.

**Example 5.1.** Tableau Example

Consider the statement

$$\phi := \forall x(P(x) \vee Q(x)) \to (\exists x P(x) \vee Q(x)) \tag{77}$$

Consider

$$\neg\phi = \neg(\forall x(P(x) \vee Q(x)) \to (\exists x P(x) \vee \forall x Q(x))) \tag{78}$$
$$\forall x(P(x) \vee Q(x)) \ \alpha \text{ expansion} \tag{79}$$
$$\neg(\exists x P(x) \vee \forall x Q(x))) \ \alpha \text{ expansion} \tag{80}$$
$$\neg\exists x P(x) \ \alpha \text{ expansion on second} \tag{81}$$
$$\neg\forall x Q(x) \ \alpha \text{ expansion on second} \tag{82}$$
$$\neg Q(p) \ \delta \text{ with new parameter } p \tag{83}$$
$$P(p) \vee Q(p) \ \gamma \text{ with existing parameter } p \text{ on first equation} \tag{84}$$
$$\neg P(p) \ \gamma \text{ with existing parameter } p \tag{85}$$

We now apply $\beta$ expansion, creating two new branches on our $\vee$ statement.

$$P(p) Q(p) \tag{86}$$

Since we have $\neg Q(p)$ and $\neg P(p)$ above the $\beta$ branch, we can close both branches. Our proof is finished.

**Example 5.2.** Resolution

Consider the statement

$$[\neg(\phi := \forall x(P(x) \vee Q(x)) \to (\exists x P(x) \vee Q(x)))] \tag{87}$$

We now notice $\alpha$ expansion

$$[\forall x(P(x) \vee Q(x))] \tag{88}$$
$$[\neg(\exists x P(x) \vee \forall x Q(x))] \tag{89}$$
$$[\neg\exists x P(x)] \tag{90}$$
$$[\neg\forall x Q(x)] \tag{91}$$
$$[\neg Q(p)] \tag{92}$$
$$[\neg P(p)] \tag{93}$$
$$[P(p) \vee Q(p)] \tag{94}$$
$$[P(p), Q(p)] \tag{95}$$
$$[P(p)] \tag{96}$$
$$[] \tag{97}$$

**Definition 5.7.** Natural Deduction

For natural deduction, we have two new rules for $\gamma E$:

Table 5: Gamma Rule

| $\gamma$ | $\delta$ |
|---|---|
| $\gamma(t)$ | $\delta(p)$ |

**Example 5.3.** Natural Deduction Example

Consider the statement

$$\phi := \forall x(P(x) \to Q(x)) \to (\forall x P(x) \to \forall x Q(x)) \tag{98}$$

$$\begin{aligned}
\neg\phi \\
\forall x(P(x) \to Q(x)) \\
\neg(\forall x P(x) \to \forall x Q(x))\ \alpha E \\
\forall x P(x) \\
\neg\forall x Q(x)\ \alpha E \\
\neg Q(p)\ \text{new variable } p \text{ from } \delta \\
P(p)\ \gamma \text{ rule with } t = p \\
P(p) \to Q(p)\ \gamma \text{ rule with } t = p \\
Q(p)\ \text{modus ponens} \\
\bot
\end{aligned}$$

**Example 5.4.** Example with multiple gamma and delta

Consider

$$F(x, y, z) := \exists x \forall y \forall z(P(y) \to Q(x)) \to (P(x) \to Q(x)) \tag{99}$$

We will prove this using tableau.

$$\begin{aligned}
\neg\exists x \forall y \forall z(P(y) \to Q(x)) \to (P(x) \to Q(x)) && (100)\\
\neg\forall y \forall z F(p, y, z)\ \text{where } p \text{ is an existing parameter} && (101)\\
\neg\forall z F(p, q, z)\ \text{where } q \text{ is a new parameter} && (102)\\
\neg F(p, q, r)\ \text{where } r \text{ is a new parameter} && (103)\\
\neg\forall y \forall z F(q, y, z)\ \gamma \text{ with now existing term } q && (104)\\
\neg F(q, q', r')\ \text{where } q', r' \text{ are new parameters} && (105)\\
(r, q'', r'')\ \gamma \text{ with now existing term } r \text{ and new parameters} && (106)\\
P(q) \to Q(r)\ \alpha_1 && (107)\\
\neg(P(p) \to Q(p))\alpha_2 && (108)\\
P(q') \to Q(r') && (109)\\
\neg(P(q) \to Q(q)) && (110)\\
P(q'') \to Q\,(r'') && (111)\\
\neg(P\,(r) \to Q\,(r)) && (112)\\
P(r) && (113)\\
\neg Q(r) && (114)\\
P(q) && (115)\\
\neg Q(q) && (116)
\end{aligned}$$

After all alpha expansion we continue with $\beta$ expansion of

$$\neg P(p), Q(r) \tag{117}$$

# 6 Program Verification

**Definition 6.1.** Weakest Precondition

Weakest preconditions can be computed formally for each language construct - they define how that construct behaves. For assignments, a formula will be true afterwards exactly when it holds beforehand with new role:

$$(\!|WP?|\!)x = x + 1(\!|x > 3|\!) \tag{118}$$

So $x + 1 > 3$ is the weakest precondition, or equivalently $x > 2$. Formally this is

$$wp(x = E, Post) = Post[\frac{x}{E}] \tag{119}$$

where $Post[\frac{x}{E}]$ is the condition obtained from Post by replacing $x$ by $E$

**Example 6.1.** Weakest Precondition

Consider

$$wp(i = i + 1, i > 0) \tag{120}$$

We obtain that

$$i + 1 > 0 \tag{121}$$
$$i > -1 \tag{122}$$

Or consider another example where

$$wp(z = x, (z \geq x \wedge z \geq y)) \tag{123}$$

Through substitution we have

$$x \geq x \wedge x \geq y \tag{124}$$
$$T \wedge x \geq y \tag{125}$$
$$x \geq y \tag{126}$$

For composites we have

$$wp(x = y + 2; y = 2 \times x, x + y > 20) \tag{127}$$
$$= wp(x = y + 2; wp(y = 2x, x + y > 20)) \tag{128}$$
$$= wp(x = y + 2, x + y > 20[\frac{y}{2} \times x]) \tag{129}$$
$$= wp(x = y + 2, x + 2x > 20) \tag{130}$$
$$= wp(x = y + 2, 3x > 20) \tag{131}$$
$$= 3(y + 2) > 20 \tag{132}$$
$$= 3y + 6 > 20 \tag{133}$$
$$= 3y > 14 \tag{134}$$

For conditional statements we want to check the post condition for both if and else, and we choose whichever is the strongest, or combine both if required.

$$wp(\text{if } x > 0 \text{ then } y = x \text{ else } y = -x, y = |x|) \tag{135}$$
$$-x = |x| \text{ for else} \tag{136}$$
$$x = |x| \text{ for then} \tag{137}$$
$$wp = (B \rightarrow wp_1) \wedge (\neg B \rightarrow wp_2) \tag{138}$$
$$= (x > 0 \rightarrow x = |x|) \wedge (x \leq 0 \rightarrow -x = |x|) \tag{139}$$
$$= T \wedge T \tag{140}$$
$$= \top \tag{141}$$

**Definition 6.2.** Hoare Logic

Any program is a sequence of instructions

$$Prog = C_1; C_2; \ldots; C_n \tag{142}$$

We lay out a proof for $Prog$ in the following format:

$$(\!|\phi_0|\!) \tag{143}$$
$$C_1 \tag{144}$$
$$(\!|\phi_1|\!) \tag{145}$$
$$C_2 \tag{146}$$
$$(\!|\phi_2|\!) \tag{147}$$
$$\ldots \tag{148}$$
$$(\!|\phi_3|\!) \tag{149}$$
$$C_n \tag{150}$$
$$(\!|\phi_n|\!) \tag{151}$$

The validity of each Hoare triples $(\!|\phi_{i-1}|\!)C_i(\!|\phi_i|\!)$ must be inferred for some rule. This then allows us to deduce $(\!|\phi_0|\!)Prog(\!|\phi_n|\!)$

**Definition 6.3.** Assignment Rule

The assignment rule states

$$\frac{}{(\!|Post[\frac{x}{E}]|\!)x = E(\!|Post|\!)}\text{Assignemnt} \tag{152}$$

The box is read in such a way, that if you have what is written in the above line, you can deduce the bottom.

**Definition 6.4.** Implied Rule

Combining observations motivates us the rule

$$\frac{Pre \to P \qquad (\!|P|\!)Prog(\!|Q|\!) \qquad Q \to Post}{(\!|Pre|\!)Prog(\!|Post|\!)}\text{Implied} \tag{153}$$

**Example 6.2.** Example

Prove that
$(\!|x = x_0 \ \wedge \ y = y_0|\!)\ x = x + y; y = x - y; x = x - y\ (\!|x = y_0 \ \wedge \ y = x_0|\!)$

$(\!|x = x_0 \ \wedge \ y = y_0|\!)$
$(\!|y = y_0 \ \wedge \ x + y - y = x_0|\!)$ Implied
$x = x + y;$
$(\!|y = y_0 \ \wedge \ x - y = x_0|\!)$ Assignment
$(\!|x - (x - y)) = y_0 \ \wedge \ x - y = x_0|\!)$ Implied
$y = x - y;$
$(\!|x - y = y_0 \ \wedge \ y = x_0|\!)$ Assignment
$x = x - y$
$(\!|x = y_0 \ \wedge \ y = x_0|\!)$ Assignment

Figure 4: Example

**Definition 6.5.** Conditional Rule

An if statement takes one of two branches, depending on whether the condition is true or false

$$\frac{(\!|Pre \wedge B|\!)C_1(\!|Post|\!) \qquad (\!|Pre \wedge \neg B|\!)C_2(\!|Post|\!)}{(\!|Pre|\!) \text{ if } B\{C_1\} \text{ else } \{C_2\}(\!|Post|\!)} \tag{154}$$

**Example 6.3.** Example

Prove that $(\!|\top|\!)$ *if* $(x < y)$ *then* $\{z = x\}$ *else* $\{z = y\}$ $(\!|z \leq x \wedge z \leq y|\!)$

$(\!|\top|\!)$
*if* $(x < y)$ *then* {
$(\!|x < y|\!)$
$(\!|x \leq x \wedge x \leq y|\!)$ Implied
$z = x$
$(\!|z \leq x \wedge z \leq y|\!)$ Assignment
} *else* {
$(\!|\neg(x < y)|\!)$
$(\!|y \leq x \wedge y \leq y|\!)$ Implied
$z = y$
$(\!|z \leq x \wedge z \leq y|\!)$ Assignment
}
$(\!|z \leq x \wedge z \leq y|\!)$ If

Figure 5: Example of Conditional Rule

**Definition 6.6.** Loop Rule

A while statement is iterated as long as the condition is satisfied

$$\frac{(\!|B \wedge L|\!)C(\!|L|\!)}{(\!|L|\!) \text{ while } B\{C\}(\!|\neg B \wedge L|\!)} \tag{155}$$

L is called the loop invariant, it holds before and after each iteration. Finding a good loop variant is the key.