ITB8832 Mathematics for Computer Science
Lecture 3 – 16 September 2024

Chapter Three

- Propositional Logic in Computer Programs
- Equivalence and Validity
- The Algebra of Propositions
- The SAT problem
- Predicate Formulas

Contents

Next section

Condition checking with propositional logic

Consider a piece of Python code such as:

```
if x > 0 or (x <= 0 and y > 100):
    %% your code here
```

- Can we determine if and when your code will be run?
- Can we write the if-condition in a simpler form?

Let us consider the following propositions:

- $A ::= \texttt{x > 0}$
- $B ::= \texttt{y > 100}$

We observe that $\texttt{x} <= 0$ is just not(A), so:

$$\texttt{x > 0 or (x <= 0 and y > 100)}$$
corresponds to A or (not(A) and B)

Condition checking with propositional logic

Consider a piece of Python code such as:

```
if x > 0 or (x <= 0 and y > 100):
    %% your code here
```

- Can we determine if and when your code will be run?
- Can we write the if-condition in a simpler form?

Let us consider the following propositions:

- $A ::= \texttt{x} > 0$
- $B ::= \texttt{y} > 100$

We observe that $\texttt{x} <= 0$ is just not(A), so:

`x > 0 or (x <= 0 and y > 100)`
corresponds to A or (not(A) and B)

Equivalent formulas

Definition

Let α and β be formulas in the variables $P_1, \ldots, P_n$.
α and β are *equivalent* if every assignment of truth values to $P_1, \ldots, P_n$ makes α and β either both true, or both false.

Equivalent formulas

Definition

Let α and β be formulas in the variables $P_1, \ldots, P_n$.
α and β are *equivalent* if every assignment of truth values to $P_1, \ldots, P_n$ makes α and β either both true, or both false.

Examples:

- $\alpha ::= P \text{ or } Q$ and $\beta ::= \text{not}(\text{not}(P) \text{ and } \text{not}(Q))$.
- $\alpha ::= P \text{ implies } (Q \text{ implies } P)$ and $\beta ::= R \text{ or } \text{not}(R)$.

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

We start with the basics of the table:

A	B	A or	(not(A)	and B)	A or B
T	T				
T	F				
F	T				
F	F				

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

We fill the rightmost column, and take a note of the values:

A	B	A or	(not(A)	and B)	A or B
T	T				T
T	F				T
F	T				T
F	F				F

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

We convert A into not(A), and take note of the values:

A	B	A or	(not(A)	and B)	A or B
T	T		F		T
T	F		F		T
F	T		T		T
F	F		T		F

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

We now determine the values of (not(A) and B):

A	B	A or	(not(A)	and B)	A or B
T	T		F	F	T
T	F		F	F	T
F	T		T	T	T
F	F		T	F	F

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

Finally, we determine the values of A or (not(A) and B):

A	B	A or	(not(A)	and B)	A or B
T	T	T	F	F	T
T	F	T	F	F	T
F	T	T	T	T	T
F	F	F	T	F	F

Truth table calculation

Claim

A or (not(A) and B) is equivalent to A or B.

Finally, we determine the values of A or (not(A) and B):

A	B	A or	(not(A)	and B)	A or B
T	T	T	F	F	T
T	F	T	F	F	T
F	T	T	T	T	T
F	F	F	T	F	F

... and we see that they always match, proving the claim.
We can then rewrite the snippet as:

```
if x > 0 or y > 100:
    %% your code here
```

Simplifying by reasoning

We can also prove the equivalence by reasoning case by case:
(and making some observations in the meantime)

$A = \mathrm{T}$ A formula of the form T or Q has truth value T.
If A is T, so are both A or (not(A) and B) and A or B.

$A = \mathrm{F}$ A formula of the form F or Q, or of the form T and Q, has the same truth value as Q.
If A is F, then not(A) and B has the same truth value of B, and so do A or (not(A) and B) and A or B.

In either case, A or (not(A) and B) and A or B take the same truth value on each assignment of A and B.

Why simplify?

1. To improve *readability*.
   Conditions with a simple structure are more easily checked than complex ones.
2. To increase *speed*.
   Less complex formulas require less time to be evaluated.
3. To reduce *cost*.
   The formula might refer to a circuit, whose realization requires materials, tools, time, and money.

Symbolic notation for logical connectives

English	Symbolic
not(P)	$\neg P, \overline{P}$
P and Q	$P \wedge Q$
P or Q	$P \vee Q$
P xor Q	$P \oplus Q$
P implies Q	$P \longrightarrow Q$
P iff Q	$P \longleftrightarrow Q$

Precedence

From strongest to weakest:

1. not$(\cdot)$
2. and
3. or
4. xor
5. implies
6. iff

For example,

$$\text{not}(A) \text{ and } B \text{ or } C \text{ implies } D \text{ iff } E \text{ xor } F$$

is a shorthand for

$$((((\text{not}(A)) \text{ and } B) \text{ or } C) \text{ implies } D) \text{ iff } (E \text{ xor } F)$$

When in doubt: use parentheses.

Next section

Contrapositives

Definition

The *contrapositive* of the formula P implies Q is the formula not(Q) implies not(P).

Contrapositives are equivalent to each other.

P	Q	P implies Q	not(Q)	implies	not(P)
T	T	T	F	T	F
T	F	F	T	F	F
F	T	T	F	T	T
F	F	T	T	T	T

Contrapositives

Definition

The *contrapositive* of the formula P implies Q is the formula not(Q) implies not(P).

Contrapositives are equivalent to each other.
For example,

If I am hungry, then I am grumpy

is equivalent to

If I am not grumpy, then I am not hungry

Converses

Definition

The *converse* of the formula P implies Q is the formula Q implies P.

Converses *are not* equivalent to each other!

P	Q	P implies Q	Q implies P
T	T	T	T
T	F	F	T
F	T	T	F
F	F	T	T

Converses

Definition

The *converse* of the formula P implies Q is the formula Q implies P.

Converses *are not* equivalent to each other!
For example,

If I am hungry, then I am grumpy

is not equivalent to

If I am grumpy, then I am hungry

Converses

Definition

The *converse* of the formula P implies Q is the formula Q implies P.

Converses *are not* equivalent to each other!
However, *conjunction of converses is equivalent to* iff *.*

P	Q	(P implies Q)	and	(Q implies P)	P iff Q
T	T	T	T	T	T
T	F	F	F	T	F
F	T	T	F	F	F
F	F	T	T	T	T

Converses

Definition

The *converse* of the formula P implies Q is the formula Q implies P.

Converses *are not* equivalent to each other!
However, *conjunction of converses is equivalent to* iff *.*
For example,

If I am hungry, then I am grumpy, and if I am grumpy, then I am hungry

is equivalent to

I am grumpy if and only if I am hungry

Validity

Definition

A propositional formula is *valid* if it is true for *every* assignment of truth values to its variables.

Validity

Definition

A propositional formula is *valid* if it is true for *every* assignment of truth values to its variables.

Examples:

- not(P and not(P)) *law of non-contradiction*
- P or not(P) *law of excluded middle*
- P iff not(not(P)) *double negation*
- P implies (Q implies P) *weakening*
- $(P \longrightarrow (Q \longrightarrow R)) \longrightarrow ((P \longrightarrow Q) \longrightarrow (P \longrightarrow R))$ *conditional modus ponens*

Validity

Definition

A propositional formula is *valid* if it is true for *every* assignment of truth values to its variables.

Examples:

- not(P and not(P)) — *law of non-contradiction*
- P or not(P) — *law of excluded middle*
- P iff not(not(P)) — *double negation*
- P implies (Q implies P) — *weakening*
- $(P \longrightarrow (Q \longrightarrow R)) \longrightarrow ((P \longrightarrow Q) \longrightarrow (P \longrightarrow R))$ — *conditional modus ponens*

Non-example:

- P, where P is any propositional variable.

Satisfiability

Definition

A propositional formula is *satisfiable* if it is true for *some* assignment of truth values to its variables.
We say that such assignment *satisfies* the formula.

Satisfiability

Definition

A propositional formula is *satisfiable* if it is true for *some* assignment of truth values to its variables.
We say that such assignment *satisfies* the formula.

Examples:

- P, where P is a propositional variable.
  That is: every atomic formula is satisfiable.
- $P \otimes Q$, where P and Q are variables and $\otimes$ is any of the binary connectives and , or , implies , iff , and xor .

Satisfiability

Definition

A propositional formula is *satisfiable* if it is true for *some* assignment of truth values to its variables.
We say that such assignment *satisfies* the formula.

Examples:

- P, where P is a propositional variable.
  That is: every atomic formula is satisfiable.
- $P \otimes Q$, where P and Q are variables and $\otimes$ is any of the binary connectives and , or , implies , iff , and xor .

Non-example:

- A and not(A), where A is any formula.

Validity, satisfiability, and equivalence

Let P and Q be formulas.

Theorem

P is valid if and only if not(P) is unsatisfiable.
P is satisfiable if and only if not(P) is not valid.

Theorem

P and Q are equivalent if and only if P iff Q is valid.

Next section

Disjunctive normal forms: An example

Let $\phi ::= A \text{ and } (B \text{ or } C)$. Consider its truth table:

A	B	C	ϕ
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

The assignments of (A, B, C) which make ϕ true are (T, T, T), (T, T, F), and (T, F, T). These are the same assignments that make the following formula true:

$$(A \text{ and } B \text{ and } C) \text{ or } (A \text{ and } B \text{ and } \overline{C}) \text{ or } (A \text{ and } \overline{B} \text{ and } C)$$

Formulas in disjunctive normal form

Definition

- A *literal* is a symbol of the form A or $\overline{A}$ where A is a propositional variable.
- An and *-clause* is a conjunction of literals where each variable appears at most once, either as itself or as its negation.
- A formula ψ in n variables $P_1, \ldots, P_n$ is in *disjunctive normal form (DNF)* if it is written as a disjunction of and -clauses.
- If every variable appears in every conjunction (either as itself or its negation) the DNF is said to be *full*.

For example, this formula is in DNF:

$$(A \text{ and } B \text{ and } C) \text{ or } (A \text{ and } B \text{ and } \overline{C}) \text{ or } (A \text{ and } \overline{B} \text{ and } C)$$

and so is this one:

$$(A \text{ and } B) \text{ or } (A \text{ and } \overline{B} \text{ and } C)$$

but these ones are not:

$$A \text{ and } (B \text{ or } C);\ \ A \text{ and } B \text{ and } C \text{ and } A;\ \ \text{not}(A \text{ and } B \text{ and } C)$$

Disjunctive normal form(s) of a formula

Definition

A *disjunctive normal form* of a formula ϕ is a formula ψ in DNF which is equivalent to ϕ.

For example,

$$(A \text{ and } B \text{ and } C) \text{ or } (A \text{ and } B \text{ and } \overline{C}) \text{ or } (A \text{ and } \overline{B} \text{ and } C)$$

is a disjunctive normal form of

$$A \text{ and } (B \text{ or } C)$$

Existence of the DNF

Theorem

Every *satisfiable* propositional formula has a DNF.

Existence of the DNF

Theorem

Every *satisfiable* propositional formula has a DNF.

Proof:

- Let $P_1, \ldots, P_n$ be the variables of the formula ϕ.
- Construct the truth table of ϕ.
- For each row where ϕ has value T, construct a conjunction (A_1 and ... and A_n) where:
  - $A_i = P_i$ if $P_i = \text{T}$ on the row;
  - $A_i = \text{not}(P_i)$ if $P_i = \text{F}$ on the row.
- The disjunction of all these conjunctions is a DNF for ϕ.

Satisfiability and DNF

The procedure in the previous slide constructs a DNF from the rows of the truth table where the formula is true.

- This presumes that there is at least one such row.
- But what if there is none?[1]

A possible way out is to use the following convention:

The DNF of an unsatisfiable formula is empty.

This is a patch rather than a fix, because we did not define propositional formulas so that they could be empty.

[1] Remarkably, the textbook says nothing about this.

Conjunctive normal forms

"Dually" to DNF, we have:

Definition

- An *or -clause* is a disjunction of literals where each variable appears at most once, either as itself or as its negation.
- A formula ψ in n variables $P_1, \ldots, P_n$ is in *conjunctive normal form (CNF)* if it is written as a conjunction of or -clauses.
- If every variable appears in every conjunction (either as itself or its negation) the CNF is said to be *full*.
- A *conjunctive normal form* of a formula ϕ is a formula ψ in CNF which is equivalent to ϕ.

Theorem

Every *non-valid* propositional formula has a CNF.

Exercise: Modify the algorithm to derive the full DNF of a satisfiable formula to obtain an algorithm that derives the full CNF of a non-valid formula.

An algebra for propositional calculus

George Boole (1815-1864) defined a set of rules for manipulating propositional formula, which are now known as *Boolean algebra*.

- These rules are given as equivalence between propositional formulas constructed via the connectives $\wedge$, $\vee$, and $\neg$.
- The reason is that $\wedge$, $\vee$, and $\neg$ form a *basis of connectives:* Every propositional formula is equivalent to a formula where the only connectives are $\wedge$, $\vee$, and $\neg$.
(For example: a DNF if it is satisfiable, or a CNF if it is not valid.)

The first axiom is the *law of double negation:*

$$\neg(\neg A) \longleftrightarrow A$$

An algebra for the propositional calculus: and

The following formulas are all valid:

$A \wedge B$	$\longleftrightarrow$	$B \wedge A$	commutativity
$(A \wedge B) \wedge C$	$\longleftrightarrow$	$A \wedge (B \wedge C)$	associativity
$A \wedge A$	$\longleftrightarrow$	A	idempotence
$A \wedge \mathsf{T}$	$\longleftrightarrow$	A	identity
$A \wedge \mathsf{F}$	$\longleftrightarrow$	F	zero
$A \wedge \overline{A}$	$\longleftrightarrow$	F	noncontradiction
$A \wedge (B \vee C)$	$\longleftrightarrow$	$(A \wedge B) \vee (A \wedge C)$	distributivity
$A \wedge (B \vee A)$	$\longleftrightarrow$	A	absorption
$\neg(A \wedge B)$	$\longleftrightarrow$	$\overline{A} \vee \overline{B}$	de Morgan's law

An algebra for the propositional calculus: or

The following formulas are all valid:

$A \vee B$	$\longleftrightarrow$	$B \vee A$	commutativity
$(A \vee B) \vee C$	$\longleftrightarrow$	$A \vee (B \vee C)$	associativity
$A \vee A$	$\longleftrightarrow$	A	idempotence
$A \vee \mathsf{F}$	$\longleftrightarrow$	A	identity
$A \vee \mathsf{T}$	$\longleftrightarrow$	T	unit
$A \vee \overline{A}$	$\longleftrightarrow$	T	excluded middle
$A \vee (B \wedge C)$	$\longleftrightarrow$	$(A \vee B) \wedge (A \vee C)$	distributivity
$A \vee (B \wedge A)$	$\longleftrightarrow$	A	absorption
$\neg(A \vee B)$	$\longleftrightarrow$	$\overline{A} \wedge \overline{B}$	de Morgan's law

A strategy for DNF

Let ϕ be an arbitrary propositional formula.

1. Apply *de Morgan's laws* until $\neg$ is only applied to single variables.
2. Apply *distributivity* to obtain a disjunction of conjunctions.
3. Apply *idempotence* to remove multiple instances of variables within conjunctions.
4. Apply *associativity* to remove unnecessary parentheses.
5. Complete each conjunction so that, for each variable P, exactly one between P and $\overline{P}$ appears in it.
   To do this, exploit that $A \longleftrightarrow A \wedge (B \vee \overline{B})$ is a valid formula, following from $A \wedge \mathsf{T} \longleftrightarrow A$ and $B \vee \overline{B} \longleftrightarrow \mathsf{T}$.
6. Simplify the formula by using distributivity, commutativity, and absorption.

Completeness of propositional calculus

Theorem

Two propositional formulas *are* equivalent *if and only if* they *can be proved* to be equivalent via the axioms of Boolean algebra.

Proof: (sketch)

- *Simple:* As all the axioms of Boolean algebra are equivalences, so must be any proposition proved starting from them.
- *Complicated:* The axioms of Boolean algebra allow conversion to disjunctive normal form, and two formulas are equivalent iff they have the same DNF (up to commutativity).

Next section

The Satisfiability problem

The *Satisfiability problem*, denoted as SAT, is:

Given an arbitrary Boolean formula ϕ,
determine if ϕ is satisfiable.

The Satisfiability problem

The *Satisfiability problem*, denoted as SAT, is:

Given an arbitrary Boolean formula ϕ,
determine if ϕ is satisfiable.

How difficult can this be?

Conceptually: not much

1. Put ϕ in disjunctive normal form.
2. Use truth tables to determine if ϕ is true for some assignment of variables.

The Satisfiability problem

The *Satisfiability problem*, denoted as SAT, is:

Given an arbitrary Boolean formula ϕ,
determine if ϕ is satisfiable.

How difficult can this be?

Conceptually: not much

1. Put ϕ in disjunctive normal form.
2. Use truth tables to determine if ϕ is true for some assignment of variables.

Computationally: A LOT

- Suppose ϕ depends on n Boolean variables.
- If ϕ is not satisfiable, we need to test *each one of the 2^n truth assignments* to prove so.
- For $n = 50$ variables, with a computer capable of 1 million such tests per second, this takes *more than thirty-five years*.

Big Oh notation

Definition

Given two functions $f, g : \mathbb{N} \to [0, +\infty)$ we say that *$f(n)$ is big Oh of $g(n)$*, and write $f(n) = O(g(n))$, if there exist $n_0 \in \mathbb{N}$ and $C > 0$ such that

$$f(n) \leq C \cdot g(n) \text{ for every } n \geq n_0 .$$

- If $T(n)$ is the maximum time required to solve SAT for a given formula, then $T(n) = O(2^n)$.
- Problems only solvable in exponential or larger time are considered to be *intractable*.

Polynomial time algorithms

Definition

An algorithm runs in *polynomial time* $T(n)$ in the size n of its input if $T(n) = O(n^k)$ for some $k \geq 1$.

The class of polynomial-time algorithms has some “good” features:

- Polynomials “do not grow too fast”.
- The sum and the product of two polynomials are polynomials.
- A *composition* of polynomials is still a polynomial:
  If $p(x)$ and $q(x)$ are polynomials, then so is $p(q(x))$, which is what you obtain if you replace every occurrence of x in $p(x)$ with $q(x)$ and simplify.
- Hence, an algorithm where all the cycles have polynomial length and all the subroutines run in polynomial time, also runs in polynomial time.

P versus NP

Definition: P

The class P is the class of the decision problems that have a *solution algorithm* which runs in polynomial time in the size of the input.

That is: problem X is in class P if and only if there are a polynomial $p(t)$ and an algorithm A running in time $O(p(n))$ for inputs of size n which, however given in input an instance I of X, produces in output the YES/NO answer to I.

Definition: NP

The class NP is the class of the decision problems that have a *verification algorithm* which runs in polynomial time in the size of the input.

That is: problem X is in class NP if and only if there are a polynomial $p(t)$ and an algorithm A running in time $O(p(n))$ for inputs of size n which, however given in input an instance I of X *and a potential witness w that the answer to I is YES*, determines if w is really so.

P versus NP

Definition: P

The class P is the class of the decision problems that have a *solution algorithm* which runs in polynomial time in the size of the input.

Definition: NP

The class NP is the class of the decision problems that have a *verification algorithm* which runs in polynomial time in the size of the input.

The following happens:

1. SAT belongs to NP.
2. For every problem X in NP there exists an algorithm that turns any instance I of X and potential witness w of I into an instance J of SAT and a potential witness z of J, in time polynomial in the size of I and w, and so that the answer to I is YES if and only if the answer to J is YES.

Consequently:

$$\text{If } \mathrm{SAT} \in \mathrm{P} \text{ then } \mathrm{P} = \mathrm{NP}.$$

What if $\mathrm{P} = \mathrm{NP}$?

The good:

- We can efficiently *design circuits*.
- We get efficient algorithms for *scheduling*.
- We can efficiently *distribute resources*.

What if $\mathrm{P} = \mathrm{NP}$?

The good:

- We can efficiently *design circuits*.
- We get efficient algorithms for *scheduling*.
- We can efficiently *distribute resources*.

The bad:

- Modern cryptography becomes *insecure*.

SAT solvers

There is currently a big interest in algorithms that, *under certain conditions,* solve SAT in polynomial time.

SAT solvers

There is currently a big interest in algorithms that, *under certain conditions,* solve SAT in polynomial time.

Question

Doesn't this presume that $\mathrm{SAT} \in \mathrm{P}$?

SAT solvers

There is currently a big interest in algorithms that, *under certain conditions,* solve SAT in polynomial time.

Question

Doesn't this presume that $\mathrm{SAT} \in \mathrm{P}$?

Answer: *no*, because

- even if *the problem as a whole* is not efficiently solvable,
- it might still be that *some well defined subclasses of cases* are.

Next section

Truth for predicates

Consider a predicate of the form: $x^2 \geq 0$.

- This is always true if x is a *real* number.
- But if x is a *complex* number, it might be false:
- For example, $i^2 = -1 < 0$.
- Worse still, $\left(\frac{1}{2} + i\frac{\sqrt{3}}{2}\right)^2 = -\frac{1}{2} + i\frac{\sqrt{3}}{2}$ is not even a real number, and cannot be said to be "smaller" or "larger" than zero.

How can we specify *when* a predicate is true?

Universal quantifier

Let $P(x)$ be a predicate depending on a variable x which takes values in a set S (the *type* of the variable).

Definition

The formula:

$$\forall x \in S \,.\, P(x)$$

is true if and only if $P(x)$ is true for *every* $x \in S$.

The formula can be read as follows:

- For every x in S, $P(x)$.
- $P(x)$ is true for every x in S.

For example, the following formulas are true:

$$\forall x \in \mathbb{R} \,.\, x^2 \geq 0\,; \;\; \forall n \in \mathbb{N} \,.\, \text{if } n \text{ is prime then } \sqrt{n} \text{ is irrational}$$

but the following ones are false:

$$\forall x \in \mathbb{C} \,.\, x^2 \geq 0\,; \;\; \forall n \in \mathbb{N} \,.\, \sqrt{n} \text{ is irrational}$$

Existential quantifier

Let $P(x)$ be a predicate depending on a variable x which takes values in a set S (the *type* of the variable).

Definition

The formula:

$$\exists x \in S \,.\, P(x)$$

is true if and only if $P(x)$ is true for *at least one* $x \in S$.

The formula can be read as follows:

- There exists x in S such that $P(x)$.
- $P(x)$ is true for some x in S.

For example, the following formulas are true:

$$\exists x \in \mathbb{R} \,.\, 5x^2 = 7 \;;\; \exists n \in \mathbb{N} \,.\, n^2 = 16$$

but the following ones are false:

$$\exists x \in \mathbb{R} \,.\, 5x^2 = -7 \;;\; \exists n \in \mathbb{N} \,.\, n^2 = 17$$

Precedence of quantifiers

Quantifiers have a *stronger* binding than propositional connectives:

$\forall x\,.\,P(x)$ implies Q stands for $(\forall x\,.\,P(x))$ implies Q.

However, some textbooks (including ours) seem to *also* use the following convention:

A quantifier using a variable x binds as many instances of x as possible before encountering another quantifier.

Example from the textbook (page 67, formula (3.27))

- Textbook: $\exists x\,.\,\forall y\,.\,P(x,y)$ implies $\forall x\,.\,\exists y\,.\,P(x,y)$.
- Meaning: $(\exists x\,.\,\forall y\,.\,P(x,y))$ implies $(\forall x\,.\,\exists y\,.\,P(x,y))$.

Again: When in doubt, use parentheses.

If you can solve any exercise, then you will pass the test

Let $\mathrm{solve}(x)$ be a predicate meaning that you can solve exercise x.
Let pass be a proposition meaning that you pass the test.

You can pass the test if you can solve only one exercise

$(\exists x \in \mathrm{Exercises} \,.\, \mathrm{solve}(x)) \longrightarrow \mathrm{pass}$

You can pass the test if you can solve one specific exercise

$\exists x \in \mathrm{Exercises} \,.\, (\mathrm{solve}(x) \longrightarrow \mathrm{pass})$

To pass the test, you need to be able to solve every single exercise

$\mathrm{pass} \longrightarrow \forall x \in \mathrm{Exercises} \,.\, \mathrm{solve}(x)$

Mixing quantifiers

Many mathematical statements involve more than one quantifier:

Goldbach's Conjecture

Every even integer larger than 2 is a sum of two primes.

If we define S as the set of the even integers larger than 2, Goldbach's conjecture can be expressed by the formula:

$$\forall n \in S \,.\, \exists p \in \mathrm{Primes} \,.\, \exists q \in \mathrm{Primes} \,.\, p + q = n$$

As p and q vary in the same set Primes, we can also use the more compact writing:

$$\forall n \in S \,.\, \exists p, q \in \mathrm{Primes} \,.\, p + q = n$$

read: "for every n in S, there exist p and q in Primes such that $p + q = n$".

Everyone has a dream

Let $\mathrm{dreams}(p, d)$ mean that person p has dream d.

Every single person has some dream

$\forall p \in \mathrm{Persons} \,.\, \exists d \in \mathrm{Dreams} \,.\, \mathrm{dreams}(p, d)$

There is a single dream everyone has

$\exists d \in \mathrm{Dreams} \,.\, \forall p \in \mathrm{Persons} \,.\, \mathrm{dreams}(p, d)$

De Morgan's laws for quantifiers

When the operator not($\cdot$) is applied to a predicate starting with a quantifier, the following happen:

$$\text{not}(\forall x \,.\, P(x)) \quad \text{is equivalent to} \quad \exists x \,.\, \text{not}(P(x))$$

$$\text{not}(\exists x \,.\, P(x)) \quad \text{is equivalent to} \quad \forall x \,.\, \text{not}(P(x))$$

Validity for predicate formulas

Intuitively, a predicate formula is valid if it is evaluated as true:

- no matter what the *domain* of the discourse is,
- no matter what the *type* of the variables are, and
- no matter what *interpretation* of its predicates is given.

This is *much harder* to formalize, and to verify, than validity of propositional formulas.

A valid predicate formula

Theorem

The following predicate formula is valid:

$$(\exists x \,.\, \forall y \,.\, P(x, y)) \text{ implies } (\forall y \,.\, \exists x \,.\, P(x, y))$$

Note the analogy with our "everyone has a dream" example:

If there is a single dream that every person has,
then every single person has some dream.

A valid predicate formula

Theorem

The following predicate formula is valid:

$$(\exists x \,.\, \forall y \,.\, P(x,y)) \text{ implies } (\forall y \,.\, \exists x \,.\, P(x,y))$$

Proof:

- If x varies in D and y varies in H, the formula becomes:

$$(\exists x \in D \,.\, \forall y \in H \,.\, P(x,y)) \text{ implies } (\forall y \in H \,.\, \exists x \in D \,.\, P(x,y))$$

- Suppose $\exists x \in D \,.\, \forall y \in H \,.\, P(x,y)$ is true:
  We want to show that $\forall y \in H \,.\, \exists x \in D \,.\, P(x,y)$ is also true.
- Take $x_0 \in D$ such that $\forall y \in H \,.\, P(x_0,y)$ is true.
- If we are given $y \in H$, we can always find $x \in D$ such that $P(x,y)$ is true, simply by choosing $x = x_0$.
- Then $\forall y \in H \,.\, \exists x \in D \,.\, P(x,y)$ is true, as we wanted.
- As the argument does not depend on the specific domain, types, and interpretation, it always works, and the predicate formula is valid.

Counter-models

Definition

Let $\phi(x_1, \ldots, x_n)$ be a predicative formula depending on the n variables x_i.
A *counter-model* for ϕ is a choice of:

- a domain D,
- types S_i for the variables x_i, and
- interpretations in D for the predicates occurring in ϕ

that make ϕ *false*.

Counter-models

Definition

Let $\phi(x_1,\ldots,x_n)$ be a predicative formula depending on the n variables x_i.
A *counter-model* for ϕ is a choice of:

- a domain D,
- types S_i for the variables x_i, and
- interpretations in D for the predicates occurring in ϕ

that make ϕ *false*.

Counter-models are at least as important as models, because they allow to *disprove implications*:

- Let P and Q be predicate formulas.
- Suppose that you want to prove that the predicate P implies Q is not valid.
- You can do so by choosing a domain, types for the variables, and interpretations which make P true and Q false.

A predicate formula with a counter-model

The following predicate formula is obtained from the one of two slides ago, swapping antecedent with consequent:

$$(\forall y\,.\,\exists x\,.\,P(x,y))\ \text{implies}\ (\exists x\,.\,\forall y\,.\,P(x,y))$$

The following is a counter-model for the formula above:

- Domain: the arithmetics of natural numbers.
- Type of the variables: natural numbers.
- Interpretation of $P(x,y)$: $x > y$.

In this counter-model, the formula means:

"if for every natural number there is a larger natural number,
then there is a natural number which is larger than every natural number"

which is clearly false.

A counter-model from Euclidean geometry

Consider the predicate formula:

$$\forall v, x, y, z \, . \, (T(v,x) \wedge T(v,y) \wedge T(v,z) \longrightarrow C(x,y) \vee C(x,z) \vee C(y,z))$$

A counter-model from Euclidean geometry

Consider the predicate formula:

$$\forall v, x, y, z \,.\, (T(v,x) \wedge T(v,y) \wedge T(v,z) \longrightarrow C(x,y) \vee C(x,z) \vee C(y,z))$$

We construct a counter-model as follows:

- As our domain, we choose Euclidean plane geometry.
- As types for variables, we make v be a straight line, and x, y, z be points.
- As interpretation for the predicates, we read $T(v,x)$ as "the straight line v goes *through* point x", and $C(x,y)$ as "the points x and y *coincide*".

A counter-model from Euclidean geometry

Consider the predicate formula:

$$\forall v, x, y, z \,.\, (T(v,x) \wedge T(v,y) \wedge T(v,z) \longrightarrow C(x,y) \vee C(x,z) \vee C(y,z))$$

We construct a counter-model as follows:

- As our domain, we choose Euclidean plane geometry.
- As types for variables, we make v be a straight line, and x, y, z be points.
- As interpretation for the predicates, we read $T(v,x)$ as "the straight line v goes *through* point x", and $C(x,y)$ as "the points x and y *coincide*".

Then the formula above is interpreted as:

"if a line of the Euclidean plane goes through three points,
then two of those three points coincide"

which is false.

. . . and a model too!

Consider again the predicate formula:

$$\forall v, x, y, z\,.\,(T(v,x) \wedge T(v,y) \wedge T(v,z) \longrightarrow C(x,y) \vee C(x,z) \vee C(y,z))$$

...and a model too!

Consider again the predicate formula:

$$\forall v, x, y, z \,.\, (T(v,x) \wedge T(v,y) \wedge T(v,z) \longrightarrow C(x,y) \vee C(x,z) \vee C(y,z))$$

We construct a model as follows:

- Domain: a *cube*.
- Variable types: v is an edge, and x, y, z are vertices.
- Interpretation: we read $T(v,x)$ as "the edge v *touches* the vertex x", and $E(x,y)$ as "the vertices x and y *coincide*".

. . . and a model too!

Consider again the predicate formula:

$$\forall v, x, y, z . (T(v,x) \wedge T(v,y) \wedge T(v,z) \longrightarrow C(x,y) \vee C(x,z) \vee C(y,z))$$

We construct a model as follows:

- Domain: a *cube*.
- Variable types: v is an edge, and x, y, z are vertices.
- Interpretation: we read $T(v,x)$ as "the edge v *touches* the vertex x", and $E(x,y)$ as "the vertices x and y *coincide*".

Then the formula above is interpreted as:

"if an edge of a cube touches three vertices,
then two of those three vertices coincide"

which is true.