Concepts of logic and computation

Software Analysis
Topic 2

Carlo A. Furia

USI – Università della Svizzera Italiana

Today’s menu

Logic

- Propositional logic
- Predicate logic
- First-order theories

Computation

- Computational models
- Computability
- Complexity

Logic

What is logic?

(Mathematical/formal) logic is a rigorous language to:

- express properties of objects
- derive other properties by calculation

What is logic?

(Mathematical/formal) logic is a rigorous language to:

- express properties of objects
- derive other properties by calculation

Different flavors of logic exist. We consider two widely used notations:

- propositional logic
- predicate logic

Syntax and semantics

Defining a logic requires to describe the logic's

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

if (C) T **else** E

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

```
if (C) T else E
```

- C is a Boolean expression

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

```
if (C) T else E
```

- *C* is a Boolean expression
- *T* and *E* are statements

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if` (C) T **`else`** E

- C is a Boolean expression
- T and E are statements
- the **`else`** part can be omitted

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if (`C`)` T `else` E

- C is a Boolean expression
- T and E are statements
- the `else` part can be omitted
- white spaces can be added anywhere

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

if (C) T **else** E

- C is a Boolean expression
- T and E are statements
- the **else** part can be omitted
- white spaces can be added anywhere

Semantics of Java conditionals:

if (C) T **else** E

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if` (C) T **`else`** E

- C is a Boolean expression
- T and E are statements
- the **`else`** part can be omitted
- white spaces can be added anywhere

Semantics of Java conditionals:

`if` (C) T **`else`** E

1. Evaluate C in the current state

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if (`C`)` T `else` E

- C is a Boolean expression
- T and E are statements
- the `else` part can be omitted
- white spaces can be added anywhere

Semantics of Java conditionals:

`if (`C`)` T `else` E

1. Evaluate C in the current state
2. If C is `true`, execute T

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if` `(`C`)` T **`else`** E

- C is a Boolean expression
- T and E are statements
- the **`else`** part can be omitted
- white spaces can be added anywhere

Semantics of Java conditionals:

`if` `(`C`)` T **`else`** E

1. Evaluate C in the current state
2. If C is **`true`**, execute T
3. If C is **`false`**, execute E

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if (`C`)` T `else` E

- C is a Boolean expression
- T and E are statements
- the `else` part can be omitted
- white spaces can be added anywhere

Semantics of Java conditionals:

`if (`C`)` T `else` E

1. Evaluate C in the current state
2. If C is `true`, execute T
3. If C is `false`, execute E
4. Continue execution with the next statement after the conditional

Syntax and semantics

Defining a logic requires to describe the logic's

called formulas

syntax: what expressions are well-formed
(can be written in the logic)

semantics: what value each well-formed expression has

Syntax of Java conditionals:

`if` `(`C`)` T **`else`** E

- C is a Boolean expression
- T and E are statements
- the **`else`** part can be omitted
- white spaces can be added anywhere

Semantics of Java conditionals:

`if` `(`C`)` T **`else`** E

1. Evaluate C in the current state
2. If C is **`true`**, execute T
3. If C is **`false`**, execute E
4. Continue execution with the next statement after the conditional
5. If evaluating C throws an exception...

Logic

Propositional logic

Syntax of propositional logic

Formulas of propositional logic are built out of:

constants $\top$ (true) and $\bot$ (false)

propositions (propositional letters): alphanumeric identifiers

connectives (operators): not $\neg$, and $\wedge$, or $\vee$, implies $\Longrightarrow$, iff $\Longleftrightarrow$

parentheses to set the application order of multiple connectives

Syntax of propositional logic

Formulas of propositional logic are built out of:

constants $\top$ (true) and $\bot$ (false)

propositions (propositional letters): alphanumeric identifiers

connectives (operators): not $\neg$, and $\wedge$, or $\vee$, implies $\implies$, iff $\iff$

parentheses to set the application order of multiple connectives

arity

	# ARGUMENTS	OTHER NAMES
$\neg$	1	negation, complement
$\wedge$	2	conjunction, product
$\vee$	2	disjunction, sum
$\implies$	2	implication
$\iff$	2	if and only if, co-implication, double implication

Syntax of propositional logic: formally

- $\top$ and $\bot$ are well-formed formulas
- If L is a propositional letter, then L is a well-formed formula
- If A is a well-formed formula, then $\neg A$ is a well-formed formula
- If A and B are well-formed formulas, then $A \land B$, $A \lor B$, $A \Longrightarrow B$, and $A \Longleftrightarrow B$ are well-formed formulas
- If A is a well-formed formula, then (A) is a well-formed formula

Syntax of propositional logic: formally

- $\top$ and $\bot$ are well-formed formulas
- If L is a propositional letter, then L is a well-formed formula
- If A is a well-formed formula, then $\neg A$ is a well-formed formula
- If A and B are well-formed formulas, then $A \wedge B$, $A \vee B$, $A \Longrightarrow B$, and $A \Longleftrightarrow B$ are well-formed formulas
- If A is a well-formed formula, then (A) is a well-formed formula

Using a BNF-like formalism:

$$F \ ::= \ \top \mid \bot \mid L \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 \mid F_1 \Longleftrightarrow F_2 \mid (\, F \,)$$

Syntax of propositional logic: formally

- $\top$ and $\bot$ are well-formed formulas
- If L is a propositional letter, then L is a well-formed formula
- If A is a well-formed formula, then $\neg A$ is a well-formed formula
- If A and B are well-formed formulas, then $A \wedge B$, $A \vee B$, $A \Longrightarrow B$, and $A \Longleftrightarrow B$ are well-formed formulas
- If A is a well-formed formula, then (A) is a well-formed formula

Using a BNF-like formalism:

$$F \ ::= \ \top \ | \ \bot \ | \ L \ | \ \neg F \ | \ F_1 \wedge F_2 \ | \ F_1 \vee F_2 \ | \ F_1 \Longrightarrow F_2 \ | \ F_1 \Longleftrightarrow F_2 \ | \ (\ F\ )$$

Even when not using a formalism, it's important definitions are rigorous (unambiguous).

Syntax of propositional logic: example

rain

Syntax of propositional logic: example

$$rain$$

$$rain \implies umbrella$$

Syntax of propositional logic: example

$$\textit{rain}$$

$$\textit{rain} \implies \textit{umbrella}$$

$$\textit{umbrella} \vee \textit{shine}$$

Syntax of propositional logic: example

$$rain$$

$$rain \implies umbrella$$

$$umbrella \lor shine$$

$$correct\_output \land termination$$

Operator precedence

Connectives have different binding power, from stronger to weaker:

1. $\neg$
2. $\wedge$
3. $\vee$
4. $\implies$
5. $\iff$

Operator precedence

Connectives have different binding power, from stronger to weaker:

1. $\neg$
2. $\wedge$
3. $\vee$
4. $\Longrightarrow$
5. $\Longleftrightarrow$

$$A \wedge \neg B \vee \neg C \wedge D \Longrightarrow \neg E \vee F \wedge G \Longleftrightarrow H \Longrightarrow J \vee \neg K \wedge L$$

is the same as

Operator precedence

Connectives have different binding power, from stronger to weaker:

1. $\neg$
2. $\wedge$
3. $\vee$
4. $\implies$
5. $\iff$

$$A \wedge \neg B \vee \neg C \wedge D \implies \neg E \vee F \wedge G \iff H \implies J \vee \neg K \wedge L$$

is the same as

$$(((A\wedge(\neg B))\vee((\neg C)\wedge D)) \implies ((\neg E)\vee(F\wedge G))) \iff (H \implies (J\vee((\neg K)\wedge L)))$$

Operator precedence

Connectives have different binding power, from stronger to weaker:

1. $\neg$
2. $\wedge$
3. $\vee$
4. $\Longrightarrow$
5. $\Longleftrightarrow$

$$A \wedge \neg B \vee \neg C \wedge D \Longrightarrow \neg E \vee F \wedge G \Longleftrightarrow H \Longrightarrow J \vee \neg K \wedge L$$

is the same as

$$(((A \wedge (\neg B)) \vee ((\neg C) \wedge D)) \Longrightarrow ((\neg E) \vee (F \wedge G))) \Longleftrightarrow (H \Longrightarrow (J \vee ((\neg K) \wedge L)))$$

In practice, we use parentheses even if they are not necessary if they help readability.

Interpretations in propositional logic

also: model

In general, the semantics (meaning) of a formula depends on whether each proposition's truth value – whether it is true or false.

An interpretation $\mathcal{M}$ of a propositional logic formula F
is an assignment of a value $\top$ (true) or $\perp$ (false)
to every proposition in F.

Interpretations in propositional logic

also: model

In general, the semantics (meaning) of a formula depends on whether each proposition's truth value – whether it is true or false.

An interpretation $\mathcal{M}$ of a propositional logic formula F
is an assignment of a value $\top$ (true) or $\bot$ (false)
to every proposition in F.

$[\![A]\!]_{\mathcal{M}} \in \{\top, \bot\}$ denotes the truth value of proposition A under $\mathcal{M}$

Semantics of propositional logic

The semantics of a formula F is its truth value under a given interpretation.

$$\mathcal{M} \models F \quad \text{means that } F \text{ is } \top \text{ under interpretation } \mathcal{M}$$

$$\mathcal{M} \not\models F \quad \text{means that } F \text{ is } \bot \text{ under interpretation } \mathcal{M}$$

Semantics of propositional logic

The semantics of a formula F is its truth value under a given interpretation.

$$\mathcal{M} \models F \quad \text{means that } F \text{ is } \top \text{ under interpretation } \mathcal{M}$$

$$\mathcal{M} \not\models F \quad \text{means that } F \text{ is } \bot \text{ under interpretation } \mathcal{M}$$

The semantics of propositional logic formulas is defined inductively:

$$\begin{array}{lll}
\mathcal{M} \models \top & & \\
\mathcal{M} \not\models \bot & & \\
\mathcal{M} \models A & \text{iff} & [\![A]\!]_{\mathcal{M}} = \top \\
\mathcal{M} \models \neg F & \text{iff} & \mathcal{M} \not\models F \\
\mathcal{M} \models F_1 \land F_2 & \text{iff} & \mathcal{M} \models F_1 \text{ and } \mathcal{M} \models F_2 \\
\mathcal{M} \models F_1 \lor F_2 & \text{iff} & \mathcal{M} \models \neg(\neg F_1 \land \neg F_2) \\
\mathcal{M} \models F_1 \implies F_2 & \text{iff} & \mathcal{M} \models \neg F_1 \lor F_2 \\
\mathcal{M} \models F_1 \iff F_2 & \text{iff} & \mathcal{M} \models (F_1 \land F_2) \lor (\neg F_1 \land \neg F_2)
\end{array}$$

Semantics of propositional logic

The semantics of a formula F is its truth value under a given interpretation.

$$
\begin{array}{ll}
\mathcal{M} \models F & \text{means that } F \text{ is } \top \text{ under interpretation } \mathcal{M} \\
\mathcal{M} \not\models F & \text{means that } F \text{ is } \bot \text{ under interpretation } \mathcal{M}
\end{array}
$$

The semantics of propositional logic formulas is defined inductively:

conjunction in meta-language

$$
\begin{array}{lll}
\mathcal{M} \models \top & & \\
\mathcal{M} \not\models \bot & & \\
\mathcal{M} \models A & \text{iff} & [\![A]\!]_{\mathcal{M}} = \top \\
\mathcal{M} \models \neg F & \text{iff} & \mathcal{M} \not\models F \\
\mathcal{M} \models F_1 \wedge F_2 & \text{iff} & \mathcal{M} \models F_1 \text{ and } \mathcal{M} \models F_2 \\
\mathcal{M} \models F_1 \vee F_2 & \text{iff} & \mathcal{M} \models \neg(\neg F_1 \wedge \neg F_2) \\
\mathcal{M} \models F_1 \implies F_2 & \text{iff} & \mathcal{M} \models \neg F_1 \vee F_2 \\
\mathcal{M} \models F_1 \iff F_2 & \text{iff} & \mathcal{M} \models (F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2)
\end{array}
$$

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \Longrightarrow umbrella$	–	
$rain \Longrightarrow umbrella$	$umbrella$	
$rain \Longrightarrow umbrella$	$rain$	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, C	

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\perp$.

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \Longrightarrow umbrella$	–	✔
$rain \Longrightarrow umbrella$	$umbrella$	
$rain \Longrightarrow umbrella$	$rain$	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, C	

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\perp$.

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \Longrightarrow umbrella$	–	✔
$rain \Longrightarrow umbrella$	$umbrella$	✔
$rain \Longrightarrow umbrella$	$rain$	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, C	

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\perp$.

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \Longrightarrow umbrella$	–	✔
$rain \Longrightarrow umbrella$	$umbrella$	✔
$rain \Longrightarrow umbrella$	$rain$	✖
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, C	

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\perp$.

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \Longrightarrow umbrella$	–	✔
$rain \Longrightarrow umbrella$	$umbrella$	✔
$rain \Longrightarrow umbrella$	$rain$	✖
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, B, C	✖
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	B, C	
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, C	

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\perp$.

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \implies umbrella$	–	✔
$rain \implies umbrella$	$umbrella$	✔
$rain \implies umbrella$	$rain$	✖
$A \land B \lor C \implies D \iff A$	A, B, C	✖
$A \land B \lor C \implies D \iff A$	B, C	✔
$A \land B \lor C \implies D \iff A$	A, C	

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\perp$.

Semantics of propositional logic: examples

FORMULA F	$[\![\cdot]\!]_{\mathcal{M}} = \top$	$\mathcal{M} \overset{?}{\models} F$
$rain \Longrightarrow umbrella$	–	✔
$rain \Longrightarrow umbrella$	$umbrella$	✔
$rain \Longrightarrow umbrella$	$rain$	✖
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, B, C	✖
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	B, C	✔
$A \wedge B \vee C \Longrightarrow D \Longleftrightarrow A$	A, C	✖

In these examples, a proposition is $\top$ in $\mathcal{M}$ iff it is listed in the second column; otherwise it is $\bot$.

Validity

A formula F is valid when $\mathcal{M} \models F$
for every possible interpretation $\mathcal{M}$.

$\models F$ denotes that F is valid

A valid propositional formula is $\top$ entirely on the basis of its propositional structure.

A valid propositional formula is also called a tautology.

A formulas that is not valid is called invalid.

Validity and invalidity: examples

$$
\begin{aligned}
F_1 &\triangleq \quad A \Longrightarrow B \\
F_2 &\triangleq \quad A \lor \neg A \\
F_3 &\triangleq \quad A \Longleftrightarrow A \\
F_4 &\triangleq \quad (A \Longrightarrow B) \land A \Longrightarrow B \\
F_5 &\triangleq \quad (A \Longleftrightarrow B) \land A \Longleftrightarrow B
\end{aligned}
$$

Validity and invalidity: examples

$$
\begin{array}{lll}
F_1 \triangleq & A \Longrightarrow B & \text{invalid} \\
F_2 \triangleq & A \vee \neg A & \\
F_3 \triangleq & A \Longleftrightarrow A & \\
F_4 \triangleq & (A \Longrightarrow B) \wedge A \Longrightarrow B & \\
F_5 \triangleq & (A \Longleftrightarrow B) \wedge A \Longleftrightarrow B &
\end{array}
$$

Validity and invalidity: examples

$F_1 \triangleq$	$A \Longrightarrow B$	invalid
$F_2 \triangleq$	$A \vee \neg A$	valid
$F_3 \triangleq$	$A \Longleftrightarrow A$	
$F_4 \triangleq$	$(A \Longrightarrow B) \wedge A \Longrightarrow B$	
$F_5 \triangleq$	$(A \Longleftrightarrow B) \wedge A \Longleftrightarrow B$	

Validity and invalidity: examples

$F_1 \triangleq$	$A \Longrightarrow B$	invalid
$F_2 \triangleq$	$A \vee \neg A$	valid
$F_3 \triangleq$	$A \Longleftrightarrow A$	valid
$F_4 \triangleq$	$(A \Longrightarrow B) \wedge A \Longrightarrow B$	
$F_5 \triangleq$	$(A \Longleftrightarrow B) \wedge A \Longleftrightarrow B$	

Validity and invalidity: examples

$F_1 \triangleq$	$A \Longrightarrow B$	invalid
$F_2 \triangleq$	$A \vee \neg A$	valid
$F_3 \triangleq$	$A \Longleftrightarrow A$	valid
$F_4 \triangleq$	$(A \Longrightarrow B) \wedge A \Longrightarrow B$	valid
$F_5 \triangleq$	$(A \Longleftrightarrow B) \wedge A \Longleftrightarrow B$	

Validity and invalidity: examples

$$
\begin{array}{rlr}
F_1 \triangleq & A \Longrightarrow B & \text{invalid} \\
F_2 \triangleq & A \vee \neg A & \text{valid} \\
F_3 \triangleq & A \Longleftrightarrow A & \text{valid} \\
F_4 \triangleq & (A \Longrightarrow B) \wedge A \Longrightarrow B & \text{valid} \\
F_5 \triangleq & (A \Longleftrightarrow B) \wedge A \Longleftrightarrow B & \text{invalid}
\end{array}
$$

Validity and invalidity: examples

$F_1 \triangleq$	$A \Longrightarrow B$	invalid
$F_2 \triangleq$	$A \vee \neg A$	valid
$F_3 \triangleq$	$A \Longleftrightarrow A$	valid
$F_4 \triangleq$	$(A \Longrightarrow B) \wedge A \Longrightarrow B$	valid
$F_5 \triangleq$	$(A \Longleftrightarrow B) \wedge A \Longleftrightarrow B$	invalid

We can build a truth table to check for validity:

$[\![A]\!]_{\mathcal{M}}$	$[\![B]\!]_{\mathcal{M}}$	$\mathcal{M} \models F_1$	$\mathcal{M} \models F_2$	$\mathcal{M} \models F_3$	$\mathcal{M} \models F_4$	$\mathcal{M} \models F_5$
$\top$	$\top$	✔	✔	✔	✔	✔
$\top$	$\bot$	✘	✔	✔	✔	✔
$\bot$	$\top$	✔	✔	✔	✔	✘
$\bot$	$\bot$	✔	✔	✔	✔	✔

Satisfiability

A formula F is satsifiable when $\mathcal{M} \models F$
for some interpretation $\mathcal{M}$.

A formulas that is not satisfiable is called unsatisfiable.

An unsatisfiable propositional formula is also called a contradiction.

Satisfiability

A formula F is satsifiable when $\mathcal{M} \models F$
for some interpretation $\mathcal{M}$.

A formulas that is not satisfiable is called unsatisfiable.

An unsatisfiable propositional formula is also called a contradiction.

Satisfiability is the dual of validity:

F is valid	iff	$\neg F$ is unsatisfiable
F is satisfiable	iff	$\neg F$ is invalid

The importance of being valid

Validity (or its dual satisfiability) is the fundamental problem in logic.

Every analysis problem reduces to a validity checking problem.

S**:** formal model of the system behavior
(for example, program behavior)

P**:** property of system behavior to be analyzed

$\underline{S \text{ satisfies property } P}$ iff

$$S \implies P \quad \text{is valid}$$

Deductive systems

Establishing validity by building truth tables is tedious, and does not generalize to logics more expressive than propositional logic.

Instead, we can use deductive systems (proof systems) to calculate the consequences of some formulas.

Deductive systems

Establishing validity by building truth tables is tedious, and does not generalize to logics more expressive than propositional logic.

Instead, we can use deductive systems (proof systems) to calculate the consequences of some formulas.

A deductive system is made of proof rules:

premises

$$\frac{P_1 \quad P_2 \quad \cdots}{C}\ \textit{rule name}$$

conclusion or deduction

If we have established that P_1, P_2, ... are all true,
we can conclude that C is true as well.

Formal proofs

A proof is a sequence of applications of proof rules in deductive system that starts from S and leads to P:

$$S \vdash P$$

Read:

- There is a proof of P from S
- P follows from S
- P can be inferred from S
- P is a theorem under assumption S

$\vdash F$ denotes that F is a theorem provable with the deductive system.

Natural deduction for propositional logic

Natural deduction is an intuitively simple deductive system for propositional logic.

Here are <u>some</u> inference rules of natural deduction:

$$\frac{A \wedge B}{B} \qquad \wedge \text{ left-elimination}$$

$$\frac{A \wedge B}{A} \qquad \wedge \text{ right-elimination}$$

deduce B assuming A

$$\frac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \Longrightarrow B} \qquad \Longrightarrow \text{ introduction}$$

$$\frac{A \Longrightarrow B \quad A}{B} \qquad \Longrightarrow \text{ elimination (modus ponens)}$$

Example of natural deduction

Let us build a proof that $\vdash (A \Longrightarrow B) \wedge A \Longrightarrow B$.

Example of natural deduction

Let us build a proof that $\vdash (A \Longrightarrow B) \wedge A \Longrightarrow B$.

1	$(A \Longrightarrow B) \wedge A$	assumption
2	$A \Longrightarrow B$	$\wedge$ right-elimination from 1
3	A	$\wedge$ left-elimination from 1
4	B	$\Longrightarrow$ elimination from 2, 3
5	$(A \Longrightarrow B) \wedge A \Longrightarrow B$	$\Longrightarrow$ introduction from [1] 4
	QED	

Example of natural deduction

Let us build a proof that $\vdash (A \Longrightarrow B) \land A \Longrightarrow B$.

1	$(A \Longrightarrow B) \land A$	assumption
2	$A \Longrightarrow B$	$\land$ right-elimination from 1
3	A	$\land$ left-elimination from 1
4	B	$\Longrightarrow$ elimination from 2, 3
5	$(A \Longrightarrow B) \land A \Longrightarrow B$	$\Longrightarrow$ introduction from [1] 4
	QED	

In tree form:

$$\dfrac{\dfrac{\dfrac{[(A \Longrightarrow B) \land A]}{A \Longrightarrow B}\ \land\text{ right-elimination} \qquad \dfrac{[(A \Longrightarrow B) \land A]}{A}\ \land\text{ left-elimination}}{B}\ \Longrightarrow\text{ elimination}}{(A \Longrightarrow B) \land A \Longrightarrow B}\ \Longrightarrow\text{ introduction}$$

Soundness and completeness

A deductive system makes it possible to analyze semantics by syntactic means – that is by calculation (symbolic manipulation).

This is only possible if the proof rules:

- do not introduce inconsistencies (soundness)
- are applicable to every formula (completeness)

A deductive system is

sound if every theorem is valid: $\vdash F$ implies $\models F$

complete if every valid formula is a theorem: $\models F$ implies $\vdash F$

for every (well-formed) formula F.

Soundness and completeness

A deductive system makes it possible to analyze semantics by syntactic means – that is by calculation (symbolic manipulation).

This is only possible if the proof rules:

- do not introduce inconsistencies (soundness)
- are applicable to every formula (completeness)

A deductive system is

sound if every theorem is valid: $\vdash F$ implies $\models F$

complete if every valid formula is a theorem: $\models F$ implies $\vdash F$

for every (well-formed) formula F.

Natural deduction (completed with other proof rules) is a sound and complete deductive system for propositional logic.

To emphasize its calculational aspects, propositional logic is also called propositional calculus.

Logic

Predicate logic

From propositions to predicates

Propositional logic is a fundamental notation that underpins pretty much every other flavor of logic.

While it is already quite useful, its expressiveness is somewhat limited: there are properties and behaviors that we cannot encode in propositional logic without abstracting away a lot of information.

Predicate logic is much more expressive than propositional logic since it allows reasoning about infinite sets of objects.

Syntax of predicate logic

Formulas of predicate logic are built out of:

constants (constant symbols): $a, b, c, \ldots$ and other lowercase alphanumeric identifiers

variables (variable symbols): $t, u, v, \ldots$ and other lowercase alphanumeric identifiers

functions (function symbols): $f, g, h, \ldots$ and other lowercase alphanumeric identifiers

predicates (predicate symbols): $P, Q, R, \ldots$ and other capitalized alphanumeric identifiers

logic quantifiers forall $\forall$ (universal quantifier) and exists $\exists$ (existential quantifier)

connectives as in propositional logic

parentheses to set the application order of multiple connectives

Terms and formulas

We first define terms t:

$$t \ ::= \ \underbrace{c}_{\text{constant}} \mid \underbrace{v}_{\text{variable}} \mid \underbrace{f(t_1, \ldots, t_n)}_{n\text{-ary function}}$$

Since we omit parentheses in functions without arguments, a constant is the same as a nullary (argumentless) function.

Then we define formulas F:

$$\begin{aligned} F \ ::= \ & \top \mid \bot \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 \mid F_1 \Longleftrightarrow F_2 \mid (\, F \,) \\ \mid \ & P(t_1, \ldots, t_n) \mid \forall t, u, \ldots \bullet F \mid \exists t, u, \ldots \bullet F \end{aligned}$$

Again, we omit parentheses in predicates without arguments, so that P is a nullary predicate or, equivalently, a proposition.

Syntax of quantified formulas

$$\begin{aligned} F ::= &\ \forall t, u, \ldots \bullet F \\ | &\ \exists t, u, \ldots \bullet F \end{aligned}$$

Syntax of quantified formulas

scope (may refer to quantified variables)

$$
\begin{aligned}
F ::= \;& \forall t, u, \ldots \bullet F \\
\mid \;& \exists t, u, \ldots \bullet F
\end{aligned}
$$

quantified variables

may be omitted

Syntax of quantified formulas

scope (may refer to quantified variables)

$$
\begin{aligned}
F ::= {} & \forall t, u, \ldots \bullet F \\
\mid {} & \exists t, u, \ldots \bullet F
\end{aligned}
$$

quantified variables

may be omitted

The binding power of $\bullet$ is weaker than that of $\Longrightarrow$ and stronger than that of $\Longleftrightarrow$:

$\forall x \bullet \neg P(x) \Longrightarrow Q(x)$ is equivalent to $\forall x \bullet (\neg P(x) \Longrightarrow Q(x))$

When we omit $\bullet$, a quantifier's binding power is the same as that of $\neg$:

$\forall x \neg P(x) \Longrightarrow Q(x)$ is equivalent to $(\forall x\, (\neg P(x))) \Longrightarrow Q(x)$

First-order quantification

Predicate logic can only quantify on variables – not on functions or predicates. Thus, its quantification is called first order.

First-order logic is another name for predicate logic.

FIRST-ORDER QUANTIFICATION	HIGHER-ORDER QUANTIFICATION
$\forall n \bullet P(n, s(n))$	$\forall P \bullet P(a, b)$

First-order quantification

Predicate logic can only quantify on variables – not on functions or predicates. Thus, its quantification is called first order.

First-order logic is another name for predicate logic.

FIRST-ORDER QUANTIFICATION	HIGHER-ORDER QUANTIFICATION
$\forall n \bullet P(n, s(n))$	$\forall P \bullet P(a, b)$

Example of higher-order quantification formula:
Leibniz's definition of equality:

$$x = y \quad \Longleftrightarrow \quad \forall P \bullet (P(x) \Longleftrightarrow P(y))$$

Two items are equal iff they have exactly the same properties.

Syntax of predicate logic: example

$$\mathit{Rain} \implies \mathit{Umbrella}$$

Syntax of predicate logic: example

$$\mathit{Rain} \implies \mathit{Umbrella}$$

$$\mathit{Person}(p) \implies \mathit{Mortal}(p)$$

Syntax of predicate logic: example

$$\mathit{Rain} \implies \mathit{Umbrella}$$

$$\mathit{Person}(p) \implies \mathit{Mortal}(p)$$

$$\mathit{Person}(\mathit{socrates})$$

Syntax of predicate logic: example

$$Rain \implies Umbrella$$

$$Person(p) \implies Mortal(p)$$

$$Person(socrates)$$

$$\forall p \bullet Person(p) \implies Mortal(p)$$

Syntax of predicate logic: example

$$Rain \implies Umbrella$$

$$Person(p) \implies Mortal(p)$$

$$Person(socrates)$$

$$\forall p \bullet Person(p) \implies Mortal(p)$$

$$\forall n \bullet \exists m \bullet Greater(m, succ(n))$$

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$$P(x)$$

$$\forall x \bullet P(x)$$

$$\forall x \bullet P(y)$$

$$R(z) \wedge \exists x \bullet \forall y \bullet Q(f(x), g(y), z)$$

$$R(c) \wedge \exists x \bullet \forall y \bullet Q(f(x), g(y), c)$$

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$$
\begin{aligned}
P(x) &\quad \textit{open} \\
\forall x \bullet P(x) & \\
\forall x \bullet P(y) & \\
R(z) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), z) & \\
R(c) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), c) &
\end{aligned}
$$

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$$
\begin{array}{rl}
P(x) & \textit{open} \\
\forall x \bullet P(x) & \textit{closed} \\
\forall x \bullet P(y) & \\
R(z) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), z) & \\
R(c) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), c) &
\end{array}
$$

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$$
\begin{aligned}
P(x) &\quad \textit{open} \\
\forall x \bullet P(x) &\quad \textit{closed} \\
\forall x \bullet P(y) &\quad \textit{open} \\
R(z) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), z) & \\
R(c) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), c) &
\end{aligned}
$$

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$P(x)$	*open*
$\forall x \bullet P(x)$	*closed*
$\forall x \bullet P(y)$	*open*
$R(z) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), z)$	*open*
$R(c) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), c)$	

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$P(x)$	*open*
$\forall x \bullet P(x)$	*closed*
$\forall x \bullet P(y)$	*open*
$R(z) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), z)$	*open*
$R(c) \land \exists x \bullet \forall y \bullet Q(f(x), g(y), c)$	*closed*

Bound and unbound variables

A quantified variable is bound by the quantifier.

A variable that is not bound is called free.

A formula is closed if all its variables are bound.
A formula is open if it's not closed.
A closed formula is also called a sentence.

$P(x)$	*open*
$\forall x \bullet P(x)$	*closed*
$\forall x \bullet P(y)$	*open*
$R(z) \wedge \exists x \bullet \forall y \bullet Q(f(x), g(y), z)$	*open*
$R(c) \wedge \exists x \bullet \forall y \bullet Q(f(x), g(y), c)$	*closed*

z is a variable

c is a constant

Interpretations in predicate logic

also: model

An interpretation $\mathcal{M}$ of a predicate logic formula F assigns:

- a value to every constant and unbound variable in F
- a concrete function to every function in F
- a concrete relation to every predicate in F

usually, total

Interpretations in predicate logic

also: model

An interpretation $\mathcal{M}$ of a predicate logic formula F assigns:

- a value to every constant and unbound variable in F
- a concrete function to every function in F
- a concrete relation to every predicate in F

usually, total

$[\![S]\!]_{\mathcal{M}}$ denotes the value/function/relation given to symbol S by $\mathcal{M}$

Semantics of predicate logic

The semantics of a formula F is its truth value under a given interpretation.

The semantics of predicate logic formulas is defined inductively. The rules of propositional logic still hold; in addition, we define the semantics of quantifiers.

Semantics of predicate logic

The semantics of a formula F is its truth value under a given interpretation.

The semantics of predicate logic formulas is defined inductively. The rules of propositional logic still hold; in addition, we define the semantics of quantifiers.

Write $\mathcal{M} \equiv_v \mathcal{M}'$ to mean that interpretations $\mathcal{M}$ and $\mathcal{M}'$ are the same except for possibly the value given to variable v.

Semantics of predicate logic

The semantics of a formula F is its truth value under a given interpretation.

The semantics of predicate logic formulas is defined inductively. The rules of propositional logic still hold; in addition, we define the semantics of quantifiers.

Write $\mathcal{M} \equiv_v \mathcal{M}'$ to mean that interpretations $\mathcal{M}$ and $\mathcal{M}'$ are the same except for possibly the value given to variable v.

$$
\begin{array}{lll}
\mathcal{M} \models \forall v \bullet F & \text{iff} & \mathcal{M}' \models F \text{ for every interpretation } \mathcal{M}' \equiv_v \mathcal{M} \\
\mathcal{M} \models \exists v \bullet F & \text{iff} & \mathcal{M}' \models F \text{ for some interpretation } \mathcal{M}' \equiv_v \mathcal{M}
\end{array}
$$

Semantics of predicate logic: examples

$$F_1 \; = \; \forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$$

Semantics of predicate logic: examples

$$F_1 \;=\; \forall x \bullet Equal(x, plus(x, one))$$

$\mathcal{M} \not\models F_1$ under an interpretation where:

- x is from a numeric domain
- *Equal* is equality $=$
- *plus* is addition $+$
- *one* is the constant 1

$\mathcal{M} \models F_1$ under an interpretation where:

- *Equal* is equality $=$
- *plus* returns its first argument

Semantics of predicate logic: examples

$$F_1 \ = \ \forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$$

$\mathcal{M} \not\models F_1$ under an interpretation where:

- x is from a numeric domain
- *Equal* is equality $=$
- *plus* is addition $+$
- *one* is the constant 1

$\mathcal{M} \models F_1$ under an interpretation where:

- *Equal* is equality $=$
- *plus* returns its first argument

$$F_2 \ = \ (\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$$

Semantics of predicate logic: examples

$$F_1 \;=\; \forall x \bullet Equal(x, plus(x, one))$$

$\mathcal{M} \not\models F_1$ under an interpretation where:

- x is from a numeric domain
- *Equal* is equality $=$
- *plus* is addition $+$
- *one* is the constant 1

$\mathcal{M} \models F_1$ under an interpretation where:

- *Equal* is equality $=$
- *plus* returns its first argument

$$F_2 \;=\; (\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$$

$\mathcal{M} \models F_2$ in every interpretation
(it can be proved from the definition of semantics)

Validity and satisfiability

The definitions of validity and satisfiability for propositional logic also apply to predicate logic.

A formula F is valid when $\mathcal{M} \models F$
for every possible interpretation $\mathcal{M}$.

A formula F is satisfiable when $\mathcal{M} \models F$
for some possible interpretation $\mathcal{M}$.

Validity and satisfiability: examples

$$
\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one})) \\
F_2 &\triangleq \quad (\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x)) \\
F_3 &\triangleq \quad \forall x \bullet P(x) \lor \neg P(x) \\
F_4 &\triangleq \quad \forall x \bullet \exists y \bullet P(x) \implies P(y) \\
F_5 &\triangleq \quad \exists x \bullet P(f(x)) \land \neg P(f(x))
\end{aligned}
$$

$$
F_6 \triangleq \quad \forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)
$$

Validity and satisfiability: examples

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet Equal(x, plus(x, one)) && \text{invalid, satisfiable} \\
F_2 &\triangleq (\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x)) \\
F_3 &\triangleq \forall x \bullet P(x) \vee \neg P(x) \\
F_4 &\triangleq \forall x \bullet \exists y \bullet P(x) \implies P(y) \\
F_5 &\triangleq \exists x \bullet P(f(x)) \wedge \neg P(f(x))
\end{aligned}
$$

$$
F_6 \triangleq \forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)
$$

Validity and satisfiability: examples

$F_1 \triangleq$	$\forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$	invalid, satisfiable
$F_2 \triangleq$	$(\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$	valid
$F_3 \triangleq$	$\forall x \bullet P(x) \vee \neg P(x)$	
$F_4 \triangleq$	$\forall x \bullet \exists y \bullet P(x) \implies P(y)$	
$F_5 \triangleq$	$\exists x \bullet P(f(x)) \wedge \neg P(f(x))$	
$F_6 \triangleq$	$\forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)$	

Validity and satisfiability: examples

$F_1 \triangleq$	$\forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$	invalid, satisfiable
$F_2 \triangleq$	$(\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$	valid
$F_3 \triangleq$	$\forall x \bullet P(x) \vee \neg P(x)$	valid
$F_4 \triangleq$	$\forall x \bullet \exists y \bullet P(x) \implies P(y)$	
$F_5 \triangleq$	$\exists x \bullet P(f(x)) \wedge \neg P(f(x))$	
$F_6 \triangleq$	$\forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)$	

Validity and satisfiability: examples

$F_1 \triangleq$	$\forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$	invalid, satisfiable
$F_2 \triangleq$	$(\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$	valid
$F_3 \triangleq$	$\forall x \bullet P(x) \vee \neg P(x)$	valid
$F_4 \triangleq$	$\forall x \bullet \exists y \bullet P(x) \implies P(y)$	valid
$F_5 \triangleq$	$\exists x \bullet P(f(x)) \wedge \neg P(f(x))$	

$$F_6 \triangleq \quad \forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)$$

Validity and satisfiability: examples

$F_1 \triangleq$	$\forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$	invalid, satisfiable
$F_2 \triangleq$	$(\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$	valid
$F_3 \triangleq$	$\forall x \bullet P(x) \vee \neg P(x)$	valid
$F_4 \triangleq$	$\forall x \bullet \exists y \bullet P(x) \implies P(y)$	valid
$F_5 \triangleq$	$\exists x \bullet P(f(x)) \wedge \neg P(f(x))$	unsatisfiable

$$F_6 \triangleq \quad \forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)$$

Validity and satisfiability: examples

$F_1 \triangleq$	$\forall x \bullet Equal(x, plus(x, one))$	invalid, satisfiable
$F_2 \triangleq$	$(\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$	valid
$F_3 \triangleq$	$\forall x \bullet P(x) \vee \neg P(x)$	valid
$F_4 \triangleq$	$\forall x \bullet \exists y \bullet P(x) \implies P(y)$	valid
$F_5 \triangleq$	$\exists x \bullet P(f(x)) \wedge \neg P(f(x))$	unsatisfiable
$F_6 \triangleq$	$\forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)$	valid

Validity and satisfiability: examples

$F_1 \triangleq$	$\forall x \bullet \mathit{Equal}(x, \mathit{plus}(x, \mathit{one}))$	invalid, satisfiable
$F_2 \triangleq$	$(\forall x \bullet P(x)) \Longrightarrow (\exists x \bullet P(x))$	valid
$F_3 \triangleq$	$\forall x \bullet P(x) \lor \neg P(x)$	valid
$F_4 \triangleq$	$\forall x \bullet \exists y \bullet P(x) \implies P(y)$	valid
$F_5 \triangleq$	$\exists x \bullet P(f(x)) \land \neg P(f(x))$	unsatisfiable
$F_6 \triangleq$	$\forall x \bullet P(x) \iff \neg \exists x \bullet \neg P(x)$	valid

F_6 expresses the duality between universal and existential quantification.

Validity and satisfiability: examples

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing

every free occurrence of variable x in F with term t.

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \land Q(x)\big) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq \quad S(x) \land \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}$$

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \land Q(x)\big) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq \quad S(x) \land \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}$$

$$\begin{aligned}
&F_1[x \mapsto f(c)] \triangleq \\
&F_2[x \mapsto f(c)] \triangleq \\
&F_3[y \mapsto f(c)] \triangleq \\
&F_4[x \mapsto f(c)] \triangleq \\
&F_5[x \mapsto f(c)] \triangleq
\end{aligned}$$

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing

every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \land Q(x)\big) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq \quad S(x) \land \forall y \bullet \big(P(x) \implies Q(y)\big)
\end{aligned}$$

$$\begin{aligned}
F_1[x \mapsto f(c)] &\triangleq \quad \forall x \bullet P(x) \\
F_2[x \mapsto f(c)] &\triangleq \\
F_3[y \mapsto f(c)] &\triangleq \\
F_4[x \mapsto f(c)] &\triangleq \\
F_5[x \mapsto f(c)] &\triangleq
\end{aligned}$$

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \land Q(x)\big) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq \quad S(x) \land \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}$$

$$\begin{aligned}
F_1[x \mapsto f(c)] &\triangleq \quad \forall x \bullet P(x) \\
F_2[x \mapsto f(c)] &\triangleq \quad \forall y \bullet P(f(c)) \\
F_3[y \mapsto f(c)] &\triangleq \\
F_4[x \mapsto f(c)] &\triangleq \\
F_5[x \mapsto f(c)] &\triangleq
\end{aligned}$$

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \wedge \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \wedge Q(x)\big) \implies \neg P(x) \vee Q(y) \\
F_5 &\triangleq \quad S(x) \wedge \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}$$

$$\begin{aligned}
F_1[x \mapsto f(c)] &\triangleq \quad \forall x \bullet P(x) \\
F_2[x \mapsto f(c)] &\triangleq \quad \forall y \bullet P(f(c)) \\
F_3[y \mapsto f(c)] &\triangleq \quad \exists x \bullet P(f(c)) \wedge \forall y \bullet Q(y) \\
F_4[x \mapsto f(c)] &\triangleq \\
F_5[x \mapsto f(c)] &\triangleq
\end{aligned}$$

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing

every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \land Q(x)\big) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq \quad S(x) \land \forall y \bullet \big(P(x) \implies Q(y)\big)
\end{aligned}$$

$$\begin{aligned}
F_1[x \mapsto f(c)] &\triangleq \quad \forall x \bullet P(x) \\
F_2[x \mapsto f(c)] &\triangleq \quad \forall y \bullet P(f(c)) \\
F_3[y \mapsto f(c)] &\triangleq \quad \exists x \bullet P(f(c)) \land \forall y \bullet Q(y) \\
F_4[x \mapsto f(c)] &\triangleq \quad \big(\forall x \bullet P(x) \land Q(x)\big) \implies \neg P(f(c)) \lor Q(y) \\
F_5[x \mapsto f(c)] &\triangleq
\end{aligned}$$

Substitutions

To reason deductively about first-order formulas, we often need to perform substitutions of variables for other terms.

$$F[x \mapsto t]$$

is the formula obtained by replacing

every free occurrence of variable x in F with term t.

$$\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \wedge \forall y \bullet Q(y) \\
F_4 &\triangleq \quad \big(\forall x \bullet P(x) \wedge Q(x)\big) \implies \neg P(x) \vee Q(y) \\
F_5 &\triangleq \quad S(x) \wedge \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}$$

$$\begin{aligned}
F_1[x \mapsto f(c)] &\triangleq \quad \forall x \bullet P(x) \\
F_2[x \mapsto f(c)] &\triangleq \quad \forall y \bullet P(f(c)) \\
F_3[y \mapsto f(c)] &\triangleq \quad \exists x \bullet P(f(c)) \wedge \forall y \bullet Q(y) \\
F_4[x \mapsto f(c)] &\triangleq \quad \big(\forall x \bullet P(x) \wedge Q(x)\big) \implies \neg P(f(c)) \vee Q(y) \\
F_5[x \mapsto f(c)] &\triangleq \quad S(f(c)) \wedge \forall y \bullet \big(P(f(c)) \Longrightarrow Q(y)\big)
\end{aligned}$$

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \quad \forall x \bullet P(x) \\
F_2 &\triangleq \quad \forall y \bullet P(x) \\
F_3 &\triangleq \quad \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq \quad (\forall x \bullet P(x) \land Q(x)) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq \quad S(x) \land \forall y \bullet (P(x) \implies Q(y))
\end{aligned}
$$

- $f(x)$ is for x in F_1
- $f(x)$ is for x in F_2
- $f(y)$ is for x in F_2
- $f(y)$ is for y in F_3
- $f(y, y)$ is for x in F_4
- $f(y, y)$ is for x in F_5

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet P(x) \\
F_2 &\triangleq \forall y \bullet P(x) \\
F_3 &\triangleq \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq (\forall x \bullet P(x) \land Q(x)) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq S(x) \land \forall y \bullet \big(P(x) \implies Q(y)\big)
\end{aligned}
$$

- $f(x)$ is free for x in F_1, because x is not free in F_1
- $f(x)$ is for x in F_2
- $f(y)$ is for x in F_2
- $f(y)$ is for y in F_3
- $f(y, y)$ is for x in F_4
- $f(y, y)$ is for x in F_5

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet P(x) \\
F_2 &\triangleq \forall y \bullet P(x) \\
F_3 &\triangleq \exists x \bullet P(y) \wedge \forall y \bullet Q(y) \\
F_4 &\triangleq (\forall x \bullet P(x) \wedge Q(x)) \implies \neg P(x) \vee Q(y) \\
F_5 &\triangleq S(x) \wedge \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}
$$

- $f(x)$ is free for x in F_1, because x is not free in F_1
- $f(x)$ is free for x in F_2, because x is not quantified in F_2
- $f(y)$ is for x in F_2
- $f(y)$ is for y in F_3
- $f(y, y)$ is for x in F_4
- $f(y, y)$ is for x in F_5

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet P(x) \\
F_2 &\triangleq \forall y \bullet P(x) \\
F_3 &\triangleq \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq (\forall x \bullet P(x) \land Q(x)) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq S(x) \land \forall y \bullet (P(x) \implies Q(y))
\end{aligned}
$$

- $f(x)$ is free for x in F_1, because x is not free in F_1
- $f(x)$ is free for x in F_2, because x is not quantified in F_2
- $f(y)$ is not free for x in F_2, because y would become quantified
- $f(y)$ is for y in F_3
- $f(y, y)$ is for x in F_4
- $f(y, y)$ is for x in F_5

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet P(x) \\
F_2 &\triangleq \forall y \bullet P(x) \\
F_3 &\triangleq \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq (\forall x \bullet P(x) \land Q(x)) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq S(x) \land \forall y \bullet \big(P(x) \Longrightarrow Q(y)\big)
\end{aligned}
$$

- $f(x)$ is free for x in F_1, because x is not free in F_1
- $f(x)$ is free for x in F_2, because x is not quantified in F_2
- $f(y)$ is not free for x in F_2, because y would become quantified
- $f(y)$ is free for y in F_3
- $f(y, y)$ is for x in F_4
- $f(y, y)$ is for x in F_5

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet P(x) \\
F_2 &\triangleq \forall y \bullet P(x) \\
F_3 &\triangleq \exists x \bullet P(y) \land \forall y \bullet Q(y) \\
F_4 &\triangleq (\forall x \bullet P(x) \land Q(x)) \implies \neg P(x) \lor Q(y) \\
F_5 &\triangleq S(x) \land \forall y \bullet (P(x) \implies Q(y))
\end{aligned}
$$

- $f(x)$ is free for x in F_1, because x is not free in F_1
- $f(x)$ is free for x in F_2, because x is not quantified in F_2
- $f(y)$ is not free for x in F_2, because y would become quantified
- $f(y)$ is free for y in F_3
- $f(y, y)$ is free for x in F_4
- $f(y, y)$ is for x in F_5

Free variables

When we substitute a term for a variable we may need to ensure that none of the free variables in the term fall into the scope of a quantifier.

A term t is free for a variable x in a formula F if no variable y in t becomes quantified anywhere a free x appears in F.

$$
\begin{aligned}
F_1 &\triangleq \forall x \bullet P(x) \\
F_2 &\triangleq \forall y \bullet P(x) \\
F_3 &\triangleq \exists x \bullet P(y) \wedge \forall y \bullet Q(y) \\
F_4 &\triangleq (\forall x \bullet P(x) \wedge Q(x)) \implies \neg P(x) \vee Q(y) \\
F_5 &\triangleq S(x) \wedge \forall y \bullet (P(x) \implies Q(y))
\end{aligned}
$$

- $f(x)$ is free for x in F_1, because x is not free in F_1
- $f(x)$ is free for x in F_2, because x is not quantified in F_2
- $f(y)$ is not free for x in F_2, because y would become quantified
- $f(y)$ is free for y in F_3
- $f(y, y)$ is free for x in F_4
- $f(y, y)$ is not free for x in F_5

Natural deduction and completeness

The natural deduction proof system can be extended to predicate logic in a way that it is sound and complete.

Some inference rules for universal quantification:

not used anywhere else

$$\frac{\forall x \bullet F}{F[x \mapsto t]}\ \forall\text{ elimination} \qquad \frac{F[x \mapsto a]}{\forall x \bullet F}\ \forall\text{ introduction}$$

provided t is free for x in F provided a is fresh everywhere

side conditions

Natural deduction and completeness

The natural deduction proof system can be extended to predicate logic in a way that it is sound and complete.

Some inference rules for universal quantification:

not used anywhere else

$$\frac{\forall x \bullet F}{F[x \mapsto t]}\ \forall \text{ elimination} \qquad \frac{F[x \mapsto a]}{\forall x \bullet F}\ \forall \text{ introduction}$$

provided t is free for x in F provided a is fresh everywhere

side conditions

Completeness of natural deduction for first-order logic was first proved by Kurt Gödel in his PhD thesis in 1929.

Natural deduction and completeness

The natural deduction proof system can be extended to predicate logic in a way that it is sound and complete.

Some inference rules for universal quantification:

not used anywhere else

$$\frac{\forall x \bullet F}{F[x \mapsto t]}\ \forall \text{ elimination} \qquad \frac{F[x \mapsto a]}{\forall x \bullet F}\ \forall \text{ introduction}$$

provided t is free for x in F — provided a is fresh everywhere

side conditions

Completeness of natural deduction for first-order logic was first proved by Kurt Gödel in his PhD thesis in 1929.

To emphasize its calculational aspects, predicate logic is also called predicate calculus.

Logic

First-order theories

Domain-specific formulas

First-order logic is a very expressive language but, in its pure form, it lacks domain-specific information useful to express properties in various domains (such as math and software).

For example, we would like to express properties such as:

FORMULA	INTENDED MEANING
$\forall n \bullet n + 1 > n$	the successor of any natural number is larger than the number
$\forall x \exists p \bullet \left(\mathit{Prime}(p) \land p > x \right)$	there are infinitely many prime numbers
$\texttt{result} \neq \texttt{null}$	program variable `result` is not **`null`**
$\forall k \bullet \left(0 \leq k < \mathit{len}(\texttt{a}) \Longrightarrow \texttt{a[}k\texttt{]} = 0 \right)$	all elements of array `a` are zero

Domain-specific formulas

First-order logic is a very expressive language but, in its pure form, it lacks domain-specific information useful to express properties in various domains (such as math and software).

For example, we would like to express properties such as:

FORMULA	INTENDED MEANING
$\forall n \bullet n + 1 > n$	the successor of any natural number is larger than the number
$\forall x \exists p \bullet \big(Prime(p) \land p > x\big)$	there are infinitely many prime numbers
$\texttt{result} \neq \texttt{null}$	program variable `result` is not **`null`**
$\forall k \bullet \big(0 \le k < len(\texttt{a}) \Longrightarrow \texttt{a[}k\texttt{]} = 0\big)$	all elements of array `a` are zero

We need means to constrain the interpretation of variables, functions, and predicates so that they reflect those of the domain we're formalizing.

First-order theories

A first-order theory T consists of:

signature Σ: constant, function, and predicate symbols used by the theory (including a declaration of their domains)

axioms $\mathcal{A}$: closed first-order formulas over Σ

A Σ-formula is a first-order formula using only constants, functions, and predicates in Σ (and any variables, quantifiers, and connectives).

The axioms constrain the interpretations of Σ-formulas in a way that the theory's symbols are interpreted according to their domain-specific meaning.

Theory of equality

The theory of equality's signature includes any constants, functions, and predicates, plus a special binary predicate $=$ for equality.

The axioms define the meaning of equality:

reflexivity: $\forall x \bullet x = x$

symmetry: $\forall x, y \bullet \big(x = y \Longrightarrow y = x\big)$

transitivity: $\forall x, y, z \bullet \big(x = y \land y = z \Longrightarrow x = z\big)$

function congruence: for every n-ary function f, $n > 0$:

$$\forall x_1, \dots, x_n, y_1, \dots, y_n \bullet \left(\begin{array}{c} \bigwedge_{1 \le k \le n} x_k = y_k \\ \Longrightarrow \\ f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \end{array}\right)$$

predicate congruence: for every n-ary predicate P, $n > 0$:

$$\forall x_1, \dots, x_n, y_1, \dots, y_n \bullet \left(\begin{array}{c} \bigwedge_{1 \le k \le n} x_k = y_k \\ \Longrightarrow \\ (P(x_1, \dots, x_n) \Longleftrightarrow P(y_1, \dots, y_n)) \end{array}\right)$$

Theory of equality

The theory of equality's signature includes any constants, functions, and predicates, plus a special binary predicate $=$ for equality.

The axioms define the meaning of equality:

theory symbols bind more strongly than quantifiers

reflexivity: $\forall x \bullet x = x$

symmetry: $\forall x, y \bullet \left(x = y \Longrightarrow y = x\right)$

transitivity: $\forall x, y, z \bullet \left(x = y \wedge y = z \Longrightarrow x = z\right)$

function congruence: for every n-ary function f, $n > 0$:

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n \bullet \left(\begin{array}{c} \bigwedge_{1 \le k \le n} x_k = y_k \\ \Longrightarrow \\ f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \end{array} \right)$$

predicate congruence: for every n-ary predicate P, $n > 0$:

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n \bullet \left(\begin{array}{c} \bigwedge_{1 \le k \le n} x_k = y_k \\ \Longrightarrow \\ (P(x_1, \ldots, x_n) \Longleftrightarrow P(y_1, \ldots, y_n)) \end{array} \right)$$

Theory of Presburger arithmetic

The theory of Presburger arithmetic's signature only includes constants 0 and 1, binary function $+$, and equality predicate $=$.

The axioms define integer linear arithmetic:

zero: $\forall x \bullet \neg(x + 1 = 0)$

successor: $\forall x, y \bullet \big(x + 1 = y + 1 \Longrightarrow x = y\big)$

plus zero: $\forall x \bullet (x + 0 = x)$

plus successor: $\forall x, y \bullet (x + (y + 1) = (x + y) + 1)$

induction: for every Presburger formula F with exactly one free variable x:

$$F[x \mapsto 0] \wedge \forall x \bullet (F[x \mapsto x] \Longrightarrow F[x \mapsto x + 1]) \\ \Longrightarrow \\ \forall x \bullet F[x \mapsto x]$$

Even though variables in Presburger arithmetic range over the nonnegative integers, one can express any linear equation over integers into a formula in Presburger arithmetic.

Theory of Peano arithmetic

The theory of Peano arithmetic's signature only includes constants 0 and 1, binary functions $+$ and $\times$, and equality predicate $=$.

Its axioms include Presburger's as well as:

times zero: $\forall x \bullet (x \times 0 = 0)$

times successor: $\forall x, y \bullet (x \times (y + 1) = (x \times y) + 1)$

Interpretations of Peano arithmetic have variables and constants ranging over the natural numbers, with $+$, $\times$, and $=$ behaving as usual in arithmetic.

Since Peano arithmetic is sufficient to express a large selection of arithmetic properties (but not all), it is also called first-order arithmetic.

Theory of arrays

The theory of arrays's signature only includes:

- the equality predicate $=$
- the read function $\mathit{read}(a, k)$ which returns element at position k in array a
- the write function $\mathit{write}(a, k, v)$ which returns an array that is the same as a but stores v at position k

The axioms include equality between array elements, as well as:

array congruence: $\forall a, j, k \bullet (j = k \Longrightarrow \mathit{read}(a, j) = \mathit{read}(a, k))$

read over write 1: $\forall a, v, j, k \bullet (j = k \Longrightarrow \mathit{read}(\mathit{write}(a, j, v), k) = v)$

read over write 2: $\forall a, v, j, k \bullet$
$(j \neq k \Longrightarrow \mathit{read}(\mathit{write}(a, j, v), k) = \mathit{read}(a, k))$

Theory validity

The axioms of a theory must be satisfied by all meaningful models of the theory.

A formula F is T-valid (valid in theory T) when $\mathcal{M} \models F$
for every interpretation such that $\mathcal{M} \models \mathcal{A}$
(every interpretation that satisfies T's axioms).

An interpretation that satisfies T's axioms is called a T-interpretation.

We write $T \models F$ to denote that formula F is T-valid.

A formula F is T-satisfiable (satisfiable in theory T) when $\mathcal{M} \models F$
for some interpretation such that $\mathcal{M} \models \mathcal{A}$
(some interpretation that satisfies T's axioms).

Theory validity: examples

$\times$ binds more strongly than $+$

$$\forall x, y \bullet \exists z \bullet (2 \times x + 1) + (2 \times y + 1) = 2 \times z$$

Theory validity: examples

$\times$ binds more strongly than $+$

$$\forall x, y \bullet \exists z \bullet (2 \times x + 1) + (2 \times y + 1) = 2 \times z$$

Valid: the sum of two odd numbers is even.

Theory validity: examples

$\times$ binds more strongly than $+$

$$\forall x, y \bullet \exists z \bullet (2 \times x + 1) + (2 \times y + 1) = 2 \times z$$

Valid: the sum of two odd numbers is even.

$$\forall x \bullet \exists y \bullet x = y + y$$

Theory validity: examples

$\times$ binds more strongly than $+$

$$\forall x, y \bullet \exists z \bullet (2 \times x + 1) + (2 \times y + 1) = 2 \times z$$

Valid: the sum of two odd numbers is even.

$$\forall x \bullet \exists y \bullet x = y + y$$

Invalid: not every integer is even.

Theory validity: examples

$\times$ binds more strongly than $+$

$$\forall x, y \bullet \exists z \bullet (2 \times x + 1) + (2 \times y + 1) = 2 \times z$$

Valid: the sum of two odd numbers is even.

$$\forall x \bullet \exists y \bullet x = y + y$$

Invalid: not every integer is even.

$$\forall x, y, z \bullet \overbrace{x \times x \times \cdots \times x}^{n \text{ times}} + \overbrace{y \times y \times \cdots \times y}^{n \text{ times}} \neq \overbrace{z \times z \times \cdots \times z}^{n \text{ times}}$$

Theory validity: examples

$\times$ binds more strongly than $+$

$$\forall x, y \bullet \exists z \bullet (2 \times x + 1) + (2 \times y + 1) = 2 \times z$$

Valid: the sum of two odd numbers is even.

$$\forall x \bullet \exists y \bullet x = y + y$$

Invalid: not every integer is even.

$$\forall x, y, z \bullet \overbrace{x \times x \times \cdots \times x}^{n \text{ times}} + \overbrace{y \times y \times \cdots \times y}^{n \text{ times}} \neq \overbrace{z \times z \times \cdots \times z}^{n \text{ times}}$$

$n \leq 2$**:** invalid (for example: $3^2 + 4^2 = 5^2$)

$n > 2$**:** valid (Fermat's last theorem)

Syntactic soundness and completeness

A theory is (syntactically) sound (consistent)
if its axioms are satisfiable.

Equivalently, in every consistent theory $T \models F$ and $T \models \neg F$ cannot both hold; otherwise, $T \models \bot$ (that is: there exists at least a model $\mathcal{M}$ such that $\mathcal{M} \models \mathcal{A} \wedge \bot$), which is impossible since $\mathcal{M} \not\models \bot$ in every interpretation.

A theory is (syntactically) complete if, for every closed formula F,
either $T \models F$ or $T \models \neg F$.

Intuitively, a theory is complete if the axioms accurately capture the intended meaning of the theory in all cases.

Syntactic incompleteness of arithmetic

Some basic first-order theories are not meant to be complete. For example, the theory of equality includes many uninterpreted symbols, and hence it is syntactically incomplete without additional axioms.

In the case of arithmetic, however, we would hope to be able to capture precisely the standard models of arithmetic with the axioms. This is not possible as shown by the famous incompleteness theorem proved by Gödel in 1931.

Every theory that is syntactically sound and includes the axioms of Peano arithmetic is also incomplete: there exist sentences F such that $\mathcal{M} \models F$ for some T-model $\mathcal{M}$ and $\mathcal{M}' \models \neg F$ for some other T-model $\mathcal{M}'$.

Syntactic incompleteness of arithmetic

Some basic first-order theories are not meant to be complete. For example, the theory of equality includes many uninterpreted symbols, and hence it is syntactically incomplete without additional axioms.

In the case of arithmetic, however, we would hope to be able to capture precisely the standard models of arithmetic with the axioms. This is not possible as shown by the famous incompleteness theorem proved by Gödel in 1931.

and recursively enumerable

Every theory that is syntactically sound and includes the axioms of Peano arithmetic is also incomplete: there exist sentences F such that $\mathcal{M} \models F$ for some T-model $\mathcal{M}$ and $\mathcal{M}' \models \neg F$ for some other T-model $\mathcal{M}'$.

actually, a simpler fragment is enough

Syntactic incompleteness of arithmetic

Some basic first-order theories are not meant to be complete. For example, the theory of equality includes many <u>uninterpreted</u> symbols, and hence it is syntactically incomplete without additional axioms.

In the case of arithmetic, however, we would hope to be able to capture precisely the standard models of arithmetic with the axioms. This is not possible as shown by the famous incompleteness theorem proved by Gödel in 1931.

and recursively enumerable

Every theory that is syntactically sound and includes the axioms of Peano arithmetic is also incomplete: there exist sentences F such that $\mathcal{M} \models F$ for some T-model $\mathcal{M}$ and $\mathcal{M}' \models \neg F$ for some other T-model $\mathcal{M}'$.

actually, a simpler fragment is enough

From the completeness of first-order deduction: there exists F such that neither F nor $\neg F$ is provable (as a theorem).

Consequences of incompleteness

Every theory that is syntactically sound and includes the axioms of Peano arithmetic is also incomplete: there exist sentences F such that $\mathcal{M} \models F$ for some T-model $\mathcal{M}$ and $\mathcal{M}' \models \neg F$ for some other T-model $\mathcal{M}'$.

- This incompleteness is fundamental: we cannot fix it by adding axioms to cover the cases of incompleteness
- The only way that we can have completeness in a theory with arithmetic is if it is unsound; but then the theory would be useless!

Consequences of incompleteness

Every theory that is syntactically sound and includes the axioms of Peano arithmetic is also incomplete: there exist sentences F such that $\mathcal{M} \models F$ for some T-model $\mathcal{M}$ and $\mathcal{M}' \models \neg F$ for some other T-model $\mathcal{M}'$.

- This incompleteness is fundamental: we cannot fix it by adding axioms to cover the cases of incompleteness
- The only way that we can have completeness in a theory with arithmetic is if it is unsound; but then the theory would be useless!

Informally, the incompleteness theorem says that mathematics cannot be reduced to syntax.

Consequences of incompleteness

Every theory that is syntactically sound and includes the axioms of Peano arithmetic is also incomplete: there exist sentences F such that $\mathcal{M} \models F$ for some T-model $\mathcal{M}$ and $\mathcal{M}' \models \neg F$ for some other T-model $\mathcal{M}'$.

- This incompleteness is fundamental: we cannot fix it by adding axioms to cover the cases of incompleteness
- The only way that we can have completeness in a theory with arithmetic is if it is unsound; but then the theory would be useless!

Informally, the incompleteness theorem says that mathematics cannot be reduced to syntax.

Gödel's second incompleteness theorem says that a theory with the same characteristics (sound, including Peano arithmetic) cannot prove its own consistency.

Some good news

Presburger arithmetic is sound and complete.

Computation

Descriptive vs. operational

Logic is a descriptive notation: it models behavior by expressing its properties.

$$\textit{rain} \Longrightarrow \textit{umbrella}$$

Descriptive vs. operational

Logic is a descriptive notation: it models behavior by expressing its properties.

$$rain \Longrightarrow umbrella$$

We now turn to operational notations, which model behavior as states and transitions between states.

Computation

Computational models

State machines

also: automaton

As operational notations we mainly consider variants of state machines.

An (abstract) state machine is a rigorous operational notation to describe computations as sequences of states and transitions.

State machines

also: automaton

As operational notations we mainly consider variants of state machines.

An (abstract) state machine is a rigorous operational notation to describe computations as sequences of states and transitions.

A state machine's definition includes:

alphabet Σ: a finite set of symbols representing events or input

syntax: states and valid transitions between states

semantics: the computations that originate by running the machine

State machines

also: automaton

As operational notations we mainly consider variants of state machines.

An (abstract) state machine is a rigorous operational notation to describe computations as sequences of states and transitions.

A state machine's definition includes:

alphabet Σ: a finite set of symbols representing events or input

syntax: states and valid transitions between states

semantics: the computations that originate by running the machine

Let's see an example of definition of a simple class of state machines.

Finite-state automata: syntax

A (deterministic) finite state automaton consists of:

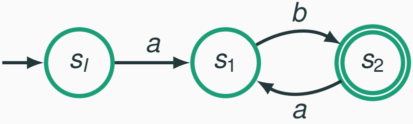
- alphabet: a finite set Σ of symbols
- states: a finite set S of state identifiers
- initial state: a state $s_I \in S$ where computations start
- final states: a subset $F \subseteq S$ of states where computations end
- transition function: a function $\delta\colon S \times \Sigma \to S$ that defines valid transitions between states

Finite-state automata: syntax

A (deterministic) finite state automaton consists of:

- alphabet: a finite set Σ of symbols
- states: a finite set S of state identifiers
- initial state: a state $s_I \in S$ where computations start
- final states: a subset $F \subseteq S$ of states where computations end
- transition function: a function $\delta \colon S \times \Sigma \to S$ that defines valid transitions between states

Finite state automata can be represented using an intuitive graphical representation:



$$\Sigma = \{a, b\} \quad S = \{s_I, s_1, s_2\} \quad F = \{s_2\}$$
$$\delta(s_I, a) = s_1 \quad \delta(s_2, a) = s_1 \quad \delta(s_1, b) = s_2$$

Finite-state automata: semantics

A word (also: trace) w is a sequence of symbols (a string) from Σ:

$$w \in \Sigma^*$$

all strings of finite length

A computation (or run) of an automaton over a trace $w = w[1]\, w[2] \cdots w[n]$ is the sequence

$$q_0 \xrightarrow{w[1]} q_1 \xrightarrow{w[2]} q_2 \xrightarrow{w[3]} \cdots \xrightarrow{w[n]} q_n$$

such that:

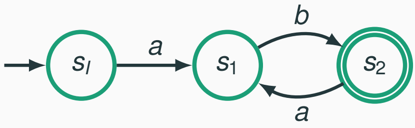
- it starts from the initial state: $q_0 = s_I$
- it respects the transition function: $\delta(q_{k-1}, \sigma_k) = q_k$

A computation is accepting if it ends in a final state: $q_n \in F$.

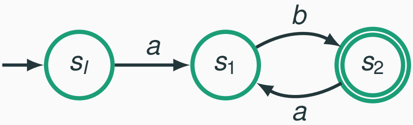
An automaton A accepts w if there exists an accepting computation of A over w.

The set of all strings accepted by an automaton A is called the language $\mathcal{L}(A)$ of A.

Finite-state automata semantics: example



Finite-state automata semantics: example



This automaton's language are all strings consisting of *a b* repeated any positive number of times.

Expressiveness

The expressiveness (also power) of a class of state machines corresponds to the class of languages of any machines that belong to the class.

Expressiveness

The expressiveness (also power) of a class of state machines corresponds to the class of languages of any machines that belong to the class.

In this course we will refer to three classes of languages:

regular languages: accepted by finite-state automata

context-free languages: accepted by pushdown automata

recursive (or decidable) languages: accepted by Turing machines

Expressiveness

The expressiveness (also power) of a class of state machines corresponds to the class of languages of any machines that belong to the class.

In this course we will refer to three classes of languages:

regular languages: accepted by finite-state automata

context-free languages: accepted by pushdown automata

recursive (or decidable) languages: accepted by Turing machines

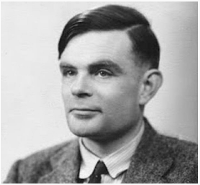
These three language classes define strictly increasing expressiveness:

- every regular language is context-free and recursive
- every context-free language is recursive

Turing machines as universal computers

Turing machines are an abstraction of general-purpose computers: whatever is computable by a Turing machine is computable by a general-purpose computer and vice versa.

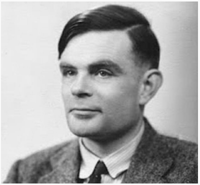
Turing machines were defined by Turing in his landmark 1936 paper “On computable numbers”



Turing machines as universal computers

Turing machines are an abstraction of general-purpose computers: whatever is computable by a Turing machine is computable by a general-purpose computer and vice versa.

Turing machines were defined by Turing in his landmark 1936 paper "On computable numbers"



The Church-Turing thesis stipulates that there is no model of computation that is implementable in the physical world and is more expressive than Turing machines.

Everything computable is computable by Turing machines

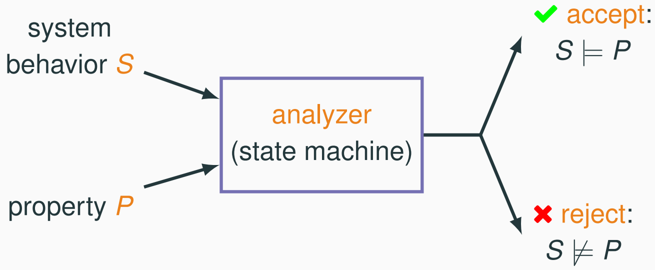
Computation

Computability

Decision problems

The operational nature of state machines makes them suitable to solve analysis problems phrased as decision problems:

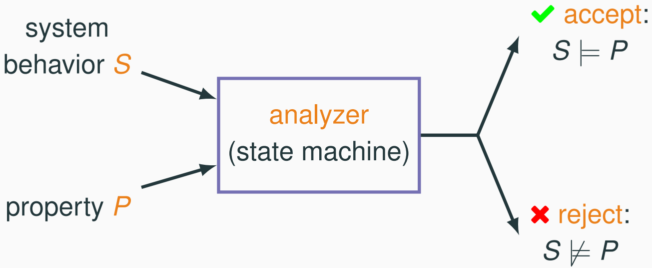
- encode system behavior S and property P as input to the machine
- run the machine on the input to check if the input is accepted



Decision problems

The operational nature of state machines makes them suitable to solve analysis problems phrased as decision problems:

- encode system behavior S and property P as input to the machine
- run the machine on the input to check if the input is accepted

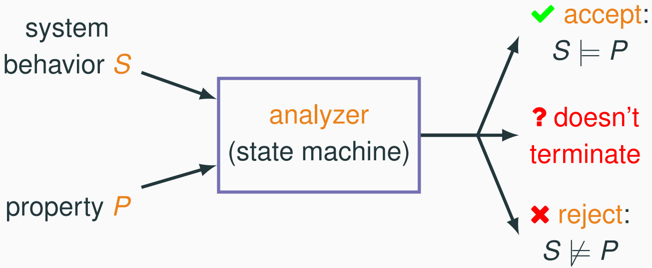


With more expressive machines, such as Turing machines, a third outcome is possible when running an analyzer.

Decision problems

The operational nature of state machines makes them suitable to solve analysis problems phrased as decision problems:

- encode system behavior S and property P as input to the machine
- run the machine on the input to check if the input is accepted



With more expressive machines, such as Turing machines, a third outcome is possible when running an analyzer.

Decidability and undecidability

The universal expressive power of Turing machines comes at a cost.

Decidability and undecidability

The universal expressive power of Turing machines comes at a cost.

Decidability and undecidability

The universal expressive power of Turing machines comes at a cost.

A decision problem is:

decidable if there exists a Turing machines that solves the problem that always halts

undecidable if it is not decidable

Semidecidability

A **decision problem** is:

decidable if there exists a Turing machines that solves the problem that always halts

semidecidable if there exists a Turing machine that always halts on accepted input

A semidecidable problem is one where we can give conclusive positive answers, but we are never sure of negative answers.

Semidecidability

A decision problem is:

decidable if there exists a Turing machines that solves the problem that always halts

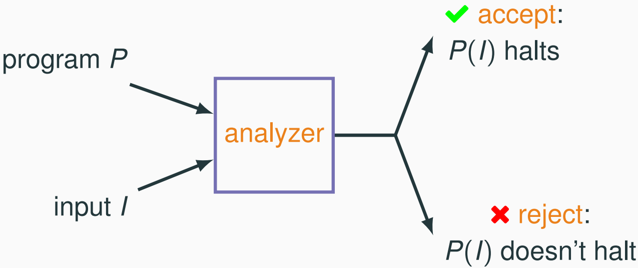
semidecidable if there exists a Turing machine that always halts on accepted input

A semidecidable problem is one where we can give conclusive positive answers, but we are never sure of negative answers.

- The language corresponding to a decidable problem is called recursive
- The language corresponding to a semidecidable problem is called recursively enumerable, because we can enumerate positive answers

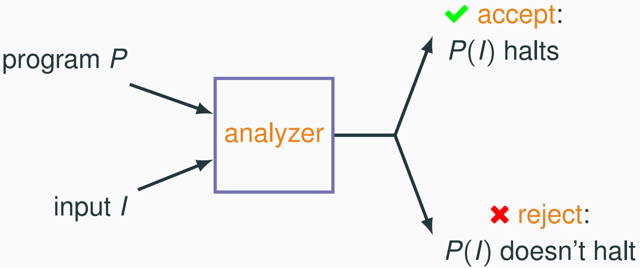
An undecidable problem

The classic undecidable problem is the halting problem:



An undecidable problem

The classic undecidable problem is the halting problem:



The halting problem is:

- semidecidable: if $P(I)$ halts, we can check that it does by simply running P on I
- undecidable: if $P(I)$ does not halt, any analyzer may not halt

Undecidability in practice

Undecidability does not mean that an analysis problem is impossible in all cases: we may still be able to build analyzers that can decide many interesting cases – but they cannot work in every single case.

Undecidability in practice

Undecidability does not mean that an analysis problem is impossible in all cases: we may still be able to build analyzers that can decide many interesting cases – but they cannot work in every single case.

> *In recent years, powerful new tools have emerged that [do not work] infrequently enough that they are useful in practice.*
>
> *"Proving program termination"*

In contrast to popular belief, proving termination is not always impossible.

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

Proving undecidability

To prove that a new analysis problem is undecidable we can reduce an undecidable problem to it:

If we could build an analyzer for an undecidable problem *U* using another analysis problem *N*, it means that *N* is undecidable as well.

The following analysis problem is undecidable:

Null safety problem: given a program *P* and input *I*, and variable `k`, decide whether *P*(*I*) never dereferences `k` when it is **`null`**.

Let's prove, by reduction from the halting problem, that null safety is undecidable.

Proving undecidability using reduction

Assume there exists a null safety analyzer:

```
// return true iff p(i) is null safe; always halts
boolean isNullSafe(Program p, Input i, Variable k)
```

Proving undecidability using reduction

Assume there exists a null safety analyzer:

```
// return true iff p(i) is null safe; always halts
boolean isNullSafe(Program p, Input i, Variable k)
```

Then we can build an analyzer for the halting problem:

```
boolean doesHalt(Program p, Input i) {
  Program p2 = new Program(
    "Integer k = null; // k is not used in p
     p.run(i);         // run p on i
     k.toString();     // null pointer dereference!");
   if (isNullSafe(p2, i, k))
     return false;   // p2 dereferences null iff p terminates
   else return true;
}
```

Program `doesHalt` always terminates (because `isNullSafe` does by hypothesis). Thus, `doesHalt` solves the halting problem in all cases; since this is impossible, `isNullSafe` cannot work.

Rice's theorem

Rice's theorem is a theoretical result that implies that undecidability is ubiquitous in program analysis:

> Rice's theorem: all non-trivial semantic properties of programs are undecidable.

Rice's theorem

Rice's theorem is a theoretical result that implies that undecidability is ubiquitous in program analysis:

> Rice's theorem: all non-trivial semantic properties of programs are undecidable.

neither true for all programs nor false for all programs

about program behavior at run time

Computation

Complexity

Complexity of programs

A program solving a decidable problem always terminates, but may take a very long time or a lot of memory.

The time complexity $T \colon \mathbb{N} \to \mathbb{N}$ of a program gives the maximum number of elementary steps $T(n)$ the program takes for an input of size n.

The space complexity $S \colon \mathbb{N} \to \mathbb{N}$ of a program gives the maximum number of memory locations $S(n)$ the program uses for an input of size n (in addition to the input).

Complexity of programs

A program solving a decidable problem always terminates, but may take a very long time or a lot of memory.

The time complexity $T\colon \mathbb{N} \to \mathbb{N}$ of a program gives the maximum number of elementary steps $T(n)$ the program takes for an input of size n.

The space complexity $S\colon \mathbb{N} \to \mathbb{N}$ of a program gives the maximum number of memory locations $S(n)$ the program uses for an input of size n (in addition to the input).

The notion of elementary step and memory location depends on the computational model, but it is surprisingly robust up to constant multiplicative factors.

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

$f \in \mathrm{O}(g)$: there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
(g is an asymptotic upper bound on f)

$f \in \Omega(g)$: there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
(g is an asymptotic lower bound on f)

$f \in \Theta(g)$: $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
(g is asymptotically equal to (a tight bound on) f)

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

$f \in \mathrm{O}(g)$: there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
(g is an asymptotic upper bound on f)

$f \in \Omega(g)$: there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
(g is an asymptotic lower bound on f)

$f \in \Theta(g)$: $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
(g is asymptotically equal to (a tight bound on) f)

	f is in:						
$f(n)$	$\mathrm{O}(n)$	$\Omega(n)$	$\mathrm{O}(n^3)$	$\Omega(n^3)$	$\mathrm{O}(5^n)$	$\mathrm{O}(10^n)$	$\mathrm{O}(10^{10^n})$
$4 \cdot \log n$							

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

- $f \in \mathrm{O}(g)$**:** there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
  (g is an asymptotic upper bound on f)
- $f \in \Omega(g)$**:** there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
  (g is an asymptotic lower bound on f)
- $f \in \Theta(g)$**:** $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
  (g is asymptotically equal to (a tight bound on) f)

	f is in:					
$f(n)$	$\mathrm{O}(n)$	$\Omega(n)$	$\mathrm{O}(n^3)$	$\Omega(n^3)$	$\mathrm{O}(5^n)$	$\mathrm{O}(10^n)$	$\mathrm{O}(10^{10^n})$
$4 \cdot \log n$	✔	✘	✔	✘	✔	✔	✔
$3 + n^2$							

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

$f \in \mathrm{O}(g)$**:** there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
(g is an asymptotic upper bound on f)

$f \in \Omega(g)$**:** there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
(g is an asymptotic lower bound on f)

$f \in \Theta(g)$**:** $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
(g is asymptotically equal to (a tight bound on) f)

	f is in:						
$f(n)$	$\mathrm{O}(n)$	$\Omega(n)$	$\mathrm{O}(n^3)$	$\Omega(n^3)$	$\mathrm{O}(5^n)$	$\mathrm{O}(10^n)$	$\mathrm{O}(10^{10^n})$
$4 \cdot \log n$	✔	✘	✔	✘	✔	✔	✔
$3 + n^2$	✘	✔	✔	✘	✔	✔	✔
$2 \cdot n \cdot \log n$							

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

$f \in \mathrm{O}(g)$**:** there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
(g is an asymptotic upper bound on f)

$f \in \Omega(g)$**:** there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
(g is an asymptotic lower bound on f)

$f \in \Theta(g)$**:** $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
(g is asymptotically equal to (a tight bound on) f)

	f is in:						
$f(n)$	$\mathrm{O}(n)$	$\Omega(n)$	$\mathrm{O}(n^3)$	$\Omega(n^3)$	$\mathrm{O}(5^n)$	$\mathrm{O}(10^n)$	$\mathrm{O}(10^{10^n})$
$4 \cdot \log n$	✔	✘	✔	✘	✔	✔	✔
$3 + n^2$	✘	✔	✔	✘	✔	✔	✔
$2 \cdot n \cdot \log n$	✘	✔	✔	✘	✔	✔	✔
$10^{10} \cdot n^{100}$							

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

$f \in \mathrm{O}(g)$: there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
(g is an asymptotic upper bound on f)

$f \in \Omega(g)$: there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
(g is an asymptotic lower bound on f)

$f \in \Theta(g)$: $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
(g is asymptotically equal to (a tight bound on) f)

	f is in:						
$f(n)$	$\mathrm{O}(n)$	$\Omega(n)$	$\mathrm{O}(n^3)$	$\Omega(n^3)$	$\mathrm{O}(5^n)$	$\mathrm{O}(10^n)$	$\mathrm{O}(10^{10^n})$
$4 \cdot \log n$	✔	✘	✔	✘	✔	✔	✔
$3 + n^2$	✘	✔	✔	✘	✔	✔	✔
$2 \cdot n \cdot \log n$	✘	✔	✔	✘	✔	✔	✔
$10^{10} \cdot n^{100}$	✘	✔	✘	✔	✔	✔	✔
10^n							

Asymptotic complexity

To capture complexities that are robust up to complexity classes, we use asymptotic complexity relations:

$f \in \mathrm{O}(g)$**:** there exist $c, k > 0$: $f(n) \leq c \cdot g(n)$ for all $n > k$
(g is an asymptotic upper bound on f)

$f \in \Omega(g)$**:** there exist $c, k > 0$: $f(n) \geq c \cdot g(n)$ for all $n > k$
(g is an asymptotic lower bound on f)

$f \in \Theta(g)$**:** $f \in \mathrm{O}(g)$ and $f \in \Omega(g)$
(g is asymptotically equal to (a tight bound on) f)

	f is in:						
$f(n)$	$\mathrm{O}(n)$	$\Omega(n)$	$\mathrm{O}(n^3)$	$\Omega(n^3)$	$\mathrm{O}(5^n)$	$\mathrm{O}(10^n)$	$\mathrm{O}(10^{10^n})$
$4 \cdot \log n$	✔	✖	✔	✖	✔	✔	✔
$3 + n^2$	✖	✔	✔	✖	✔	✔	✔
$2 \cdot n \cdot \log n$	✖	✔	✔	✖	✔	✔	✔
$10^{10} \cdot n^{100}$	✖	✔	✖	✔	✔	✔	✔
10^n	✖	✔	✖	✔	✖	✔	✔

Complexity of programs: bubble sort

```python
def bubble_sort(a):
  changed = True
  while changed:
    changed = False
    for k in range(len(a) - 1):
      if a[k] > a[k + 1]:
         a[k], a[k + 1] = a[k + 1], a[k] # swap a[k], a[k + 1]
         changed = True
  return a
```

Complexity of programs: bubble sort

```
def bubble_sort(a):
  changed = True
  while changed:
    changed = False
    for k in range(len(a) - 1):
      if a[k] > a[k + 1]:
         a[k], a[k + 1] = a[k + 1], a[k] # swap a[k], a[k + 1]
         changed = True
  return a
```

1. The outer loop executes at most $n =$ `len(a)` times

Complexity of programs: bubble sort

```python
def bubble_sort(a):
  changed = True
  while changed:
    changed = False
    for k in range(len(a) - 1):
      if a[k] > a[k + 1]:
         a[k], a[k + 1] = a[k + 1], a[k] # swap a[k], a[k + 1]
         changed = True
  return a
```

1. The outer loop executes at most $n =$ `len(a)` times
2. The inner loop takes $\Theta(n)$ steps

Complexity of programs: bubble sort

```python
def bubble_sort(a):
  changed = True
  while changed:
    changed = False
    for k in range(len(a) - 1):
      if a[k] > a[k + 1]:
         a[k], a[k + 1] = a[k + 1], a[k] # swap a[k], a[k + 1]
         changed = True
  return a
```

1. The outer loop executes at most $n =$ `len(a)` times
2. The inner loop takes $\Theta(n)$ steps
3. The overall time complexity is $T \in \Theta(n^2)$

Complexity of programs: bubble sort

```python
def bubble_sort(a):
  changed = True
  while changed:
    changed = False
    for k in range(len(a) - 1):
      if a[k] > a[k + 1]:
         a[k], a[k + 1] = a[k + 1], a[k] # swap a[k], a[k + 1]
         changed = True
  return a
```

1. The outer loop executes at most $n =$ `len(a)` times
2. The inner loop takes $\Theta(n)$ steps
3. The overall time complexity is $T \in \Theta(n^2)$
4. The overall space complexity is $S \in \Theta(1)$

Complexity of programs: merge sort

```python
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

```python
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

Complexity of programs: merge sort

```python
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

```python
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` + `len(right)` times

Complexity of programs: merge sort

```python
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

```python
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` + `len(right)` times
2. The overall time complexity is $T \in \Theta(n)$, where $n =$ `len(left + right)`

Complexity of programs: merge sort

```
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

```
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` + `len(right)` times
2. The overall time complexity is $T \in \Theta(n)$, where $n =$ `len(left + right)`
3. The overall space complexity is $S \in \Theta(n)$

Complexity of programs: merge sort

```
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

1. There are $\Theta(\log n)$ recursive calls

```
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` + `len(right)` times
2. The overall time complexity is $T \in \Theta(n)$, where $n =$ `len(left + right)`
3. The overall space complexity is $S \in \Theta(n)$

Complexity of programs: merge sort

```python
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

1. There are $\Theta(\log n)$ recursive calls
2. Each recursive call executes `merge` on a progressively smaller portion of the array

```python
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` + `len(right)` times
2. The overall time complexity is $T \in \Theta(n)$, where $n =$ `len(left + right)`
3. The overall space complexity is $S \in \Theta(n)$

Complexity of programs: merge sort

```
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

1. There are $\Theta(\log n)$ recursive calls
2. Each recursive call executes `merge` on a progressively smaller portion of the array
3. $T \in \Theta(n \cdot \log n)$

```
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` $+$ `len(right)` times
2. The overall time complexity is $T \in \Theta(n)$, where $n =$ `len(left + right)`
3. The overall space complexity is $S \in \Theta(n)$

Complexity of programs: merge sort

```
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  left = merge_sort(a[:mid])
  right = merge_sort(a[mid:])
  return merge(left, right)
```

1. There are $\Theta(\log n)$ recursive calls
2. Each recursive call executes `merge` on a progressively smaller portion of the array
3. $T \in \Theta(n \cdot \log n)$
4. $S \in \Theta(n)$

```
def merge(left, right):
  result = []
  x, y = 0, 0
  while x < len(left) and y < len(right):
    if left[x] <= right[y]:
      result += [left[x]]
      x += 1
    else:
      result += [right[y]]
      y += 1
  result += left[x:len(left)] + right[y:len(right)]
  return result
```

1. The loop executes at most `len(left)` + `len(right)` times
2. The overall time complexity is $T \in \Theta(n)$, where $n =$ `len(left + right)`
3. The overall space complexity is $S \in \Theta(n)$

Complexity of problems

The complexity of a problem summarizes the complexity of all programs solving the problem.

A problem P has time complexity:

- $\mathrm{O}(g)$ if there exists a program with time complexity $t \in \mathrm{O}(g)$ that solves P
- $\Omega(g)$ if every program that solves P has time complexity $t \in \Omega(g)$
- $\Theta(g)$ if P has time complexity $\mathrm{O}(g)$ and $\Omega(g)$

Similar definitions apply to space complexity.

Complexity of useful problems

Comparison-based sorting has time complexity:

Complexity of useful problems

Comparison-based sorting has time complexity:

- $O(n^2)$: see bubble sort

Complexity of useful problems

Comparison-based sorting has time complexity:

- $\mathrm{O}(n^2)$: see bubble sort
- $\mathrm{O}(n \cdot \log n)$: see merge sort

Complexity of useful problems

Comparison-based sorting has time complexity:

- $\mathrm{O}(n^2)$: see bubble sort
- $\mathrm{O}(n \cdot \log n)$: see merge sort
- $\Omega(n \cdot \log n)$: proof based on counting comparisons

Complexity of useful problems

Comparison-based sorting has time complexity:

- $\mathrm{O}(n^2)$: see bubble sort
- $\mathrm{O}(n \cdot \log n)$: see merge sort
- $\Omega(n \cdot \log n)$: proof based on counting comparisons
- $\Theta(n \cdot \log n)$

Complexity of useful problems

Comparison-based sorting has time complexity:

- $\mathrm{O}(n^2)$: see bubble sort
- $\mathrm{O}(n \cdot \log n)$: see merge sort
- $\Omega(n \cdot \log n)$: proof based on counting comparisons
- $\Theta(n \cdot \log n)$

Other time complexities:

comparison-based sorting: $\Theta(n \cdot \log n)$

counting-based sorting:

searching in unsorted list:

searching in sorted list:

Complexity of useful problems

Comparison-based sorting has time complexity:

- $O(n^2)$: see bubble sort
- $O(n \cdot \log n)$: see merge sort
- $\Omega(n \cdot \log n)$: proof based on counting comparisons
- $\Theta(n \cdot \log n)$

Other time complexities:

comparison-based sorting: $\Theta(n \cdot \log n)$

counting-based sorting: $\Theta(n)$
lower bound: have to scan all elements

searching in unsorted list:

searching in sorted list:

Complexity of useful problems

Comparison-based sorting has time complexity:

- $O(n^2)$: see bubble sort
- $O(n \cdot \log n)$: see merge sort
- $\Omega(n \cdot \log n)$: proof based on counting comparisons
- $\Theta(n \cdot \log n)$

Other time complexities:

comparison-based sorting: $\Theta(n \cdot \log n)$

counting-based sorting: $\Theta(n)$
lower bound: have to scan all elements

searching in unsorted list: $\Theta(n)$
lower bound: have to scan all elements

searching in sorted list:

Complexity of useful problems

Comparison-based sorting has time complexity:

- $\mathrm{O}(n^2)$: see bubble sort
- $\mathrm{O}(n \cdot \log n)$: see merge sort
- $\Omega(n \cdot \log n)$: proof based on counting comparisons
- $\Theta(n \cdot \log n)$

Other time complexities:

comparison-based sorting: $\Theta(n \cdot \log n)$

counting-based sorting: $\Theta(n)$
lower bound: have to scan all elements

searching in unsorted list: $\Theta(n)$
lower bound: have to scan all elements

searching in sorted list: $\mathrm{O}(\log n)$
upper bound: binary search

Complexity classes

Problems with similar asymptotic complexity are grouped in complexity classes.

time: $\mathrm{TIME}(g)$ is the set of all problems of time complexity $\mathrm{O}(g)$

space: $\mathrm{SPACE}(g)$ is the set of all problems of space complexity $\mathrm{O}(g)$

Complexity classes

Problems with similar asymptotic complexity are grouped in complexity classes.

time: $\mathrm{TIME}(g)$ is the set of all problems of time complexity $\mathrm{O}(g)$

space: $\mathrm{SPACE}(g)$ is the set of all problems of space complexity $\mathrm{O}(g)$

Since writing a memory location takes at least one computational step, and you cannot reuse time:

$$\mathrm{TIME}(g) \subseteq \mathrm{SPACE}(g)$$

Some complexity classes

P is the class $\bigcup_k \mathrm{TIME}(n^k)$ of all problems solvable in polynomial time

EXP is the class $\bigcup_k \mathrm{TIME}(\exp(n^k))$ of all problems solvable in exponential time

PSPACE is the class $\bigcup_k \mathrm{SPACE}(n^k)$ of all problems solvable in polynomial space

EXPSPACE is the class $\bigcup_k \mathrm{SPACE}(\exp(n^k))$ of all problems solvable in exponential space

Nondeterminism

All computational models seen so far are deterministic: for every input there exists only one possible computation.

Nondeterminism

All computational models seen so far are deterministic: for every input there exists only one possible computation.

Nondeterministic computational models are useful to capture problems that are hard to solve but easy to check deterministically.

A nondeterministic program is one with many possible different executions for the same input: the program accepts iff there exists at least one possible execution that accepts.

Nondeterminism

All computational models seen so far are deterministic: for every input there exists only one possible computation.

Nondeterministic computational models are useful to capture problems that are hard to solve but easy to check deterministically.

A nondeterministic program is one with many possible different executions for the same input: the program accepts iff there exists at least one possible execution that accepts.

```python
def sort_det(a):
  return merge_sort(a)
```

```python
def sort_nondet(a):
  b = shuffle(a)    # pick a shuffle
  for k in range(len(b) - 1):
    if b[k] > b[k + 1]:
      return None   # fail
  return b          # success
```

Nondeterminism

All computational models seen so far are deterministic: for every input there exists only one possible computation.

Nondeterministic computational models are useful to capture problems that are hard to solve but easy to check deterministically.

A nondeterministic program is one with many possible different executions for the same input: the program accepts iff there exists at least one possible execution that accepts.

```
def sort_det(a):
  return merge_sort(a)
```

all shuffles executed in parallel!

```
def sort_nondet(a):
  b = shuffle(a)    # pick a shuffle
  for k in range(len(b) - 1):
    if b[k] > b[k + 1]:
      return None   # fail
  return b          # success
```

Nondeterminism

All computational models seen so far are deterministic: for every input there exists only one possible computation.

Nondeterministic computational models are useful to capture problems that are hard to solve but easy to check deterministically.

A nondeterministic program is one with many possible different executions for the same input: the program accepts iff there exists at least one possible execution that accepts.

all shuffles executed in parallel!

```
def sort_det(a):
  return merge_sort(a)
```

```
def sort_nondet(a):
  b = shuffle(a)    # pick a shuffle
  for k in range(len(b) - 1):
    if b[k] > b[k + 1]:
      return None   # fail
  return b          # success
```

Nondeterminism is a useful model of some problems,
but it is not efficiently implementable.

Nondeterministic complexity classes

time: NTIME(g) is the set of all problems of time complexity $\mathrm{O}(g)$ using nondeterminism

space: NSPACE(g) is the set of all problems of space complexity $\mathrm{O}(g)$ using nondeterminism

Nondeterministic complexity classes

time: NTIME(g) is the set of all problems of time complexity $\mathrm{O}(g)$ using nondeterminism

space: NSPACE(g) is the set of all problems of space complexity $\mathrm{O}(g)$ using nondeterminism

Savitch's theorem shows that nondeterminism does not significantly increase space complexity:

$$\mathrm{SPACE}(g) \subseteq \mathrm{NSPACE}(g) \subseteq \mathrm{SPACE}(g^2)$$

Some nondeterministic complexity classes

NP is the class $\bigcup_k \mathrm{NTIME}(n^k)$ of all problems solvable in nondeterministic polynomial time

NEXP is the class $\bigcup_k \mathrm{NTIME}(\exp(n^k))$ of all problems solvable in nondeterministic exponential time

Complexity class hierarchy

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE$$

Complexity class hierarchy

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \underset{\textstyle = \mathrm{NPSPACE}}{\underset{\uparrow}{\mathrm{PSPACE}}} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \underset{\textstyle = \mathrm{NEXPSPACE}}{\underset{\uparrow}{\mathrm{EXPSPACE}}}$$

Tractability and polynomial correlations

Polynomial time roughly corresponds to computationally tractable.

Tractability and polynomial correlations

Polynomial time roughly corresponds to computationally tractable.

Two functions f, g are polynomially correlated if there exist two polynomial functions p, q such that $f \in \mathrm{O}(p(g))$ and $g \in \mathrm{O}(r(f))$.

Tractability and polynomial correlations

Polynomial time roughly corresponds to computationally tractable.

Two functions f, g are polynomially correlated if there exist two polynomial functions p, q such that $f \in \mathrm{O}(p(g))$ and $g \in \mathrm{O}(r(f))$.

- All polynomials polynomially correlate
- n^k and $n \cdot \log n$ polynomially correlate
- 5^n and 10^{4n} polynomially correlate

Tractability and polynomial correlations

Polynomial time roughly corresponds to computationally tractable.

Two functions f, g are polynomially correlated if there exist two polynomial functions p, q such that $f \in \mathrm{O}(p(g))$ and $g \in \mathrm{O}(r(f))$.

- All polynomials polynomially correlate
- n^k and $n \cdot \log n$ polynomially correlate
- 5^n and 10^{4n} polynomially correlate

Tractable complexity classes are polynomially correlated. More precisely: complexity classes that are closed under polynomial correlation are robust with respect to the notion of tractability.

Polynomial reductions

A problem P_2 polynomially reduces to another problem P if there exists a polynomial-time algorithm R that transforms every instance j of P_2 into an instance $R(j)$ of P such that:

1. the size of $R(j)$ polynomially correlates to the size of j; and
2. $P(R(j)) = P_2(j)$ (P on $R(j)$ computes P_2 on j)

Informally: if P_2 polynomially reduces to P,
P can be used to solve P_2 with at most polynomial slow-down
(within the same complexity class as P)

Completeness

Completeness characterize the hardest problems in each class.

A problem P is C-hard for a complexity class C if
every problem in C polynomially reduces to P.

Completeness

Completeness characterize the hardest problems in each class.

A problem P is C-hard for a complexity class C if
every problem in C polynomially reduces to P.

A problem P is C-complete for a complexity class C if
$P \in$ C and P is C-hard.

Completeness

Completeness characterize the hardest problems in each class.

A problem P is C-hard for a complexity class C if
every problem in C polynomially reduces to P.

A problem P is C-complete for a complexity class C if
$P \in \mathrm{C}$ and P is C-hard.

Once we have identified one problem P that is C-complete, any other problem P_2 such that P polynomially reduces to P_2 is also C-hard.

NP-completeness

NP-complete problems are considered key intractable problems:

- there are thousands of them in very different domains
- if we could solve one of them efficiently (in polynomial time), we would immediately solve all of them as efficiently
- every attempt to design an efficient algorithm for them has failed, but we can solve many “average” instances efficiently in practice

NP-completeness

NP-complete problems are considered key intractable problems:

- there are thousands of them in very different domains
- if we could solve one of them efficiently (in polynomial time), we would immediately solve all of them as efficiently
- every attempt to design an efficient algorithm for them has failed, but we can solve many “average” instances efficiently in practice

Some examples of NP-complete problems:

- graph coloring
- traveling salesman problem
- integer knapsack problem
- integer programming
- longest common subsequence of n strings
- Rubik’s cube
- SAT (satisfiability of propositional logic)
- serializability of database histories

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✖	✖	✖
Shortest path					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✖	✖	✖
Shortest path	✔	✔	✖	✖	✖
Primality testing					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✖	✖	✖
Shortest path	✔	✔	✖	✖	✖
Primality testing	✔	✔	✖	✖	✖
Factoring					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✖	✖	✖
Shortest path	✔	✔	✖	✖	✖
Primality testing	✔	✔	✖	✖	✖
Factoring	✖	✔	✖	✖	✖
Graph isomorphism					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✓	✓	✗	✗	✗
Shortest path	✓	✓	✗	✗	✗
Primality testing	✓	✓	✗	✗	✗
Factoring	✗	✓	✗	✗	✗
Graph isomorphism	✗	✓	✗	✗	✗
SAT					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✖	✖	✖
Shortest path	✔	✔	✖	✖	✖
Primality testing	✔	✔	✖	✖	✖
Factoring	✖	✔	✖	✖	✖
Graph isomorphism	✖	✔	✖	✖	✖
SAT	✖	✔	✔	✖	✖
Mahjong ($n \times n$ board)					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✖	✖	✖
Shortest path	✔	✔	✖	✖	✖
Primality testing	✔	✔	✖	✖	✖
Factoring	✖	✔	✖	✖	✖
Graph isomorphism	✖	✔	✖	✖	✖
SAT	✖	✔	✔	✖	✖
Mahjong ($n \times n$ board)	✖	✖	✖	✔	✖
Equivalence of regexes					

Examples of problems and their complexity

	P	NP	NP-c.	PSPACE-c.	EXPSPACE-c.
Sorting	✔	✔	✘	✘	✘
Shortest path	✔	✔	✘	✘	✘
Primality testing	✔	✔	✘	✘	✘
Factoring	✘	✔	✘	✘	✘
Graph isomorphism	✘	✔	✘	✘	✘
SAT	✘	✔	✔	✘	✘
Mahjong ($n \times n$ board)	✘	✘	✘	✔	✘
Equivalence of regexes	✘	✘	✘	✘	✔

P vs. NP

Most of the complexity class inclusions are not strict:

- we don't have a proof that P $\neq$ NP
- we don't have a proof that P $\neq$ PSPACE

P vs. NP

Most of the complexity class inclusions are not strict:

- we don't have a proof that P $\neq$ NP
- we don't have a proof that P $\neq$ PSPACE

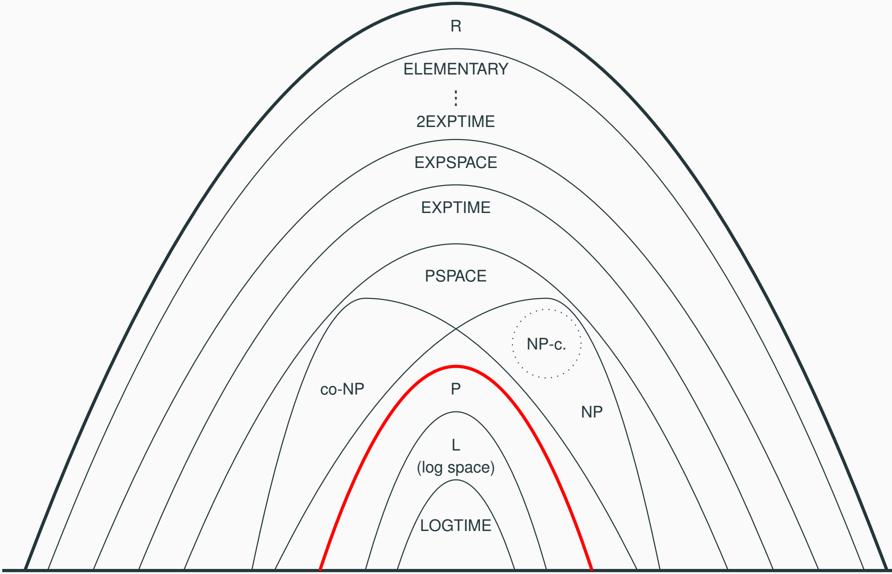
However, most experts agree that P $\neq$ NP is the most likely scenario.

> *Like any other successful scientific hypothesis, the* P $\neq$ NP *hypothesis has passed severe tests that it had no good reason to pass were it false.*
>
> *Aaronson: "The scientific case for* P $\neq$ NP*"*

P vs. NP

Complexity classes diagram

P (in red) is considered the boundary between feasible and infeasible.



Probabilistic and quantum models

Similarly to nondeterministic computational models, there are other computational models that extend the deterministic one.

probabilistic: where computations can take random choices

quantum: where computations have access to a quantum-mechanical state

Whether these models are polynomially-equivalent to deterministic computational models is an open question.
(Best guesses: the probabilistic model is equivalent, but the quantum model is more powerful.)

Complexity of logic

Validity (and its dual satisfiability) is the key decision problem in logic.

The complexity of some logics we know:

The complexity of a logic is the complexity of its decision problem.

Complexity of logic

Validity (and its dual satisfiability) is the key decision problem in logic.

The complexity of some logics we know:

> The **complexity** of a logic is the complexity of its **decision problem**.

propositional: NP-complete

first-order: undecidable but semidecidable

theory of equality: quantifier-free fragment decidable in P

linear arithmetic: decidable with complexity $\Omega\left(2^{2^{k \cdot n}}\right)$ and $O\left(2^{2^{2^{h \cdot n}}}\right)$

Complexity theory for verification experts

Nadia Polikarpova
@polikarn

Following

software engineer: linear is fast, quadratic is slow
complexity theorist: P is fast, NP-hard is slow
verification researcher: decidable is fast, undecidable is slow

5:47 PM - 13 Dec 2018

Complexity theory for verification experts

Nadia Polikarpova
@polikarn

Following

software engineer: linear is fast, quadratic is slow
complexity theorist: P is fast, NP-hard is slow
verification researcher: decidable is fast, undecidable is slow

5:47 PM - 13 Dec 2018

Carlo A. Furia @bugcounting · Dec 13
Replying to @polikarn
When dealing with real-time temporal logic verification, sometimes I even thought: decidable is fast, semidecidable is still OK, I'll start to worry when I hit \Sigma_1^1

4 2 9

Nadia Polikarpova @polikarn · Dec 13
Totally true in synthesis too. No alternating quantifiers, no problem 😏

1 6

Summary

Summary

Logic is a fundamental mathematical descriptive language to express and derive properties of interest.

We will use extensively both the simpler propositional logic and the more expressive first-order predicate logic.

The fundamental decision problems in logic are validity and its dual satisfiability.

Operational notations, such as abstract state machines, describe computations as sequences of transitions between states.

Computational problems are classified according to their decidability and complexity.

These slides' license