IN5170/IN9170 – Models of concurrency

Program Analysis (Part 1), lecture 7

Fall 2022

12. 09. 2022



Reminder last week

- Using invariants to guide development
- Design invariants for concurrency and ensure they hold

```
int nr := 0;    # nunber of active readers
int nw := 0;    # number of active writers

## Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

```
process Reader [i=1 to M]{
  while (true) {
    ...
    < await (nw=0)
      nr := nr+1>;
    read from DB;
    < nr := nr - 1>
  }
}
```

```
process Writer [i=1 to N]{
  while (true) {
    ...
    < await (nr = 0 and nw = 0)
      nw := nw+1>;
    write to DB;
    < nw := nw - 1>
  }
}
```

Reminder: Condition Synchronization

```
int nr := 0, nw = 0;          # counter variables (as before)
sem e  := 1;                  # entry semaphore
int dr := 0; sem r := 0;      # delay counter + sem for reader
int dw := 0; sem w := 0;      # delay counter + sem for writer

# invariant  RW: (nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1
```

```
process Reader [i=1 to M]{   # entry condition: nw = 0
  while (true) {
    ...
      P(e);
      if (nw > 0) { dr := dr + 1;   # < await (nw=0)
                    V(e);           #     nr:=nr+1 >
                    P(r)};
      nr:=nr+1; SIGNAL;

      read from DB;

      P(e); nr:=nr-1; SIGNAL;       # < nr:=nr-1 >
  }
}
```

Central question for next lectures: Is my program correct?

- Does a given program behave as intended?
- How to express intended behavior?
- How to detect surprising behavior?

  $\Longrightarrow$ Formal analysis

Central question for next lectures: Is my program correct?

- Does a given program behave as intended?
- How to express intended behavior?
- How to detect surprising behavior?
  $\Longrightarrow$ Formal analysis

$$x := 5; \{\ x = 5\ \}\langle x := x + 1\rangle; \{\ x = ?\ \}$$

- Clear: $x = 5$ *immediately* after first assignment
- Will $\{\ x = 6\ \}$ hold when the second assignment is executed?
  - Depends on other processes $\Rightarrow$ **interference**

Invariants

Focus in this course: invariants, e.g., for semaphores (last week) or monitors (next week) and interference (CW 38).

Central question for next lectures: Is my program correct?

- Does a given program behave as intended?
- How to express intended behavior?
- How to detect surprising behavior?
  $\Longrightarrow$ Formal analysis

$$x := 5; \{\ x = 5\ \}\langle x := x + 1\rangle; \{\ x = ?\ \}$$

- Clear: $x = 5$ *immediately* after first assignment
- Will $\{\ x = 6\ \}$ hold when the second assignment is executed?
  - Depends on other processes $\Rightarrow$ **interference**

Invariants

Focus in this course: invariants, e.g., for semaphores (last week) or monitors (next week) and interference (CW 38).

Today (Part I): Basic machinery for sequential program reasoning

Verification of global invariants

To prove a global invariant I, we must prove:

$$\{I\}\ \mathtt{s}\ \{I\}$$

for each atomic statement/critical region s: if I holds before s, then it holds afterwards.

Verification of global invariants

To prove a global invariant I, we must prove:

$$\{I\}\ \mathtt{s}\ \{I\}$$

for each atomic statement/critical region s: if I holds before s, then it holds afterwards.

Example (Consider the await language:)

- $\{I\} < \mathtt{s} > \{I\}$ reduces to $\{I\}\ \mathtt{s}\ \{I\}$

  Note: This is a sequential verification problem!

- $\{I\} < \text{await } \mathtt{b}; \mathtt{s} > \{I\}$ reduces to $\{I \wedge b\}\ \mathtt{s}\ \{I\}$

  Note: Again a sequential verification problem!

Verification of global invariants

To prove a global invariant I, we must prove:

$$\{I\}\ \mathtt{s}\ \{I\}$$

for each atomic statement/critical region s: if I holds before s, then it holds afterwards.

Example (Consider the await language:)

- $\{I\} < \mathtt{s} > \{I\}$ reduces to $\{I\}\ \mathtt{s}\ \{I\}$

  Note: This is a sequential verification problem!

- $\{I\} < \text{await } \mathtt{b}; \mathtt{s} > \{I\}$ reduces to $\{I \wedge b\}\ \mathtt{s}\ \{I\}$

  Note: Again a sequential verification problem!

Decomposition

How to decompose a verification problem into sequential pieces?

Concurrent executions

- Concurrent program: several threads operating on (here) *shared* variables
- Parallel updates to x and y:

$$\text{co } \langle x := x * 3; \rangle \parallel \langle y := y * 2; \rangle \text{ oc}$$

Concurrent executions

- Concurrent program: several threads operating on (here) *shared* variables
- Parallel updates to x and y:

$$\text{co } \langle x := x * 3; \rangle \parallel \langle y := y * 2; \rangle \text{ oc}$$

- Every (concurrent) execution can be written as a sequence of atomic operations
- Two possible execution paths for the above program
- Generally, if n processes executes m atomic operations each:

$$\frac{(n*m)!}{m!^n} \qquad \text{If n=3 and m=4:} \frac{(3*4)!}{4!^3} = 34650$$

Will all different execution paths result in different states?

General Analysis Techniques

- Testing/debugging

How to verify program properties?

General Analysis Techniques

- Testing/debugging

Testing

- Execute for some input/scheduling
- Check whether result is as expected
- Check is either manual (debugging) or automatic (testing)

\+ Easy to do

o Increases confidence in correctness of the program

\- Cannot prove absence of bugs

General Analysis Techniques

- Testing/debugging
- Operational Reasoning: Systematic Testing/Explicit State Model Checking

How to verify program properties?

General Analysis Techniques

- Testing/debugging
- Operational Reasoning: Systematic Testing/Explicit State Model Checking

Operational Reasoning

- Like testing, but make sure to explore every execution path
- Essentially an exhaustive case analysis

\+ Easy to implement, automatic

\+ Proves absence of bugs

\- Does not scale, even with

General Analysis Techniques

- Testing/debugging
- Operational Reasoning: Systematic Testing/Explicit State Model Checking
- Formal Analysis: Deductive Verification/Symbolic Model Checking

How to verify program properties?

General Analysis Techniques

- Testing/debugging
- Operational Reasoning: Systematic Testing/Explicit State Model Checking
- Formal Analysis: Deductive Verification/Symbolic Model Checking

Formal Analysis

- Specifies intended behavior precisely
- Uses assertional reasoning to deduce correctness of a program *without* executing it
- Formal argument that the specification is correct

\+ Proves absence of bugs
\+ Scales reasonable well
\- Not always automatic: requires additional user input

Outline

- What is a state?
- What is an assertion?
- What are different specification patterns?
- What are formal systems?

States

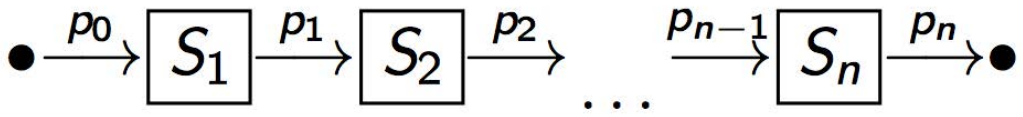
- The *state* of a program consists of the values of the program variables at a point in time, example: $\{ x = 2 \wedge y = 3 \}$
- The state space of a program is given by the different values that the declared variables can take
- Sequential program: one execution thread operates on its own state space
- The state may be *changed* by assignments (in "imperative" programming)

Example

$$\{ x = 5 \wedge y = 5 \}\mathtt{x := x * 2;}\{ x = 10 \wedge y = 5 \}\mathtt{y := y * 2;}\{ x = 10 \wedge y = 10 \}$$

Executions

- Given program S as sequence $S_1; S_2; \ldots; S_n;$, starting in a state p_0:



  where $p_1, p_2, \ldots p_n$ are the different states during execution
- Can be documented by: $\{p_0\}S_1\{p_1\}S_2\{p_2\}\ldots\{p_{n-1}\}S_n\{p_n\}$
- p_0, p_n gives an external specification of the program: $\{p_0\}S\{p_n\}$
- We refer to p_0 as the *initial* state and p_n as the *final* state

Example (from previous slide)

$$\{\ x = 5 \wedge y = 5\ \}\ \mathtt{x} := \mathtt{x} * 2;\ \mathtt{y} := \mathtt{y} * 2;\ \{\ x = 10 \wedge y = 10\ \}$$

Want to express more general properties of programs, like invariants:

$$\{\ x = y\ \}\ \mathtt{x} := \mathtt{x} * 2; \mathtt{y} := \mathtt{y} * 2;\ \{\ x = y\ \}$$

- If the assertion $x = y$ holds, when the program *starts*, $x = y$ will also hold when/if the program *terminates*
- Does not talk about specific, concrete *values* of x and y, but about *relations* between their values
- Assertions characterize sets of states

Example

The assertion $x = y$ describes *all* states where the values of $\mathtt{x}$ and $\mathtt{y}$ are equal, like $\{x = -1 \wedge y = -1\}$, $\{x = 1 \wedge y = 1\}$, ...

state assertion P: defines the *set* of states where P is true

Example

$x = y$	all states where x has same value as y
$x \leq y$:	all states where the value of x is less or equal to the value of y
$x=2 \wedge y=3$	the state where x is 2 and y is 3
$x=2 \vee y=3$	the state where x is 2 or y is 3
$x=2 \Rightarrow y=3$	the state where y is 3 if x is 2
true	all states
false	no state

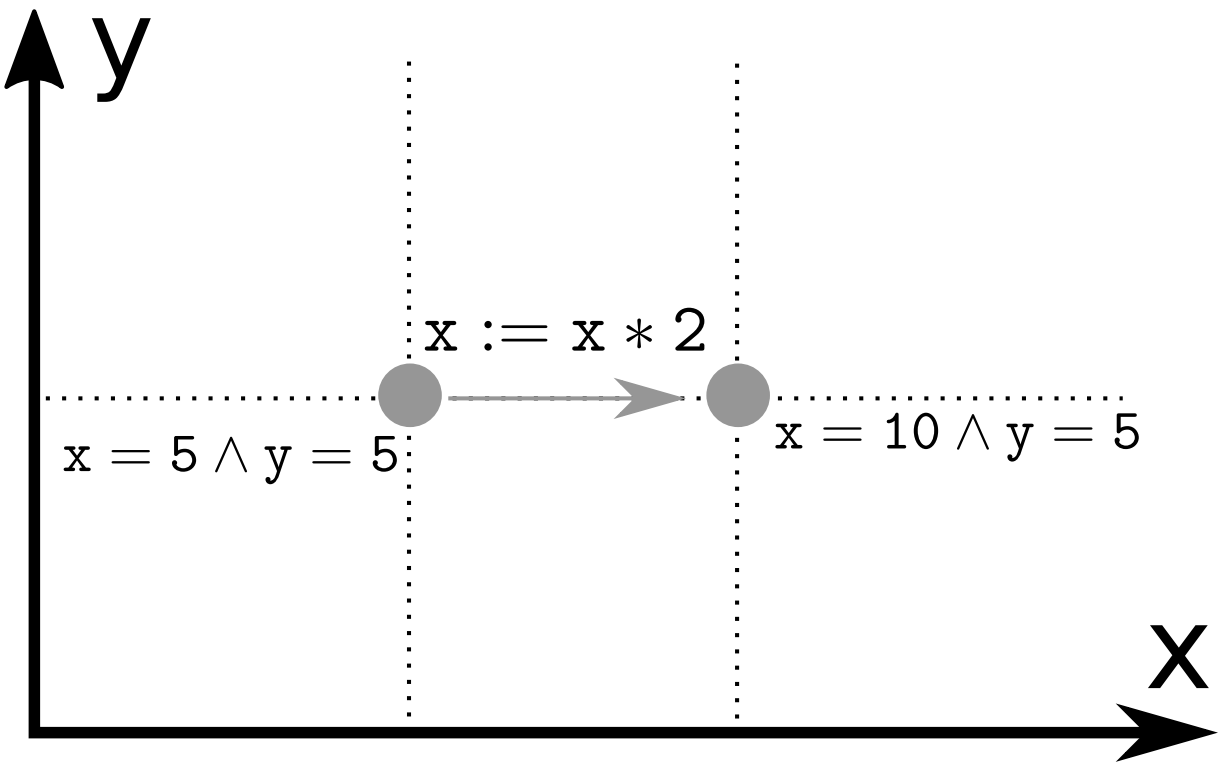
Example

$$\{\ x = y\ \}\mathtt{x} := \mathtt{x} * 2;\{\ x = 2 * y\ \}\mathtt{y} := \mathtt{y} * 2;\{\ x = y\ \}$$

Assertions may or may not say something *correct* for the behavior of a program. In this example, the assertions say something correct.

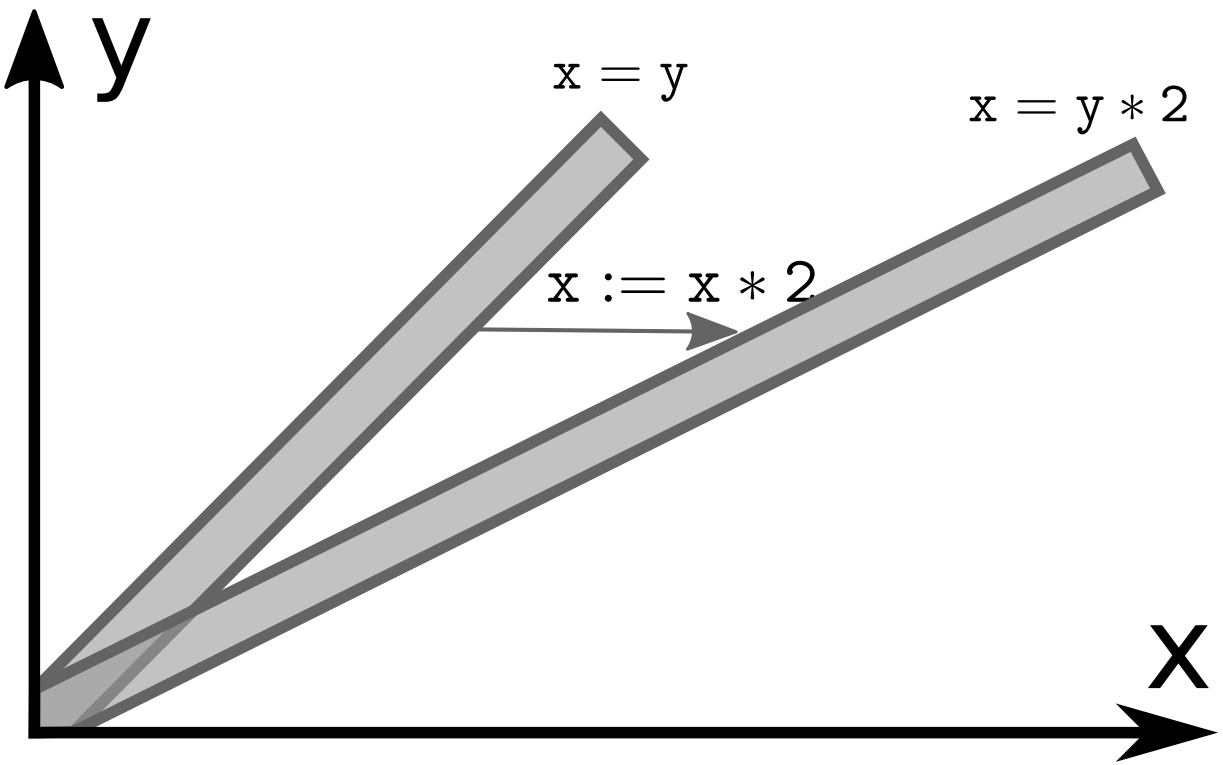
States as Predicates

- A statement transforms a state into another
- If states are described by predicates...
- ...statements are *predicate transformers*



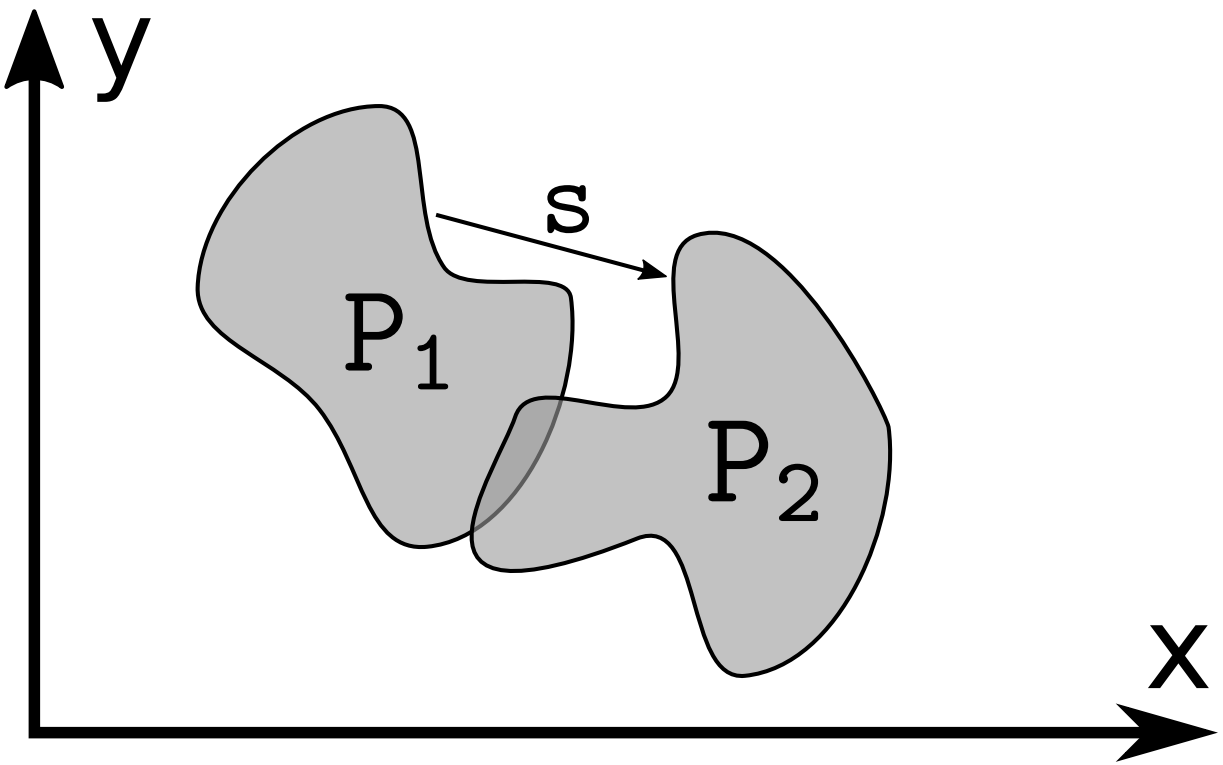
States as Predicates

- A statement transforms a state into another
- If states are described by predicates...
- ...statements are *predicate transformers*



States as Predicates

- A statement transforms a state into another
- If states are described by predicates...
- ...statements are *predicate transformers*



Are we using assertions directly? No

Are we using assertions directly? No

Use of Formal Specifications

- Guides the development of the program
- High-level design, documentation, testing

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions

Pre/post conditions

- Condition on starting state and final state
- No specification of intermediate state
- Can be seen as input/output specification

```
/*@ requires i > 0; ensures result > 0; @*/
Int double(Int i){ return i*i; }
```

Roughly $\{i > 0\}\texttt{res} := \texttt{i} * \texttt{i}\{res > 0\}$

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses

Loop Invariants, Termination Witnesses

- Condition that is both pre- and post-condition for iterations
- Termination witnesses: specify a term to prove termination
- Must be positive and monotonously descreasing

```
/*@ invariant i > 0; @*//*@ decreases i @*/
while( i > 0 ) {res += i*i; i = i+1;}
```

$\{i > 0 \wedge old = i\}\texttt{res} := \texttt{i} * \texttt{i}; \texttt{i} = \texttt{i} + 1\{i > 0 \wedge old > i\}$

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses
- Class invariants, data invariants

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses
- Class invariants, data invariants

Class Invariants

- Similar as loop invariants, but for methods
- Condition that has to hold whenever a method starts or terminates

```
/*@ invariant i >= 0; @*/
class C{ Int i = 0;
void inc(){i = i + 1}...
```

$\{i >= 0\}\mathtt{i} = \mathtt{i} + 1\{i >= 0\}$

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses
- Class invariants, data invariants
- Global and local invariants for concurrent systems

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses
- Class invariants, data invariants
- Global and local invariants for concurrent systems

Global/Local invariants

- Specify state at every point for whole system (global) or one thread (local)
- May be violated inside critical regions
- Same idea as before: invariant at every point control changes

Are we using assertions directly? No

Kinds of Formal Specifications

- Pre/post conditions
- Loop invariants, termination witnesses
- Class invariants, data invariants
- Global and local invariants for concurrent systems

Case Study: TimSort

- Standard sorting algorithm in Java, Python, Android libraries
- Relies on invariants on intermediate states
- Formal analysis discovered that invariants can be violated and exception is thrown
- Smallest known counter-example has length $2^{16} = 65536$

Formal System

Formal Analysis

- Establish program properties, using formal reasoning
- Help in understanding how a program behaves
- Useful for program construction
- Implemented in analysis tools

Formal system

- *Syntax*: What is the language we are considering?
  For example, formulas, programs, programs with formulas
- *Semantics*: What is the meaning of the language?
  many possiblities, e.g., state transformations
- *Axiomatics*: What are the operations on the syntax?
  here: proof rules
- We focus on syntax and axiomatics

In more detail, our formal system consists of:

- A language, formed by syntactic building blocks:
  - A set of *symbols* (constants, variables,...)
  - A set of *formulas* (meaningful combination of symbols)
- A derivation machinery/proof system
  - A set of *axioms* (assumed to be true)
  - Defines the meaning of individual program statements [1]
  - A set of *inference rules*

$$\frac{H_1 \quad \ldots \quad H_n}{C}$$

  - H_i: *premise* (sometimes called *assumption*),
  - C : *conclusion*
  - intention: conclusion is true if all the premises are true
  - The inference rules specify how to derive new formulas from axioms and already derived formulas.
- a semantics, defining the meaning of the formulas

[1] **NB:** An axiom can be seen as a rule with no premise.

First-Order Logic (FOL)

Used for assertions/program specifications. Will be embeded into program logic. Three syntactic categories: *Variables*, *Terms*, *Formulas*

First-Order Logic (FOL)

Used for assertions/program specifications. Will be embeded into program logic. Three syntactic categories: *Variables, Terms, Formulas*

Variables

- Symbols: $x, y, z, \ldots$
- Come in two sorts:
  - Program variables that can change during execution, but not be used in quantifiers
  - Logical variables that cannot changes, but be used in quantifiers

The semantics of a variable is a value

First-Order Logic (FOL)

Used for assertions/program specifications. Will be embeded into program logic. Three syntactic categories: *Variables, Terms, Formulas*

Terms

- Formed by function symbols $+, -\ldots$, refers to variables and other terms.

$$t ::= f(t, \ldots, t) \mid v$$

- Each function symbol takes fixed number of parameters (*arity*)
- A function symbol with arity 0 is a constant $(0, 1, \ldots)$

1 ✓ $\quad x$ ✓ $\quad +(1,1) \equiv 1+1$ ✓ $\quad 1+(x*2)$ ✓ $\quad x = 1$ X

The semantics of a term is a value, namely the one resulting from evaluating it

First-Order Logic (FOL)

Used for assertions/program specifications. Will be embeded into program logic. Three syntactic categories: *Variables, Terms, Formulas*

Formulas

- Formed by connectives relation symbols $\leq, \geq, = \ldots$
- Connectives $\neg, \vee, \wedge, \Rightarrow$
- Quantifiers $\forall, \exists$

The semantics of a term is a *truth value*

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

Example (Polly can Fly)

$$\Big(\big(\forall x.\ \mathrm{isBird}(x) \Rightarrow \mathrm{canFly}(x)\big) \wedge \big(\forall y.\ \mathrm{isParrot}(y) \Rightarrow \mathrm{isBird}(y)\big)\Big) \Rightarrow \big(\mathrm{isParrot}(\mathrm{Polly}) \Rightarrow \mathrm{canFly}(\mathrm{Polly})\big)$$

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

$\exists x.\ \forall y.\ y \geq x$

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

$\exists x.\ \forall y.\ y \geq x$ Existance of a minimum

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

$\exists x.\ \forall y.\ y \geq x$ Existance of a minimum

$\forall x.\ \forall y.\ (y \geq 0 \wedge x \geq 0) \rightarrow x * y \geq 0$

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

$\exists x.\ \forall y.\ y \geq x$	Existance of a minimum
$\forall x.\ \forall y.\ (y \geq 0 \wedge x \geq 0) \rightarrow x * y \geq 0$	Sign of multiplication

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

$\exists x.\ \forall y.\ y \geq x$	Existance of a minimum
$\forall x.\ \forall y.\ (y \geq 0 \wedge x \geq 0) \rightarrow x * y \geq 0$	Sign of multiplication
$\forall x.\ \forall y.\ x > y \rightarrow (\exists z.\ x > z \wedge z > y)$	

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Syntax (FOL Formulas)

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

$\exists x.\ \forall y.\ y \geq x$	Existance of a minimum
$\forall x.\ \forall y.\ (y \geq 0 \wedge x \geq 0) \rightarrow x * y \geq 0$	Sign of multiplication
$\forall x.\ \forall y.\ x > y \rightarrow (\exists z.\ x > z \wedge z > y)$	Density

[2] $A(x)$ to indicate that, here, A (typically) contains x.

Propositional Logic

Propositional Logic is First-Order Logic only with relation symbols with arity 0. Rules applicable on FOL as well.

Typical axioms:

$$A \vee \neg A \qquad A \Rightarrow A \qquad \mathit{true}$$

Typical rules:

$$\frac{A \quad B}{A \wedge B}\ \text{And-I} \qquad \frac{A}{A \vee B}\ \text{Or-I} \qquad \frac{A \Rightarrow B \quad A}{B}\ \text{Impl-E/modus ponens}$$

Example

$$\frac{x = 5 \qquad y = 5}{x = 5 \wedge y = 5}\ \text{And-I} \qquad \frac{x = 5}{x = 5 \vee y = 5}\ \text{Or-I}$$

$$\frac{x \geq 0 \Rightarrow y \geq 0 \qquad x \geq 0}{y \geq 0}\ \text{Impl-E}$$

Formal Systems:Important terms

- **Proof**: derivation tree where all leaf nodes are axioms
- **Theorems**: a "formula" P derivable in a given proof system
  written: $\vdash P$
- **Interpretation**: defines each formula as either *semantically true* or *false*. Semantically true is called *valid*.
  written $\models P$
- **Soundness** (of the logic): If we can prove ("derive") some formula P (in the logic) then P is actually semantically true[3]

$$\forall P.\ \vdash P \ \Rightarrow\ \models P$$

- **Completeness**: If a formula P is semantically true, then it can be proved

$$\forall P.\ \models P \ \Rightarrow\ \vdash P$$

Soundness and completeness means: Semantically true formulas and provable formulas are the same, or

validity = provability (or $\models = \vdash$*)*

[3] The quantifier and implication is *not* part of the logic in question

Program/Hoare Logic

Program Logic (PL)

- FOL is one of the most used specification logics
- Commonly extended to be more usable
- How to connect FOL with programs?

Program Logic (PL)

Given two FOL assertions P_1, P_2, and a program S, a formula of the program logic has the form.

$$\{ P_1 \} S \{ P_2 \}$$

- PL lets us *express* and *prove* properties about programs
- Formulas of this form are commonly called *"Hoare triples"*
- Intuition: P_1 pre-condition, and P_2 post-condition of S

Example: $\{ x = y \}\ \mathtt{x} := \mathtt{x} * 2;\ \mathtt{y} := \mathtt{y} * 2;\ \{ x = y \}$

[Gordon, 2016, Hoare, 1969, Dahl, 1992]

The Formal System for PL (Hoare logic)

Express and prove program properties

$$\{P\}\ S\ \{Q\}$$

- (P, Q) is the specification of the program S, called a contract
- Code analysis by proving the specification (in PL)
- Symbolic and static analysis:
  - No need to execute the code in order to do the analysis
- An interpretation maps triples to *true* or *false*
  - $\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 1\ \}$ should be semantically *true*
  - $\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 0\ \}$ should be semantically *false*

References I

[Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.

[Dahl, 1992] Dahl, O.-J. (1992).
Verifiable Programming.
International Series in Computer Science. Prentice Hall, New York, N.Y.

[Gordon, 2016] Gordon, M. (2016).
Background reading on Hoare Logic.
https://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf.

[Hoare, 1969] Hoare, C. A. R. (1969).
An Axiomatic Basis of Computer Programming.
Communications of the ACM, 12:576–580.