1. Boolean algebras, logic, and induction

1. Propositional logic and Boolean algebras
2. Predicate logic
3. Some proof methods
4. Proofs of correctness
5. Formal methods in verification

Objective

1. Explain and apply logical inference

Readings: Epp, Sec. 6.4; 11.1-11.3; handouts
Subtopic outcomes

1.1 Explain propositional logic as a Boolean algebra (1a)
1.2 Distinguish predicate from propositional logic (1b)
1.3a Write a direct proof
1.3b Prove by induction a theorem in number theory (1c)
1.4 Explain the basic concepts of algorithm correctness proofs (1d)
1.5 Prove, by induction, the correctness of a simple algorithm (1e)

1. Propositional logic and Boolean algebras

- What is reasoning?
- What is algebra?
**Boolean algebras**

- **Definition:** An algebra is a set of values and a set of operations on the values
- **Definition:** A Boolean algebra is a set $A$ and two binary operations on $A$ that have commutative, associative, distributive, identity, and complement properties
- **Examples:**
  - Propositional logic = ($\{F, T\}, \{\lor, \land\}$)
  - Set theory = ($\varnothing(U), \{\cap, \cup\}$)

**Identity and complement**

- **Identity:** there exist elements called 0 and 1 $\in A$ s.t. $(\forall a \in A)\ ((a + 0 = a) \land (a \times 1 = a))$
- **Complement:** $(\forall a \in A)\ (\exists a^{-1} \in A)$
  - $a + a^{-1} = 1$
  - $a \times a^{-1} = 0$
- $a^{-1}$ is the complement of $a$ under each of the operations
- The operations enable transforming an identity into its dual, by exchanging elements 0 and 1, and operations + and ×; e.g., $a + 0 = a$, $a \times 1 = a$
Properties of Boolean algebras

• Complements and identity elements are unique
• $0^{-1} = 1, 1^{-1} = 0$
• Double complement identity:
  \[(\forall a \in A)(a^{-1})^{-1} = a\]
• Idempotent: $a + a = a, aa = a$
• Universal bound: $a + 1 = 1, a \times 0 = 0$
• De Morgan’s Laws: $(a + b) a = a, ab + a = a$
• These properties correspond to propositional logic, i.e. $f = 0, t = 1, f^{-1} = t, (t \lor t = t), (t \land t = t)$

Propositional logic syntax

• Propositional logic (propositional calculus) is a Boolean algebra with the values True and False and the operations $\neg, \land, \lor, \implies$
• It is a language of assertions that each evaluate to true or false, including assertions of the form
  – true, false
  – symbols ($p, q, r, ...$) denoting assertions
  – negation ($\neg p$)
  – conjunction: $p \land q$
  – disjunction: $p \lor q$
  – implication: $p \implies q$ (equivalent to ($\neg q \lor p$))
Semantics of propositional logic

- Let \( p, q \), be formulas
- Wherever \( p \) is true, \( \neg p \) is false
- \( (p \land q) \) is true when both \( p \) and \( q \) are true
- \( (p \lor q) \) is true when either \( p \) or \( q \) or both are true
- \( (p \rightarrow q) \) is true when \( p \) is false or \( q \) is true
- \( (p \iff q) \) or \( (p \equiv q) \) or \( (p \text{ iff } q) \) is true when \( p \) and \( q \) have the same values under all interpretations (truth assignments)

Normal forms

- **Conjunctive normal form** (CNF): a form of logic sentences that consist of the joining of disjunctive (OR) clauses by conjunction (AND) operators
- **Example**: \( (p \lor q) \land (\neg q \lor r) \)
- **Disjunctive normal form** (DNF): a form of logic sentences that consist of the joining of conjunctive (AND) clauses by disjunction (OR) operators
- **Example**: \( (p \land q) \lor (\neg q \land r) \)
Interpreting

- An interpretation of a set of formulas is an assignment of truth values to the symbols.
- Example: An interpretation of \((p \land \neg(p \lor q))\) is \(p = \text{true}, q = \text{false}\). Under this interpretation, \((p \land \neg(p \lor q))\) is false.
- A formula is satisfiable if it has an interpretation under which it is true.

Truth Tables

- A truth table lists the values of a formula under all possible interpretations.
- Example:

<table>
<thead>
<tr>
<th>(p)</th>
<th>(q)</th>
<th>((p \land \neg q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f)</td>
<td>(f)</td>
<td>(f)</td>
</tr>
<tr>
<td>(f)</td>
<td>(t)</td>
<td>(f)</td>
</tr>
<tr>
<td>(t)</td>
<td>(f)</td>
<td>(t)</td>
</tr>
<tr>
<td>(t)</td>
<td>(t)</td>
<td>(f)</td>
</tr>
</tbody>
</table>

- Formulas with the same truth table are equivalent: for formulas \(\phi\) and \(\varphi\), 
  \((\phi \leftrightarrow \varphi)\) or \((\phi \equiv \varphi)\) or \((\phi \text{ iff } \varphi)\)
Problems in propositional logic

• **Evaluate**: given a particular set of variable assignments and a formula, evaluate formula

• **Satisfiability (SAT)**: Given a formula $\phi$, does a set of variable assignments exist that satisfies $\phi$ (makes $\phi$ true)?

• **Validity**: does $\phi$ hold under *all* variable assignments?

• **Examples**: $(p \land \neg p)$ is not satisfiable;
  $(p \lor \neg p)$ is satisfiable and valid;
  $p \land q$ evaluates to *true* if $p, q$ are *true*

Inference and entailment

• Truth of a sentence with variables is relative to its *model* (set of variable assignments)

• **Entailment**: $\alpha \models \beta$ (in every model where $\alpha$ is true, $\beta$ is true)

• **Inference** is finding a desired instance of entailment

• Expression $x$ *logically follows* from a set $S$ of sentences iff every interpretation that satisfies $S$ also satisfies $x$

• **Inference rule**: A validity-maintaining procedure for deriving sentences from other sentences
Inference rules

- Modus ponens:
  $$((p \rightarrow q) \land p) \rightarrow q$$
- *Example:* $$((\text{night} \rightarrow \text{dark}) \land \text{night}) \rightarrow \text{dark}$$
- Modus tollens:
  $$((p \rightarrow q) \land \neg q) \rightarrow \neg p$$
- *Example:* $$((\text{night} \rightarrow \text{dark}) \land \neg \text{dark}) \rightarrow \neg \text{night}$$

2. Predicate logic

- Does propositional logic have limitations?
- How does logic express *properties* of objects?
- How can logic express *some* and *every*?
Predicate calculus

- **Predicate calculus** (first-order logic, FOL) extends propositional calculus by enabling quantifiers and formulas in *predicate* (functional) form
- We address:
  - Predicates
  - Applications
  - Variables and quantifiers
  - Inference

Predicates

- A *predicate* is a Boolean function, i.e., it returns a truth value; it is a *property*
- **Examples:**
  - $(\forall x) \ x < x + 1$  universal quantifier
  - $prime(5)$  predicate denoting a property
  - $(\exists x) \ x > 1$  existential quantifier
- *Arity* of a predicate is its number of parameters
Applications of predicate logic

- Theorems are expressed and proven in FOL
- Algorithm verification uses predicate logic; e.g., \( \text{Sorted}(A) \iff (\forall i < |A|) A[i] \leq A[i + 1] \)
- Theory of formal languages and automata uses FOL
- Artificial intelligence uses FOL to express knowledge and to implement automated reasoning

Variables in predicate logic

- In unquantified expressions \( \text{sum}(x, 5), \text{odd}(x), \text{prime}(x) \), \( x \) is an unbound placeholder that stands for an unknown constant
- “\( \text{sum}(x, 5) = 8 \)”, “\( \text{odd}(x) = \text{true} \)” are meaningless without having a value for \( x \)
- “\( x = 3 \rightarrow \text{sum}(x, 5) = 8 \)”, “\( x = 1 \rightarrow \text{odd}(x) = \text{true} \)” are meaningful and true statements
- Quantifiers (\( \forall \), \( \exists \)) bind variables: e.g., (\( \exists x \)) \( P(x) \)
- Boolean variables are names for assertions
Quantifiers

• For sentence $S$,
  $-(\exists x) S$ is true iff some assignment under $I$
  has an assignment to $x$ s.t. $S$ is true
  $-(\forall x) S$ is true iff $S$ is true for all assignments
  of values to $x$ under $I$

• Correspondence between quantifiers:
  $-(\exists x) P(x) \iff (\neg \forall x) \neg P(x)$
  $-(\forall x) P(x) \iff (\neg \exists x) \neg P(x)$

Multiple quantifiers

• Order does not matter when multiple quantifiers
  are the same

• Examples:
  $(\forall x) (\forall y) P(x,y) \iff (\forall y) (\forall x) P(x,y) \iff$
  $(\forall x, y) P(x, y)$

• The order and scope of quantifiers may determine
  the meaning of an expression:
  $(\forall x) (\exists y) x > y$ Any $x$ is greater than some $y$ (true)
  $(\exists y) (\forall x) x > y$ Some $y$ is less than all $x$ (false)
Transitivity and universal transitivity

- Inference is transitive:
  \[(p \rightarrow q \land q \rightarrow r) \rightarrow (p \rightarrow r)\]

- Universal transitivity:
  \[(((\forall x) P(x) \Rightarrow Q(x)) \land ((\forall x) Q(x) \Rightarrow R(x))) \Rightarrow (\forall x) P(x) \Rightarrow R(x)\]

- Transitive property of equality:
  \[(p = q) \land (q = r) \Rightarrow (p = r)\]

Satisfiability, validity, and consistency

- An interpretation \(I\) satisfies a sentence if it makes the sentence true.

- If \(I\) satisfies sentence \(S\) for every set of variable assignments, then \(I\) is a model of \(S\).

- \(S\) or a set of sentences is satisfiable if some \(I\) satisfies it.

- If \(S\) is true under all interpretations, then \(S\) is valid; \((\forall x) (P(x) \lor \neg P(x))\) is valid.

- If \(S\) is false under all interpretations it is inconsistent; \((\forall x) (P(x) \land \neg P(x))\) is inconsistent.
3. Some proof methods

• How can we be sure that we know what we know?

Proof techniques

• Proof procedure: An inference rule and an algorithm for applying it to a set of sentences to yield a new sentence

• Induction and construction are ways to prove theorems about sets

• Unification and resolution are used in the artificial intelligence fields of automated reasoning and expert systems
Quantifiers and proofs

- **Rule of universal instantiation**: If \( P \) is a property of all of a set’s elements, then it is a property of any particular element

- **Universal modus ponens**:
  \[
  ((\forall x) \ (P(x) \Rightarrow Q(x)) \land P(a)) \Rightarrow Q(a)
  \]

- **Universal modus tollens** (used in proof by contradiction):
  \[
  ((\forall x) \ (P(x) \Rightarrow Q(x)) \land \neg Q(a)) \Rightarrow P(a)
  \]

Proof by induction

- Uses the *induction principle*:
  For a set \( A \) of natural numbers, if:
  - \( 0 \in A \), and
  - \((x \in A) \) implies \((x + 1) \in A\)
  ...then \( A \) is the set of all natural numbers

- To show that predicate \( P \) is true for all natural numbers, an inductive proof shows that \( P \) is true for \( 0 \) and that \( P(x) \Rightarrow P(x + 1) \)

- For example, \( P \) may be the claim that algorithm \( A \) works for a given natural-number input
Sample proof by induction

• *Theorem:* \( \sum_{k=1}^{n} k = n(n + 1) \div 2 \)

• *Base:* \( \sum_{k=1}^{1} k = (1^2 + 1) \div 2 = 1 \)

• *Induction:*
  1. Suppose for some \( n \), \( \sum_{k=1}^{n} k = n(n + 1) \div 2 \)
  2. Then \( \sum_{k=1}^{n+1} k = n(n + 1) \div 2 + (n + 1) \)
     \[= (n^2 + n) \div 2 + (2n + 2) \div 2 \]
     \[= (n^2 + 3n + 2) \div 2 \]
     \[= (n + 1)(n + 2) \div 2 \text{ (algebra)} \]
  3. Hence predicate holds for \( (n + 1) \), hence applies to all natural numbers

Proofs by construction

• Existentially quantified assertions may be proven by *constructive* proof:

• *Construction:* giving a set of instructions to yield an object that has the desired property

• *Example:* To prove that some number is divisible by both \( a \) and \( b \), we construct an instance, such as \( ab \)
Proofs by contraposition and counter example

• *Proof by contraposition* uses principle that the contrapositive of a statement is equivalent to the statement, i.e., \((p \rightarrow q) \iff (\neg q \rightarrow \neg p)\)

• *Proof by counter example* shows that the negation of an existential assertion holds, by exhibiting a case where the assertion fails

• Example: to show that not all numbers are even, we provide the number 3

Common errors in proofs

• Arguing from examples
• Misusing variables; e.g., using one variable to refer to two different values
• Jumping to a conclusion; i.e., skipping proof steps
• Circular reasoning, i.e., using in a proof the assertion that is to be proven
• Confusing quantifiers, e.g., confusing any with some
• Misusing if, e.g., confusing with because
Unification

- **Definition:** finding substitutions that make different sentences look alike
- **Formally:** $\text{Unify}(p, q) = \theta$ where $[\theta / p] = [\theta / q]$  
- **Example:** $\text{Unify}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) = [x / \text{Jane}]$

- **Cases for Unify($E_1$, $E_2$):**
  - $E_1 = E_2$: Return $\{\}$ (no substitution)
  - $E_1$ is a variable: return $\{E_2 / E_1\}$
  - $E_2$ is a variable: return $\{E_1 / E_2\}$
  - Otherwise: decompose $E_1$, $E_2$, return a composition of unification of the components

Backward and forward chaining

- Forward chaining is data driven; applies Modus Ponens; is sound and complete
- Forward chaining generates all possible entailments
- Backward chaining is goal driven; works backward from assertion to be proven
- Deciding entailment with Horn clauses by forward chaining is $O(n)$ in size of knowledge base
Resolution theorem proving

- Resolution: A proof method that shows that the theorem’s negation leads to a contradiction
- Resolution begins by putting premises or axioms in clause form, a normal form using disjunctions of literals (atomic expressions or their negations)
- Intuition: If we know that in one case at least one of a list of assertions holds, and otherwise at least one of a different list of assertions holds, then we know that at least one of the entire list of assertions holds

Resolution rule

- Resolution rule:

\[
(p \lor q_1 \lor q_2 \lor \ldots \lor q_m) \land (\neg p \lor r_1 \lor r_2 \lor \ldots \lor r_n) \Rightarrow q_1 \lor q_2 \lor \ldots \lor q_m \lor r_1 \lor r_2 \lor \ldots \lor r_n
\]
- Complementary literals are used to generate a disjunction out of two disjunctions
Example of resolution

- **Goal**: To prove that Fido will die

- **Predicate form**
  \[(\forall x) \text{dog}(x) \rightarrow \text{animal}(x)\]
  dog(fido)
  \[(\forall y) \text{animal}(y) \rightarrow \text{die}(y)\]
  \[\therefore \text{die}(fido) \text{ by modus ponens}\]

- **Clause form (resolution)**:
  \[-\text{dog}(x) \lor \text{animal}(x)\]
  dog(fido)
  \[-\text{animal}(y) \lor \text{die}(y)\]
  Negate the assertion to be proven: \[-\text{die}(fido)\]
  This leads to a clash after substitutions lead to die(fido)

Complexity of resolution proof

- For \(n\) clauses, at first level of proof tree there are \(n^2\) possible ways to combine them
- Problem is exponential-time
- Heuristics are important to solve problem
- **Strategies**:
  - Breadth-first search: guarantees shortest solution path
  - Set of support
  - Unit preference
  - Linear input form
4. Inductive proofs of correctness

- How do we specify the intent of a program?
- How is software verified reliably?
- Is testing indispensable?
- Do comments always specify *what happens* in a program?

Proofs about algorithms

- An algorithmic problem may be specified by *preconditions and postconditions*, which are assertions that an algorithm fits its specification
- A *loop invariant* assertion at the top of a loop asserts that a weaker version of the postcondition holds at that instant in time
- The loop invariant is used in an inductive proof to show that the postcondition holds after the loop terminates
Logical specification of algorithmic problems

- One part of specification of a problem is an assertion that holds as a postcondition.
- Part of specification is a set of preconditions that must hold for any input.
- Examples:
  - Precondition for binary search, or postcondition for any sort: \((\forall i < |A|)(A[i] \leq A[i+1])\)
  - Postcondition of any search: \((\text{Return value} = \text{true})\) iff \((\exists i \leq |A|) A[i] = \text{key})\)

The terms of a contract

- A precondition of an algorithm tells what must be true about inputs if the algorithm is to work.
- A postcondition asserts that the algorithm has done its job correctly.
- Example:
  ```c
  int sum_of_linear_series(int n)
  // Precondition: n > 0
  {
      int sum = 0;
      for (int i = 1; i <= n; ++i)
          sum += i;
      // Postcondition: sum = 1 + 2 + ... + n
      return sum;
  }```
  ```c
  int sum_of_linear_series(int n)
  // Precondition: n > 0
  {
      int sum = 0;
      for (int i = 1; i <= n; ++i)
          sum += i;
      // Postcondition: sum = 1 + 2 + ... + n
      return sum;
  }```
**Loop invariant:**

An assertion, about the state of an algorithmic process, that is true at the start of each iteration of a loop, and that helps to establish the validity of a postcondition

*Rationale:* If we can show that an assertion is true at the top of the loop and true throughout its execution, then we can show that the assertion is true after the loop terminates.

---

**Loop invariants and induction**

- A *loop invariant* helps take us logically from a precondition to a postcondition

  \[
  \text{sum} \leftarrow 0
  \]

  \[
  \text{for } \text{count} \leftarrow 1 \text{ to } n
  \]

  \[
  \text{LI: } \sum_{k=1}^{\text{count}-1} k
  \]

  \[
  \text{sum} \leftarrow \text{sum} + \text{count}
  \]

- If the loop invariant is valid at the start of this loop’s body on every iteration, then it will also be valid after the loop terminates, when \( \text{count} = n \).
Loop invariant theorem

- **Theorem:** A loop is correct w.r.t. its pre- and postconditions iff
  - The precondition implies that the loop invariant holds for argument 0 \((P(0))\)
  - \((\forall x) P(x) \Rightarrow P(x + 1)\)
  - The loop exit condition eventually holds
  - \(P(x)\) upon loop exit implies that postcondition holds
- **Proof** uses principle of mathematical induction

Proof using loop invariants

- Inductive proof of correctness using loop invariants relies on
  - *Base case:* LI holds before first loop iteration
  - *Inductive case:* If LI holds after \(n\) iterations, then it holds after \((n + 1)\) iterations
  - *Termination:* value tested for exit converges
- \(P(n)\) denotes, “LI holds after \(n^{th}\) iteration”
Largest\((A)\)

**Precondition:** \(A\) is an array

\[
y \leftarrow A[1] \\
i \leftarrow 2 \\
\text{While } i \leq |A| \\
\quad \text{Invariant: } y = \max\{A[1 \ldots i-1]\} \\
\quad \text{if } A[i] > y \\
\quad \quad y \leftarrow A[i] \\
\quad i \leftarrow i + 1 \\
\text{Postcondition: } y = \max\{A[1 \ldots |A|]\}
\]

- Loop invariant resembles the postcondition, but is weaker
- If the invariant holds during the loop iteration, it will hold immediately after the loop stops

**Linear search**

Search\((A, x)\)

\[
y = false \\
\text{for } i \leftarrow 1 \text{ to } |A| \\
\quad \text{// LI: } y = \text{true iff } x \text{ in } A[1 \ldots i-1] \\
\quad \text{if } [i] = x \\
\quad \quad y = \text{true} \\
\quad \text{// post: } y = \text{true iff } x \text{ in } A
\]

- **Can you prove that the postcondition always holds?**
Array-insertion code

void insert(double A[], int& n_elts, double x) {
    // Preconditions: A not full; A is ascending
    int i = n_elts;
    while (x < A[i - 1] && i > 0) {
        // LI: x is less than any value in A[0..n_elts-1]
        // A[0..n_elts-1] is ascending
        A[i] = A[i-1];
        ++i;
    }
    A[i] = x;  // Move elements greater than new_item to the right
    ++n_elts;
    // Postconditions: A contains ‘x’, is ascending,
    // size of A is ‘n_elts’.
}

Insertion-sort(A)

Precondition: A is an array
num_sorted ← 1
Repeat
    Invariant: A[1...num-sorted] is ascending
    i ← num-sorted + 1
    A ← Array-insert (A, num_sorted, A[ i ] )
    num_sorted ← num_sorted + 1
until num_sorted = |A|
Postcondition: A is sorted ascending
5. Formal methods of verification

- What is a language for specification of interactive systems?
- What can be proven about systems that provide services?
- How are $\phi$ and $\varphi$ pronounced?

Specification of problems

- **Algorithmic problem (function):** a set of (input, output) pairs
- **Sequential-interaction problem (service):** a set of dynamically generated streams of I/O pairs
- **Multi-stream interaction problem (mission):** a set of possibly asynchronous I/O streams, possibly in real time and with dynamic creation/destruction of connections
- Spec for multi-stream interaction problem may include number of streams, constraints on computing power of agents, and time constraints
**Hoare triples**

- Let $P$ be a program
- Let $\phi$, $\psi$ be assertions about the state of a program, i.e., about the values of variables at a certain time
- Specification $\langle \phi \rangle P \langle \psi \rangle$ asserts that if $P$ runs starting in a state that satisfies $\phi$, then the state after termination of $P$ will satisfy $\psi$

---

**Partial and total correctness**

- $\langle \phi \rangle P \langle \psi \rangle$ is *satisfied under partial correctness* ($\models_{\text{par}} \langle \phi \rangle P \langle \psi \rangle$) iff postcondition $\psi$ is satisfied after execution of $P$ for all initial states that satisfy $\phi$, whenever $P$ terminates; and $P$ sometimes terminates
- Hence partial correctness may hold even if $P$ sometimes goes into an infinite loop
- *Total correctness* holds iff ($\models_{\text{par}} \langle \phi \rangle P \langle \psi \rangle$) and $P$ always terminates
Loop invariants

- One algorithm-verification method proves that an assertion (the invariant) holds at the top of the loop body on each iteration.
- Technique is to work backward from loop postconditions, derive invariant, prove invariant.
- Proof must also show that loop terminates.
- Example:

```
sum ← 0
for count ← 1 to n
   & Invariant: sum = \sum_{k=1}^{count} k
   sum ← sum + count
& Postcondition: sum = \sum_{k=1}^{n} k
```

Example: Factorial

```
Factorial(x)
\langle x \geq 0 \rangle
y ← 1
t ← 1
while (t \leq x) begin
   LI: y = (t - 1)!
y ← ty
t ← t + 1
end
\langle y = x! \rangle
```
### Basic proof rules

1. **Composition (sequence)**

\[ (\phi) P_1 (\eta) \wedge (\eta) P_2 (\psi) \Rightarrow (\phi) P_1 P_2 (\psi) \]

- Used to prove that a sequence of statements yields a certain postcondition

2. **Assignment**

3. **Branch (if)**

4. **Loop (while)**

### Assignment proof rule

\[ (\psi [E / x]) x \leftarrow E (\psi) \]

where “\( \psi [E / x] \)” means formula \( \psi \) with all occurrences of \( x \) replaced by \( E \);
e.g., if \( \psi = (x > 2) \), and \( E = 4 \), then \( \psi [E / x] \) means “4 > 2”

- Used to prove that a postcondition holds after an assignment statement
### Proof rule for *if*

3. *If*-statement:

\[
\langle \phi \land b \rangle P_1 \langle \psi \rangle \land \langle \phi \land \neg b \rangle P_2 \langle \psi \rangle \Rightarrow \\
\langle \phi \rangle \text{if} b \{P_1\} \text{else} \{P_2\} \langle \psi \rangle
\]

- Used to show that an *if...else* statement that tests \(b\) will satisfy postcondition \(\psi\)

- **Example:** Let \(P_1 = "y \leftarrow (-x)"\), \(P_2 = "y \leftarrow x\)”, \(b = (x < 0)\), \(\psi = (y = |x|)\).

Then we have a proof that an *if...else* statement that tests \(x < 0\), correctly computes \(y = \text{abs}(x)\)

### Example with *If*

**Abs(x)**

If \(x \geq 0\)

- \(B: x \geq 0\)

  \[
y \leftarrow x \quad > \psi: y = \text{Abs}(x) \quad \text{[IF rule (3)]}
\]

else

- \(B: x < 0\)

  \[
y \leftarrow (-x)
\]

  \[
  > \psi: y = \text{Abs}(x) \quad \text{[IF rule (3)]}
\]

>Post (\(\psi\)): \(y = \text{Abs}(x)\)
### While statement proof rule

4. \( \langle \phi \land b \rangle \ P \langle \phi \rangle \ \Rightarrow \ \langle \phi \rangle \ \text{while} \ b \ \{P\} \ \langle \phi \land \lnot b \rangle \)

- \( \phi \) is called the “invariant” here
- Rule is used to show that if exit condition \( b \) and loop invariant \( \phi \) hold before loop starts, then postcondition \( \phi \) will hold after loop exits
- **Example:**
  Let \( b = (i < x) \), \( P = \text{“} y \leftarrow 2y, \ i \leftarrow i - 1 \text{”} \),
  \( \phi = (y = 2^i) \); then
  \( \langle \phi \land b \rangle \ P \langle \phi \rangle \ \Rightarrow \ \langle \phi \rangle \ \text{while} \ b \ \{P\} \ \langle \phi \land \lnot b \rangle \)

### While example algorithm

The following algorithm is correct under proof rules 1 (sequence), 2 (assignment), and 4 (while)

**Power-of-2(x)**

\( y \leftarrow 1 \)
\( i \leftarrow 0 \)
While \( i < x \)
  > \text{Li: } y = 2^i
  \( y \leftarrow 2y \)
  \( i \leftarrow i + 1 \)
>Post: \( y = 2^x \ [y = 2^i \land x = i] \)
Intuition of if and while rules

If-statement rule:
- if some precondition and postcondition hold for a certain pair of statements ($C_1$ and $C_2$)
- ...then the same pair of assertions holds for an if-else statement executing either $C_1$ or $C_2$

While-statement rule:
- if a precondition and postcondition hold for a certain statement ($C$),
- ...then they hold for a loop that iterates $C$.

Loop invariants and proofs

- E. Dijkstra: Understanding a while loop is equivalent to understanding what its invariant is
- An invariant that is useful for proving $\preceq \phi \puceq P \preceq \psi$ will express the unchanging relationship between variables manipulated by the body of a while statement
- Technique: Push postcondition backwards through a while statement body
**Proof example**

\[
\langle a, b \geq 0 \rangle \\
y \leftarrow a \\
z \leftarrow 0 \\
\langle y = a + z \land b > 0 \rangle \\
\text{While } z < b \text{ do} \\
\langle y = a + z \land b > 0 \rangle \\
y \leftarrow \text{succ}(y) \\
\langle y = a + z + 1 \rangle \\
z \leftarrow \text{succ}(z) \\
\langle y = a + z \rangle \\
\langle z = b \rangle \\
\langle z = b \land y = a + b \rangle
\]

- This code adds \(a\) and \(b\), using the increment operation.
- Loop invariant is \(y = a + z\).
- When \(z\) reaches \(b\), \(y\) is the sum of \(a\) and \(b\).

---

**Specification and verification of interactive systems**

- *Certain properties* of reactive systems may be verified or invalidated by use of temporal logic and a technique called *model checking*.
- Model checking uses *computation paths*.
- Proofs may be in Computation Tree Logic (CTL).
Temporal logic

- Temporal logic reasons about dynamic processes
- Used in verification of *reactive systems* in which input and output alternate
- In interactive systems, computation may be pictured as infinite
- *Model checking*, developed in U.S. and France in 1980s, uses notations Computation Tree Logic (CTL), CTL*, LTL, others
- Current primary use of model checking may be to verify hardware designs

Kinds of properties verified

- *Safety* (undesirable states won’t be visited)
- *Example*: Deadlock will never occur; microwave oven will not enter (power-on, door-open) state
- *Liveness* (desired states will obtain)
- *Example*: All resource requests will eventually be fulfilled
- Properties apply to *computational paths* through the tree of all possible paths
References


