2. Formal verification of algorithms

1. Loops and recurrences
2. Inductive proofs of correctness
3. Formal proof methods and Hoare triples

Inquiry

Is it worthwhile to prove claims about the behaviors of systems?
2. Verification

Topic objective

2. Explain and apply the method of proving correctness of an algorithm inductively, using postcondition and loop invariant.

Background outcomes

2.0a Write a direct proof**
2.0b Describe mathematical induction**
2.0c Use induction to prove a theorem about numbers**
2.0d Write and explain an array traversal**
2.0e Explain the linear-search algorithm**
2.0f Explain a sorting algorithm**
2.0g Write a recursive method for a while loop**
2.0h Write an algorithm with nested loops**
1. Loops and recurrences

- What is recursion?
- Have you converted between recursive and iterative code? What are pros and cons?
- Which is true?
  - Some loops cannot be coded recursively
  - Any while loop can be written as a recursive method
  - Any recursive method can be converted to a while loop

Peano’s axioms: definition by induction

1. \( (base) \) 0 is a natural number \((0 \in \mathbb{N})\)
2. \( (inductive) \) Every natural number \(n\) has a unique successor, \(n'\), also a natural number \((\forall n \in \mathbb{N}) n' \in \mathbb{N}\)
3. All natural numbers follow (1) or (2) \((\forall n \in \mathbb{N}) n = 0 \lor (\exists m \in \mathbb{N}) n = m'\)

- \textit{Significance}: These axioms, or assumptions, provide a formal logical basis to work with counting numbers. \textit{Note}: (2) is recursive
- Computation is a formal way to manipulate numbers and objects representable by them.
Recursion and algorithms

- Proposition: any counted-loop algorithm may be expressed recursively
- Proof (by construction):

Iterative:  
\[ A(x, n) \]
1. \( y \leftarrow x \)
2. for \( i \leftarrow 1 \) to \( n \)
   - \( y \leftarrow f(y) \)
3. Return \( y \)

> Post: \( y = f^n(x) \)

Recursive:
\[ A(x, n) \]
1. if \( n = 0 \) return \( x \)
2. else return \( A(x, n - 1) \)

Recurrence relations

- Definition: A set of tuples that define a sequence by relating an element to some of its predecessors
- Example: Fibonacci sequence, where \( f_1 = f_2 = 1, f_n = f_{n-1} + f_{n-2} \) for \( n \geq 3 \)
- Recurrence relations define the computable functions \( f: \mathbb{N} \rightarrow \mathbb{N}, \phi: \Sigma^* \rightarrow \Sigma^* \)
- Notation: as above or as on next slide
A recurrence may define a function

\[ \text{sum}(a, b) = \begin{cases} 
    a & \text{if } b = 0 \\
    \text{sum} \left( \text{succ}(a), \text{pred}(b) \right) & \text{otherwise}
\end{cases} \]

\[ \text{product}(a, b) = \begin{cases} 
    0 & \text{if } b = 0 \\
    a + \text{product}(a, b-1) & \text{otherwise}
\end{cases} \]

• These recurrences suggest algorithms to compute the functions defined

Multiplication and exponentiation

\[ \text{factorial}(n) = \begin{cases} 
    1 & \text{if } n \leq 1 \\
    n \times \text{factorial}(n - 1) & \text{otherwise}
\end{cases} \]

\[ \text{Pow}(a,b) = \begin{cases} 
    1 & \text{if } b = 0 \\
    a \times \text{Pow}(a, b-1) & \text{otherwise}
\end{cases} \]
Recurrences that return strings

Where “+” denotes concatenation and $\lambda$ is the null string (language at right)

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 0 + f(x - 1) & \text{otherwise} \end{cases} \quad 00^*$$

$$f(x) = \begin{cases} \lambda & \text{if } x = 0 \\ 01 & \text{if } x = 1 \\ 0 + f(x - 1) + 1 & \text{otherwise} \end{cases} \quad 0^n1^n$$

Computing *max* of an array

**Max(A)**

> *Precondition: A is a nonempty array*

$y \leftarrow A[1]$

if $|A| = 0$ throw exception

if $|A| = 1$

return $A[1]$


return $A[1]$

else return $Max(A[2 \ldots |A|])$
2. Verification

Defining \textit{max} for arrays

(a) \textit{Set definition:}
\[ \text{max}(A) = x \in A \text{ s.t. } (\forall i \leq |A|) \ x \geq A[i] \]
(b) \textit{Algorithmic definition} for max elt. of array:
\[
\text{Max-elt}(A) = \begin{cases} 
\uparrow & \text{if } |A| = 0 \quad \uparrow \text{means function} \\
A[1] & \text{if } |A| = 1 \quad \text{undefined} \\
\text{Max-elt}(A[2 \ldots |A|]) & \text{otherwise}
\end{cases}
\]

Converting associative binary operators to \(n\)-ary operators

- If \(\oplus\) is an associative binary operator on values in set \(S\) (stored as an array), then \(f : S^n \rightarrow S\) is the corresponding operator on arrays of \(S\), where
\[
f(A) = \begin{cases} 
\uparrow & \text{if } |A| = 0 \\
A[1] & \text{if } |A| = 1 \\
A[1] \oplus f(A[2 \ldots |A|]) & \text{otherwise}
\end{cases}
\]
Subtopic outcome

2.1a Write a simple recurrence**
2.1b Derive a *while* loop from a recurrence

2. Loop invariants and induction

- What’s *induction*?
- Who here has worked with loop invariants?
Assertions and correctness

- A comment that is an assertion tells not what occurs, but something about values of variables and expressions
- Valid assertions can help us establish that our code does what we claim
- Chief tools for showing algorithm correctness:
  - Preconditions
  - Postconditions
  - Loop invariants

Pre- and post- conditions

- Precondition: An assertion about values of inputs and variables before an algorithm executes
- Postcondition: An assertion that is claimed to hold after execution if the precondition holds
- Example: Adding a series of numbers
  - Precondition: total is 0
  - Postcondition: total stores the sum of all input values
The terms of a contract

- A *precondition of a method* tells what must be true about parameters if the method is to work.
- A *postcondition* asserts that the method 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;
}
```

Semantics and correctness

- *Syntax:* Form of language expression
- *Syntactic correctness* of a program: Adherence to a set of grammar rules
- *Semantics:* Meaning (for programs, meaning is what the program does)
- *Semantic correctness:* Satisfaction of a program’s specifications in all cases
- Compiler checks syntax; testing and verification check semantics
Conditions for algorithm correctness

For all inputs:
1. Algorithm eventually halts. (Thus, values that control termination of a loop must converge)
2. Algorithm must have a result that satisfies postcondition

Goal of a correctness proof

- To use the precondition and the code to show by inference that the postcondition holds
- Example:
  input \( r, h \)
  \( a \leftarrow \pi r^2 \)
  \( y \leftarrow ah \)
- Theorem (postcondition): \( y \) is the volume of a cylinder of radius \( r \) and height \( h \)
- Proof: the two assignments give \( y \) the known volume of a cylinder, i.e., \( \text{end-area} \times \text{height} \)
Loop invariants

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

\[
\text{sum} \leftarrow 0 \\
\text{for count} \leftarrow 1 \text{ to } n \\
\text{LI: } \text{sum} = \sum_{k=1}^{\text{count}-1} k \\
\text{sum} \leftarrow \text{sum} + \text{count}
\]

- 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:**

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.
How to prove a loop postcondition

• Find an assertion that
  – Is similar to the postcondition, but weaker;
  – Is true at the top of the loop, each iteration

• Example:
  input x
  for i ← 1 to 3 do
    > LI: y = x(i − 1)
    y ← y + x
    > Post: y = 3x

Proof by induction that y = 3x

• Base:
  y = 0 on first iteration

• Induction:
  If y = x (i − 1) on the ith iteration, then y = x
  (m − 1) on the (m = i + 1)th iteration

• Conclusion:
  After three iterations, y = 3x
Proofs by induction

- Use the *principle of mathematical induction*: 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 $(P$ true for $x)$ implies that $P$ is true for $(x + 1)$

- Here, $P$ may be the claim that algorithm $A$ works for a given natural-number input

Induction using loop invariants

- 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”
2. Verification

**Largest(A)**

*Precondition:* A is an array

\[ y \leftarrow A[1] \]
\[ i \leftarrow 2 \]

While \[ i \leq |A| \]

*Invariant:* \[ y = \max\{A[1 \ldots i - 1]\} \]

\[ \text{if } A[i] > y \]
\[ y \leftarrow A[i] \]
\[ i \leftarrow i + 1 \]

*Postcondition:* \[ y = \max\{A[1 \ldots |A|]\} \]

*Observation:* If the invariant holds during the loop iteration, it will hold immediately after the loop stops.

**Linear search**

**Search(A, x)**

\[ y = false \]

for \[ i \leftarrow 1 \text{ to } |A| \]

// Li: \[ y = true \text{ iff } x \text{ in } A[1 \ldots i - 1] \]

\[ \text{if } [i] = x \]
\[ y = true \]

// post: \[ y = true \text{ iff } x \text{ in } A \]

\[ \text{Search}(A, x) = \]
\[ \begin{cases} 
false & \text{if } |A| = 0 \\
true & \text{if } A[1] = x \\
\text{Search}(A[2..|A|], x) & \text{otherwise}
\end{cases} \]
Example: sum of array elements

Theorem: The algorithm below sums array $A$:

\[
\begin{align*}
\text{sum} & \leftarrow 0 \\
\text{for } i & \leftarrow 1 \text{ to } |A| \\
\text{sum} & \leftarrow \text{sum} + A[i]
\end{align*}
\]

Proof:
1. For any $i$, let $P(i)$ be the assertion that after $i$ iterations, $\text{sum}$ stores sum of all elements to $i$
2. If $P$ holds after $i$ iterations, it will after $(i + 1)$ iterations, because each iteration adds $A[i]$
3. By induction, after $|A|$ iterations, $\text{sum}$ stores the sum of all array elements

Correctness problem

- Prove that this algorithm multiplies $x$, $y$:

\[
\begin{align*}
\text{Product (x, y)} \\
\text{result} & \leftarrow 0 \\
\text{for } i & \leftarrow 1 \text{ to } x \\
\text{result} & \leftarrow \text{result} + y \\
\text{Return result}
\end{align*}
\]

- What are precondition, postcondition, loop invariant?
- How does loop invariant ensure postcondition is valid?
**Is-largest \((A, x)\)**

**Precondition:** \(A\) is an array, \(x\) is of \(A\)’s base type

If \(\text{Size}(A) = 0\)

\[
\text{return } false
\]

\[
\text{max } \leftarrow A[1]
\]

For \(k \leftarrow 2\) to \(\text{Size}(A)\)

\[
[\text{Loop invariant: } \text{max} \text{ is largest in } A[1..k-1].]
\]

if \(A[k] > \text{max}\)

\[
\text{max } \leftarrow A[k]
\]

**Postcondition:** \(\text{max} \text{ is value of lgst element of } A.\)

if \(\text{max} = x\)

\[
\text{return } true
\]

otherwise

\[
\text{return } false
\]

---

**Search-stack \((S, key)\)**

**Preconditions:** \(S\) is a stack, \(key\) is of \(S\)’s base type;

\(aux\) is empty stack

**Postconditions:**

(a) returns \(true\) if \(key\) is in \(S\); otherwise \(false\);

(b) \(S\) is in same state as at start.

\[
\text{found } \leftarrow false
\]

while not empty(\(S\))

\[
[\text{Loop invariant: } \text{found} \text{ is true if and only if } key \text{ is in } aux]
\]

\[
\text{test } \leftarrow \text{Pop}(S)
\]

if \(\text{test} = key\)

\[
\text{found } \leftarrow true
\]

\[
\text{Push}(\text{aux}, \text{test})
\]

**Postcondition:**

(a) \(\text{found} \) is true iff \(key\) in \(aux\).

while not empty \((aux)\)

\[
[\text{Loop invariant: } aux \text{ reversed + } S \text{ contains original contents of } S]
\]

\[
\text{Push}(S, \text{Pop}(aux))
\]

**Postcondition:**

(b) \(S\) in same state as at start of algorithm.

return \(\text{found}\)
Limits on use of postconditions

- Note that for postcondition to be used, algorithm must not alter values of any parameters
- Example:
  \[\text{Sum}(a, b)\]
  \[
  \begin{align*}
  y & \leftarrow a \\
  \text{while } b > 0 & \quad \begin{align*}
  a & \leftarrow \text{succ}(a) \\
  b & \leftarrow b - 1
  \end{align*} \\
  \text{// Post: } y = a + b
  \end{align*}
  \]
  Misleading! \( b = 0 \) now!

Nested-loop correctness is proven in two steps

- Examples of algorithms with nested loops:
  - Finding duplicates in an array
  - Sorting algorithms
  - Order statistic (kth-largest array element)
- A nested loop is verified by verifying each loop
Array-insertion code

```c
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[i-1 .. n_elts-1]
        // A[0..n_elts-1] is ascending
        A[i] = A[i-1];
        ++i;
    }
    A[i] = x;
    ++n_elts;
    // Postconditions: A contains 'x'
    // A is ascending, size of A is 'n_elts'.
}
```

Insertion-sort(A)

**Precondition:** A is an array

```
um_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
Bubble-sort(A)

//Precondition: A is an array
np ← 0  // number of passes
Repeat
    //Invariant: Rightmost (np) elements are sorted;
    //if swapped is false and np > 0, then A is sorted
    swapped ← false
    for i ← 1 to size(A) - 1
        //Invariant: (∀x ∈ A[0..i-1]) A[ i ] ≥ x
        if A[ i ] > A[ i + 1]
            swap(A[ i ], A[ i + 1 ])
            swapped ← true
    np ← np + 1
until swapped is false
//Postcondition: A is sorted ascending

Subtopic outcomes

2.2a Explain the basic terminology of formal verification**

2.2b Given an algorithm and a loop invariant, explain why the algorithm is correct**

2.2c Given an algorithm and a postcondition, write an appropriate loop invariant*

2.2d Write an algorithm and prove its correctness*
3. Formal proof methods and Hoare triples

- How much does mathematical knowledge depend on formal rules?
- Is it worthwhile in software engineering to give extra attention to the correctness of some processes?

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 $\psi$
Example: Factorial

**Factorial**

\[ \langle x \geq 0 \rangle \]

\[ y \leftarrow 1 \]

\[ t \leftarrow 1 \]

while \((t \leq x)\) begin

\[ \text{Loop invariant: } y = (t - 1)! \]

\[ y \leftarrow ty \]

\[ t \leftarrow t + 1 \]

end

\[ \langle y = x! \rangle \]

---

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
Basic proof rules

1. Composition (sequence):
   \[ \langle \phi \rangle P_1 \langle \eta \rangle \land \langle \eta \rangle P_2 \langle \psi \rangle \Rightarrow \langle \phi \rangle P_1 P_2 \langle \psi \rangle \]
   • Used to prove that a sequence of statements yields a certain postcondition

2. Assignment:
   \[ \langle \psi [E/x] \rangle x \leftarrow E \langle \psi \rangle \]
   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 \( P_1 = \langle y \leftarrow (\neg x) \rangle \), \( P_2 = \langle y \leftarrow x \rangle \), \( b = (x < 0) \), \( \psi = (y = |x|) \), then we have a proof that an if...else statement that tests \( x < 0 \), correctly computes \( y = \text{abs}(x) \)
Example with *If*

\[ \text{Abs}(x) \]

**Pre \((\phi)\):**
- If \( x \geq 0 \)
  - \( B: x \geq 0 \)
  - \( y \leftarrow x \quad \psi: y = \text{Abs}(x) \) [IF rule (3)]
- else
  - \( B: x < 0 \)
  - \( y \leftarrow (-x) \quad \psi: y = \text{Abs}(x) \) [IF rule (3)]

**Post \((\psi)\):** \( y = \text{Abs}(x) \)

---

**While statement proof rule**

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

- \( \phi \) 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 \( b = (i < x) \), \( P = “y \leftarrow 2y, \ i \leftarrow i - 1” \),
  - \( \phi = (y = 2^i) \); then
  - \( \langle \phi \land b \rangle P \langle \phi \rangle \Rightarrow \langle \phi \rangle \quad \text{while } b \{ P \} \langle \phi \land \neg b \rangle \)
**While example algorithm**

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

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

\[
\begin{align*}
y &\leftarrow 1 \\
i &\leftarrow 0 \\
\text{While } i < x \\
&> Li: y = 2^i \\
y &\leftarrow 2y \\
i &\leftarrow i + 1 \\
&> Post: y = 2^x \text{ [because } y = 2^i \land x = i \text{]}
\end{align*}
\]

**Intuition**

**If-statement rule:**
- if some precondition and postcondition hold for a certain pair of statements \((C_1 \text{ and } C_2)\)
- \(\ldots\) then the same pair of assertions holds for an \textit{if-else} statement executing either \(C_1\) or \(C_2\)

**While-statement rule:**
- if a precondition and postcondition hold for a certain statement \((C)\),
- \(\ldots\) 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 \( \langle \phi \rangle P \langle \psi \rangle \) 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

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

- 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 \( b \)
Recurrences and performance

• Recurrences express functions, algorithms, and the running-time functions for the algorithms
• A recurrence may be used to define a function inductively (recursively) by giving the base-case definition and the recursive-case definition
• Likewise a time recurrence defines the time function for a recursive algorithm
• A formula defines how to convert from the recurrence that expresses the algorithm to a time recurrence

Subtopic outcome

2.3 Explain formal rules for algorithm correctness proofs
References
