Chapter 3 (Part 3): Mathematical Reasoning, Induction & Recursion

- Recursive Algorithms (3.5)
- Program Correctness (3.6)

Recursive Algorithm (3.5)

- Goal: Reduce the solution to a problem with a particular set of input to the solution of the same problem with smaller input values
- Example:

$$gcd(a,b) = gcd(b \textbf{ mod } a, a)$$

$$gcd(4,8) = gcd(8 \textbf{ mod } 4, 4) = gcd(0,4) = 4$$

Recursive algorithm (.5) (cont.)

◆Definition 1:

> An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.

◆Example: Give a recursive algorithm for computing a^n; $a \neq 0$, $n>0$

Solution: $a^{n+1} = a*a^n$ for $n>0$ $\quad a^0 = 1$

Recursive algorithm (.5) (cont.)

◆A recursive algorithm for computing a^n

```
Procedure power(a: nonzero real number,
           n: nonnegative integer)
if n = 0 then power(a, n):= 1
Else power(a,n) := a * power(a, n-1)
```

Recursive algorithm (.5) (cont.)

- Example: Give a recursive algorithm for computing the greatest common divisor of two nonnegative integers a and b (a<b)

Solution:

gcd(a,b) = gcd(b mod, a)
and the condition gcd(0,b) = b (b>0).

Recursive algorithm (.5) (cont.)

- A recursive algorithm for computing gcd(a,b)

```
procedure gcd(a, b: nonnegative integers
         with a<b)
if a = 0 then gcd(a,b) := b
else gcd(a,b) := gcd(b mod a, a)
```

Recursive algorithm (.5) (cont.)

- Example: Express the linear search as a recursive procedure: search x in the search sequence $a_i a_{i+1} \ldots a_j$.
- A Recursive linear search algorithm

```
procedure search(i, j, x)
if a_i = x then
  location := i
else if i = j then
  location := 0
else
  search(i + 1, j,x)
```

Recursive algorithm (.5) (cont.)

- Recursion & iteration
  - We need to express the value of a function at a positive integer in terms of the value of the function at smaller integers
  - Example: Compute a recursive procedure for the evaluation of n!

Recursive algorithm (.5) (cont.)

- A recursive procedure for factorials

```
procedure factorial(n: positive integer
if n = 1 then
      factorial(n) := 1
else
      factorial(n) := n * factorial(n - 1)
```

Recursive algorithm (.5) (cont.)

- Recursion & iteration (cont.)
  - However, instead of reducing the computation to the evaluation of the function at smaller integers, we can start by 1 and explore larger in an iterative way

Recursive algorithm (.5) (cont.)

- An iterative procedure for factorials

```
procedure iterative factorial(n: positive
integer)
x := 1
for i := 1 to n
      x := i * x
{x is n!}
```

Recursive algorithm (.5) (cont.)

- Recursion & iteration (cont.)
  - An other example is the recursive algorithm for Fibonacci numbers

```
procedure fibonacci(n: nonnegative integer)
if n = 0 then fibonacci(0) := 0
else if n = 1 then fibonacci(1) := 1
else fibonacci(n) := fibonacci(n - 1) +
      fibonacci(n - 2)
```

Recursive algorithm (.5) (cont.)

- An iterative algorithm for computing Fibonacci numbers

```
procedure iterative fibonacci(n: nonnegative integer)
if n = 0 then y := 0
else
begin
        x := 0
        y := 1
        for i := 1 to n - 1
        begin
                z := x + y
                x := y
                y := z
        end
end
{y is the nth Fibonacci number}
```

Program Correctness (3.6)

- Introduction

  Question: How can we be sure that a program always produces the correct answer?

  - The syntax is correct (all bugs removed!)
  - Testing a program with a randomly selected sample of input data is not sufficient
  - Correctness of a program should be proven!
  - Theoretically, it is never possible to mechanize the proof of correctness of complex programs
  - We will cover some of the concepts and methods that prove that "simple" programs are corrects

Program Correctness (3.6) (cont.)

- Program verification
  - To prove that a program is correct, we need two parts:
    1. For every possible input, the correct answer is obtained if the program terminates
    2. The program always terminates

Program Correctness (3.6) (cont.)

- Program verification (cont.)
  - Definition 1:

> A program, or program segment, S is said to be partially correct with respect to the initial assertion p and the final assertion q if whenever p is true for the input values of S and S terminates, then q is true for the output values of S. The notation p{S}q indicates that the program, or program segment, S is partially correct with respect to the initial assertion p and the final assertion q.

Program Correctness (3.6) (cont.)

- This definition of partial correctness (has nothing to do whether a program terminates) is due to Tony Hoare
- Example: Show that the program segment

```
y := 2
z := x + y
```

is correct with respect to the initial assertion p: x =1 and the final assertion q: z =3.

Solution: p is true $\Rightarrow$ x = 1 $\Rightarrow$ y := 2 $\Rightarrow$ z := 3 $\Rightarrow$ partially correct w.r.t. p and q

Program Correctness (3.6) (cont.)

- Rules of inference
  - Goal: Split the program into a series of subprograms and show that each subprogram is correct. This is done through a rule of inference.
  - The program S is split into 2 subprograms S_1 and S_2 ($S = S_1; S_2$)
  - Assume that we have S_1 correct w.r.t. p and q (initial and final assertions)
  - Assume that we have S_2 correct w.r.t. q and r (initial and final assertions)

Program Correctness (3.6) (cont.)

- It follows:

"if p is true $\wedge$ (S_1 executed and terminates) then q is true"
"if q is true $\wedge$ (S_2 executed and terminates) then r is true"
"thus, if p = true and S = S_1; S_2 is executed and terminates then r = true"

This rule of inference is known as the composition rule.

- It is written as:

$p \{S_1\} q$
$q \{S_2\} r$

$\therefore p \{S1; S2) r$

Program Correctness (3.6) (cont.)

- Conditional Statements
  - Assume that a program segment has the following form:

1. "if *condition* then S" where S is a block of statement

   Goal: Verify that this piece of code is correct

   Strategy:

a) We must show that when p is true and *condition* is also true, then q is true after S terminates
b) We also must show that when p is true and *condition* is false, then q is true

Program Correctness (3.6) (cont.)

- We can summarize the conditional statement as:

  $(p \wedge \textit{condition})\ \{S\}q$
  $(p \wedge \neg \textit{condition}) \Rightarrow q$

  $\therefore p\ \{\textbf{if}\ \textit{condition}\ \textbf{then}\ S)\ q$

- Example: Verify that the following program segment is correct w.r.t. the initial assertion T and the final assertion $y \geq x$

```
if x > y then
    y:= x
```

Solution:

a) If T = true and x>y is true then the final assertion $y \geq x$ is true
b) If T = true and x>y is false then $x \leq y$ is true $\Rightarrow$ final assertion is true again

Program Correctness (3.6) (cont.)

2. "if *condition* then S_1 else S_2"
   if *condition* is true then S_1 executes; otherwise S_2 executes

   This piece of code is correct if:

a) If p = true $\wedge$ *condition* = true $\Rightarrow$ q = true after S_1 terminates
b) If p = true $\wedge$ *condition* = false $\Rightarrow$ q = true after S_2 terminates

$(p \wedge \textit{condition})\ \{S_1\}q$
$(p \wedge \neg \textit{condition})\ \{S_2\}q$

$\therefore p\ \{\textbf{if}\ \textit{condition}\ \textbf{then}\ S_1\ \textbf{else}\ S_2)\ q$

Program Correctness (3.6) (cont.)

- Example: Check that the following program segment

```
if x < 0 then
        abs := -x
else
        abs := x
```

is correct w.r.t. the initial assertion T and the final assertion abs = |x|.

Solution:

a) If T = true and (x<0) = true $\Rightarrow$ abs := -x; compatible with definition of abs

b) If T = true and (x<0)= false $\Rightarrow$ (x $\geq$ 0) = true $\Rightarrow$ abs := x; compatible with abs definition.

Program Correctness (3.6) (cont.)

- Loop invariants
  - In this case, we prove codes that contain the while loop: "while *condition* S"
  - Goal: An assertion that remains true each time S is executed must be chosen Such an assertion is called a loop invariant. In other words, p is a loop invariant if:

$$(p \wedge \textit{condition})\ \{S\}p \text{ is true}$$

  - If p is a loop invariant, then if p is true before the program segment is executed p and $\neg$*condition* are true after termination, if it occurs. We can write the rule of inference as: $(p \wedge \textit{condition})\ \{S\}p$

$$\therefore p\ \{\textbf{while}\ \textit{condition}\ S\}\ (\neg \textit{condition} \wedge p)$$

Program Correctness (3.6) (cont.)

- Example: Determine the loop invariant that verifies that the following program segment terminates with factorial = n! when $n \geq 0$.

```
i := 1
factorial := 1
While i < n
      begin
            i := i + 1
            factorial := factorial * i
      end
```

Program Correctness (3.6) (cont.)

Solution:

Choose p = (factorial = i! $\wedge$ ($i \leq n$))

a) Prove that p is in fact a loop invariant
b) If p is true before execution, p and $\neg$condition are true after termination
c) Prove that the **while** loop terminates

Program Correctness (3.6) (cont.)

a) P is a loop invariant:

Suppose p is true at the beginning of the execution of the **while** loop and the **while** *condition* holds;

$\Leftrightarrow$ factorial = i! $\wedge$ i < n

$i_{new} = i + 1$

$factorial_{new} = factorial * (i + 1) = (i + 1)! = i_{new}!$

Since $i < n \Rightarrow i_{new} = i + 1 \leq n$

$\Rightarrow$ p true at the end of the execution of the loop

$\Rightarrow$ p is a loop invariant

Program Correctness (3.6) (cont.)

b) Before entering the loop, $i = 1 \leq n$ and
factorial :=1 = 1! = i! $\Rightarrow (i \leq n) \wedge$ (factorial = i!) = true
$\Rightarrow$ p = true

Since p is a loop invariant $\Rightarrow$ through the inference rule, if the **while** loop terminates $\Rightarrow$ p = true and i < n false
$\Rightarrow$ i = n and factorial = i! = n!

c) **While** loop terminates:

when the program starts, i = 1
after (n –1) **traversals** of the loop $\Rightarrow$ i = n $\Rightarrow$ stop loop.