

David M. Keil  
Framingham State University

## 2. Formal verification

1. Assertions and correctness
2. Loop invariants and induction
3. Formal proof methods and Hoare triples

*Readings:* Drozdek and Simon;  
Hamburger and Richards; Huth and Ryan

David Keil    Analysis of Algorithms    2. Verification    1/12    1

## Topic objectives

- 2a. Explain the basic terminology of formal verification
- 2b. Given an algorithm commented with a postcondition, write an appropriate loop invariant
- 2c. Given an algorithm commented with a loop invariant, explain why the algorithm is correct

David Keil    Analysis of Algorithms    2. Verification    1/12    2

## Inquiry

Does it pay to *prove* claims about the behaviors of systems?

### 1. 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: *preconditions*, *postconditions*, *loop invariants*

## Pre- and post- conditions

- *Precondition*: An assertion about what the state of inputs is 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*:

```
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

## 2. Loop invariants and induction

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

sum  $\leftarrow$  0

for *count*  $\leftarrow$  1 to *n*

1 + 2 + ... + *count* - 1

$$\text{LI: } \textit{sum} = \sum_{k=1}^{\textit{count}-1} k$$

*sum*  $\leftarrow$  *sum* + *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 *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

## 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  $(P$  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  $n^{\text{th}}$  iteration”

**Largest(A)****Precondition:**  $A$  is an array $y \leftarrow A[1]$  $i \leftarrow 2$ While  $i \leq |A|$ **Invariant:**  $y = \max\{A[1 \dots i-1]\}$ if  $A[i] > y$  $y \leftarrow A[i]$  $i \leftarrow i+1$ **Postcondition:**  $y = \max\{A[1 \dots |A|]\}$ 

- Loop invariant is an assertion of the same form as the postcondition, except weaker
- We use the observation that if the invariant holds during the loop iteration, it will hold immediately after the loop stops

**Linear search****Search(A, x)** $y = \text{false}$ for  $i \leftarrow 1$  to  $|A|$ // LI:  $y = \text{true}$  iff  $x$  in  $A[1 \dots i-1]$ if  $A[i] = x$  $y = \text{true}$ // post:  $y = \text{true}$  iff  $x$  in  $A$ **Search(A, x) =**

$$\begin{cases} \text{false} & \text{if } |A| = 0 \\ \text{true} & \text{if } A[1] = x \\ \text{Search}(A[2..|A|], x) & \text{otherwise} \end{cases}$$

## Correctness of *strcpy*

```
void strcpy(char* dest, char* source)
{
    int i=0;
    do
        dest[i] = source[i];
    while (source[i++] != '\0');
}
```

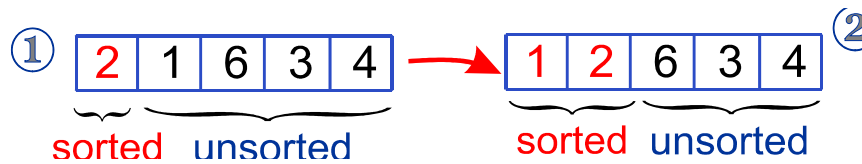
- **Precondition:** *dest* has space for all chars in *source* available for memory writing
- **Postcondition:** All characters from address *dest* until '\0' match those in *source*
- **Loop invariant:** All characters from address *dest* until *dest* [*i* - 1] match *source*[0..*i* - 1], or else *i* = 0
- Argument that postcondition holds:
  - Each *dest*[*i*] gets its value from *source*[*i*]
  - Eventually a '\0' in *source* will terminate loop

## 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[i-1 .. n_elts-1]
        // A[0..n_elts-1] is ascending
        A[i] = A[i-1]; // Move elements greater than
        ++i;           // new_item to the right
    }
    A[i] = x; // Drop new_item in place
    ++n_elts;
    // Postconditions: A contains 'x'
    // A is ascending, size of A is 'n_elts'.
}
```

**Insertion-sort(A)****Precondition:** *A* is an array $num\_sorted \leftarrow 1$ 

Repeat

**Invariant:**  $A[1 \dots num\_sorted]$  is ascending $i \leftarrow num\_sorted + 1$  $A \leftarrow \text{Array-insert}(A, num\_sorted, A[i])$  $num\_sorted \leftarrow num\_sorted + 1$ until  $num\_sorted = |A|$ **Postcondition:** *A* is sorted ascending**Bubble-sort(A)****//Precondition:** *A* is an array $np \leftarrow 0$  // number of passes

Repeat

**//Invariant:** Rightmost ( $np$ ) elements are sorted;//if *swapped* is false and  $np > 0$ , then *A* is sorted $swapped \leftarrow false$ for  $i \leftarrow 1$  to  $size(A) - 1$ **//Invariant:**  $(\forall x \in A[0..i-1]) A[i] \geq x$ if  $A[i] > A[i + 1]$      $swap(A[i], A[i + 1])$      $swapped \leftarrow true$  $np \leftarrow np + 1$ until *swapped* is false**//Postcondition:** *A* is sorted ascending

## Correctness problem

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

```
Product (x, y)
result ← 0
For i ← 1 to x
    result ← result + y
Return result
```

- 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  $\text{Size}(A) = 0$

    return *false*

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

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

[Loop invariant: max is largest in A [1 .. k - 1].]

    if  $A[k] > \text{max}$

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

[Postcondition: max is value of lgst element of A.]

if  $\text{max} = x$

    return *true*

otherwise

    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.]

found ← false

while not empty(S)

[Loop invariant: found is true if and only if key is in aux]

test ← Pop(S)

if test = key

found ← true

Push(aux, test)

[Postcondition: (a) found is true iff key in aux.]

while not empty (aux)

[Loop invariant: aux reversed + S contains original contents of S]

Push(S, Pop(aux))

[Postcondition: (b) S in same state as at start of algorithm.]

return found

**Limits on use of postconditions**

- Note that for postcondition to be used, algorithm must **not** alter values of any parameters

- *Example:*

**Sum(a, b)**

y ← a

while b > 0

    a ← succ(a)

    b ← b - 1

// Post: y = a + b

**Misleading! b = 0 now!**



### 3. Formal proof methods and 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  $\psi$

### Method

- This verification method uses loop invariants. Technique is to work backward from loop postconditions, derive invariant, prove invariant.
- Proof must show that loop terminates
- *Example:*

$sum \leftarrow 0$

for  $count \leftarrow 1$  to  $n$

> Invariant:  $sum = \sum_{k=1}^{count-1} k$

$sum \leftarrow sum + count$

> Postcondition:  $sum = \sum_{k=1}^n k$

## Example: Factorial

### Factorial(x)

```

<x ≥ 0>
y ← 1
t ← 1
while (t ≤ x) begin
  LI: y = (t - 1)!
  y ← ty
  t ← t + 1
end
<y = x!>

```

## 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):

$$\langle \phi \rangle P_1 \langle \eta \rangle \wedge \langle \eta \rangle P_2 \langle \psi \rangle \Rightarrow \langle \phi \rangle P_1 P_2 \langle \psi \rangle$$

- 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:

$$\langle \phi \wedge b \rangle P_1 \langle \psi \rangle \wedge \langle \phi \wedge \neg b \rangle P_2 \langle \psi \rangle \Rightarrow \langle \phi \rangle \mathbf{if } b \{ P_1 \} \mathbf{else } \{ P_2 \} \langle \psi \rangle$$

- 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  $y = abs(x)$

## Example with *If*

### 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.  $\langle \phi \wedge b \rangle P \langle \phi \rangle \Rightarrow \langle \phi \rangle \text{while } b \{P\} \langle \phi \wedge \neg b \rangle$

- $\phi$  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

$\langle \phi \wedge b \rangle P \langle \phi \rangle \Rightarrow \langle \phi \rangle \text{while } b \{P\} \langle \phi \wedge \neg b \rangle$

## While example algorithm

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

### Power-of-2(x)

$y \leftarrow 1$

$i \leftarrow 0$

While  $i < x$

> *LI*:  $y = 2^i$

$y \leftarrow 2y$

$i \leftarrow i + 1$

> *Post*:  $y = 2^x$  [because  $y = 2^i \wedge x = i$ ]

## Intuition

### 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  $\langle \phi \rangle P \langle \psi \rangle$  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

```

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

```

- 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  $b$

## Terminology

assertion	partial and total correctness
convergence	partition
correctness	postcondition
Hoare triple	precondition
induction principle	quantifier
inductive proof	reactive system
insertion sort	transition system
loop invariant	

## References

- Grunberg and Peled. *Model Checking*. MIT Press, 1999.
- H. Hamburger and D. Richards. *Logic and Language Models for Computer Science*. Prentice Hall, 2002.
- M. Huth & M. Ryan. *Logic in Computer Science: Modeling and Reasoning About Systems*. Cambridge Univ., 2000.