Reasoning with invariants

Jordi Cortadella
Department of Computer Science
Invariants

• Invariants help to ...
  – Define how variables must be initialized before a loop
  – Define the necessary condition to reach the post-condition
  – Define the body of the loop
  – Detect whether a loop terminates

• It is crucial, but not always easy, to choose a good invariant.

• Recommendation:
  – Use invariant-based reasoning for all loops (possibly in an informal way)
  – Use formal invariant-based reasoning for non-trivial loops
General reasoning for loops

Initialization;

// Invariant: a proposition that holds
// * at the beginning of the loop
// * at the beginning of each iteration
// * at the end of the loop

// Invariant
while (condition) {
    // Invariant \land condition
    Body of the loop;
    // Invariant
}

// Invariant \land \neg condition
Example with invariants

• Given \( n \geq 0 \), calculate \( n! \)

• Definition of factorial:

\[
n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n
\]

(particular case: \( 0! = 1 \))

• Let’s pick an invariant:
  – At each iteration we will calculate \( f = i! \)
  – We also know that \( i \leq n \) at all iterations
Calculating n!

// Pre: n ≥ 0
// Returns n!
int factorial(int n) {
    int i = 0;
    int f = 1;
    // Invariant: f = i! and i ≤ n
    while (i != n) {
        // f = i! and i < n
        i = i + 1;
        f = f*i;
        // f = i! and i ≤ n
    }
    // f = i! and i ≤ n and i ≠ n
    // f = n!
    return f;
}
Reversing digits

• Write a function that reverses the digits of a number (representation in base 10)

• Examples:

  35276  →  67253
  19    →  91
  3     →  3
  0     →  0
Reversing digits

// Pre: n ≥ 0
// Returns n with reversed digits (base 10)

int reverse_digits(int n) {
  int r;

  r = 0;
  // Invariant (graphical): →
  while (n != 0) {
    r = 10*r + n%10;
    n = n/10;
  }

  return r;
}
Palindromic vector

• Design a function that checks whether a vector is a palindrome (the reverse of the vector is the same as the vector). For example:

9 -7 0 1 -3 4 -3 1 0 -7 9

is a palindrome.
Palindrome vector

// Returns true if A is a palindrome, and false otherwise.
bool palindrome(const vector<int>& A);

Invariant:

The fragments A[0..i-1] and A[k+1..A.size()-1] are reversed
// Returns true if A is a palindrome, // and false otherwise.

bool palindrome(const vector<int>& A) {
    int i = 0;
    int k = A.size() - 1;

    while (i < k) {
        if (A[i] != A[k]) return false;
        else {
            i = i + 1;
            k = k - 1;
        }
    }
    return true;
}
Classify elements

• We have a vector of elements \( V \) and an interval \([x, y]\) \((x \leq y)\). Classify the elements of the vector by putting those smaller than \(x\) in the left part of the vector, those larger than \(y\) in the right part and those inside the interval in the middle. The elements do not need to be ordered.

• Example: interval \([6, 9]\)
Classify elements

- **Invariant:**
  - At each iteration, we treat the element in the middle
    - If it is smaller, swap the elements in left and the middle (left →, mid →)
    - If larger, swap the elements in the middle and the right (←right)
    - If inside, do not move the element (mid →)
  - End of classification: when mid > right. Termination is guaranteed since mid and right get closer at each iteration.

- **Initially:** left = mid = 0, right = size-1
Classify elements

// Pre:  x <= y
// Post: the elements of V have been classified moving those
//       smaller than x to the left, those larger than y to the
//       right and the rest in the middle.

void classify(vector<int>& V, int x, int y) {
    int left = 0;
    int mid = 0;
    int right = V.size() - 1;

    // Invariant: see the previous slide
    while (mid <= right) {
        if (V[mid] < x) {
            // Move to the left part
            swap(V[mid], V[left]);
            left = left + 1;
            mid = mid + 1;
        } else if (V[mid] > y) {
            // Move to the right part
            swap(V[mid], V[right]);
            right = right - 1;
        } else mid = mid + 1;
    }
}
Vector fusion

• Design a function that returns the fusion of two ordered vectors. The returned vector must also be ordered. For example, C is the fusion of A and B:

A: -9 -7 0 1 3 4
B: -8 -7 1 2 2 4 5
C: -9 -8 -7 -7 0 1 1 2 2 3 4 4 5
Vector fusion

// Pre: A and B are sorted in ascending order.
// Returns the sorted fusion of A and B.
vector<int> fusion(const vector<int>& A, const vector<int>& B);

Invariant:

• C contains the fusion of A[0..i-1] and B[0..j-1]
• All the visited elements are smaller than or equal to the non-visited ones.
// Pre: A and B are sorted in ascending order.
// Returns the sorted fusion of A and B.

vector<int> fusion(const vector<int>& A, const vector<int>& B) {
    vector<int> C;
    int i = 0, j = 0;
    while (i < A.size() and j < B.size()) {
        if (A[i] <= B[j]) {
            C.push_back(A[i]);
            i = i + 1;
        } else {
            C.push_back(B[j]);
            j = j + 1;
        }
    }

    while (i < A.size()) {
        C.push_back(A[i]);
        i = i + 1;
    }

    while (j < B.size()) {
        C.push_back(B[j]);
        j = j + 1;
    }

    return C;
}
• Using invariants is a powerful methodology to derive correct and efficient iterative algorithms.

• Recommendation to find a good invariant for a loop:
  – Consider the iterative progress of the algorithm.
  – Try to describe the state of the program at the beginning of an iteration (this is the invariant!).
  – Declare the variables required to describe the invariant.
  – Derive the condition, loop body and initialization of the variables of the loop (the order is not important)