2. Algorithm Correctness

José Proença Pedro Ribeiro

Algorithms (CC4010) 2024/2025

CISTER – U.Porto, Porto, Portugal

https://fm-dcc.github.io/alg2425

Motivation

slides by Pedro Ribeiro, slides 1

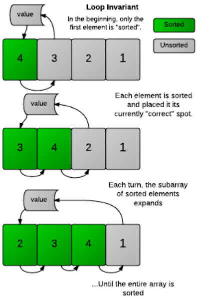
pages 1-5

Correctness and Loop Invariants

Pedro Ribeiro

DCC/FCUP

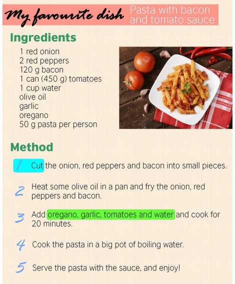
2018/2019

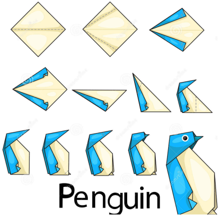


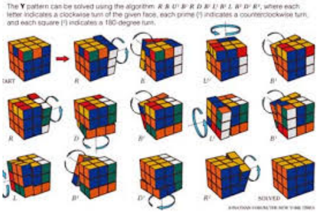
On Algorithms

What are algorithms? A set of **instructions** to solve a **problem**.

- The problem is the **motivation** for the algorithm
- The instructions need to be **executable**
- Typically, there are **different algorithms** for the same problem [how to choose?]
- **Representation**: description of the instructions that is understandable for the intended audience







On Algorithms

"Computer" Science version

- An algorithm is a **method** for solving a (computational) problem
- Algorithms are the **ideas** behind the programs and are independent from the programming language, the machine, ...
- A **problem** is characterized by the description of its **input** and **output**

A classical example:

Sorting Problem

Input: a sequence of $\langle a_1, a_2, \ldots, a_n \rangle$ of n numbers
Output: a permutation of the numbers $\langle a_1^{'}, a_2^{'}, \ldots, a_n^{'} \rangle$ such that $a_1^{'} \leq a_2^{'} \leq \ldots \leq a_n^{'}$

Example instance for the sorting problem

Input: 6 3 7 9 2 4
Output: 2 3 4 6 7 9

On Algorithms

What do we aim for?

- What **properties** do we want on an algorithm?

Correction

It has to solve correctly **all instances** of the problem

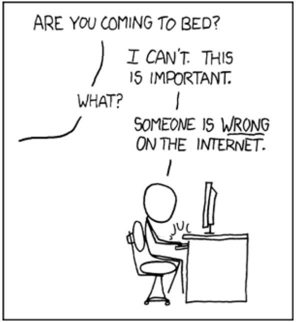
Efficiency

The performance (**time** and **memory**) has to be adequate

- This course is about **designing** correct and efficient algorithms and how to **prove** they meet the specifications

About correction

- In this lecture we will (mostly) worry about **correction**
  - Given an algorithm, it is not often obvious or trivial to know if it is **correct**, and even less so to **prove** this.
  - By learning how to reason about correctness, we also gain **insight** into what really makes an algorithm work



Specification

Ex. 2.1: What do these functions do?

```
int fa (int x, int y){
  // pre: True
  ...
  // pos: (m == x || m == y) &&
  //      (m >= x && m >= y)
  return m;
}
```

```
int fb (int x, int y){
  // pre: x >= 0 && y >= 0
  ...
  // pos: x % r == 0 && y % r == 0
  return r;
}
```

```
int fc (int x, int y){
  // pre: x > 0 && y > 0
  ...
  // pos: r % x == 0 && r % y == 0
  return r;
}
```

```
int fd (int a[], int N){
  // pre: N>0
  ...
  // pos:
  //  (forall_{0<=i<N} x<=a[i]) &&
  //  (exists_{0<=i<N} x==a[i])
  return x;
}
```

When is an algorithm correct?

Ex. 2.2: Formulate pre- and post-conditions:

`int prod (int x, int y)` – product of two integers

`int gcd (int x, int y)` – greatest common divisor of 2 positive integers

`int sum (int v[], int N)` – sum of elements in an array

`int maxPOrd (int v[], int N)` – length of the longest sorted prefix of an array

`int isSorted (int v[], int N)` – tests if an array is sorted (growing)

Hoare triples

A triple $\{P\}S\{Q\}$ is a valid Hoare triple when

if $[P\ \textit{holds}]$ and $[S\ \textit{is executed}]$ then $[Q\ \textit{holds}]$

Ex. 2.3: Find initial states that show these are not valid (and fix pre-cond.)

1. `{True} r=x+y; {r≥x}`
2. `{True} x=x+y; y=x-y; x=x-y; {x==y}`
3. `{True} x=x+y; y=x-y; x=x-y; {x≠y}`
4. `{True} if(x>y) r=x-y; else r=y-x; {r>0}`
5. `{True} while (x>0) {y=y+1; x=x-1;} {y>x}`

Finding weakest preconditions online: https://cister-labs.github.io/whilelang-scala

Partial correctness

Using rules for Hoare triples

$$\frac{P \Rightarrow Q \text{ after doing } S}{\{P\}\ S\ \{Q\}}$$

$$[\text{SEQ}]\frac{P \Rightarrow Q[x \backslash E]}{\{P\}\ x\texttt{:=}E\ \{Q\}} \qquad [\text{WHILE}]\frac{P \Rightarrow I \qquad \{I \wedge c\}\ S\ \{I\} \qquad (I \wedge \neg c) \Rightarrow Q}{\{P\}\ \texttt{while}\ c\ S\ \{Q\}}$$

1. **Initialisation:** $P \Rightarrow I$ *(P is the precondition right before the cycle)*
   Before the cycle the invariant holds.
2. **Maintenance:** $\{I \wedge c\}\ S\ \{I\}$ *(or $I \wedge c \Rightarrow I'$, where I' is the invariant after S)*
   Assuming the invariant holds before an iteration; it must be valid after it.
3. **Usefulness *(Termination)*:** $(I \wedge \neg c) \Rightarrow Q$ *(simplify $I \wedge c$ until obtain Q)*
   After the cycle the post-condition holds.

slides by Pedro Ribeiro, slides 1

pages 6-10

Loops

- We will tackle one of the most fundamental (and most used) algorithmic patterns: a **loop** (e.g. `for` or `while` instructions)

Example loop: summing integers from 1 to n

$sum = 0$
$i = 1$
while $(i \leq n)$ {
  $sum = sum + i$
  $i = i + 1$
}

- We will talk about how to prove that a **loop** is correct
- We will show how this is also useful for **designing** new algorithms

Loop Invariants

Definition of Loop Invariant

A **condition** that is necessarily true immediately before (and immediately after) each iteration of a loop

Note that this says nothing about its truth or falsity part way through an iteration.

Instructions are for computers, invariants are for humans

- The loop program statements are **"operational"**, they are **"how to do"** instructions
- Invariants are **"assertional"**, capturing **"what it means"** descriptions

Anatomy of a loop

Consider a simple loop: **while (B)** { **S** }

- **Q**: precondition (assumptions at the beginning)
- **B**: the stop condition (defining when the loop end)
- **S**: the body of the loop (a set of statements)
- **R**: postcondition (what we want to be true at the end)

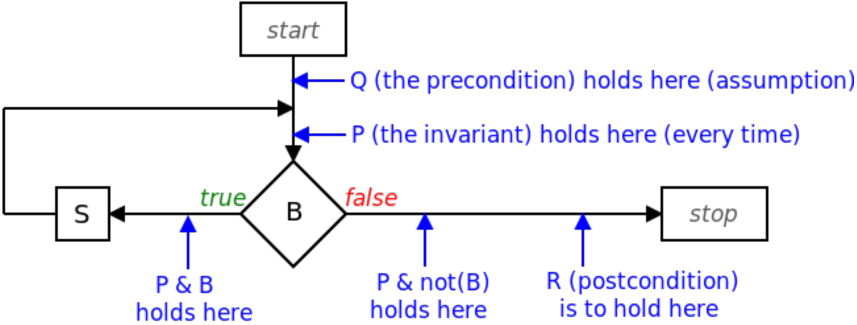
Example loop: summing integers from 1 to n

$sum = 0$
$i = 1$
while $(i \leq n)$ {
  $sum = sum + i$
  $i = i + 1$
}

- **Q**: $sum = 0$ and $i = 1$
- **B**: $i \leq N$
- **S**: $sum = sum + i$ followed by $i = i + 1$
- **R**: $sum = \sum_{i=1}^{n} i$

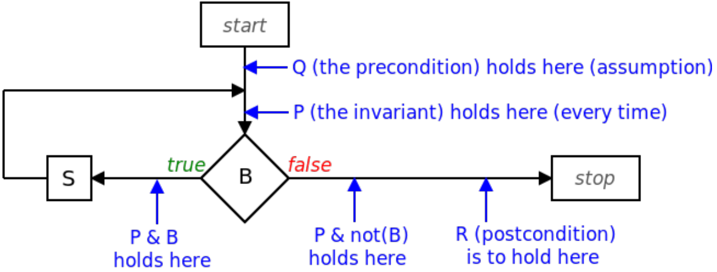
The invariant?

- **P**: an invariant (condition that holds at the start of each iteration)



- To be **useful**, the invariant P that we seek should be such that: $P \wedge not(B) \rightarrow R$
  - For the example sum loop, it could be: $sum = \sum_{i=1}^{i-1} i$

How to show that an invariant is really one?



- First, show that $Q \rightarrow P$
  (truth precondition Q guarantees truth of invariant P)
  - For the example sum loop: sum=0 which is $= \sum_{i=1}^{0} i$
- If $P \wedge B$, then after executing S, then P holds after executing S
  (the statements S of the loop guarantee that P is respected)
  - For the example sum loop: $\sum_{i=1}^{i-1} + i = \sum_{i=1}^{i}$

How to show that an invariant is really one?

Initialization

The invariant is true prior to the first iteration of the loop

$P \Rightarrow I$

in the slide before:

$Q \Rightarrow P$

Maintenance

If it is true before an iteration of the loop, it remains true before the next iteration

$I \wedge c \Rightarrow I'$

in the slide before:

$P \wedge B \Rightarrow P$ after executing S

Usefulness *(termination)*

When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

$(I \wedge \neg c) \Rightarrow Q$

in the slide before:

$P \wedge \neg B \Rightarrow R$

```
int mult1 (int x, int y){
  // pre: x>=0
  int a, b, r;
  a=x; b=y; r=0;
  while (a!=0){
    r = r+b;
    a = a-1;
  }
  // pos: r == x * y
  return r;
}
```

```
int mult2 (int x, int y){
  // pre: x>=0
  int a, b, r;
  a=x; b=y; r=0;
  while (a!=0) {
    if (a%2 == 1) r = r+b;
    a=a/2;
    b=b*2;
  // pos: r == x * y
  return r;
}
```

Ex. 2.4: Check if *Initialization* and *Maintenance* holds for these formulae

$r == a * b$	$r \geq 0$	$b == 0$
$a \geq 0$	$a == x$	$a * b == x * y$
$b \geq 0$	$a \neq x$	$a * b + r == x * y$

Exercises

```c
int mult1 (int x, int y){
  // pre: x>=0
  int a, b, r;
  a=x; b=y; r=0;
  while (a!=0){
    r = r+b;
    a = a-1;
  }
  // pos: r == x * y
  return r;
}
```

```c
int mult2 (int x, int y){
  // pre: x>=0
  int a, b, r;
  a=x; b=y; r=0;
  while (a!=0) {
    if (a%2 == 1) r = r+b;
    a=a/2;
    b=b*2;
  // pos: r == x * y
  return r;
}
```

Ex. 2.5: Find loop invariants to prove partial correctness

Some intuition – mult1(4,5)

```
1  int mult1 (int x, int y){
2    // pre: x>=0
3    int a, b, r;
4    a=x; b=y; r=0;
5    while (a>0){
6      r = r+b;
7      a = a-1;
8    }
9    // pos: r == x * y
10   return r;
11 }
```

line	x	y	a	b	r
4	4	5	4	5	0
6	4	5	4	5	5
7	4	5	3	5	5
6	4	5	3	5	10
7	4	5	2	5	10
6	4	5	2	5	15
7	4	5	1	5	15
6	4	5	1	5	20
7	4	5	0	5	20
10	4	5	0	5	20

Some intuition – mult1(4,5)

```
1  int mult1 (int x, int y){
2    // pre: x>=0
3    int a, b, r;
4    a=x; b=y; r=0;
5    while (a>0){
6      r = r+b;
7      a = a-1;
8    }
9    // pos: r == x * y
10   return r;
11 }
```

line	x	y	a	b	r
4	4	5	4	5	0
6	4	5	4	5	5
7	4	5	3	5	5
6	4	5	3	5	10
7	4	5	2	5	10
6	4	5	2	5	15
7	4	5	1	5	15
6	4	5	1	5	20
7	4	5	0	5	20
10	4	5	0	5	20

- x and y never change
- r grows proportionally as a shrinks
- guess:
  $I \triangleq a*b + r = x*y$

Some intuition – mult1(4,5)

```
1  int mult1 (int x, int y){
2    // pre: x>=0
3    int a, b, r;
4    a=x; b=y; r=0;
5    while (a>0){
6      r = r+b;
7      a = a-1;
8    }
9    // pos: r == x * y
10   return r;
11 }
```

line	x	y	a	b	r
4	4	5	4	5	0
6	4	5	4	5	5
7	4	5	3	5	5
6	4	5	3	5	10
7	4	5	2	5	10
6	4	5	2	5	15
7	4	5	1	5	15
6	4	5	1	5	20
7	4	5	0	5	20
10	4	5	0	5	20

- x and y never change
- r grows proportionally as a shrinks
- guess:
  $I \triangleq \texttt{a*b} + \texttt{r} = \texttt{x*y}$
- Need to show:

$$\texttt{x>=0} \Rightarrow I'$$

$$I \wedge \texttt{a>0} \Rightarrow I'$$

$$I \wedge \neg(\texttt{a>0}) \Rightarrow \texttt{r} = \texttt{x*y}$$

Some intuition – mult1(4,5)

```
1  int mult1 (int x, int y){
2    // pre: x>=0
3    int a, b, r;
4    a=x; b=y; r=0;
5    while (a>0){
6      r = r+b;
7      a = a-1;
8    }
9    // pos: r == x * y
10   return r;
11 }
```

line	x	y	a	b	r
4	4	5	4	5	0
6	4	5	4	5	5
7	4	5	3	5	5
6	4	5	3	5	10
7	4	5	2	5	10
6	4	5	2	5	15
7	4	5	1	5	15
6	4	5	1	5	20
7	4	5	0	5	20
10	4	5	0	5	20

- x and y never change
- r grows proportionally as a shrinks
- guess:
  $I \triangleq \texttt{a*b} + \texttt{r} = \texttt{x*y}$
- Need to show:

$$\texttt{x>=0} \Rightarrow I'$$

$$I \wedge \texttt{a>0} \Rightarrow I'$$

$$I \wedge \neg(\texttt{a>0}) \Rightarrow \texttt{r} = \texttt{x*y}$$

- (Not all works – enrich invariant!)

More exercises

```
int serie(int n){
  // pre: n>=0
  int r=0, i=1;
  // inv: ??
  while (i!=n+1) {
    r = r+i; i = i+1;
  }
  // pos: r == n * (n+1) / 2;
  return r;
}
```

```
int mod(int x, int y) {
  // pre: x>=0 && y>0
  int r = x;
  while (y <= r) {
    r = r-y;
  }
  // pos: 0 <= r < y && exists_{q}
      x == q*y + r
  return r;
}
```

Ex. 2.5: Find loop invariants

Even more exercises (@home)

```
int minInd (int v[], int N) {
  // pre: N>0
  int i = 1, r = 0;
  // inv: ???
  while (i<N) {
    if (v[i] < v[r]) r = i;
    i = i+1; }
  // pos: 0 <= r < N && forall_{0 <= k < N} v[r] <= v[k]
return r; }

int minimum (int v[], int N) {
  // pre: N>0
  int i = 1, r = v[0];
  // inv: ???
  while (i!=N) {
     if (v[i] < r) r = v[i];
     i=i+1; }
  // pos: (forall_{0 <= k < N} r <= v[k]) &&
  //       (exists_{0 <= p < N} r == v[p])
  return r;
}

int sum (int v[], int N) {
  // pre: N>0
  int i = 0, r = 0;
  // inv: ???
  while (i!=N) {
    r = r + v[i]; i=i+1;
  }
  // pos: r == sum_{0 <= k < N} v[k]
  return r;
}
```

```
int sqr1 (int x) {
  // pre: x>=0
  int a = x, b = x, r = 0;
  // inv: ??
  while (a!=0) {
    if (a%2 != 0) r = r + b;
    a=a/2; b=b*2;
  }
  // pos: r == x^2
  return r;
  }

int sqr2 (int x){
  // pre: x>=0
  int r = 0, i = 0, p = 1;
  // inv: ??
  while (i<x) {
    i = i+1; r = r+p; p = p+2;
  }
  // pos: r == x^2
  return r;
}

int ssearch (int x, int a[], int N){
  // pre: N>0 &&
  //      forall_{0 < k < N-1} a[k-1]<=a[k]
  int p = -1, i = 0;
  // inv: ??
  while (p == -1 && i < N && x >= a[i]) {
    if (a[i] == x) p = i;
    i = i+1;
  }
  // pos: (p == -1 && forall_{0 <= k < N} a[k] != x) ||
  //      ( (0 <= p < N ) && x == a[p])
  return p;
}
```

Complete correctness

Partial/Complete correctness

Given $\{P\}\ S\ \{Q\}$

Partial correctness

if [P *holds*] and [S *is executed*] then [Q *holds*]

Complete correctness

if [P *holds*] and [S *is executed*] then [Q *holds*] AND S terminates

Given $\{P\}\ S\ \{Q\}$

Partial correctness

if [P *holds*] and [S *is executed*] then [Q *holds*]

Complete correctness

if [P *holds*] and [S *is executed*] then [Q *holds*] AND S terminates

Enough to show the existence of a loop variant

Technique that measures the distance between the current state and the final state.

A loop variant V is an integer expression s.t.

- is positive in the beginning of each round ($c \land I \Rightarrow V > 0$)
- decreases in every round ($c \land I \Rightarrow V > V'$)

```
r=x;
q=0;
while (y <= r) {
  r = r-y;
  q = q+1;
}
```

- $V = r - y$ is not a good variant
- …

Technique that measures the distance between the current state and the final state.

A loop variant V is an integer expression s.t.

- is positive in the beginning of each round ($c \wedge I \Rightarrow V > 0$)
- decreases in every round ($c \wedge I \Rightarrow V > V'$)

```
r=x;
q=0;
while (y <= r) {
  r = r-y;
  q = q+1;
}
```

- $V = r - y$ is not a good variant
- $V = r - y + 1$ is a good variant

Technique that measures the distance between the current state and the final state.

A loop variant V is an integer expression s.t.

- is positive in the beginning of each round ($c \land I \Rightarrow V > 0$)
- decreases in every round ($c \land I \Rightarrow V > V'$)

```
r=x;
q=0;
while (y <= r) {
  r = r-y;
  q = q+1;
}
```

- $V = r - y$ is not a good variant
- $V = r - y + 1$ is a good variant

  $y \leq r \Rightarrow V > 0$ at each round

  $V > V'$ after each round

Exercises

```
int sum(int v[], int N) {
  int i = 0, r = 0;
  while (i!=N) {
    // variant: ???
    r = r + v[i];
    i=i+1;
  }
  return r;
}
```

Ex. 2.6: Find variant above

Ex. 2.7: Find variants of the loops in previous exercises (when searching for invariants)