2.5 Iteration

Slide 220

Iteration

Slide 221

Loops

In imperative languages common loop blocks include:

- `while`
- `for`,
- `repeat-until`

All of these can be expressed as `while` loops, so we will focus on reasoning about them.

An example:

```
1 for(i=0; i<a.length; i++){
2   res = res + a[i];
3 }
```

```
1 i=0;
2 while(i < a.length){
3   res = res + a[i];
4   i++;
5 }
```

Slide 222

Anatomy of a Loop

General structure of a `while` loop:

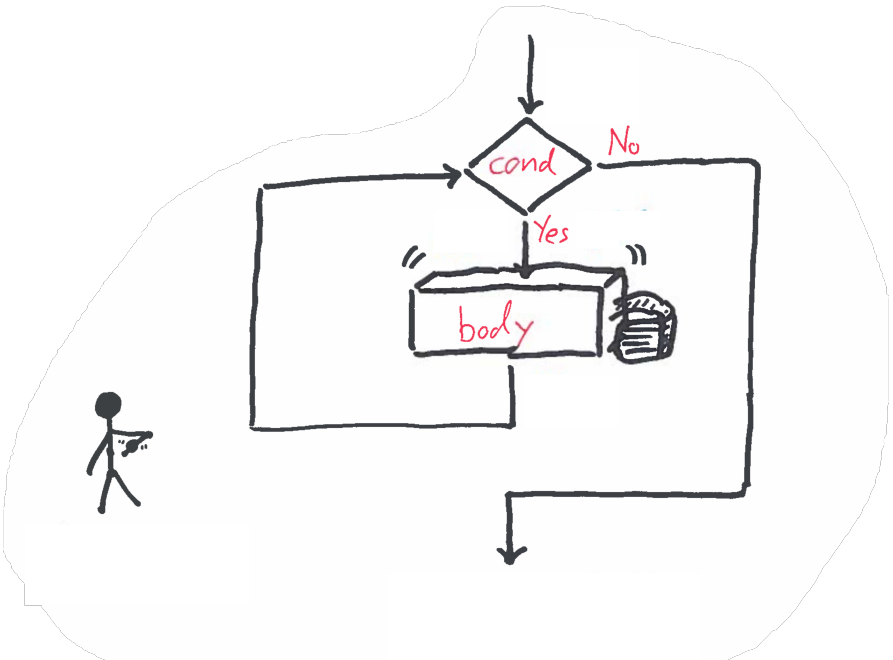
```
while (cond) { // loop condition
  code // loop body
}
// MID: M
```

A loop typically consists of:

- a loop condition (`cond`)
- a loop body (`code`)
- a mid-condition (M) that summarises the effect of the loop

Slide 223

Anatomy of a Loop



Reasoning about Loops - Example

```
1  int sum(int[] a)
2  // PRE: a ≠ null
3  // POST: a ≈ a0  ∧  r = ∑ a[0..a.length)
4  {
5     int res = 0;
6     int i = 0;
7     // MID:
8     //
9     while (i < a.length) {
10       res = res + a[i];
11       i++;
12       // MID:
13    }
14    // MID:
15    return res;
16 }
```

2.5.1 Loop Invariants and Loop Variants - Informally

Slide 225

How do we Reason about Loops?

```
while (cond) { // loop condition
  code // loop body
}
// MID: M
```

If we get to the midcondition, then we know the loop must have terminated.

This means:

- the loop has been executed m times, for some $m \in \mathbb{N}$ (possibly 0).
- `cond` must **not** hold (otherwise we would still be in the loop).

Slide 226

How do we Reason about Loops?

```
while (cond) { // loop condition
  code // loop body
}
// MID: M
```

Therefore, if we can find a property P, such that:

- $\forall n \in \mathbb{N}$.[P holds after n iterations of the loop]
- if P holds and the loop condition does not hold, then this implies the midcondition M

Then we know that:

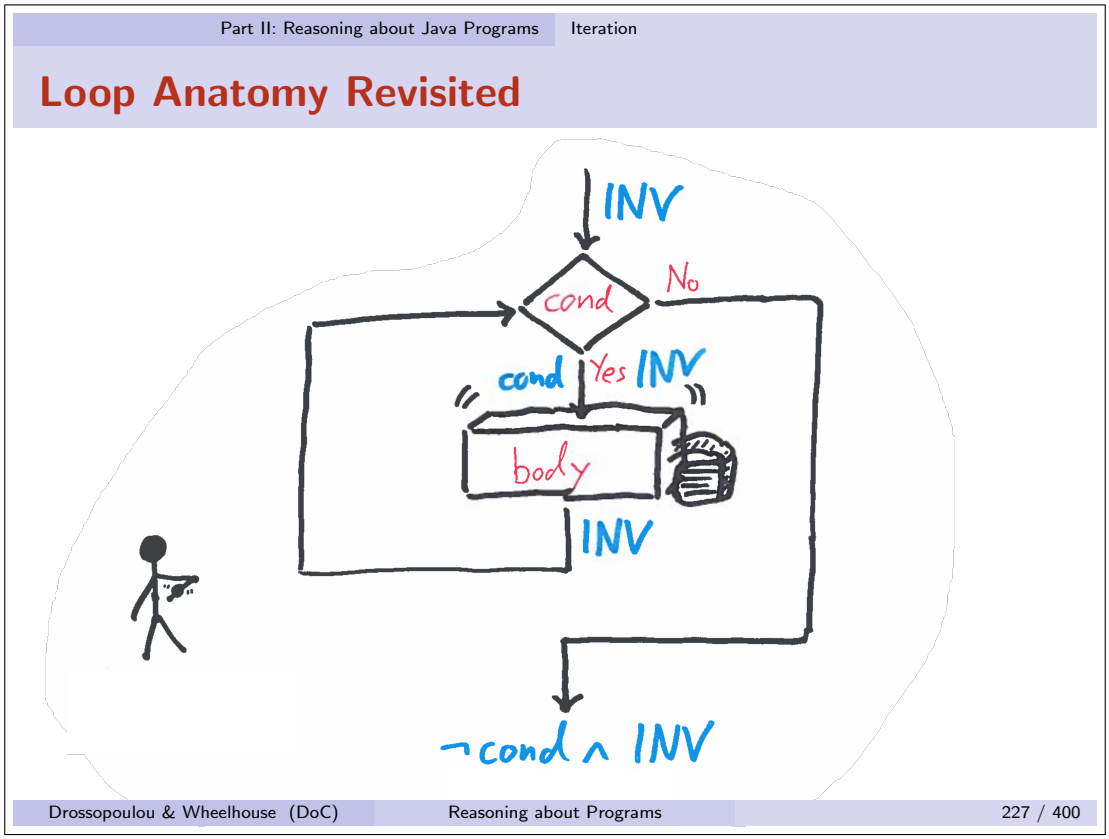
- *If* the loop terminates, then it will reach a state that satisfies M.

This property P is called the *loop invariant*.

A *loop invariant* generalises the effect of the loop after an arbitrary number of iterations of the loop. It replaces the need to track the midcondions before, after or during the loop.

However, it is usually more convenient to write a midcondition after the loop and use this to help construct our loop invariant. This is the approach that we will take for the rest of this course.

Slide 227



Slide 228

Loops and the Induction Principle

How do we show that: $\forall n \in \mathbb{N}.[\ P$ holds after n iterations of the loop $]$?

Recall the induction principle:

$$[\ P(0)\ \wedge\ \forall k \in \mathbb{N}.[\ P(k) \longrightarrow P(k+1)\ ]\ ]\ \longrightarrow\ \forall n \in \mathbb{N}.\ P(n)$$

So, we need to show:

- P holds after 0 iterations
  i.e. P holds immediately before the loop
- If P holds after k iterations, then P holds after $k+1$ iterations
  i.e. if P and `cond` hold, and loop body is executed, then P holds again

Slide 229

Developing the Invariant for our Example

```
1  int sum(int[] a)
2  // PRE: a ≠ null                                              (P)
3  // POST: a ≈ a0  ∧  r = ∑a[0..a.length)                       (Q)
4  {
5     int res = 0;
6     int i = 0;
7     // INV: a ≈ a0  ∧  a ≠ null
8     //       ∧  0 ≤ i ≤ a.length  ∧  res = ∑a[0..i)            (I)
9     while (i < a.length) {                                     (cond)
10       res = res + a[i];
11       i++;
12       // MID:  ⟷ I
13    }
14    // MID: a ≈ a0  ∧  res = ∑a[0..a.length)                   (M)
15    return res;
16 }
```


Note that the MID at the end of the loop body (line 12) is not something we would nor-

mally need to write, it is just here to illustrate when the invariant must be re-established.

Slide 230

The Invariant Graphically

to appear

Slide 231

Loop Invariant Proof Obligations (Informal)

In our example we named the predicates as follows:

- the *loop invariant*:
  $I \longleftrightarrow \texttt{a} \approx \texttt{a}_0 \wedge \texttt{a} \neq \texttt{null} \wedge 0 \leq \texttt{i} \leq \texttt{a.length} \wedge \texttt{res} = \sum \texttt{a}[0..\texttt{i})$
- the *loop condition*:
  $\texttt{cond} \longleftrightarrow \texttt{i} < \texttt{a.length}$
- the *mid-condition* after the loop:
  $M \longleftrightarrow \texttt{a} \approx \texttt{a}_0 \wedge \texttt{res} = \sum \texttt{a}[0..\texttt{a.length})$

We have the following obligations to check:

line 7: The code before the loop must establish that I holds initially.

line 11: The loop body must preserve I whenever `cond` holds.

line 13: Immediately after the loop I and $\neg$`cond` must imply M.

Slide 232

Loop Invariant Proof Obligations (Informal)

So, we need to:

1. Prove that the code preceding the loop establishes the invariant
   i.e. if P holds and we run lines 5 and 6, then I now holds
2. Prove that invariant is preserved by the loop body
   i.e. if I and `cond` hold and we run lines 10 and 11, then I still holds
3. Prove that on termination of the loop the mid-condition holds
   i.e. if I and $\neg$`cond` hold, then M holds

Slide 233

Reasoning about Loops - Establish Invariant

line 7: Prove that the code preceding the loop establishes the invariant
i.e. if P holds and we run lines 5 and 6, then I now holds

$$P[\mathtt{a} \mapsto \mathtt{a}_0] \ \wedge \ \mathtt{res} = 0 \ \wedge \ \mathtt{i} = 0 \ \wedge \ \mathtt{a} \approx \mathtt{a}_0 \longrightarrow I$$

Slide 234

Reasoning about Loops - Maintain Invariant

line 11: Prove that the invariant is preserved by the loop body
i.e. if I and `cond` hold and we run lines 10 and 11, then I still holds

$$I \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{res}' = \texttt{res} + \texttt{a[i]} \wedge \texttt{i}' = \texttt{i} + 1 \wedge \texttt{a}' \approx \texttt{a}$$
$$\longrightarrow$$
$$I[\texttt{a} \mapsto \texttt{a}', \texttt{i} \mapsto \texttt{i}', \texttt{res} \mapsto \texttt{res}']$$

Slide 235

Reasoning about Loops - Loop Exit

line 12: Prove that on termination of the loop the mid-condition holds
i.e. if I and $\neg$`cond` hold, then M holds

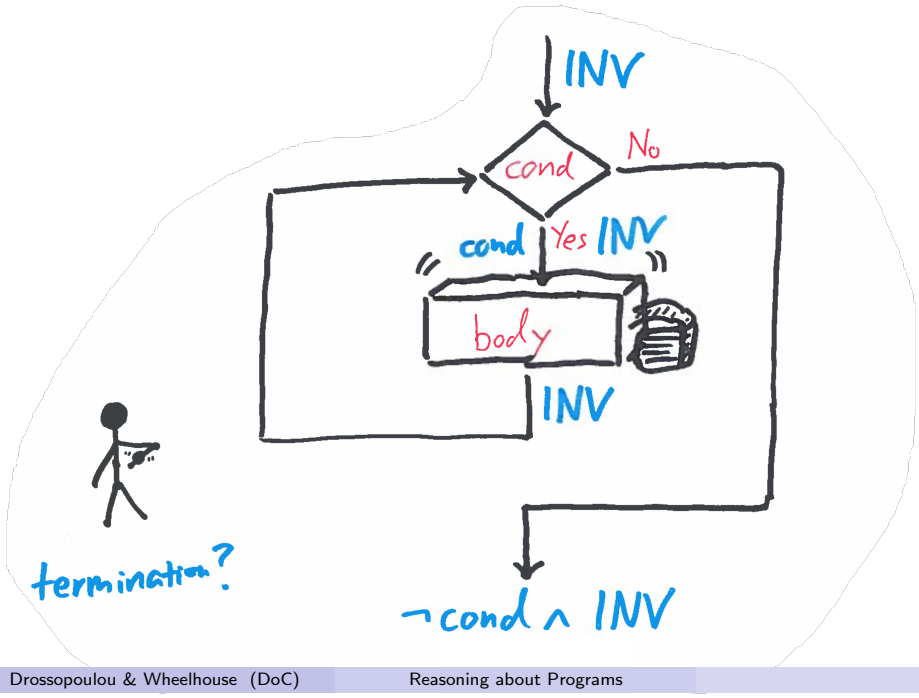
$$I \wedge \texttt{i} \geq \texttt{a.length}$$
$$\longrightarrow$$
$$M$$

No code has been executed here, so there is not need for primed variables in the code

effect or postcondition.

Slide 236

What about Termination?



Slide 237

What about Termination?

We need some way to track the progress of the loop.

We do this by finding an integer expression which:

- is larger than some value at the end of each loop iteration
- decreases in *every* loop iteration

We can then be sure that the loop will terminate.

Such an expression is called the *loop variant*.

Actually, the condition presented here is slightly stronger than necessary. In general, it is sufficient to prove that the variant decreases in the sense of some well-founded ordering.

Slide 238

Developing the Variant for our Example

```
1   int sum(int[] a)
2   // PRE: a ≠ null
3   // POST: a ≈ a0 ∧ r = ∑ a[0..a.length)
4   {
5      int res = 0;
6      int i = 0;
7      // INV: a ≈ a0 ∧ a ≠ null
8      //      ∧ 0 ≤ i ≤ a.length ∧ res = ∑ a[0..i)
9      // VAR: a.length − i
10     while (i < a.length) {
11        res = res + a[i];
12        i++;
13     }
14     // MID: a ≈ a0 ∧ res = ∑ a[0..a.length)
15     return res;
16  }
```

Slide 239

The variant graphically

to appear

Slide 240

Total vs. Partial Correctness

Partial Correctness: if the code is executed in a state satisfying its precondition, then *if* it terminates it must reach a state that satisfies its postcondition.

Total Correctness: as partial correctness above, but we also know that the code will terminate.

Why do we distinguish between partial correctness and total correctness?

Some reasons include:

- Different Proof Obligations
- Desired Property Types
- Non-Terminating Code

Firstly, depending on the code in question, it may be quite hard to prove one of partial

correctness or termination, even though the other might be easy. In fact, in some cases one of these proofs might actually be impossible (you will probably have heard of the famous Halting Problem).

Secondly, there might be some properties of your code that are more important to you than termination. For example, you might not know if your code will terminate, but still want to be able to prove that if it does terminate then it will have solved your problem.

Thirdly, there are some pieces of code that are legitimately non-terminating (web-servers, operating systems, etc...) but we may still want or need to verify their correctness.

There are probably many other good reasons for the distinction.

Slide 241

Total vs. Partial Correctness - Examples

Consider the following code snippets and specifications:

```
1  // PRE: true
2  acc = i;
3  cnt = 0;
4  while(acc != j) {
5      acc++;
6      cnt++;
7  }
8  // POST: cnt = j - i
```

```
1  // PRE: i ≤ j
2  acc = i;
3  cnt = 0;
4  while(acc != j) {
5      acc++;
6      cnt++;
7  }
8  // POST: cnt = j - i
```

Question: Are these code snippets partially and/or totally correct with respect to their specifications?

The first code snippet is partially correct, but it is not totally correct. If the input value of `i` is bigger than `j` then the loop will execute indefinitely.

By contrast, the specification of the second code snippet rules out the case mentioned above and is now both partially *and* totally correct,

Variants and Invariants - Intuitions

Intuitively:

- The *invariant* says that the program is doing "OK so far".
- Each execution of the loop's body maintains the validity of the *invariant*.
- The *variant* measures progress towards completion.

So:

- If the loop terminates, then we can use the invariant to help prove the midcondition after the loop.
- However, we need to rule out the possibility of looping forever.
  We do this by proving that the variant is bounded and decreases on *every* loop iteration.

2.5.2 Loop Invariant and Loop Variants - Examples

Slide 243

The Coffee Bean Problem

Consider a jar of **red** and **black** coffee beans and an infinite supply of **black** beans. Now perform the following algorithm:

```
1   put several beans into the jar;
2   while ( jar has more than one bean ){
3       select two beans randomly;
4       if they have the same colour,
5           then replace them by a black bean;
6       if they have different colour,
7           then replace them by a red bean;
8   }
```

Question 1. Does this process terminate?

Question 2. If it does, what colour will the last bean be?

Slide 244

The Coffee Bean Problem

Remember that we reason about loops in terms of variants and invariants, so:

```
1   put several beans in the jar;
2   // MID: jar has at least one bean
3   // INV: parity of red beans
4   // VAR: number of beans
5   while ( jar has more than one bean ){
6       select two beans randomly;
7       if they have the same colour,
8           then replace them by a black bean;
9       if they have different colour,
10          then replace them by a red bean;
11  }
12  // MID: colour of last bean in jar =
13  red if odd number of red beans initially, black otherwise
```

Slide 245

The Invariant Graphically

to appear

Slide 246

Worked Example - Array Sum Version 1

```
1   int sumArray1 (int[] a)
2   // PRE: a ≠ null
3   // POST: a ≈ a0 ∧ r = ∑ a[0..a.length)
4   {
5       int res = 0;
6       int i = 0;
7       // INV: a ≈ a0 ∧ a ≠ null
8       //      ∧ 0 ≤ i ≤ a.length ∧ res = ∑ a[0..i)
9       // VAR: a.length − i
10      while ( i < a.length ) {
11          res = res + a[i];
12          i++;
13      }
14      // MID: a ≈ a0 ∧ res = ∑ a[0..a.length)
15      return res;
16  }
```


We have already seen this version of the array sum code.

Slide 247

The Invariant Graphically

to appear

Slide 248

Worked Example - Array Sum Version 2

```
1   int sumArray2 (int[] a)
2   // PRE: a ≠ null
3   // POST: a ≈ a0  ∧  r = ∑ a[0..a.length)
4   {
5       int res = 0;
6       int i = a.length - 1;
7       // INV: a ≈ a0  ∧  −1 ≤ i ≤ a.length − 1
8       //       ∧  res = ∑ a[i+1..a.length)
9       // VAR: i
10      while ( i >= 0 ) {
11          res = res + a[i];
12          i--;
13      }
14      // MID: a ≈ a0  ∧  res = ∑ a[0..a.length)
15      return res;
16  }
```


This version of the array sum code traverses the array from the end to the beginning.

Slide 249

The Invariant Graphically

to appear

Slide 250

Worked Example - Find Element Version 1

```
1   int find1 (int[] a, int x)
2   // PRE: a ≠ null ∧ ∃j∈ℕ.[ 0 ≤ j < a.length ∧ a[j] = x ]
3   // POST: 0 ≤ r < a0.length ∧ a0[r] = x
4   //        ∧ ∀j∈ℕ.[ 0 ≤ j < r ⟶ a0[j] ≠ x ]
5   {
6       int i = 0;
7       // INV: 0 ≤ i ≤ a.length ∧ a ≈ a0 ∧ a ≠ null
8       //       ∧ ∀j∈ℕ.[ 0 ≤ j < i ⟶ a[j] ≠ x ]
9       // VAR: a.length − i
10      while ((i < a.length) && (a[i] != x)) {
11          i++;
12      }
13      // MID: 0 ≤ i < a0.length ∧ a0[i] = x
14      //       ∧ ∀j∈ℕ.[ 0 ≤ j < i ⟶ a0[j] ≠ x ]
15      return i;
16  }
```


Note in this example that due to Java's call by value semantics and since the code does

not redefine `int x`, we do not need to track the potential changes in state to this variable. In particular, this means we are safe to refer to `x` in our invariant, rather than x_0.

Of course, we could choose to explicitly track the potential change in state throughout the code, but this would cause us some extra notational overhead that we would prefer to avoid if possible.

Also be aware that the loop in the `find1` method might finish before the variant reaches zero, but this is not a problem. The variant is giving us a worst-case estimate of the number of iterations of the loop that are left to go.

Slide 251

The Invariant Graphically

to appear

Slide 252

Worked Example - Find Element Version 2

```
1   int find2 (int[] a, int x)
2   // PRE: a ≠ null
3   // POST: 0 ≤ r ≤ a0.length
4   //       ∧ (r < a0.length  ⟶  a0[r] = x)
5   //       ∧ ∀j∈ℕ.[ 0 ≤ j < r  ⟶  a0[j] ≠ x ]
6   {
7       int i = 0;
8       // INV: 0 ≤ i ≤ a.length  ∧  a ≈ a0  ∧  a ≠ null
9       //       ∧ ∀j∈ℕ.[ 0 ≤ j < i  ⟶  a[j] ≠ x ]
10      // VAR: a.length − i
11      while ((i < a.length) && (a[i] != x)) {
12          i++;
13      }
14      // MID: 0 ≤ i ≤ a0.length
15      //       ∧ (i < a0.length  ⟶  a0[i] = x)
16      //       ∧ ∀j∈ℕ.[ 0 ≤ j < i  ⟶  a0[j] ≠ x ]
17      return i;
18  }
```


Note that `find2` has exactly the same implementation as `find1`, only now we have a much more useful specification that does not rely on us already knowing that `x` is in the array.

Slide 253

The Invariant Graphically

to appear

Slide 254

Worked Example - Negate Array

```
1   void arrayNeg (int[] a)
2   // PRE: a ≠ null
3   // POST: ∀j∈ℕ.[ 0 ≤ j < a.length  ⟶  a[j] = −1 ]
4   {
5       int i = 0;
6       // INV: 0 ≤ i ≤ a.length  ∧  a ≠ null
7       //       ∧  ∀j∈ℕ.[ 0 ≤ j < i  ⟶  a[j] = −1 ]
8       //       ∧  ∀j∈ℕ.[ i ≤ j < a.length  ⟶  a[j] = a0[j] ]
9       // VAR: a.length − i
10      while (i < a.length) {
11          a[i] = -1;
12          i++;
13      }
14      // MID: ∀j∈ℕ.[ 0 ≤ j < a.length)  ⟶  a[j] = −1 ]
15  }
```


In this example we see a loop that actually modifies the input array. We now cannot

establish that $\mathtt{a} \approx \mathtt{a}_0$ throughout the loop, but we can instead track which parts of the array have been modified and which have not been modified.

Note that the given `arrayNeg` spec is actually quite loose. The code could legally delete the input array and return a new one of any length (including an empty array). However, this doesn't stop the code from correctly doing the right thing.

Slide 255

The Invariant Graphically

to appear

Worked Examples - Summary

The previous examples demonstrate:

- Different code can have the same specification.
  (e.g. `sumArray1` and `sumArray2`)
- The same code can have many different specifications.
  (e.g. `find1` and `find2`)

Invariant design tips:

- Use a diagram to represent the state of the program at the beginning of each loop iteration, then translate this diagram into logic.
- Use the postcondition as a guide to find the midcondition after the last loop in the code, taking into account any lines of code that appear after loop but before the end of the method.
- Use the midcondition after the loop and the condition of the loop to help you find the invariant.

2.5.3 Loop Invariant and Loop Variants - Formally

Slide 257

We've seen the informal arguments,
but how *exactly* do we reason formally about loops?

Just as you would dress up for an interview, presentation, concert or award...

we need to dress up our proofs for public display.

Slide 258

Some Useful Notation

Before proceeding, we formally introduce some helpful notation for describing ranges and arrays. For natural numbers i, m and n:

$$\begin{array}{rcl} i \in (m..n) & \longleftrightarrow & m < i < n \\ i \in [m..n) & \longleftrightarrow & m \leq i < n \\ i \in (m..n] & \longleftrightarrow & m < i \leq n \\ i \in [m..n] & \longleftrightarrow & m \leq i \leq n \end{array}$$

This notation has a natural lifting to arrays. For an array a, value v and natural numbers m and n:

$$v \in \mathtt{a}[m..n) \longleftrightarrow \exists i \in [m..n).[\ 0 \leq i < \mathtt{a.length} \wedge \mathtt{a}[i] = v\ ]$$

We can now more elegantly describe properties of arrays, for example:

$$\mathtt{a}[m..n) \leq x \longleftrightarrow \forall v \in \mathtt{a}[m..n).[\ v \leq x\ ]$$

Note: we usually prefer closed-open intervals, e.g. $[m..n)$ or $\mathtt{a}[m..n)$.

We refer to $\mathtt{a}(m..n)$ as an *array slice*. Some more examples of our array slice notation are:

- $\mathtt{a}(m..n] \geq x \longleftrightarrow \forall v \in \mathtt{a}(m..n].[\ v \geq x\ ]$
- $\mathtt{a}[m..n] \neq x \longleftrightarrow \forall v \in \mathtt{a}[m..n].[\ v \neq x\ ]$

We will see later that using closed-open intervals for array slices is often convenient for our reasoning.

Slide 259

Notation Reminder

Recall that we introduced the following notational shorthands earlier in the course, which we can now express more succinctly:

- $\texttt{a} \approx \texttt{b}$ means that the arrays `a` and `b` are identical,
  i.e. $\texttt{a.length} = \texttt{b.length} \ \wedge \ \forall j \in [0..\texttt{a.length}).[\ \texttt{a}[j] = \texttt{b}[j]\ ]$
- $\sum \texttt{a}[x..y)$ is the sum of the elements of array `a` from index x up to but not including index y,
  i.e. $\sum \texttt{a}[x..y) \ = \ \sum_{k=x}^{y-1} \texttt{a}[k] \ = \ \texttt{a}[x] + \texttt{a}[x+1] + ... + \texttt{a}[y-1]$

Important: if $m < n$ then $\sum_{x=n}^{m} f(x) \ = \ 0$ by definition.

Our previous definition of $\texttt{a} \approx \texttt{b}$ used a second conjunct of the form $\forall j \in \mathbb{N}.[\ 0 \leq j < \texttt{a.length} \ \longrightarrow \ \texttt{a}[j] = \texttt{b}[j]\ ]$ which is a little more long-winded than what we can manage with this new notation.

Slide 260

Reasoning about Loops

How do we prove that a loop satisfies its specification?

```
// PRE: P
while (cond) {
    body
}
// POST: Q
```

We have to find a loop invariant I and a loop variant V such that the invariant holds before, during and after the loop and the variant is bounded and decreases on every loop iteration.

$$\frac{P \longrightarrow I \qquad \{\ I\ \wedge\ \texttt{cond}\ \}\ \ \texttt{body}\ \ \{\ I\ \} \qquad I \wedge \neg\texttt{cond} \longrightarrow Q}{\{\ P\ \}\ \ \texttt{while(cond)}\{\ \texttt{body}\ \}\ \ \{\ Q\ \}}$$

The above Hoare logic rule for `while` captures that partial correctness requirements for a while-loop.

Slide 261

Reasoning about Loops

```
// INV: I
// VAR: V
while (cond) {
    body
}
// MID: M
```

Partial Correctness:

(a) The loop invariant I holds before the loop is entered.

(b) Given the condition, the loop body re-establishes the loop invariant I.

(c) Termination of the loop and the loop invariant I imply the mid-condition M immediately after loop.

Total Correctness: *(as above and additionally)*

(d) The variant V is bounded.

(e) The variant V decreases with each loop iteration.

Slide 262

Formal Proof Example - culSum

```
1  int culSum(int[] a)
2  // PRE: a ≠ null                                                    (P)
3  // POST: a.length = a0.length ∧ r = ∑a0[0..a.length)
4           ∧ ∀k ∈ [0..a.length).[ a[k] = ∑a0[0..k+1) ]               (Q)
5  {
6     int res = 0;
7     int i = 0;
8     // INV: a ≠ null ∧ a.length = a0.length ∧ 0 ≤ i ≤ a.length
9              ∧ res = ∑a0[0..i) ∧ ∀k ∈ [0..i).[ a[k] = ∑a0[0..k+1) ]
10             ∧ ∀k ∈ [i..a.length).[ a[k] = a0[k] ]                  (I)
11    // VAR: a.length − i                                            (V)
12    while (i < a.length) {
13       res = res + a[i];
14       a[i] = res;
15       i++;
16    }
17    // MID: a.length = a0.length ∧ res = ∑a0[0..a.length)
18             ∧ ∀k ∈ [0..a.length).[ a[k] = ∑a0[0..k+1) ]            (M)
19    return res;
20 }
```


This method calculates the sum of an array, but it also replaces each element of the array

with the cumulative sum of the elements up to (and including) that element from the original input array.

Slide 263

The Invariant Graphically

to appear

Note that an alternative formulation of the unmodified part of the array (using our new array-slice notation) would be:

$$\texttt{a}[\texttt{i..a.length}) \approx \texttt{a}_0[\texttt{i..a.length})$$

Slide 264

culSum - (a) Invariant holds before loop is entered

$$P[\mathtt{a} \mapsto \mathtt{a}_0] \ \wedge \ \mathtt{res} = 0 \ \wedge \ \mathtt{i} = 0 \ \wedge \ \mathtt{a} \approx \mathtt{a}_0 \longrightarrow I$$

Given:

(1)	$\mathtt{a}_0 \neq \mathtt{null}$	from PRE
(2)	$\mathtt{res} = 0$	from code line 6
(3)	$\mathtt{i} = 0$	from code line 7
(4)	$\mathtt{a} \approx \mathtt{a}_0$	implicit from code

To show:

(α)	$\mathtt{a} \neq \mathtt{null}$	INV
(β)	$\mathtt{a.length} = \mathtt{a}_0\mathtt{.length}$	INV
(γ)	$0 \leq \mathtt{i} \leq \mathtt{a.length}$	INV
(δ)	$\mathtt{res} = \sum \mathtt{a}_0[0..\mathtt{i})$	INV
(ϵ)	$\forall k \in [0..\mathtt{i}).[\ \mathtt{a}[k] = \sum \mathtt{a}_0[0..k+1)\ ]$	INV
(ζ)	$\forall k \in [\mathtt{i}..\mathtt{a.length}).[\ \mathtt{a}[k] = \mathtt{a}_0[k]\ ]$	INV

Note that we could, instead, choose to prime all of the program variables on the left-hand-side of the code effect givens and in the invariant I, rather than substituting $\mathtt{a}_0$ in the precondition. However, given that neither of the variables `res` or `i` exists before we run the initialisation code, it is a little lighter on notation (and still correct) to do as above.

Slide 265

culSum - (a) Invariant holds before loop is entered

Proof:

α follows from (4) and (1)

β follows from (4) and (1)

(5) $0 \leq 0 \leq \texttt{a.length}$ from (4) and (1)
γ follows from (5) and (3)

(6) $\sum \texttt{a}_0[0..0) = 0$ from def. of $\sum$
(7) $\sum \texttt{a}_0[0..\texttt{i}) = 0$ from (6) and (3)
δ follows from (7) and (2)

(8) $\forall k \in [0..0).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$ from empty range
ϵ follows from (8) and (3)

(9) $\forall k \in [0..\texttt{a.length}).[\ \texttt{a}[k] = \texttt{a}_0[k]\ ]$ from (4)
ζ follows from (9) and (3)

Slide 266

culSum - (b) Loop body re-establishes invariant

$$I \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{res}' = \texttt{res} + \texttt{a[i]} \wedge \texttt{a}'[\texttt{i}] = \texttt{res}' \wedge \texttt{i}' = \texttt{i} + 1 \wedge \forall k \in [0..\texttt{a.length}) \backslash \{\texttt{i}\}.[\ \texttt{a}'[k] = \texttt{a}[k]\ ] \longrightarrow I[\texttt{a} \mapsto \texttt{a}', \texttt{i} \mapsto \texttt{i}', \texttt{res} \mapsto \texttt{res}']$$

Given:

(1)	$\texttt{a} \neq \texttt{null}$	from INV
(2)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	from INV
(3)	$0 \leq \texttt{i} \leq \texttt{a.length}$	from INV
(4)	$\texttt{res} = \sum \texttt{a}_0[0..\texttt{i})$	from INV
(5)	$\forall k \in [0..\texttt{i}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from INV
(6)	$\forall k \in [\texttt{i}..\texttt{a.length}).[\ \texttt{a}[k] = \texttt{a}_0[k]\ ]$	from INV
(7)	$\texttt{i} < \texttt{a.length}$	cond
(8)	$\texttt{res}' = \texttt{res} + \texttt{a[i]}$	from code line 13
(9)	$\texttt{a}'[\texttt{i}] = \texttt{res}'$	from code line 14
(10)	$\texttt{i}' = \texttt{i} + 1$	from code line 15
(11)	$\forall k \in [0..\texttt{a.length}) \backslash \{\texttt{i}\}.[\ \texttt{a}'[k] = \texttt{a}[k]\ ]$	implicit from code
(12)	$\texttt{a}'\texttt{.length} = \texttt{a.length}$	implicit from code

Slide 267

culSum - (b) Loop body re-establishes invariant

$$I \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{res}' = \texttt{res} + \texttt{a[i]} \wedge \texttt{a}'\texttt{[i]} = \texttt{res}' \wedge \texttt{i}' = \texttt{i} + 1 \wedge \forall k \in [0..\texttt{a.length}) \backslash \{\texttt{i}\}.[\ \texttt{a}'[k] = \texttt{a}[k]\ ] \longrightarrow I[\texttt{a} \mapsto \texttt{a}', \texttt{i} \mapsto \texttt{i}', \texttt{res} \mapsto \texttt{res}']$$

To show:

(α)	$\texttt{a}' \neq \texttt{null}$	INV
(β)	$\texttt{a}'\texttt{.length} = \texttt{a}_0\texttt{.length}$	INV
(γ)	$0 \leq \texttt{i}' \leq \texttt{a}'\texttt{.length}$	INV
(δ)	$\texttt{res}' = \sum \texttt{a}_0[0..\texttt{i}')$	INV
(ϵ)	$\forall k \in [0..\texttt{i}').[\ \texttt{a}'[k] = \sum \texttt{a}_0[0..k+1)\ ]$	INV
(ζ)	$\forall k \in [\texttt{i}'..\texttt{a}'\texttt{.length}).[\ \texttt{a}'[k] = \texttt{a}_0[k]\ ]$	INV

Slide 268

culSum - (b) Loop body re-establishes invariant

Proof:

α follows from (1), (9) and (11)

β follows from (12) and (2)

(13)	$0 \leq \texttt{i} < \texttt{a.length}$	from (3) and (7)
(14)	$0 \leq \texttt{i} + 1 \leq \texttt{a.length}$	from (13)

γ follows from (14), (10) and (12)

(15)	$\texttt{res}' = \sum \texttt{a}_0[0..\texttt{i}) + \texttt{a[i]}$	from (8) and (4)
(16)	$\texttt{res}' = \sum \texttt{a}_0[0..\texttt{i}) + \texttt{a}_0\texttt{[i]}$	from (15) and (6)
(17)	$\texttt{res}' = \sum \texttt{a}_0[0..\texttt{i}+1)$	from (16) and def. of $\sum$

δ follows from (17) and (10)

(18)	$\forall k \in [0..\texttt{i}).[\ \texttt{a}'[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from (5) and (11)
(19)	$\texttt{a}'\texttt{[i]} = \sum \texttt{a}_0[0..\texttt{i}+1)$	from (9) and (17)
(20)	$\forall k \in [0..\texttt{i}+1).[\ \texttt{a}'[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from (18) and (19)

ϵ follows from (20) and (10)

(21)	$\forall k \in [\texttt{i}+1..\texttt{a.length}).[\ \texttt{a}'[k] = \texttt{a}_0[k]\ ]$	from (6) and (11)

ζ follows from (21), (10) and (12)

Slide 269

culSum - (c) Midcondition holds straight after loop

$$I \wedge \texttt{i} \geq \texttt{a.length} \longrightarrow M$$

Given:

(1)	$\texttt{a} \neq \texttt{null}$	from INV
(2)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	from INV
(3)	$0 \leq \texttt{i} \leq \texttt{a.length}$	from INV
(4)	$\texttt{res} = \sum \texttt{a}_0[0..\texttt{i})$	from INV
(5)	$\forall k \in [0..\texttt{i}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from INV
(6)	$\forall k \in [\texttt{i}..\texttt{a.length}).[\ \texttt{a}[k] = \texttt{a}_0[k]\ ]$	from INV
(7)	$\texttt{i} \geq \texttt{a.length}$	¬cond

To show:

(α)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	MID
(β)	$\texttt{res} = \sum \texttt{a}_0[0..\texttt{a.length})$	MID
(γ)	$\forall k \in [0..\texttt{a.length}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	MID

Slide 270

culSum - (c) Midcondition holds straight after loop

Proof:

α follows directly from (2)

(8) $\texttt{i} = \texttt{a.length}$ from (3) and (7)

β follows from (4) and (8)

γ follows from (5) and (8)

Notice that here we did **not** need to refer to the code at all

It is worth briefly remarking that substituting (8) into (6) leads to an empty range. So

we no longer have any assertion about any unmodified part of the array. However, this property is not needed to establish our desired postcondition.

Slide 271

culSum - (d)+(e) Show that the loop terminates

$$I \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{res}' = \texttt{res} + \texttt{a[i]} \wedge \texttt{a}'\texttt{[i]} = \texttt{res}' \wedge \texttt{i}' = \texttt{i} + 1 \wedge \forall k \in [0..\texttt{a.length}) \backslash \{\texttt{i}\}.[\ \texttt{a}'[k] = \texttt{a}[k]\ ] \longrightarrow V \text{ bound} \wedge V[\texttt{a} \mapsto \texttt{a}', \texttt{i} \mapsto \texttt{i}', \texttt{res} \mapsto \texttt{res}'] < V$$

Given:

(1)	$\texttt{a} \neq \texttt{null}$	from INV
(2)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	from INV
(3)	$0 \leq \texttt{i} \leq \texttt{a.length}$	from INV
(4)	$\texttt{res} = \sum \texttt{a}_0[0..\texttt{i})$	from INV
(5)	$\forall k \in [0..\texttt{i}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from INV
(6)	$\forall k \in [\texttt{i}..\texttt{a.length}).[\ \texttt{a}[k] = \texttt{a}_0[k]\ ]$	from INV
(7)	$\texttt{i} < \texttt{a.length}$	cond
(8)	$\texttt{res}' = \texttt{res} + \texttt{a[i]}$	from code line 13
(9)	$\texttt{a}'\texttt{[i]} = \texttt{res}'$	from code line 14
(10)	$\texttt{i}' = \texttt{i} + 1$	from code line 15
(11)	$\forall k \in [0..\texttt{a.length}) \backslash \{\texttt{i}\}.[\ \texttt{a}'[k] = \texttt{a}[k]\ ]$	implicit from code
(12)	$\texttt{a}'\texttt{.length} = \texttt{a.length}$	implicit from code

Note that the givens above are the same as for proof (b) since we are again verifying code that runs within the body of the loop.

Slide 272

culSum - (d)+(e) Show that the loop terminates

$$I \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{res}' = \texttt{res} + \texttt{a[i]} \wedge \texttt{a}'\texttt{[i]} = \texttt{res}' \wedge \texttt{i}' = \texttt{i} + 1 \wedge \forall k \in [0..\texttt{a.length}) \backslash \{\texttt{i}\}.[\ \texttt{a}'[k] = \texttt{a}[k]\ ] \longrightarrow V \text{ bound} \wedge V[\texttt{a} \mapsto \texttt{a}', \texttt{i} \mapsto \texttt{i}', \texttt{res} \mapsto \texttt{res}'] < V$$

To show:

(α) $\texttt{a.length} - \texttt{i} \geq 0$ — VAR bounded

(β) $\texttt{a}'\texttt{.length} - \texttt{i}' < \texttt{a.length} - \texttt{i}$ — VAR decreases

Slide 273

CulSum - (d)+(e) Show that the loop terminates

Proof:

(13) $\texttt{i} \leq \texttt{a.length}$ — from (3)

(14) $0 \leq \texttt{a.length} - \texttt{i}$ — from (3) ($-\texttt{i}$ from both sides)

α follows from (14) by rearrangement

(15) $\texttt{i}' > \texttt{i}$ — from (10)

(16) $-\texttt{i}' < -\texttt{i}$ — from (15) (mult both sides by -1)

(17) $\texttt{a.length} - \texttt{i}' < \texttt{a.length} - \texttt{i}$ from (16) (add a.length to both sides)

β follows from (17) and (12)

Slide 274

Are we nearly there yet...?

Have we now shown that the `sum` method satisfies its specification?

- We still need to show that the code after the loop establishes the postcondition.

Have we shown that the method will always terminate?

- Yes! We know that the loop terminates and straight-line code always terminates.

Is there anything else that could potentially go wrong?

- There could be an illegal array access on lines 13 or 14.
- There could be an integer overflow on lines 13 or 15.

Slide 275

culSum - (f) Postcondition established

$$M \ \wedge \ \mathbf{r} = \texttt{res} \longrightarrow Q$$

Given:

(1)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	from MID
(2)	$\texttt{res} = \sum \texttt{a}_0[0..\texttt{a.length})$	from MID
(3)	$\forall k \in [0..\texttt{a.length}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from MID
(4)	$\mathbf{r} = \texttt{res}$	from code line 19

To show:

(α)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	POST
(β)	$\mathbf{r} = \sum \texttt{a}_0[0..\texttt{a.length})$	POST
(γ)	$\forall k \in [0..\texttt{a.length}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	POST

Slide 276

culSum - (f) Postcondition established

Proof:

α follows directly from (1)

β follows from (4) and (2)

γ follows directly from (3)

Slide 277

culSum - (g) Array accesses are legal

$$I \wedge \texttt{i} < \texttt{a.length} \longrightarrow 0 \leq \texttt{i} < \texttt{a.length}$$

Given:

(1)	$\texttt{a} \neq \texttt{null}$	from INV
(2)	$\texttt{a.length} = \texttt{a}_0\texttt{.length}$	from INV
(3)	$0 \leq \texttt{i} \leq \texttt{a.length}$	from INV
(4)	$\texttt{res} = \sum \texttt{a}_0[0..\texttt{i})$	from INV
(5)	$\forall k \in [0..\texttt{i}).[\ \texttt{a}[k] = \sum \texttt{a}_0[0..k+1)\ ]$	from INV
(6)	$\forall k \in [\texttt{i}..\texttt{a.length}).[\ \texttt{a}[k] = \texttt{a}_0[k]\ ]$	from INV
(7)	$\texttt{i} < \texttt{a.length}$	cond

To show:

(α)	$0 \leq \texttt{i} < \texttt{a.length}$	legal array acces on line 9

Proof:

α follows from (3) and (7)

Slide 278

Culsum - (h) Integer overflows

Remember that for the scope of this course we assume a "perfect" machine that can represent *any* integer.

In this course we choose not to be constrained by the machine's architecture. We are effectively reasoning independently of the platform that the code is running on.

Of course, in practice, this *is* something that you would need to be worried about. After all, this kind of problem is what caused the Ariane 5 Disaster.

[Extra] Loops with Breaks and Continues

The Java keywords `break` and `continue` can be very useful constructs for manipulating the control flow of a loop.

`break` immediately terminates the execution of the loop, returning the control flow to the code outside the loop.

`continue` immediately stops the execution of the current iteration of the loop, returning the control flow to the condition check at the start of a new iteration.

The reasoning for each of these constructs ties in rather neatly with what we have already considered.

On the execution of the `break` keyword we will leave the loop, so we need to establish that the midcondion M after the loop now holds at this point in the program. Note that when we `break` out of the loop, we do not need to re-establish that the loop invariant I still holds.

On the execution of the `continue` keyword we will start a new loop iteration, so we need to establish that the loop invariant now holds at this point in the program.

Slide 279

Comparison with Recursion

There is a a lot of similarity in how we reason about loops and recursive methods:

- Loop invariants extract the essence of what a loop is promising to achieve.
- Method specifications extract the essence of what a method is promising to achieve.
- There is an underlying inductive thinking in the notion of loop invariants.
- There is an underlying inductive thinking in the specification of recursive functions.

Compare Iteration and Recursion

Compare the specification and verification of our iterative `culSum` with our recursive `sum`:

- Loop invariant from `culSum` corresponds to precondition of `sumAux`.
- Post-condition of `culSum` corresponds to post-condition of `sumAux`.
- **(a)** Proving that the invariant holds immediately before entering the loop, corresponds to proving that the precondition of `sumAux` holds before its call in `sum`.
- **(b)** Proving that when the condition holds the loop body preserves the invariant, corresponds to proving that the precondition of `sumAux` holds before the recursive call and that its return can be used to prove the postcondition of the caller.
- **(c)** Proving that the invariant and negation of the loop condition imply the midcondition after the loop, corresponds to proving that when exiting the body of `sumAux` we satisfy the postcondition of `sum`.

Summary of Part II

Slide 281

Phew, we made it!

So, through:

- specifications
- preconditions
- postconditions
- midconditions
- loop invariants
- loop variants
- and *careful book-keeping*

we can analyse and (more importantly) *understand* any program!

Slide 282

Part II – Conclusion

- Predicates describe the program state at particular program points.
- Pre-conditions, Post-conditions and Mid-conditions describe the state immediately before, after or at some intermediate point in the code.
- These terms are relative, that is, the post-condition of some code serves as the pre-condition of the subsequent code.
- A loop's pre-condition should imply its invariant; the loop body and condition should preserve its invariant; the loop invariant and negation of condition should imply the loop's post-condition.
- The possible values for a loop's variant should be bounded, and should decrease with every loop iteration.