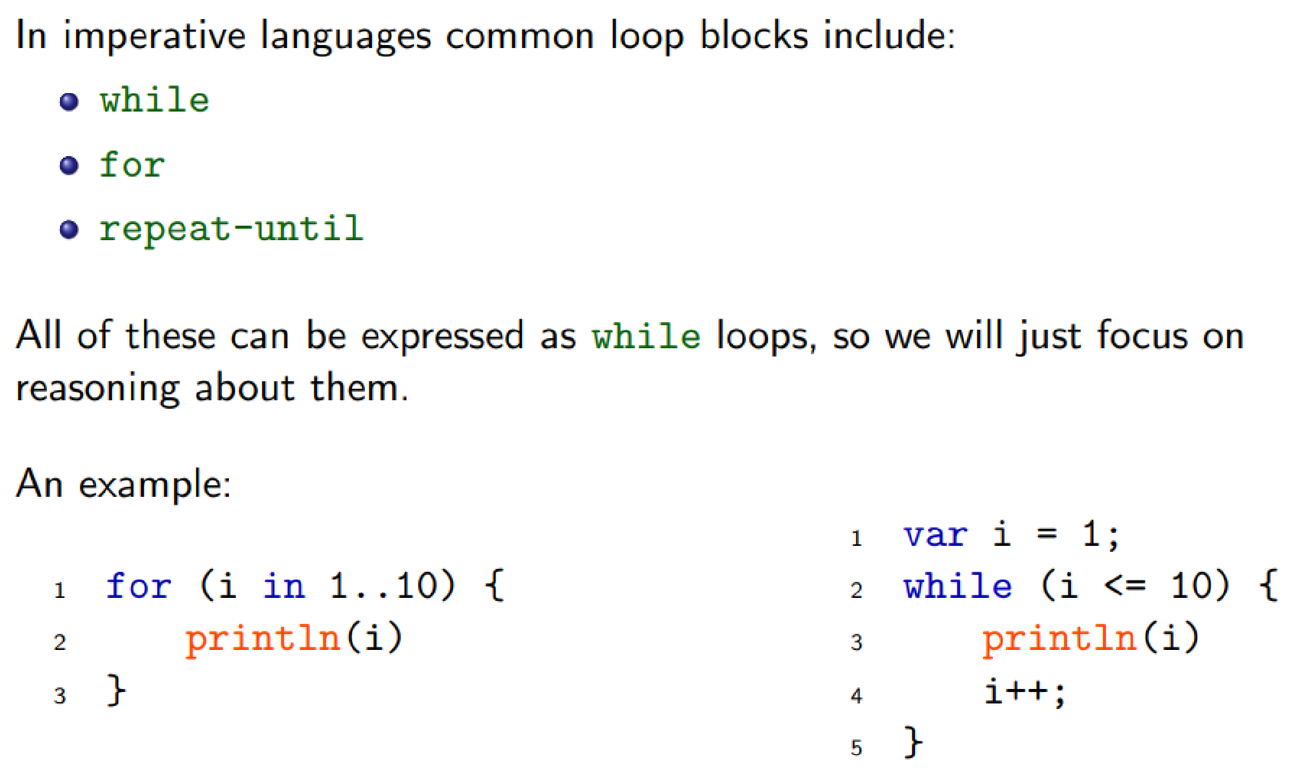
Iteration

Loops



In imperative languages common loop blocks include:

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

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

An example:

```
for (i in 1..10) {
    println(i)
}
```

```
var i = 1;
while (i <= 10) {
    println(i)
    i++;
}
```

We can create inference rules for all forms of loops once we have a rule for the `While` loop, so there is no need to treat each such loop block separately.

Indeed, it is desirable to keep the core of our Reasoning system as small as possible, so that is is easier to prove properties about the system itself (such as soundness and completeness).

Anatomy of a Loop

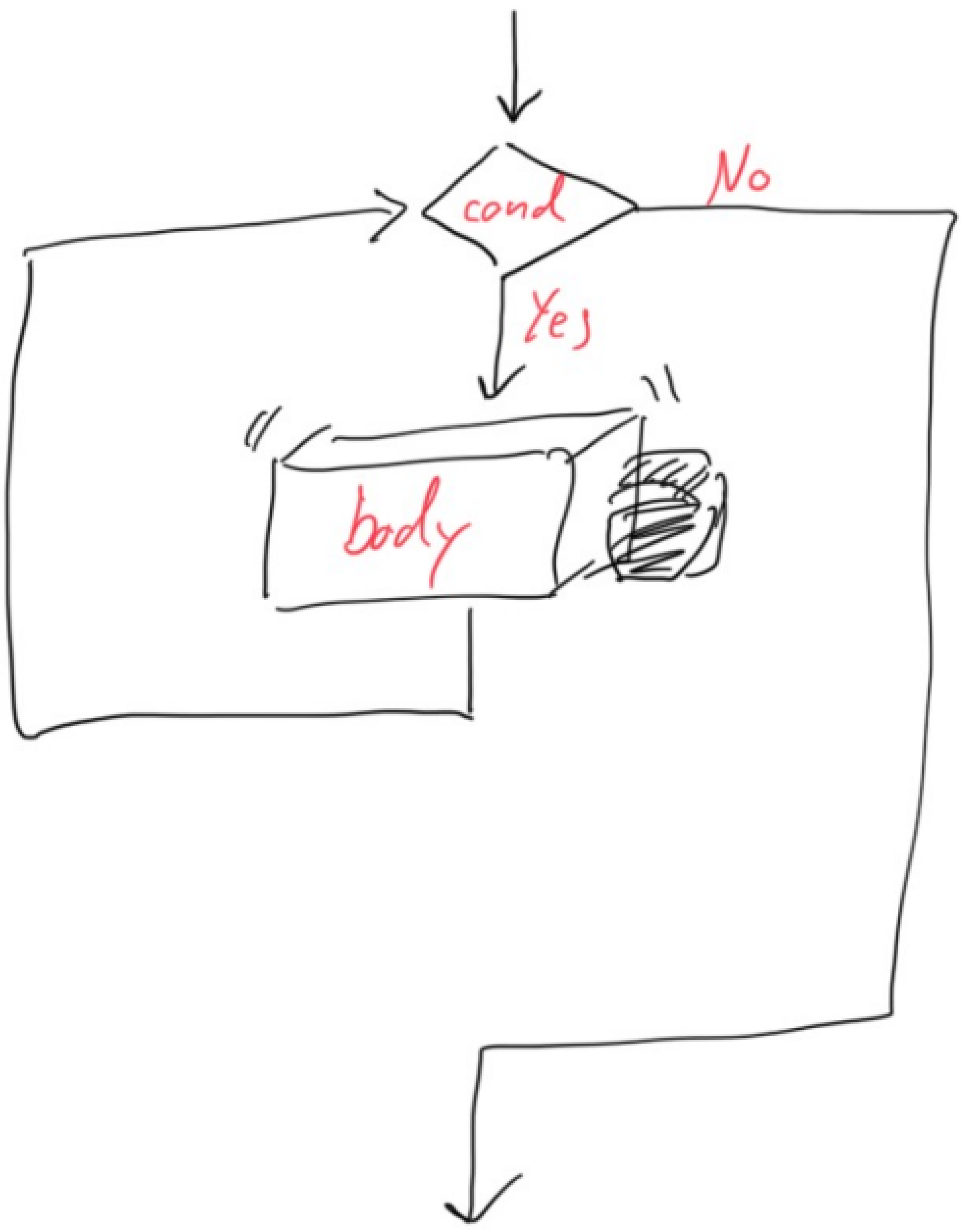
General structure of a `while` loop:

```
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

Anatomy of a Loop



Reasoning about Loops - Example

```
1   fun sum(a: Array<Int>): Int
2   // PRE: true
3   // POST: $a[..) \approx a[..)_{pre} \;\land\; r = \sum a[..)$
4   {
5       var res = 0
6       var i = 0
7       // MID:
8       //
9       while (i < a.size) {
10          res += a[i]
11          i++
12          // MID:
13      }
14      // MID:
15      return res
16  }
```

Even without knowing how to reason about the loop in this code, we can still construct mid-conditions for lines 7 and 14

- Line 7: Considering the implicit and explicit effects of code lines 5 and 6 we get:
  - $a[..) \approx a[..)_{pre} \cap res = 0 \cap i = 0$
- Line 14: Working backwards from the post-condition of sum we can deduce that before the return we need the following to hold:
  - $a[..) \approx a[..)_{pre} \cap res = \sum a[..)$

Now we must work out what mid-condition we would write on line 12. We clearly need $a[..) \approx a[..)_{pre}$ throughout the loop body, otherwise we won't be able to establish this before the return, but how do we capture the possible states of `res` and `i` when they change on each loop iteration?

How do we Reason about Loops?

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

Observe: If we get to the mid-condition, 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 inside the loop)

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 mid-condition M

Then we know that:

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

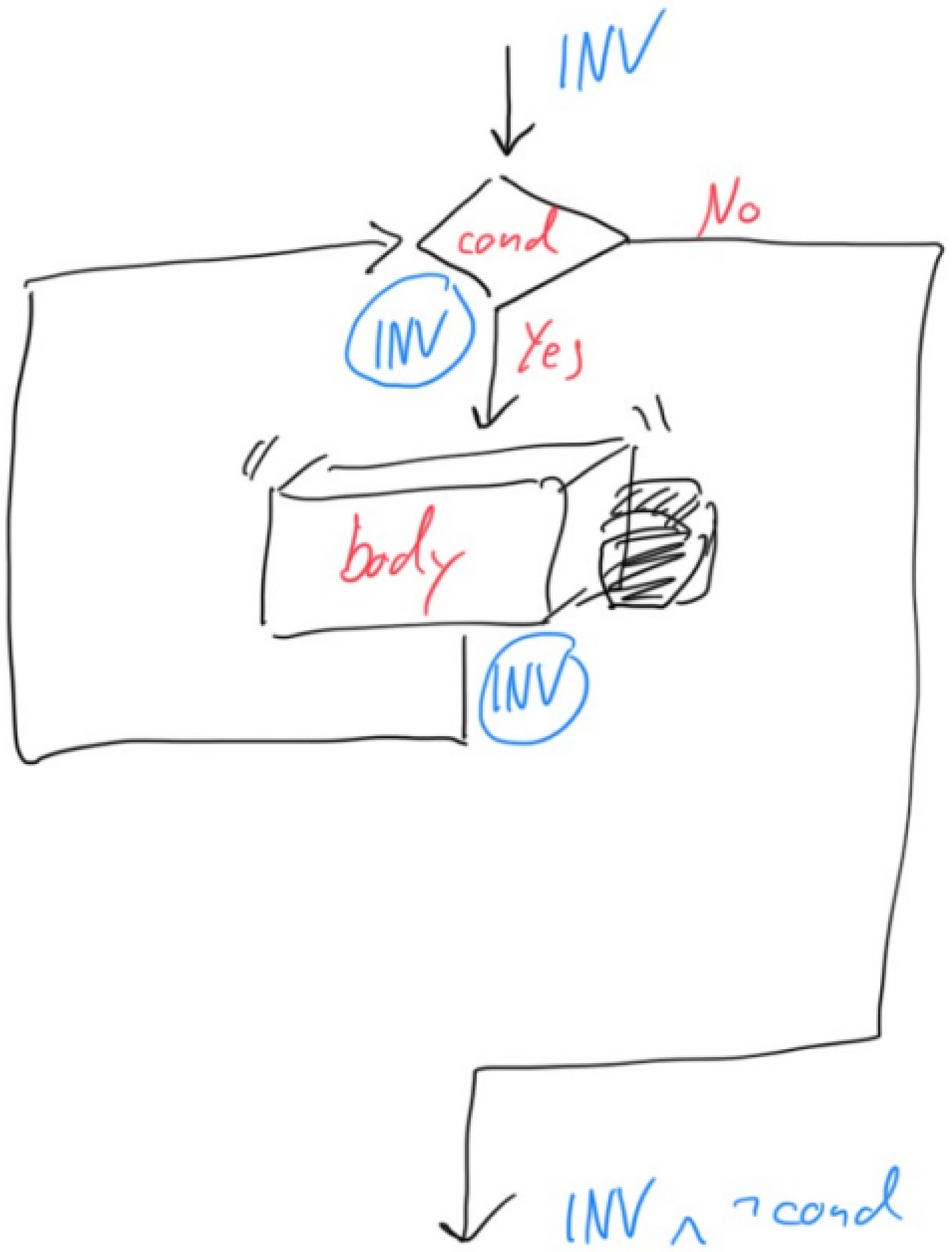
This property P is called the *loop invariant*

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 mid-conditions before, after or during the loop.

However, it is usually more convenient to write a mid-condition 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 module.**

Loop Anatomy Revisited



Loops and the Induction Principle

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

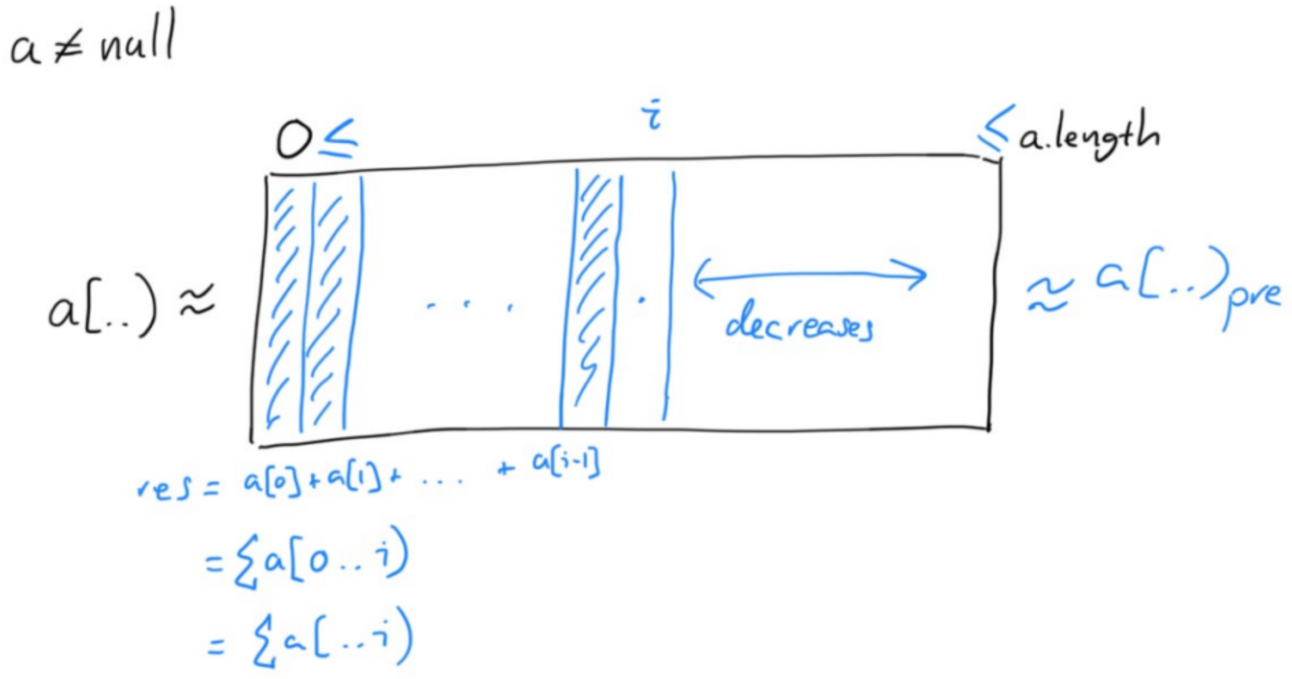
Recall the induction principle:

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

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 `code` is run, then P holds again

The Invariant Graphically



Developing the Invariant for our Example

```
1   fun sum(a: Array<Int>): Int
2   // PRE: true                                                  (P)
3   // POST: $a[..) \approx a[..)_{pre} \;\wedge\; r = \sum a[..)$          (Q)
4   {
5      var res = 0
6      var i = 0
7      // INV: $a[..) \approx a[..)_{pre}$
8      //        $\wedge\; 0 \leq i \leq a.size \;\wedge\; res = \sum a[..i)$   (I)
9      while (i < a.size) {                                     (cond)
10        res += a[i]
11        i++
12        // MID:  $\longrightarrow I$
13     }
14     // MID: $a[..) \approx a[..)_{pre} \;\wedge\; res = \sum a[..)$          (M)
15     return res
16  }
```

Note that the MID at the end of the loop body (line 12) is not something we would normally need to write, it is just here to illustrate when the invariant must be re-established.

Loop Invariant Proof Obligations (Informal)

In our example we named the predicates as follows:

- the *loop invariant*:

  $$I \triangleq a[..) \approx a[..)_{pre} \;\wedge\; 0 \leq \mathtt{i} \leq \mathtt{a.size} \;\wedge\; \mathtt{res} = \sum a[..\mathtt{i})$$

- the *loop condition*:

  $$\mathtt{cond} \triangleq \mathtt{i} < \mathtt{a.size}$$

- the *mid-condition* after the loop:

  $$M \triangleq a[..) \approx a[..)_{pre} \;\wedge\; \mathtt{res} = \sum a[..)$$

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.

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

To be able to reason about the general behaviour of the loop on an arbitrary iteration, we need the loop invariant to hold at the start and end of every loop iteration. This includes before the first time we are about to run the loop body and also immediately after the final time we have run the loop body.

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

$$\boxed{\begin{array}{c} P \land \texttt{var res = 0} \land \texttt{var i = 0} \land \texttt{a}[..) \approx \texttt{a}[..)_{pre} \\ \longrightarrow \\ I \end{array}}$$

For this proof obligation we get to assume the pre-condition P, the explicit effects of the code (declaration and initial assignment for the variables `res` and `i`) and implicit effects of the code (that the array `a` is not modified before we enter the loop).

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

$$\boxed{\begin{array}{l} I[\mathtt{i} \mapsto \mathtt{i}_{old}, \mathtt{res} \mapsto \mathtt{res}_{old}] \\ \wedge\ (\mathtt{i} < \mathtt{a.size})[\mathtt{i} \mapsto \mathtt{i}_{old}, \mathtt{res} \mapsto \mathtt{res}_{old}] \\ \wedge\ \mathtt{res\ +=\ a[i]}\ \wedge\ \mathtt{i++} \\ \qquad\qquad \longrightarrow \\ \qquad\qquad I \end{array}}$$

For this proof obligation we need to carefully track the potential change in state caused by the code in the loop body. Thus, in our given invariant I and loop condition we need to substitute all of the program variables modified by the code in the loop body with their $\_{\mathrm{old}}$ values.

As before, we also need to track the explicit effects of the code.

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

$$\boxed{\begin{array}{c} I \wedge \neg(\mathtt{i} < \mathtt{a.size}) \\ \longrightarrow \\ M \end{array}}$$

This final proof obligation is a bit simpler, as no code is actually executed here. This means that there is no need to distinguish between the current and old values of our program state. We are simply checking that one description of our program state can be re-expressed in another way.

What about Termination?

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

We do this by finding an integer expression which:

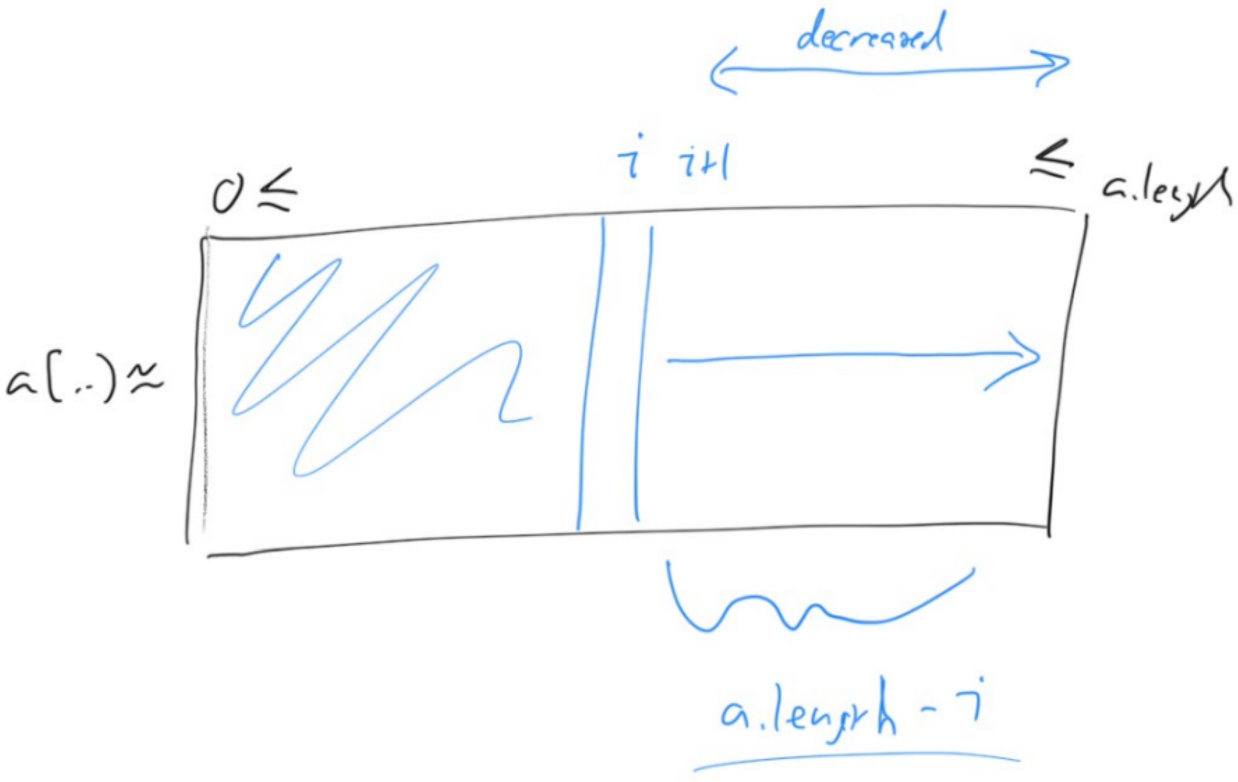
- is larger than some value at the end of each loop iteration
- decreases in *every* loop iteration

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. However, for the sake of simplicity, we will favour loop variants that decrease.

The Variant graphically



Developing the Variant for our Example

```
fun sum(a: Array<Int>): Int
// PRE: true                                        (P)
// POST: a[..] ≈ a[..]_pre  ∧  r = ∑ a[..]            (Q)
{
   var res = 0
   var i = 0
   // INV: a[..] ≈ a[..]_pre
   //         ∧  0 ≤ i ≤ a.size  ∧  res = ∑ a[..i)   (I)
   // VAR: a.size − i                               (V)
   while (i < a.size) {                             (cond)
      res += a[i]
      i++
   }
   // MID: a[..] ≈ a[..]_pre  ∧  res = ∑ a[..]        (M)
   return res
}
```

Total vs. Partial Correctness - Examples

Consider the following code snippets and specifications:

```
// PRE: true
var acc = i
var cnt = 0
while(acc != j) {
    acc++
    cnt++
}
// POST: cnt = j − i
```

```
// PRE: i ≤ j
var acc = i
var cnt = 0
while(acc != j) {
    acc++
    cnt++
}
// POST: cnt = j − i
```

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 mid-condition 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.

The Coffee Bean Problem

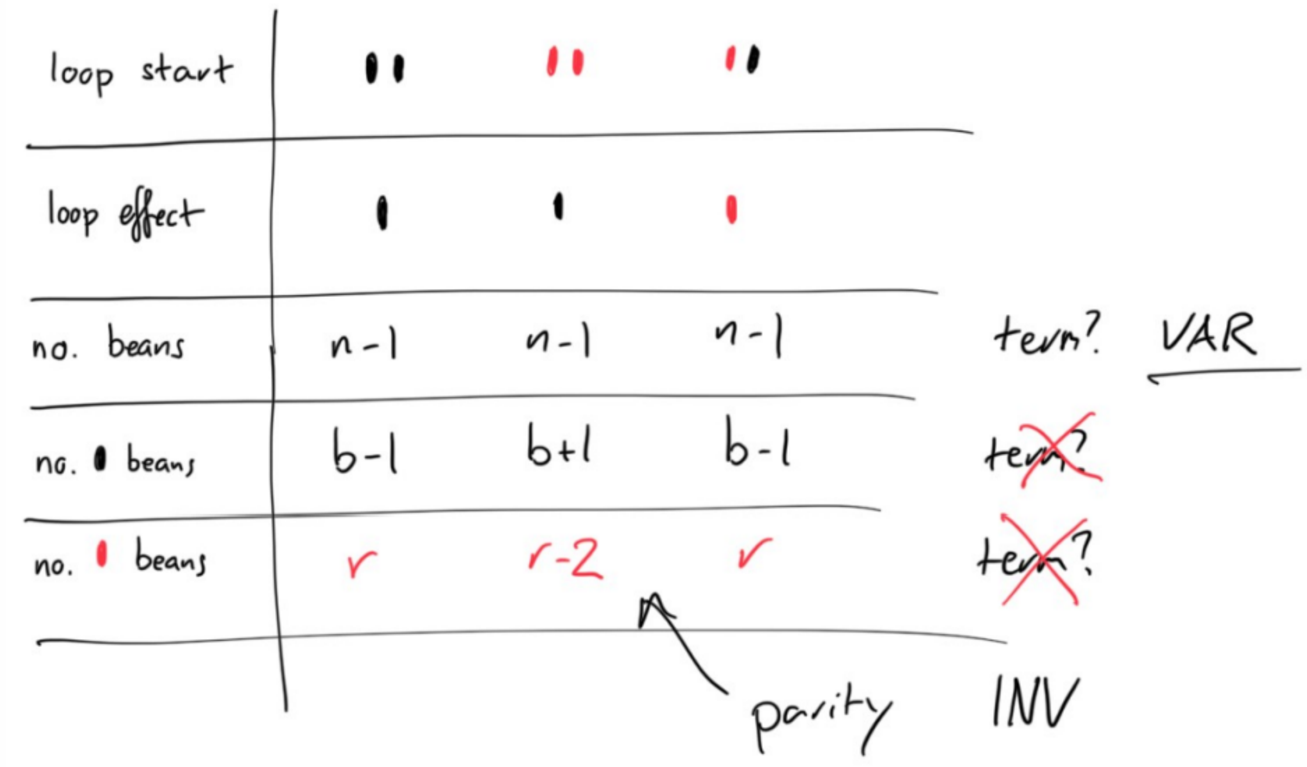
Consider a jar of **red** and **black** coffee beans and an infinite supply of **black** beans. Now perform the following algorithm:

```
put several beans into the jar
while ( jar has more than one bean ){
    select two beans randomly
    if they have the same colour,
        then replace them by a black bean
    if they have different colour,
        then replace them by a red bean
}
```

Question 1. Does this process terminate?

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

The Invariant Graphically



The Coffee Bean Problem

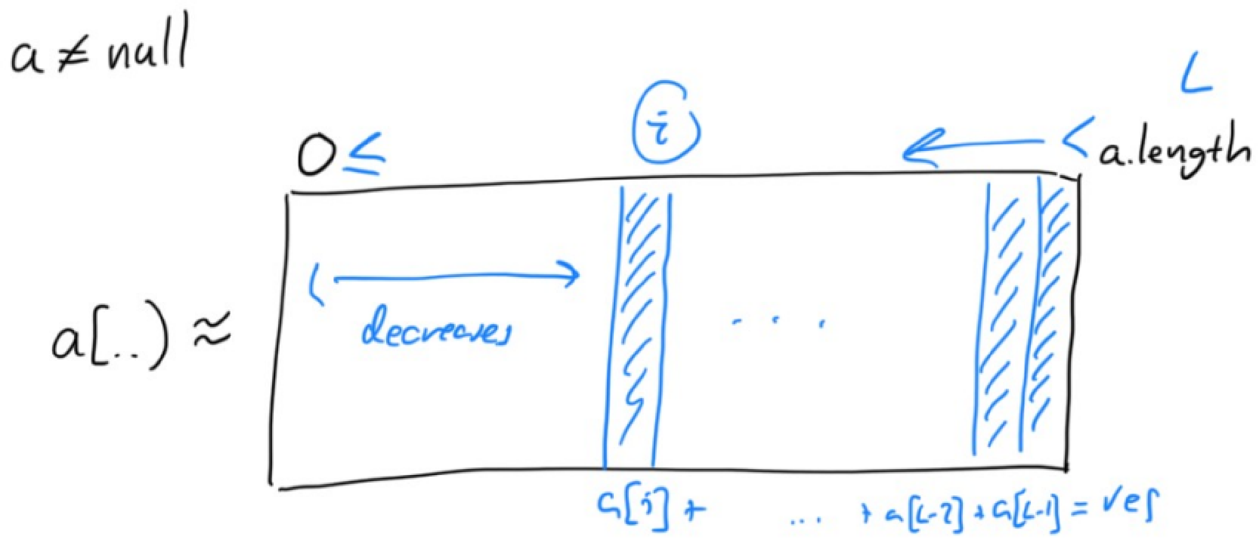
Remember that we reason about loops in terms of variants and invariants:

```
put several beans in the jar
// MID: jar has at least one bean
// INV: parity of red beans
// VAR: number of beans
while ( jar has more than one bean ){
    select two beans randomly
    if they have the same colour,
        then replace them by a black bean
    if they have different colour,
        then replace them by a red bean
}
// MID: colour of last bean in jar =
red if odd number of red beans initially, black otherwise
```

Worked Example - Array Sum Version 1

```
fun sumArray1(a: Array<Int>): Int
// PRE: true                                                        (P)
// POST: $a[..) \approx a[..)_{pre} \;\wedge\; r = \sum a[..)$      (Q)
{
    var res = 0
    var i = 0
    // INV: $a[..) \approx a[..)_{pre}$
    //         $\wedge\; 0 \le i \le a.size \;\wedge\; res = \sum a[..i)$   (I)
    // VAR: $a.size - i$                                            (V)
    while ( i < a.size ) {                                          (cond)
        res += a[i]
        i++
    }
    // MID: $a[..) \approx a[..)_{pre} \;\wedge\; res = \sum a[..)$  (M)
    return res
}
```

The Invariant Graphically

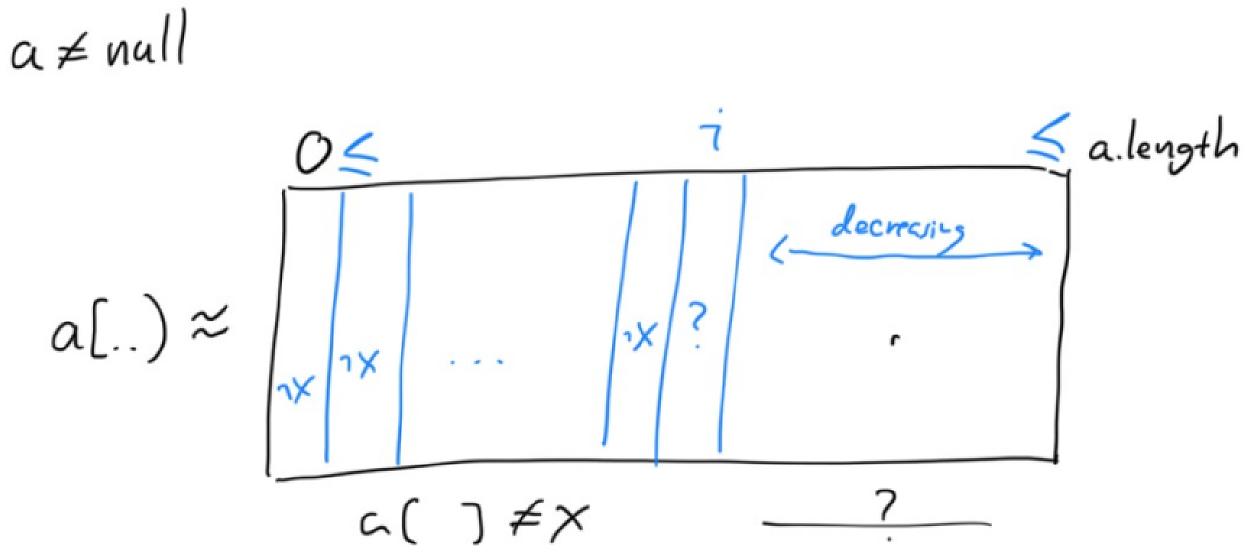


Worked Example - Array Sum Version 2

```
fun sumArray2(a: Array<Int>): Int
// PRE: true                                              (P)
// POST: a[..) ≈ a[..)_pre  ∧  r = ∑a[..)                 (Q)
{
    var res = 0
    var i = a.size - 1
    // INV: a[..) ≈ a[..)_pre  ∧  −1 ≤ i ≤ a.size − 1
    //          ∧  res = ∑a[i+1..)                        (I)
    // VAR: i                                             (V)
    while ( i >= 0 ) {                                 (cond)
        res += a[i]
        i--
    }
    // MID: a[..) ≈ a[..)_pre  ∧  res = ∑a[..)            (M)
    return res
}
```

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

The Invariant Graphically

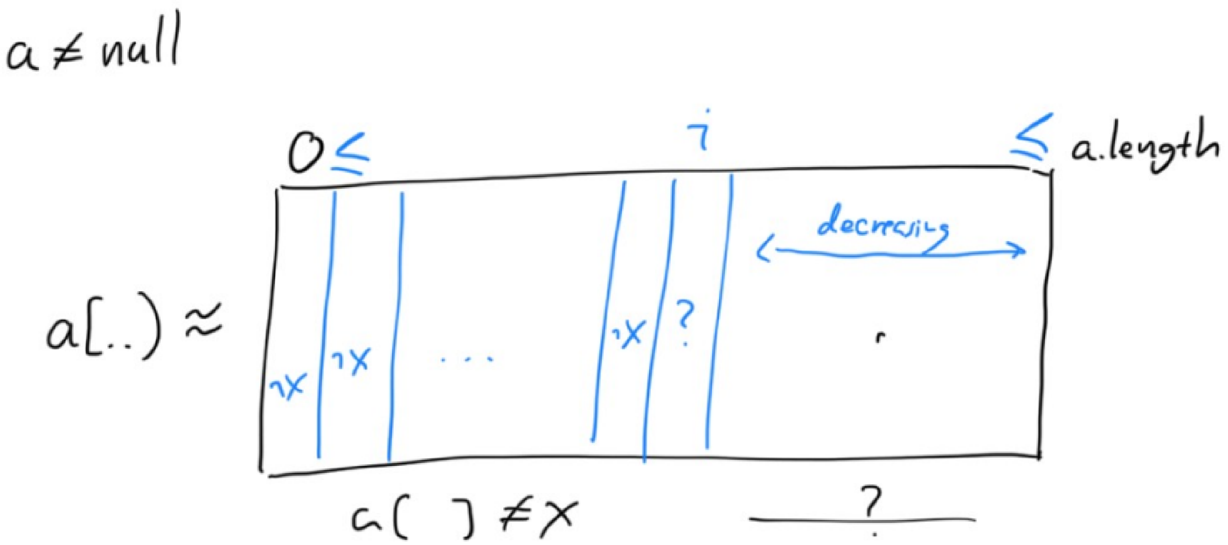


Worked Example - Find Element Version 1

```
fun find1 (a: Array<Int>, x: Int): Int
// PRE: x ∈ a[..)                                              (P)
// POST: 0 ≤ r < a.size  ∧  a[..r)_pre ≠ x  ∧  a[r]_pre = x    (Q)
{
    var i = 0
    // INV: a[..) ≈ a[..)_pre  ∧  x ∈ a[..)
    //         ∧  0 ≤ i ≤ a.size  ∧  a[..i)_pre ≠ x            (I)
    // VAR: a.size − i                                         (V)
    while ((i < a.size) && (a[i] != x)) {                  (cond)
        i++
    }
    // MID: 0 ≤ i < a.size  ∧  a[..i)_pre ≠ x  ∧  a[i]_pre = x  (M)
    return i
}
```

Be aware that the loop in the `find1` function might finish before the variant reaches 0, 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.

The Invariant Graphically

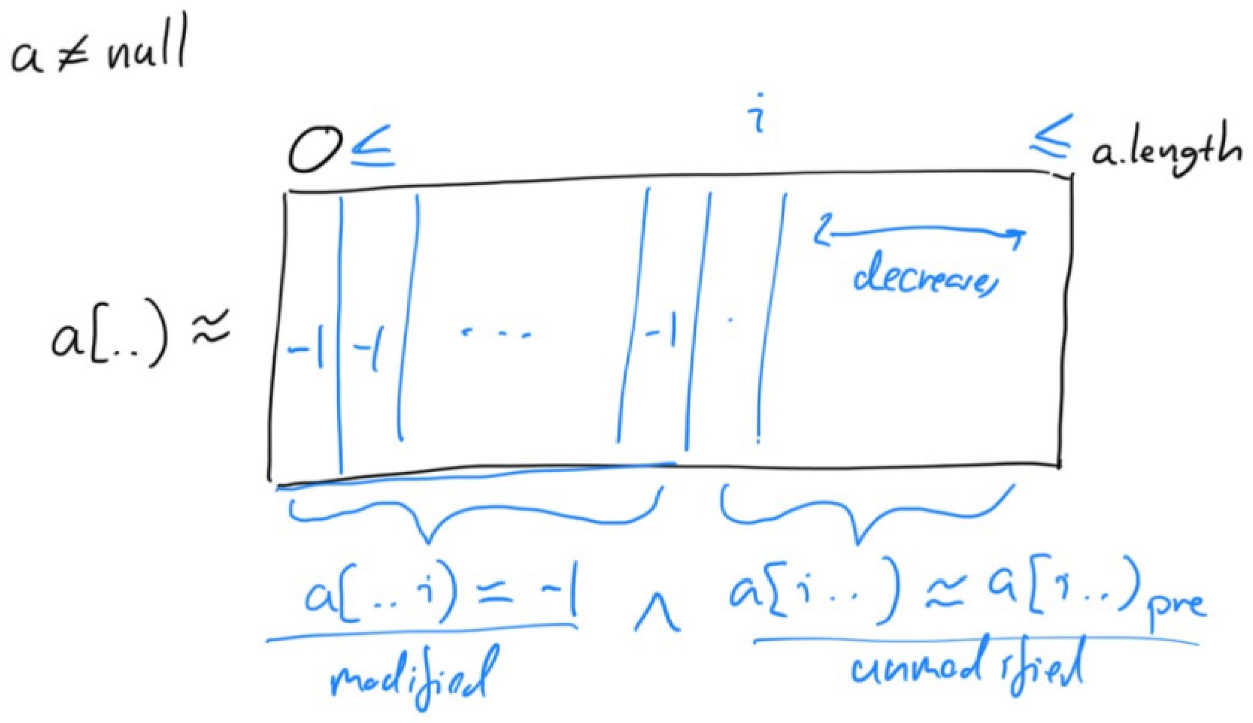


Worked Example - Find Element Version 2

```
fun find2 (a: Array<Int>, x: Int): Int
// PRE: true                                                        (P)
// POST: 0 ≤ r ≤ a.size ∧ a[..r)_pre ≠ x
//          ∧ (r < a.size ⟶ a[r]_pre = x)                          (Q)
    var i = 0
    // INV: a[..) ≈ a[..)_pre
    //          ∧ 0 ≤ i ≤ a.size ∧ a[..i)_pre ≠ x                  (I)
    // VAR: a.size − i                                              (V)
    while ((i < a.size) && (a[i] != x)) {                        (cond)
        i++
    }
    // MID: 0 ≤ i ≤ a.size ∧ a[..i)_pre ≠ x
    //          ∧ (i < a.size ⟶ a[i]_pre = x)                      (M)
    return i
}
```

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.

The Invariant Graphically



Worked Example - Negate Array

```
fun arrayNeg (a: Array<Int>)
// PRE: true                                              (P)
// POST: a[..) = −1                                       (Q)
{
    var i = 0
    // INV: 0 ≤ i ≤ a.size
    //          ∧  a[..i) = −1  ∧  a[i..) ≈ a[i..)_pre    (I)
    // VAR: a.size − i                                    (V)
    while (i < a.size) {                                  (cond)
        a[i] = -1
        i++
    }
    // MID: a[..) = −1                                    (M)
}
```

In this example we see a function that modifies its input array within a loop. We now cannot establish that $\mathrm{a}[..) \approx \mathrm{a}[..)_{\mathrm{pre}}$ throughout the loop, but we can instead track which parts of the array have been modified and which have not been modified.

Note the the specification for `arrayNeg` is quite loose and does not require us to do anything with the contents of the input array. Therefore, the assertion $\mathrm{a}[\mathrm{i}..) \approx \mathrm{a}[\mathrm{i}..)_{\mathrm{pre}}$ in the invariant is arguably redundant.

However, it is good practice to track everything we know about the program state in the invariant, in case it turns out we need to rely on it later.

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 post-condition as a guide to find the mid-condition after the last loop in the code, taking into account any lines of code that appear after loop but before the end of the function.
- Use the mid-condition after the loop and the condition of the loop to help you find the invariant.

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.

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

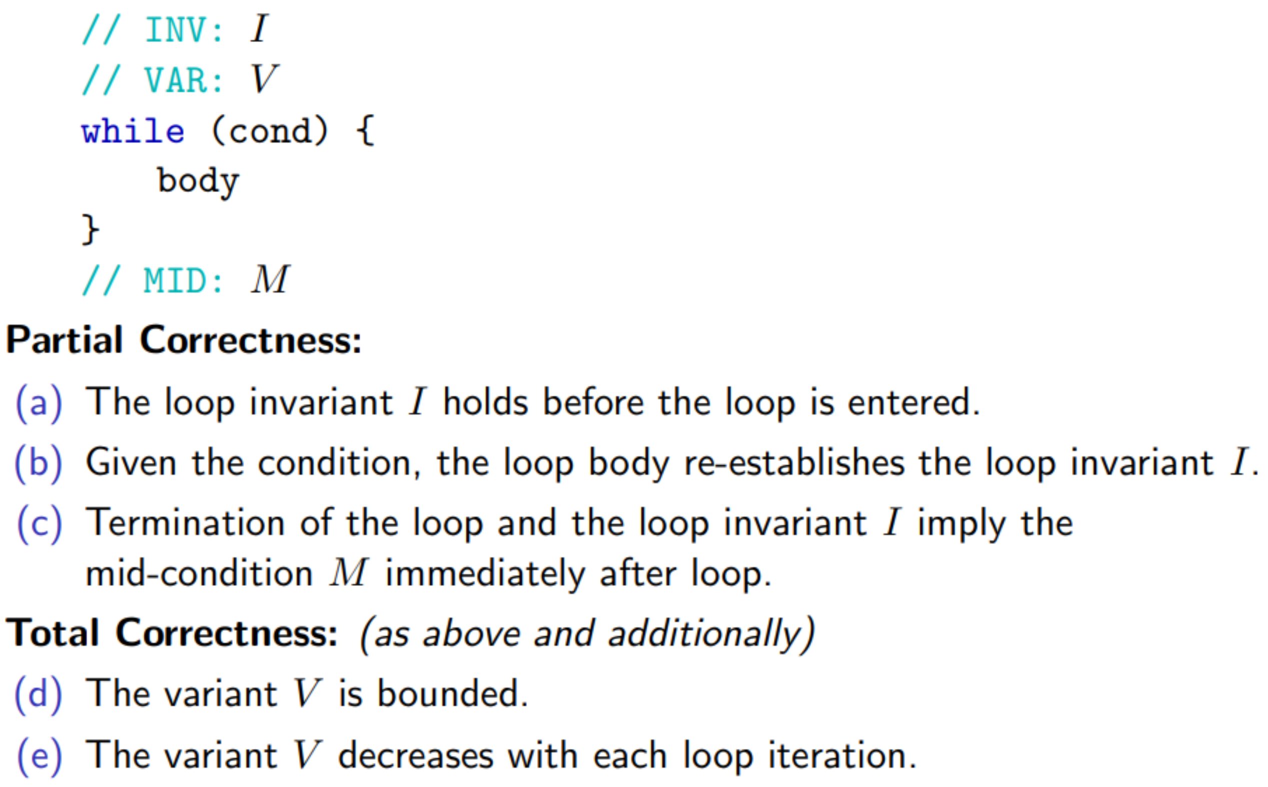
The above Hoare logic rule for `while` captures the **partial correctness** requirements for a while-loop.

Note that this Hoare logic rule imposes three proof obligations on us:

- $\mathrm{P} \implies \mathrm{I}$

- $\{ \mathrm{I} \land \mathrm{cond} \}\ \mathrm{body}\ \{ \mathrm{I} \}$
- $\mathrm{I} \land \neg \mathrm{cond} \implies \mathrm{Q}$

The first and last of these proof obligations are simple assertions on the current program state. However, the second is itself a Hoare Triple, which will require more care.



```
// 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.

Proving **total correctness** imposes two further proof obligations on us, namely:

- $\exists \mathrm{c} \in \mathbb{Z}.\ \{ \mathrm{I} \land \mathrm{cond} \}\ \mathrm{body}\ \{ \mathrm{V} \geq \mathrm{c} \}$
- $\{ \mathrm{I} \land \mathrm{cond} \}\ \mathrm{body}\ \{ \mathrm{V}_{\mathrm{old}} > \mathrm{V} \}$

where $\mathrm{V}_{\mathrm{old}}$ is the variant V with all variables/object attributes modified by the code `body` replaced with their $_{-\mathrm{old}}$ values (i.e. values before running the code `body`)

The purpose of these proof obligations is to show that $\mathrm{I} \land \mathrm{cond}$ hold only a finite number of times

The first of these proof obligations requires us to find a lower-bound for the variant V . Remember that in this module we look for an integer expression that has a fixed lower bound (typically 0), but in general any well-founded ordering (such as increasing integer values or lexicographical ordering) with a limit will do.

The second proof obligation requires us to prove that the variant monotonically decreases on every loop iteration (i.e. moves towards the lower-bound identified above). This obligation requires us to relate the values stored in each program variable used in the variant both before and after the execution of the code `body`

Re-establishing the Loop Invariant

To prove the Hoare triple:

$$\{\ I \ \wedge\ \mathtt{cond}\ \}\ \mathtt{body}\ \{\ I\ \}$$

requires us to show the following:

$$I[\overline{\mathtt{mod}} \mapsto \overline{\mathtt{mod}}_{old}] \ \wedge\ \mathtt{cond}[\overline{\mathtt{mod}} \mapsto \overline{\mathtt{mod}}_{old}] \ \wedge\ \mathtt{body} \ \longrightarrow\ I$$

where $\overline{\mathtt{mod}}$ is the list of variables/object-attributes modified by the code $\mathtt{body}$, denoted: $Mod(\mathtt{body})$

In proving that the loop re-establishes the invariant, we need to take care to track the potential change in state caused by the `body` code. This means that we need to capture the explicit and implicit effects of the code and also make variable substitutions on the initial program state assertions to correctly set-up our proof obligation

To help with this, we make use of the `Mod` function, which gives us a set of all of the variables/object-attributes modified by some code. We then convert the set to a list in alphabetical order, to ensure a consistent renaming of each modified variable or object (array) attribute

As an example consider:

$$\begin{aligned} Mod(\mathtt{res\ =\ res\ +\ a[i]}) &= \{\mathtt{res}\} \\ Mod(\mathtt{a[i]\ =\ res}) &= \{\mathtt{a[i]}\} \\ Mod(\mathtt{i++}) &= \{\mathtt{i}\} \end{aligned}$$

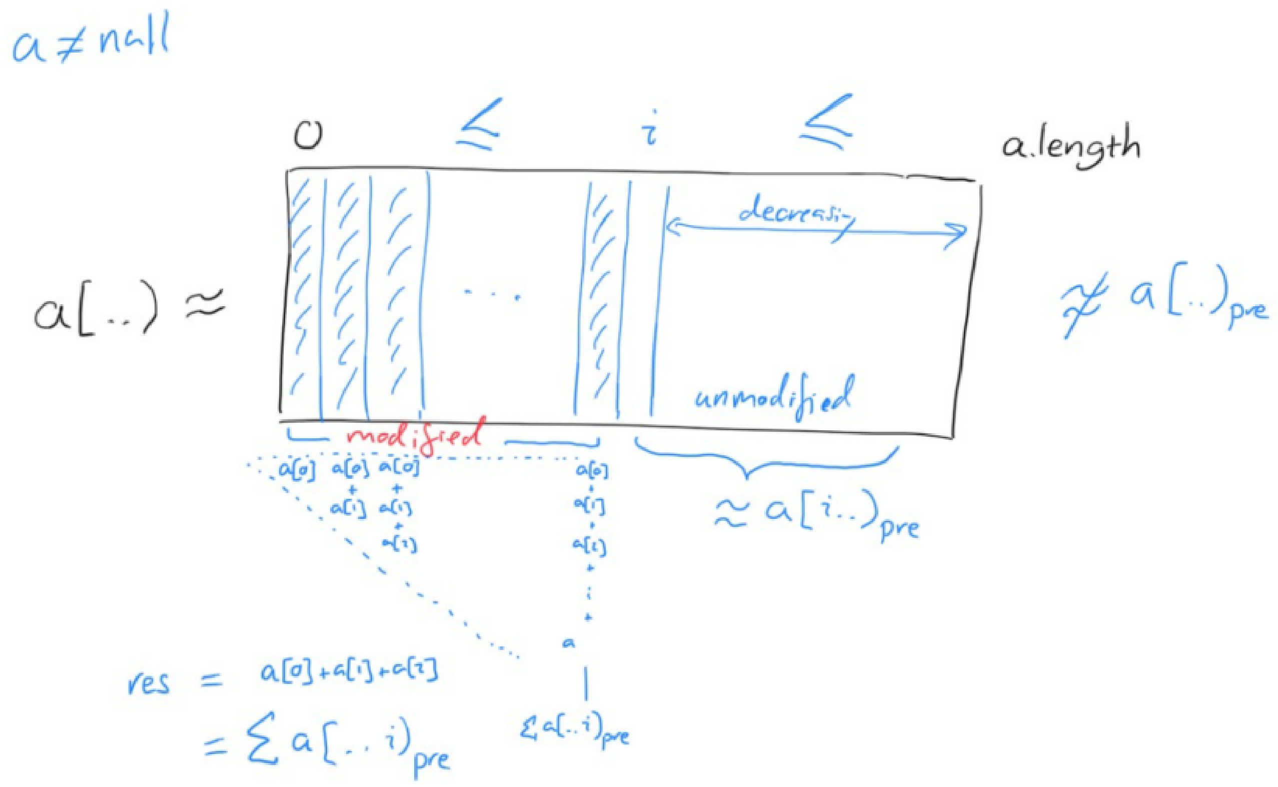
Therefore:

$$Mod(\texttt{res = res + a[i]; a[i] = res; i++;}) = \{\texttt{res}, \texttt{a[i]}, \texttt{i}\}$$

$$\overline{\texttt{mod}} = [\texttt{a[i]}, \texttt{i}, \texttt{res}]$$

Recall, from earlier in the module, that substitutions on array elements often require us to break up the array into the modified and unmodified pieces.

The Invariant Graphically



The most important part of this diagram is tracking which parts of the array `a` have been modified by the loop and which parts are still unmodified from the provided array's contents

It is also useful to note how the value of `res` matches the cumulative sum that is going to be stored in the array at each iteration of the loop

Formal Proof Example - `culSum`

```
fun culSum(a: Array<Int>): Int
// PRE: true                                                                                        (P)
// POST: $\texttt{r} = \sum \texttt{a}[..)_{pre} \ \land \ \forall k \in [0..\texttt{a.size}).[\ \texttt{a}[k] = \sum \texttt{a}[..k{+}1)_{pre}\ ]$        (Q)
{
   // MID: $\texttt{a}[..) \approx \texttt{a}[..)_{pre}$                                                              (M_0)
   var res = 0
   var i = 0
   // INV: $0 \leq \texttt{i} \leq \texttt{a.size} \ \land \ \texttt{a}[\texttt{i}..) \approx \texttt{a}[\texttt{i}..)_{pre}$
              $\land \ \texttt{res} = \sum \texttt{a}[..\texttt{i})_{pre} \ \land \ \forall k \in [0..\texttt{i}).[\ \texttt{a}[k] = \sum \texttt{a}[..k{+}1)_{pre}\ ]$     (I)
   // VAR: $\texttt{a.size} - \texttt{i}$                                                                      (V)
   while (i < a.size) {
      res += a[i]
      a[i] = res
      i++
   }
   // MID: $\texttt{res} = \sum \texttt{a}[..)_{pre}$
              $\land \ \forall k \in [0..\texttt{a.size}).[\ \texttt{a}[k] = \sum \texttt{a}[..k{+}1)_{pre}\ ]$                    (M)
   return res
}
```

This function 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

Note that due to Kotlin's call by value nature, and that `culSum` does not shadow the array reference a, we do not need to track the distinction between a and $\mathrm{a}_{\mathrm{pre}}$ or similarly $\mathrm{a.size}$ and $\mathrm{a.size}_{\mathrm{pre}}$ in any of our assertions

However, the function does modify the contents of the input array a, so we do need to track the state of the array contents carefully. We have added an extra mid-condition M_0 to capture the initial (unmodified) contents of the input array a

`culSum` - (a) Invariant holds before loop is entered

$$\boxed{\begin{array}{c} M_0 \ \land\ \texttt{var res = 0} \ \land\ \texttt{var i = 0} \\ \longrightarrow \\ I \end{array}}$$

Given:

(1)	$\texttt{a}[..) \approx \texttt{a}[..)_{pre}$	from M_0
(2)	$\texttt{res} = 0$	from code line 6
(3)	$\texttt{i} = 0$	from code line 7

To show:

(α)	$0 \leq \texttt{i} \leq \texttt{a.size}$	I
(β)	$\texttt{a}[\texttt{i}..) \approx \texttt{a}[\texttt{i}..)_{pre}$	I
(γ)	$\texttt{res} = \sum \texttt{a}[..\texttt{i})_{pre}$	I
(δ)	$\forall k \in [0..\texttt{i}).[\ \texttt{a}[k] = \sum \texttt{a}[..k+1)_{pre}\ ]$	I

In this proof obligation, we are verifying that the loop invariant (I) is established by the function's initialisation code. In this case this is simply the initialisation of the variables res and i with 0 values and the **implicit** information (from M_0) that the contents of the array a have not been modified **yet**

Note that no substitutions are required on the mid-condition M_0, because there are no modifications to any of the existing program state. The variables res and i are being declared for the first time by this code, and this code doesn't modify the input array a

Note that we could choose to substitute $\mathrm{res}_{\mathrm{old}}$ for res and $\mathrm{i}_{\mathrm{old}}$ for i in the mid-condition M_0 anyway, but since neither of these variables exists before we run the initialisation code, this would be a vacuous substitution

We will always choose to omit such vacuous substitutions to keep our notation a little lighter, where possible

`culSum` - (a) Invariant holds before loop is entered

Proof:

(4)	$a \neq \texttt{null}$	from type of a
(5)	$0 \leq 0 \leq \texttt{a.size}$	from (4)

α follows from (5) and (3)

(6)	$a[0..) \approx a[0..)$	from (1)

β follows from (6) and (3)

(7)	$\sum a[..0)_{pre} = 0$	from def. of $\sum$
(8)	$\sum a[..\texttt{i})_{pre} = 0$	from (7) and (3)

γ follows from (8) and (2)

(9)	$\forall k \in [0..0).[\ \texttt{a}[k] = \sum a[..k+1)_{pre}\ ]$	from empty range

δ follows from (9) and (3)

The proof of invariant initialisation is not that demanding, but it is important to try to start the proof of each invariant conjunct from some known mathematical or logical step, so that we are being clear and precise with our reasoning

The conclusion of each invariant conjunct proof above is always a simple substitution of a concrete value with its corresponding initialised variable. This is common for all such invariant initialisation proofs

`culSum` - (b) Loop body re-establishes invariant

$$
\begin{array}{c}
I[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] \\
\wedge\ (\texttt{i < a.size})[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] \\
\wedge\ \texttt{res += a[i]}\ \wedge\ \texttt{a[i] = res}\ \wedge\ \texttt{i++} \\
\longrightarrow \\
I
\end{array}
$$

Given:

(1)	$0 \leq \texttt{i}_{old} \leq \texttt{a.size}$	from I
(2)	$\texttt{a[i}_{old}\texttt{]}_{old} = \texttt{a[i}_{old}\texttt{]}_{pre}$	from I
(3)	$a[\texttt{i}_{old}+1..) \approx a[\texttt{i}_{old}+1..)_{pre}$	from I
(4)	$\texttt{res}_{old} = \sum a[..\texttt{i}_{old})_{pre}$	from I
(5)	$\forall k \in [0..\texttt{i}_{old}).[\ \texttt{a}[k] = \sum a[..k+1)_{pre}\ ]$	from I
(6)	$\texttt{i}_{old} < \texttt{a.size}$	cond
(7)	$\texttt{res} = \texttt{res}_{old} + \texttt{a[i}_{old}\texttt{]}_{old}$	from code line 12
(8)	$\texttt{a[i}_{old}\texttt{]} = \texttt{res}$	from code line 13
(9)	$\texttt{i} = \texttt{i}_{old} + 1$	from code line 14

The set-up of the givens for the proof that the loop body re-establishes the invariant may look rather intimidating, but this can actually be broken down into simpler pieces that should each be easily understandable

First, we have the givens (1-5) that come from the fact that the loop invariant (I) must hold at the start of each loop iteration. These assertions need to have any variables that are modified by the code in the loop body replaced with their $_{\mathrm{old}}$ values. This introduces a slight challenge for the second conjunct of the invariant I, namely $\mathrm{a[i..)} \approx \mathrm{a[i..)_{pre}}$, which describes a range within which one element $\mathrm{a[i]}$ is updated by the loop body. We must consider the implicit effects of the code in the loop body (i.e. those things which are not modified by the code), and we do this by breaking up the array property from the invariant into the part of the array that will be modified and the part that will not be modified. Thus, we obtain the givens (2) and (3)

Next, we have a given (6) that comes directly from the fact that the loop condition (cond) must hold for the program's execution to have entered the loop body

Finally, we have the givens (7-9) that come from the logical and mathematical descriptions of the explicit effects of the code in the loop body

`culSum` - (b) Loop body re-establishes invariant

$$\boxed{\begin{array}{c} I[\mathtt{a[i]} \mapsto \mathtt{a[i]}_{old}, \mathtt{i} \mapsto \mathtt{i}_{old}, \mathtt{res} \mapsto \mathtt{res}_{old}] \\ \land\ (\mathtt{i} < \mathtt{a.size})[\mathtt{a[i]} \mapsto \mathtt{a[i]}_{old}, \mathtt{i} \mapsto \mathtt{i}_{old}, \mathtt{res} \mapsto \mathtt{res}_{old}] \\ \land\ \mathtt{res} \mathrel{+}= \mathtt{a[i]}\ \land\ \mathtt{a[i]} = \mathtt{res}\ \land\ \mathtt{i}{+}{+} \\ \longrightarrow \\ I \end{array}}$$

To show:

$$\begin{array}{lll} (\alpha) & 0 \le \mathtt{i} \le \mathtt{a.size} & I \\ (\beta) & \mathtt{a[i..)} \approx \mathtt{a[i..)}_{pre} & I \\ (\gamma) & \mathtt{res} = \sum \mathtt{a[..i)}_{pre} & I \\ (\delta) & \forall k \in [0..\mathtt{i}).[\ \mathtt{a}[k] = \sum \mathtt{a}[..k{+}1)_{pre}\ ] & I \end{array}$$

`culSum` - (b) Loop body re-establishes invariant

Proof:

(10)	$0 \le \mathtt{i}_{old} < \mathtt{a.size}$	from (1) and (6)
(11)	$0 \le \mathtt{i}_{old} + 1 \le \mathtt{a.size}$	from (10) by arith.

α follows from (11) and (9)

β follows from (3) and (9)

(12)	$\mathtt{res} = \sum \mathtt{a}[..\mathtt{i}_{old})_{pre} + \mathtt{a[i}_{old}\mathtt{]}_{old}$	from (7) and (4)
(13)	$\mathtt{res} = \sum \mathtt{a}[..\mathtt{i}_{old})_{pre} + \mathtt{a[i}_{old}\mathtt{]}_{pre}$	from (12) and (2)
(14)	$\mathtt{res} = \sum \mathtt{a}[..\mathtt{i}_{old}+1)_{pre}$	from (13) and def. of $\sum$

γ follows from (14) and (9) [$\mathtt{a[i}_{old}\mathtt{]}$ array access valid by (10)]

(15)	$\mathtt{a[i}_{old}\mathtt{]} = \sum \mathtt{a}[..\mathtt{i}_{old}+1)_{pre}$	from (8) and (14)
(16)	$\forall k \in [0..\mathtt{i}_{old}+1).[\ \mathtt{a}[k] = \sum \mathtt{a}[..k+1)_{pre}\ ]$	from (5) and (15)

δ follows from (16) and (9) [$\mathtt{a[i}_{old}\mathtt{]}$ array access valid by (10)]

`culSum` - (c) Mid-condition holds straight after loop

$$\boxed{\begin{array}{c} I \ \wedge\ \neg(\mathtt{i < a.size}) \\ \longrightarrow \\ M \end{array}}$$

Given:

(1)	$0 \le \mathtt{i} \le \mathtt{a.size}$	from I
(2)	$\mathtt{a}[\mathtt{i}..) \approx \mathtt{a}[\mathtt{i}..)_{pre}$	from I
(3)	$\mathtt{res} = \sum \mathtt{a}[..\mathtt{i})_{pre}$	from I
(4)	$\forall k \in [0..\mathtt{i}).[\ \mathtt{a}[k] = \sum \mathtt{a}[..k+1)_{pre}\ ]$	from I
(5)	$\mathtt{i} \ge \mathtt{a.size}$	$\neg$cond

To show:

(α)	$\mathtt{res} = \sum \mathtt{a}[..)_{pre}$	M
(β)	$\forall k \in [0..\mathtt{a.size}).[\ \mathtt{a}[k] = \sum \mathtt{a}[..k+1)_{pre}\ ]$	M

The proof obligation for showing that the mid-condition (in this case M) holds straight after the loop is in the context where we know that the loop invariant (I) must still hold (we've only just left the loop), but we also know that the loop condition must also have just evaluated to false

This gives rise to the informal obligation: $\mathrm{I} \wedge \neg\mathrm{cond} \rightarrow \mathrm{M}$

The most interesting part of this obligation is what we get from the $\neg\mathrm{cond}$ conjunct (the rest is just writing out I and M)

In this particular example, the loop condition (cond) is $\mathrm{i} < \mathrm{a.size}$

Thus, $\neg\mathrm{cond} \triangleq \neg(\mathrm{i} < \mathrm{a.size}) \leftrightarrow \mathrm{i} \geq \mathrm{a.size}$

It is important that we only take what is given to us by the negation of the loop condition at this point in the proof set-up

During the proof itself, we will be able to deduce that $\mathrm{i} = \mathrm{a.size}$ from the above $\mathrm{i} \geq \mathrm{a.size}$ and the loop invariant's $0 \leq \mathrm{i} \leq \mathrm{a.size}$ (i's exact equality being the only possible state that satisfies both assertions)

However, this derivation is a part of the proof itself, **not** something that you should write in the givens for this proof obligation

`culSum` - (c) Mid-condition holds straight after loop

Proof:

(6) $\mathtt{i} = \mathtt{a.size}$ from (1) and (5)

(7) $\mathtt{res} = \sum \mathrm{a}[..\mathtt{a.size})_{pre}$ from (3) and (6)

α follows from (7) and def. of a[..)

β follows from (4) and (6)

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

It is worth briefly remarking that substituting (6) into (2) leads to the assertion: $\mathrm{a}[\mathrm{a.size}..) \approx \mathrm{a}[\mathrm{a.size}..)_{\mathrm{pre}}$ but this actually describes an empty range. So, we no longer have any assertion about the unmodified part of the array

However, this make sense, as in our desired post-condition all of the array contents have been updated.

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

$$\begin{gathered} I[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] \\ \land\ (\texttt{i < a.size})[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] \\ \land\ \texttt{res += a[i]} \ \land\ \texttt{a[i] = res} \ \land\ \texttt{i++} \\ \longrightarrow \\ V \geq 0 \ \land\ V[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] > V \end{gathered}$$

Given:

(1)	$0 \leq \texttt{i}_{old} \leq \texttt{a.size}$	from I
(2)	$\texttt{a[i}_{old}\texttt{]}_{old} = \texttt{a[i}_{old}\texttt{]}_{pre}$	from I
(3)	$\texttt{a}[\texttt{i}_{old}{+}1..) \approx \texttt{a}[\texttt{i}_{old}{+}1..)_{pre}$	from I
(4)	$\texttt{res}_{old} = \sum \texttt{a}[..\texttt{i}_{old})_{pre}$	from I
(5)	$\forall k \in [0..\texttt{i}_{old}).[\ \texttt{a[}k\texttt{]} = \sum \texttt{a}[..k{+}1)_{pre}\ ]$	from I
(6)	$\texttt{i}_{old} < \texttt{a.size}$	cond
(7)	$\texttt{res} = \texttt{res}_{old} + \texttt{a[i}_{old}\texttt{]}_{old}$	from code line 12
(8)	$\texttt{a[i}_{old}\texttt{]} = \texttt{res}$	from code line 13
(9)	$\texttt{i} = \texttt{i}_{old} + 1$	from code line 14

Since the proof set-up is identical for each termination property, we tend to prove both properties at the same time

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

Also note that even though the only program variable used in our variant V for this code is i, we still perform the full substitutions on the old value of the variant $\mathrm{V}_{\mathrm{old}}$ just in case. Of course, only the i variable will be affected by this

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

$$\begin{gathered} I[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] \\ \land\ (\texttt{i < a.size})[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] \\ \land\ \texttt{res += a[i]} \ \land\ \texttt{a[i] = res} \ \land\ \texttt{i++} \\ \longrightarrow \\ V \geq 0 \ \land\ V[\texttt{a[i]} \mapsto \texttt{a[i]}_{old}, \texttt{i} \mapsto \texttt{i}_{old}, \texttt{res} \mapsto \texttt{res}_{old}] > V \end{gathered}$$

To show:

(α)	$\texttt{a.size} - \texttt{i} \geq 0$	$V \geq 0$
(β)	$\texttt{a.size} - \texttt{i}_{old} > \texttt{a.size} - \texttt{i}$	$V_{old} > V$

Proof:

(10) $\mathtt{i}_{old} + 1 \leq \mathtt{a.size}$ from (6)
(11) $\mathtt{i} \leq \mathtt{a.size}$ from (10) and (9)
(12) $0 \leq \mathtt{a.size} - \mathtt{i}$ from (11) ($-\mathtt{i}$ from both sides)
α follows from (12) by rearrangement

(13) $\mathtt{i}_{old} < \mathtt{i}$ from (9)
(14) $-\mathtt{i}_{old} > -\mathtt{i}$ from (13) ($\times$ both sides by -1)
(15) $\mathtt{a.size} - \mathtt{i}_{old} > \mathtt{a.size} - \mathtt{i}$ from (14) (add $\mathtt{a.size}$ to both sides)
β follows directly from (15)

Are we nearly there yet...?

Have we now shown that the sum function satisfies its specification?

- We still need to show that the code after the loop establishes the post-condition

Have we shown that the function 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 integer overflow on lines 12 or 14

`culSum` - (f) Post-condition established

$$\boxed{\begin{array}{c} M \ \land \ \texttt{return res} \\ \longrightarrow \\ Q \end{array}}$$

Given:

(1)	$\texttt{res} = \sum \texttt{a}[..)_{pre}$	from M
(2)	$\forall k \in [0..\texttt{a.size}).[\ \texttt{a}[k] = \sum \texttt{a}[..k{+}1)_{pre}\ ]$	from M
(3)	$\texttt{r} = \texttt{res}$	from code line 18

To show:

(α)	$\texttt{r} = \sum \texttt{a}[..)_{pre}$	Q
(β)	$\forall k \in [0..\texttt{a.size}).[\ \texttt{a}[k] = \sum \texttt{a}[..k{+}1)_{pre}\ ]$	Q

Proof:

α follows from (3) and (1)

β follows directly from (2)

`Culsum` - (g) Integer overflows

Remember that for the scope of this module we assume a “perfect” machine that can represent any integer

In this module 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

Comparison with Recursion

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

- Loop invariants extract the essence of what a loop is promising to achieve
- Function specifications extract the essence of what a function 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 pre-condition 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 pre-condition 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 pre-condition of `sumAux` holds before the recursive call and that its return can be used to prove the post-condition of the caller
- (c) Proving that the invariant and negation of the loop condition imply the mid-condition after the loop, corresponds to proving that on exiting the body of `sumAux` we satisfy the post-condition of `sum`