# Lectures Notes on Loops and Proofs in Hoare Logic

Software Verification

Nova School of Science and Technology
Mário Pereira    mjp.pereira@fct.unl.pt

Version of October 27, 2025

These notes are based on lecture notes by Bernardo Toninho.

## 1 Hoare Logic Revisited

Recall the rules of *Hoare Logic* defined over triples, of the form $\{P\}\, S\ \{Q\}$, capturing the effect of program fragments of a small imperative language with loops ($S$) and assertions over a set of predefined state variables as pre-conditions ($P$) and post-conditions ($Q$).

A Hoare triple $\{P\}\, S\ \{Q\}$ can be understood as "If program statement $S$ is executed in a state where the pre-condition $P$ holds, then, if the execution of $S$ terminates, it is guaranteed that the post-condition $Q$ holds in its final state". Of course, this statement says nothing about whether a final state is actually reachable and provides no guarantees in the case where the program does not terminate. The rules that comprise this system of Hoare logic, presented earlier, are the following:

$$\frac{}{\{A\}\,\texttt{skip}\,\{A\}}\ \textsc{Skip} \qquad \frac{\{A\}\,e_1\,\{B\} \qquad \{B\}\,e_2\,\{C\}}{\{A\}\,e_1; e_2\,\{C\}}\ \textsc{Sequence}$$

$$\frac{\{A \land t\}\,e_1\,\{B\} \qquad \{A \land \neg t\}\,e_2\,\{B\}}{\{A\}\,\texttt{if}\ t\ \texttt{then}\ e_1\ \texttt{else}\ e_2\,\{B\}}\ \textsc{Conditional} \qquad \frac{}{\{P[x \mapsto t]\}\,x := t\,\{P\}}\ \textsc{Assign}$$

$$\frac{\models P \implies P' \qquad \{P'\}\,e\,\{Q'\} \qquad \models Q' \implies Q}{\{P\}\,e\,\{Q\}}$$

In general, rule (Assign) also allows for the deduction

$$\{\mathit{true}\}\,x := E\,\{x = E\}$$

whose proof results from combining the rule (Assign) and rule (Consequence) in a derivation with the form

$$\frac{(E = E) \implies \mathit{true} \qquad \{E = E\}\,x := E\,\{x = E\}}{\{\mathit{true}\}\,x := E\,\{x = E\}}$$

We have also intuitively discussed a notion of *strongest post-condition* and its dual concept, *weakest pre-condition.* The strongest post-condition is an assertion $A$ such that all other valid post-conditions for the given program statement *are implied* by $A$. Dually, the weakest pre-condition $B$ is such that all other valid pre-conditions of the program statement imply $B$. Generally, we want pre-conditions to be as *weak* as possible (*i.e.*, as unrestrictive as they can, while the correctness of the code is guaranteed) and post-conditions to be as *strong* as possible (*i.e.*, to provide the most precise constraint possible on the outcome of the execution of a program).

## 2 Loops and Loop Invariants

To understand how the rule for loops works, it is instructive to think of a loop as if it were unfolded into an arbitrarily deep nesting of conditionals as follows:

```
while E do S ≜ if E then S ;
                   if E then S;
                       if E then S, ...
                       else skip
                   else skip
                 else skip
```

With this structure in mind, let us now consider an assertion $J$ as the strongest post-condition for $S$, the body of the loop. We can then see that it must also be the pre-condition for the next iteration of the loop, which is just another execution of $S$.

Using the rule (CONDITIONAL) as a reference, we can see that all the conditional statements should have the same approximate post-condition and that the pre-condition of all these statements matches their post-condition together with the condition in the loops guard. Also, we know that after the last iteration of the loop (assuming there is one), the loop condition must be false. This reasoning is represented in the following rule:

$$\frac{\{J \wedge t\}\, e\, \{J\}}{\{J\}\, \texttt{while t do e}\, \{J \wedge \neg t\}}$$

where the assertion $J$ is called the *loop invariant.* A loop invariant is a condition that (1) must hold at the entry point of a loop, (2) is valid at the start of each iteration (as well as the loop's guard condition), (3) must be valid at the end of each iteration, and (4) is valid at the exit point of the loop (together with the negation of the guard of the loop).

The verification of a loop is related to the use of induction in the verification of programs. To verify a loop, we must show that the invariant $J$ holds at the entry point of the loop, which is akin to showing the inductive base case. We then show that, assuming $J$ (and the loop condition) we can show that $J$ holds *after* one execution of the loop body. This is essentially showing the inductive case: we assume the property of interest $J$ holds of the previous iteration and show that it holds after one more execution of the loop body. This provides a sound proof technique for loops in just the same way that mathematical induction is a sound proof technique for natural numbers: For any given (finite) number of iterations of the loop, we know that the invariant will hold afterwards: if the loop body is never executed, then the fact that the invariant $J$ held before the loop means that it must still hold (and that the loop condition is false); if the loop runs for one iteration, then we know the loop body preserves $J$, and since $J$ held at loop entry, we know it will hold after the iteration. And so on for any given number of iterations the loop runs for.

Note that, just as finding the right inductive property that makes our intended result go through can often require a great deal of ingenuity, loop invariants cannot be automatically inferred in the general case and require human input. In most cases (but not all), a good rule of thumb is that the loop invariant can be determined from the post-condition of the loop by replacing the limits of the loop by its cursor.

Consider the following example of a Hoare triple for a program that iteratively counts up from $0$ to $n$:

```
{0 ≤ n}
i := 0;
while i < n do {
  i := i + 1
}
{i = n}
```

If we write intermediate assertions in this program, we obtain the listing below (we write Hoare triples in a more compact form for the sake of conciseness):

```
{0 ≤ n}
i := 0;
{i = 0 ∧ 0 ≤ n}
{0 ≤ i ≤ n}
while i < n do {
  {0 ≤ i ≤ n ∧ i < n}
  {0 ≤ i < n}
  {0 ≤ i + 1 ≤ n}
  i := i + 1
  {0 ≤ i ≤ n}
}
{0 ≤ i ≤ n ∧ i >= n}
{i = n}
```

Notice the assertion $\{0 \leq i \leq n\}$ that can be proven true before the loop is such that, if assumed at the start of the loop body, can be established as a post-condition for the loop body. This means that $\{0 \leq i \leq n\}$ is an invariant for the loop. Moreover, when combined with the negated loop guard $i >= n$ we can deduce that $i = n$, which is the post-condition we want. The general reasoning technique consists of establishing the loop invariant before the loop, then breaking the invariant (to "make progress") and re-establishing it at the end of the loop body.

We now consider a slightly more interesting loop: the sum of all numbers from $0$ to $n$ and the proof that the sum of consecutive $n$ numbers is given

$$\frac{n(n+1)}{2}$$

The Why3 code that computes the sum is:

```
 1 let sum_iter (n:int) : (res:int)
 2   requires { n >= 0 }
 3   ensures  { res * 2 = n * (n+1) }
 4 = let ref s = 0 in
 5   let ref i = 0 in
 6   assert { s * 2 = i * (i+1) };
 7   while i < n do
 8     variant   { n - i }
 9     invariant { 0 <= i <= n }
10     invariant { s * 2 = i * (i+1) }
11     i <- i + 1;
12     assert { s * 2 = (i-1) * ((i-1)+1) };
13     s <- s + i;
14     assert { s * 2 = i * (i+1) }
15   done;
16   assert { i = n };
17   assert { s * 2 = n * (n+1) };
18   s
```

The loop invariant, declared in line 10, is valid at the beginning of each iteration, is broken by the increment of line 11, and re-established by the assignment of line 13. Notice that we need an extra invariant $0 \leq i \leq n$ that helps to frame the possible values of variable $i$. The conclusion $i = n$ in line 16, derived from the invariant above and the negation of the condition, support the post-condition of the method.

Notice that loop invariants are usually intuitive conditions over the "progress" of a loop. They are part of common developer intuitive reasoning when building such programs, but formulating them precisely and in the "right way" can require some practice. For instance, when determining the maximum value of the array, a common loop invariant is that "all elements to the left of the cursor are smaller or equal to the value computed up to any point in the loop"; when searching an unsorted array, the invariant can be that "all elements to the left of the cursor are different from the value being searched"; when searching a sorted array using binary search, the invariant should say that "the searched element is between the lower and the higher limits (cursors)"; when sorting an array it should be possible to maintain the information that "all elements to the left of the cursor are already sorted"; and when reversing a list we know that "that all elements to the left of the cursor are already placed on the right-hand side of the resulting list".

A quick note on a difference between Why3 and Hoare logic: the **`variant`** clause in line 8 provides a measure that can be shown to *strictly* decrease with each loop iteration. Moreover, the value can be shown to be *bounded*, so that it cannot decrease forever. This means that Why3 actually proves that the loop terminates, and so proves something *stronger* than the Hoare logic rule for loops given above.

## 3 Case Study: Fast Exponentiation

We now carry out an extended verification example of an implementation of "fast" exponentiation, which computes an exponent of an integer using fewer multiplications than the naive implementation of exponentiation (which performs as many multiplications as the value of the exponent). The algorithm actually computes $\lfloor \log(n) \rfloor$ squares and at most $\lfloor \log(n) \rfloor$ multiplications, where $\lfloor \cdot \rfloor$ is the floor function and $n$ is the value of the exponent.

Before looking at the code, and certainly before verifying it, we must first understand the key insights of the algorithm. The algorithm reduces the number of multiplications by exploiting the observation that, when exponents are powers of two, we can perform the overall calculation by *repeated squaring*: to calculate a number of the form $b^{2^n}$, for some positive $n$, we compute a square $n$ times (e.g. $3^{16} = 3^{2^4} = (3^2)^8 = 9^8 = (9^2)^4 = (81^2)^2 = 6561^2 = 43046721$). Of course, not all exponents powers of two, but all natural numbers can be expressed as a *sum* of powers of two, and so we can always write an exponent as a sum of powers of two (think of writing a number in base 2). This means that we can calculate arbitrary powers using this approach, even those that are not of the form $b^{2^n}$, by first writing the exponent as a sum of powers of two and then calculating the product of the corresponding powers of two. For example, if we consider $3^{18}$, we can write 18 as $16 + 2 = 2^4 + 2$ and therefore $3^{18} = 3^{(2+16)} = 3^2 \cdot 3^{16}$, which we can now compute using the repeated squaring approach:

$$3^2 = 9,$$
$$3^{16} = 3^{2^4} = (3^2)^8 = 9^8 = (9^2)^4 = 81^4 = (81^2)^2 = 6561^2 = 43046721$$

and so we now simply compute $3^2 \times 3^{16} = 9 \times 43046721 = 387420489$.

In fact, we can do even better. Instead of first going through the repeated squaring and then multiplying the various powers we can actually combine both steps in a single iterative process. In each iteration we compute a square and at the same time determine whether or not that power of two is used in the exponent as a sum of powers, effectively exploiting the following property of all natural numbers $n$:

$$x^n = \begin{cases} x \times (x^2)^{\frac{n-1}{2}} & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}} & \text{if } n \text{ is even} \end{cases} \tag{1}$$

Note that, when dealing with integers in code, the property above essentially states that if the exponent $n$ is even, then $x^n = (x^2)^{(n/2)}$, where $/$ denotes euclidian (or integer) division. If $n$ is odd, then $x^n = x * (x^2)^{(n/2)}$.

Using this revised approach, we can calculate $3^{18}$ as follows: Consider the variables $r$ for the result, and $c$ for the current base. Start with $r \leftarrow 1$, which is $3^0$ and $c \leftarrow 3$, which is $3^1$. Since 18 is even, square $c$ ($c \leftarrow c * c$) and $c$ is now $9 = (3^2)$ and halve the exponent to 9. Since 9 is odd, our result $r$ is updated to $r \leftarrow r * c = (3^0) * (3^2)$, the exponent is updated to 4 and $c$ is squared $c \leftarrow c * c = 9^2$. Since 4 is even, square $c$ ($c \leftarrow c * c = 81^2$) and halve the exponent to 2. Since 2 is even, square $c$ one more time ($c = 6561^2 = 3^{16}$) and halve the exponent to 1. Since 1 is odd, update $r$ ($r \leftarrow r * c = (3^0) * (3^2) * (3^{16})$) and we have now computed our result. Note how we update $r$ exactly when we have computed one of the powers we were interested in. As foreshadowing for what comes next, it is helpful to think of the relationship between $x^n$, r and c, throughout the execution of the loop.

Now that we understand the algorithm, here is the code in Why3 (without any verification or specification hints):

```
let fast_exp (x: int) (n: int) : int
= let ref r = 1 in

  let ref c = x in
  let ref i = n in

  while i > 0 do
    if mod i 2 <> 0 then begin
        r <- r * c;
        i <- i - 1
    end;
    c <- c * c;
    i <- div i 2;
  done;
  r
```

Since we want to prove the function calculates exponents, lets define a suitable functional specification:

```
let rec function exp (x: int) (n: int) : int
  requires { n >= 0 }
  variant  { n }
= if n = 0 then 1
  else x * exp x (n-1)
```

and so we have the following pre- and post-conditions for `fast_exp`:

```
let fast_exp (x: int) (n: int) : int
  requires { n >= 0 }
  ensures  { result = exp x n }
```

Now comes the tricky part, the loop invariant: at the end of the loop, we want `result = exp x n` to hold, but this is not good enough as a loop invariant. The trick is realizing that we are building up to `exp x n` by roughly cutting `i` in half and squaring `c` with each iteration, but multiplying our "running total" `r` by `c` for odd exponents. With some thought, we can see that the property `r * exp c i = exp x n` gives us exactly what we need: first we can see that it holds at loop entry since `r=1`, `i=n` and `c=x`; second, we can see that when the loop terminates, since `i=0`, the post condition follows immediately from the loop invariant.

Let us now observe that the loop invariant is preserved by the loop body: we start by assuming `r * exp c i = exp x n` (or in a more readable form, $r \times c^i = x^n$) holds and must show that the same property holds after all the updates to `r`, `i` and `c`. Lets assume that `i` is odd and update the assertion we need to prove accordingly, due to the assignments in the conditional and the update to `c` and `i` afterwards. We end up with having to prove: `(r*c) * exp (c*c) ((i-1)/2) = exp x n`, or in more readable form $r \times c \times (c^2)^{(i-1)/2}$. Now remember Property 1 from the previous page: for odd `i` we have that $c^i = c \times (c^2)^{(i-1)/2}$, and so what we need to prove can be further simplified to $r \times c^i$, which equals $x^n$ by the loop invariant we started with! On the other hand, If `i` is even, our variable assignments are such that we end up with having to prove: `r * exp (c*c) (i/2) = exp x n`, or $r \times (c^2)^{i/2} = x^n$, which by Property 1 is the same as $r \times c^i = x^n$, which again is exactly the loop invariant we started out with! And we are done!

Actually, we are *almost* done. The reasoning we just did is too clever for Why3. Specifically, Property 1 is something that Why3 doesn't know is true, and so we must help the verification process with a lemma, which we prove by induction on $n$, performing case analysis on whether $n$ is odd or even and appealing to the appropriate inductive hypothesis (the conditions $n > 0$ and $n > 1$ are necessary to ensure that $n - 2 \geq 0$ in both cases):

```
let rec lemma exp_lemma (x n: int)
  requires { n >= 0 }
  variant { n }
  ensures { mod n 2 =  0 -> exp x n = exp (x * x) (div n 2) }
  ensures { mod n 2 <> 0 -> exp x n = x * exp (x * x) (div (n-1) 2) }
= if mod n 2 =  0 && n > 1 then exp_lemma x (n - 2);
  if mod n 2 <> 0 && n > 1 then exp_lemma x (n - 2)
```

If the above reasoning seems strange, try to prove the property by hand and it will become clearer (noting that appeals to the induction hypothesis map to recursive calls in Why3).

This extended example may seem like a lot: the algorithm is probably something you've not seen before and understanding how and why it works is not trivial, and we can only prove the implementation correct if we actually understand how it works! Not only that, we had to convince Why3 of a mathematical property, which ultimately requires understanding how to prove it inductively. Indeed, there was some work involved in these last 3 pages, but the hardest part is really producing the right invariant, which ultimately arises from understanding the relationship between the three variables and our intended result throughout the iterations of the loop. The rest is just a bit of math.

If you think this is completely out of reach, think again: this was actually the first handout from the 2020/2021 edition of this course, and you will be happy to know that over 70% of all submissions were essentially completely correct.

## 4 Algorithmic Approach to Verification

The inference system defined by the rules of Hoare logic is not *syntax directed*, that is, it is not possible to directly implement the rules as if they were an algorithm. While the rule for sequencing, when read bottom-up, requires inventing the intermediate assertion $B$, the real culprit is the rule of consequence, which can be applied at any point during a Hoare logic proof, for any program fragment.

However, it is possible to implement reasoners that do so-called forward or backward reasoning with Hoare logic rules, either starting from the preconditions and computing the most general post-condition, or starting from a post-condition and determining the weakest pre-condition that supports it. Dijkstra [Dij75] proposed such a strategy based on backward reasoning that produces proof obligations from an input Hoare triple. Proof obligations are assertions that must be proven correct for its source Hoare triple to be valid.

Dijkstra's approach defines the algorithm called "weakest pre-condition", which is inductively (recursively) defined on the possible cases of the program fragment given as input. It also takes as input the post-condition for the program fragment. It eliminates the need for the use of the non-algorithmic rule of Consequence, making the reasoning fully determined by the program fragment under consideration. The output of such function is an assertion that is the weakest pre-condition necessary to support the given post-condition.

$$\mathrm{WP}(e, Q)$$

So, if one starts with a Hoare Triple

$$\{A\}\ \mathsf{P}\ \{B\}$$

We can prove it valid if $A \implies \mathrm{WP}(\mathsf{P}, B)$. This implication is the generated proof obligation.

The original algorithm is defined for a more general programming language, the so-called language of guarded commands. The following cases, defined for our language of interest, are very similar to the rules of Hoare logic, but they crucially will not need to use the rule of (Consequence) to build the proof obligation.

**Skip.** The case for `skip` does not add any information to the post-condition.

$$\mathrm{WP}(\texttt{skip}, Q) \equiv Q$$

**Sequencing.** The case for the sequencing of statements computes the intermediate assertion as the weakest pre-condition of the second statement. Which, unlike in the sequence rule of *Hoare Logic*, is deterministic. This means that we don't need to "guess" any intermediate assertion.

$$\mathrm{WP}(e_1\ ;\ e_2, Q) \equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q))$$

**Assignment.** The case for assignment implements the backward reasoning of Hoare's formulation:

$$\mathrm{WP}(x := t, Q) \equiv Q[x \mapsto t\,]$$

**Conditional.** The common post-condition B is used as input in computing the weakest pre-condition of both sub-statements, just as in the Hoare logic rule. The rule of Consequence is embedded in the reasoning by using two implications that must hold as pre-condition:

$$\mathrm{WP}(\texttt{if t then } e_1 \texttt{ else } e_2, Q) \equiv \ (t \implies \mathrm{WP}(e_1, Q)) \wedge (\neg t \implies \mathrm{WP}(e_2, Q))$$

**Loops.** In loops, we have a loop invariant $(J)$ as input and determine three proof obligations as the computed pre-conditions, as follows: the loop invariant must be valid before the loop (as its pre-condition); the loop invariant must support the body of the loop together with the guard of the loop; and, the loop invariant must support the post-condition together with the negated loop guard.

$$
\begin{array}{ll}
\mathrm{WP}(\texttt{while t do } e\,, Q) \equiv & \\
\quad \exists J : \texttt{Prop}. & \text{some } \textit{invariant property } J \\
\quad\quad J \wedge & \quad \text{that holds at the loop entry} \\
\quad\quad \forall x_1 \dots x_k. & \quad \text{and is preserved} \\
\quad\quad\quad (J \wedge \phantom{\neg} t \rightarrow \mathrm{WP}(e, J)) \wedge & \quad\quad \text{after a single iteration,} \\
\quad\quad\quad (J \wedge \neg t \rightarrow Q) & \text{is strong enough to prove } Q
\end{array}
$$

We cannot know the values of modified references after $n$ iterations

- therefore, we prove preservation and the post for arbitrary values
- invariant must provide the needed information about the state

Finding an appropriate invariant is *difficult* in the general case

- this is equivalent to constructing a proof of $Q$ by induction

We can ease the task of automated tools by providing annotations:

$$
\begin{array}{ll}
\mathrm{WP}(\texttt{while t invariant } J \texttt{ do } e \texttt{ done}, Q) \equiv & \text{the given invariant } J \\
\quad J \wedge & \text{holds at the loop entry,} \\
\quad \forall x_1 \dots x_k. & \text{is preserved after} \\
\quad\quad (J \wedge \phantom{\neg} t \rightarrow \mathrm{WP}(e, J)) \wedge & \quad \text{a single iteration,} \\
\quad\quad (J \wedge \neg t \rightarrow Q) & \text{and suffices to prove } Q
\end{array}
$$

$x_1 \dots x_k$ references modified in $e$

For our purposes, it is more than enough to use a simplified version for the weakest pre-condition of loops:

$$
\begin{array}{lll}
\mathrm{WP}(\texttt{while t invariant } J \texttt{ do } e \texttt{ done}, Q) & \equiv & J \wedge \\
& & (J \wedge t \implies \mathrm{WP}(e, J)) \wedge \\
& & (J \wedge \neg t \implies Q)
\end{array}
$$

## 4.1 Example

Consider the program $P$ below:

$$
\begin{array}{ll}
P \triangleq & i := 0; \\
& s := 0; \\
& \texttt{while } i < n \texttt{ do } \{ \\
& \quad i := i + 1 \\
& \quad s := s + i \\
& \}
\end{array}
$$

From the Hoare triple $\{0 \leq n\}$ P $\{s = \sum_0^n j\}$, we can use the weakest pre-condition algorithm to generate proof obligations. We will also use as input the loop invariant $I = 0 \leq i \leq n \wedge s = \sum_0^i j$, which states that "up to iteration $i$", the value of $s$ contains the sum of all numbers from $0$ to $i$. The first step is to break the program into a sequence of two assignments and a while loop, and apply the case for sequencing (twice):

$$\begin{aligned}&\mathrm{WP}(P, s = \textstyle\sum_0^n j) = \mathrm{WP}(i := 0, \mathrm{WP}(s := 0, \mathrm{WP}(P', s = \sum_0^n j)))\\ &\quad \text{with } P' = \texttt{while} i < n ...\end{aligned}$$

The next step is to use the post-condition and invariant and compute the weakest pre-condition for the while loop:

$$\begin{aligned}&\mathrm{WP}(P', s = \textstyle\sum_0^n j) = I \wedge ((i < n \wedge I) \implies \mathrm{WP}(Q, I)) \wedge (i \geq n \wedge I \implies s = \sum_0^n j)\\ &\quad \text{with } Q = (i := i + 1 \ ; \ s := s + i)\\ &\quad \text{and } I = (0 \leq i \leq n \wedge s = \textstyle\sum_0^i j)\end{aligned}$$

Now, we compute the weakest pre-condition of the loop body in the expression above ($\mathrm{WP}(Q, I)$) in two steps:

$$\begin{aligned}&\mathrm{WP}(Q, I) = \mathrm{WP}(i := i + 1, \mathrm{WP}(s := s + i, I))\\ &\mathrm{WP}(s := s + i, I = 0 \leq i \leq n \wedge s + i = \textstyle\sum_0^i j)\\ &\mathrm{WP}(i := i + 1, 0 \leq i \leq n \wedge s + i = \textstyle\sum_0^i j) = 0 \leq i + 1 \leq n \wedge s + i = \sum_0^{i+1} j\end{aligned}$$

Finally, we can simplify the pre-condition (which is usually only done in the end),

$$\begin{aligned}&\mathrm{WP}(P', s = \textstyle\sum_0^n j)\\ &= 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\ &\wedge\ i < n \wedge 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j \implies 0 \leq i + 1 \leq n \wedge s + i = \sum_0^{i+1} j\\ &\wedge\ i \geq n \wedge 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j \implies s = \sum_0^n j\\ &= 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\ &\wedge\ 0 \leq i < n \wedge s = \textstyle\sum_0^i j \implies 0 \leq i + 1 \leq n \wedge s + i = \sum_0^{i+1} j\\ &\wedge\ i = n \wedge s = \textstyle\sum_0^i j \implies s = \sum_0^n j\\ &= 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\ &\wedge\ \texttt{true}\\ &\wedge\ \texttt{true}\\ &= 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\end{aligned}$$

To apply this pre-condition as post-condition of the first step

$$\mathrm{WP}(P, s = \sum_0^n j) = \mathrm{WP}(i := 0, \mathrm{WP}(s := 0, 0 \leq i \leq n \wedge s = \sum_0^i j))$$

By applying the case for assignment twice we obtain.

$$\mathrm{WP}(\mathsf{P}, \mathsf{s} = \sum_{0}^{n} \mathsf{j}) = 0 \leq 0 \leq n \wedge 0 = \sum_{0}^{0} \mathsf{j} = 0 \leq n$$

Which is exactly the pre-condition of the initial triple. The resulting proof obligation is the trivial expression $0 \leq n \implies 0 \leq n$.

## 5 Total correctness – Termination

To show total correctness of a program fragment with a while loop, we also need to prove termination. Like in the case of loop invariants, we need to have as input to the verification reasoning an integer function ($\mathsf{T}$), called the loop *variant*, defined on the state of the program. The case of *wp* using the variant function can then be defined as follows

$$\mathrm{WP}(\texttt{while E do P}, \mathsf{B}) \triangleq \forall_{\mathsf{V}}. \begin{cases} \mathrm{I} \\ \wedge (\mathsf{E} \wedge \mathrm{I} \implies \mathsf{T} > 0) \\ \wedge ((\mathsf{E} \wedge \mathrm{I} \wedge \mathsf{T} = \mathsf{V}) \implies \mathrm{WP}(\mathsf{P}, \mathrm{I} \wedge \mathsf{T} < \mathsf{V})) \\ \wedge (\neg \mathsf{E} \wedge \mathrm{I} \implies \mathsf{B}) \end{cases}$$

Notice the quantification over an integer value $\mathsf{V}$. The first case of the pre-condition refers to the loop invariant (I). The second case states that the loop variant must be strictly positive before every iteration, which means that the loop invariant and the guard must imply that. The third case states that besides supporting the loop invariant as post condition, the loop invariant and the guard must state that the the value of the loop variant decreases in one iteration. Finally, the negated guard and the loop invariant must imply the post-condition of the loop.

This variant function is the expression that the Why3 verifier expects in the **`variant`** clause in loops and recursive definitions.

### 5.1 Example

Using the same example as above, we can see that the first step is the same,

$$\mathrm{WP}(\mathsf{P}, \mathsf{s} = \textstyle\sum_0^n \mathsf{j}) = \mathrm{WP}(i := 0, \mathrm{WP}(\mathsf{s} := 0, \mathrm{WP}(\mathsf{P}', \mathsf{s} = \textstyle\sum_0^n \mathsf{j})))$$
with $\mathsf{P}' = \texttt{while } i < n \texttt{ do ...}$

And that the next step is to analyze the while loop given a loop invariant and a loop variant function.

$$\begin{aligned} \mathrm{WP}(\mathsf{P}', \mathsf{s} = \textstyle\sum_0^n \mathsf{j}) = \forall_{\mathsf{V}}.\mathrm{I} &\wedge (i < n \wedge \mathrm{I} \implies \mathsf{T}) \\ &\wedge (i < n \wedge \mathrm{I} \wedge \mathsf{T} = \mathsf{V} \implies \mathrm{WP}(\mathsf{Q}, \mathrm{I} \wedge \mathsf{T} < \mathsf{V})) \\ &\wedge (i \geq n \wedge \mathrm{I} \implies \mathsf{s} = \textstyle\sum_0^n \mathsf{j}) \end{aligned}$$
with $\mathsf{Q} = (i := i + 1 \;;\; \mathsf{s} := \mathsf{s} + i)$
and $\mathrm{I} = (0 \leq i \leq n \wedge \mathsf{s} = \sum_0^i \mathsf{j})$
and $\mathsf{T} = n - i$

The next step is to compute the weakest pre-condition of the loop's body in the expression above ($\mathrm{WP}(Q, I \wedge T < V)$), including the loop variant, in two steps

$$\mathrm{WP}(Q, I \wedge T < V) = \mathrm{WP}(i := i+1, \mathrm{WP}(s := s+i, I \wedge T < V))$$
$$\mathrm{WP}(s := s+i, I \wedge T < V) = 0 \leq i \leq n \wedge s+i = \textstyle\sum_0^i j \wedge n-i < V$$
$$\mathrm{WP}(i := i+1, 0 \leq i \leq n \wedge s+i = \textstyle\sum_0^i j \wedge n-i < V) = 0 \leq i+1 \leq n \wedge s+i = \textstyle\sum_0^{i+1} j \wedge n-(i+1) < V$$

Finally, we can simplify the pre-condition so far (which usually is only done in the end),

$$\begin{aligned}
&\mathrm{WP}(P', s = \textstyle\sum_0^n j)\\
&= \forall_V.\ 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\
&\wedge\ i < n \wedge 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j \wedge n-i = V \implies 0 \leq i+1 \leq n \wedge s+i = \textstyle\sum_0^{i+1} j \wedge n-(i+1) < V\\
&\wedge\ i \geq n \wedge 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j \implies s = \textstyle\sum_0^n j\\
&= \forall_V.\ 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\
&\wedge\ 0 \leq i < n \wedge s = \textstyle\sum_0^i j \wedge n-i = V \implies 0 \leq i+1 \leq n \wedge s+i = \textstyle\sum_0^{i+1} j \wedge n-(i+1) < V\\
&\wedge\ i = n \wedge s = \textstyle\sum_0^i j \implies s = \textstyle\sum_0^n j\\
&= \forall_V.\ 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\
&\wedge\ 0 \leq i < n \wedge s = \textstyle\sum_0^i j \wedge n = V+i \implies 0 \leq i+1 \leq n \wedge s+i = i + \textstyle\sum_0^i j = \textstyle\sum_0^{i+1} j \wedge V-1 < V\\
&\wedge\ \texttt{true}\\
&= \forall_V.\ 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j\\
&\wedge\ \texttt{true}\\
&\wedge\ \texttt{true}\\
&= 0 \leq i \leq n \wedge s = \textstyle\sum_0^i j
\end{aligned}$$

The remaining reasoning follows as previously, which establishes that the fragment will terminate for any value of $n$ which is greater or equal than $0$.

## References

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975.