Proof Of Correctness (Sections 1.6, 2.3)

1.1. Introduction

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

Donald E. Knuth

How to check if a program is correct?
Why checking if a program is correct?
What is a correct program?

Définition. Specification: precise description of how a program should behave.
A program is correct, if it behaves in accordance with its specifications.

How to check if a program is correct?
- Testing
  - Designing good test data sets
    You can prove there is a bug. No proof of no bug!
- Proof of Correctness

1.2. Correctness

1.2.1. Assertions.

... 
  x=sqrt(r); // r should be >= 0!
  x=a[i];   // i should be in the correct range
...

Définition. Assertion: condition on the variables of a program that should be verified at some given step, if the program is running correctly.

Using assert in C/C++:

```c
#include <assert.h>
...
assert(r>=0);
x=sqrt(r);
assert((0<=i) && (i<MAX));
x=a[i];
...
```

Each assertion is automatically checked:
- If the assertion is not verified, the program stops immediately with a message like:

  Assertion failed, line 35 of file foo.c

- That’s more informative for the user than a segmentation fault at the following line.
- This can be deactivated by adding a #define NDEBUG preprocessor directive.

With C++, java and other languages, you can use exceptions instead of assert:
- More informative error messages
- Error recovery mechanisms

Static type checking also makes some kind of assertion checking.

One way or the other, check assertions in your programs.

### 1.2.2. Specification of a function:

```c
int factorial(int n) {
    // Preconditions : n is a non-negative integer
    // Postcondition : returns n!
    ...
    int factorial(int n) {
        assert(n>=0);
        if (n==0) return 1;
        return n*factorial(n-1);
    }
}
```

The specification of a program can be formalized as follow:
- $P$ : program
- $X$ : input
- $P(X)$ : output
- $Q$ : precondition : predicate $Q(X)$ on the input
- $R$ : postcondition : predicate $R(X, P(X))$ on the input and the output

**Exemple.** Program to compute of a square root

- $P : y := \sqrt{x}$;
- $X : x$
- $P(x) : y$
- $Q(x) : x \geq 0$
- $R(x, y) : y^2 = x$

**Définition.** $P$ is correct if $(\forall X) \ Q(X) \rightarrow R(X, P(X))$.

A Hoare triple is just a short hand notation : $\{Q\} \ P \ \{R\}$

**Exemple.** $\{x \geq 0\} \ y := \sqrt{x}; \ \{y^2 = x\}$

### 1.3. Proof of Correctness

#### 1.3.1. Divide and Conquer.

Let $P$ be a big program:
To prove \( \{ Q \} \ P \{ R \} \), we break \( P \) into elementary steps, and insert assertions that describe the state of the program at each step:

\[
\begin{align*}
&\{ Q \} \\
&s_0 \\
&s_1 \\
&\{ R_1 \} \\
&s_2 \\
&\{ R_1 \} \\
&s_3 \\
&\vdots \\
&s_{n-1} \\
&\{ R \}
\end{align*}
\]

\( P \) is correct, if each step is correct, i.e. the following Hoare triples hold:

- \( \{ Q \} s_0 \{ R_1 \} \),
- \( \{ R_1 \} s_1 \{ R_2 \} \),
- \( \ldots \)
- \( \{ R_{n-1} \} s_{n-1} \{ R \} \)

To prove that the elementary steps are correct, we will use some syntactic rules, exactly as we did in formal logic.

### 1.3.2. Assignment rule

Consider the following Hoare triple: \( \{ Q \} \ x := e \ { R \} \)

**Théorème.** If from \( Q \) you can derive \( R \) with \( x \) substituted everywhere by \( e \), then the Hoare triple is valid.

**Exemple.** \( \{ x = 2 \} \ y := x + 1 \ { y = 3 \} \)

When substituting, \( y = 3 \) becomes \( x + 1 = 3 \). From \( x = 2 \), you can deduce \( x + 1 = 3 \).

So this Hoare triple holds.

**Exemple.** \( \{ x > 0 \} \ x := x + 1 \ { x > 1 \} \)

Here it can be confusing.

The same name \( x \) stands for both the value of \( x \) before and after the assignments.

If you get confused, just rename the variable:

\( \{ x_0 > 0 \} \ x_1 := x_0 + 1 \ { x_1 > 1 \} \)

When substituting, \( x_1 > 1 \) becomes \( x_0 + 1 > 1 \), which you can deduce from \( x_0 > 0 \).
1.3.3. **Conditional Rule.** Consider the following Hoare triple:

\[
\{ Q \} \\
\text{if condition } B \text{ then} \\
P_1 \\
\text{else} \\
P_2 \\
\text{end if} \\
\{ R \}
\]

**Theorem.** If the Hoare triples \( \{ Q \land B \} \quad P_1 \{ R \} \) and \( \{ Q \land B' \} \quad P_2 \{ R \} \) hold, then the Hoare triple above holds.

**Example.** [7, Exercise 11 p. 78]

1.3.4. **Loop Rule.** Consider the following Hoare triple:

\[
\{ Q \} \\
\text{while condition } B \text{ do} \\
P \\
\text{end while} \\
\{ R \}
\]

\( \{ Q \} \) describes the state of the program before the loop.

We need to have a predicate which describes the state of the program DURING the loop:

The loop invariant \( \{ S \} \)

Most of the time, \( \{ Q \} \) will do the job, but not always.

**Theorem.** If the Hoare triple \( \{ S \land B \} \quad P \quad \{ S \} \) holds, then the following Hoare triple will hold:

\[
\{ S \} \\
\text{while condition } B \text{ do} \\
P \\
\text{end while} \\
\{ S \land B' \}
\]

1.3.5. **A simple example.** Consider this little program:

```plaintext
Product(x,y)  // Precondition : x and y are non-negative integers  // Postcondition : returns the product of x and y
begin
  i := 0;
  j := 0;
  while ( (i=x)' ) do
    j = j + y;
    i := i + 1;
  end while
  // Assertion : \{ j = xy \}
  return(j);
end;
```
It’s pretty clear that this algorithm it’s correct. Let’s check it anyway to see how it works.

Démonstration. We need a loop invariant, which will describe the state of the program after \( k \) iterations. A reasonable guess is \( S : j = iy \)

Let \( i_k, j_k \) and \( S_k \) be the value of \( i, j \) and \( S \) after \( k \) iterations.

Base case : using the assignment rule, \( i_0 = 0 \) and \( j_0 = 0 \) so \( S_0 \) holds.

Induction step :
Assume the invariant holds after \( k - 1 \) iterations : \( j_{k-1} = i_{k-1}y \)
We need to prove that the invariant is preserved by the \( k \) th iteration. Otherwise said, we have to check that the following Hoare triple holds :

\[
\{ S_{k-1} \land (i = x) \}' \}
\]
\[
j_k := j_{k-1} + y ;
\]
\[
i_k := i_{k-1} + 1 ;
\]
\[
\{ S_k \}
\]

Let’s applying the assignment rule.
By substituting \( i_k \) and \( j_k \) by the expressions they have been assigned with, we get :

\[
S_k \iff j_k = i_k y
\]
\[
\iff j_{k-1} + y = (i_{k-1} + 1)y \quad \text{(By the substitution rule)}
\]
\[
\iff i_{k-1} y + y = (i_{k-1} + 1)y \quad \text{(By } S_{k-1})
\]
\[
\iff i_{k-1} y + y = i_{k-1} y + y
\]

So, \( S_k \) indeed holds.

By induction, after any number \( k \) of iterations, \( S_k \) still hold.

At the end, both \( S_k \) and \( i = x \) hold, so, as expected : \( j = iy = xy \)

Remarque. We have proved that after the end of the execution the result is correct. But what if the program does not terminate and loop forever? This is not usually consider a proper behavior.

We only have spoken about partial-correctness.

A full proof of correctness needs also a proof of termination!

1.3.6. A more sophisticated example : The Euclidean algorithm.

\texttt{GCD}(a,b)

Begin

// Precondition : a and b are non-negative integers
// with a\textgreater{}=b.

// Postcondition :
// returns \text{gcd}(a,b), the greater common divisor of
// a and b.

Local variables : i, j, q, r

i :=a;

j :=b;

while \( j > 0 \) do

// Assertion :
// r is the rest of the integer division i/j
r := i mod j;
i := j;
j := r;
end while
// Assertion : i is the gcd of a and b.
return(i);
end;

Exercise 1. Use the Euclidean algorithm to compute:
GCD(8,8), GCD(14,4), GCD(31,17).

Here the proof of correctness of the algorithm is non-trivial.

Démonstration. Let \(i_k\) and \(j_k\) be the value of \(i\) and \(j\) after \(k\) iterations.
We need to find an invariant which describes the state of the program after each iteration.

Take \(S_k : \gcd(i_k, j_k) = \gcd(a, b)\).

1. Base case :
   Before the loop, \(i_0 = a\) and \(j_0 = b\).
   So the invariant \(S_0\) holds : \(\gcd(i_0, j_0) = \gcd(a, b)\);

2. Induction step :
   Assume \(S_{k-1}\) holds, that is \(\gcd(i_{k-1}, j_{k-1}) = \gcd(a, b)\).
   We just need to prove that \(\gcd(i_k, j_k) = \gcd(i_{k-1}, j_{k-1})\).
   By the assignment rule, we have :
   - \(r\) is the rest of the division of \(i_{k-1}\) by \(j_{k-1}\),
   - \(i_k = j_{k-1}\),
   - \(j_k = r\).
   So, by the definition of the integer division, we have :
   \[i_{k-1} = qj_{k-1} + j_k\]
   for some integer \(q\).

   (a) Assume \(x\) divides both \(i_k\) and \(j_k\).
   Then, of course \(j_{k-1}\), since \(j_{k-1} = i_k\).
   By the equation above, \(x\) also divides \(i_{k-1}\).

   (b) Assume \(x\) divides both \(i_{k-1}\) and \(j_{k-1}\).
   Then of course \(x\) divides \(i_k\).
   Once again, using the equation above, \(x\) also divides \(j_k\).

   Therefore, \(\gcd(i_k, j_k) = \gcd(i_{k-1}, j_{k-1}) = \gcd(a, b)\), as wanted.

3. At the end :
   By induction, the loop invariant \(S\) still holds : \(\gcd(i, j) = \gcd(a, b)\)
   Moreover, the loop condition failed : \(j = 0\).
   So, \(\gcd(a, b) = \gcd(i, j) = \gcd(i, 0) = i\), as expected.

\[\square\]
1.4. Conclusion

1.4.1. A note on automatic program proving:
- There is no mechanical way to decide even if a program terminates or not.
- Traditional imperative languages:
  - Automatic proving is very difficult because of side effects
  - Side effects need to be considered as some form of output
- Functional languages (lisp, scheme, ML, caml):
  - Designed to make it easier to prove correctness
  - Fun to program. Try them!
- Two difficulties:
  - Finding the correct assertions
  - Checking that the implications hold
- Mixed approach:
  - The programmer gives the main assertions
  - The prover derives the other assertions

1.4.2. Check assertions in your programs!
- Use static type checking / assert / exceptions
- When debugging, you can ask for all assertions to be checked
- Documentation
- Help for the proof