Construction and Verification of Software

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

September 29, 2025

Lecture 4

based on previous editions by João Seco, Luís Caires, and Bernardo Toninho

also based on lectures by Andrei Paskevich and Claude Marché

Administrativia

The first handout is due for this Saturday, at 23:59.

Delivery method: send an email to

`mjp.pereira@fct.unl.pt`

with subject `[CVS-2025] Handout 1`.

You should attach to your email

- the Rocq file with your solution (mandatory)
- a text file explaining your decisions and difficulties (optional)

The email should contain information about the team members

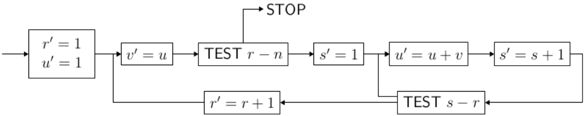
- numbers
- names

Verification of Imperative Programs

1. An historic perspective
2. The While language
3. Introduction to Hoare Logic
4. Introduction to the Why3 tool

A Bit of History

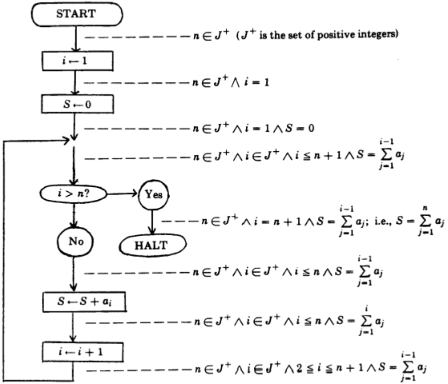
A. M. Turing. Checking a large routine. 1949.



What does this program do?

"How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows."

R. Floyd. Assigning meaning to programs. 1967.

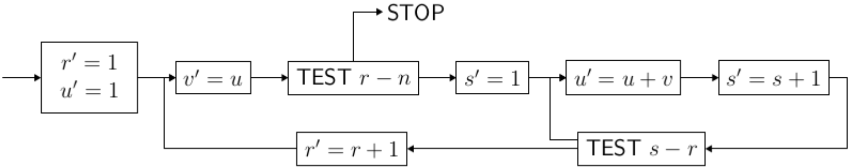


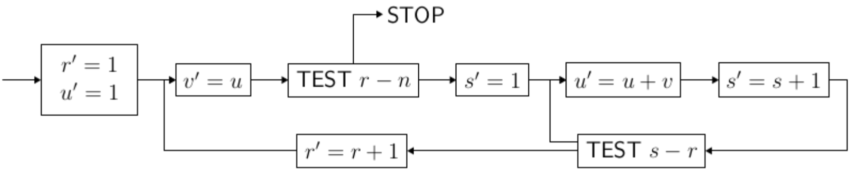
“Robert Floyd’s, "Assigning Meanings to Programs," opened the field of program verification. His basic idea was to attach so-called "tags" in the form of logical assertions to individual program statements or branches that would define the effects of the program based on a formal semantic definition of the programming language.”

C. A. R. Hoare.
An axiomatic basis for computer programming.
Commun. ACM, 1969

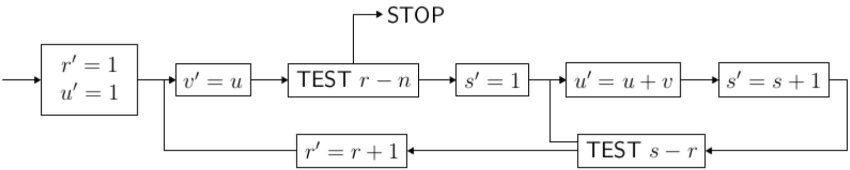
Proof of a program: FIND.
Commun. ACM, 1971

"Computer Programming is an exact science in that all the properties of a program and all consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."


STOP
$r' = 1$
$u' = 1$
$v' = u$
TEST $r - n$
$s' = 1$
$u' = u + v$
$s' = s + 1$
$r' = r + 1$
TEST $s - r$




```
u := 1
for r = 0 to n - 1 do
  v := u
  for s = 1 to r do
    u := u + v
```




```
precondition { n ≤ 0 }
u := 1
for r = 0 to n - 1 do
  v := u
  for s = 1 to r do
    u := u + v
postcondition { u = fact(n) }
```

Now what?

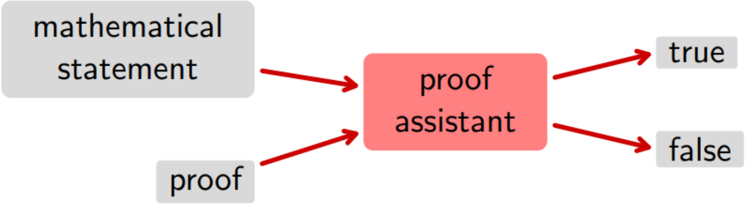
What do we do with this implementation and logical specification?

> *Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it.*

We could perform a manual proof (as Turing and Hoare did) but it is long, tedious, and error-prone.

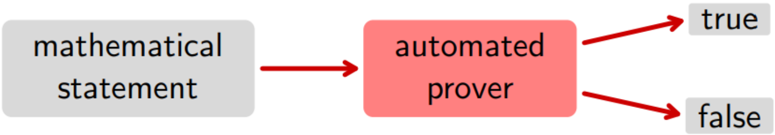
So we turn to tools that mechanize mathematical reasoning.

Build a proof and ask a proof assistant to check it.



Example: Coq, Isabelle, PVS, HOL Light

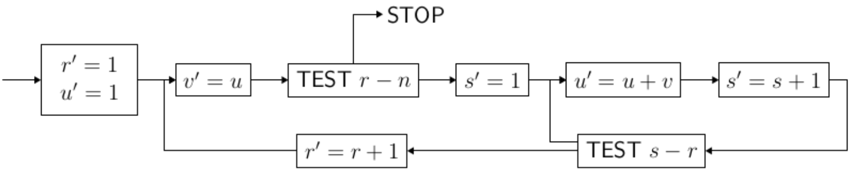
The dream:



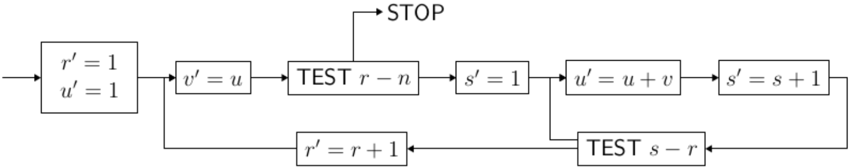
It is not possible to implement such a program
(Turing/Church, 1936, from Gödel)

Theorem: mathematicians will always have jobs!

Kurt Gödel




```
precondition { n ≤ 0 }
u := 1
for r = 0 to n - 1 do
  v := u
  for s = 1 to r do
    u := u + v
postcondition { u = fact(n) }
```



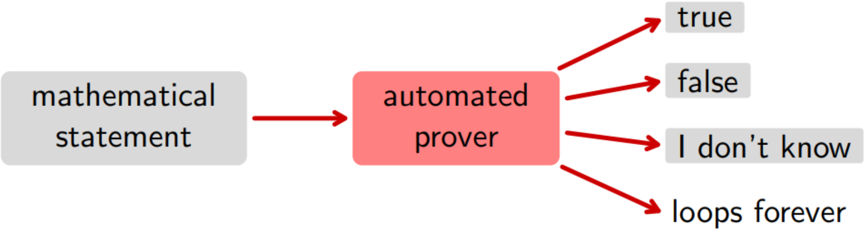

```
precondition { n ≤ 0 }
u := 1
for r = 0 to n - 1 do     invariant { u = fact(r) }
  v := u
  for s = 1 to r do     invariant { u = s × fact(r) }
    u := u + v
postcondition { u = fact(n) }
```

```
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

```
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
  (0 ≤ n - 1 →
    1 = fact(0) ∧
    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →
        (1 > r → u = fact(r + 1)) ∧
        (1 ≤ r →
          u = 1 * fact(r) ∧
          (∀ u1:int.
            (∀ s:int. 1 ≤ s ∧ s ≤ r → u1 = s * fact(r) →
              (∀ u2:int.
                 u2 = u1 + u → u2 = (s + 1) * fact(r))) ∧
              (u1 = (r + 1) * fact(r) → u1 = fact(r + 1))))) ∧
      (u = fact((n - 1) + 1) → u = fact(n))))
```

```
function fact(int) : int
axiom fact0: fact(0) = 1
```

```
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
```



Examples: Z3, CVC5, Alt-Ergo, Vampire, SPASS, etc.

Contract-based Verification

A prime on Hoare Logic

```
var sum := 1;
var count := 0;
while sum <= n {
  count := count + 1;
  sum := sum + 2 * count + 1;
}
return count;
```

What is the result of this expression for a given `n`?

Example — *ISQRT*

```
var sum := 1;
var count := 0;
while sum <= n {
  count := count + 1;
  sum := sum + 2 * count + 1;
}
return count;
```

What is the result of this expression for a given `n`?

Informal specification:

- at the end, `count` contains the truncated square root of `n`
- for instance, given $\mathtt{n} = 42$, the returned value is 6

A statement about program correctness:

$$\{P\}\ e\ \{Q\}$$

P	precondition property
e	expression
Q	postcondition property

What is the meaning of a Hoare triple?

$\{P\}\,e\,\{Q\}$ if we execute e in a state that satisfies P,
then the computation either diverges
or terminates in a state that satisfies Q

This is partial correctness: we say nothing about termination.

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x := x + 2\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while true do skip` $\{\boxed{\boxed{\texttt{false}}}\}$

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x := x + 2\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while true do skip` $\{\boxed{\boxed{\texttt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x := x + 2\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\mathtt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \mathtt{result} = 2w\}$
- $\{\mathtt{true}\}$ `while true do skip` $\{\boxed{\boxed{\mathtt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

In our square root example:

$$\{?\}\ \mathit{ISQRT}\ \{?\}$$

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x := x + 2\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while true do skip` $\{\boxed{\boxed{\texttt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

In our square root example:

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{?\}$$

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x := x + 2\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ **while** `true` **do** `skip` $\{\boxed{\boxed{\texttt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

In our square root example:

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{\texttt{result}^2 \leqslant n < (\texttt{result} + 1)^2\}$$

While – A Simple Imperative Programming Language

E	$::=$		Expressions
		num	Integer
	$\mid$	x	Variable
	$\mid$	$E + E \mid \ldots$	Integer operations
	$\mid$	$E < E \mid \ldots$	Relational operations
	$\mid$	E and $E \mid \ldots$	Boolean operations
P	$::=$		Programs
		skip	No operation
	$\mid$	$x := E$	Assignment
	$\mid$	$P;P$	Sequential Composition
	$\mid$	if E then P else P	Conditional
	$\mid$	while E do P	Iteration

An imperative program is a state transformer.
It transforms an initial state into a target state.

What is a program state? A mapping of state variables to values:

$$\sigma = \{x \mapsto 1; y \mapsto 2; w \mapsto 3\}$$

An imperative program transforms states into states

$$P \triangleq \texttt{x := y + x; w := w - x}$$

If P is executed in state σ it yields state σ' where

$$\sigma' = \{x \mapsto 3; y \mapsto 2; w \mapsto 0\}$$

We say that P transforms σ into σ'.

Initially (1970): axiomatic semantics of programs

Inference rules to construct valid triples:

$$\frac{}{\{P\}\,\texttt{skip}\,\{P\}}$$

$$\frac{}{\{P[x \mapsto t]\}\, x := t\, \{P\}}$$

$$\frac{\{P\}\, e_1\, \{Q\} \qquad \{Q\}\, e_2\, \{R\}}{\{P\}\, e_1\ ;\ e_2\, \{R\}}$$

Notation $P[x \mapsto t]$: replace in P every occurrence of x with t

Consequence rule:

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

Example: proof of $\{x = 1\}\, x := x + 2\, \{x = 3\}$

$$\cfrac{\models x = 1 \to x + 2 = 3 \qquad \cfrac{\begin{array}{c}(x = 3)[x \mapsto x + 2] \equiv x + 2 = 3 \\ \vdots\end{array}}{\{x + 2 = 3\}\, x := x + 2\, \{x = 3\}}}{\{x = 1\}\, x := x + 2\, \{x = 3\}}$$

Rules for `if` and `while`:

$$\frac{\{P \wedge t\}\, e_1\, \{Q\} \qquad \{P \wedge \neg t\}\, e_2\, \{Q\}}{\{P\}\, \texttt{if}\ t\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{Q\}}$$

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

Formula J is a loop invariant. (more on this later)

Finding a right invariant is a major difficulty.

A program proof in Hoare logic adds assertions between program statements, making sure that all Hoare triples are satisfied/valid.

Consider the following code snippet:

```
if (x > y) then
  z := x
else
  z := y
```

A Hoare Logic "proof" may look like

```
{ true }
if (x > y) then
  { (x > y) }
  z := x
  { (x > y) ∧ (z == x) }
else
  { (x <= y) }
  z := y
  { (x <= y) ∧ (z == y) }
{ (x > y) ∧ (z == x) || (x <= y) ∧ (z == y)}
{ z == max(x, y) }
```

Homework: use Hoare rules to check this derivation is valid.

Rules of Hoare Logic (general form)

The inference rules of Hoare logic are used to derive (valid) Hoare triples given some already derived Hoare triples.

$$\frac{\{A_1\}\,P_1\,\{B_1\} \qquad \{A_n\}\,P_n\,\{B_n\}}{\{A\}\,C(P_1,\ldots,P_n)\,\{B\}}$$

What is nice here:

- the program in the conclusion contains the subprograms $P_1,\ldots,P_n$ as components
- we derive properties of the composite from the properties of its parts (compositionality)
- pretty much the same as with a type system

E	$::=$		Expressions
	$\mid$	$\dots$	
S	$::=$		Statements
	$\mid$	$\dots$	
	$\mid$	$x := m(E_1, \dots, E_n)$	Call + Assignment
D	$::=$		Declarations
	$\mid$	$\texttt{method}\, m(x_1, \dots, x_n)\ \texttt{returns}\,(r)$ $\texttt{requires}\, \mathit{Pre}(x_1, \dots, x_n)$ $\texttt{ensures}\, \mathit{Post}(x_1, \dots, x_n, r)$ $\{S\}$	
P	$::=$		Program
	$\mid$	$\overline{D}$	

Declarations annotated with pre- and post- conditions.

A program P is a set of method declarations.

Each method declaration is validated as follows:

1. assume its pre-condition
2. prove its post-condition.

$$\frac{\{Pre(x_1,\ldots,x_n)\}\, S\, \{Post(x_1,\ldots,x_n,r)\}}{\begin{array}{l}\texttt{method}\, m(x_1,\ldots,x_n)\,\texttt{returns}\,(r)\\ \texttt{requires}\, Pre(x_1,\ldots,x_n)\\ \texttt{ensures}\, Post(x_1,\ldots,x_n,r)\ \{S\}\end{array}}$$

Method calls built into a form of assignment:

$$\texttt{method}\, m(x_1,\ldots,x_n)\, \texttt{returns}\, (r)$$
$$\texttt{requires}\, Pre(x_1,\ldots,x_n)$$
$$\texttt{ensures}\, Post(x_1,\ldots,x_n,r)\ \{S\}$$

$$\frac{A \Rightarrow Pre(E_1,\ldots,E_n) \qquad Post(E_1,\ldots,E_n,r) \Rightarrow B[x \mapsto r]}{\{A\}\, x := m(E_1,\ldots,E_n)\, \{B\}}$$

Each method call is validated as follows:

1. prove the (instantiated) pre-condition of m

$$A \implies Pre(E_1,\ldots,E_n)$$

2. assume the (instantiated) post-condition of m

$$Post(E_1,\ldots,E_n,r) \implies B[x \mapsto r]$$

Method calls built into a form of assignment:

$$\begin{array}{l}\texttt{method}\, m(x_1,\ldots,x_n)\,\texttt{returns}\,(r)\\ \texttt{requires}\, Pre(x_1,\ldots,x_n)\\ \texttt{ensures}\, Post(x_1,\ldots,x_n,r)\ \{S\}\end{array}$$

$$\frac{A \Rightarrow Pre(E_1,\ldots,E_n) \qquad Post(E_1,\ldots,E_n,r) \Rightarrow B[x \mapsto r]}{\{A\}\, x := m(E_1,\ldots,E_n)\, \{B\}}$$

Calls are opaque. We only know what is in the post-condition.

Verification with method calls is modular.

Derive

1. $$\{y > 0\}\, x := y\, \{x > 0 \wedge y == x\}$$

2. $$\{x == y\}\, x := 2 * x\, \{2 * y == x\}$$

3. $$\{P(y) \wedge Q(w)\}\, x := y; y := w; w := x\, \{P(w) \wedge Q(y)\}$$

Introduction to the Why3 tool

demo

Bibliography

- R. W. Floyd.
  Assigning Meanings to Programs.
  Proc. of Symposia in Applied Mathematics, 1967

- C. A. R. (Tony) Hoare.
  An Axiomatic Basis for Computer Programming.
  Communications of the ACM, 1969

- E. W. Dijkstra.
  A Discipline of Programming. Prentice Hall, 1976

- J. B. Almeida, M. J. Frade, J. S. Pinto, S. Melo de Sousa.
  Rigorous Software Development. Springer, 2011

- The Why3 Development Team.
  The Why3 Platform. Version 1.8, September 2025