5: The Hoare Logic

- a formal system to rigorously reason about the correctness of programs
- usage to prove correctness “by hand”
  - difficult and error prone
  - doable for small algorithms
  - unfeasible for large programs
- learning goals
  - learn the rules of Hoare Logic
  - apply them to tiny algorithms “by hand”
  - use a tool applying them for automatic verification ($\rightarrow$ section on SPARK)
- The goal behind the goals:
  - understand how tool-based program verification works (without knowing the HL, you could not reasonably use such tools).

History

Floyd, 1967: Usage of asserts as a tool for proofs of correctness

Hoare, 1969: Axiomatic definition of the program correctness via "Hoare-Triples":

{Precond.} Program(-fragment) {Postcond.}

Distinction between partial and total correctness

Dijkstra 1975: "Weakest precondition" as the foundation for automatically generated proofs – though not with using the computers available in 1975

partial and total correctness

- Consider a program S, and assume its precondition is satisfied. Notation as a “Hoare Triple”:

$$\{\text{Precondition}\}\ S\ \{\text{Postcondition}\}.$$

- S is **partially correct**, if, when it delivers a result, the result satisfies the postcondition. S may not always deliver a result.
- S is **totally correct**, if it s partially correct *and* it always terminates (without raising an exception – though Hoare didn’t consider exception in 1969).
- What S does if its precondition is not satisfied, doesn’t affect correctness.

Examples

Which of the examples is

(a) totally correct,

(b) partially but not totally correct, or

(c) even partially incorrect?

1. $\{\mathbf{X = 1}\}$ **null**; $\{\mathbf{X = 0}\}$.
2. $\{\mathbf{True}\}$ X := 1; $\{\mathbf{X = 0}\}$.
3. $\{\mathbf{X = 1}\}$ **null**; $\{\mathbf{X = 1}\}$.
4. $\{\mathbf{X = 1}\}$ **loop null**; **end loop**; $\{\mathbf{X = 0}\}$.
5. $\{\mathbf{True}\}$ X := 1; $\{\mathbf{X = 1}\}$
6. $\{\mathbf{False}\}$ X := 1; $\{\mathbf{X = 0}\}$

Mathematical Proofs

- New proofs are examined carefully when (and even before) they are published (“peer review”).
- Sometimes, people find flaws in proofs published many years before.

Proofs in Computer Science

- Proof of correctness for *Algorithms* and *Communication Protocols*: “peer review”, as in mathematics.
- Proofs of correctness for software (implementation of algorithms): no “peer review”, hardly any incentive.
- The proofs are often not more easy to understand than the source code.
- *If the computer can’t verify the proofs – who else will do?*

Correctness

$\{X \geq 1\}$
while X $>$ 1 **loop**
    if X **mod** 2 = 0 **then**
        X := X / 2;
    else
        X := 3$*$X $+$ 1;
    end if;
end loop;
$\{X = 1\}$

- Partial correctness: Yes We Can!
- Total correctness: Unsolved problem from mathematics.
- Note: Our “Integers” are mathematical (elements of $\mathbb{Z}$).

Surprise?

- Given integers X and Y, search for statement(-sequence) S with

$$\{\text{True}\}\ S\ \{\text{Y} = \max(\text{X},\text{Y})\}.$$

  Which of these statement(-sequence)s are correct?
- Y := X;
- X := Y;
- X := 0;
- **if** X $>$ Y **then** X := Y; **end if**;
- **if** X $>$ Y **then** Y := X; **end if**;
- X := 0; Y := 0;

Guess which solution is desired? ;-)

The (perhaps) correct Specification

- Consider

$$\{\mathbf{X}^{\text{old}} = \mathbf{X} \wedge \mathbf{Y}^{\text{old}} = \mathbf{Y}\}\ S\ \{\mathbf{Y} = \max(\mathbf{X}^{\text{old}}, \mathbf{Y}^{\text{old}})\}.$$

  with S="**if** X $>$ Y **then** Y := X; **end if**;"
- This is the excepted behaviour. Observe the "gnost variables", such as X^{old} and Y^{old} that represent the "old" values of X and Y.
- Also correct, S= "**if** X $>$ Y **then** Y := X; **else** X := Y **end if**;"
- The defense against changing X:

$$\{\mathbf{X}^{\text{old}} = \mathbf{X} \wedge \mathbf{Y}^{\text{old}} = \mathbf{Y}\}\ S\ \{\mathbf{X} = \mathbf{X}^{\text{old}} \wedge \mathbf{Y} = \max(\mathbf{X}^{\text{old}}, \mathbf{Y}^{\text{old}})\}.$$

- Classical Hoare logic requires to specify *all variables* which didn't change. This scales extremely bad for larger applications!
- Tools avoid that need – e.g., SPARK's data flow analysis.

Case study: Sort array $A[1..N]$

- 1st idea: define $\mathbf{S(A)} = \mathbf{A[1]} \leq \mathbf{A[2]} \leq \cdots \mathbf{A[N]}$.
  ($\mathbf{S(A)}$ =”A is sorted”).

$$\{\text{True}\}\ \text{Sort}(A);\ \{\mathbf{S(A)}\}$$

- This is too weak! (why?) What else do we need?
- 2nd idea: $\mathbf{P}(\mathbf{A}^{\text{old}}, \mathbf{A})$ =”A is a permutation of A^{old}”;

$$\{\text{True}\}\ \text{Sort}(A);\ \{\mathbf{S(A)} \wedge \mathbf{P}(\mathbf{A}^{\text{old}}, \mathbf{A})\}$$

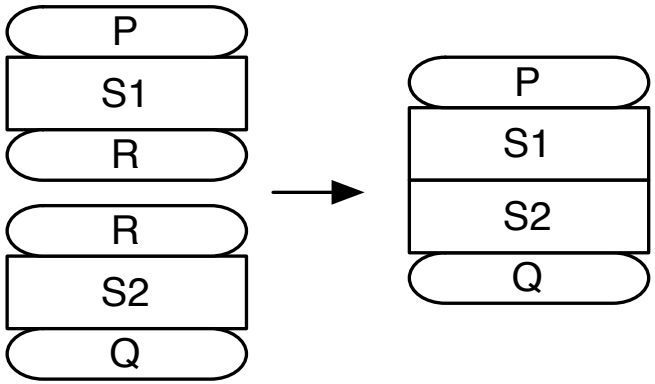
- This does what we want.
  But how shall we actually formalize $\mathbf{P}(\mathbf{A}^{\text{old}}, \mathbf{A})$?

5.1: Building Blocks and a Toy Language

- First order logic
- Two axioms and four other rules to define a “toy language”

  . . . but a real language can be derived from this
- Logical deduction (application of rules and axioms) to define new rules – and to prove them!

Rule of Composition



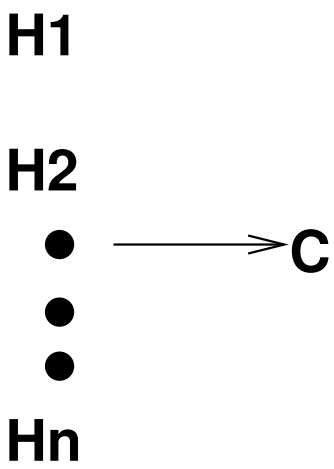
$$\frac{\{\mathbf{P}\}\ S_1\ \{\mathbf{R}\},\quad \{\mathbf{R}\}\ S_2\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ S_1;\ S_2;\ \{\mathbf{Q}\}}$$

Statements S_1 and S_2 can merge into $S_1; S_2$ if the postcondition of S_1 is the same as the precondition for S_2.

Example:

- S_1===$\{X \geq 1\}X := X + 1; \{X \geq 2\}$
- S_2===$\{X \geq 2\}X := X/2; \{X \geq 1\}$

How all rules look like, in general

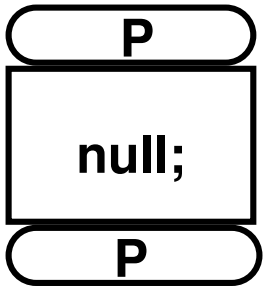


$$\frac{H_1, H_2, \ldots, H_n}{C}$$

- "Conclusion" C
- "Hypothesis" $H_1 \wedge H_2, \wedge \cdots, \wedge H_n$

"Axiom": A rule with $(H_1 \wedge H_2, \wedge \cdots, \wedge H_n) = \text{true}$.

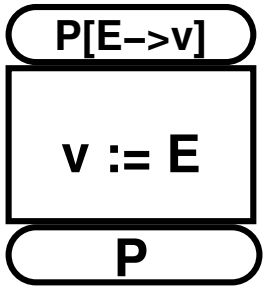
Rule/Axiom 1: Empty Statement



$$\frac{\text{True}}{\{\mathbf{P}\}\,null;\,\{\mathbf{P}\}}$$

The axiom ensures that the **null;** statement doesn't change the state of the program. Whatever holds true before **null;** also holds true afterwards, and vice versa.

Axiom 2: Assignment



$$\frac{\text{True}}{\{\mathbf{P}[\text{Expr} \rightarrow \mathbf{v}]\}\ v := \text{Expr};\ \{\mathbf{P}\}}$$

If P and Expr are expressions, "$P[\text{Expr} \rightarrow v]$" denotes the expression where all "free" occurrences of v in P are replaced by Expr.

Example:

$$\{\mathbf{2 * X = A}\} X := 2 * X; \{\mathbf{X = A}\}$$

If the equation $X = A$ shall hold *after* the assignment,
then the equation $2 * X = A$ must hold *before*.

P === $X = A$;

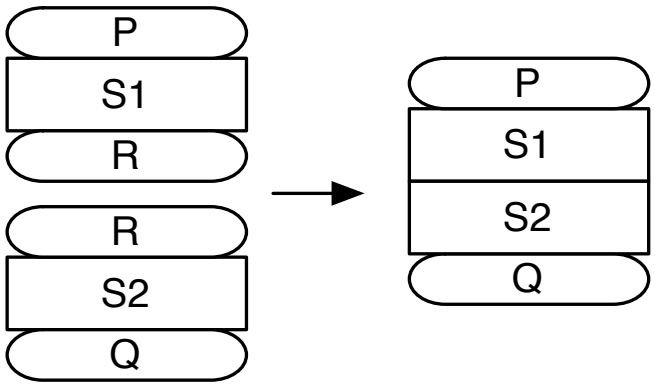
v === X

Expr === $2 * X$

$P[\text{Expr} \rightarrow v]$ === $2 * X = A$

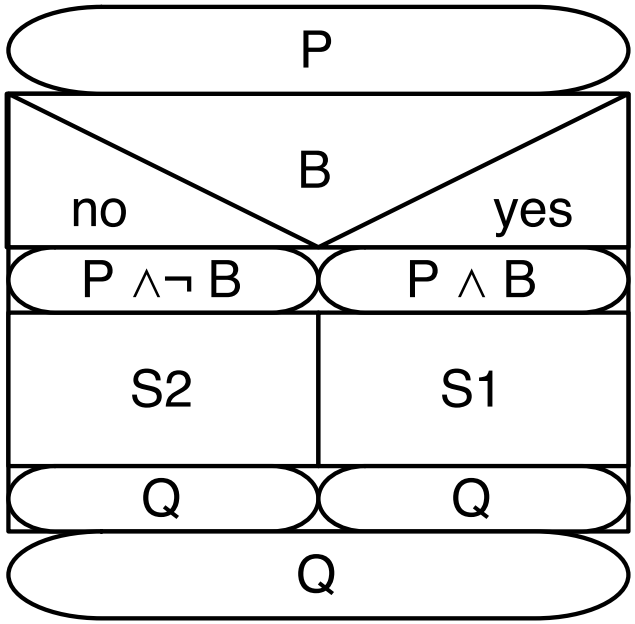
Rule 3: Composition

(We have seen that before)



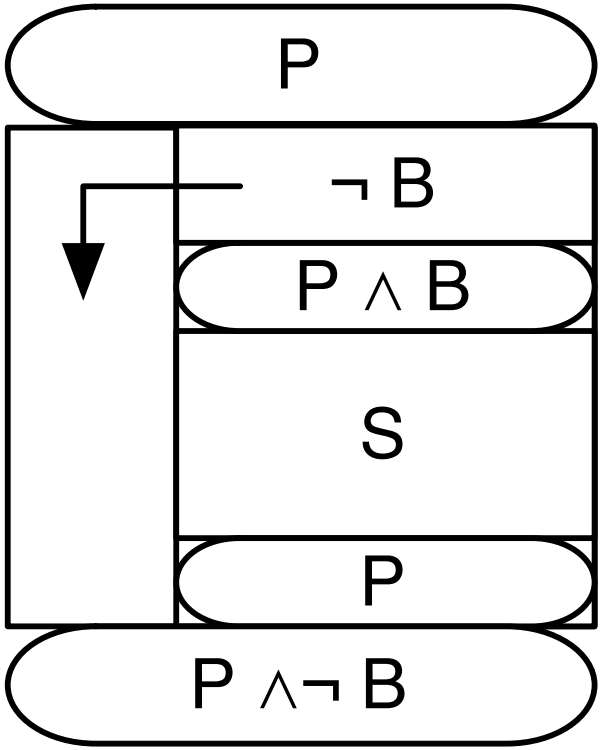
$$\frac{\{P\}\ S_1\ \{R\},\quad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2;\ \{Q\}}$$

Rule 4: IF-THEN-ELSE



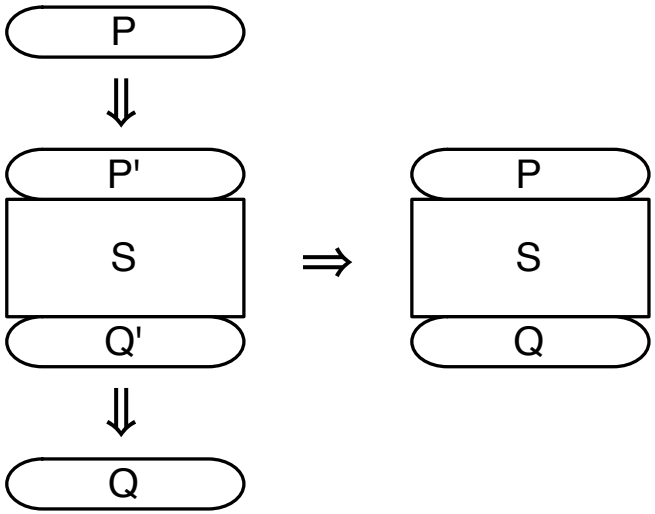
$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S_1\ \{\mathbf{Q}\},\quad \{\mathbf{P},\ \neg\mathbf{B}\}\ S_2\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ \textit{if}\ \mathbf{B}\ \textit{then}\ S_1;\ \textit{else}\ S_2;\ \textit{end if};\ \{\mathbf{Q}\}}$$

Rule 5: WHILE



$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S\ \{\mathbf{P}\}}{\{\mathbf{P}\}\ \textit{while}\ \mathbf{B}\ \textit{loop}\ S;\ \textit{end loop};\ \{\mathbf{P},\ \neg\mathbf{B}\}}$$

Rule 6: Conclusion



$$\frac{\mathbf{P} \Rightarrow \mathbf{P'}, \quad \{\mathbf{P'}\}\ S\ \{\mathbf{Q'}\}, \quad \mathbf{Q'} \Rightarrow \mathbf{Q}}{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}$$

Deriving new rules (1)

Conclusion:

$$\frac{\mathbf{P} \Rightarrow \mathbf{P}', \quad \{\mathbf{P}'\}\ S\ \{\mathbf{Q}'\}, \quad \mathbf{Q}' \Rightarrow \mathbf{Q}}{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}$$

Abstract Logic:

$\mathbf{P} \Rightarrow \mathbf{P}$ and $\mathbf{Q}' \Rightarrow \mathbf{Q}$.

Postcondition Weakening

$$\frac{\{\mathbf{P}\}\ S\ \{\mathbf{Q}'\}, \quad \mathbf{Q}' \Rightarrow \mathbf{Q}}{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}$$

Precondition Strengthening

$$\frac{\mathbf{P} \Rightarrow \mathbf{P}', \quad \{\mathbf{P}'\}\ S\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}$$

Deriving new rules (2)

Rule of Composition:

$$\frac{\{\mathbf{P}\}\ S_1\ \{\mathbf{R}\},\quad \{\mathbf{R}\}\ S_2\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ S_1;\ S_2;\ \{\mathbf{Q}\}}$$

Conclusion:

$$\frac{\mathbf{P}\ \Rightarrow\ \mathbf{P}',\quad \{\mathbf{P}'\}\ S\ \{\mathbf{Q}'\},\quad \mathbf{Q}'\ \Rightarrow\ \mathbf{Q}}{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}$$

extended Rule of Composition:

$$\frac{\{\mathbf{P}\}\ S_1\ \{\mathbf{R}\},\quad \{\mathbf{R}'\}\ S_2\ \{\mathbf{Q}\},\quad \mathbf{R}\ \Rightarrow\ \mathbf{R}'}{\{\mathbf{P}\}\ S_1;\ S_2;\ \{\mathbf{Q}\}}$$

Our Toy Language

Ada-Code Fragments:

- variables of a (mathematical) integer type
- expressions
  - integer
  - boolean
- statements:
  - **null**;
  - assignment: variable **:=** integer-expr
  - sequence of statements
- control structures:
  - **if** boolean-expr **then** statement **else** statement **end if**;
  - **while** boolean-expr **loop** statement **end loop**

5.2: The WHILE rule is special

$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S\ \{\mathbf{P}\}}{\{\mathbf{P}\}\ \ \textbf{while}\ \ \mathbf{B}\ \textbf{loop}\ S;\ \textbf{end loop}\ ;\ \ \{\mathbf{P},\ \neg\mathbf{B}\}}$$

1. WHILE separates partial from complete correctness.
   Note that the language we currently have does not support (recursive) subprograms.
2. WHILE-loops are difficult to prove correct, both manually and (especially) automatically.

Invariant

An important tool to analyze loops

P is an **invariant** of a loop, if it holds before and after each iteration of the loop.

Example:

$\{\mathbf{Y = 0}\}$ **while** $X > 0$ **loop** $Y := Y+1; X := X-1;$ **end loop**; $\{\mathbf{Y = X^{old}}\}$

What invariant would you use?

Variant

Another tool to analyze loops – for total correctness

An integer expression V is a **variant** of a loop,

1. if $V \geq 0$ and
2. if V decreases during each iteration.

Example:

$$\{\mathbf{Y = 0}\}\ \textbf{while}\, X > 0\, \textbf{loop}\ Y := Y+1; X := X-1; \textbf{end loop};\ \ \{\mathbf{Y = X^{old}}\}$$

What is the variant of the loop?

While-Loop

Proof

Example:
How would you prove the following code correct?
How would you prove it terminates?

while $X > 0$ **loop**
$Y := Y + 1;$
$X := X - 1;$
end loop;

5.3: Verifying Code Fragments

How would you prove the following claim?

$$\{\mathbf{X} = \mathbf{X}^{\text{old}},\ \mathbf{Y} = \mathbf{Y}^{\text{old}}\} \quad T := X;\ X := Y;\ Y := T; \quad \{\mathbf{X} = \mathbf{Y}^{\text{old}},\ \mathbf{Y} = \mathbf{X}^{\text{old}}\}$$

Recall the Rules

Assignment, Composition, Postcondition Weakening

$$\frac{\text{True}}{\{\mathbf{P}[\text{Expr}\rightarrow\mathbf{v}]\}\ v{:=}\text{Expr};\ \{\mathbf{P}\}} \qquad \frac{\{\mathbf{P}\}\ S_1\ \{\mathbf{R}\},\quad \{\mathbf{R}\}\ S_2\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ S_1;\ S_2;\ \{\mathbf{Q}\}} \qquad \frac{\{\mathbf{P}\}\ S\ \{\mathbf{Q}'\},\quad \mathbf{Q}'\Rightarrow\mathbf{Q}}{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}$$

The proof:

$$\{\mathbf{X}=\mathbf{X}^{\text{old}},\ \mathbf{Y}=\mathbf{Y}^{\text{old}}\}\ \ T:=X;\ \ \{\mathbf{T}=\mathbf{X}^{\text{old}},\ \mathbf{X}=\mathbf{X}^{\text{old}},\ \mathbf{Y}=\mathbf{Y}^{\text{old}}\}$$

$$\{\mathbf{T}=\mathbf{X}^{\text{old}},\ \mathbf{X}=\mathbf{X}^{\text{old}},\ \mathbf{Y}=\mathbf{Y}^{\text{old}}\}\ \ X:=Y;\ \ \{\mathbf{T}=\mathbf{X}^{\text{old}},\ \mathbf{X}=\mathbf{Y}^{\text{old}},\ \mathbf{Y}=\mathbf{Y}^{\text{old}}\}$$

$$\{\mathbf{T}=\mathbf{X}^{\text{old}},\ \mathbf{X}=\mathbf{Y}^{\text{old}},\ \mathbf{Y}=\mathbf{Y}^{\text{old}}\}\ \ Y:=T;\ \ \{\mathbf{T}=\mathbf{X}^{\text{old}},\ \mathbf{X}=\mathbf{Y}^{\text{old}},\ \mathbf{Y}=\mathbf{X}^{\text{old}}\}$$

$$\{\mathbf{T}=\mathbf{X}^{\text{old}},\ \mathbf{X}=\mathbf{Y}^{\text{old}},\ \mathbf{Y}=\mathbf{X}^{\text{old}}\}\ \ \Longrightarrow\ \ \{\mathbf{X}=\mathbf{Y}^{\text{old}},\ \mathbf{Y}=\mathbf{X}^{\text{old}}\}$$

Much Ado about Nothing?

- Much Ado, to prove a triviality.
- Manual proofs are error-prone

Searching for possible proof strategies

Something, a computer can do, (mostly) without human intervention

The claim

$$\{\mathbf{X = 5}\} \quad X := 2 * x; \quad \{\mathbf{X > 0}\}$$

is correct, but not tight. I.e., we could replace the precondition by a weaker one, or the postcondition by a stronger one.

There are several weaker preconditions and several stronger postconditions. Which one should the computer choose?

The two extreme cases are the following:

Weakest Precondition: $\mathbf{X > 0}$

Strongest Postcondition: $\mathbf{X = 10}$

Strongest Postcondition

Theorem: Let S be a program and $\mathbf{P}$ a precondition. Then there exists a unique $\mathbf{Q}^*$ with

- $\{\mathbf{P}\}\ S\ \{\mathbf{Q}^*\}$
- and for all $\mathbf{Q_i}$ with $\{\mathbf{P}\}\ S\ \{\mathbf{Q_i}\}$ it holds that $\mathbf{Q}^* \Rightarrow \mathbf{Q_i}$.

Proof (Sketch): Assume $\mathbf{Q_1}$ and $\mathbf{Q_2}$ are both postconditions:

$$\{\mathbf{P}\}\ S\ \{\mathbf{Q_1}\} \quad \text{and} \quad \{\mathbf{P}\}\ S\ \{\mathbf{Q_2}\}.$$

Then $\mathbf{Q_0} = \mathbf{Q_1} \wedge \mathbf{Q_2}$ is also a postcondition:

$$\{\mathbf{P}\}\ S\ \{\mathbf{Q_0}\}.$$

Furthermore, $\mathbf{Q_0}$ is at least as strong as either $\mathbf{Q_1}$ or $\mathbf{Q_2}$:

$$\mathbf{Q_0} \rightarrow \mathbf{Q_1} \quad \text{and} \quad \mathbf{Q_0} \rightarrow \mathbf{Q_2}.$$

Definition: $\mathbf{Q}^*$ the **strongest postcondition**.

Weakest Precondition

Theorem: Let S be a program amd $\mathbf{Q}$ a postcondition. there exists a unique $\mathbf{P}^*$ with

- $\{\mathbf{P}^*\}\ S\ \{\mathbf{Q}\}$
- for all $\mathbf{P_i}$ with $\{\mathbf{P_i}\}\ S\ \{\mathbf{Q}\}$ it holds that $\mathbf{P_i} \Rightarrow \mathbf{P}^*$.

Proof: (similar to the strongest postcondition case)

Definition: $\mathbf{P}^*$ is the **weakest precondition**.

Remark 1: Though each of the strongest postcondition $\mathbf{Q}^*$ and the weakest precondition $\mathbf{P}^*$ is logically unique, but it can be represented by different expressions.
E.g., the the following expressions are logically the same: "$X \geq 0$", "$X = \text{abs}(X)$", and "$X \geq 1 \vee X = 0$".

Remark 2: In practice, the weakest precondition is preferred over the strongest postcondition.

Weakest precondition calculus

For all (deterministic) programs S, the following rules/laws hold:

Miracles excluded:

$$WP(S, \text{False}) = \text{False}$$

Validity rule:

$$WP(S, \text{True}) = \text{True}$$

Conjunction and disjunction:

$$\text{WP}(S, Q) \wedge \text{WP}(S, R) = \text{WP}(S, Q \wedge R)$$

$$\text{WP}(S, Q) \vee \text{WP}(S, R) = \text{WP}(S, Q \vee R)$$

Conclusions:

$$\text{Für } Q \rightarrow R\text{:} \quad \text{WP}(S, Q) \rightarrow \text{WP}(S, R)$$

How the computer computes WP$(\cdot, \cdot)$

Empty Statement:

$$\mathrm{WP}(\textbf{null};, \mathbf{P}) = \mathbf{P}$$

Assignment:

$$\mathrm{WP}(X := \mathrm{Expr};, \mathbf{Q}) = \mathbf{Q}[\mathrm{Expr} \to X]$$

Composition:

$$\mathrm{WP}(S; T;, \mathbf{Q}) = \mathrm{WP}(S, \mathrm{WP}(T, \mathbf{Q}))$$

IF-THEN-ELSE:

$$\begin{aligned} & \mathrm{WP}(\textbf{if } \mathbf{B} \textbf{ then } S \textbf{ else } T \textbf{ end if};, \mathbf{Q}) \\ = \quad & (\mathbf{B} \Rightarrow \mathrm{WP(S,Q)}) \ \wedge \ (\neg\mathbf{B} \Rightarrow \mathrm{WP(T,Q)}) \end{aligned}$$

WHILE:

$$\mathrm{WP}(\textbf{while } \mathbf{B} \textbf{ loop } S \textbf{ end while};, \mathbf{Q}) \quad = \quad \textbf{???}$$

The woe with WHILE

$$\text{WP}(\textbf{while B loop } S \textbf{ end while};, \mathbf{Q}) \quad = \quad \textbf{???}$$

We write cnt(i) for "the loop is iterated exactly i times", and $\overline{\text{cnt}(\infty)}$ for "the loop runs forever". Write S^i for i repetions of S (e.g, $S^3 = S; S; S;$). Then

$$\textbf{???} = \overline{\text{cnt}(\infty)} \vee \left(\bigvee_{i \in \{0,1,2,3,4,\ldots\}} (\text{cnt}(i) \wedge \text{WP}(S^i;, \mathbf{Q})) \right)$$

This gives us the WP – but it is hardly useful for our purpose.

WHILE: What to do in practice

Consider a loop

$$\{\mathbf{P}\}\ \textbf{while}\ \mathbf{B}\ \textbf{loop}\ S\ \textbf{end while}\ \{\mathbf{Q}\}$$

with a precondition $\{\mathbf{P}\}$ and a postcondition $\{\mathbf{Q}\}$.

We need "cut the loop" by a loop invariant **I**. We require it to satisfy:

1. $\mathbf{P} \Rightarrow \mathbf{I}$ ("**I** is true before the loop"),
2. $\mathbf{I} \wedge \mathbf{B} \rightarrow \mathrm{WP}(S, \mathbf{I})$
   ("**I** remains intact under iteration of the loop body"), and
3. $\mathbf{I} \wedge \neg\mathbf{B} \rightarrow \mathbf{Q}$
   ("When **I** holds at loop termination, the postcondition holds").

People usually write a loop with an invariant as

$$\{\mathbf{P}\}\ \textbf{while}\ \{\mathbf{I}\}\ \mathbf{B}\ \textbf{loop}\ S\ \textbf{end while}\ \{\mathbf{Q}\}$$

But note that **I** must hold even if **B** is false from the beginning. It would be more clear to write $\{\mathbf{P}\}\ \{\mathbf{I}\}\ \textbf{while}\ \mathbf{B}\ \textbf{loop}\ S\ \{\mathbf{I}\}\ \textbf{end while}\ \{\mathbf{Q}\}$.

Examples

Assignment, Composition, IF-THEN-ELSE

Assignment:

$$\mathrm{WP}(\text{“}c := a + b;\text{”}, \mathbf{c = 6}) = \mathbf{a + b = 6}.$$

Composition:

$$\begin{aligned} & \mathrm{WP}(\text{“}b := a;\, c := a + b;\text{”}, \mathbf{c = 6}) \\ = \quad & \mathrm{WP}(\text{“}b := a;\text{”}, \mathbf{a + b = 6}) \;=\; \mathbf{a + a = 6}. \end{aligned}$$

(logically the same as $\mathbf{a = 3}$).

IF-THEN-ELSE:

$$\begin{aligned} & \mathrm{WP}(\text{“}\textbf{if } X >= 0 \textbf{ then null else } X := -X;\ \textbf{end if;}\text{”}, \mathbf{X > 0}) \\ = \quad & \ldots \end{aligned}$$

Example

WHILE

$\{\mathbf{N}^{\text{old}} = \mathbf{N} > \mathbf{0},\ \mathbf{M}^{=}\mathbf{M} \geq \mathbf{0}\}$
$R := 1;$
$S := 0;$
$\{\textbf{Invariant}\}$
while $\mathbf{S} < \mathbf{M}$ **loop**
    $R := R * N;$
    $S := S + 1;$
$\{\textbf{Invariant}\}$
end loop;
$\{\mathbf{N}^{\text{old}} = \mathbf{N},\ \mathbf{M}^{\text{old}} = \mathbf{M},\ \mathbf{R} = \mathbf{N}^{\mathbf{M}}\}$

How to find loop invariants?

$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ \ S\ \ \{\mathbf{P}\}}{\{\mathbf{P}\}\quad \textit{while}\ \mathbf{B}\ \textit{loop}\ S;\ \textit{end loop};\quad \{\mathbf{P},\ \neg\mathbf{B}\}}$$

- The invariant **P** should hold before the loop, and after each loop iteration.
- Informally, it describes that
  - what has been achieved by the loop iterations so far
  - together with what will be achieved by future loop iterations
  - will provide the claimed result.

Back to the example!

$\{\mathbf{N^{old} = N > 0},\ \mathbf{M^{old} = M \geq 0}\}$
$R := 1;$
$S := 0;$
$\{\mathbf{Invariant}\}$
while $\mathbf{S} < \mathbf{M}$ **loop**
    $R := R * N;$
    $S := S + 1;$
$\{\mathbf{Invariant}\}$
end loop;
$\{\mathbf{N^{old} = N},\ \mathbf{M^{old} = M},\ \mathbf{R = N^M}\}$

Rule of thumb:

- Take the postcondition and
- describe it as being dependent from the local variables (as opposed from the input parameters),
- such that when leaving the loop, the invariant is the same as the postcondition.
- Here: $\mathbf{Invariant} := \mathbf{R = N^S}$.

Continuing the example

$\{\mathbf{N}^{\text{old}} = \mathbf{N} > \mathbf{0},\ \mathbf{M}^{\text{old}} = \mathbf{M} \geq \mathbf{0}\}$
$R := 1;$
$S := 0;$
$\{\mathbf{R} = \mathbf{N}^{\mathbf{S}}\}$
while $S < M$ loop
    $R := R * N;$
    $S := S + 1;$
$\{\mathbf{R} = \mathbf{N}^{\mathbf{S}}\}$
end loop;
$\{\mathbf{N}^{\text{old}} = \mathbf{N},\ \mathbf{M}^{\text{old}} = \mathbf{M},\ \mathbf{R} = \mathbf{N}^{\mathbf{S}},\ \mathbf{S} \geq \mathbf{M}\}$

- Are we done?
- Not quite. We leave the while loop when $\neg(\mathbf{S} < \mathbf{M})$ holds.
- This is equivalent to $\mathbf{S} \geq \mathbf{M}$, but
- we would need $\mathbf{S} = \mathbf{M}$.

Finishing the example

$\{\mathbf{N}^{\text{old}} = \mathbf{N} > \mathbf{0},\ \mathbf{M}^{\text{old}} = \mathbf{M} \geq \mathbf{0}\}$
$R := 1;$
$S := 0;$
$\{\mathbf{R} = \mathbf{N}^{\mathbf{S}}\}$
while $S < M$ loop
    $R := R * N;$
    $S := S + 1;$
$\{\mathbf{R} = \mathbf{N}^{\mathbf{S}}\}$
end loop;
$\{\mathbf{N}^{\text{old}} = \mathbf{N},\ \mathbf{M}^{\text{old}} = \mathbf{M},\ \mathbf{R} = \mathbf{N}^{\mathbf{S}},\ \mathbf{S} = \mathbf{M}\}$

- Are we done?
- Not quite. We leave the loop when $\neg(\mathbf{S} < \mathbf{M})$ holds.
- This is equivalent to $\mathbf{S} \geq \mathbf{M}$, but
- we actually need $\mathbf{S} = \mathbf{M}$.
- The simplest way to do so is to write “while $S \neq M$”.

Two programs to compute the factorial. Which Invariants would you choose?

$\{\mathbf{X} = \mathbf{N},\ \mathbf{Y} = \mathbf{1}\}$
while $\mathbf{X} \neq \mathbf{0}$ loop
$Y := Y * X;$
$X := X - 1;$
end loop;
$\{\mathbf{X} = \mathbf{0},\ \mathbf{Y} = \mathbf{N}!\}$

$\{\mathbf{X} = \mathbf{0},\ \mathbf{Y} = \mathbf{1}\}$
while $\mathbf{X} < \mathbf{N}$ loop
$X := X + 1;$
$Y := Y * X;$
end loop;
$\{\mathbf{X} \geq \mathbf{N},\ \mathbf{Y} = \mathbf{N}!\}$

Total Correctness

- to claim total correctness, we write

$$[\mathbf{P}]\ S\ [\mathbf{Q}]$$

- this holds, if the partial correctness is given

$$\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}$$

- **and** S always terminates (assuming **P**).

Rules for programs without loops

No surprise!

- Replace "$\{\ldots\}$" by "$[\ldots]$" in all axioms and rules – except for WHILE. (We later will elaborate on the assignment axiom ...)
- If S is without a WHILE loop, this trivially gives

$$\frac{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}}{[\mathbf{P}]\ S\ [\mathbf{Q}]}$$

WHILE

Consider the following proof:

1. $\{\mathbf{P}\}$ **null**; $\{\mathbf{P}\}$ (Axiom)
2. $\{\mathbf{P}, \mathbf{P}\}$ **null**; $\{\mathbf{P}\}$ (Precondition Strengthening)
3. $\{\mathbf{P}, \mathbf{P}\}$ **while P loop null**; **end loop**; $\{\mathbf{P}, \neg\mathbf{P}\}$ (WHILE)

Let us try the easy way:
If we just replace "$\{\ldots\}$" by "$[\ldots]$", we get a **a contradition**:

$$[\mathbf{P}, \mathbf{P}] \quad \textbf{while P loop } S \textbf{ end loop}; \quad [\mathbf{P}, \neg\mathbf{P}]$$

No suprise – our loop doesn't terminate, and **our replacement rule is plain wrong** for WHILE loops!

WHILE (2)

To prove the termination of

while B loop S **end loop**;

one proves the existence of an integer $V \geq 0$,
which decreases with every iteration of S.
This integer V is **the variant**.

Back to the old example

$\{\mathbf{N^{old} = N > 0},\ \mathbf{M^{old} = M \geq 0}\}$
$R := 1;$
$S := 0;$
$\{\mathbf{R = N^S}\}$
while $S < M$ loop
$\quad R := R * N;$
$\quad S := S + 1;$
$\{\mathbf{R = N^S}\}$
end loop;
$\{\mathbf{N^{old} = N},\ \mathbf{M^{old} = M}, \mathbf{R = N^S},\ \mathbf{S = M}\}$

- What could be the variant?

The assignment: take heed!

For the assignment, we simply replace “{. . .}” by “[. . .]”. I.e.,

$$\{\mathbf{P}[\text{Expr} \rightarrow \mathbf{v}]\}\ \text{v:=Expr;}\ \{\mathbf{P}\}$$

becomes

$$[\mathbf{P}[\text{Expr} \rightarrow \mathbf{v}]]\ \text{v:=Expr;}\ [\mathbf{P}]$$

which describes the expected behaviour well.

BUT:

This assumes that the computation of “Expr” always terminates. For more general languages, which, e.g., allow to call recursive functions in expressions, this is plain wrong.

How to deal with errors

e.g., when computing X/Y and $Y = 0$

1. Just ignore the problem
   - $X/0$ is an arbitrary number, who cares?
   - But then, why do you need a proof at all?
2. Raise an exception / an error
   - may be OK for partial but not for total correctness
3. Continue with some special value
   - $X/0$ may be “Not_A_Number”,
   - as will be $(X/0) * 0$, $(X/0) + A$, ...
4. Defend against such cases in advance
   - X/Y enforces **$Y \neq 0$** as an **additional precondition**
   - ($\rightarrow$ SPARK).

5.4: Extending the Toy Language IF-THEN-(no-ELSE)

Syntax:

- **if** B **then** S **end if**;

Semantic:

- **if** B **then** S **else null**; **end if**;

Justification

IF-THEN-ELSE:

$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S_1\ \{\mathbf{Q}\},\quad \{\mathbf{P},\ \neg\mathbf{B}\}\ S_2\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ \textit{if}\ \mathbf{B}\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{end if};\ \{\mathbf{Q}\}}$$

Set $S_2 = \textit{null};$:

$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S_1\ \{\mathbf{Q}\},\quad \{\mathbf{P},\ \neg\mathbf{B}\}\ \textit{null};\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ \textit{if}\ \mathbf{B}\ \textit{then}\ S_1\ \textit{else null};\ \textit{end if};\ \{\mathbf{Q}\}}$$

Empty Statement:

$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S\ \{\mathbf{Q}\}\quad (P \wedge \neg\mathbf{B}) \Rightarrow Q}{\{\mathbf{P}\}\ \textit{if}\ \mathbf{B}\ \textit{then}\ S\ \textit{end if};\ \{\mathbf{Q}\}}$$

Wow! That was really easy!

REPEAT-UNTIL

Syntax:

- **repeat** S **until** B; (∗ this is Pascal ∗)

Semantic:

- S **while** (¬ B) **loop** S **end loop**; — this is Ada

Justification

WHILE:

$$\frac{\{\mathbf{P},\ \mathbf{B}\}\ S\ \{\mathbf{P}\}}{\{\mathbf{P}\}\ \textit{while}\ \mathbf{B}\ \textit{loop}\ S\ \textit{end loop};\ \{\mathbf{P},\ \neg\mathbf{B}\}}$$

This gives

$$\frac{\{\mathbf{P}\}\ S\ \{\mathbf{Q}\},\quad \{\mathbf{Q},\ \neg\mathbf{B}\}\ S\ \{\mathbf{Q}\}}{\{\mathbf{P}\}\ \textit{repeat}\ S\ \textit{until}\ \mathbf{B};\ \{\mathbf{Q},\ \mathbf{B}\}}$$

FOR-loops

Syntax:

for *I* **in** *A* .. *B* **loop** S **end loop**;

Simple semantic for FOR-loops:

I :=A;
while $I \leq B$ **loop**
    S
    $I := I + 1$;
end loop;

Semantic Problem – FOR-loops are Difficult

There is a problem with our simple semantic:

- FOR-loops should always terminate, unlike WHILE-loops. (There is an obvious variant ...).
- But our simple semantic definition allows to write FOR-loops to runing forever. Example:

$$\{\textbf{True}\} \quad \textbf{for } I \textbf{ in } 1 \,..\, 1 \textbf{ loop } I := I - 1; \textbf{ end loop}; \quad \{\mathbf{2 = 1}\} \text{ ,}$$

- Note: This is the toy language and *not* Ada. For Ada the assignment to a loop parameter would produce a compiler error.
- With a stricter definition and some constraints (e.g., prohibit assigning a value to the loop counter), the variant for

$$\textbf{for } I \textbf{ in } A \,..\, B \textbf{ loop } S; \textbf{ end loop};$$

  is just $B - I + 1$.
- Many old programming languages (e.g., Algol-60) lack a sufficiently strict semantic for FOR-loops ...
- ... and many new programming languages also (e.g., Java).

Arrays

Now it gets even more difficult!

Recall the assignment rule:

$$\{\mathbf{P}[\text{Expr} \rightarrow \mathbf{v}]\}\ v := \text{Expr};\ \{\mathbf{P}\}$$

For assignments related to an array $A(\ldots)$, we get

$$\{\mathbf{P}[\text{Expr}_\mathbf{2} \rightarrow \mathbf{A}(\text{Expr}_\mathbf{1})]\}\ A(\text{Expr}_1) := \text{Expr}_2;\ \{\mathbf{P}\}.$$

So where is the problem?

Aliasing

Recall

$$\{\mathbf{P}[\mathrm{Expr}_2 \to \mathbf{A}(\mathrm{Expr}_1)]\}\ A(\mathrm{Expr}_1) := \mathrm{Expr}_2;\ \{\mathbf{P}\}.$$

and define

- $\mathbf{P}$ as "$(X = Y) \wedge (A(Y) = 0)$",
- Expr_1 as "X" and
- Expr_2 as "1".

Then $\mathbf{P}[\mathrm{Expr}_2 \to \mathbf{A}(\mathrm{Expr}_1)]$ reduces to

$$\mathbf{P}[\mathbf{1} \to \mathbf{A}(\mathbf{X})].$$

and this is just

$$\mathbf{P},$$

since $\mathbf{P}$ contains no expression $A(X)$.

Thus:

$$\{\mathbf{X} = \mathbf{Y}, \ldots\}\ \ A(X) := 1;\ \ \{\mathbf{A(Y)} = \mathbf{0}\}.$$

Ouch!

Aliasing

$A(X)$ and $A(Y)$ are two different expressions, but can denote the same element of A. In that case, changing either actually changes the other one.

Hoare's solution

An assignment

$$A(\text{Expr1}) := \text{Expr2}$$

is treated as

$$A := A^{\underline{(\text{Expr1}) \leftarrow \text{Expr2}}}.$$

Here $A^{\underline{(\text{Expr1}) \leftarrow \text{Expr2}}}$ denotes the array, which you get if you replace $A(\text{Expr1})$ by Expr2 and leave all other elements of A unchanged.

Two new axioms

New assignment axiom: ("A(Expr1) is replaced by"):

$$A^{(\text{Expr1}) \leftarrow \text{Expr2}}(\text{Expr1}) = \text{Expr2}.$$

Addional "Frame"-axiom ("the other elements of A remain unchanged"):

IF

$$\text{Expr}_1 \neq \text{Expr}_3$$

THEN

$$A^{(\text{Expr1}) \leftarrow \text{Expr2}}(\text{Expr3}) = A(\text{Expr3}).$$

Example: Swap $A(X)$ and $A(Y)$

$\{\mathbf{A(X) = X^{old}},\ \mathbf{A(Y) = Y^{old}}\}$
T := $A(X)$;
$A(X) := A(Y)$;
$A(Y) := T$;
$\{\mathbf{A(X) = Y^{old}},\ \mathbf{A(Y) = X^{old}}\}$

The proof

$$\{\mathbf{A(X)} = \mathbf{X}^{\text{old}},\ \mathbf{A(Y)} = \mathbf{Y}^{\text{old}}\}$$

$$\mathsf{T} := A(X);$$

$$\{\quad \mathbf{A(Y)} = \mathbf{Y}^{\text{old}}(!),\quad \mathbf{T} = \mathbf{X}^{\text{old}}\quad\}$$

$$\{\quad \left(\mathbf{A}^{(\mathsf{X}) \leftarrow A(Y)}\right)^{(\mathsf{Y}) \leftarrow \mathsf{T}}(\mathbf{X}) = \mathbf{Y}^{\text{old}},\quad \mathbf{T} = \mathbf{X}^{\text{old}}\quad\}$$

$$A(X) := A(Y);$$

$$\{\quad \mathbf{A}^{(\mathsf{Y}) \leftarrow \mathsf{T}}(\mathbf{X}) = \mathbf{Y}^{\text{old}},\quad \mathbf{T} = \mathbf{X}^{\text{old}}\quad\}$$

$$\{\quad \mathbf{A}^{(\mathsf{Y}) \leftarrow \mathsf{T}}(\mathbf{X}) = \mathbf{Y}^{\text{old}},\quad \mathbf{A}^{(\mathsf{Y}) \leftarrow \mathsf{T}}(\mathbf{Y}) = \mathbf{X}^{\text{old}}\quad\}$$

$$A(Y) := T;$$

$$\{\mathbf{A(X)} = \mathbf{Y}^{\text{old}},\ \mathbf{A(Y)} = \mathbf{X}^{\text{old}}\}$$

Still missing (!)

$$\left(\mathbf{A}\underline{(\mathrm{X}) \leftarrow A(Y)}\right)\underline{\frac{(\mathrm{Y}) \leftarrow \mathrm{T}}{}}(\mathbf{X}) = \mathbf{A}(\mathbf{Y})$$

- Case 1: $X \neq Y$. (That is easy.)
- Case 2: $X = Y$. (That is a bit tricky!)

Pointers / access types

- In principle, one can treat the heap like a huge array of storage places.
- But if a program makes significant usage of heap-allocated data, proving properties becomes extremely difficult.
- Current research issue, see, e.g.,: Bertrand Meyer, “The Theory and Calcus of Aliasing”:
  ⟨http://bertrandmeyer.com/2010/01/21/the-theory-and-calculus-of-aliasing/⟩

Parameterless procedures

Syntax:

- **procedure** P **is begin** S_0; **end**

Semantic:

- Whenever P is called, we actually execute S_0.

Rule for non-recursive parameterless procedures:

$$\frac{\{\mathbf{P}\}\quad S_0;\quad \{\mathbf{Q}\}}{\{\mathbf{P}\}\quad P;\quad \{\mathbf{Q}\}}$$

Nonrecursive procedures (and functions) with parameters, e.g.,

procedure $P(X : \ldots)$ **is** S_0 **end** P;

Calling P(Expr) is the same as assigning Expr to X then runing S_0.

Functions / Prozedures in general

- Recursive procedures:
  - partial correcntess similar to the nonrecursive case
  - termination needs an upper bound (like the variant for WHILE-loops) for the recursion depth
- Theorem: There is no complete* and sound** system for any language which combines
  - procedures with procedure-parameters (**access procedure**),
  - recursion,
  - statical scoping,
  - global variables, and
  - locally defined procedures as procedure-parameters.
- This combination of features is supported by many languages, starting from Algol-60.

---

* complete: all correct statements are provable

** sound: no incorrect statements are provable