Construction and Verification of Software

Master Programme in Computer Science

Mário Pereira   mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

October 6, 2025

Lecture 5

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

also based on lectures by Andrei Paskevich and Claude Marché

Verification of Imperative Programs (2/n)

1. Hoare Logic Recap
2. Hoare Logic Rule for Loops
3. Loop Invariants
4. Weakest Pre-condition Calculus

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

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

Rules of Hoare Logic (general form) – Recap

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

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

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

$$
\begin{array}{lcll}
E & ::= & & \text{Expressions} \\
 & | & \ldots & \\
 & & & \\
S & ::= & & \text{Statements} \\
 & | & \ldots & \\
 & | & x := m(E_1, \ldots, E_n) & \text{Call + Assignment} \\
 & & & \\
D & ::= & & \text{Declarations} \\
 & | & \texttt{method}\, m(x_1, \ldots, x_n)\ \texttt{returns}\,(r) & \\
 & & \texttt{requires}\, Pre(x_1, \ldots, x_n) & \\
 & & \texttt{ensures}\, Post(x_1, \ldots, x_n, r) & \\
 & & \{S\} & \\
 & & & \\
P & ::= & & \text{Program} \\
 & | & \overline{D} &
\end{array}
$$

Procedures and method calls – Recap

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.

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

Procedures and method calls – Recap

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:

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

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

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

Verification with method calls is modular.

Rules for `if` and `while`:

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

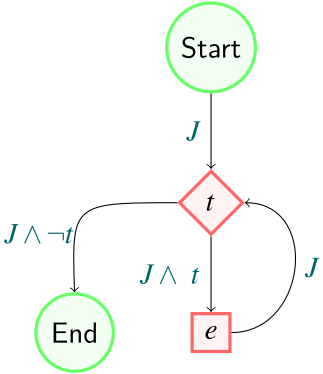
$$\frac{\{J \wedge t\}\, e\, \{J\}}{\{J\}\, \texttt{while}\ t\ \texttt{do}\ e\, \{J \wedge \neg t\}}$$

Formula J is a loop invariant.

Finding a right invariant is a major difficulty.

Loops in Hoare Logic

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



Hoare Logic Rule for Loops – In Words

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

J	Invariant initially valid
$J \wedge t$	Loop condition is true
e	Execution of the loop body
J	Invariant re-established
$J \wedge t$	Loop condition is true
e	Execution of the loop body
J	Invariant re-established
$\vdots$	(any number of iterations)
J	Invariant re-established
$J \wedge \neg t$	Loop exits; condition is false

$$J = \text{Invariant Condition}$$

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

We cannot predicate in general for how many iterations the `while` loop will run (undecidability of the halting problem).

We approximate all iterations by an invariant condition.

A loop invariant holds at loop entry and at loop exit.

$$J = \text{Invariant Condition}$$

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

If the invariant holds initially and is preserved by the loop body, it will hold when the loop terminates.

No matter how many iterations will run.

Finding the good invariant requires patience and creativity.

$$J = \text{Invariant Condition}$$

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

The invariant describes the state in any possible iteration.

The invariant works like the induction hypothesis in a proof.

- The base case is the loop executed 0 times,
- the loop body is the induction step that iterates from n to $n+1$,
- There must exist a valid induction measure.

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

$\{0 \leq n\}$

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

$\{i == n\}$

$\{0 \leq n\}$

```
i := 0;
```

$\{i == 0 \land 0 \leq n\}$

$\{0 \leq i \leq n\}$

```
while i < n do {
```

$\{0 \leq i \leq n \land i < n\}$

$\{0 \leq i < n\}$

$\{0 \leq i + 1 \leq n\}$

```
  i := i + 1;
```

$\{0 \leq i \leq n\}$

```
}
```

$\{0 \leq i \leq n \land i == n\}$

$\{i == n\}$

Loop Invariants, in Practice

Consider the following program, lets call it `P`:

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

What is a good specification for `P`? What does `P` compute?

Consider the following program, lets call it P:

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

What is a good specification for P? What does P compute?

$$\{0 \leq n\}\,\mathtt{P}\,\{s == \sum_{j=0}^{n} j\}$$

Is this a good specification for P?

Lets resort to Hoare Logic to mechanically check this triple.

$\{0 \leq n\}$

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

$\{s == \sum_{j=0}^{n} j\}$

$\{0 \leq n\}$

```
s := 0;
```

$\{s == 0 \land 0 \leq n\}$

```
i := 0;
```

$\{i == 0 \land s == 0 \land 0 \leq n\}$

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

$\{s == \sum_{j=0}^{n} j\}$

$\{0 \leq n\}$

```
s := 0;
```

$\{s == 0 \wedge 0 \leq n\}$

```
i := 0;
```

$\{i == 0 \wedge s == 0 \wedge 0 \leq n\}$

$\{s == 0 \wedge 0 \leq i \leq n\}$

$$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$$

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

$$\{s == \sum_{j=0}^{n} j\}$$

$\{0 \le n\}$

```
s := 0;
```

$\{s == 0 \land 0 \le n\}$

```
i := 0;
```

$\{i == 0 \land s == 0 \land 0 \le n\}$

$\{s == 0 \land 0 \le i \le n\}$

$$\{s == \sum_{j=0}^{i} j \land 0 \le i \le n\}$$

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

$$\{i == n \land s == \sum_{j=0}^{i} j\}$$

$$\{s == \sum_{j=0}^{n} j\}$$

```
{0 ≤ n}
s := 0;
{s == 0 ∧ 0 ≤ n}
i := 0;
{i == 0 ∧ s == 0 ∧ 0 ≤ n}
{s == 0 ∧ 0 ≤ i ≤ n}
```

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$ *//invariant initially holds*

```
while i < n do {
```

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n \wedge i < n\}$ *//invariant holds at the beginning of iteration*

```
  i := i + 1;
  s := s + i;
```

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$ *//invariant is maintained*

```
}
```

$\{i == n \wedge s == \sum_{j=0}^{i} j\}$

$\{s == \sum_{j=0}^{n} j\}$

The loop invariant:

- may be broken inside the loop body
- must be re-established at the end.

Notice how the assignment rule

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

breaks the invariant...

$$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n \wedge i < n\}$$

```
i := i + 1;
```

$$\{s == \sum_{j=0}^{i-1} j \wedge 0 \leq i-1 \leq n \wedge i-1 < n\}$$

$$\{s == \sum_{j=0}^{i-1} j \wedge 0 \leq i \leq n\}$$

The loop invariant:

- may be broken inside the loop body
- must be re-established at the end.

Notice how the assignment rule

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

and then re-establishes it

$$\{s == \sum_{j=0}^{i-1} j \land 0 \leq i \leq n\}$$

```
s := s + i;
```

$$\{s == (\sum_{j=0}^{i-1} j) + i \land 0 \leq i \leq n\}$$

$$\{s == \sum_{j=0}^{i} j \land 0 \leq i \leq n\}$$

$\{0 \leq n\}$

```
s := 0;
```

$\{s == 0 \wedge 0 \leq n\}$

```
i := 0;
```

$\{i == 0 \wedge s == 0 \wedge 0 \leq n\}$

$\{s == 0 \wedge 0 \leq i \leq n\}$

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$ *//invariant initially holds*

while i < n **do** {

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n \wedge i < n\}$ *//invariant holds at the beginning of iteration*

```
  i := i + 1;
  s := s + i;
```

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$ *//invariant is maintained*

}

$\{i == n \wedge s == \sum_{j=0}^{i} j\}$

$\{s == \sum_{j=0}^{n} j\}$

Checking Loop Invariants – the big picture

$\{0 \leq n\}$

```
s := 0;
```

$\{s == 0 \wedge 0 \leq n\}$

```
i := 0;
```

$\{i == 0 \wedge s == 0 \wedge 0 \leq n\}$

$\{s == 0 \wedge 0 \leq i \leq n\}$

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$ *//invariant initially holds*

```
while i < n do {
```

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n \wedge i < n\}$ *//invariant holds at the beginning of iteration*

```
  i := i + 1;
```

$\{s == \sum_{j=0}^{i-1} j \wedge 0 \leq i-1 \leq n \wedge i-1 < n\}$

$\{s == \sum_{j=0}^{i-1} j \wedge 0 \leq i \leq n\}$ *//invariant is broken*

```
  s := s + i;
```

$\{s == (\sum_{j=0}^{i-1} j) + i \wedge 0 \leq i \leq n\}$ *//invariant is restored*

$\{s == \sum_{j=0}^{i} j \wedge 0 \leq i \leq n\}$ *//invariant is maintained*

```
}
```

$\{i == n \wedge s == \sum_{j=0}^{i} j\}$

$\{s == \sum_{j=0}^{n} j\}$

Hints for finding loop invariants

First: carefully think about the post condition of the loop.

Typically the post-condition talks about a property "accumulated" across a "range" (this is why you are using a loop, right?)

- maximum of all elements in an array
- sort visited elements in a data structure

Hints for finding loop invariants

Second: design a "generalized" version of the post- condition, in which the already visited part of the data is made explicit as a function of the "loop control variable" (generalizing the i.h, remember?)

The loop body may temporarily break the invariant, but must restore it at the end of the body.

Important: make sure that the invariant together with the termination condition really implies your post-condition

Examples of invariants we will need

Max of an array

- All scanned elements of the array are smaller than the max so far

Array searching (unsorted)

- All elements left of the index are different from the value being searched

Array searching (sorted)

- The element is between the lower and higher limits

Sorting

- Everything to the left of the cursor is sorted

List Reversing

- All elements on cursor's left are placed rightly in the result

Weakest Pre-condition Calculus

Proving the correctness of a program with Hoare Logic

- requires attaching pre-/post-conditions at every instruction
- manual derivations, *i.e.*, lack of automation.

Hoare Logic is not syntax direct: remember the CONSEQUENCE rule.

- There is no algorithm that directly encodes this formal system.

Can we remedy this situation?

- We suspect we can, since we have already used the Why3 tool.

How can we establish the correctness of a program?

One solution: Edsger Dijkstra, 1975

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

e expression

Q postcondition

computes the weakest precondition P such that $\{P\}\, e\, \{Q\}$

$\mathrm{WP}(e, Q)$ is recursively defined on the cases of e.

$$x := 3 * x * y \qquad \{\ x \text{ is even }\}$$

$\{\ 3xy \text{ is even }\}\qquad x := 3 * x * y \qquad \{\ x \text{ is even }\}$

Intuition of WP

$$\{\ 3xy \text{ is even }\ \} \qquad x := 3 * x * y \qquad \{\ x \text{ is even }\ \}$$

$$\{\ Q[s]\ \} \qquad x := s \qquad \{\ Q[x]\ \}$$

$$\{\ 3xy \text{ is even }\ \} \qquad x := 3 * x * y \qquad \{\ x \text{ is even }\ \}$$

$$\{\ Q[s]\ \} \qquad x := s \qquad \{\ Q[x]\ \}$$

$$\begin{array}{rl} \texttt{if}\ c\ \texttt{then} & e_1 \\ \texttt{else} & e_2 \end{array} \qquad \{\ Q\ \}$$

$$\{\ 3xy \text{ is even }\ \} \qquad x := 3 * x * y \qquad \{\ x \text{ is even }\ \}$$

$$\{\ Q[s]\ \} \qquad x := s \qquad \{\ Q[x]\ \}$$

$$\begin{array}{rl} \texttt{if}\ c\ \texttt{then} & e_1\, Q \quad \{\ Q\ \} \\ \texttt{else} & e_2\, Q \end{array}$$

$\{\ 3xy \text{ is even }\} \qquad x := 3 * x * y \qquad \{\ x \text{ is even }\}$

$\{\ Q[s]\ \} \qquad x := s \qquad \{\ Q[x]\ \}$

$\texttt{if}\ c\ \texttt{then}\ P_1\, e_1\, Q \quad \{\ Q\ \}$
$\texttt{else}\ P_2\, e_2\, Q$

$$\{\ 3xy \text{ is even }\} \qquad x := 3 * x * y \qquad \{\ x \text{ is even }\}$$

$$\{\ Q[s]\ \} \qquad x := s \qquad \{\ Q[x]\ \}$$

$$\begin{array}{lll} \{\ \text{if } c \text{ then } P_1 & \texttt{if } c \texttt{ then } P_1\, e_1\, Q & \{\ Q\ \} \\ \quad\quad \text{else } P_2\ \} & \quad\quad \texttt{else } P_2\, e_2\, Q & \end{array}$$

$$\{\ 3xy \text{ is even }\}\qquad x := 3 * x * y \qquad \{\ x \text{ is even }\}$$

$$\{\ Q[s]\ \}\qquad x := s \qquad \{\ Q[x]\ \}$$

$$\begin{array}{lll} \{\ \text{if } c \text{ then } P_1 & \texttt{if}\ c\ \texttt{then}\ P_1\, e_1\, Q & \{\ Q\ \} \\ \quad\quad \text{else } P_2\ \} & \quad\quad \texttt{else}\ P_2\, e_2\, Q & \end{array}$$

$$\texttt{if}\ c\ \texttt{then}\quad e \qquad \{\ Q\ \}$$

$\{\ 3xy \text{ is even }\}$ $\quad x := 3 * x * y \quad$ $\{\ x \text{ is even }\}$

$\{\ Q[s]\ \}$ $\quad x := s \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1$
$\text{else } P_2\ \}$ $\quad$ $\texttt{if } c \texttt{ then } P_1\, e_1\, Q$
$\texttt{else } P_2\, e_2\, Q$ $\quad$ $\{\ Q\ \}$

$\texttt{if } c \texttt{ then } P\, e\, Q$ $\quad$ $\{\ Q\ \}$

$$\{\ 3xy \text{ is even}\ \} \qquad x := 3 * x * y \qquad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \} \qquad x := s \qquad \{\ Q[x]\ \}$$

$$\begin{array}{lll} \{\ \text{if } c \text{ then } P_1 & \texttt{if } c \texttt{ then } P_1\, e_1\, Q & \{\ Q\ \} \\ \quad\ \text{else } P_2\ \} & \qquad\ \texttt{else } P_2\, e_2\, Q & \end{array}$$

$$\begin{array}{lll} \{\ \text{if } c \text{ then } P & \texttt{if } c \texttt{ then } P\, e\, Q & \{\ Q\ \} \\ \quad\ \text{else } Q\ \} & & \end{array}$$

$\{ \ 3xy \text{ is even} \ \}$	$x := 3 * x * y$	$\{ \ x \text{ is even} \ \}$
$\{ \ Q[s] \ \}$	$x := s$	$\{ \ Q[x] \ \}$
$\{ \text{ if } c \text{ then } P_1$ $\text{else } P_2 \ \}$	$\texttt{if } c \texttt{ then } P_1\, e_1\, Q$ $\texttt{else } P_2\, e_2\, Q$	$\{ \ Q \ \}$
$\{ \text{ if } c \text{ then } P$ $\text{else } Q \ \}$	$\texttt{if } c \texttt{ then } P\, e\, Q$	$\{ \ Q \ \}$
	$\texttt{while } c \texttt{ do } e$	$\{ \ Q \ \}$

$\{\ 3xy \text{ is even }\}\qquad x := 3 * x * y \qquad \{\ x \text{ is even }\}$

$\{\ Q[s]\ \}\qquad x := s \qquad \{\ Q[x]\ \}$

$$\left\{\ \begin{array}{l}\text{if } c \text{ then } P_1\\ \qquad\ \text{else } P_2\end{array}\ \right\}\qquad \begin{array}{l}\texttt{if } c \texttt{ then } P_1\, e_1\, Q\\ \qquad\ \ \texttt{else } P_2\, e_2\, Q\end{array}\qquad \{\ Q\ \}$$

$$\left\{\ \begin{array}{l}\text{if } c \text{ then } P\\ \qquad\ \text{else } Q\end{array}\ \right\}\qquad \texttt{if } c \texttt{ then } P\, e\, Q\qquad \{\ Q\ \}$$

? $\qquad \texttt{while } c \texttt{ do } e \qquad \{\ Q\ \}$

$$
\begin{array}{rcl}
\mathrm{WP}(\mathtt{skip}, Q) & \equiv & Q \\[1em]
\mathrm{WP}(x := t, Q) & \equiv & Q[x \mapsto t] \\[1em]
\mathrm{WP}(e_1 \;;\; e_2, Q) & \equiv & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)) \\[1em]
\mathrm{WP}(\mathtt{if}\ t\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, Q) & \equiv & (t \implies \mathrm{WP}(e_1, Q)) \wedge \\
 & & (\neg t \implies \mathrm{WP}(e_2, Q))
\end{array}
$$

```
if odd q then r := r + p;
p := p + p;
q := half q
```

```
if odd q then
    r := r + p
else
    skip;
p := p + p;
q := half q
```

```
if odd q then
    r := r + p
else
    skip;
p := p + p;
q := half q
```

$Q[p, q, r]$

```
if odd q then
    r := r + p
else
    skip;
p := p + p;
```

$Q[p, \mathtt{half}\ q, r]$

```
q := half q
```

$Q[p, q, r]$

```
if odd q then
    r := r + p
else
    skip;
```

$Q[p+p, \texttt{half}\ q, r]$

$p := p + p$;

$Q[p, \texttt{half}\ q, r]$

$q := \texttt{half}\ q$

$Q[p, q, r]$

```
if odd q then
    r := r + p
  Q[p + p, half q, r]
else
    skip ;
  Q[p + p, half q, r]
  p := p + p ;
Q[p, half q, r]
  q := half q
Q[p, q, r]
```

$$
\begin{array}{l}
\quad \textbf{if}\ \texttt{odd}\ q\ \textbf{then} \\
\qquad Q[p+p, \texttt{half}\ q, r+p] \\
\qquad\quad r := r\ \texttt{+}\ p \\
\qquad Q[p+p, \texttt{half}\ q, r] \\
\quad \textbf{else} \\
\qquad Q[p+p, \texttt{half}\ q, r] \\
\qquad\quad \texttt{skip}\,; \\
\qquad Q[p+p, \texttt{half}\ q, r] \\
\quad p := p\ \texttt{+}\ p\,; \\
Q[p, \texttt{half}\ q, r] \\
\quad q := \texttt{half}\ q \\
Q[p, q, r]
\end{array}
$$

$$(\texttt{odd}\ q \rightarrow Q[p+p, \texttt{half}\ q, r+p]) \land$$
$$(\neg\ \texttt{odd}\ q \rightarrow Q[p+p, \texttt{half}\ q, r])$$

if odd q **then**

$Q[p+p, \texttt{half}\ q, r+p]$

$r := r$ + p

$Q[p+p, \texttt{half}\ q, r]$

else

$Q[p+p, \texttt{half}\ q, r]$

skip ;

$Q[p+p, \texttt{half}\ q, r]$

$p := p$ + p ;

$Q[p, \texttt{half}\ q, r]$

$q :=$ half q

$Q[p, q, r]$

Consider the following program, lets call it P:

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

1. Compute $\mathrm{WP}(P, z == \mathtt{max}(x,y))$.
2. Is the following triple

$$\{\mathtt{true}\}\,\mathtt{P}\,\{z == \mathtt{max}(x,y)\}$$

   valid?
3. What about

$$\{x \geq 0 \wedge y \geq 0\}\,\mathtt{P}\,\{z == \mathtt{max}(x,y)\}$$

   ? The answer comes in a few minutes.

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

$x_1 \dots x_k$ references modified in e

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:

$$
\begin{array}{l}
\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{do}\ e\ \texttt{done}, Q) \equiv \\
\quad J \land \\
\quad \forall x_1 \ldots x_k. \\
\qquad (J \land t \rightarrow \mathrm{WP}(e, J)) \land \\
\qquad (J \land \neg t \rightarrow Q)
\end{array}
$$

the given invariant J
holds at the loop entry,
is preserved after
a single iteration,
and suffices to prove Q

$x_1 \ldots x_k$ references modified in e

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 invariant J[p,q,r] do
    if odd q then r := r + p ;
    p := p + p ;
    q := half q
done ;
r
```

result $= a * b$

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 invariant J[p,q,r] do
    if odd q then r := r + p;
    p := p + p;
    q := half q
done;
```

$r = a * b$

```
r
```

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 invariant J[p,q,r] do
    if odd q then r := r + p ;
    p := p + p ;
    q := half q
  J[p, q, r]
done ;
r = a ∗ b
r
```

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 invariant J[p,q,r] do
    (odd q → J[p + p, half q, r + p]) ∧
  (¬ odd q → J[p + p, half q, r])
    if odd q then r := r + p ;
    p := p + p ;
    q := half q
  J[p, q, r]
done ;
r = a * b
r
```

```
  let ref p = a in
  let ref q = b in
  let ref r = 0 in
J[p,q,r] ∧
∀p q r. J[p,q,r] →
  (q > 0 →
      (odd q → J[p + p, half q, r + p]) ∧
    (¬ odd q → J[p + p, half q, r])) ∧
  (q ⩽ 0 →
    r = a ∗ b)
  while q > 0 invariant J[p,q,r] do
      if odd q then r := r + p;
      p := p + p;
      q := half q
  done;
  r
```

$$
\begin{array}{l}
J[a,b,0] \wedge \\
\forall p\, q\, r.\; J[p,q,r] \rightarrow \\
\quad (q > 0 \rightarrow \\
\qquad\quad (\texttt{odd}\ q \rightarrow J[p+p, \texttt{half}\ q, r+p]) \wedge \\
\qquad (\neg\ \texttt{odd}\ q \rightarrow J[p+p, \texttt{half}\ q, r])) \wedge \\
\quad (q \leqslant 0 \rightarrow \\
\qquad r\ =\ a * b)
\end{array}
$$

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 invariant J[p,q,r] do
    if odd q then r := r + p;
    p := p + p;
    q := half q
done;
r
```

Theorem

For any e and Q, the triple $\{\mathrm{WP}(e, Q)\}\, e\, \{Q\}$ is valid.

Can be proved by induction on the structure of the program e w.r.t. some reasonable semantics (axiomatic, operational, etc.)

Corollary

To show that $\{P\}\, e\, \{Q\}$ is valid, it suffices to prove $P \implies \mathrm{WP}(e, Q)$.
(it looks like the $\textsc{Consequence}$ *rule from Hoare Logic, right?)*

This is what $\textsc{Why3}$ does.

Weakest Pre-condition Calculus – Example, again

Consider the following program, lets call it P:

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

1. Compute $\mathrm{WP}(P, z == \mathtt{max}(x,y))$.
2. Is the following triple

$$\{\mathtt{true}\}\,\mathtt{P}\,\{z == \mathtt{max}(x,y)\}$$

   valid?
3. What about

$$\{x \geq 0 \land y \geq 0\}\,\mathtt{P}\,\{z == \mathtt{max}(x,y)\}$$

   ?

Consider the following program, lets call it S:

```
x := y;
y := w;
w := x;
```

1. Using weakest pre-condition calculus, prove the following triple

$$\{P(y) \land Q(w)\}\, S\, \{P(w) \land Q(y)\}$$

   is valid. (here, P and Q are any properties)