

Software Verification

Master Programme in Computer Science

Mário Pereira `mjp.pereira@fct.unl.pt`

Nova School of Science and Technology, Portugal

October 14, 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:

$$\overline{\{P\} \text{ skip } \{P\}}$$

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

$$\frac{\{P\} e_1 \{Q\} \quad \{Q\} e_2 \{R\}}{\{P\} e_1 ; e_2 \{R\}}$$

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

Consequence rule:

$$\frac{\models P \Rightarrow P' \quad \{P'\} e \{Q'\} \quad \models Q' \Rightarrow Q}{\{P\} e \{Q\}}$$

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

$$\frac{\models x = 1 \rightarrow x + 2 = 3 \quad \begin{array}{c} (x = 3)[x \mapsto x + 2] \equiv x + 2 = 3 \\ \vdots \\ \{x + 2 = 3\} x := x + 2 \{x = 3\} \end{array}}{\{x = 1\} x := x + 2 \{x = 3\}}$$

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.

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

What is nice here:

- the program in the conclusion contains the subprograms P_1, \dots, 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

Procedures and method calls – Recap

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

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.

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

Procedures and method calls – Recap

Method calls built into a form of assignment:

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

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

Each **method call** is validated as follows:

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

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

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

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

Procedures and method calls – Recap

Method calls built into a form of assignment:

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

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

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

Verification with method calls is **modular**.

Rules for **if** and **while**:

$$\frac{\{P \wedge t\} e_1 \{Q\} \quad \{P \wedge \neg t\} e_2 \{Q\}}{\{P\} \text{if } t \text{ then } e_1 \text{ else } e_2 \{Q\}}$$

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

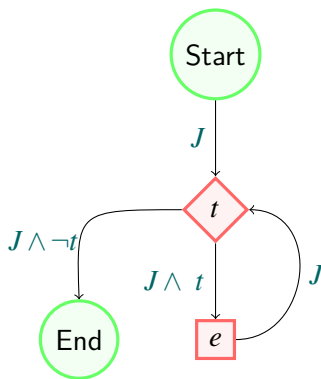
Formula J is a **loop invariant**.

Finding a right invariant is a **major difficulty**.

Loops in Hoare Logic

Hoare Logic Rule for Loops – In a Picture

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



Hoare Logic Rule for Loops – In Words

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

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 = Invariant Condition

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

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**.

Hoare Logic Rule for Loops

J = Invariant Condition

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

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 = Invariant Condition

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

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 {  
     $i := i + 1;$   
}  
 $\{i == n\}$ 
```

$\{0 \leq n\}$

$i := 0;$

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

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

while $i < n$ **do** {

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

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

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

$i := i + 1;$

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

}

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

$\{i == n\}$

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?

Loop Invariants – Challenge

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?

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

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

}

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

$\{0 \leq n\}$

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**;

}

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

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

Checking Loop Invariants

```
{0 ≤ n}
s := 0;
{s == 0 ∧ 0 ≤ n}
i := 0;
{i == 0 ∧ s == 0 ∧ 0 ≤ n}
{s == 0 ∧ 0 ≤ i ≤ n}
{s == ∑j=0i j ∧ 0 ≤ i ≤ n}    //invariant initially holds
while i < n do {
    {s == ∑j=0i j ∧ 0 ≤ i ≤ n ∧ i < n} //invariant holds at the beginning of iteration
    i := i + 1;
    s := s + i;
    {s == ∑j=0i j ∧ 0 ≤ i ≤ n} //invariant is maintained
}
{i == n ∧ s == ∑j=0i j}
{s == ∑j=0n 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

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

s := s + i;

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

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

Checking Loop Invariants – the big picture

```
{0 ≤ n}
s := 0;
{s == 0 ∧ 0 ≤ n}
i := 0;
{i == 0 ∧ s == 0 ∧ 0 ≤ n}
{s == 0 ∧ 0 ≤ i ≤ n}
{s == ∑j=0i j ∧ 0 ≤ i ≤ n}    //invariant initially holds
while i < n do {
    {s == ∑j=0i j ∧ 0 ≤ i ≤ n ∧ i < n} //invariant holds at the beginning of iteration
    i := i + 1;
    s := s + i;
    {s == ∑j=0i j ∧ 0 ≤ i ≤ n} //invariant is maintained
}
{i == n ∧ s == ∑j=0i j}
{s == ∑j=0n j}
```

Checking Loop Invariants – the big picture

```
{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 ==  $\sum_{j=0}^{i-1} j \wedge 0 \leq i-1 \leq n \wedge i-1 < n$ }
  {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 ∧ 0 ≤ i ≤ n} //invariant is restored
  {s ==  $\sum_{j=0}^i j \wedge 0 \leq i \leq n$ } //invariant is maintained
}
```

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

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

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

e expression

Q postcondition

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

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

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

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

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

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

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

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

if c **then** e_1 $\{ Q \}$
else e_2

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

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

if c **then** $e_1 Q$ $\{ Q \}$
else $e_2 Q$

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

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

$$\begin{array}{l} \text{if } c \text{ then } P_1 e_1 Q \\ \text{else } P_2 e_2 Q \end{array} \quad \{ Q \}$$

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

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

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

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

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

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

$\text{if } c \text{ then } e \quad \{ Q \}$

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

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

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

$\text{if } c \text{ then } P e Q \quad \{ Q \}$

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

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

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

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

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

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

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

$\{ \text{if } c \text{ then } P \text{ else } Q \} \quad \text{if } c \text{ then } P e Q \quad \{ Q \}$

$\text{while } c \text{ do } e \quad \{ Q \}$

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

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

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

$\{ \text{if } c \text{ then } P \text{ else } Q \} \quad \text{if } c \text{ then } P e Q \quad \{ Q \}$

$? \quad \text{while } c \text{ do } e \quad \{ Q \}$

$$\text{WP}(\text{skip}, Q) \equiv Q$$

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

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

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

Swimming up the waterfall

```
if odd  $q$  then  $r := r + p$ ;  
 $p := p + p$ ;  
 $q := \text{half } q$ 
```

if odd q then

$r := r + p$

else

skip;

$p := p + p;$

$q := \text{half } q$

if odd q then

$r := r + p$

else

skip;

$p := p + p;$

$q := \text{half } q$

$Q[p, q, r]$

if odd q then

$r := r + p$

else

skip;

$p := p + p;$

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

$q := \text{half } q$

$Q[p, q, r]$

if odd q then

$r := r + p$

else

skip;

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

$p := p + p;$

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

$q := \text{half } q$

$Q[p, q, r]$

if odd q then

$r := r + p$

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

else

skip;

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

$p := p + p;$

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

$q := \text{half } q$

$Q[p, q, r]$


```
if odd  $q$  then
   $Q[p + p, \text{half } q, r + p]$ 
   $r := r + p$ 
   $Q[p + p, \text{half } q, r]$ 
else
   $Q[p + p, \text{half } q, r]$ 
  skip;
   $Q[p + p, \text{half } q, r]$ 
 $p := p + p;$ 
 $Q[p, \text{half } q, r]$ 
 $q := \text{half } q$ 
 $Q[p, q, r]$ 
```

```
(odd  $q \rightarrow Q[p + p, \text{half } q, r + p]$ )  $\wedge$   
( $\neg$  odd  $q \rightarrow Q[p + p, \text{half } q, r]$ )  
  if odd  $q$  then  
     $Q[p + p, \text{half } q, r + p]$   
     $r := r + p$   
     $Q[p + p, \text{half } q, r]$   
  else  
     $Q[p + p, \text{half } q, r]$   
    skip;  
     $Q[p + p, \text{half } q, r]$   
   $p := p + p$ ;  
   $Q[p, \text{half } q, r]$   
   $q := \text{half } q$   
   $Q[p, q, r]$ 
```

Weakest Pre-condition Calculus – Example

Consider the following program, lets call it P:

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

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

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

valid?

3. What about

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

? The answer comes in a few minutes.

$\text{WP}(\text{while } t \text{ do } e, Q) \equiv$

$\exists J : \text{Prop.}$

$J \wedge$

$\forall x_1 \dots x_k.$

$(J \wedge t \rightarrow \text{WP}(e, J)) \wedge$

$(J \wedge \neg t \rightarrow Q)$

some *invariant property* J

that holds at the loop entry

and is preserved

after a single iteration,

is strong enough to prove Q

$x_1 \dots x_k$ 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

Definition of WP: annotated loops

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**:

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

$x_1 \dots x_k$ references modified in e

Russian Peasant Multiplication

```
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  $r := r + p$ ;
   $p := p + p$ ;
   $q := \text{half } q$ 
done;
 $r$ 
result =  $a * b$ 
```

Russian Peasant Multiplication

```
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  $r := r + p$ ;
   $p := p + p$ ;
   $q := \text{half } q$ 
done;
 $r = a * b$ 
 $r$ 
```

Russian Peasant Multiplication

```
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  $r := r + p$ ;
   $p := p + p$ ;
   $q := \text{half } q$ 
   $J[p, q, r]$ 
done;
 $r = a * b$ 
 $r$ 
```


Russian Peasant Multiplication

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 invariant  $J[p, q, r]$  do
  ( $\text{odd } q \rightarrow J[p + p, \text{half } q, r + p]$ )  $\wedge$ 
  ( $\neg \text{odd } q \rightarrow J[p + p, \text{half } q, r]$ )
  if odd q then  $r := r + p$ ;
   $p := p + p$ ;
   $q := \text{half } q$ 
   $J[p, q, r]$ 
done;
 $r = a * b$ 
 $r$ 
```

Russian Peasant Multiplication

```
let ref p = a in
let ref q = b in
let ref r = 0 in
J[p,q,r] ∧
∀pqr. 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
```

Russian Peasant Multiplication

$J[a, b, 0] \wedge$
 $\forall pqr. J[p, q, r] \rightarrow$
 $(q > 0 \rightarrow$
 $(\text{odd } q \rightarrow J[p + p, \text{half } q, r + p]) \wedge$
 $(\neg \text{odd } q \rightarrow J[p + p, \text{half } q, r])) \wedge$
 $(q \leq 0 \rightarrow$
 $r = a * b)$
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 $\text{odd } q$ **then** $r := r + p;$
 $p := p + p;$
 $q := \text{half } q$
done;
 r

Theorem

For any e and Q , the triple $\{\text{WP}(e, Q)\} e \{Q\}$ 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 $P \implies \text{WP}(e, Q)$.
(it looks like the CONSEQUENCE rule from Hoare Logic, right?)*

This is what 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 $\text{WP}(P, z == \max(x, y))$.
2. Is the following triple

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

valid?

3. What about

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

?

Weakest Pre-condition Calculus – Exercise

Consider the following program, lets call it S:

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

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

$$\{P(y) \wedge Q(w)\} S \{P(w) \wedge Q(y)\}$$

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