

Programming with Proofs

Verifying Imperative Programs

Copyright 2020-22, Graeme Smith and Cesare Tinelli.

Produced by Cesare Tinelli at the University of Iowa from notes originally developed by Graeme Smith at the University of Queensland. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Hoare triples

For predicates P and Q and program S , the *Hoare triple*



states the following:

*if S is started in any state that satisfies P ,
then S will not crash (or do other bad things) and
will terminate in some state satisfying Q*

Examples:

$\{ x == 1 \}$	$x := 20$	$\{ x == 20 \}$
$\{ x < 18 \}$	$y := 18 - x$	$\{ y \geq 0 \}$
$\{ x < 18 \}$	$y := 5$	$\{ y \geq 0 \}$

Non-example: $\{ x < 18 \} x := y \{ y \geq 0 \}$

Reducing an Imperative Program to a Formula

```
{ true }  
// Check that this implication is valid (negation is unsatisfiable)  
{ (a  $\Rightarrow$  (b  $\Rightarrow$  (true  $\Leftrightarrow$  (a  $\Rightarrow$  b)))  $\wedge$  ( $\neg$ b  $\Rightarrow$  (false  $\Leftrightarrow$  (a  $\Rightarrow$  b))))  $\vee$  ( $\neg$ a  $\Rightarrow$  (true  $\Leftrightarrow$  (a  $\Rightarrow$  b))) }  
if (a) {  
  { (b  $\Rightarrow$  (true  $\Leftrightarrow$  (a  $\Rightarrow$  b)))  $\wedge$  ( $\neg$ b  $\Rightarrow$  (false  $\Leftrightarrow$  (a  $\Rightarrow$  b))) }  
    if (b) {  
      { true  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
        res := true  
      { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
    } else {  
      { false  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
        res := false  
      { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
    }  
  { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
} else {  
  { true  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
    res := true  
  { res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }  
}  
{ res  $\Leftrightarrow$  (a  $\Rightarrow$  b) }
```



Forward reasoning

Constructing a postcondition from a given precondition

In general, there are many possible postconditions

Examples:

1. $\{ x == 0 \} \quad y := x + 3 \quad \{ y < 100 \}$

2. $\{ x == 0 \} \quad y := x + 3 \quad \{ x == 0 \}$

3. $\{ x == 0 \} \quad y := x + 3 \quad \{ 0 \leq x \ \&\& \ y == 3 \}$

4. $\{ x == 0 \} \quad y := x + 3 \quad \{ 3 \leq y \}$

5. $\{ x == 0 \} \quad y := x + 3 \quad \{ \text{true} \}$

Strongest postcondition

Forward reasoning constructs the **strongest** (i.e., most specific) postcondition

$$\{ x == 0 \} \quad y := x + 3 \quad \{ 0 == x \ \&\& \ y == 3 \}$$

Def: A is *stronger* than B if $A \implies B$ is a valid formula

Def: A formula is *valid* if it is true for any valuation of its free variables

Backward reasoning

Construct a precondition for a given postcondition

Again, there are many preconditions

Examples:

1. $\{ x \leq 70 \} \quad y := x + 3 \quad \{ y \leq 80 \}$
2. $\{ x == 65 \ \&\& \ y < 21 \} \quad y := x + 3 \quad \{ y \leq 80 \}$
3. $\{ x \leq 77 \} \quad y := x + 3 \quad \{ y \leq 80 \}$
4. $\{ x*x + y*y \leq 2500 \} \quad y := x + 3 \quad \{ y \leq 80 \}$
5. $\{ \text{false} \} \quad y := x + 3 \quad \{ y \leq 80 \}$

Weakest precondition

Backward reasoning constructs the **weakest** (i.e., most general) precondition

$$\{ x \leq 77 \} \quad y := x + 3 \quad \{ y \leq 80 \}$$

Def: A is *weaker* than B if $B \implies A$ is a valid formula

Weakest precondition for assignment

Given $\{ \textcolor{brown}{?} \} \textcolor{brown}{x} \textcolor{blue}{:=} \textcolor{brown}{E} \{ \textcolor{brown}{Q} \}$

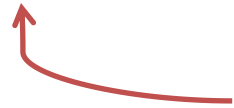
we construct $\textcolor{brown}{?}$ by replacing each x in $\textcolor{brown}{Q}$ with $\textcolor{brown}{E}$ (denoted by $\textcolor{brown}{Q}[\textcolor{brown}{x} \backslash \textcolor{brown}{E}]$)

Weakest precondition for assignment

Given

$$\{Q[x \setminus E]\} \ x := E \ \{ Q \}$$

Examples: $\{ ? \} \ y := a + b \ \{ 25 \leq y \}$


$$25 \leq a + b$$

1. $\{ 25 \leq x + 3 + 12 \} \ a := x + 3 \ \{ 25 \leq a + 12 \}$

2. $\{ x + 1 \leq y \} \ x := x + 1 \ \{ x \leq y \}$

3. $\{ 3 * 2 * x + 5 * y < 100 \} \ x := 2 * x \ \{ 3 * x + 5 * y < 100 \}$

Swap example

```
var tmp := x;
```

```
x := y;
```

```
y := tmp;
```

Swap example

```
{ x == X && y == Y }
```

```
var tmp := x;
```

```
x := y;
```

```
y := tmp;
```

```
{ x == Y && y == X }
```

The initial values of x and y are specified using **logical variables** X and Y

Swap example

```
{ x == X && y == Y }  
{ ? }  
var tmp := x;  
{ ? }  
x := y;  
{ ? }  
y := tmp;  
{ x == Y && y == X }
```

The initial values of x and y are specified using **logical variables** X and Y

Swap example

```
{ x == X && y == Y }  
{ ? }  
var tmp := x;  
{ ? }  
x := y;  
{ x == Y && tmp == X }  
y := tmp;  
{ x == Y && y == X }
```

Swap example

```
{ x == X && y == Y }  
{ ? }  
var tmp := x;  
{ y == Y && tmp == X }  
x := y;  
{ x == Y && tmp == X }  
y := tmp;  
{ x == Y && y == X }
```

Swap example

```
{ x == X && y == Y }  
{ y == Y && x == X }  
var tmp := x;  
{ y == Y && tmp == X }  
x := y;  
{ x == Y && tmp == X }  
y := tmp;  
{ x == Y && y == X }
```

Swap example

```
{ x == X && y == Y }  
{ y == Y && x == X }  
var tmp := x;  
{ y == Y && tmp == X }  
x := y;  
{ x == Y && tmp == X }  
y := tmp;  
{ x == Y && y == X }
```

The final step is the *proof obligation* that

$$(x == X \ \&\& \ y == Y) ==> (y == Y \ \&\& \ x == X)$$

is valid

Program-proof bookkeeping

{ x == X && y == Y }

x := y - x;

y := y - x;

x := y + x;

{ x == Y && y == X }

Program-proof bookkeeping

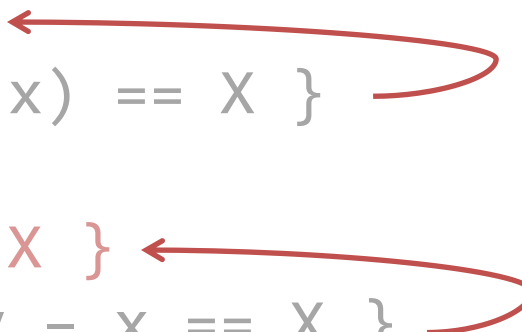
```
{ x == X && y == Y }  
{ y - (y - x) + (y - x) == Y && y - (y - x) == X }  
x := y - x;  
{ y - x + x == Y && y - x == X }  
y := y - x;  
{ y + x == Y && y == X }  
x := y + x;  
{ x == Y && y == X }
```

The constructed precondition simplifies to

$y == Y \ \&\& \ x == X$

Program-proof bookkeeping

```
{ x == X && y == Y }  
{ y == Y && x == X } ←  
{ y == Y && y - (y - x) == X }  
x := y - x;  
{ y == Y && y - x == X } ←  
{ y - x + x == Y && y - x == X }  
y := y - x;  
{ y + x == Y && y == X }  
x := y + x;  
{ x == Y && y == X }
```



We are also allowed to **strengthen** the conditions as we work backwards (but not weaken them!)

What about strongest postconditions?

Consider $\{ w < x \ \&\& \ x < y \} \ x := 100 \ \{ ? \}$

Obviously, $x == 100$ is a postcondition, but it is **not** the strongest

Something **more** is implied by the precondition:

there exists an n such that $w < n \ \&\& \ n < y$

which is equivalent to saying that $w + 1 < y$

In general:

$$\{ P \} \ x := E \ \{ \text{exists } n :: P[x \backslash n] \ \&\& \ x == E[x \backslash n] \}$$

WP and SP

Let P be a predicate on the pre-state of a program S and let Q be a predicate on the post-state of S

$WP [S, Q]$ denotes the weakest precondition of S wrt Q

$SP [S, P]$ denotes the strongest postcondition of S wrt P

$$WP [x := E, Q] = Q[x \backslash E]$$

$$SP [x := E, P] = \text{exists } n :: P[x \backslash n] \ \&\& \ x == E[x \backslash n]$$

Control flow

Until now:

Assignment: `x := E`

Variable introduction: `var x`

Next:

Sequential composition: `S ; T`

Conditions: `if B { S } else { T }`

Method calls: `r := M(E)`

Later:

Loops: `while B { S }`

Sequential composition

$$\begin{aligned} S ; T & \quad \{ P \} S \{ Q \} T \{ R \} \\ & \quad \{ P \} S \{ Q \} \text{ and } \{ Q \} T \{ R \} \end{aligned}$$

Strongest postcondition

$$\text{let } Q = \text{SP}[S, P]$$

$$\text{SP}[S; T, P] = \text{SP}[T, Q] = \text{SP}[T, \text{SP}[S, P]]$$

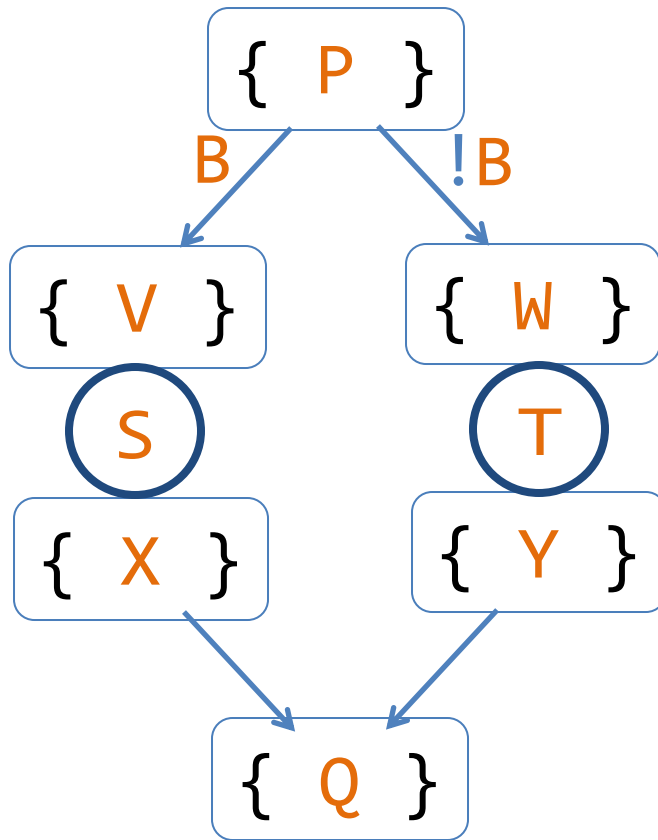
Weakest precondition

$$\text{let } Q = \text{WP}[T, R]$$

$$\text{WP}[S; T, R] = \text{WP}[S, Q] = \text{WP}[S, \text{WP}[T, R]]$$

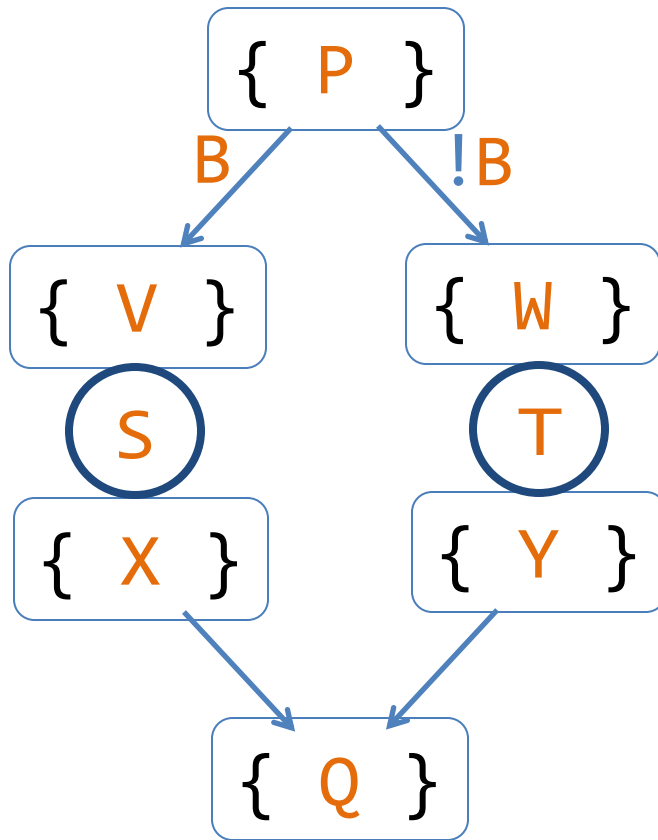
Conditional control flow

```
if B { S } else { T }
```



Conditional control flow

```
if B { S } else { T }
```

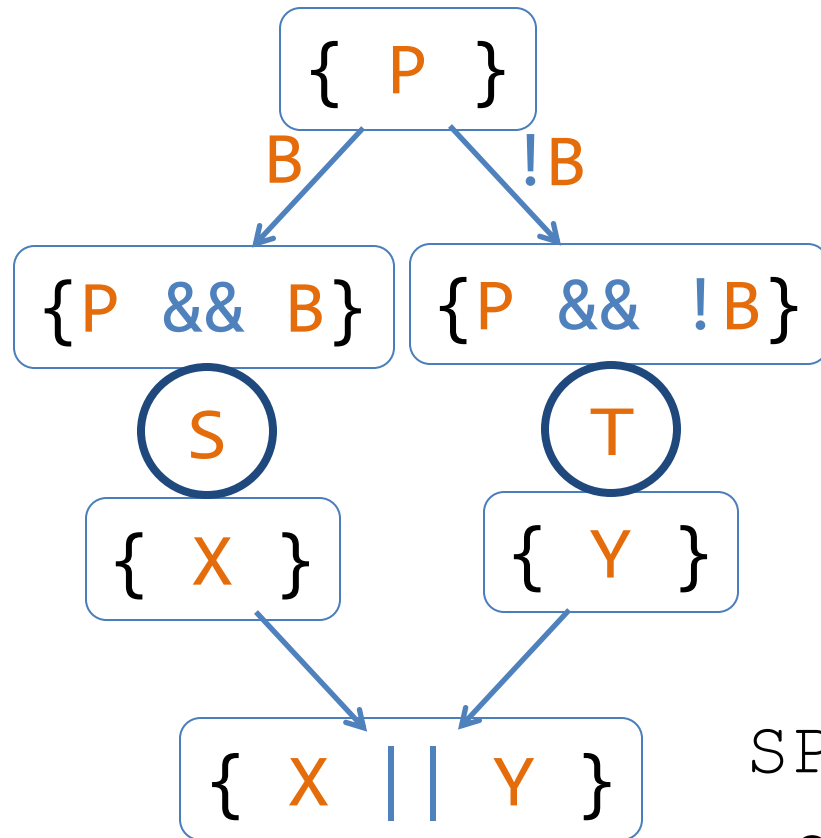


Floyd-Hoare logic tells us:

1. $P \ \&\& \ B \ ==> \ V$
2. $P \ \&\& \ !B \ ==> \ W$
3. $\{ V \} \ S \ \{ X \}$
4. $\{ W \} \ T \ \{ Y \}$
5. $X \ ==> \ Q$
6. $Y \ ==> \ Q$

Strongest postcondition

`if B { S } else { T }`



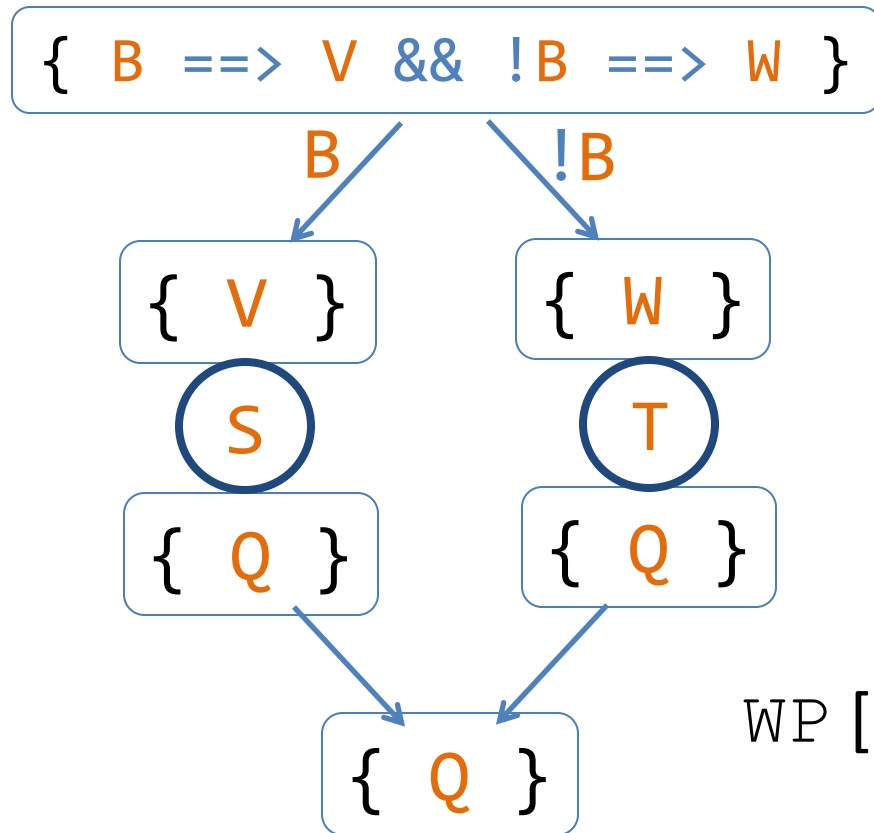
$$X = \text{SP} [S, P \ \&\& \ B]$$

$$Y = \text{SP} [T, P \ \&\& \ !B]$$

$$\begin{aligned} \text{SP} [\text{if } B \{ S \} \text{ else } \{ T \}, P] = \\ \text{SP} [S, P \ \&\& \ B] \parallel \text{SP} [T, P \ \&\& \ !B] \end{aligned}$$

Weakest precondition

`if B { S } else { T }`



$$V = \text{WP} [S, Q]$$

$$W = \text{WP} [T, Q]$$

$$\begin{aligned} \text{WP} [\text{if } B \{ S \} \text{ else } \{ T \}, Q] = \\ (B \Rightarrow \text{WP} [S, Q]) \ \&\& \\ (!B \Rightarrow \text{WP} [T, Q]) \end{aligned}$$

Weakest precondition (example)

```
if x < 3 {  
  
    x, y := x + 1, 10;  
  
} else {  
  
    y := x;  
  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {  
  
    x, y := x + 1, 10;  
  
} else {  
  
    y := x;  
    { x + y == 100 }  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {  
  
    x, y := x + 1, 10;  
  
} else {  
  
    { x + x == 100 }  
    y := x;  
    { x + y == 100 }  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {  
  
    x, y := x + 1, 10;  
  
} else {  
    { x == 50 }  
    { x + x == 100 }  
    y := x;  
    { x + y == 100 }  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {  
    { x == 89 }  
    { x + 1 + 10 == 100 }  
    x, y := x + 1, 10;  
    { x + y == 100 }  
} else {  
    { x == 50 }  
    { x + x == 100 }  
    y := x;  
    { x + y == 100 }  
}  
{ x + y == 100 }
```


Weakest precondition (example)

```
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }  
if x < 3 {  
    { x == 89 }  
    { x + 1 + 10 == 100 }  
    x, y := x + 1, 10;  
    { x + y == 100 }  
} else {  
    { x == 50 }  
    { x + x == 100 }  
    y := x;  
    { x + y == 100 }  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
{ x == 50 } { (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }  
  if x < 3 {  
    { x == 89 }  
    { x + 1 + 10 == 100 }  
    x, y := x + 1, 10;  
    { x + y == 100 }  
  } else {  
    { x == 50 }  
    { x + x == 100 }  
    y := x;  
    { x + y == 100 }  
  }  
  { x + y == 100 }
```

Refresher: Implication properties

$A \implies B$ equiv. to $\neg A \vee B$

Hence,

$A \implies \text{true}$	equiv. to	true
$A \implies \text{false}$	"	$\neg A$
$\text{true} \implies B$	"	B
$\text{false} \implies B$	"	true

Useful law for simplifying predicates

$A \implies (B \implies C)$ equiv. to $(A \wedge B) \implies C$

Weakest precondition (example)

```
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }  
if x < 3 {  
    x, y := x + 1, 10;  
} else {  
    y := x;  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }  
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }  
if x < 3 {  
    x, y := x + 1, 10;  
} else {  
    y := x;  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
{ (x >= 3 && x < 3) || (x >= 3 && x == 50) ||  
  (x == 89 && x < 3) || (x == 89 && x == 50) }  
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }  
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }  
if x < 3 {  
    x, y := x + 1, 10;  
} else {  
    y := x;  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
{ false || x == 50 || false || false }  
{ (x >= 3 && x < 3) || (x >= 3 && x == 50) ||  
  (x == 89 && x < 3) || (x == 89 && x == 50) }  
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }  
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }  
if x < 3 {  
    x, y := x + 1, 10;  
} else {  
    y := x;  
}  
{ x + y == 100 }
```

Weakest precondition (example)

```
{ x == 50 }
{ false || x == 50 || false || false }
{ (x >= 3 && x < 3) || (x >= 3 && x == 50) ||
  (x == 89 && x < 3) || (x == 89 && x == 50) }
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
    x, y := x + 1, 10;
} else {
    y := x;
}
{ x + y == 100 }
```


Demo in Velvet

```
method IfElse (mut x : Int) (mut y : Int) return (res: Int)
  require x = 50
  ensures x + y = 100 do
    if x < 3 then
      x := x + 1
      y := 10
    else
      y := x
    return x

prove_correct IfElse by
  loom_solve
```

Method correctness

Given

```
method M(x: Tx) return (y: Ty)  
  require P  
  ensures Q  
{  
  B  
}
```

we need to prove

$$P \implies \text{WP}[B, Q]$$

Method calls

Methods are *opaque*, i.e., *we reason in terms of their specifications, not* their implementations

Given

```
method Triple(x: int) return (y: int)
  ensures y == 3 * x
```

we expect to be able to prove, for instance, the following method call

```
{ true } v := Triple(u + 4) { v == 3 * (u + 4) }
```

Parameters

We need to **relate** the **actual** parameters (of the method call) with the **formal** parameters (of the method)

To avoid any name clashes, we first **rename** the formal parameters to **fresh** variables:

```
method Triple(x': int) return (y': int)
  ensures y' == 3 * x'
```

Then, for a call `v := Triple(u + 1)` we have

```
x' := u + 1
v  := y'
```

Assumptions

The caller can assume that the method's postcondition holds

We introduce a **new statement**, **assume E**, to capture this

$$\text{SP} [\text{assume } E, P] = E \ \&\& \ P$$

$$\text{WP} [\text{assume } E, Q] = E \implies Q$$

The semantics of $v := \text{Triple}(u + 1)$ is then given by

```
var x'; var y';  
x' := u + 1;  
assume y' == 3 * x';  
v := y'
```

```
method Triple(x': int)  
returns (y': int)  
ensures y' == 3 * x'
```

Weakest precondition

$$\text{WP } [r := M(E), Q] = \text{forall } y' :: R[x, y \setminus E, y'] \implies Q[r \setminus y']$$

where x is M 's input, y is M 's output, and R is M 's postcondition

Example. Let Q be $v == 48$ for the method:

```
method Triple(x: int) returns (y: int)
    ensures y == 3 * x
```

```
{ u == 15 }
```

```
{ 3 * (u + 1) == 48 }
```

```
{ forall y' :: y' == 3 * (u + 1) ==> y' == 48 }
```

```
v := Triple(u + 1);
```

```
{ v == 48 }
```

Assertions

`assert E` does nothing when `E` holds,
otherwise it crashes the program

```
method Triple(x: int) return (r: int) do
  let y := 2 * x
  r := x + y
  assert r = 3 * x
```

$WP[\text{assert } E, Q] = E \ \&\& \ Q$

$SP[\text{assert } E, P] = P \ \&\& \ E$

Method calls with preconditions

Given

```
method M(x: X) return (y: Y)
  require P
  ensures R
```

The semantics of $r := M(E)$ is

```
var xE ; var yr ;
xE := E ;
assert P[x\ xE] ;
assume R[x, y\ xE, yr] ;
r := yr
```

$$\text{WP}[r := M(E), Q] = P[x \setminus E] \ \&\& \ \text{forall } y_r :: R[x, y \setminus E, y_r] \implies Q[r \setminus y_r]$$

Demo in Velvet

```
method Triple (x: Int) return (r: Int)
```

```
  ensures r = 3 * x
```

```
  do
```

```
    let y := 2 * x
```

```
    let mut r := x + y
```

```
    return r
```

```
prove_correct Triple by
```

```
  loom_solve
```

```
method TestTriple (x : Int) return (u : Unit) do
```

```
  -- Note the syntax for method calls
```

```
  let r : Int ← Triple x
```

```
  assert r = x * 3
```

```
prove_correct TestTriple by
```

```
  loom_solve
```

Loops in Velvet

```
while G
  invariant J
  decreases M
do
  Body
```

G: *loop guard*, Boolean expression

M: (optional) *termination measure*, expression whose value is expected to decrease at each loop iteration

J: *loop invariant*, condition expected to hold at each iteration

Hoare triples for loops

```
{ J }  
while G  
    invariant J  
{ J && !G }
```

Example

```
r := 0;  
N := 104;  
while (r+1)*(r+1) <= N  
    invariant 0 <= r && r*r <= N  
assert 0 <= r && r*r <= N < (r+1)*(r+1);
```

Floyd-Hoare logic for loop body

For a loop

```
while G
  invariant J
do
  Body
```

we need to prove

```
{ J && G }
Body
{ J }
```

Quotient modulus

```
x := 0;  
y := 191;  
while !(y < 7)  
invariant 0 <= y && 7*x + y == 191  
assert x == 191 / 7 && y == 191 % 7;
```

Full program

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
    { 0 <= y && 7*x + y == 191 && 7 <= y }
    { 0 <= y - 7 && 7*x + 7 + (y - 7) == 191 }
    y := y - 7;
    { 0 <= y && 7*x + 7 + y == 191 }
    { 0 <= y && 7*(x + 1) + y == 191 }
    x := x + 1;
    { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Demo in Velvet

```
method QuotientModulus (mut x : Int) (mut y : Int) return (res: Int × Int)
  require x = 0 && y = 191
  do
    while !(y < 7)
      invariant 0 <= y && 7 * x + y == 191
      do
        y := y - 7
        x := x + 1
      assert x = 191 / 7
      assert y = 191 % 7
    return (x, y)
```

```
#eval (QuotientModulus 0 191).run
```

Leap to the answer

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
    { 0 <= y && 7 * x + y == 191 && 7 <= y }

    x := 27
    y := 2
    { 0 <= y && 7 * x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```


Leap to the answer

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
    { 0 <= y && 7 * x + y == 191 && 7 <= y }
    { true }
    { 0 <= 2 && 7 * 27 + 2 == 191 }
    x := 27
    y := 2
    { 0 <= y && 7 * x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Computing sums

```
while n != 33  
  invariant s == n * (n - 1) / 2  
  do  
    { s == n * (n - 1) / 2 && n != 33 }
```

```
{ s == n * (n - 1) / 2 }
```

```
assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
    { s == n * (n - 1) / 2 && n != 33 }
    { s == n * (n - 1) / 2 }
    s := s + n;
    { s = n * (n - 1) / 2 + n }
    { s = (n*n - n) / 2 + 2*n / 2 }
    { s == (n*n - n + 2*n) / 2 }
    { s == (n*n + n) / 2 }
    { s == (n + 1) * n / 2 }
    { s == (n + 1) * (n + 1 - 1) / 2 }
    n := n + 1;
    { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Full program

Need to choose initial values of s and n to establish invariant

```
s := 0;  
n := 0;  
while n != 33  
    invariant s == n * (n - 1) / 2  
    do  
        s := s + n;  
        n := n + 1;
```

Reasoning about Termination

Loop termination

For a loop

```
while G
  invariant J
  decreasing D
  do
    Body
```

we need to prove

```
{ J && G }
ghost var d := D;
Body
{ D < d && 0 <= D }
```

Ghost variables are for reasoning only. They are not part of the compiled code.



Termination of quotient modulus

```
x, y := 0, 191;  
while 7 <= y  
  invariant 0 <= y && 7 * x + y == 191  
  decreasing y  
do  
  y := y - 7;  
  x := x + 1;
```

```
{ 0 <= y && 7 * x + y == 191 && 7 <= y }
```

```
ghost var d := y;
```

```
y := y - 7;
```

```
x := x + 1;
```

```
{ d > y && y >= 0 }
```

- $y < d$ follows from $y := y - 7$

- $0 \leq d$ follows from $0 \leq y$ in invariant

Quick body

```
x, y := 0, 191;  
while 7 <= y  
  invariant 0 <= y && 7 * x + y == 191  
  decreasing y  
do  
  y := 2;  
  x := 27;
```

```
{ 0 <= y && 7 * x + y == 191 && 7 <= y }
```

```
ghost var d := y;
```

```
y := 2;
```

```
x := 27;
```

```
{ d > y && y >= 0 }
```

- $y < d$ follows from $7 \leq y$ in invariant
- $0 \leq d$ follows from $0 \leq y$ in invariant

Complete loop rule

```
{ J }  
while G  
  invariant J  
  decreasing D  
do  
  Body  
{ J && !G }
```

```
{ J && G }  
ghost var d := D;  
Body  
{ J && d > D && D >= 0 }
```

Integer square root

```
method SquareRoot(N: Nat) return (r: Nat)  
  ensures r*r <= N < (r+1)*(r+1)
```

Integer square root

```
method SquareRoot(N: Nat) return (r: Nat)
  ensures r*r <= N && N < (r+1)*(r+1)
```

Loop design pattern

For a postcondition $A \ \&\& \ B$, use
A as the invariant and $\neg B$ as the guard

do

```
while (r+1)*(r+1) <= N
  invariant r*r <= N
```

Integer square root

```
method SquareRoot(N: Nat) return (r: Nat)
  ensures r*r <= N && N < (r+1)*(r+1)
```

Loop design pattern

For a postcondition $A \ \&\& \ B$, use
 A as the invariant and $\neg B$ as the guard

```
do
  let mut r := 0;
  while (r+1)*(r+1) <= N
    invariant r*r <= N
  do
    r := r + 1
```

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration, add a **new variable** s and maintain **invariant**

$$s == (r + 1) * (r + 1)$$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add a new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

$$\{ s == (r + 1) * (r + 1) \}$$
$$\{ s == (r + 1 + 1) * (r + 1 + 1) \}$$
$$r := r + 1$$
$$\{ s == (r + 1) * (r + 1) \}$$

A more efficient algorithm

Rather than calculate $(r + 1) * (r + 1)$ on each iteration add a new variable $s == (r + 1) * (r + 1)$

Then we have s initially 1 , loop guard $s \leq N$ and invariant $s == (r + 1) * (r + 1)$

```
{ s == (r + 1) * (r + 1) }  
{ s + 2*r + 3 == (r + 1) * (r + 1) + 2*r + 3 }  
s := s + 2*r + 3;  
{ s == (r + 1) * (r + 1) + 2*r + 3 }  
{ s == r*r + 2*r + 1 + 2*r + 3 }  
{ s == r*r + 4*r + 4 }  
{ s == (r + 1 + 1) * (r + 1 + 1) }  
r := r + 1  
{ s == (r + 1) * (r + 1) }
```

Full program

```
method SquareRoot(N: Nat) return (r: Nat)
  ensures r*r <= N < (r+1)*(r+1)
do
  let mut r := 0;
  var s := 1;
  while s <= N
    invariant r*r <= N
    invariant s == (r+1)*(r+1)
  do
    s := s + 2*r + 3;
    r := r + 1;
```


More examples

Iterative Fibonacci

```
def Fib (n: Nat) := match n with
  | 0 => 0
  | 1 => 1
  | n + 2 => Fib (n + 1) + Fib n
```

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x = Fib(n)
  do
    x := 0;
    var i := 0;
    while i != n
      invariant 0 <= i <= n
      invariant x = Fib(i)
```

Iterative Fibonacci

Loop design technique (*Replace a constant by a variable*)

For a loop to establish a condition $P[E]$, where E is an expression that maintains a constant value throughout the loop,

- use a variable i that the loop changes until it equals E , and
- make $P[i]$ a loop invariant

Example: to establish $x = \text{Fib}(n)$ introduce i and

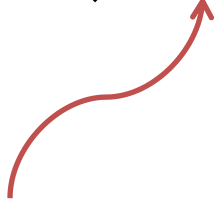
invariant $x = \text{Fib}(i)$

Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
  do
    let mut x := 0
    let mut y := 1
    let mut i := 0
    while i != n
      invariant 0 <= i && i <= n
      invariant x = Fib(i)
      do
        ...
        i := i + 1;
      }
  }
```

Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
  do
    let mut x := 0
    let mut y := 1
    let mut i := 0
    while i != n
      invariant 0 <= i && i <= n
      invariant x = Fib(i)
      invariant x == Fib(i) && y == Fib(i + 1)
      do
        ...
        i := i + 1;
```



Cannot use $y == \text{Fib}(i-1)$
as not defined when $i == 0$

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
    && y == Fib(i+1) && i != n}
```

```
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
    && y == Fib(i+1) && i != n}
{ 0 <= i <= n    && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && y == Fib(i+1)
    && x+y == Fib(i) + Fib(i+1) }
let tmp := x; x := y; y := tmp + y
{ 0 <= i+1 <= n && x == Fib(i+1)
    && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
    && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Full program

```
method ComputeFib(n: Nat) return (x: Nat)
  ensures x = Fib(n)
  do
    let mut x := 0
    let mut y := 1
    let mut i := 0;
    while i != n
      invariant 0 <= i && i <= n
      invariant x = Fib(i)
      invariant y = Fib(i + 1)
      do
        let tmp := x; x := y; y := tmp + y
        i := i + 1
```


Demo in Velvet

```
@[grind]
✓ def isMax (mx: Int) (arr: Array Int) :=
  | forall i, ( h: i < arr.size ) -> mx >= (arr[i]'h)

✓ method maxElem (arr: Array Int) return (res: Int)
  require arr.size > 0
  ensures isMax res arr
  do
    let mut i := 0
    let mut mx := arr[i]!
    i := i + 1
    while i < arr.size
      invariant 0 <= i ∧ i <= arr.size
      invariant forall j, j < i -> mx >= arr[j]!
      do
        let elem := arr[i]!
        if elem > mx then
          | mx := elem
        else
          | mx := mx
        i := i + 1
      return mx

✓ prove_correct maxElem by
  | loom_solve
```

What Have We Learned

- Theorems as Types
- Proofs as Programs
- Basics of Lean
- Inductive Definitions
- Proofs by Induction
- Relations as Predicates
- State Machines as Relations
- Stating Safety Properties
- Basics of SMT solvers
- Verifying State Machines in Veil
- Subset/Refinement Types
- Setoids and Quotient Types
- Defining $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$
- Hoare-Style Logic
- Weakest Preconditions
- Loop Invariants
- Reasoning about Arrays
- Verifying Programs in Velvet

What's Next

- Research Project:
 - Proving a cute mathematical theorem
 - Formalising a state Machine / distributed protocol in Veil
 - Verifying an algorithm/data structure in Velvet
- For inspiration:
 - Some ideas: <https://ilyasergey.net/PWP25/projects.html>
 - I'm happy to discuss project ideas!

And then what?

- Books and Resources:
 - *The Hitchhiker's Guide to Logical Verification* (2025)
 - Functional Programming in Lean
 - Theorem Proving in Lean 4
 - <https://leanprover.zulipchat.com/>
- Conferences to check out: POPL, PLDI, OOPSLA, CAV
 - PLMW: <https://www.sigplan.org/Conferences/PLMW/>
 - VMW: <https://i-cav.org/2024/workshops/mentoring/>
- Researchers to follow (very incomplete list):
 - Isil Dillig, Peter Müller, Peter O'Hearn, Leonardo de Moura, Leslie Lamport, Rustan Leino, Kenneth McMillan, Ranjit Jhala, Thomas Wies, Derek Dreyer, Swarat Chaudhuri, Clément Pit-Laudel

VERSE lab: Verified Systems Engineering

software
systems

formal
verification

Multi-Modal Verification in Lean
Correct-by-Construction Program Synthesis
Foundations of Proofs about Programs

Veil: A Framework for Automated and Interactive Verification of Transition Systems

George Pirlea¹, Vladimir Gladstein¹, Elad Kinsbruner^{2*}, Qiyuan Zhao¹, and Ilya Sergey¹



¹ National University of Singapore
[verse-lab.github.io](https://github.com/verse-lab)
² Technion, Haifa

Velvet: A Multi-Modal Verifier for Effectful Programs

Vladimir Gladstein
National University of Singapore
vgladst@comp.nus.edu.sg

George Pirlea

Qiyuan Zhao

Foundational Multi-Modal Program Verifiers

VLADIMIR GLADSTEIN, National University of Singapore, Singapore

Compositional Verification of Composite Byzantine Protocols

Oivuan Zhao

George Pirlea

Karolina Grzeszkiewicz

National University of Singapore
Singapore, Singapore
gpirllea@comp.nus.edu.sg

Yale-NUS College
Singapore, Singapore
karolina.grzeszkiewicz@u.yale-nus.edu.sg

Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis

Yuxi Ling*

Gokul Rajiv*

Kiran Gopinathan[†]

Ilya Sergey*

*National University of Singapore

[†]University of Illinois Urbana-Champaign

If you

- Like understanding *complex software artefacts* (compilers, concurrency, distributed systems)
- Enjoy thinking about **logics** and **type systems**
- Love **metaprogramming** and hacking proofs in **Lean**
- Want to publish at **top PL/FM venues** (PLDI, POPL, ICFP, CAV)



Join VERSE lab @ NUS!

- **Fully-funded 5-year PhD positions**
- Also hiring **Postdocs** and **Interns**
- Get to live in **Singapore**: the world's most **diverse** and **safest city**
- and travel around the world



Lab retreat
August 2025,
Langkawi, Malaysia



This is a durian.
We have them too.



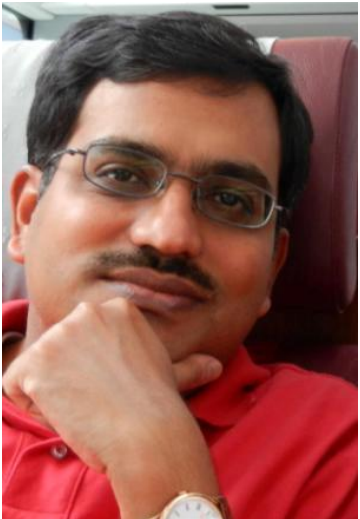
More PL/SE/FM at NUS School of Computing



Umang Mathur
dynamic concurrency analysis,
algorithmic verification
focs-lab.comp.nus.edu.sg



Manuel Rigger
testing for databases,
compilers, solvers
nus-test.github.io



Abhik Roychoudhury
program repair, fuzzing, LLMs
nus-tss.github.io



Prateek Saxena
PL for security
comp.nus.edu.sg/~prateeks

nus-plse.github.io