

# CS:5810 Formal Methods in Software Engineering

## Introduction to Floyd-Hoare Logic

*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.*

# From contracts to Floyd-Hoare Logic

In the **design-by-contract** methodology, contracts are usually assigned to procedures or modules

In general, it is possible to assign **contracts** to each **statement** of a program

A **formal framework** for doing this was developed by Tony Hoare, formalizing a reasoning technique by Robert Floyd

It is based on the notion of a **Hoare triple**

Dafny is based on Floyd-Hoare Logic

# 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:**  $\{ x < 18 \} x := y \{ y \geq 0 \}$

# 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

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

**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  $\{ ? \} x := E \{ Q \}$

we construct  $?$  by replacing each  $x$  in  $Q$  with  $E$  (denoted by  $Q[x \setminus E]$ )



# 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

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

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!)

# Simultaneous assignments

Dafny allows several assignments in one statement

## Examples:

`x, y := 3, 10;` sets `x` to 3 and `y` to 10

`x, y := x + y, x - y;` sets `x` to the sum of `x` and `y`  
and `y` to their difference

All right-hand sides are computed before any variables are assigned.  
Note difference with

`x := x + y; y := x - y;`

# Simultaneous assignments

The weakest precondition of

$$x_1, x_2 := E_1, E_2$$

is constructed by replacing in postcondition  $Q$

- each  $x_1$  with  $E_1$  and
- each  $x_2$  with  $E_2$  (denoted  $Q[x_1, x_2 \setminus E_1, E_2]$ )

**Example:**

$$\begin{array}{ll} \{ x == X \ \&\& \ y == Y \} & \\ \{ y == Y \ \&\& \ x == X \} & Q[x, y \setminus E, F] \\ x, y := y, x & E, F \\ \{ x == Y \ \&\& \ y == X \} & Q \end{array}$$

# Variable introduction

`var x := tmp;` is actually **two** statements:

`var x; x := tmp;`

Cannot assume anything about value of introduced variable

`{ forall x :: Q } var x { Q }`

**Examples:**

`{ forall x :: 0 <= x } var x { 0 <= x }`

`{ forall x :: 0 <= x*x } var x { 0 <= x*x }`



# 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:**

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

# 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  $P$

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

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



# 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

$$S ; T \quad \{ P \} S \{ Q \} T \{ R \}$$
$$\{ P \} S \{ Q \} \text{ and } \{ Q \} T \{ R \}$$

## Strongest postcondition

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

$$SP[S; T, P] = SP[T, Q] = SP[T, SP[S, P]]$$

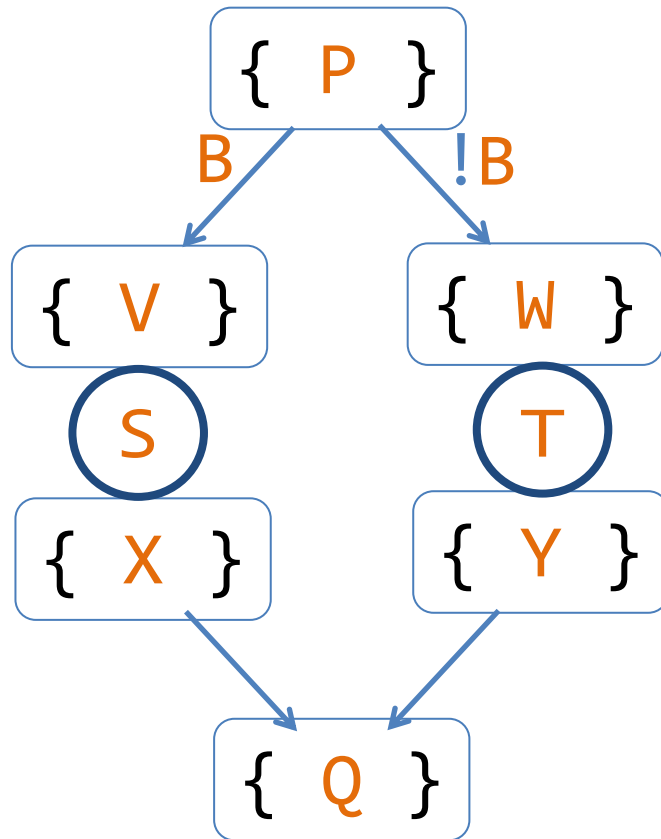
## Weakest precondition

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

$$WP[S; T, R] = WP[S, Q] = WP[S, WP[T, R]]$$

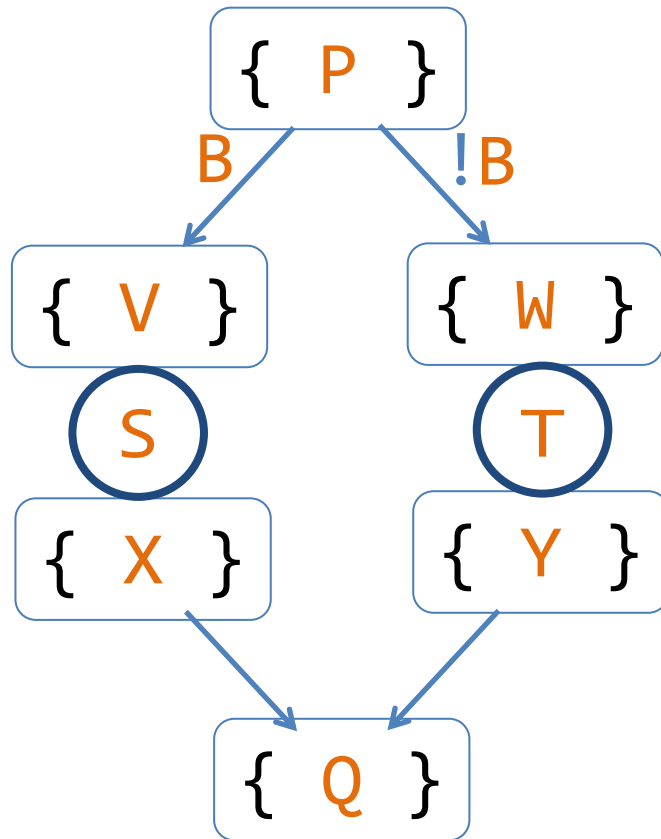
# Conditional control flow

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



# Conditional control flow

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

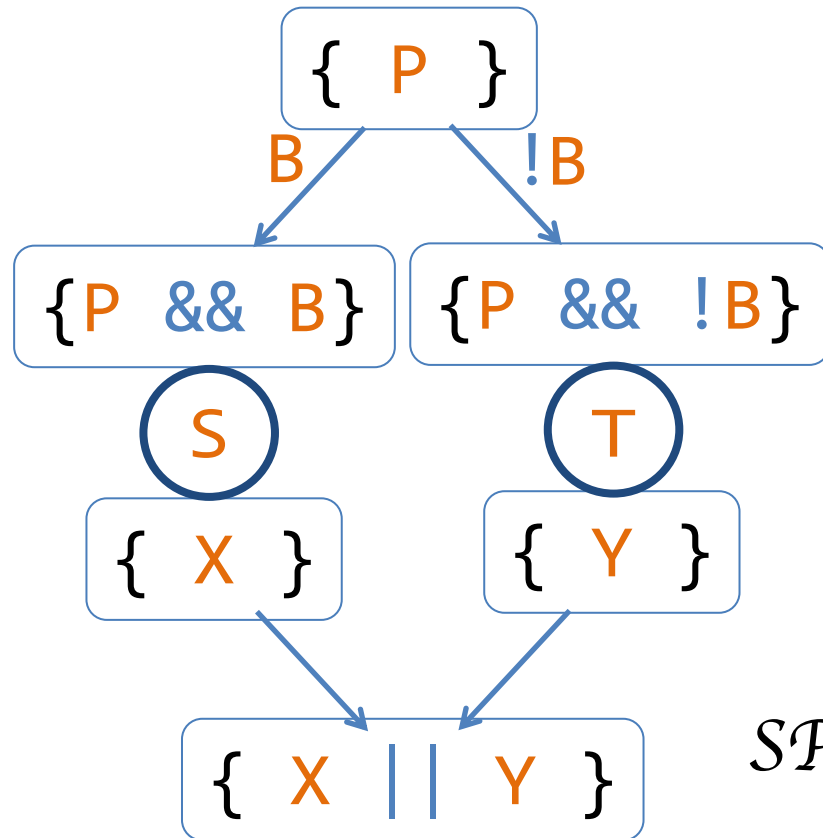


Floyd-Hoare logic tells us:

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 }



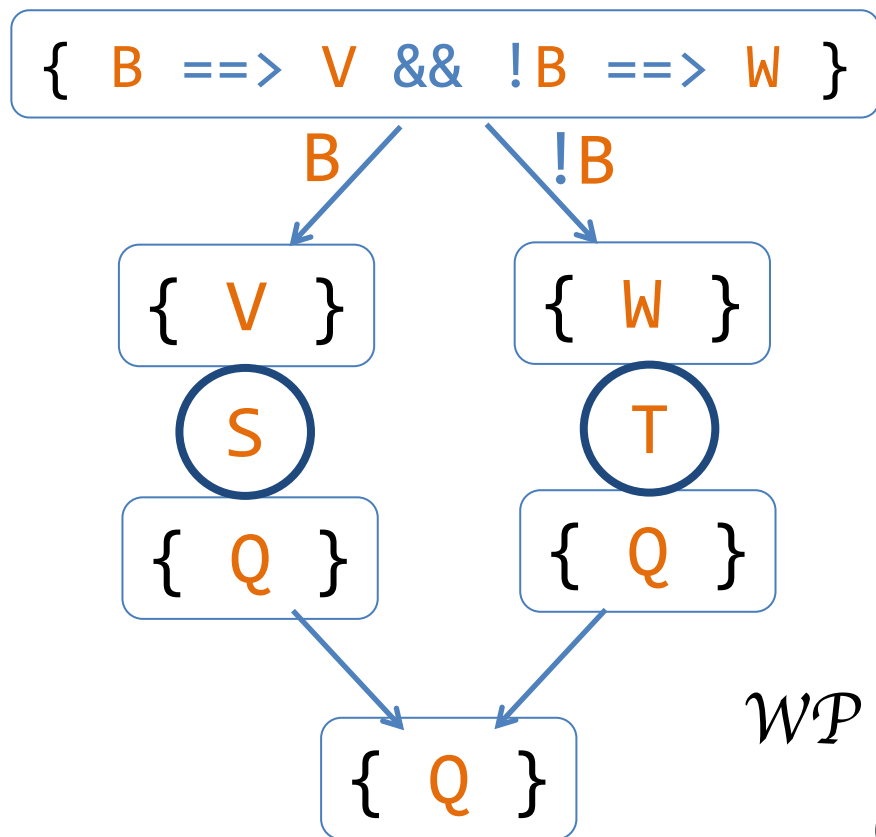
$$X = SP [P \ \&\& \ B, S]$$

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

$$SP [\text{if } B \{ S \} \text{ else } \{ T \}, P] = SP [P \ \&\& \ B, S] \ || \ SP [P \ \&\& \ !B, T]$$

# Weakest precondition

if B { S } else { T }



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

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

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

# 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	$\text{true}$
$A \implies \text{false}$	"	$\neg A$
$\text{true} \implies B$	"	$B$
$\text{false} \implies B$	"	$\text{true}$

Useful law for simplifying predicates

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

# 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 }
```

# Method correctness

Given

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

we need to prove

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

# Method calls

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

Given

```
method Triple(x: int) returns (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) returns (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

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

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

The semantics of `v := 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

method  $M(x: X)$  returns  $(y: Y)$  ensures  $R[x, y]$

$$\begin{aligned} & \mathcal{WP} [r := M(E), Q] && \text{with } x_E, y_r \text{ fresh} \\ &= \mathcal{WP} [\text{var } x_E; \text{var } y_r; x_E := E; \text{assume } R[x, y \setminus x_E, y_r]; r := y_r, Q] \\ &= \mathcal{WP} [\text{var } x_E, \mathcal{WP} [\text{var } y_r, \mathcal{WP} [x_E := E, \mathcal{WP} [\text{assume } R[x, y \setminus x_E, y_r], \mathcal{WP} [r := y_r, Q]]]]] \\ &= \mathcal{WP} [\text{var } x_E, \mathcal{WP} [\text{var } y_r, \mathcal{WP} [x_E := E, \mathcal{WP} [\text{assume } R[x, y \setminus x_E, y_r], Q[r \setminus y_r]]]]] \\ &= \mathcal{WP} [\text{var } x_E, \mathcal{WP} [\text{var } y_r, \mathcal{WP} [x_E := E, R[x, y \setminus x_E, y_r] \implies Q[r \setminus y_r]]]] \\ &= \mathcal{WP} [\text{var } x_E, \mathcal{WP} [\text{var } y_r, R[x, y \setminus E, y_r] \implies Q[r \setminus y_r]]] \\ &= \mathcal{WP} [\text{var } x_E, \text{forall } y_r :: R[x, y \setminus E, y_r] \implies Q[r \setminus y_r]] \\ &= \text{forall } x_E :: \text{forall } y_r :: R[x, y \setminus E, y_r] \implies Q[r \setminus y_r] \\ &= \text{forall } y_r :: R[x, y \setminus E, y_r] \implies Q[r \setminus y_r] && \text{since } x_E \text{ not in } Q \end{aligned}$$



# Weakest precondition

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

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) returns (r: int) {  
    var 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) returns (y: Y)
  requires 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
```

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

# Function calls

```
function Average(a: int, b: int): int {  
    (a + b) / 2  
}
```

No output  
parameters

An expression,  
not a statement

Functions are *transparent*: we reason about them in terms of their definition, not a specification

```
method Triple(x: int) returns (r: int)  
    ensures r == Average(2*x, 4*x)
```

# Function calls

In Dafny, **functions** are part of the **specification**

If you want to use a function **in code**, you need to use a *function method*

```
function method Average(a: int, b: int): int {  
    (a + b) / 2  
}
```

```
method Triple(x: int) returns (r: int)  
    ensures r == 3*x  
{  
    r := Average(2*x, 4*x);  
}
```

# Partial expressions

An expression may be not always well defined,  
e.g.,  $c/d$  when  $d$  evaluates to  $\emptyset$

Associated with such *partial expressions* are *implicit assertions*

## Example:

```
assert d !=  $\emptyset$  && v !=  $\emptyset$ ;  
if c/d < u/v {  
    assert  $\emptyset$  <= i < a.Length;  
    x := a[i];  
}
```

# Partial expressions

Functions may have preconditions, making calls to them partial

**Example:** given

```
function method MinusOne(x: int): int  
  requires 0 < x
```

the call `z := MinusOne(y + 1)` has an implicit assertion

```
assert 0 < y + 1
```

# Exercises

1. Suppose you want  $x + y == 22$  to hold after the statement

```
if x < 20 { y := 3; } else { y := 2; }
```

In which states can you start the statement? In other words, compute the weakest precondition of the statement with respect to  $x + y == 22$ . Simplify the condition after you have computed it.

2. Compute the weakest precondition for the following statement with respect to  $y < 10$ . Simplify the condition.

```
if x < 8 {  
  if x == 5 { y := 10; } else { y := 2; }  
} else {  
  y := 0;  
}
```



# Exercises

3. Compute the weakest precondition for the following statement with respect to  $y \% 2 == 0$  (that is, "y is even"). Simplify the condition.

```
if x < 10 {  
  if x < 20 { y := 1; } else { y := 2; }  
} else {  
  y := 4;  
}
```

4. Compute the weakest precondition for the following statement with respect to  $y \% 2 == 0$  (that is, "y is even"). Simplify the condition.

```
if x < 8 {  
  if x < 4 { x := x + 1; } else { y := 2; }  
} else {  
  if x < 32 { y := 1; } else { }  
}
```

# Exercises

5. Determine under which circumstances the following program establishes  $0 \leq y < 100$ . Try first to do that in your head. Write down the answer you come up with, and then write out the full computations to check that you got the right answer.

```
if x < 34 {
  if x == 2 { y := x + 1; } else { y := 233; }
} else {
  if x < 55 { y := 21; } else { y := 144; }
}
```

6. Which of the following Hoare-triple combinations are valid?

- a)  $\{0 \leq x\} x := x + 1 \{ -2 \leq x \} y := 0 \{ -10 \leq x \}$
- b)  $\{0 \leq x\} x := x + 1 \{ \text{true} \} x := x + 1 \{ 2 \leq x \}$
- c)  $\{0 \leq x\} x := x + 1; x := x + 1 \{ 2 \leq x \}$
- d)  $\{0 \leq x\} x := 3 * x; x := x + 1 \{ 3 \leq x \}$
- e)  $\{x < 2\} y := x + 5; x := 2 * x \{x < y\}$

# Exercises

7. Compute the weakest precondition of the following statements with respect to the postcondition  $x + y < 100$ .

a)  $x := 32; y := 40$

b)  $x := x + 2; y := y - 3 * x$

8. Compute the weakest precondition of the following statement with respect to the postcondition  $x < 10$ :

a)  $\text{if } x \% 2 == 0 \{ y := y + 3; \} \text{ else } \{ y := 4; \}$

b)  $\text{if } y < 10 \{ y := x + y; \} \text{ else } \{ x := 8; \}$

9. Compute the weakest precondition of the following statements with respect to the postcondition  $x < 100$ . Simplify your answer.

a)  $\text{assert } y == 25$

d)  $\text{assert } x \leq 100$

b)  $\text{assert } 0 \leq x$

e)  $\text{assert } 0 \leq x < 100$

c)  $\text{assert } x < 200$

# Exercises

**10.** If  $x'$  does not appear in the desired postcondition  $Q$ , then prove that  $x' := E; \text{assert } P[x \setminus x']$  is the same as  $\text{assert } P[x \setminus E]$  by showing that the weakest preconditions of these two statements with respect to  $Q$  are the same.

**11.** What implicit assertions are associated with the following expressions?

- a)  $x / (y + z)$
- b)  $a[2 * i]$
- c)  $\text{MinusOne}(\text{MinusOne}(y))$

**12.** What implicit assertions are associated with the following expressions?

**Note:** The right-hand expression in a conjunction is only evaluated when the left-hand conjunction is true.

- a)  $a / b < c / d$
- b)  $a / b < 10 \ \&\& \ c / d < 100$
- c)  $\text{MinusOne}(y) == 8 \implies a[y] == 2$