Practical Formal Methods

Introduction to Floyd-Hoare Logic

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 (seen before)

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*

precondition → `{ P } S { Q }` ← postcondition

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 >= 0 }
{ x < 18 } y := 5          { y >= 0 }
```

Non-example: `{ x < 18 } x := y { y >= 0 }`

Forward reasoning

Constructing a postcondition from a given precondition

In general, there are many possible postconditions

Examples:

```
1. { x == 0 }  y := x + 3  { y < 100 }
2. { x == 0 }  y := x + 3  { x == 0 }
3. { x == 0 }  y := x + 3  { 0 <= x && y == 3 }
4. { x == 0 }  y := x + 3  { 3 <= y }
5. { x == 0 }  y := x + 3  { true }
```

Strongest postcondition

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

```
{ x == 0 }  y := x + 3  { 0 == x && y == 3 }
```

Def: A is *stronger* than B if A ==> 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 <= 70 }  y := x + 3  { y <= 80 }
2.   { x == 65 && y < 21 }  y := x + 3  { y <= 80 }
3.              { x <= 77 }  y := x + 3  { y <= 80 }
4.   { x*x + y*y <= 2500 }  y := x + 3  { y <= 80 }
5.               { false }  y := x + 3  { y <= 80 }
```

Weakest precondition

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

```
{ x <= 77 }  y := x + 3  { y <= 80 }
```

Def: A is *weaker* than B if B ==> 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\E])

Weakest precondition for assignment

Given `{Q[x\E]} x := E { Q }`

Examples: `{ ? } y := a + b { 25 <= y }`

`25 <= a + b`

```
1. { 25 <= x + 3 + 12 } a := x + 3 { 25 <= a + 12 }
2.          { x + 1 <= y } x := x + 1 { x <= 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!)

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 $Q[x_1,x_2\backslash E_1,E_2]$)

Example:

```
{ x == X && y == Y }
{ y == Y && x == X }      Q[x,y\E,F]
x, y := y, x              E, F
{ x == Y && y == X }      Q
```

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:

false

```
{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 { exists n :: P[x\n] && x == E[x\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\E]

SP [x := E, P] = exists n :: P[x\n] && x == E[x\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   { P } S { Q } T { R }

{ P } S { Q } and { Q } T { R }

Strongest postcondition

let Q = SP [S, P]

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

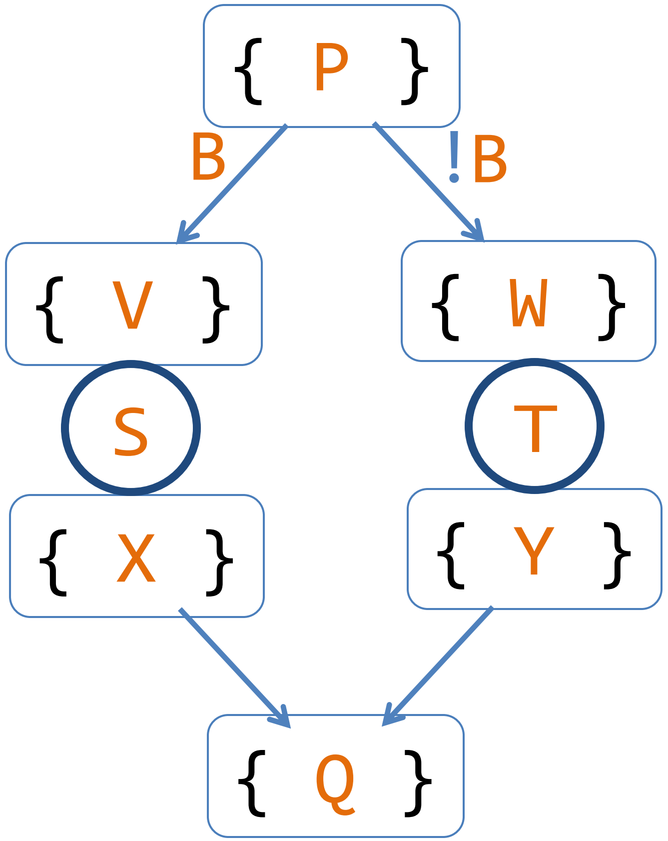
Weakest precondition

let Q = WP [T, R]

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

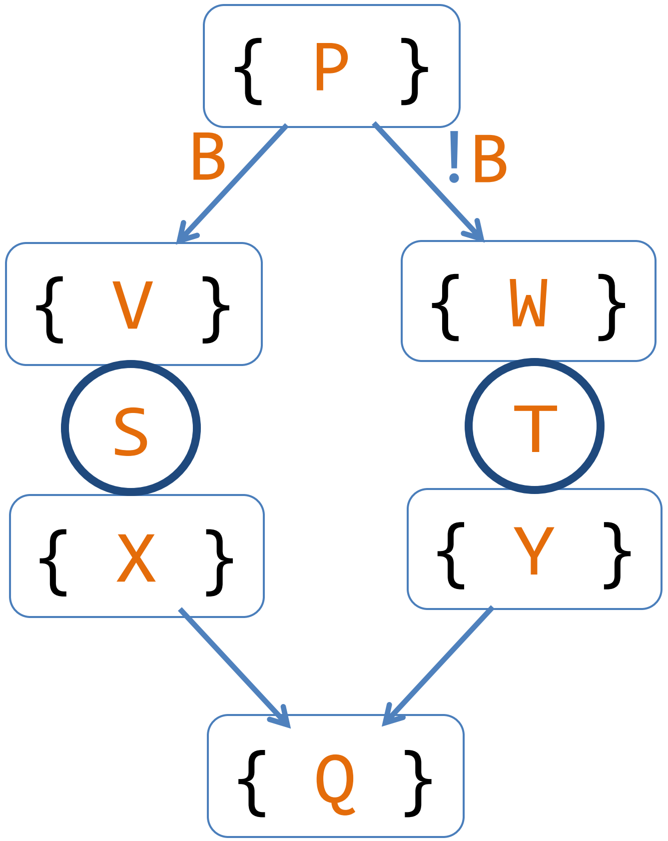
Conditional control flow

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



Conditional control flow

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

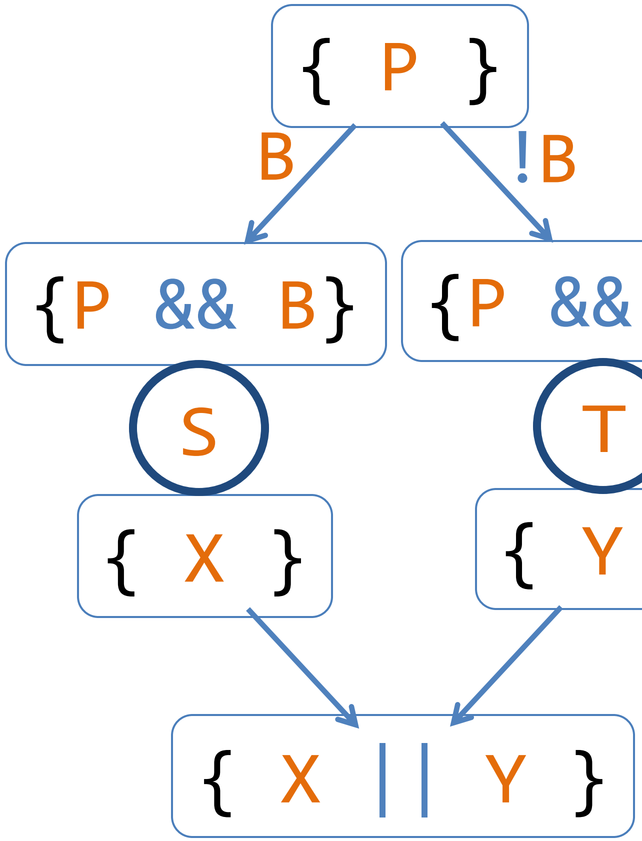


Floyd-Hoare logic tells us:

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 }
```



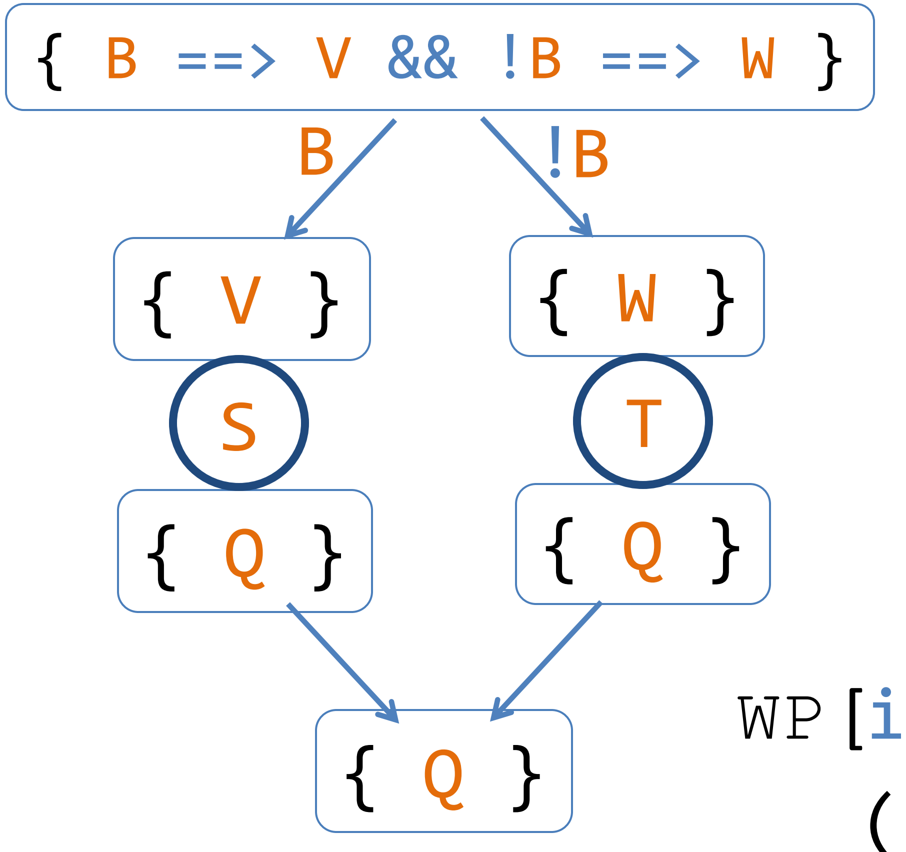

```
X = SP [S, P && B]

Y = SP [T, P && !B]
```

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

Weakest precondition

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



V = WP [S, Q]

W = WP [T, Q]

```
WP [if B { S } else { T }, Q] =
    ( B ==> WP [S, Q]) &&
    (!B ==> WP [T, Q])
```

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 ==> B equiv. to !A || B

Hence,

A ==> true	equiv. to	true
A ==> false	"	! A
true ==> B	"	B
false ==> B	"	true

Useful law for simplifying predicates

A ==> (B ==> C) equiv. to (A && B) ==> 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 ==> 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[assume E, P] = E && P`

`WP[assume E, Q] = E ==> 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

`WP[r := M(E), Q] = forall y':: R[x,y\E,y'] ==> Q[r\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 [assert E, Q] = E && Q

SP [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 x_E ; var y_r ;
x_E := E ;
assert P[x\x_E] ;
assume R[x,y\x_E,y_r] ;
r := y_r
```

```
WP[r := M(E),Q] = P[x\E] && 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 `0`

Associated with such *partial expressions* are implicit assertions

Example:

```
assert d != 0 && v != 0;
if c/d < u/v {
  assert 0 <= 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
```

Next lecture

- Programs with loops
- Iterative computations
- Arrays