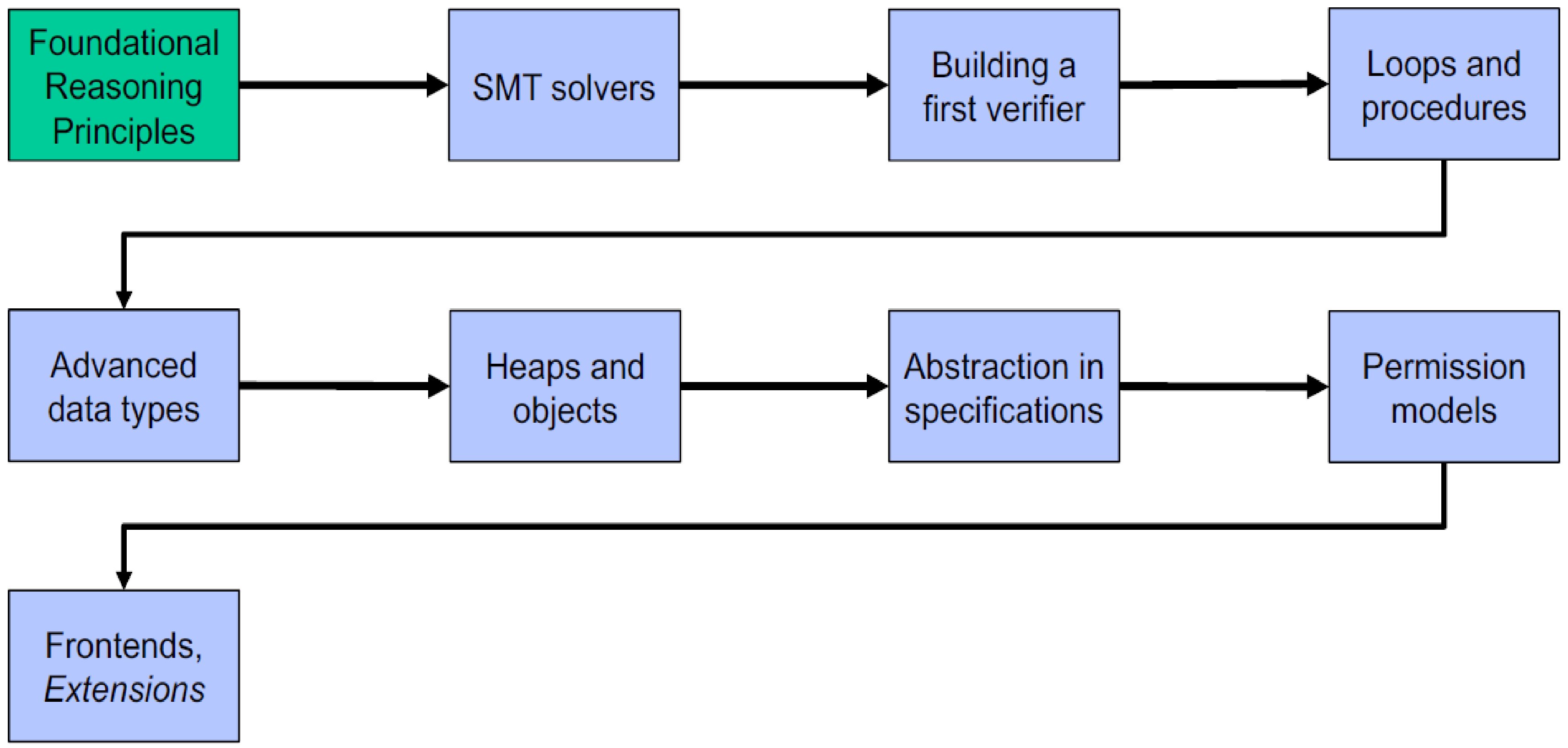
Methodologies for Software Processes

Lecture 2

Tentative course outline



Outline

1. Why do we need formal foundations?
2. Formalizing contracts
3. Reasoning about contracts
4. Epilogue

The Program Verification Task

Given a **program**

and a **specification spec,**

give a **proof**

that all program executions

comply with **spec**

```
fn abs(x:i32) -> i32 {
  if x >= 0 {
    return x
  } else {
    return -x
  }
}
```

spec: abs(x) returns |x|

Does every execution comply with **spec**?

Verifying abs(x)

```
// Viper model of abs(x)
method abs(x: Int) returns (r: Int)
  ensures x >= 0 ==> r == x
  ensures x <= 0 ==> r == -x
{
  if (x < 0) { r := -x } else { r := x }
}
```

```
fn abs(x:i32) -> i32 {
  if x >= 0 {
    return x
  } else {
    return -x
  }
}
```

`i32`: 32-bit integers in two's complement!

```
i32::MIN    is    -2_147_483_648i32
i32::MAX    is     2_147_483_647i32

abs(i32::MIN) == ???
```

spec: `abs(x)` returns |x|

Problem: Viper model does not capture the semantics of the Rust program

Refined verification of abs(x)

```
define i32MIN (-2147483648)
define i32MAX (2147483647)

method abs(x: Int) returns (r: Int)
  requires i32MIN <= x && x <= i32MAX
  ensures i32MIN <= r && r <= i32MAX
  ensures x >= 0 ==> r == x
  ensures x <= 0 ==> r == -x
{
  if (x < 0) { r := -x } else { r := x }
}
```

Refined specification for abs(x)

```
define i32MIN (-2147483648)
define i32MAX (2147483647)

method abs(x: Int) returns (r: Int)
  requires i32MIN <= x && x <= i32MAX
  requires x != i32MIN
  ensures i32MIN <= r && r <= i32MAX
  ensures x >= 0 ==> r == x
  ensures x <= 0 ==> r == -x
{
  if (x < 0) { r := -x } else { r := x }
}
```

Verification must be rooted in rigorous mathematics

Given a **program**

semantics

and a **specification spec,**

formal logic

give a **proof**

automated provers

that all program executions

comply with **spec**

program logics

✓

~

~

Floyd-Hoare logic

?

Outline

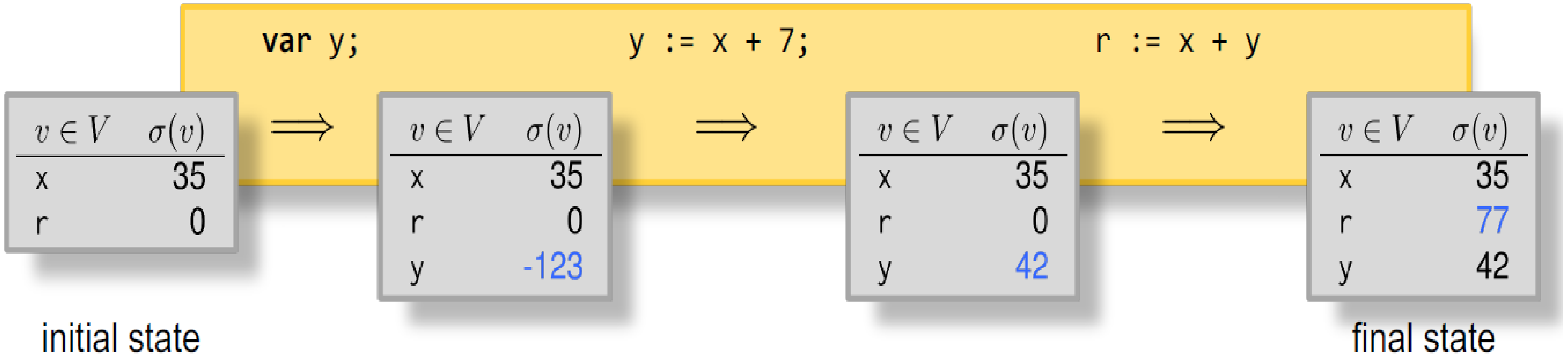
1. Why do we need formal foundations?
2. Formalizing contracts
3. Reasoning about contracts
4. Epilogue

Making it formal

Program states assign **values** to **variables in Var**

$$\textbf{States} \quad = \quad \{\, \sigma : V \to \texttt{Int} \mid V \subseteq \textbf{Var} \text{ and } V \text{ finite} \,\}$$

Program semantics describes how *states* evolve during program execution


```
var y;                  y := x + 7;                  r := x + y
```

$v \in V$	$\sigma(v)$
x	35
r	0

initial state

$\Longrightarrow$

$v \in V$	$\sigma(v)$
x	35
r	0
y	-123

$\Longrightarrow$

$v \in V$	$\sigma(v)$
x	35
r	0
y	42

$\Longrightarrow$

$v \in V$	$\sigma(v)$
x	35
r	77
y	42

final state

Making it formal

Predicates capture properties of program states

$$\mathbf{Pred} \quad = \quad \{\, P \mid P\colon \mathbf{States} \to \mathbf{Bool} \,\}$$

Logical characterization

```
x != 0
```

Set characterization

$$P = \{\, \sigma \in \mathbf{States} \mid \sigma(\mathrm{x}) \neq 0 \,\}$$

Making it formal

Floyd-Hoare **triples** capture properties of (possibly infinitely many) executions

$$\{ \mathit{Pre} \}\ S\ \{ \mathit{Post} \}$$

acceptable initial states

acceptable final states

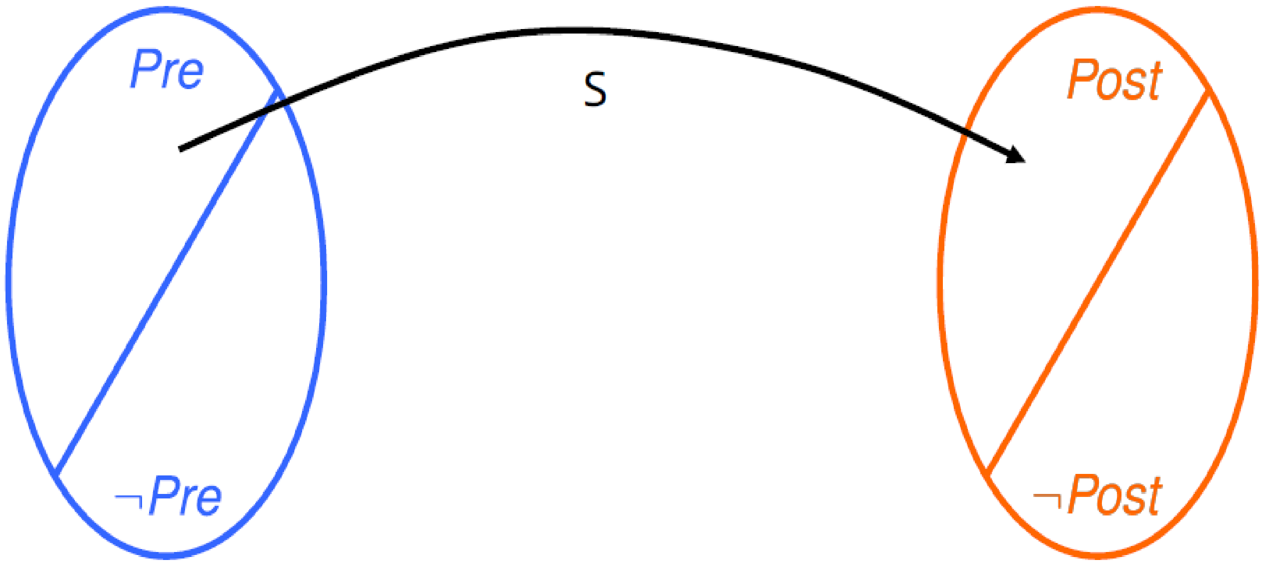
```
method foo(x: Int)
  returns (r: Int)
  requires x > 0
  ensures r > y
{
  var y: Int
  y := x + 7
  r := x + y
}
```

```
{ x > 0 }
  var y;
  y := x + 7;
  r := x + y
{ r > y }
```

- Implicit I/O parameters
- Omit types (only **`Int`**)
- Moved pre- and postcondition

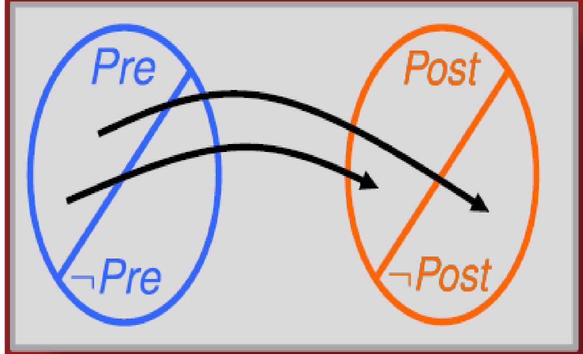
Making it formal

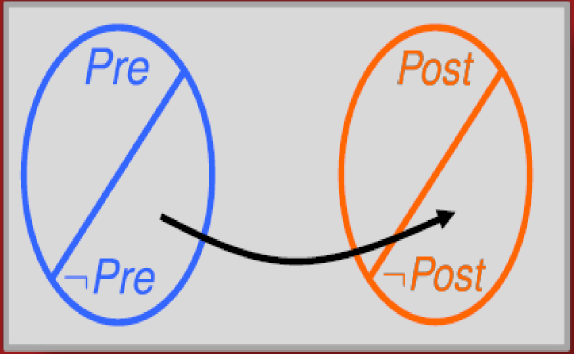
The triple `{ Pre } S { Post }` is **valid** if and only if
when program `S` is started in any state in `Pre`,
then `S` terminates in a state in *Post*.

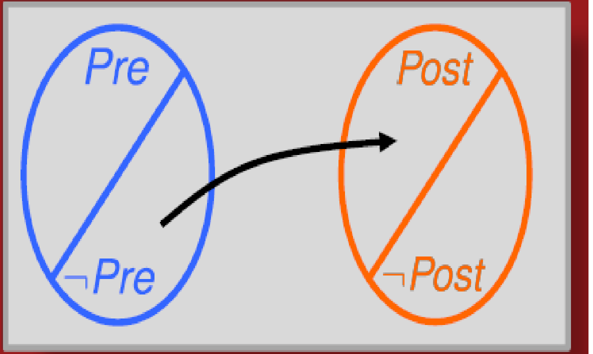


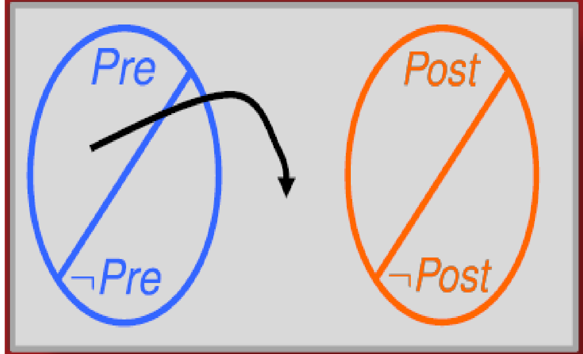
Which pictures correspond to valid Floyd-Hoare triples?

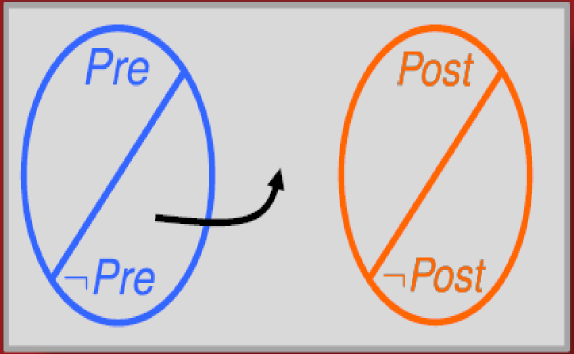
`{ Pre } S { Post }` is **valid** iff when program `S` is started in any state in *Pre*, then `S` terminates in a state in *Post*.

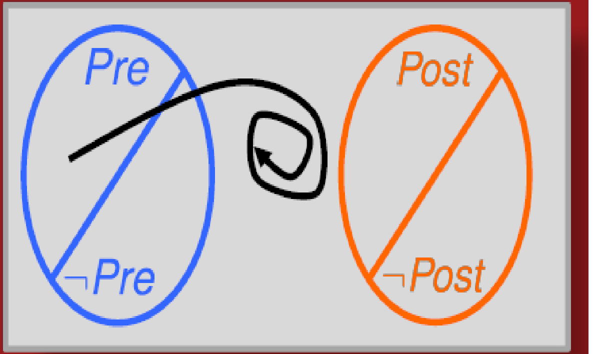






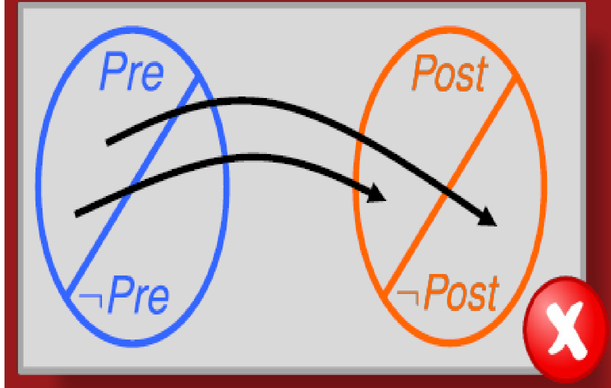


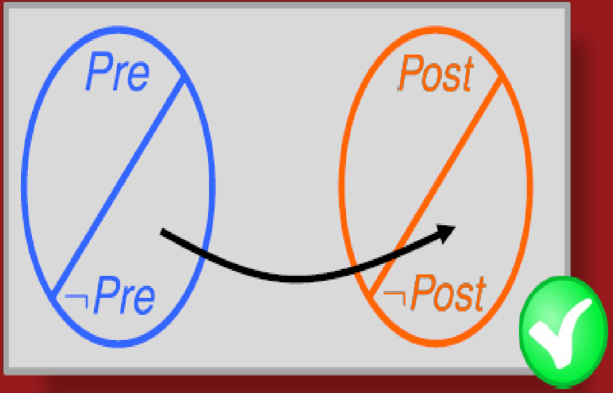


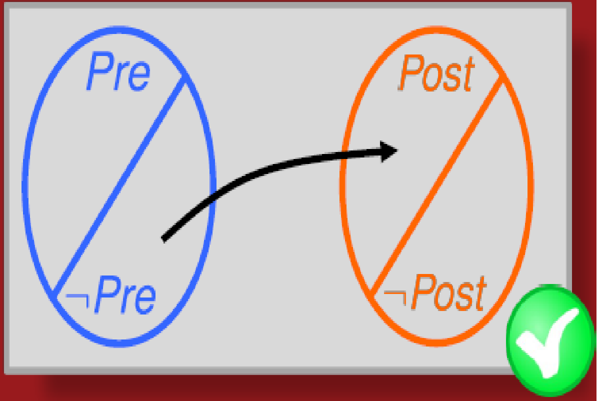


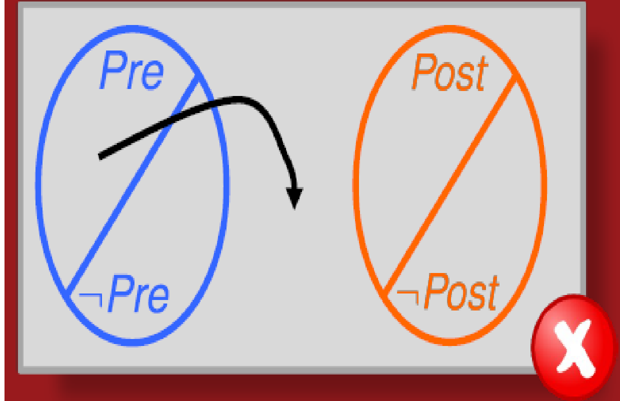
Solution

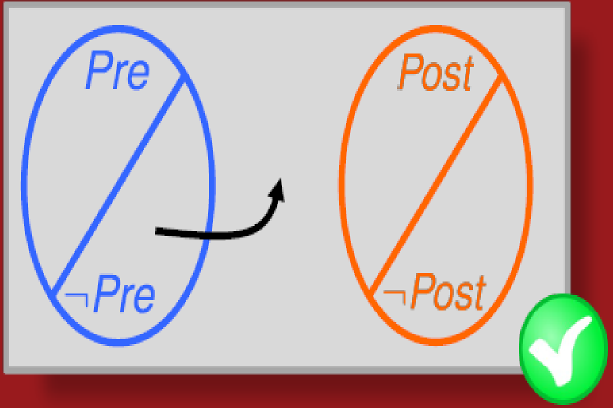
{ *Pre* } S { *Post* } is **valid** iff when program S is started in any state in *Pre*, then S terminates in a state in *Post*.

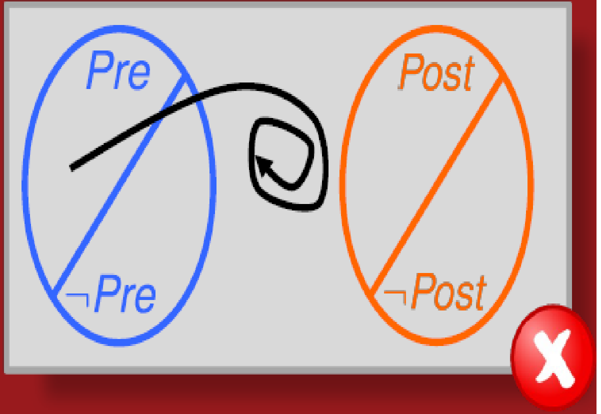










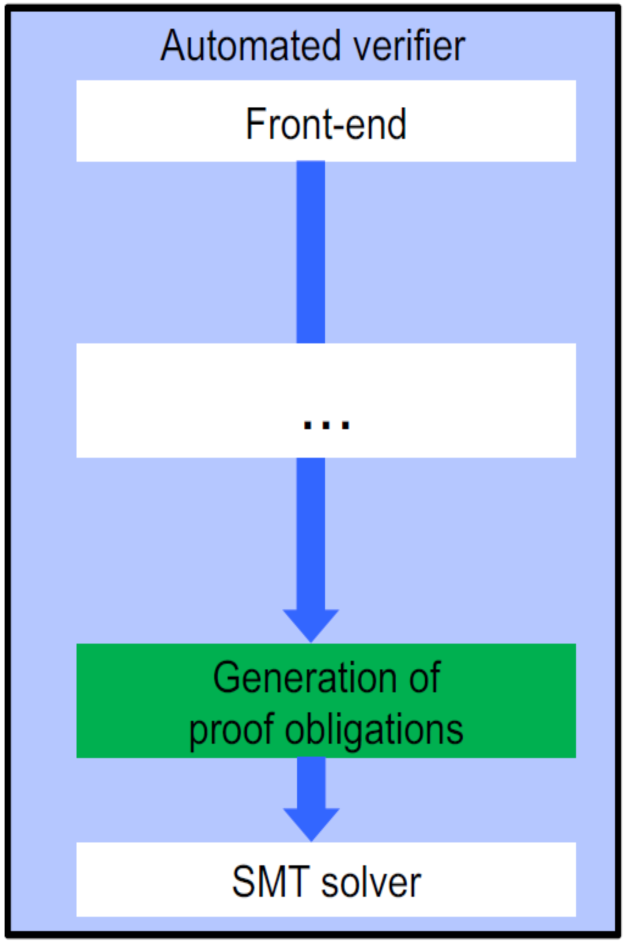


Outline

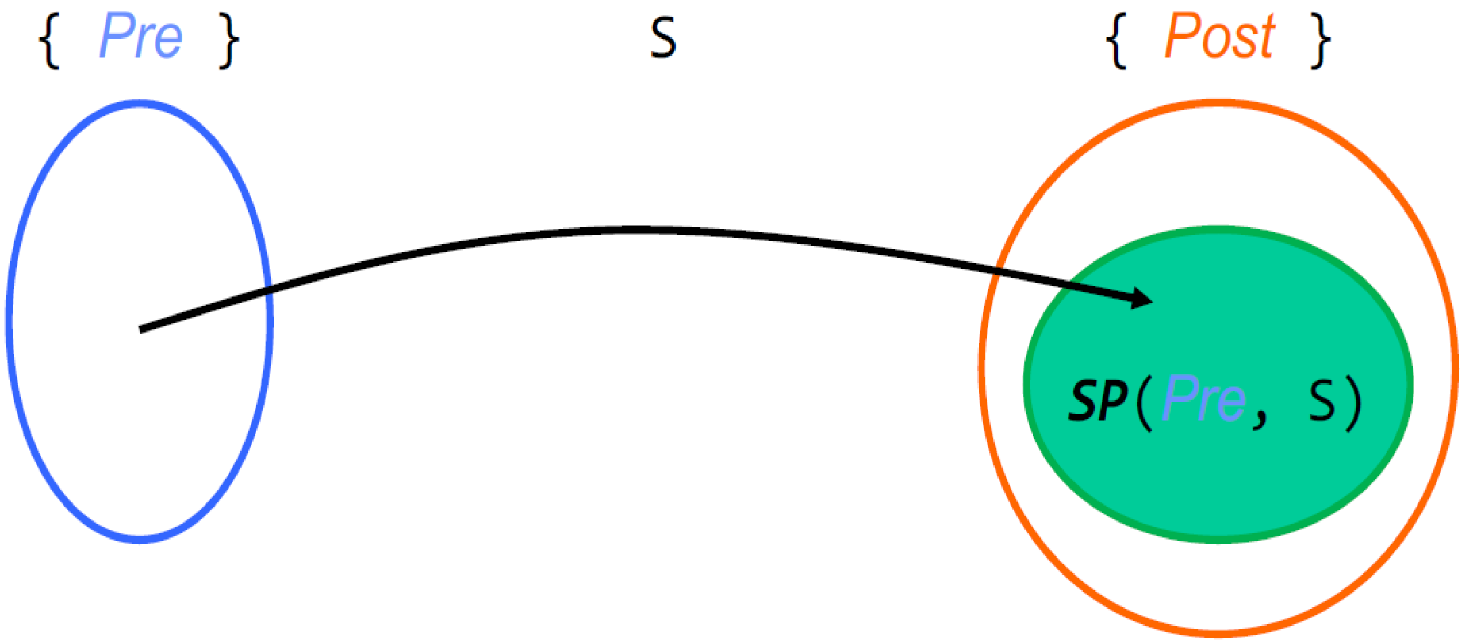
1. Why do we need formal foundations?
2. Formalizing contracts
3. Reasoning about contracts
4. Epilogue

How do we systematically prove a triple valid?

- Determine a **verification condition** VC
  - VC is a predicate
  - VC is **valid** iff it is true for *all* states
- **Soundness:** VC is valid → triple is valid
- **Completeness:** triple is valid → VC is valid
- **Predicate transformers** describe how *predicates* evolve during program execution



Forward Reasoning

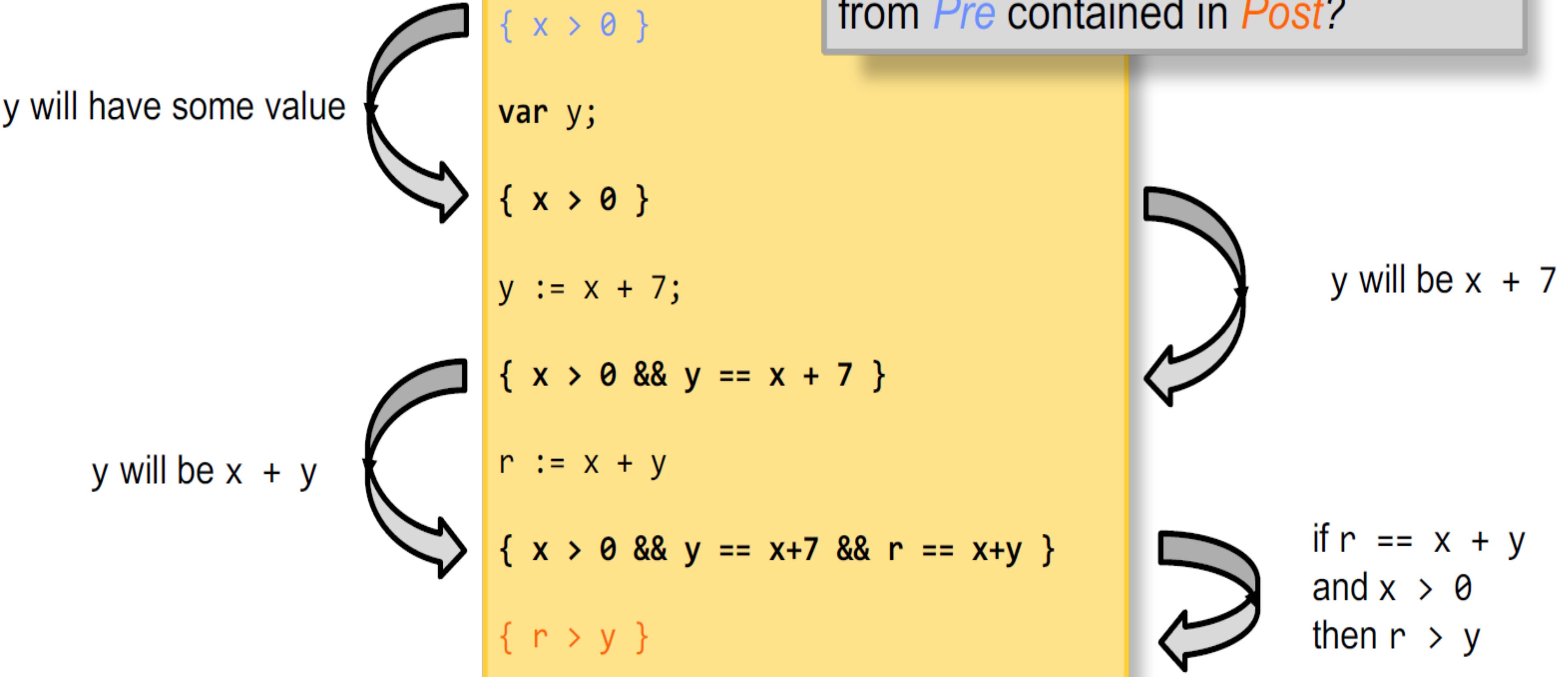


Forward VC: is the strongest postcondition ***SP***(*Pre*, S)
(all final states that we can reach from *Pre*)
of *Pre* and program S contained in *Post*?

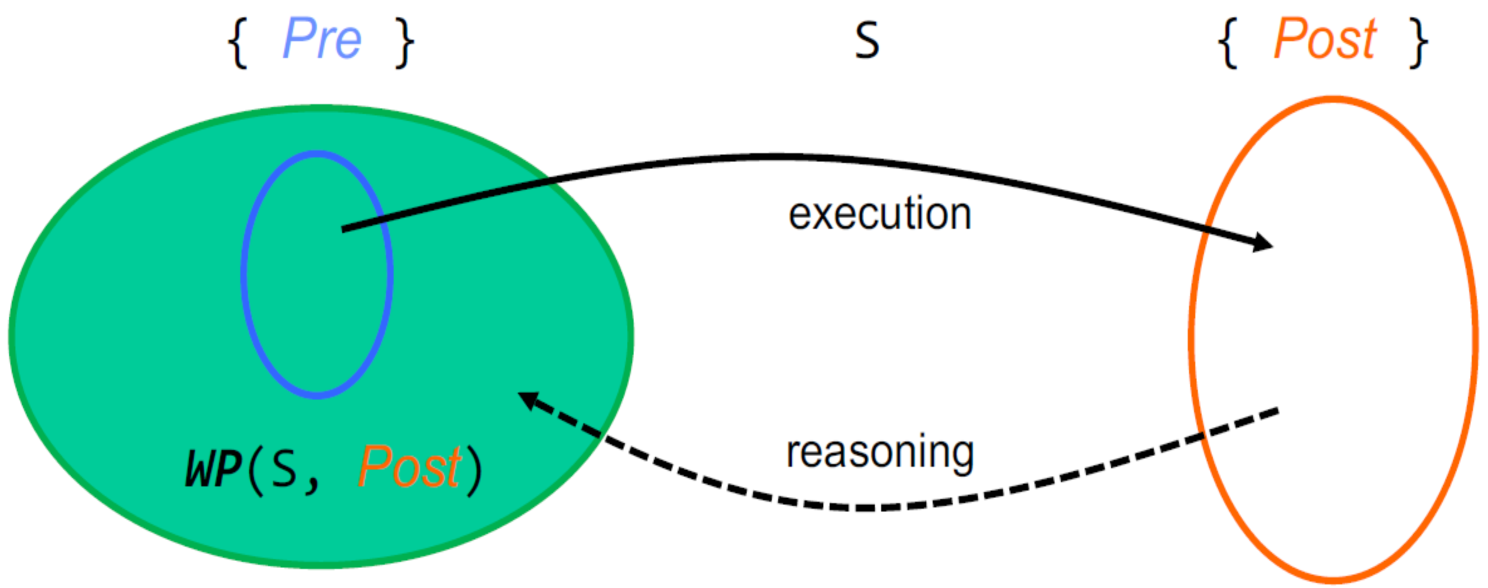
Informal Forward Reasoning

Are all final states that we can reach from *Pre* contained in *Post*?

```
{ x > 0 }

var y;

{ x > 0 }

y := x + 7;

{ x > 0 && y == x + 7 }

r := x + y

{ x > 0 && y == x+7 && r == x+y }

{ r > y }
```



Backward Reasoning



Backward VC: is *Pre* included in the **weakest precondition** ***WP***(S, *Post*)
(all initial states from which we must terminate in *Post*)
of program S and *Post*?

Informal Backward Reasoning

```
{ x > 0 }

{ forall y :: x > 0 }

var y;

{ x > 0 }

y := x + 7;

{ x + y > y}

r := x + y

{ r > y }
```

this is true if `x > 0`

y could have any value before its declaration

whatever we assign to y, we have `x > 0`

`x + y` must have been greater than y before

Is *Pre* included in all initial states from which we must terminate in *Post*?

PL0: a first programming language

x is a variable in **Var**

z is a constant in **Int**

Arithmetic expressions

```
a     ::=     x  |  z  |  a + a  |  a - a  |  a / a  |  a % a
```

Boolean expressions

```
b     ::=     true | false | a < a | a = a | b && b | b || b | !b | ...
```

Predicates (incomplete)

```
P, Q, R  ::=  b  |  P && P  |  P ==> P  |  exists x :: P  | forall x :: P | ...
```

Statements in **PL0**

```
S     ::=     var x  |  x := a  |  S;S  |  S [] S  |  assert P  |  assume P
```

Local variable declarations: **var** x

```
{ true }
var x;
{ x >= 0 }
```
✗

```
{ x == 0 }
var x;
{ x <= 0 }
```
✗

```
{ x == 5 && y > x }
var x;
{ y > 5 }
```
✓

Declares an uninitialized variable x that overshadows any existing x.

SP(P, **var** x) ::= **exists** x :: P

WP(**var** x, Q) ::= **forall** x :: Q

Assertions: **assert** R

```
{ x > 7 }
assert x > 5
{ x > 7 }
```
✓

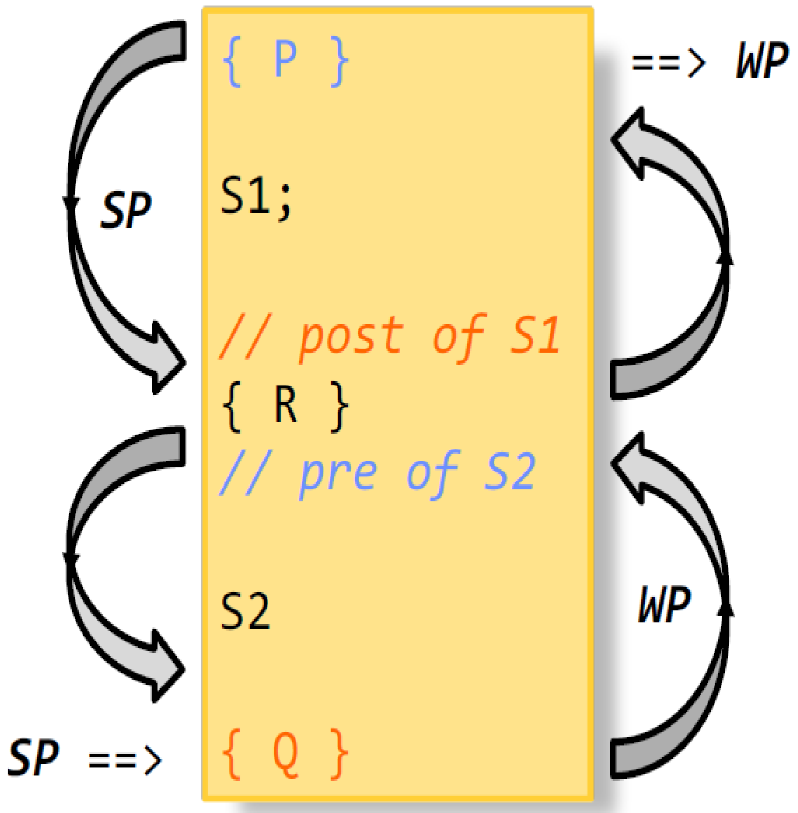
```
{ x > 5 }
assert x > 7
{ x > 7 }
```
✗

Crashes if R does not hold in the current state; otherwise, *no effect*.

SP(P, **assert** R) ::= P && R

WP(**assert** R, Q) ::= R && Q

Sequential composition: `S1;S2`



```
{ P }

S1;

// post of S1
{ R }
// pre of S2

S2

{ Q }
```

First execute S1, then S2.

SP`(P, S1;S2)   ::=   `***SP***`(`***SP***`(P,S1), S2)`

WP`(S1;S2, Q)   ::=   `***WP***`(S1, `***WP***`(S2, Q))`

Nondeterministic choice: S1 [] S2

```
{ x == 7 }
assert x > 0 [] assert y > 0
{ x > 0 }
```
✗

```
{ x > 0 && y > 0 }
assert x*x > 0 [] assert x+y > 0
{ x * x > 0 || x + y > 0 }
```
✓

```
{ x > 0 && y > 0 }
assert x > 0 [] assert y > 0
{ x > 0 }
```
✓

Executes *either* S1 *or* S2.

SP(P, S1 [] S2)
  ::= ***SP***(P,S1) || ***SP***(P, S2)

WP(S1 [] S2, Q)
  ::= ***WP***(S1, Q) && ***WP***(S2, Q)

Assumptions: `assume R`

- Verification-specific statement
- *Not* executable
- Part of trusted code base

Nothing happens if `R` holds in the current state; otherwise, ***magic***.

```
{ x == 0 }
assume x > 0
{ x > 0 }
```
✓

```
{ x > 0 }
assume x > 5
{ x > 5 }
```
✓

```
{ true }
assume false
{ false }
```
✓

```
{ x == 1 }
assume x > 0
{ x == 2 }
```
✗

SP`(P, assume R)  ::=  P && R`

WP`(assume R, Q)  ::=  R ==> Q`

Assignment: x := a

```
{ y > 0 }
x := 17 + y
{ y > 0 && x == 17 + y }
```
✓

```
{ y < 23 }
x := 23
{ y < x }
```
✓

```
{ x + 1 > 42 }
x := x + 1
{ x > 42 }
```
✓

```
{ x > 42 }
x := x + 1
{ x > 42 && x == x + 1 }
```
✗

Assigns the value of a (evaluated in the initial state) to x in the final state.

`WP(x := a, Q) ::= Q[x / a]`

`E[x / F]`: E where every x is replaced by F

Assignment: x := a

```
{ y > 0 }
x := 17 + y
{ y > 0 && x == 17 + y }
```
✓

```
{ y < 23 }
x := 23
{ y < x }
```
✓

```
{ x + 1 > 42 }
x := x + 1
{ x > 42 }
```
✓

```
{ x > 42 }
x := x + 1
{ x > 42 && x == x + 1 }
```
✗

Assigns the value of a (evaluated in the initial state) to x in the final state.

WP`(x := a, Q)  ::=  Q[x / a]`

`E[x / F]`: E where every x is replaced by F

SP`(P, x := a)  ::=  P && x == a` ✗

Assignment: `x := a`

```
{ y > 0 }
x := 17 + y
{ y > 0 && x == 17 + y }
```
✓

```
{ y < 23 }
x := 23
{ y < x }
```
✓

```
{ x + 1 > 42 }
x := x + 1
{ x > 42 }
```
✓

```
{ x > 42 }
x := x + 1
{ x > 42 && x == x + 1 }
```
✗

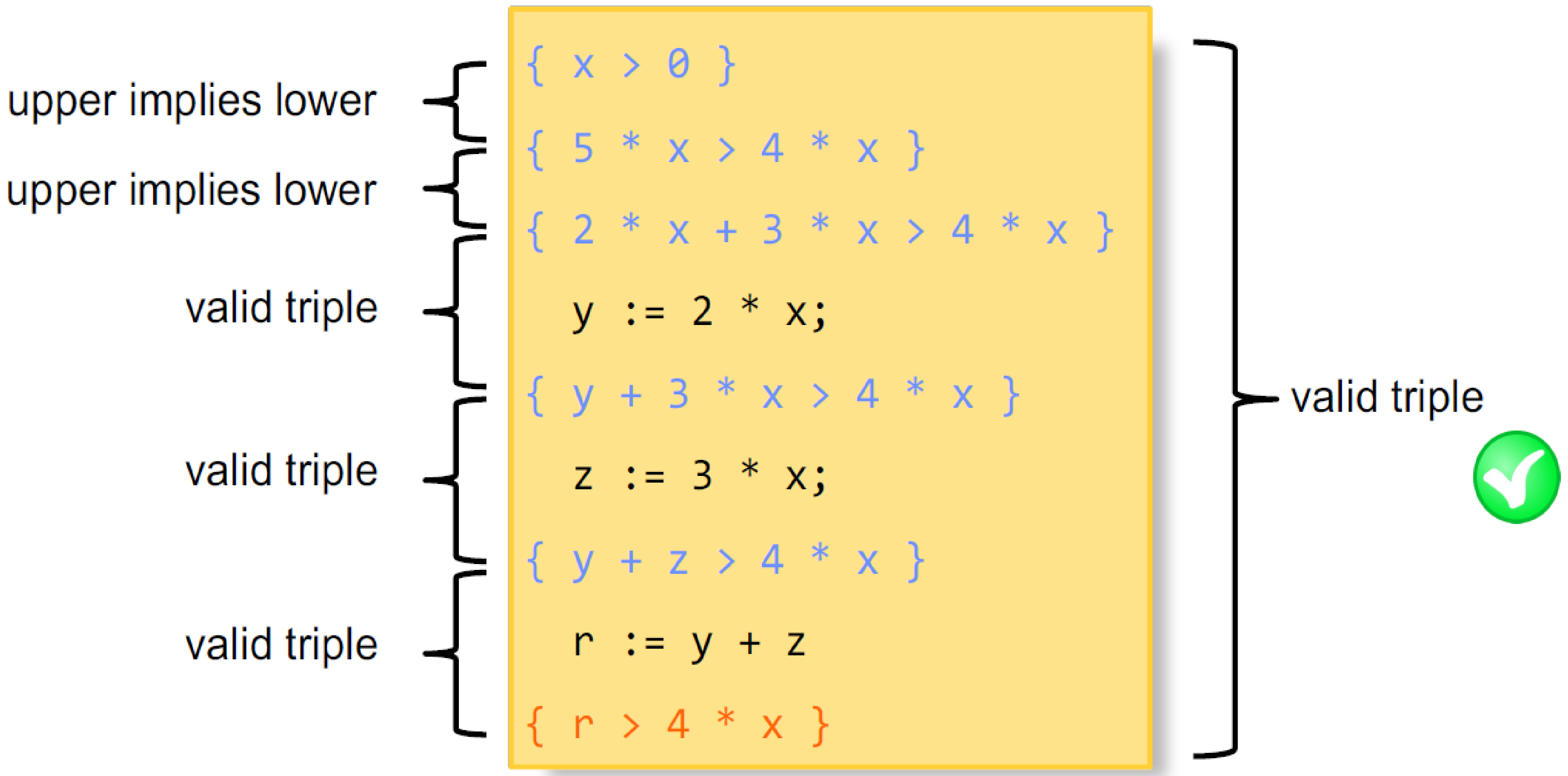
Assigns the value of a (evaluated in the initial state) to x in the final state.

WP`(x := a, Q)  ::=  Q[x / a]`

`E[x / F]`: E where every x is replaced by F

SP`(P, x := a)  ::=  `**`exists`**` x0 ::`
`         P[x / x0] && x == a[x / x0]`

Proof annotations via overlapping Floyd-Hoare triples



```
{ x > 0 }
{ 5 * x > 4 * x }
{ 2 * x + 3 * x > 4 * x }
  y := 2 * x;
{ y + 3 * x > 4 * x }
  z := 3 * x;
{ y + z > 4 * x }
  r := y + z
{ r > 4 * x }
```

Outline

1. Why do we need formal foundations?
2. Formalizing contracts
3. Reasoning about contracts
4. Epilogue

Strongest Post vs. Weakest Pre – Does it matter?

```
{ x == 0 } assert x < 0 { x == 1 }
```

✗

```
  x == 0 ==> WP(assert x < 0, x == 1)
=
  x == 0 ==> (x < 0 && x == 1)

// not valid
```

✗

```
  SP(x == 0, assert x < 0) ==> x == 1
=
  (x == 0 && x < 0) ==> x == 1

// valid
```

✓

Total correctness:

Pre ==> ***WP***(S, *Post*) valid

iff { *Pre* } S { *Post* } valid

iff when program S is started in any state in *Pre*,
then S terminates in a state in *Post*.

Partial correctness:

SP(*Pre*, S) valid

iff when program S is started in any state in *Pre*
and terminates without crashing,
then S terminates in a state in *Post*.

Wrap-up

```
method foo(...)
  returns (...)
  requires Pre
  ensures Post
{
  S
}
```

```
{ Pre }
  S
{ Post }
```

Verification condition
(total correctness)

`Pre ==> WP(S, Post)` valid?

Verification condition
(partial correctness)

`SP(S, Post) ==> Pre` valid?

Operational Semantics

We will define the semantics of statements using four execution relations:

1. a relation $\Longrightarrow_a$ for evaluating arithmetic expressions in a program state;
2. a relation $\Longrightarrow_b$ for evaluating Boolean expressions in a program state;
3. a relation $\Longrightarrow_P$ for evaluating predicates in a program state; and
4. a relation $\Longrightarrow$ for performing one execution step from one program *configuration* (a statement coupled with a program state) to another configuration.

Evaluation of arithmetic expressions

For arithmetic expressions `a` in `AExpr`, we denote by $\mathtt{a}, \sigma \Longrightarrow_a \mathtt{v}$ that evaluating expression `a` in state σ yields integer value `v`.

Formally, the evaluation relation

$$\Longrightarrow_a \subseteq \ (\mathtt{AExpr} \times \mathtt{States}) \times \mathtt{Int}$$

is given by the following inference rules:

$$\frac{}{\mathtt{v}, \sigma \Longrightarrow_a \mathtt{v}} \qquad \frac{}{\mathtt{x}, \sigma \Longrightarrow_a \sigma(\mathtt{x})}$$

$$\frac{\bullet \in \{+, -, *, /, \%\} \qquad \mathtt{a1}, \sigma \Longrightarrow_a \mathtt{v1} \qquad \mathtt{a2}, \sigma \Longrightarrow_a \mathtt{v2} \qquad \mathtt{v} = \mathtt{v1} \bullet \mathtt{v2}}{\mathtt{a1} \bullet \mathtt{a2}, \sigma \Longrightarrow_a \mathtt{v}}$$

Notice that the last rule from above uses the standard arithmetic operators on integers `v1` and `v2` with one exception: we assume that all arithmetic operators are defined for every input, that is, they are total functions. If an operation is not well-defined, then it returns some value but we do not know which one. For example, evaluating `5 / 0` yields such an unknown value. We will address how to spot and report such runtime errors later in the course.

Evaluation of Boolean expressions

For Boolean expressions `b` in `BExpr`, we denote by $\mathtt{b}, \sigma \Longrightarrow_b \mathtt{t}$ that evaluating expression `b` in state σ yields truth value `t`.

Formally, the evaluation relation

$$\Longrightarrow_b \subseteq \ (\mathtt{BExpr} \times \mathtt{States}) \times \mathtt{Bool}$$

is defined by structural induction using the following inference rules:

$$\frac{}{\mathtt{true}, \sigma \Longrightarrow_b \mathtt{true}} \qquad \frac{}{\mathtt{false}, \sigma \Longrightarrow_b \mathtt{false}}$$

$$\frac{\mathtt{a1}, \sigma \Longrightarrow_a \mathtt{v1} \qquad \mathtt{a2}, \sigma \Longrightarrow_a \mathtt{v2} \qquad \mathtt{t} = (\mathtt{v1} \bowtie \mathtt{v2}) \qquad \bowtie \in \{\mathtt{==}, \mathtt{!=}, \mathtt{<=}, \mathtt{>=}, \mathtt{<}, \mathtt{>}\}}{\mathtt{a1} \bowtie \mathtt{a2}, \sigma \Longrightarrow_b \mathtt{t}}$$

$$\frac{\mathtt{b}, \sigma \Longrightarrow_b \mathtt{false}}{\mathtt{!b}, \sigma \Longrightarrow_b \mathtt{true}} \qquad \frac{\mathtt{b}, \sigma \Longrightarrow_b \mathtt{false}}{\mathtt{!b}, \sigma \Longrightarrow_b \mathtt{true}}$$

$$\frac{\mathtt{b1}, \sigma \Longrightarrow_b \mathtt{false}}{\mathtt{b1\ \&\&\ b2}, \sigma \Longrightarrow_b \mathtt{false}} \qquad \frac{\mathtt{b2}, \sigma \Longrightarrow_b \mathtt{false}}{\mathtt{b1\ \&\&\ b2}, \sigma \Longrightarrow_b \mathtt{false}} \qquad \frac{\mathtt{b1}, \sigma \Longrightarrow_b \mathtt{true} \quad \mathtt{b2}, \sigma}{\mathtt{b1\ \&\&\ b2}, \sigma \Longrightarrow_b}$$

$$\frac{\mathtt{b1}, \sigma \Longrightarrow_b \mathtt{true}}{\mathtt{b1\ ||\ b2}, \sigma \Longrightarrow_b \mathtt{true}} \qquad \frac{\mathtt{b2}, \sigma \Longrightarrow_b \mathtt{true}}{\mathtt{b1\ ||\ b2}, \sigma \Longrightarrow_b \mathtt{true}} \qquad \frac{\mathtt{b1}, \sigma \Longrightarrow_b \mathtt{false} \quad \mathtt{b2}, \sigma =}{\mathtt{b1\ ||\ b2}, \sigma \Longrightarrow_b \mathtt{fa}}$$

In the above rules, we write $\mathtt{v1} \bowtie \mathtt{v2}$ to denote the value obtained from performing a standard comparison $\bowtie$ between two integer values.

Evaluation of predicates

For predicates `P` in `Pred`, we denote by $\mathrm{P}, \sigma \Longrightarrow_b \mathtt{t}$ that evaluating predicate `P` in state σ yields truth value `t`.

Formally, the evaluation relation

$$\Longrightarrow_P \subseteq \ (\mathtt{Pred} \times \mathtt{States}) \times \mathtt{Bool}$$

is defined by structural induction using the following inference rules:

$$\frac{\mathrm{b}, \sigma \Longrightarrow_b \mathtt{t}}{\mathrm{b}, \sigma \Longrightarrow_P \mathtt{t}} \qquad \frac{\mathrm{P}, \sigma \Longrightarrow_P \mathtt{false}}{!\mathrm{P}, \sigma \Longrightarrow_P \mathtt{true}} \qquad \frac{\mathrm{P}, \sigma \Longrightarrow_P \mathtt{false}}{!\mathrm{P}, \sigma \Longrightarrow_P \mathtt{true}}$$

$$\frac{\mathtt{v} \in \mathtt{Int} \quad \mathrm{P}, \sigma[\mathtt{x\ :=\ v}] \Longrightarrow_P \mathtt{true}}{\mathtt{exists\ x\ ::\ } \mathrm{P}, \sigma \Longrightarrow_P \mathtt{true}}$$

$$\frac{\ldots \quad \mathrm{P}, \sigma[\mathtt{x\ :=\ -1}] \Longrightarrow_P \mathtt{false} \quad \mathrm{P}, \sigma[\mathtt{x\ :=\ 0}] \Longrightarrow_P \mathtt{false} \quad \mathrm{P}, \sigma[\mathtt{x\ :=\ 1}] \Longrightarrow_P \mathtt{fa}}{\mathtt{exists\ x\ ::\ } \mathrm{P}, \sigma \Longrightarrow_P \mathtt{false}}$$

$$\frac{\mathtt{v} \in \mathtt{Int} \quad \mathtt{P}, \sigma[\mathtt{x\ :=\ v}] \Longrightarrow_P \mathtt{false}}{\mathtt{forall\ x\ ::\ P}, \sigma \Longrightarrow_P \mathtt{false}}$$

$$\frac{\ldots \quad \mathtt{P}, \sigma[\mathtt{x\ :=\ -1}] \Longrightarrow_P \mathtt{true} \quad \mathtt{P}, \sigma[\mathtt{x\ :=\ 0}] \Longrightarrow_P \mathtt{true} \quad \mathtt{P}, \sigma[\mathtt{x\ :=\ 1}] \Longrightarrow_P \mathtt{true}}{\mathtt{forall\ x\ ::\ P}, \sigma \Longrightarrow_P \mathtt{true}}$$

$$\frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{P1\ \&\&\ P2}, \sigma \Longrightarrow_P \mathtt{false}} \qquad \frac{\mathtt{P2}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{P1\ \&\&\ P2}, \sigma \Longrightarrow_P \mathtt{false}}$$

$$\frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{true} \quad \mathtt{P2}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{P1\ \&\&\ P2}, \sigma \Longrightarrow_P \mathtt{true}}$$

$$\frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{P1\ ||\ P2}, \sigma \Longrightarrow_P \mathtt{true}} \qquad \frac{\mathtt{P2}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{P1\ ||\ P2}, \sigma \Longrightarrow_P \mathtt{true}}$$

$$\frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{false} \quad \mathtt{P2}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{P1\ ||\ P2}, \sigma \Longrightarrow_P \mathtt{false}}$$

$$\frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{true} \quad \mathtt{P2}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{P1\ ==>\ P2}, \sigma \Longrightarrow_P \mathtt{true}} \qquad \frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{true} \quad \mathtt{P2}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{P1\ ==>\ P2}, \sigma \Longrightarrow_P \mathtt{false}}$$

$$\frac{\mathtt{P1}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{P1\ ==>\ P2}, \sigma \Longrightarrow_P \mathtt{true}}$$

Program Configurations

A *program configuration* is either a pair $(\mathtt{S}, \sigma)$ consisting of a statement $\mathtt{S}$ and a program state σ, or a pair $(\mathtt{done}, \sigma)$ indicating termination with final state σ, or $\mathtt{error}$, which indicates the occurrence of a runtime error, or $\mathtt{magic}$ indicating that we ended up in a magical place where anything is possible (i.e., we can verify any property, even wrong ones). Formally, the set $\mathtt{Conf}$ of program configurations is given by

$$\mathtt{Conf} \ ::= \ (\mathtt{Stmt} \times \mathtt{States}) \cup (\{\mathtt{done}\} \times \mathtt{States}) \cup \{\mathtt{error}, \mathtt{magic}\}\,.$$

Execution relation for statements

For `PL0` statements `S` in `Stmt`, we denote by $\mathtt{S}, \sigma \Longrightarrow C$ that executing one step starting in configuration $(\mathtt{S}, \sigma)$ results in configuration C. Formally, the execution relation

$$\Longrightarrow \subseteq \ \mathtt{Conf} \times \mathtt{Conf}$$

is defined by structural induction using the following inference rules:

$$\frac{\mathtt{v} \in \mathtt{Int}}{\mathtt{var\ \ x}, \sigma \Longrightarrow \mathtt{done}, \sigma[\mathtt{x\ \ :=\ \ v}]}$$

$$\frac{\mathtt{a}, \sigma \Longrightarrow_a \mathtt{v}}{\mathtt{x\ \ :=\ \ a}, \sigma \Longrightarrow \mathtt{done}, \sigma[\mathtt{x\ \ :=\ \ v}]}$$

$$\frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{assert\ P}, \sigma \Longrightarrow \mathtt{done}, \sigma} \qquad \frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{assert\ P}, \sigma \Longrightarrow \mathtt{error}}$$

$$\frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{assume\ P}, \sigma \Longrightarrow \mathtt{done}, \sigma} \qquad \frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{assume\ P}, \sigma \Longrightarrow \mathtt{magic}}$$

$$\frac{\mathtt{S1}, \sigma \Longrightarrow \mathtt{S1}', \sigma'}{\mathtt{S1;S2}, \sigma \Longrightarrow \mathtt{S1}';\mathtt{S2}, \sigma'} \qquad \frac{\mathtt{S1}, \sigma \Longrightarrow \mathtt{done}, \sigma'}{\mathtt{S1;S2}, \sigma \Longrightarrow \mathtt{S2}, \sigma'}$$

$$\frac{\mathtt{S1}, \sigma \Longrightarrow \mathtt{error}}{\mathtt{S1;S2}, \sigma \Longrightarrow \mathtt{error}} \qquad \frac{\mathtt{S1}, \sigma \Longrightarrow \mathtt{magic}}{\mathtt{S1;S2}, \sigma \Longrightarrow \mathtt{magic}}$$

$$\frac{}{\mathtt{S1\ \ []\ \ S2}, \sigma \Longrightarrow \mathtt{S1}, \sigma} \qquad \frac{}{\mathtt{S1\ \ []\ \ S2}, \sigma \Longrightarrow \mathtt{S2}, \sigma}$$

Notice that we can always perform at least one step of the execution relation until we reach a final configuration of the form `error`, `magic`, or `(done, _)`.

Floyd-Hoare triples

Extended execution relation

Towards a formal definition of Floyd-Hoare triples, we first extend the execution relation $\Longrightarrow$ to perform exactly $n \geq 0$ (as opposed to exactly one) execution steps:

- $C \Longrightarrow^0 C$, and
- $C \Longrightarrow^{n+1} C'$ if $C \Longrightarrow C''$ and $C'' \Longrightarrow^n C'$.

We write $C \Longrightarrow^* C'$ if there exists some $n \geq 0$ such that $C \Longrightarrow^n C'$. Furthermore, we write $C \not\Longrightarrow^n$ if there exists *no* configuration C' such that $C \Longrightarrow^n C'$.

Formal Definition

A **Floyd-Hoare triple** $\{\mathrm{P}\}\ \mathtt{S}\ \{\mathrm{Q}\}$ consists of a precondition P in `Pred`, a program statement $\mathtt{S}$, and a postcondition $\mathtt{Q}$ in `Pred`.

Intuitively, a triple is valid if every execution of program $\mathtt{S}$ that starts in a state satisfying the precondition P will not cause a runtime error (for example, by failing an assertion) and will eventually terminate in a state satisfying the postcondition $\mathtt{Q}$.

Towards a formal definition, recall from the definition of our language that $dom(\sigma)$ denotes the variables for which the state σ is defined. Moreover, let $\mathtt{Var}(\mathrm{P})$ denote all free variables that appear in predicate P. Then $\mathtt{Var}(\mathrm{P}) \subseteq dom(\sigma)$ means that state σ assigns values at least to all variables in predicate P.

The Floyd-Hoare triple $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ is *valid* (for total correctness) if and only if for every state σ with $\mathtt{Var}(\mathtt{P}) \subseteq dom(\sigma)$, if $\mathtt{P}, \sigma \Longrightarrow_P \mathtt{true}$, then

1. (safety) there exists no $n \geq 0$ such that $\mathtt{S}, \sigma \Longrightarrow^n \mathtt{error}$;
2. (termination) there exists $n \geq 0$ such that for every $m \geq n$, $\mathtt{S}, \sigma \not\Longrightarrow^m$; and
3. (partial correctness) if $\mathtt{S}, \sigma \Longrightarrow^* (\mathtt{done}, \sigma')$ and $\mathtt{Var}(\mathtt{Q}) \subseteq dom(\sigma')$, then $\mathtt{Q}, \sigma' \Longrightarrow_P \mathtt{true}$.

Here, the first condition (safety) ensures that we cannot encounter a runtime error (for example an assertion failure) when starting execution in a state that satisfies the precondition. The second condition ensures termination, that is, we cannot perform execution steps ad infinitum. The third condition makes sure that we satisfy the postcondition whenever we manage to terminate without a runtime error.

We say that a triple is valid for *partial correctness*, whenever the third (but not necessarily the first or second) property holds.

In principle, we can directly apply the above definition of validity together with our operational semantics to check whether a triple is valid. However, in most cases this is not feasible, since we would have to consider all possible program executions.

Instead, we use verification conditions to reason about the validity of a triple in a symbolic way, that is by reasoning about predicates, without invoking the operational program semantics. In the lecture, we considered two such approaches for constructing verification conditions: weakest preconditions and strongest postconditions.

Weakest preconditions

Weakest preconditions generate a verification condition for checking the validity of Floyd-Hoare triples by starting at the postcondition and moving backward through the program.

Definition

The *weakest precondition* of a program statement `S` and a postcondition `Q` is the predicate `WP` describing the largest possible set of states such that the Floyd-Hoare triple

```
{ WP } S { Q }
```

is valid for total correctness. In other words, whenever `{ P } S { Q }` is valid, then `P` implies `WP`.

Intuition

The weakest precondition is a predicate, which describes all possible initial states (think: inputs) on which we can run program `S` such that it terminates without runtime errors in final states (think: outputs) specified by the postcondition `Q`.

When determining the weakest precondition of a program `S` and a postcondition `Q`, we thus aim to modify `Q` in a way that reverts changes made to the state by `S`. The result is a predicate that is true for a given initial state whenever running `S` on that initial state leads to a final state satisfying `Q` again. Thus, one can think about the weakest precondition as the predicate that anticipates the truth value of postcondition `Q` but *before* (that is, evaluated in initial states) rather than *after* (that is, evaluated in final states) running program `S`.

Computation

We typically do not need to apply the above (semantic) definition, since we can compute weakest preconditions (syntactically) on the program structure using the following recursive definition:

```
S                        WP(S, Q)
================================================
var x                    forall x :: Q
assert R                 R && Q
assume R                 R ==> Q
x := a                   Q[x / a]
S1; S2                   WP(S1, WP(S2, Q))
S1 [] S2                 WP(S1, Q) && WP(S2, Q)
```

Here, `Q[x / a]` is the predicate `Q` in which every (free) appearance of `x` has been replaced by `a`. A formal definition is given below.

We briefly go over the above rules for computing weakest preconditions.

Variable declarations

The variable declaration `var x` declares `x` as an unitialized (integer-valued) variable named `x`. All information about the value of some other variables named `x` that might have existed before are forgotten. This is reflected by the operational rule

$$\frac{\mathtt{v} \in \mathtt{Int}}{\mathtt{var\ x}, \sigma \Longrightarrow \mathtt{done}, \sigma[\mathtt{x\ :=\ v}]}$$

which nondeterministically assigns any integer $\mathtt{v}$ to `x`.

Now, assume that postcondition `Q` holds for the final state $\sigma[\mathtt{x\ :=\ v}]$. What are the possible initial states such that executing `var x` could lead to such a state?

In fact, such states must coincide with $\sigma[\mathtt{x\ :=\ v}]$ for all variables except `x`, since `var x` does not change any other variable. Moreover, the value of `x` does not matter at all, since `x` will be set to an arbitrary integer by executing `var x`.

Hence, we can (but, due to nondeterminism do not have to) reach $\sigma[\mathtt{x\ :=\ v}]$ from any state $\sigma[\mathtt{x\ :=\ u}]$ for all initial values $\mathtt{u}$ of `x`.

As a predicate, we can use a universal quantifier over `x` to express that the value of `x` does not matter. This yields `WP(var x, Q) ::= forall x :: Q`.

Assignments

The assignment `x := a` evaluates expression `a` in the current (initial) program state and assigns the resulting value to variable `x`. This is reflected by the operational rule

$$\frac{\mathtt{a}, \sigma \Longrightarrow_{a} \mathtt{v}}{\mathtt{x\ :=\ a}, \sigma \Longrightarrow \mathtt{done}, \sigma[\mathtt{x\ :=\ v}]}$$

Similarly to the variable declaration, only variable `x` is potentially changed by the assignment. However, the original value of `x` now does matter, since we may need it to evaluate expression `a`. For example, the assignment `x := x + 1` depends on the value of `x` in the initial state.

Now, assume that postcondition `Q` holds for the final state $\sigma[\mathbf{x} := \mathbf{v}]$. What are the possible initial states such that executing `x := a` leads to such a state?

Intuitively, we would like to undo the change made by the assignment. For example, if `x >= 17` holds after the assignment `x := x + 1`, then `x >= 16` should hold before the assignment. One way to do this is to observe that the expression `x + 1` is evaluated in initial states, just like the weakest precondition. We can thus use it to refer to the value `v`. To undo the assignment, we then replace every occurrence of `x` by `x+1` and obtain `x + 1 >= 17`.

The effect of such a replacement is that the effect of setting `x` to the value of `x+1` now appears on both sides of every constraint involving `x`. For the arithmetic operations in our language, they will thus cancel each other out. More concretely, `x + 1 >= 17` is equivalent to `x >= 16`, our desired precondition.

Formally, the weakest precondition of assignments is defined as `WP(x:=a, Q) ::= Q[x/a]`, that is we formally substitute `x` by `a` in `Q`.

The following lemma justifies the above intuition, that is, we can substitute `x` by `a` to "undo" the effect of executing the assignment `x := a`:

Substitution Lemma Consider a predicate `Q`, an arithmetic expression `a`, and a state σ. Moreover, assume that `a` evaluates to value `v` for state σ, that is, $\mathtt{a}, \sigma \rightarrow \mathtt{v}$.

Then, $\sigma[\mathtt{x} \ := \ \mathtt{v}] \models Q$ if and only if $\sigma \models Q[\mathtt{x} \ / \ \mathtt{a}]$.

Assertions

The assertion `assert R` checks whether the predicate `R` holds in the current (initial) state and causes a runtime error if this is not the case; otherwise, it has no effect. This is reflected by two operational rules, one for the case that `R` holds and one for the case that it does not hold:

$$\frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{assert\ P}, \sigma \Longrightarrow \mathtt{done}, \sigma} \qquad \frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{assert\ P}, \sigma \Longrightarrow \mathtt{error}}$$

Now, assume that postcondition `Q` holds for the final state σ. We do not consider the configuration `error` because reason about total correctness; a triple is not valid if we can reach `error` from an initial state that satisfies the precondition.

What are the possible initial states such that executing `assert R` leads to σ? We first observe that the assertion does not change σ: if `Q` holds after execution of `assert R`, then it also holds before its execution. We then reach the final state σ if and only if initial state σ additionally satisfies the predicate `R`.

Hence, the weakest precondition of assertions is defined as `WP(assert R, Q) ::= R && Q`.

Assumptions

The assumption `assume R` is a somewhat strange statement because we cannot execute it on an actual machine; it is a verification-specific statement, similar to making an assumption in a mathematical proof. Intuitively, `assume R` checks whether the predicate `R` holds in the current (initial) state. Similar to an assertion, there is no effect if the predicate holds. However, in contrast to an assertion, we reach a "magical state" in which everything goes if the predicate does not hold. This corresponds to the case that we have made a wrong assumption in a proof: from a wrong assumption we can conclude whatever we want, so the result becomes worthless. This is reflected by two operational rules, one for the case that `R` holds and one for the case that it does not hold:

$$\frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{true}}{\mathtt{assume}\ \ \mathtt{P}, \sigma \Longrightarrow \mathtt{done}, \sigma} \qquad \frac{\mathtt{P}, \sigma \Longrightarrow_P \mathtt{false}}{\mathtt{assume}\ \ \mathtt{P}, \sigma \Longrightarrow \mathtt{magic}}$$

Now, assume that postcondition `Q` holds for the final state σ. As for assertions, we do not consider the configuration `magic`; executions that move to `magic` cannot invalidate a triple anyway.

What are the possible initial states such that executing `assume R` leads to σ? We first observe that the assertion does not change σ: if `Q` holds after execution of `assume R`, then it also holds before its execution. There are now two cases:

1. σ satisfies `R` and postcondition `Q`, or
2. σ does not satisfy `R` and we thus do not care about the result due to a wrong assumption; the weakest precondition should be `true` in this case, since everything follows from a wrong assumption.

Formally, this can be expressed as the predicate `(R && Q) || !R` or, equivalently, `WP(assume R) ::= R ==> Q`.

Sequential composition

The sequential composition `S1;S2` first executes program `S1` and then program `S2`. This is reflected by four operational rules

$$\frac{\texttt{S1}, \sigma \Longrightarrow \texttt{S1}', \sigma'}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{S1}'\texttt{;S2}, \sigma'} \qquad \frac{\texttt{S1}, \sigma \Longrightarrow \texttt{done}, \sigma'}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{S2}, \sigma'}$$

$$\frac{\texttt{S1}, \sigma \Longrightarrow \texttt{error}}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{error}} \qquad \frac{\texttt{S1}, \sigma \Longrightarrow \texttt{magic}}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{magic}}$$

Here, the first rule keeps executing `S1`; termination of `S1` is handled by the second rule, which moves on with executing `S2`. The last two rules propagate the case that we already encountered a runtime error or a wrong assumption.

What are the possible initial states such that executing `S1;S2` leads to final states that satisfy postcondition `Q`?

By the inductive definition of weakest preconditions, we can assume that we already know how to construct weakest preconditions for `S1` and `S2`, namely `WP(S2, Q)` and `WP(S1, WP(S2, Q))`. The set of all initial states on which we can run `S2` to end up in a state that satisfies `Q` is given by `WP(S2, Q)`. We can use this predicate as the postcondition of program `S1`. By the same argument, `WP(S1, WP(S2, Q))` is the set of all initial states on which we can run `S1` to end up in a state that satisfies `WP(S2, Q)`.

Putting both arguments together, `WP(S1, WP(S2, Q))` is the set of all initial states such that running `S1` followed by `S2` leads us to states that satisfy `Q`. Hence, the weakest precondition of sequential composition is defined as `WP(S1;S2, Q) ::= WP(S1, WP(S2, Q))`.

Nondeterministic choice

The nondeterministic choice `S1 [] S2` executes either program `S1` or program `S2`; we do not know which one. This is reflected by two operational rules, one for running `S1` and one for running `S2`:

$$\frac{}{\texttt{S1 [] S2}, \sigma \Longrightarrow \texttt{S1}, \sigma} \qquad \frac{}{\texttt{S1 [] S2}, \sigma \Longrightarrow \texttt{S2}, \sigma}$$

What are the possible initial states such that executing `S1 [] S2` leads to final states that satisfy postcondition `Q`? By the inductive definition of weakest preconditions, we can assume that we already know how to construct weakest preconditions for `S1` and `S2`, namely `WP(S1, Q)` and `WP(S2, Q)`. Since we do not know whether `S1` or `S2` is executed, we can only guarantee that we will certainly reach a state that satisfies `Q` if we start in a state from which it does not matter whether we run `S1` or `S2`. Our desired initial states must thus satisfy both `WP(S1, Q)` and `WP(S2, Q)`.

Hence, the weakest precondition of nondeterministic choice is defined as `WP(S1 [] S2, Q) ::= WP(S1, Q) && WP(S2, Q)`.

Variable substitution

`F[x/a]` denotes the *substitution* of every (free) occurence of variable `x` by expression `a` in `F`. It is defined recursively as follows:

```
F                       F[x/a]
============================================
true                     true
false                    false
x                        a
y                        y    (where x != y)
v                        v
a1 <> a2                 a1[x/a] <> a2[x/a]
                             for <> in { ==, !=, <=, >=, <, >, +, *, - /, % }
Q1 && Q2                 Q1[x/a] && Q2[x/a]
Q2 || Q2                 Q1[x/a] || Q2[x/a]
!Q1                      !(Q1[x/a])
Q1 ==> Q2                Q1[x/a] ==> Q2[x/a]
exists y :: Q1           exists y :: Q1[x/a]
forall y :: Q1           forall y :: Q1[x/a]
```

Strongest postconditions

Strongest postconditions generate a verification condition for checking the validity of Floyd-Hoare triples by starting at the precondition and moving forward through the program, just like an ordinary program execution.

Definition

The *strongest postcondition* of a program statement `S` and a pprecondition `P` is the predicate `SP` describing the smallest possible set of states such that the Floyd-Hoare triple

```
{ P } S { SP }
```

is valid for *partial* correctness. In other words, whenever `{ P } S { Q }` is valid for partial correctness, then `SP` implies `Q`.

Notice that strongest postcondition on their own only make sense for partial correctness. For example, how should we choose `SP` such that `{ true } assert false { SP }` is valid? There is no `SP` such that the triple is valid for total correctness. For partial correctness, we have `SP = false` and, in fact, every postcondition leads to a triple that is valid for partial correctness.

Intuition

The strongest postcondition of a program `S` and a precondition `P` is the predicate that consists of all final states in which program `S` terminates when started on states that satisfy `P`. It thus corresponds to symbolically executing the program, that is, we run the program by modifying sets of states - described by predicates - instead of single states.

When determining strongest postconditions, we thus aim to modify the precondition such that it captures the changes made to the program states by the executed program.

Computation

Similar to weakest preconditions, we can compute strongest postconditions on the program structure using the following recursive definition:

```
S                       SP(P, S)
==================================================================
var x                   exists x :: P
assert R                P && R
assume R                P && R
x := a                  exists x0 :: P[x / x0] && x == a[x / x0]
S1; S2                  SP(SP(P, S1), S2)
S1 [] S2                SP(P, S1) || SP(P, S2)
```

Here, `P[x / x0]` is the predicate `P` in which every (free) appearance of `x` has been substituted by `x0`.

As for weakest preconditions, we briefly go over the above rules for computing strongest postconditions.

Variable declarations

The variable declaration `var x` declares `x` as an unitialized (integer-valued) variable named `x`. All information about the value of some other variables named `x` that might have existed before are forgotten. This is reflected by the operational rule

$$\frac{\mathbf{v} \in \mathtt{Int}}{\mathtt{var\ x}, \sigma \Longrightarrow \mathtt{done}, \sigma[\mathtt{x\ :=\ v}]}$$

which nondeterministically assigns any integer $\mathbf{v}$ to `x`.

Now, assume that precondition `P` holds for the initial state σ. What are the final states that we can reach by running `var x` on σ? The state $\sigma[\mathtt{x\ :=\ v}]$ for some (nondeterministically chosen) integer v. Hence, the strongest postcondition is `sp(P, var x) ::= exists x :: P`.

Assignments

The assignment `x := a` evaluates expression `a` in the current (initial) program state and assigns the resulting value to variable `x`. This is reflected by the operational rule

$$\frac{\mathbf{a}, \sigma \Longrightarrow_a \mathbf{v}}{\mathtt{x\ :=\ a}, \sigma \Longrightarrow \mathbf{done}, \sigma[\mathtt{x\ :=\ v}]}$$

Similarly to the variable declaration, only variable `x` is potentially changed by the assignment.

Now, assume that precondition `P` holds for the initial state σ. What are the possible final states that we can reach by executing `x := a` on σ?

A first guess might be to define `SP(P, x := a) ::= P && x == a`, that is, `P` should still hold and after the assignment `x` equals `a`. However, the following triples are *not* valid:

- `{ x == 4 } x := 7 { x == 4 && x == 7 }`
- `{ true } x := x + 1 { true && x == x + 1 }`

Hence, the above proposal is not necessarily correct if `P` or `a` depend on `x`. The underlying issue is that both `P` and `a` are originally evaluated in initial states but the strongest postconditions talks about final states. The variable `x` is the only variable that might have a different value before and after executing the assignment.

To define the correct strongest postcondition, we introduce an auxiliary variable, say `x0`, that represents the original value of `x` when evaluated in some initial state. The predicate `P[x / x0]` then means that precondition `P` was true in some initial state; similarly, `a[x/x0]` refers to the evaluation of `a` in the same initial state. Put together, we then use the original proposal but state that `P` and `a` were true in some initial state.

Hence, the strongest postcondition of assignments is defined as `SP(P, x:=a) ::= exists x0 :: P[x / x0] && x == a[x / x0]`.

As an example, let consider precondition `x == 4` and the assignment `x := x + 7`:

```
sp(x == 4, x := x + 7)
= (by definition)
exists x0 :: x0 == 4 && x == x0 + 7
= (simplification)
x == 11
```

Indeed, `x` will be 11 after running `x := x + 7` on a state where `x` is initially 4.

Assumptions

The assumption `assume R` is a somewhat strange statement because we cannot execute it on an actual machine; it is a verification-specific statement, similar to making an assumption in a mathematical proof. Intuitively, `assume R` checks whether the predicate `R` holds in the current (initial) state; there is no effect if the predicate holds; we reach a "magical state" in which everything goes if the predicate does not hold. This is reflected by two operational rules, one for the case that `R` holds and one for the case that it does not hold:

$$\frac{\mathrm{P}, \sigma \Longrightarrow_{p} \texttt{true}}{\texttt{assume P}, \sigma \Longrightarrow \texttt{done}, \sigma} \qquad \frac{\mathrm{P}, \sigma \Longrightarrow_{p} \texttt{false}}{\texttt{assume P}, \sigma \Longrightarrow \texttt{magic}}$$

Now, assume that precondition `P` holds for the initial state σ. We do not consider the configuration `magic`; executions that move to `magic` cannot invalidate a triple anyway.

What are the possible final states that we can reach by executing `assume R` on σ?

Since we do not have to consider executions that move to `magic`, we know that σ satisfies `R`. Moreover, since the assumption that does not change σ, `P` also holds in the final state.

Hence, the strongest postcondition of assumptions is defined as `SP(P, assume R) ::= P && R`.

Assertions

The assertion `assert R` checks whether the predicate `R` holds in the current (initial) state and causes a runtime error if this is not the case; otherwise, it has no effect. Since we reason about partial correctness with strongest postconditions, it does not matter whether we reach an error configuration or a magical configuration.

The strongest postcondition of the assertion `assert R` thus does not differ from the strongest postcondition of the assumption `assume R`. That is, `SP(P, assert R) ::= P && R`.

Sequential composition

The sequential composition `S1;S2` first executes program `S1` and then program `S2`. This is reflected by four operational rules

$$\frac{\texttt{S1}, \sigma \Longrightarrow \texttt{S1}', \sigma'}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{S1}'\texttt{;S2}, \sigma'} \qquad \frac{\texttt{S1}, \sigma \Longrightarrow \texttt{done}, \sigma'}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{S2}, \sigma'}$$

$$\frac{\texttt{S1}, \sigma \Longrightarrow \texttt{error}}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{error}} \qquad \frac{\texttt{S1}, \sigma \Longrightarrow \texttt{magic}}{\texttt{S1;S2}, \sigma \Longrightarrow \texttt{magic}}$$

Here, the first rule keeps executing `S1`; termination of `S1` is handled by the second rule, which moves on with executing `S2`. The last two rules propagate the case that we already encountered a runtime error or a wrong assumption.

What are the possible final states that we can reach when executing `S1;S2` on some initial state that satisfies precondition `P`?

By the inductive definition of strongest postconditions, we can assume that we already know how to construct strongest postconditions for `S1` and `S2`, namely `SP(P, S1)` and `SP(SP(P, S1), S2)`.

`SP(P, S1)` is then the set of all final states we can reach after running `S1` on a state satisfying `P`. We then use this predicate as the precondition of `S2`. By the same argument, `SP(SP(P, S1), S2)` is then the set of final states we can reach after running `S2` on a state satisfying `SP(P, S1)`.

Putting both arguments together, `SP(SP(P, S1), S2)` is the set of all final states reached after running `S1` followed by `S2` on initial states that satisfy `P`.
Hence, the strongest postcondition of sequential composition is defined as `SP(P, S1;S2) ::= SP(SP(P, S1), S2)`.

Nondeterministic choice

The nondeterministic choice `S1 [] S2` executes either program `S1` or program `S2`; we do not know which one. This is reflected by two operational rules, one for running `S1` and one for running `S2`:

$$\frac{}{\texttt{S1 [] S2}, \sigma \Longrightarrow \texttt{S1}, \sigma} \qquad \frac{}{\texttt{S1 [] S2}, \sigma \Longrightarrow \texttt{S2}, \sigma}$$

If we run `S1 [] S2` on an initial states that satisfy precondition `P`, then we end up in the states reached by executing `S1` on `P` or in the states reached by executing `S2` on `P`, respectively.

Hence, the strongest postcondition of nondeterministic choice is defined as `SP(P, S1 [] S2) ::= SP(P, S1) || SP(P, S2)`.

Summary

- Program verification means proving that a program meets its specification.
- How automated program verifcation compares, on a high level, to other techniques that aim to increase our confidence in the correctness of software.
- What is the look and feel of automated verifiers that follow the principle of *verification like compilation*, including *modular reasoning* with *contracts*, *incremental* construction of verified code, and the notion of a *trusted codebase*.
- Program verification relies on rigorous mathematical definitions. We formalize the verification problem using *Floyd-Hoare triples*. There are different notions of validity or correctness for such triples; the strongest one is *total correctness*.
- To check whether a Floyd-Hoare triple is valid, we attempt to construct a *verification condition*, a logical predicate, which we can then check for validity, that is, it must be true for all possible program states.
- There are two systematic approaches for constructing verification conditions: start at the precondition and construct a suitable predicate by moving forward through the program or start at the postcondition and move backward through the program. The resulting *predicate transformers* compute the *strongest postcondition* and the *weakest precondition*, respectively.