02245 – PROGRAM VERIFICATION

Christoph Matheja

(some slides have been developed together with Peter Müller)

Fall 2022

Outline

1. Why Program Verification?

2. Course Overview

3. Course Organization

4. Getting Started

How much confidence do we have in computer systems?

more confidence

Testing is insufficient

- 1994 Intel® Pentium® Floating-point Division bug
- Estimate: 1 in 9 billion floating-point divisions inaccurate
- Issue: missing entries in the lookup table
- Recall losses: $475 million (> 5 billion DKK in 2019)
- Bug was detected during experiments on number theory

extensive testing

no confidence

How much confidence do we have in computer systems?

more confidence

extensive testing

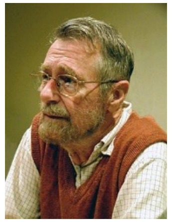
no confidence

OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

- TimSort: default sorting algorithm in OpenJDK and Android SDK
- Certain large arrays (>= 67M) lead to index-out-of-bounds errors
- Multiple attempts to fix related errors were ineffective

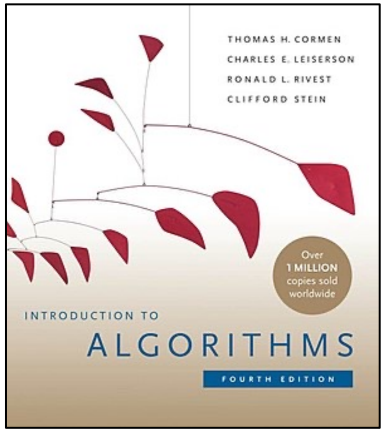
> Program testing can be very effective to show the presence of bugs, but it is hopelessly inadequate for showing their absence.



Edsger W. Dijkstra

How much confidence do we have in computer systems?

more confidence

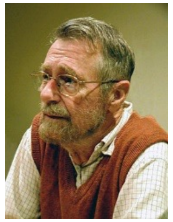


correctness arguments

extensive testing

no confidence

> The only effective way to raise the confidence level of a program is to give a convincing proof of its correctness.



Edsger W. Dijkstra

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

At the beginning of each loop iteration:

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

credits: Cormen et al., Introduction to Algorithms, 2009

Textbook-style correctness arguments are insufficient

- Binary search in `java.util.Arrays` (2006)
- Faithful implementation of algorithm from *Programming Pearls, Bentley, 1986*

Is this implementation correct?

```
public static int binarySearch(
    int[] a, int key) {
  int low = 0;
  int high = a.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;
    int midVal = a[mid];

    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1); // key not found
}
```

How much confidence do we have in computer systems?

more confidence

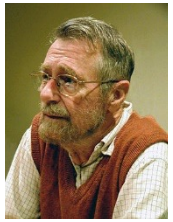
correctness proofs

correctness arguments

extensive testing

no confidence

The only effective way to raise the confidence level of a program is to give a convincing proof of its correctness.



Edsger W. Dijkstra

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica*, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan[†]
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance. Chord is simple, routing a key through a se-

All 7 claimed invariants turned out to be **incorrect**!

How much confidence do we have in computer systems?

more confidence

machine-checked proofs

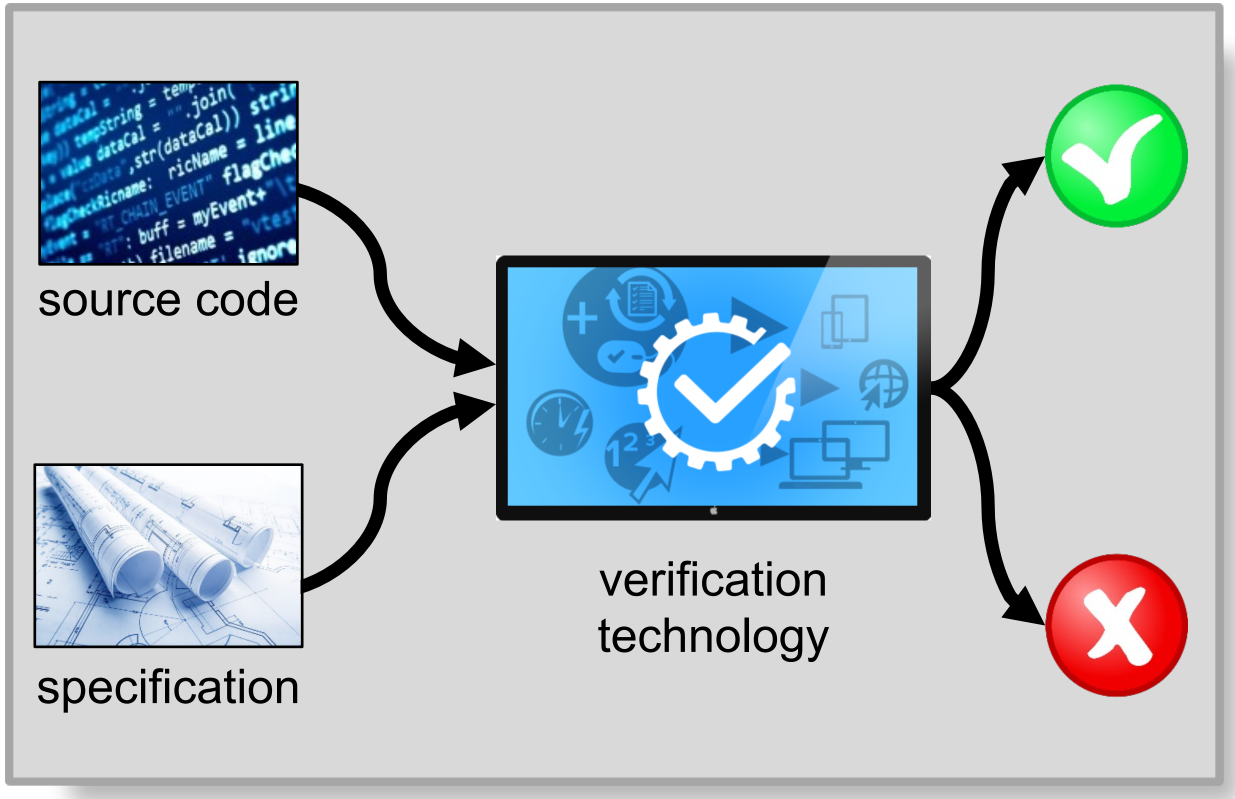
correctness proofs

correctness arguments

extensive testing

no confidence

our focus: deductive verification tools



Interactive verification

- Success stories:
  - CompCert: formally verified C compiler (2008)
  - seL4: formally verified high-performance operating system microkernel (2009)
  - EveryCrypt: formally verified crypto library (2020)

- Strengths:
  - Can handle complex systems and properties
  - Well-established trusted code base

- Weaknesses
  - Requires expert knowledge
  - Very labor-intensive (CompCert: > 6 person years)
  - Possible detachment from production code or vendor lock-in

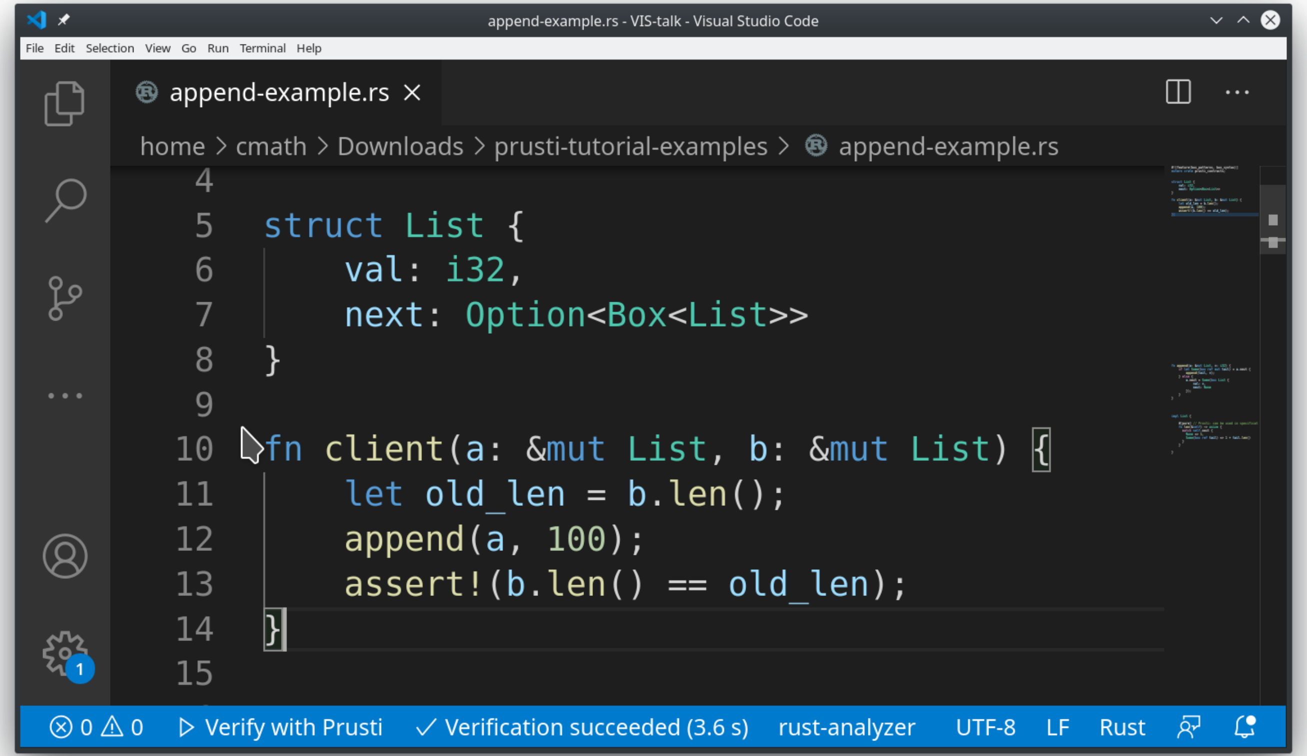
Automated (or auto-active) Verification

- Idea: “use verification like compilation”
  - Specifications take the form of source code annotations
  - Analogies: TypeScript, Rust ownership & traits, Python type hints

- Strengths:
  - Substantially less effort than interactive verification
  - Integrates into existing development processes
  - More annotations ➔ more correctness guarantees

- Weaknesses:
  - Less expressive than interactive verification
  - May produce false positives (due to undecidability)
  - Still requires effort and expertise

Prusti – a Rust Verifier



(live demo)

(more examples in teaser video)

Outline

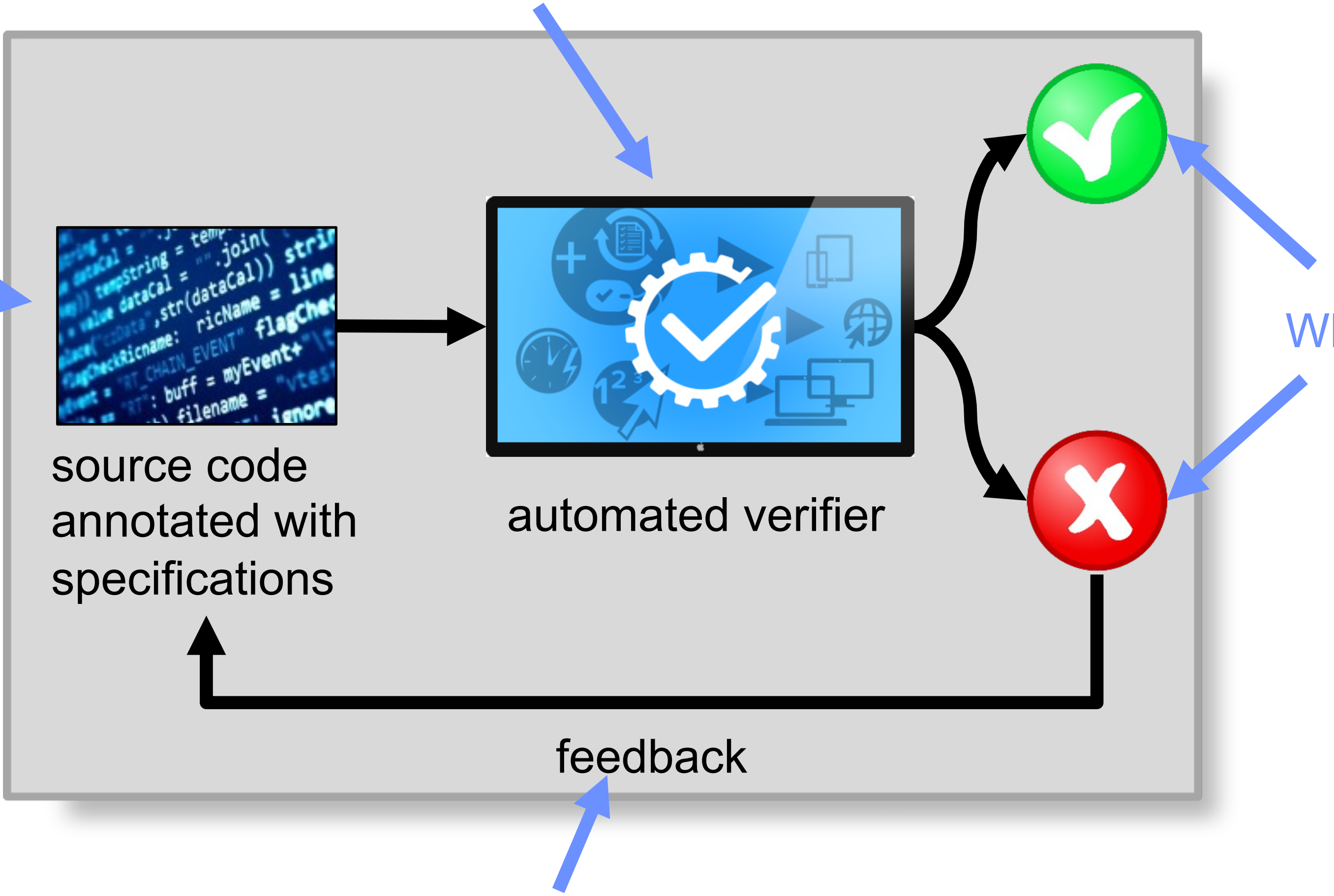
1. Why Program Verification?
2. Course Overview
3. Course Organization
4. Getting Started

Course objectives

How does this work?
How do we apply and implement this?

How to specify properties?

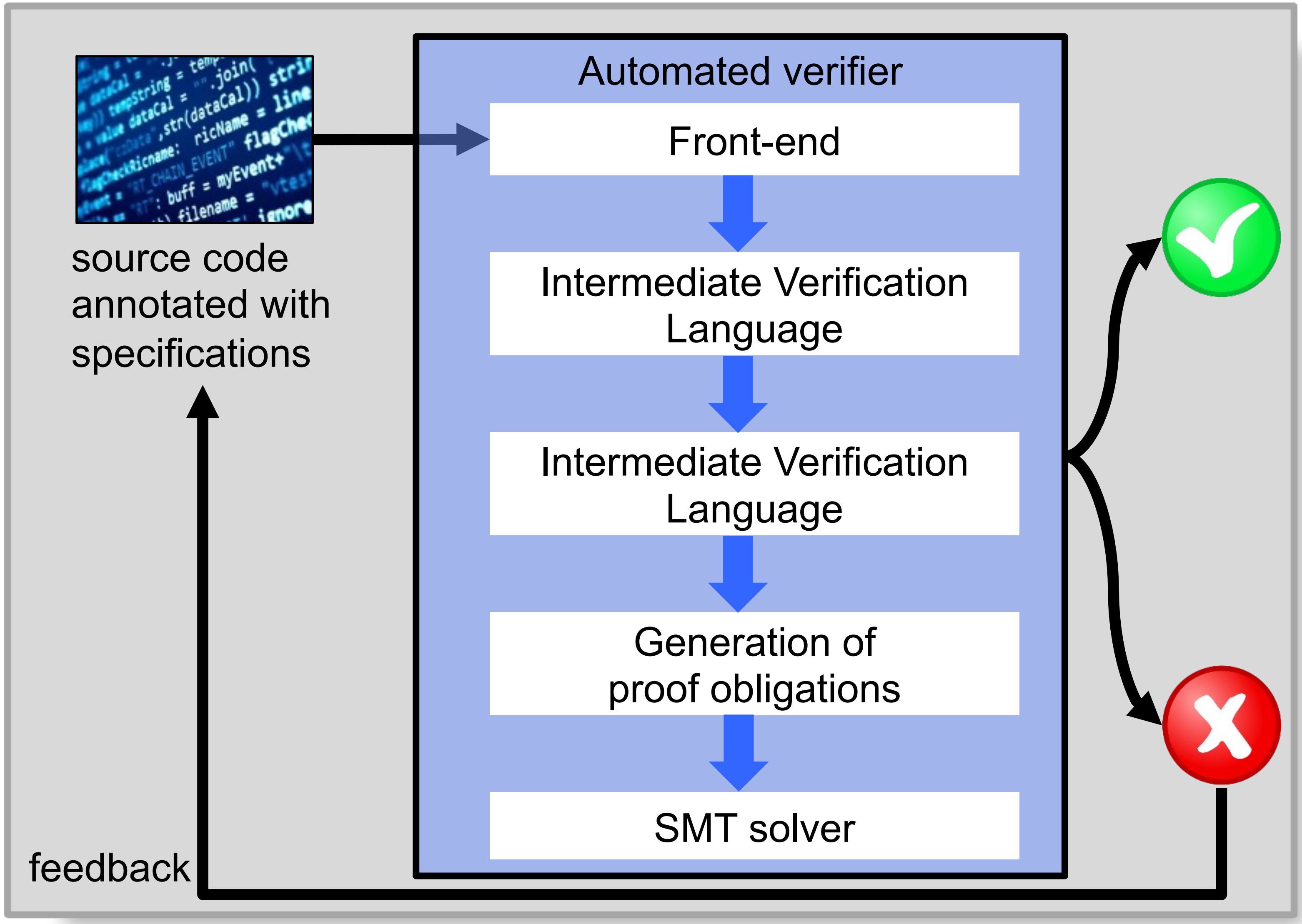
How to write good specifications?



Why these results?

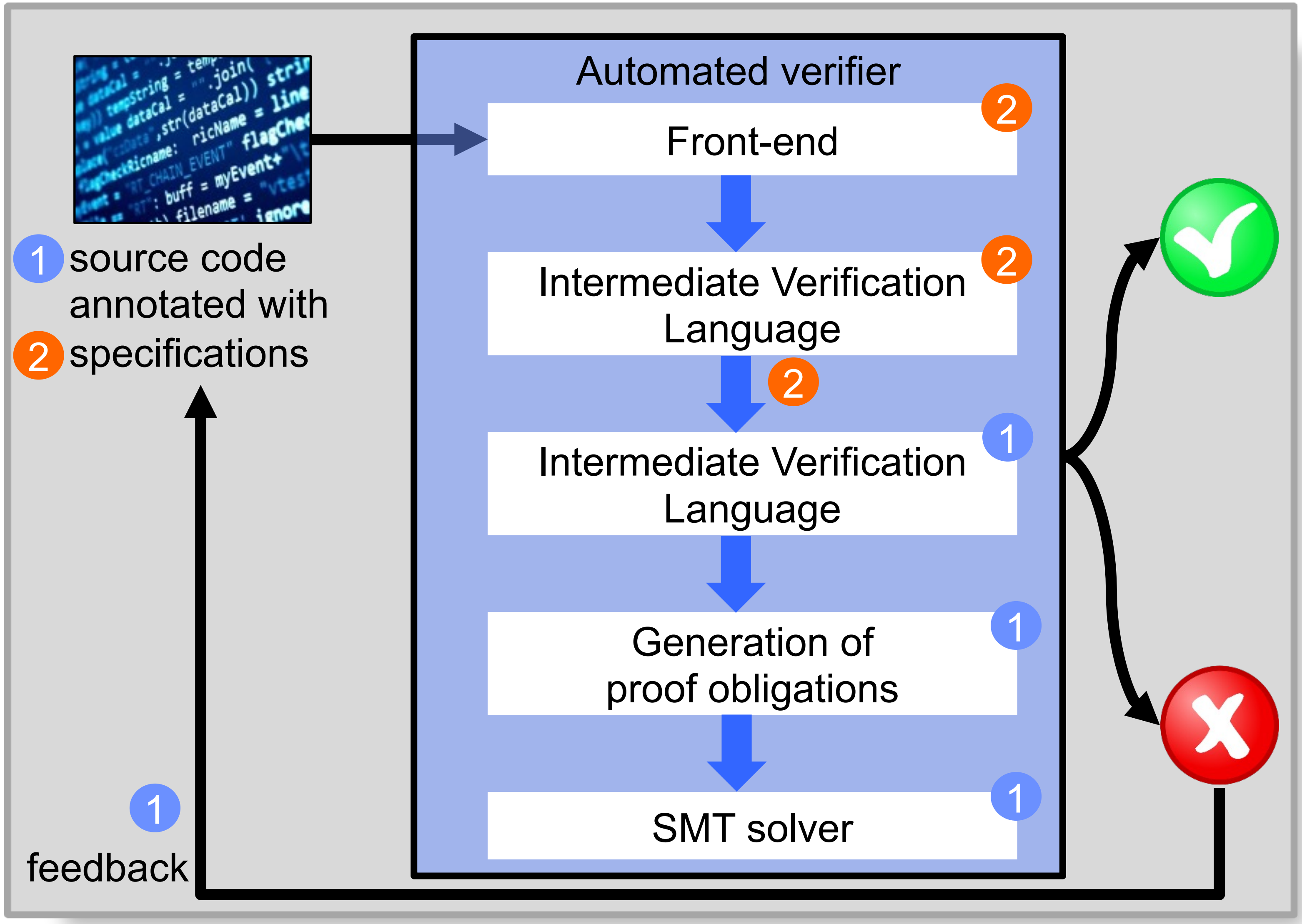
What does this tell us? How do we get this?

Architecture of automated program verifiers



- Automated verifiers are often implemented as a tool stack
- Stepwise compilation of programs into logical formulas (and back for error reporting)
- Each transformation deals with one verification problem
- Requirements:
  - reasoning principles
  - verification methodologies
  - engineering practices

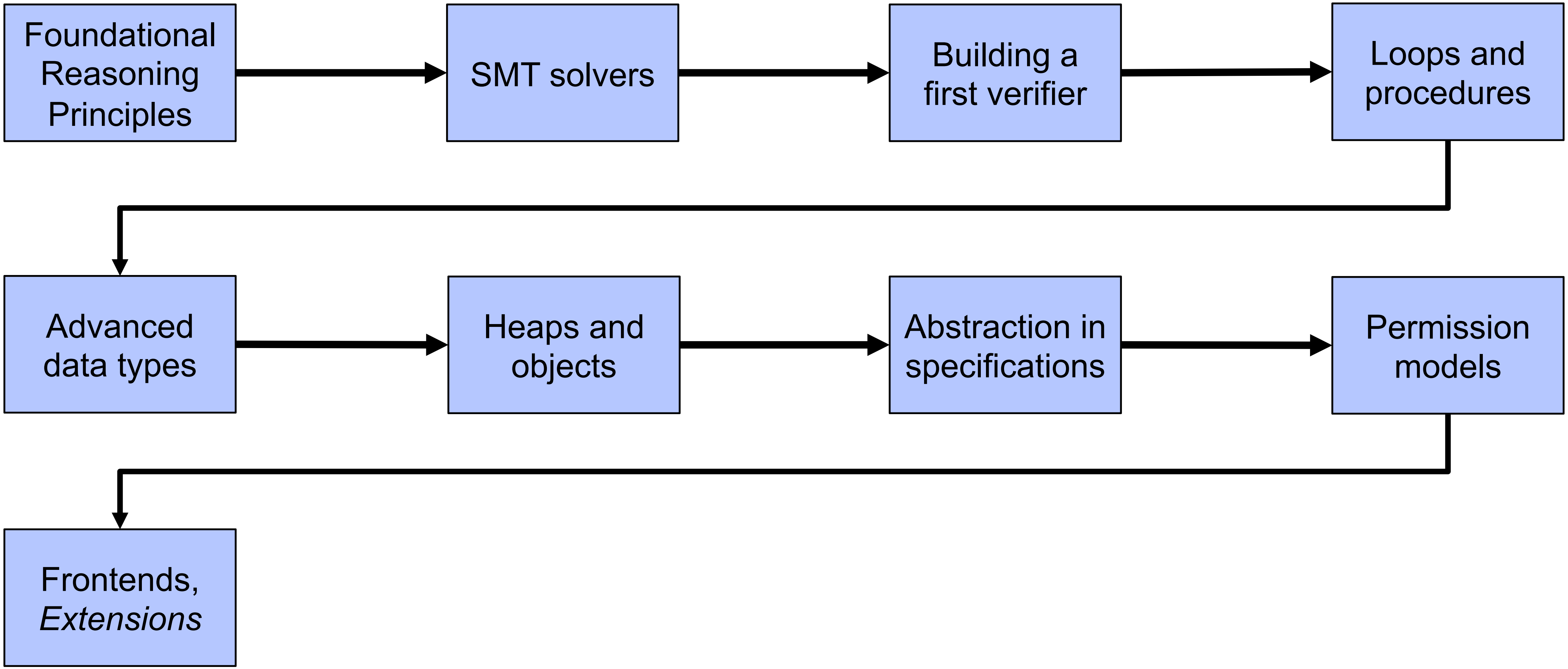
Roadmap



1. We learn how to build and use a verification tool for a small programming language
   - Core reasoning principles
   - Generation of proof obligations
   - Working with SMT solvers
   - Error reporting

2. We extend the language by advanced features
   - Verification challenges
   - Advanced reasoning and specification principles
   - Automation via encoding to lower levels

Tentative course outline



Outline

1. Why Program Verification?
2. Course Overview
3. Course Organization
4. Getting Started

Infrastructure

- Website: http://courses.compute.dtu.dk/02245
  - Course material (slides + webpage) is self-contained; reading references is optional
  - Material will be available at least one day before each lecture

- 7.5 ETCS course ➔ involves homework

- Classes
  - Lectures: Thursday 13:00 – 17:00, room B321-H033
  - Question time (for help with material, homework, etc.)
    - Physical: Monday 13:00 – 14:00, room B321-017
    - Online: Tuesday 18:00 – 19:00, MS Teams

Lectures are meant to be interactive (red slides and boxes)

- Many in-class exercises involve verification tools
  - Make sure to have them at hand when coming to class
  - Typically 5 – 30 min for each exercise
  - Teamwork is encouraged

- Discuss exercise solutions

- Feel free to ask questions at any time

- Feedback is highly appreciated
  - This is new material, your feedback will improve it ☺

> Think about questions in these boxes before the lecture

Examination

- Completeness and quality of group projects (size: 2-3)
  - 15% Homework: preparation for projects
    - Weekly deadline until project release
    - Solutions will be marked and discussed in class
  - 40% Project A: build a verification tool from scratch
  - 60% Project B: design a new verification methodology
  - Yes, the total is 115% ☺
  - Project deadline: November 27, 23:59
  - No reports but submissions must be well-documented and justified

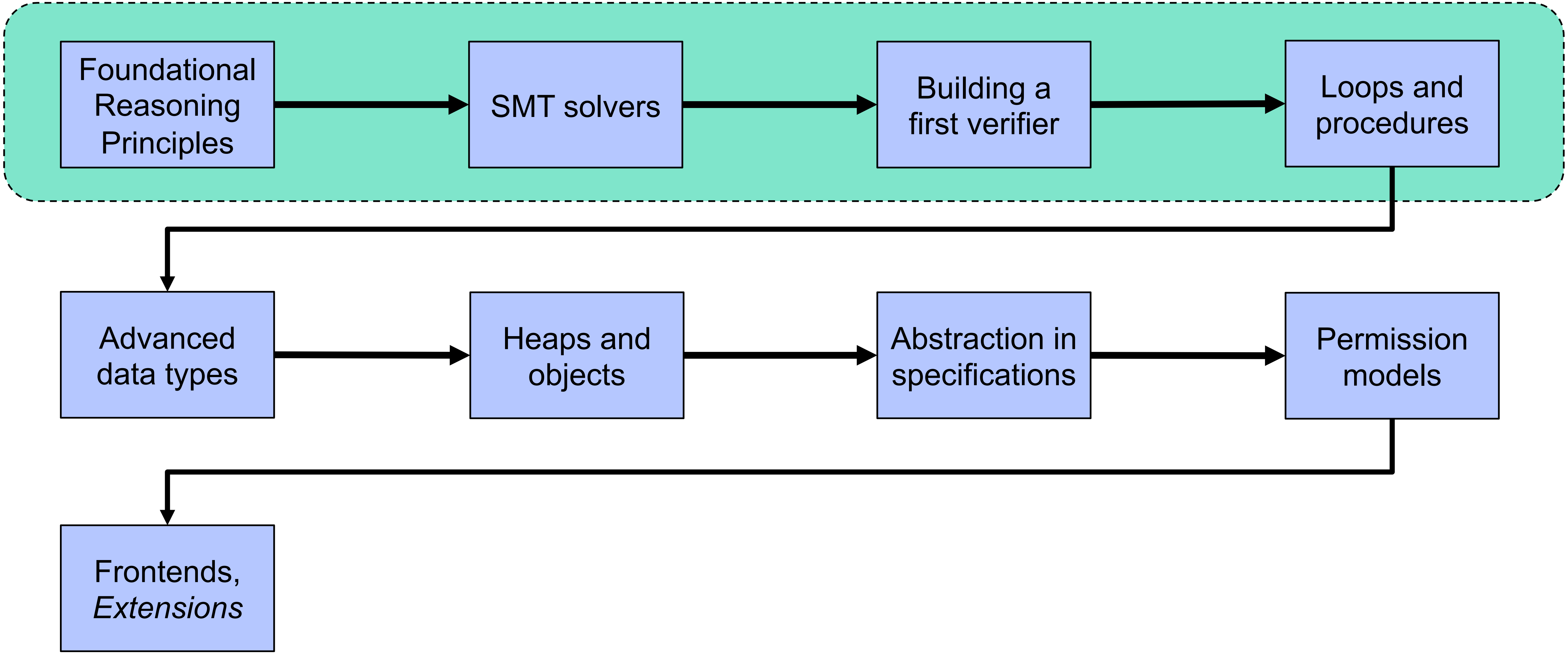
- Individual oral exam
  - Project presentation (ca. 7min, no slides needed)
  - Discussion of projects and course content (ca. 20 min)

Outline

1. Why Program Verification?
2. Course Overview
3. Course Organization
4. Getting Started

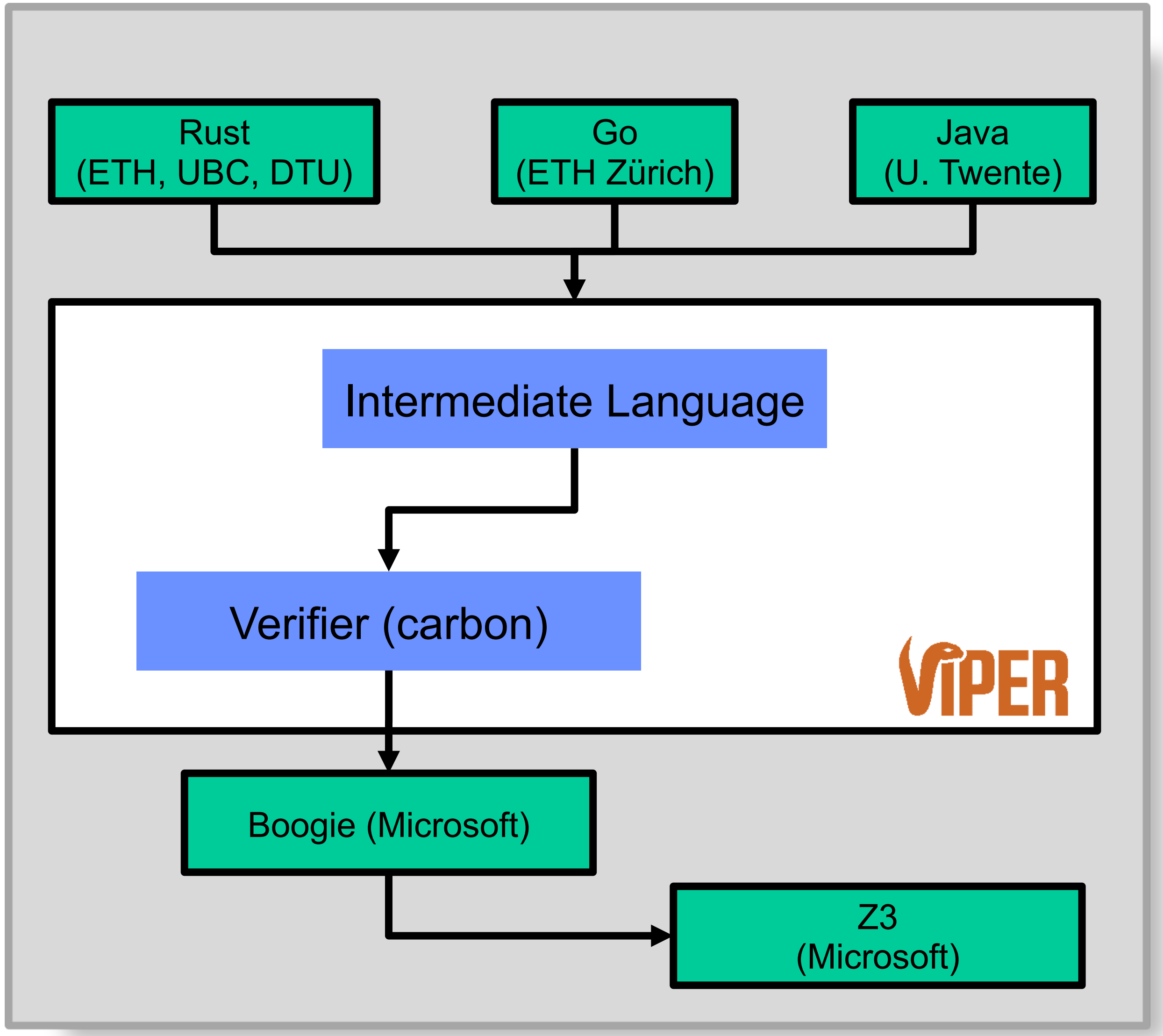
Tentative course outline

But first: using a verifier



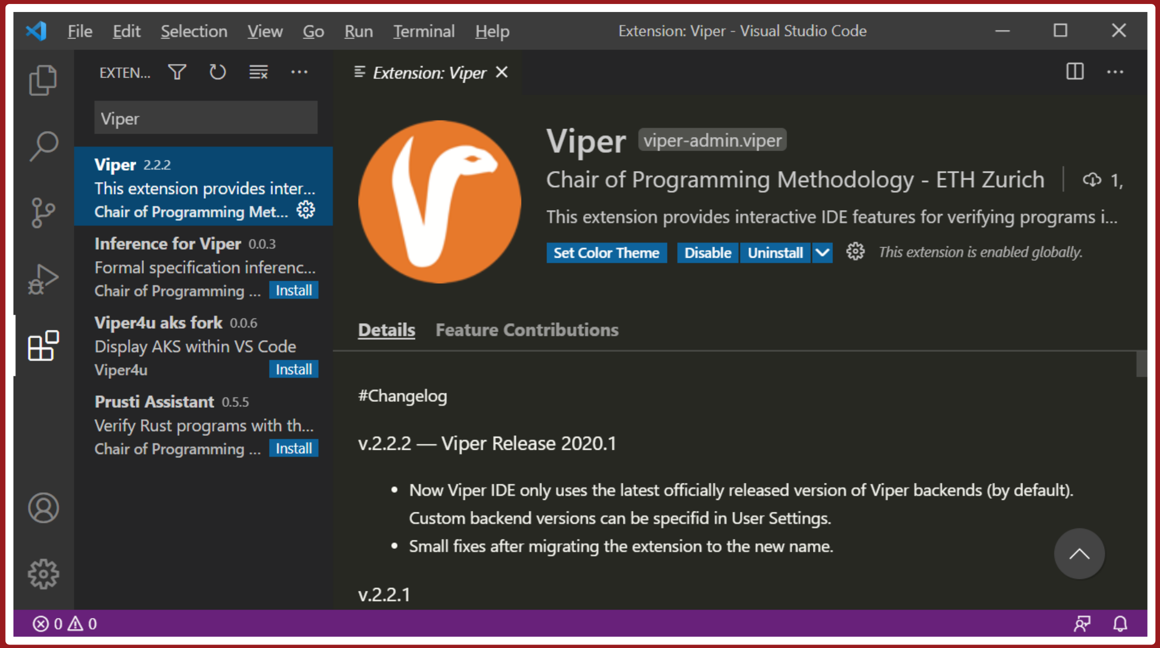
The Viper Verification Framework

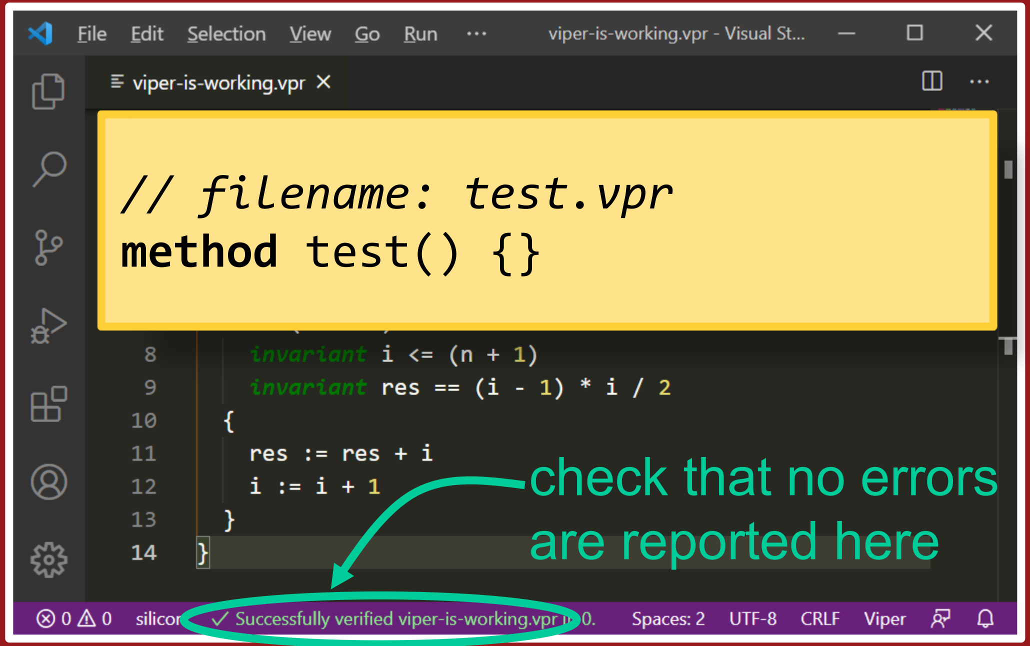
- Viper language
  - Models verification problems
  - Some statements are not executable
- Two verification backends
  - Carbon (close to what you will build)
  - Silicon
- **For now:** Programming language with a built-in verifier
- **Later:** Automate new methodologies



Installing Viper

- Install Java 11+ (64-bit)
  - set Java_HOME and PATH
- Install Visual Studio Code (64-bit)
- In Visual Studio Code:
  - Open the extensions browser (⇧+Ctrl+X or ⇧+⌘+X)
  - Search for *Viper*
  - Install the extension and restart
- Create and verify the file test.vpr (right)
- Switch to carbon and verify test.vpr again
  - click on silicon (bottom left) to switch





Viper methods

```
method triple(x: Int) returns (r: Int)
{
    r := 3 * x
}
```

Viper methods

all Viper statements are put in **methods**

read-only **input** parameters

```
method triple(x: Int, flag: Bool)
  returns (r: Int)
{
    if (flag) {
        r := 3 * x
    } else {
        var y: Int
        y := x + x
        r := x + y
    }
}
```

read-write **output** parameters

no explicit return statements

local variable declaration

Assertions

```
method triple(x: Int, flag: Bool)
  returns (r: Int)
{
    if (flag) {
        r := 3 * x
        assert r > 0
    } else {
        var y: Int
        y := x + x
        r := x + y
        assert r == 3 * x
    }
}
```

- **`assert`** `expr` tests if `expr` evaluates to `true`
  - Yes: no effect
  - No: runtime error
- Testing: no assertion error for *chosen* inputs
- Verification: no assertion error for *all* inputs

Which assertions hold?

Postconditions

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
{
  var y: Int
  y := x + x
  r := x + y
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

- Postconditions specify how returned outputs are related to inputs
  - Default: `true`

Postconditions

```
method triple(x: Int) returns (r: Int)
    ensures r == 3 * x
{
  var y: Int
  y := x + x
  r := x + y
}
```

check: r == 3 * x

```
method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

learn: z == 3 * 7

- Postconditions specify how returned outputs are related to inputs
  - Default: `true`
- *Checked* against implementation for all possible parameters
- *Guaranteed* to hold after method calls for supplied parameters

Alternative Implementation

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
{
     r := x / 2
     r := 6 * r
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

x = 7

x = 3

x = 18

- Some implementations do not work for arbitrary inputs
- A **precondition** filters out undesirable inputs

Preconditions

```
method triple(x: Int) returns (r: Int)
   requires x % 2 == 0
   ensures r == 3 * x
{
  r := x / 2
  r := 6 * r
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

- Preconditions specify on what inputs a method can be called
  - Default: `true`

Preconditions

```
method triple(x: Int) returns (r: Int)
   requires x % 2 == 0
   ensures r == 3 * x
{
  r := x / 2
  r := 6 * r
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

`r == 3 * x` for even `x`

`7 % 2 == 1`

- Preconditions specify on what inputs a method can be called
  - Default: `true`
- *Guaranteed* at the beginning of method implementation
- *Checked* before method calls for supplied parameters

Exercise

Write at least two Viper implementations for the method below that verify.

Try to find one that does *not* compute the maximum.

```
method max(x: Int, y: Int) returns (r: Int)
    ensures r >= x
    ensures r >= y // conjunction of postconditions
{
    // TODO
}
```

Contracts

A method **contract** consist of the method's
- name,
- input and output parameters, and
- pre- and postconditions.

Contracts must be upheld by method calls and implementations.

```
method triple(x: Int) returns (r: Int)
   requires x % 2 == 0
   ensures r == 3 * x
{
  // implementation
  r := x / 2
  r := 6 * r
}
```

✔

```
method client()
{
  triple(7)
  // violates precond.
}
```

✘

Underspecification

```
method triple(x: Int) returns (r: Int)
    requires x > 3
    ensures r > x
{
    r := 3 * x
}
```

- Implementation details are often irrelevant
- Contracts may
  - require more than an implementation needs
  - ensure less than an implementation gives

Give another contract implementation.

Verifying Method Calls

```
method triple(x: Int) returns (r: Int)
   requires x > 0
   ensures r > x
{
   r := 3 * x
}

method client() {
  var z: Int
  z := triple(7)
  assert z > 5
  assert z == 21
}
```

What is happening here?

Abstract Methods

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
```

```
method isqrt(x: Int) returns (r: Int)
    requires x >= 0
    ensures x >= r * r
    ensures x < (r+1) * (r+1)
```

```
method foo(a: Int) returns (b: Int)
    requires a > 0
    ensures b > a
{
    b := isqrt(a)
    b := triple(a)
}
```

- Contracts without Implementations
  - abstract from hard-to-verify code
  - abstract from unknown implementation
- Verification and good software engineering facilitate each other
  - *Incremental development* by refinement
  - Contracts become simpler if every method has a *single responsibility*
  - Avoid premature optimizations

Exercise

Consider the method `maxSum` with the following signature:

```
method maxSum(x: Int, y: Int) returns (sum: Int, max: Int)
```

`maxSum` is supposed to store the sum of x and y in variable `sum` and the maximum of x and y in variable `max`, respectively.

a) Define a reasonable contract for `maxSum`.

b) Implement a method that calls `maxSum` on 1723 and 42. Test your contract by adding assertions after the call. Improve your contract if any assertion fails.

c) Implement `maxSum`.

Now, consider a method `reconstructMaxSum` that tries to determine the values of `maxSum`'s input parameters from the output parameters, i.e. it reconstructs x and y from sum and max.

d) Write an abstract method with a postcondition specifying the behaviour of `reconstructMaxSum`.

e) Can you give an implementation of `reconstructMaxSum`? If not, can you implement it after adding a precondition?

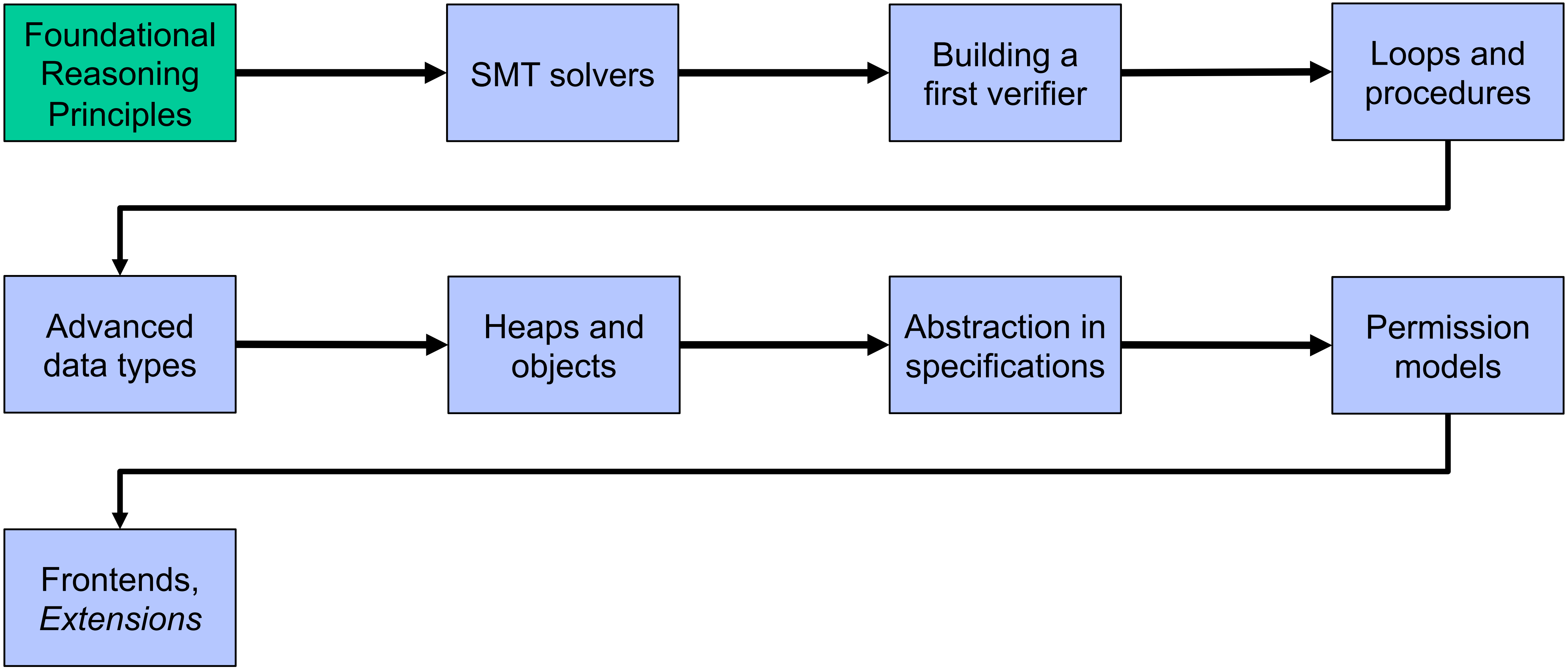
f) Write a client to test your implementation of `reconstructMaxSum`.

More abstract methods

```
method unsound(x: Int)
  returns (r: Int)
  ensures r != r


method test() {
  var a: Int
  a := unsound(17)
  assert 2 != 2
}
```

Wrap-up: Informal Overview

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
  }
}
```

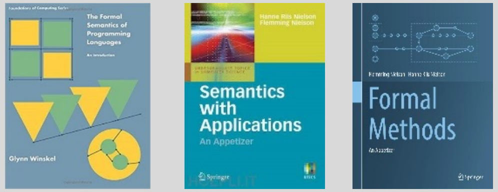
spec: `abs(x)` returns `|x|`

Does every execution comply with **spec**?

Verification must be rooted in rigorous mathematics

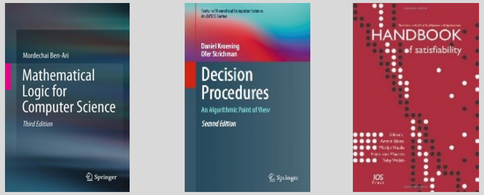
Given a **program**

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**

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

Program **semantics** describes how *states* evolve during program execution

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

initial state

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

$\Longrightarrow$ (var y;)

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

$\Longrightarrow$ (y := x + 7;)

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

$\Longrightarrow$ (r := x + y)

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

final state

Making it formal

Predicates capture properties of program states

$$\textbf{Pred} \quad = \quad \{\, P \mid P\colon \textbf{States} \;\rightarrow\; \texttt{Bool} \,\}$$

Logical characterization

```
x != 0
```

Set characterization

$$P = \{\sigma \in \textbf{States} \mid \sigma(\mathsf{x}) \neq 0\}$$

Making it formal

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

{ *Pre* } S { *Post* }

acceptable initial states → *Pre*

acceptable final states → *Post*

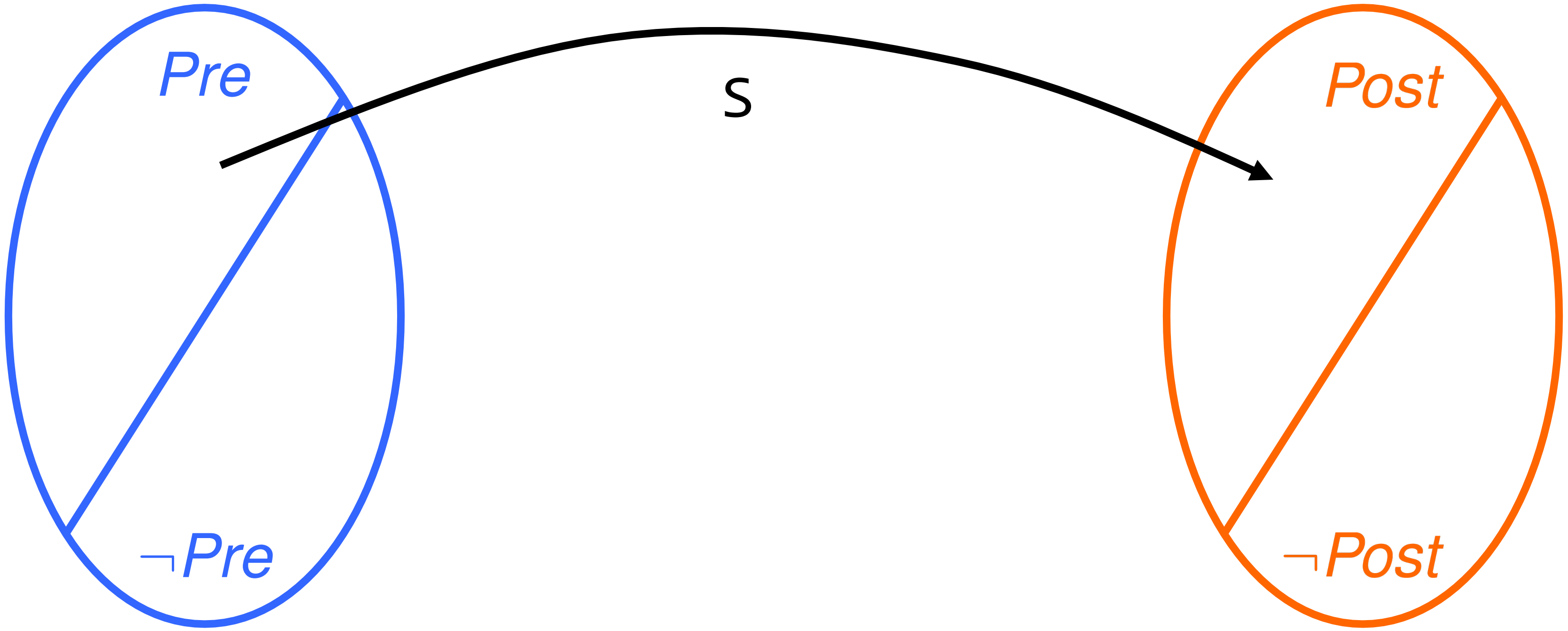
```
method foo(x: Int)
  returns (r: Int)
  requires x > 0
  ensures r > y
{
  var y: Int
  y := x + 7
  r := x + y
}
```

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

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

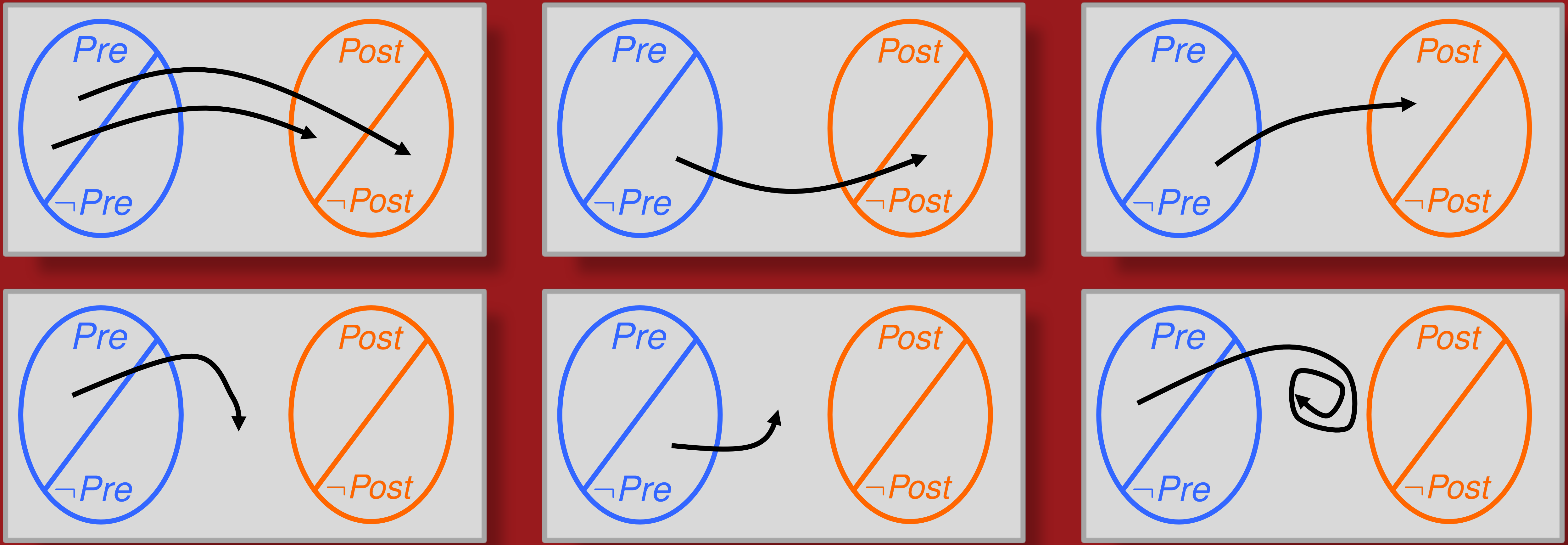
Making it formal

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`.



Which triples are valid?

```
{ x == 1 }
y := 2 * x + 1
{ y < 42 }
```

```
{ x == 1 }
y := 2 * x + 1
{ y >= 3 }
```

```
{ x == 1 }
y := 2 * x + 1
{ y > 0 }
```

```
{ x == 1 }
y := 2 * x + 1
{ true }
```

```
{ x == 1 }
y := 2 * x + 1
{ y == 3 && x == 1 }
```

```
{ x == 1 }
y := 2 * x + 1
{ y <= 17 }
```

```
{ x < 3 }
y := 2 * x + 1
{ y <= 17 }
```

```
{ false }
y := 2 * x + 1
{ y <= 17 }
```

```
{ x + x <= x }
y := 2 * x + 1
{ y <= 17 }
```

```
{ x == 5 || x == 7 }
y := 2 * x + 1
{ y <= 17 }
```

Outline

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

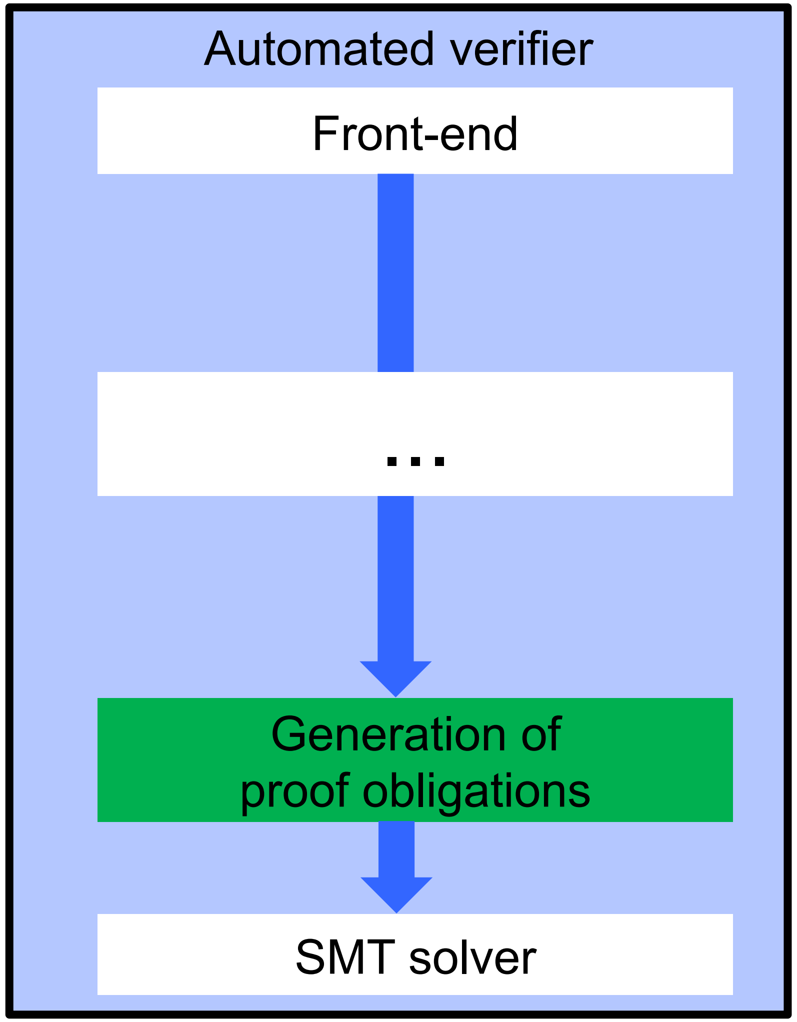
Reasoning about triples

- Argue *as rigorously as possible* that the Floyd-Hoare triple described by the following Viper method is valid.
- Hint: annotate the file *03-quintuple.vpr*

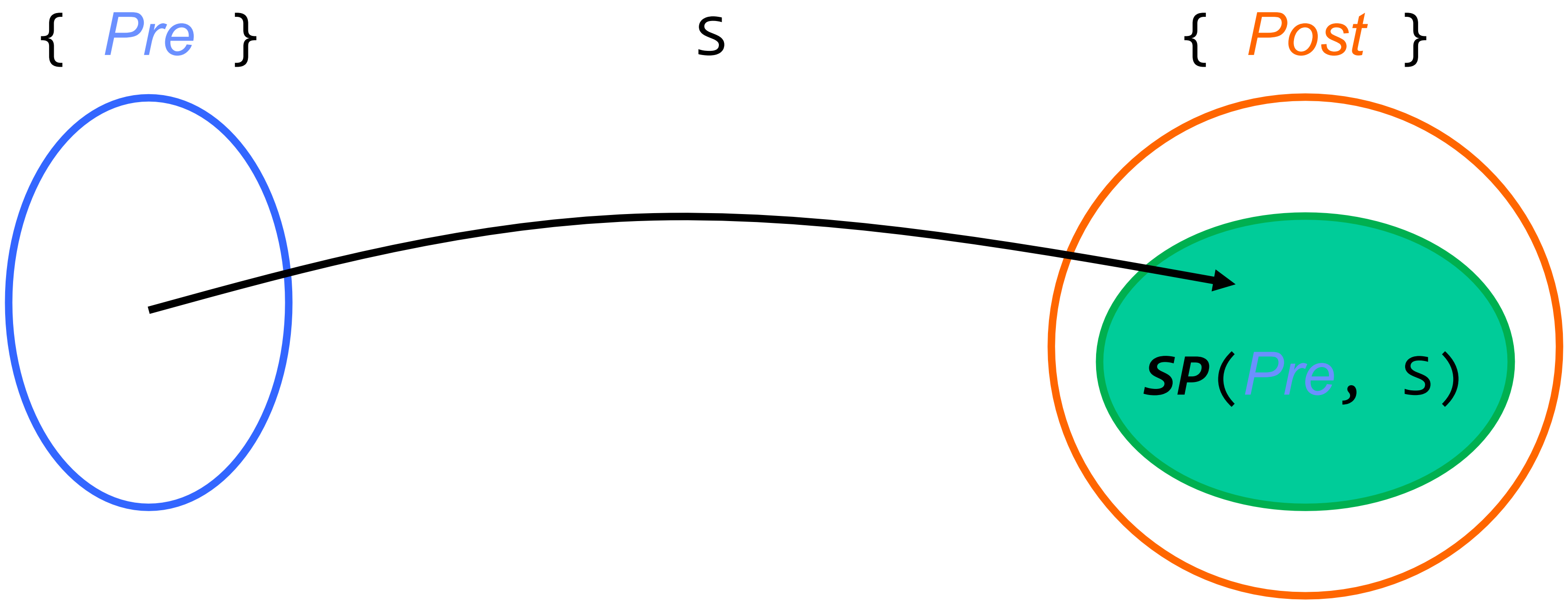
```
method quintuple(x: Int) returns (r: Int)
    requires x > 0
    ensures r > 4 * x
{
    var y: Int
    y := 2 * x
    var z: Int
    z := 3 * x
    r := y + z
}
```

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*?

y will have some value

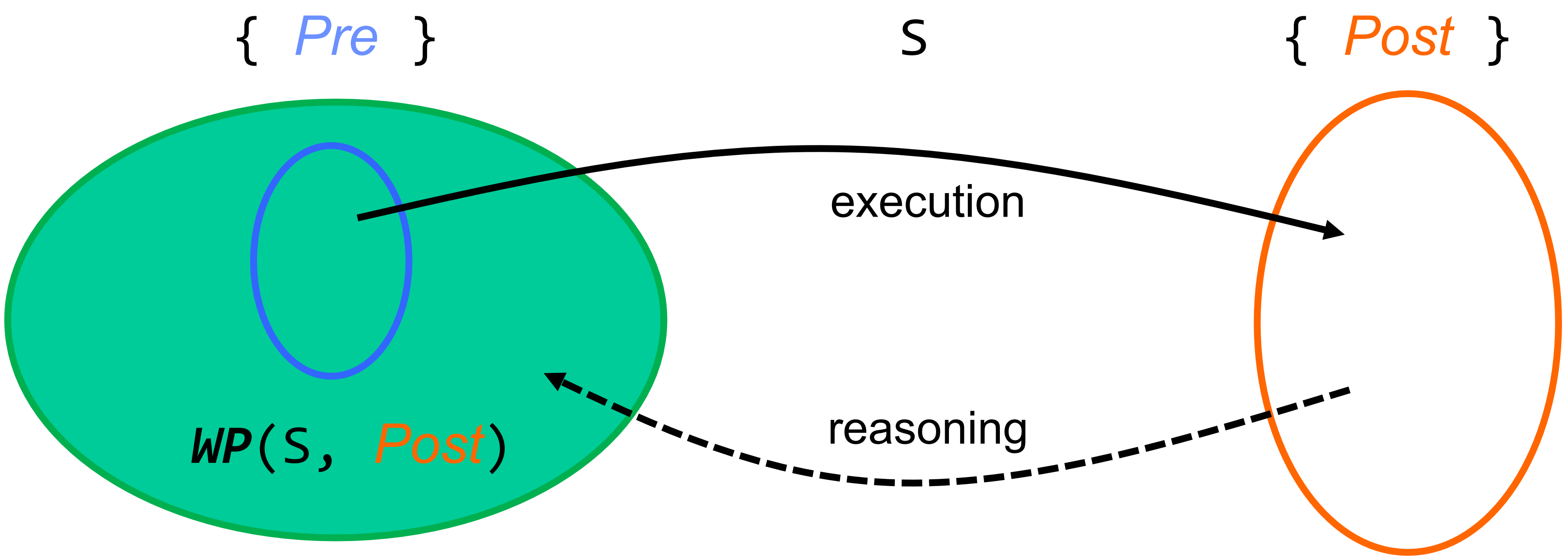
```
{ x > 0 }

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 }
```

y will be x + 7

y will be x + y

if r == x + y
and x > 0
then 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 }
```

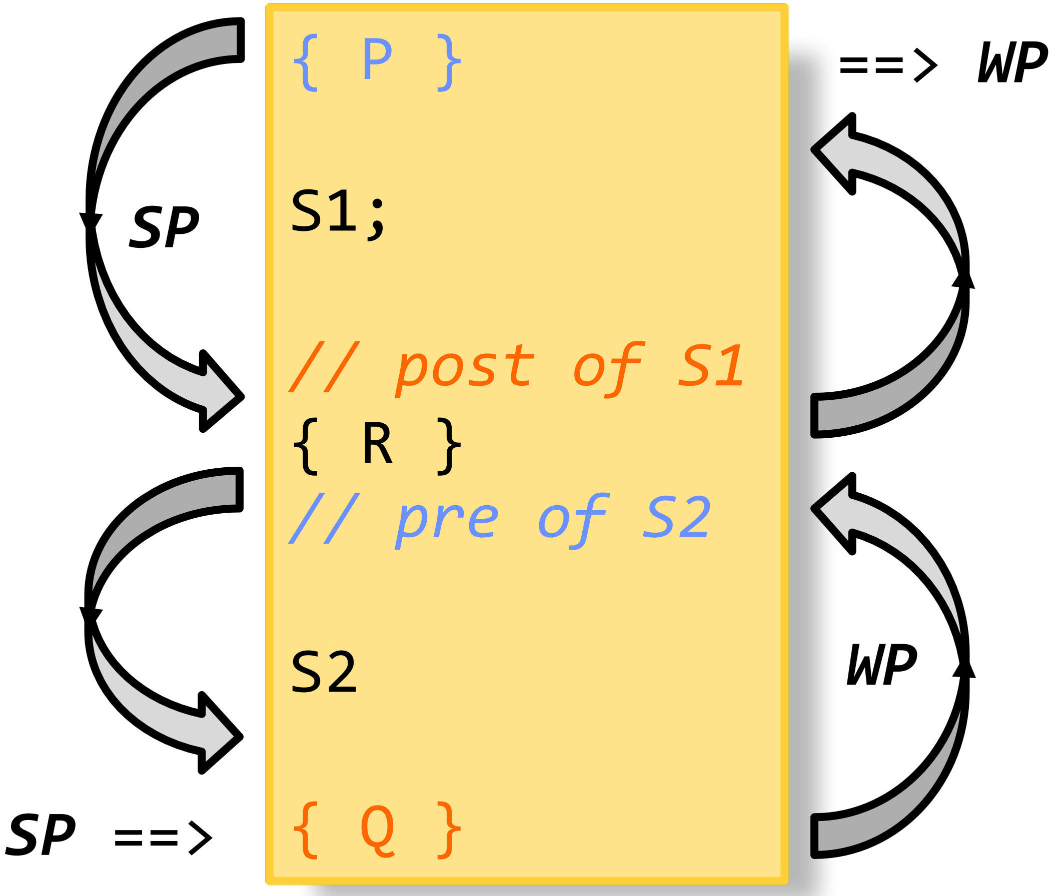
✗

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`



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

```
{ 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 }
```

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

```
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 }
```

- upper implies lower ({ x > 0 } → { 5 * x > 4 * x })
- upper implies lower ({ 5 * x > 4 * x } → { 2 * x + 3 * x > 4 * x })
- valid triple (y := 2 * x;)
- valid triple (z := 3 * x;)
- valid triple (r := y + z)

Whole program: valid triple ✓

Exercise

- What is wrong with the following proof?

```
{ true }
  x := 0;
{ x == 0 }
{ x == 0 && y == 6 }
  x := x + 2;
{ x == 2 && y == 6 }
{ x + y == 8 }
  y := x + y
{ y == 8 }
```

Exercise

Left half of room: use ***WP*** to check which triples are valid

Right half of room: use ***SP*** to check which triples are valid

```
{ 0 <= x }
  x := x + 1
{ -2 <= x }
  y := 0
{ -10 <= x }
```

```
{ 0 <= x }
  x := x + 1
{ true }
  y := 0
{ -10 <= x }
```

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

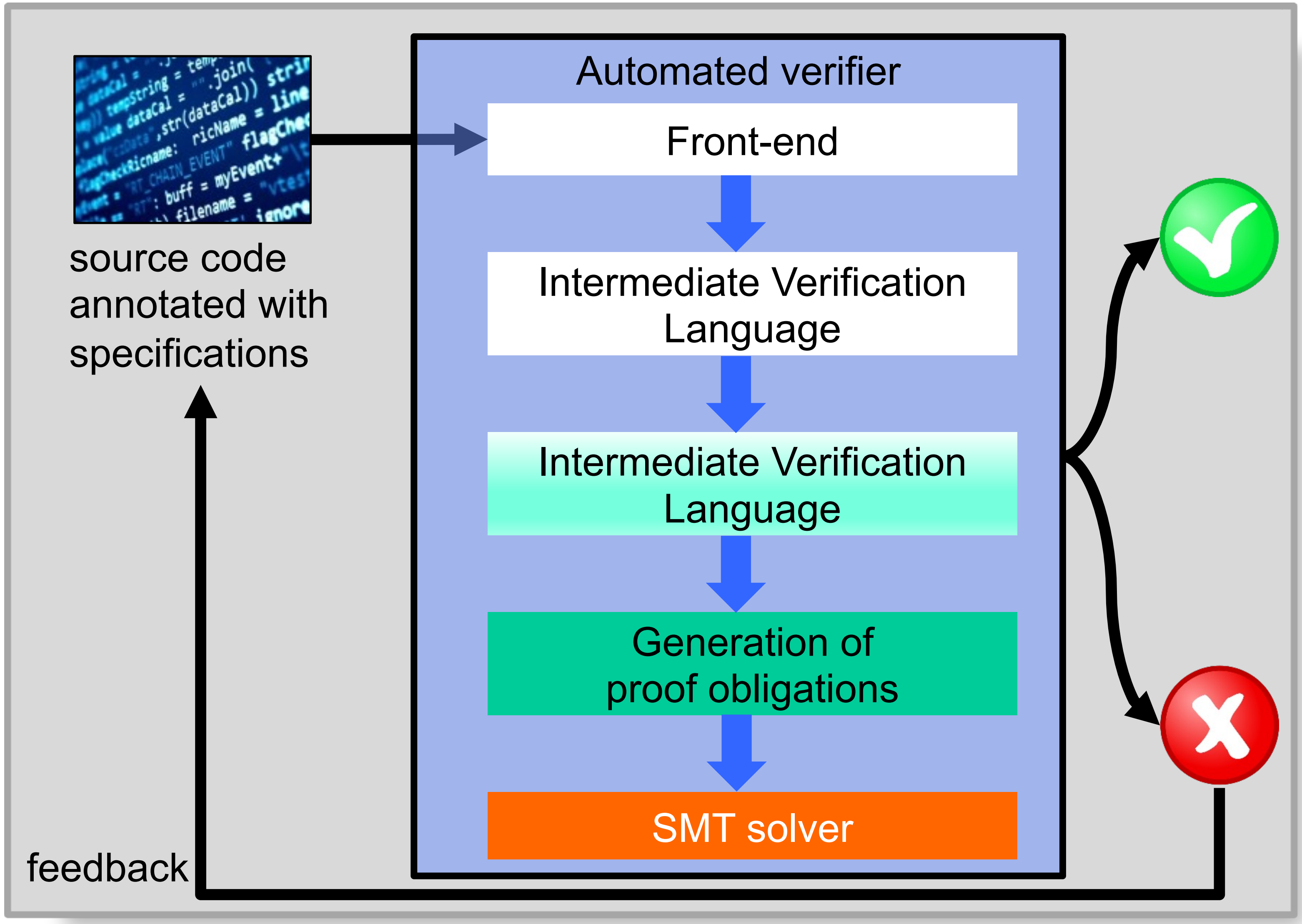
```
SP(P, x := a)  ::=  exists x0 :: P[x / x0] && x == a[x / x0]
WP(x := a, Q)  ::=  Q[x / a]
```

Outline

Strongest Post vs. Weakest Pre – Does it matter?

Wrap-up

Where are we?



- Viper language
- ***WP*** (our preference)
- ***SP*** (used later)
- ***next lecture***