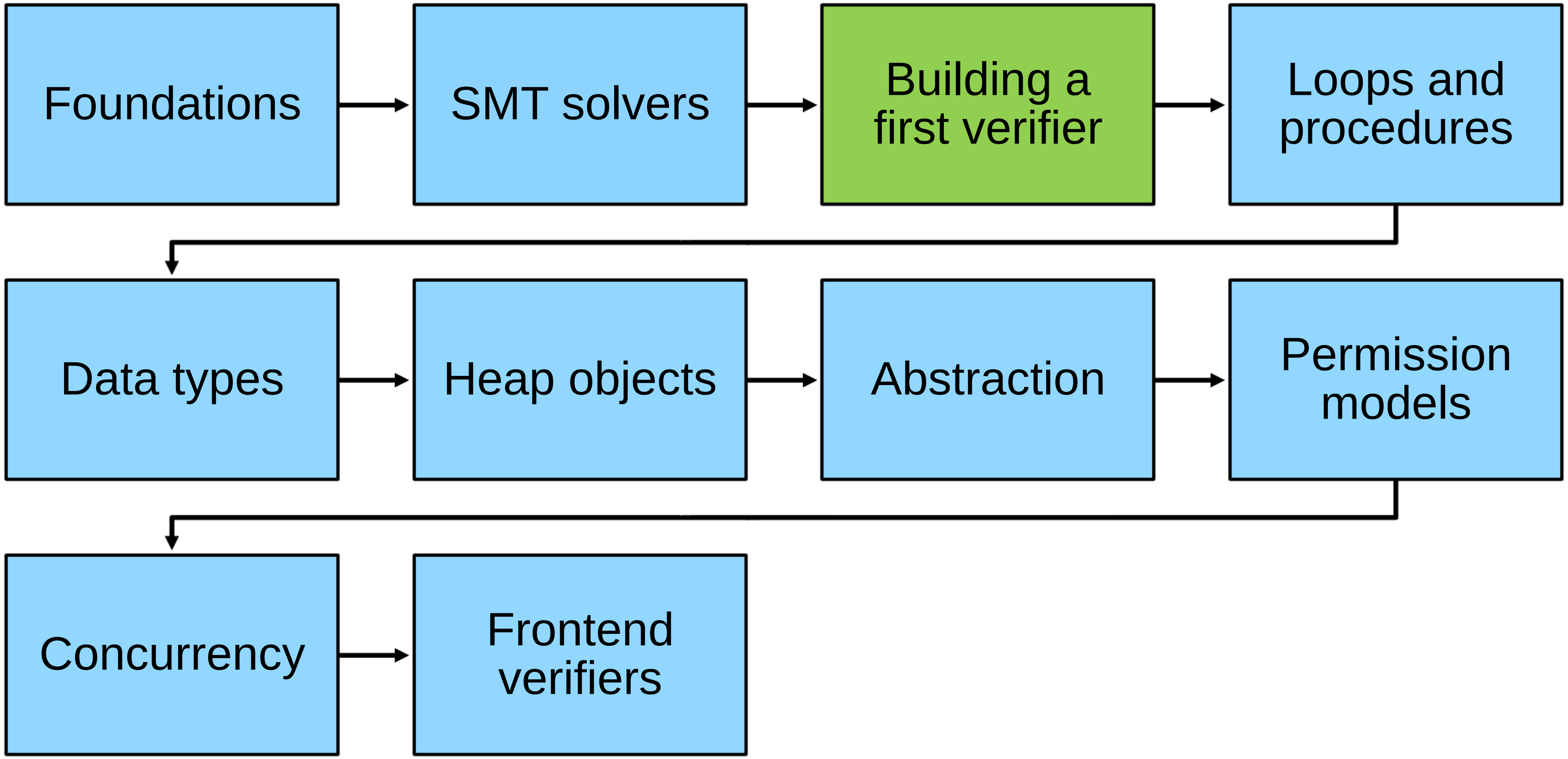
Peter Müller and Marco Eilers
(slides developed in cooperation with Christoph Matheja)

PROGRAM VERIFICATION

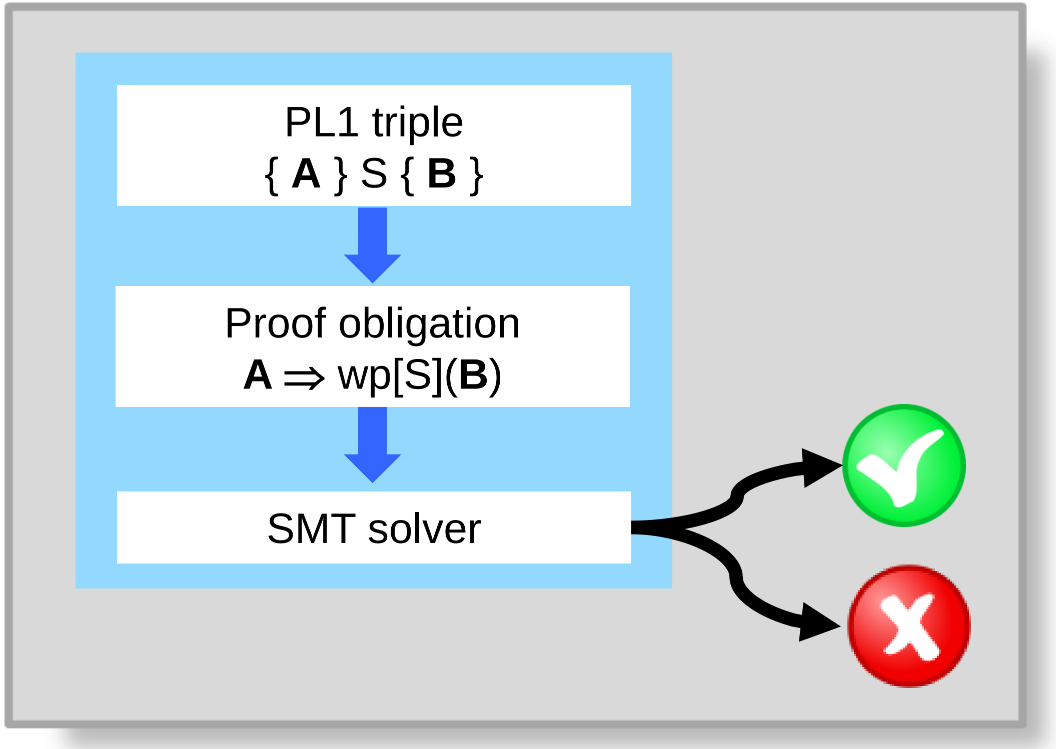
ETH zürich

Spring 2023

Outline


Foundations
SMT solvers
Building a first verifier
Loops and procedures
Data types
Heap objects
Abstraction
Permission models
Concurrency
Frontend verifiers

A naive first verifier



- Is PL1 sufficiently expressive?
  - Is it easy to encode interesting programs?
  - Is it possible to express interesting verification problems?

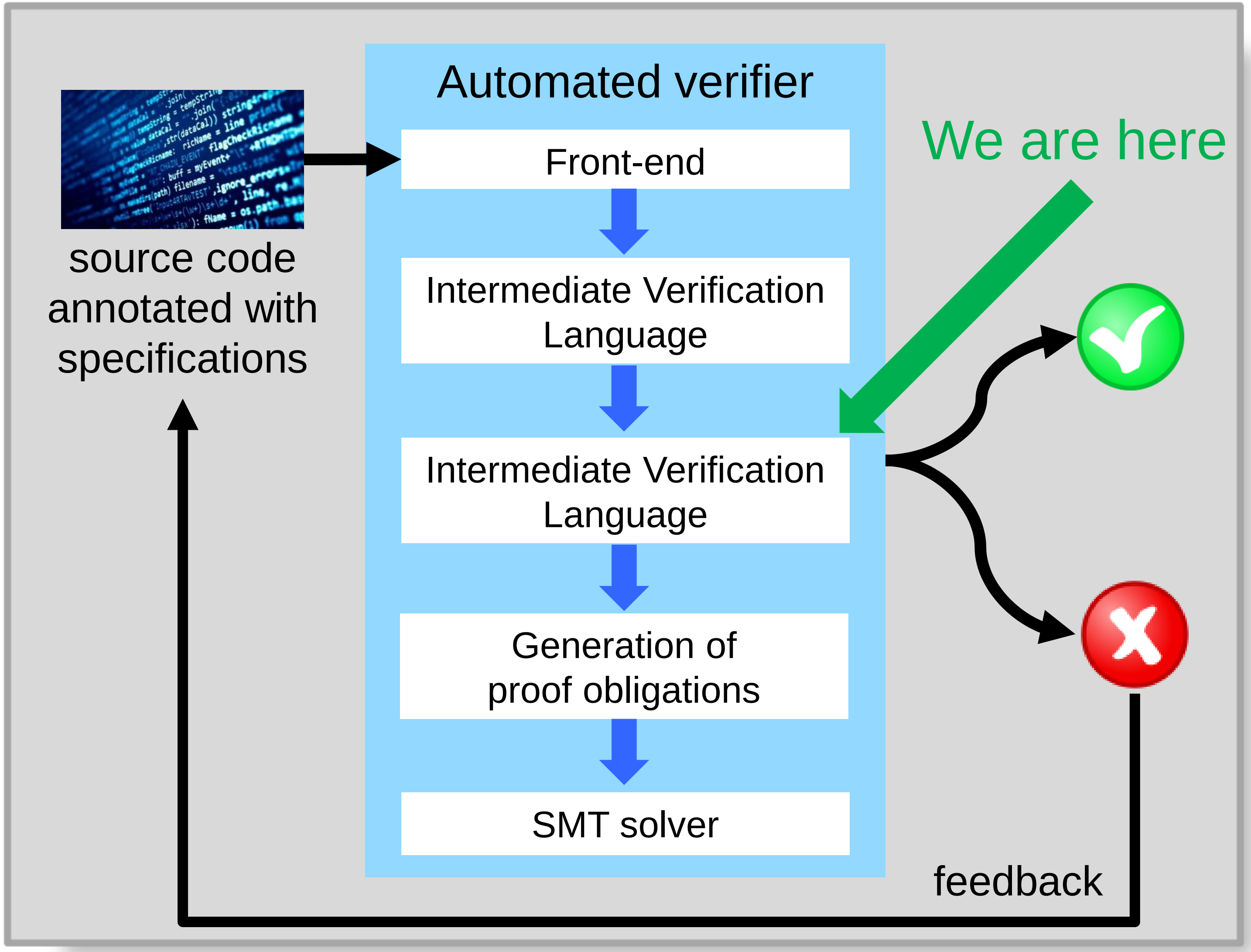
- How can we provide useful error messages?

Building a first verifier

1. Two intermediate verification languages

2. Error reporting

Roadmap



- PL1 allows us to compute proof obligations easily
- To be suitable as low-level IVL, it should also facilitate the encoding of more complex programming concepts
  - Loops
  - Procedures
  - Data structures
  - Concurrency

How expressive is PL1?

PL1 statements

S ::= **skip**
| v := E
| **abort** | **diverge**
| S ; S
| **if** (E) { S } **else** { S }
| **if** (*) { S } **else** { S }

Expressions

E ::= c | x | E + E | E * E | E − E
| E < E | E ∧ E | E ∨ E | ¬E | ...

Types

T ::= **Bool** | **Int** | **Rational** | **Real**

- Let's encode some additional statements
  - **assert** E
  - **assume** E

New statement: **assert** E

- Assertions make existing knowledge explicit in the program
  - crash whenever it violates our knowledge
  - otherwise, do nothing

```
if (x > 0) {
  x := -x
} else {
  skip
}
assert x >= 0
```

✗

```
if (x < 0) {
  x := -x
} else {
  skip
}
assert x >= 0
```

✓

Operational semantics

$$\frac{[\![E]\!](\sigma) = \mathtt{true}}{\langle\, \mathtt{assert}\ E, \sigma \,\rangle \rightarrow \langle\, \mathbf{term}, \sigma \,\rangle}$$

$$\frac{[\![E]\!](\sigma) = \mathtt{false}}{\langle\, \mathtt{assert}\ E, \sigma \,\rangle \rightarrow \langle\, \mathbf{err}, \sigma \,\rangle}$$

What is the weakest precondition of **assert** E?

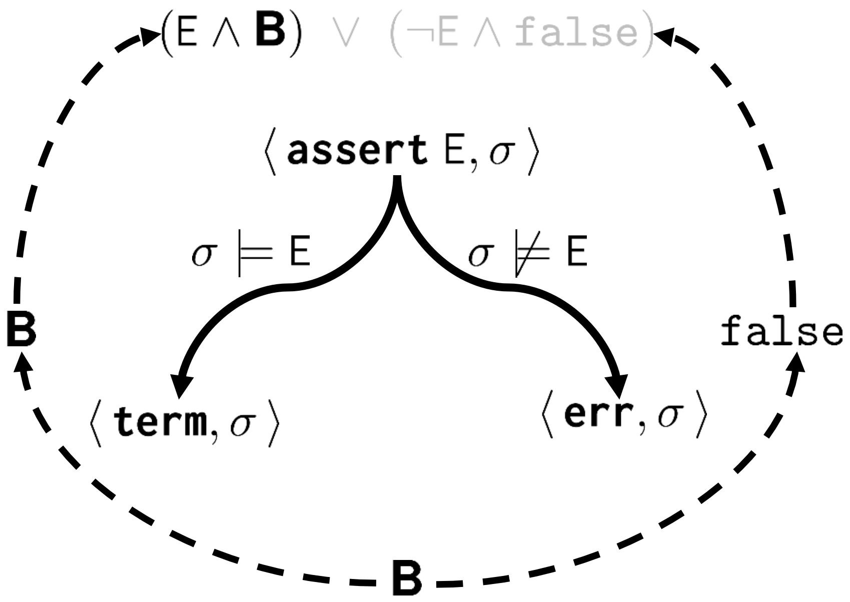
How can we encode **assert** E in PL1?

Encoding assert statements in PL1

Operational semantics

$$\frac{[\![E]\!](\sigma) = \mathtt{true}}{\langle \mathbf{assert}\ E, \sigma \rangle \rightarrow \langle \mathbf{term}, \sigma \rangle}$$

$$\frac{[\![E]\!](\sigma) = \mathtt{false}}{\langle \mathbf{assert}\ E, \sigma \rangle \rightarrow \langle \mathbf{err}, \sigma \rangle}$$



$$[\![\mathbf{assert}\ E]\!] = \mathbf{if}\ (E)\ \{\ \mathbf{skip}\ \}\ \mathbf{else}\ \{\ \mathbf{abort}\ \}$$

"is encoded as"

$$\begin{aligned} & wp[\![\mathbf{assert}\ E]\!](\mathbf{B}) \\ = \ & E \wedge \mathbf{B} \\ = \ & (E \wedge \mathbf{B}) \vee (\neg E \wedge \mathtt{false}) \\ = \ & (E \wedge wp[\![\mathbf{skip}]\!](\mathbf{B})) \vee (\neg E \wedge wp[\![\mathbf{abort}]\!](\mathbf{B})) \\ = \ & wp[\![\mathbf{if}\ (E)\ \{\ \mathbf{skip}\ \}\ \mathbf{else}\ \{\ \mathbf{abort}\ \}]\!](\mathbf{B}) \end{aligned}$$

Encoding assert statements in PL1: discussion

$$[\![\textbf{assert}\ \mathsf{E}]\!] = \textbf{if}\ (\mathsf{E})\ \{\ \textbf{skip}\ \}\ \textbf{else}\ \{\ \textbf{abort}\ \}$$

- E is an expression of PL1
  - It is often useful to have a more expressive language to express assertions
  - For instance, quantifiers are useful to express properties of arrays (e.g., sortedness)
- The problem cannot be solved (easily) by extending the expression syntax
  - Expressions must be efficiently executable, which is not always the case for quantifiers
  - Procedure calls in expressions cannot be encoded easily into an SMT formula

New statement: **`assume`** E

- Assumptions add unverified knowledge
  - properties of the execution environment
  - (e.g., about results of system calls)
  - properties that are justified elsewhere
    (e.g., a mathematical fact or properties guaranteed by a type system)

```
// Fermat's last theorem
assume 0 < x && 0 < y && 0 < z ==>
       x*x*x + y*y*y != z*z*z
```

Operational semantics

$$\frac{[\![\mathsf{E}]\!](\sigma) = \mathtt{true}}{\langle \mathtt{assume}\ \mathsf{E}, \sigma \rangle \rightarrow \langle \mathtt{term}, \sigma \rangle}$$

$$\frac{[\![\mathsf{E}]\!](\sigma) = \mathtt{false}}{\langle \mathtt{assume}\ \mathsf{E}, \sigma \rangle \rightarrow \langle \mathtt{diverge}, \sigma \rangle}$$

- If E holds, **`assume`** `E` is equivalent to **`skip`**
- Otherwise, magic happens

What is the weakest precondition of **`assume`** `E`?

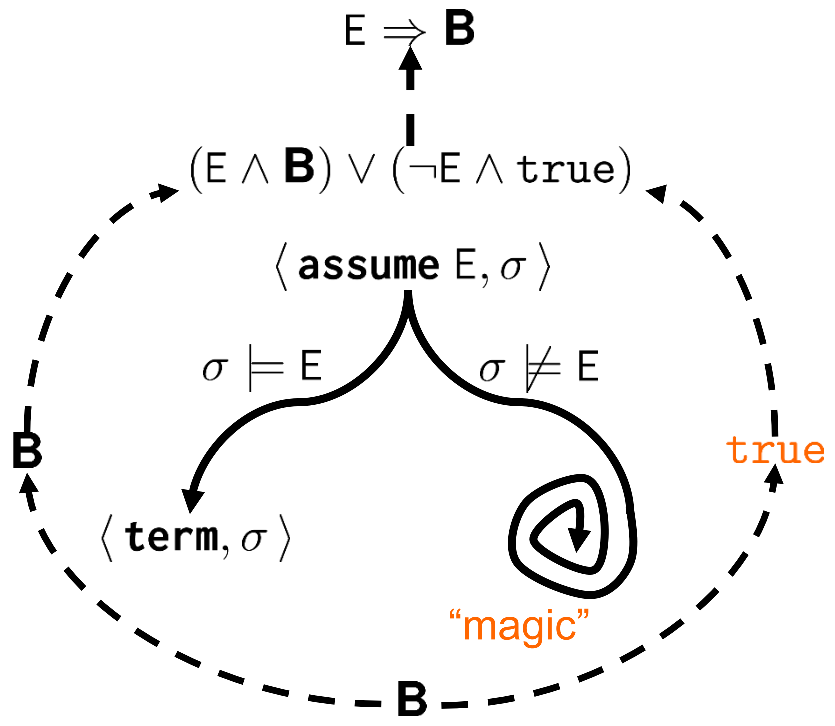
How can we encode **`assume`** `E` in PL1?

Encoding assume statements in PL1

Operational semantics

$$\frac{[\![\mathsf{E}]\!](\sigma) = \mathtt{true}}{\langle\, \mathbf{assume}\ \mathsf{E}, \sigma \,\rangle \ \rightarrow\ \langle\, \mathbf{term}, \sigma \,\rangle}$$

$$\frac{[\![\mathsf{E}]\!](\sigma) = \mathtt{false}}{\langle\, \mathbf{assume}\ \mathsf{E}, \sigma \,\rangle \ \rightarrow\ \langle\, \mathbf{diverge}, \sigma \,\rangle}$$



$$[\![\mathbf{assume}\ \mathsf{E}]\!] \ = \ \mathbf{if}\ (\ \mathsf{E}\ )\ \{\ \mathbf{skip}\ \}\ \mathbf{else}\ \{\ \mathbf{diverge}\ \}$$

$$\begin{aligned} & wp[\![\mathbf{assume}\ \mathsf{E}]\!](\mathbf{B}) \\ = \ & \mathsf{E} \Rightarrow \mathbf{B} \\ = \ & (\mathsf{E} \wedge \mathbf{B}) \vee (\neg \mathsf{E} \wedge \mathtt{true}) \\ = \ & (\mathsf{E} \wedge wp[\![\mathbf{skip}]\!](\mathbf{B})) \vee (\neg \mathsf{E} \wedge wp[\![\mathbf{diverge}]\!](\mathbf{B})) \\ = \ & wp[\![\mathbf{if}\ (\ \mathsf{E}\ )\ \{\ \mathbf{skip}\ \}\ \mathbf{else}\ \{\ \mathbf{diverge}\ \}]\!](\mathbf{B}) \end{aligned}$$

Encoding assume statements in PL1: discussion

$$[\![\,\texttt{assume}\ \mathsf{E}\,]\!] \;=\; \texttt{if}\,(\mathsf{E})\,\{\,\texttt{skip}\,\}\ \texttt{else}\,\{\,\texttt{diverge}\,\}$$

- Like for assert statements, it would be useful to have a richer language than PL1 expressions

```
// Fermat's last theorem
assume forall x, y, z ::
       0<x && 0<y && 0<z ==>
       x*x*x + y*y*y != z*z*z
```

- Encoding works only as long as we focus on partial correctness
  - Encoding behaves like `assert E` for total correctness

- Generally, assume statements have to be used with great care to avoid introducing invalid assumptions

```
x := 0; y := 0; z := 0
assume x*x*x + y*y*y != z*z*z
assert false
```

Assertions

- To increase expressiveness without sacrificing efficient executability, we distinguish between (executable) expressions and (non-executable) assertions

Expressions
$E \;::=\; c \mid x \mid E+E \mid E*E \mid E-E \mid E<E \mid E \wedge E \mid E \vee E \mid \neg E \mid \ldots$

Like before

Assertions
$\mathbf{A} \;::=\; E \mid \forall x: T :: \mathbf{A} \mid \exists x: T :: \mathbf{A} \mid \mathbf{A} \Rightarrow \mathbf{A} \mid \ldots$

FO logic over suitable theories

- Expressions are used in all standard statements (assignments, if, while, etc.)

- Assertions are used as pre- and postconditions, and in assert and assume statements
  - As a consequence, *wp* yields an assertion

Toward a better IVL

- Since assert and assume statements are very common in verification problems, we support them natively
  - No encoding required
  - Works for partial and total correctness

$$
\begin{aligned}
S \quad ::= \quad & x := E \\
| \; & \textbf{assert } A \\
| \; & \textbf{assume } A \\
| \; & S ; S \\
| \; & \textbf{if } (*) \, \{ S \} \textbf{ else } \{ S \}
\end{aligned}
$$

- This allows us to remove other statements that can now be encoded easily

$$[\![\textbf{skip}]\!] \; = \; \textbf{assert } \texttt{true}$$

$$[\![\textbf{abort}]\!] \; = \; \textbf{assert } \texttt{false}$$

$$[\![\textbf{diverge}]\!] \; = \; \textbf{assume } \texttt{false}$$

$$[\![\textbf{if } (E) \, \{ S_1 \} \textbf{ else } \{ S_2 \}]\!] \; = \; \textbf{if } (*) \, \{ \textbf{assume } E; \; S_1 \} \textbf{ else } \{ \textbf{assume } \neg E; \; S_2 \}$$

Exercise: encoding of if-statements

$$[\![\,\mathtt{if}\ (\mathsf{E})\ \{\,\mathsf{S}_1\,\}\ \mathtt{else}\ \{\,\mathsf{S}_2\,\}\,]\!] \;=\; \mathtt{if}\ (*)\ \{\,\mathtt{assume}\ \mathsf{E};\ \mathsf{S}_1\,\}\ \mathtt{else}\ \{\,\mathtt{assume}\ \neg\mathsf{E};\ \mathsf{S}_2\,\}$$

Show that the encoding of if-statements preserves the weakest precondition:

$$wp[\![\,\mathtt{if}\ (\mathsf{E})\ \{\,\mathsf{S}_1\,\}\ \mathtt{else}\ \{\,\mathsf{S}_2\,\}\,]\!](\mathbf{A}) \;=\; wp[\![\,\mathtt{if}\ (*)\ \{\,\mathtt{assume}\ \mathsf{E};\ \mathsf{S}_1\,\}\ \mathtt{else}\ \{\,\mathtt{assume}\ \neg\mathsf{E};\ \mathsf{S}_2\,\}\,]\!](\mathbf{A})$$

PL2: a consolidated verification language

Types

$T ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Rational} \mid \texttt{Real}$

We assume that all variables, expressions, assertions, and programs are well-typed.

Expressions (executable)

$E ::= c \mid x \mid E + E \mid E * E \mid E - E \mid E < E \mid E \land E \mid E \lor E \mid \neg E \mid \ldots$

Assertions (FO logic over suitable theories)

$\mathbf{A} ::= E \mid \forall x : T :: \mathbf{A} \mid \exists x : T :: \mathbf{A} \mid \mathbf{A} \Rightarrow \mathbf{A} \mid \ldots$

PL2 statements

$$\begin{aligned} S ::= \; & x := E \\ \mid\; & \textbf{assert } \mathbf{A} \\ \mid\; & \textbf{assume } \mathbf{A} \\ \mid\; & S ; S \\ \mid\; & \textbf{if } (*)\ \{ S \}\ \textbf{else}\ \{ S \} \end{aligned}$$

Verification problem for PL2

- So far, we showed $\models \{\, \mathbf{A} \,\}\ \mathtt{S}\ \{\, \mathbf{B} \,\}$ by proving $\models \mathbf{A} \Rightarrow \mathit{wp}[\![\, \mathtt{S} \,]\!]\,(\mathbf{B})$

- In PL2, we can encode pre- and postconditions into the program

$$
\begin{aligned}
& \mathit{wp}[\![\, \mathtt{assume}\ \mathbf{A};\ \mathtt{S};\ \mathtt{assert}\ \mathbf{B} \,]\!]\,(\mathtt{true}) \\
=\ & \mathit{wp}[\![\, \mathtt{assume}\ \mathbf{A};\ \mathtt{S} \,]\!]\,(\mathit{wp}[\![\, \mathtt{assert}\ \mathbf{B} \,]\!]\,(\mathtt{true})) \\
=\ & \mathit{wp}[\![\, \mathtt{assume}\ \mathbf{A};\ \mathtt{S} \,]\!]\,(\mathbf{B}) \\
=\ & \mathit{wp}[\![\, \mathtt{assume}\ \mathbf{A} \,]\!]\,(\mathit{wp}[\![\, \mathtt{S} \,]\!]\,(\mathbf{B})) \\
=\ & \mathbf{A} \Rightarrow \mathit{wp}[\![\, \mathtt{S} \,]\!]\,(\mathbf{B})
\end{aligned}
$$

- Consequently, we do not have to consider pre- and postconditions explicitly

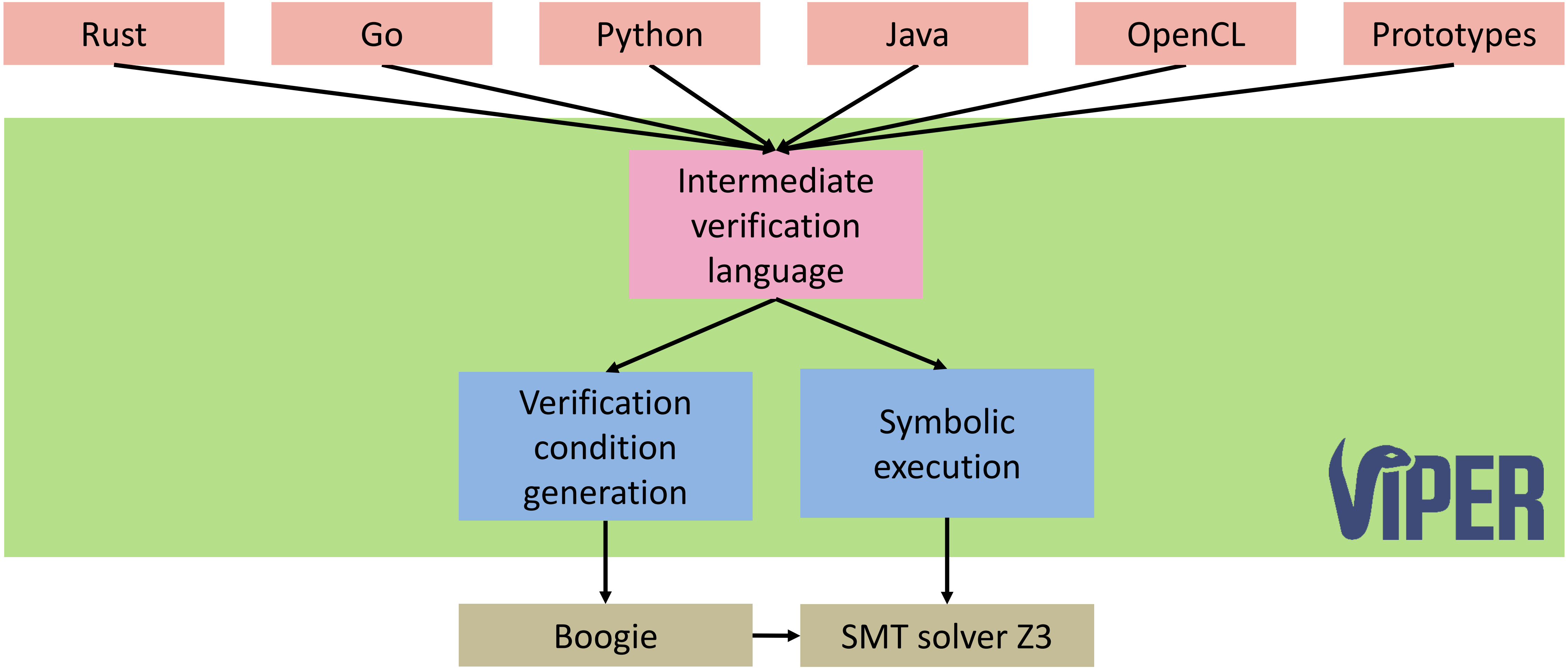
Verification problem for PL2

Given a PL2 program $\mathtt{S}$, is $\mathit{wp}[\![\, \mathtt{S} \,]\!]\,(\mathtt{true})$ valid?

Summary: weakest preconditions for PL2

PL2 program S	$wp[\![\, S \,]\!](\mathbf{B})$
$x := E$	$\mathbf{B}[x \,/\, E]$
assert $\mathbf{A}$	$\mathbf{A} \wedge \mathbf{B}$
assume $\mathbf{A}$	$\mathbf{A} \Rightarrow \mathbf{B}$
$S_1 \,;\, S_2$	$wp[\![\, S_1 \,]\!](wp[\![\, S_2 \,]\!](\mathbf{B}))$
if $(\,*\,)\ \{\, S_1 \,\}$ **else** $\{\, S_2 \,\}$	$wp[\![\, S_1 \,]\!](\mathbf{B}) \wedge wp[\![\, S_2 \,]\!](\mathbf{B})$

The Viper verification infrastructure



The PL2 fragment of Viper

```
method main() { // name is irrelevant for now

  var x: Int; var y: Int

  assume x == 1; // semicolon is optional

  if (x != y) {
    x := x + y
    y := x – y
    x := x – y
  } else {
    // else-block is optional
  }

  assert y == 1
}
```

Viper statements need to be placed in methods

Preamble for variables in the precondition

Precondition { x == 1 }

Statements, expressions, and assertions include PL2 up to minor syntax changes, in particular, if-statements instead of non-deterministic choice

Postcondition { y == 1 }

Macros

Viper supports simple parameterized macros

- syntactically inlined if invoked
- not type-checked before inlining
- untyped parameters
- recursion is not allowed

```
// Macro for expressions / assertions
define inc(a) (a + 1)

method main() {
  assert inc(16) == 17
}
```

```
// Macros for program statements
define isPositive(i) {
  assert i > 0
}

method main() {
  var x: Int
  if (x > 17) {
    isPositive(x)
  }
}
```

Viper examples

```
method main() {
  var x: Int
  var y: Int
  var res: Int

  assume true

  res := x*x + 2*x*y + y*y

  assert res == (x+y) * (x+y)
}
```

```
define diverge() {
  assume false
}

define skip() {
  assert true
}

define abort() {
  assert false
}

method main() {
  skip()
  diverge()
  abort()
}
```

Exercise: first Viper example

Use Viper to expose the overflow issue in the code below for 16-bit integers in two's complement.

Recall: `INT_MAX` = +32767, `INT_MIN` = -32768

```
if (i < 0) {
  res := -i
} else {
  res := i
}
```

Global variable declarations

- So far, we assumed implicitly that all programs and specifications are correctly typed

- In an implementation of a verifier, we need to make the types explicit, especially because SMT solvers require variables to be declared with a sort

```
x = Int('x')
```

- We tacitly assume a preamble of variable declarations

> Global declarations
> $D ::= \textbf{var}\ x : T \mid D ; D$

All variables in a Hoare triple must be declared in the preamble

```
var x: Int;
var b: Bool
// -----------
{ x = 23 }
if (b) {
  x := 42
} else {
  x := x + 17
}
{ x ≥ 40 }
```

- The initial value of all variables is unknown
  - We check the validity of $\models A \Rightarrow wp[\![S]\!](B)$ for all interpretations that is, for all initial variable values

Local variable declarations

```
var x: Int
var y: Int
// ----
{ x ≠ y }
if (x > 0) {
  var t: Int
  t := x
  x := y
  y := t
} else {
  skip
}
{ x ≠ y }
```

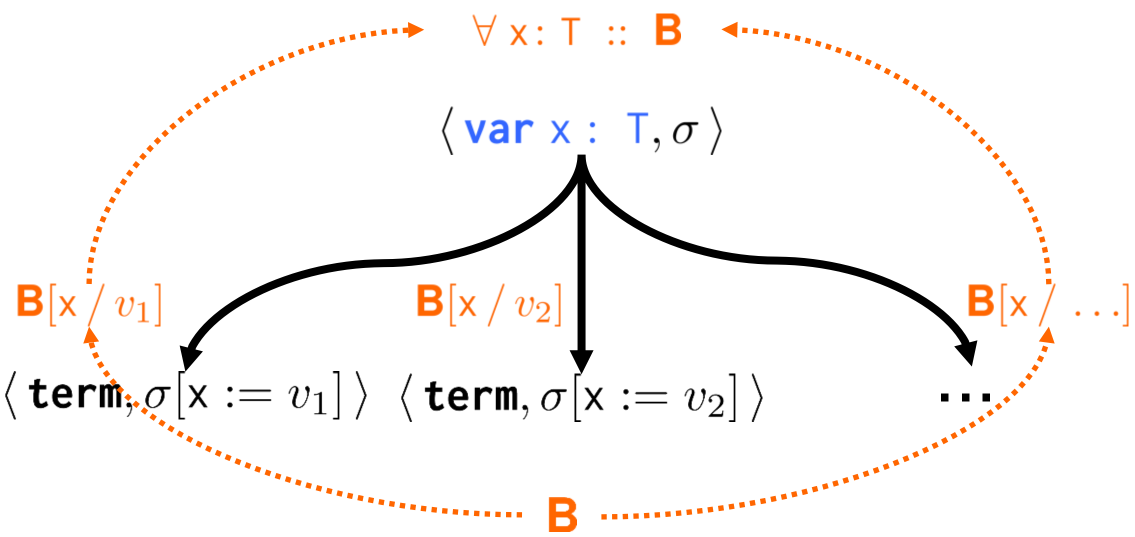
- Local variables improve the structure and readability of code
- `var x: T` declares a local variable
- Rules (checked by type checker)
  - All variables must be declared before they are used (local or global)
  - Local variables cannot be used outside the scope that declares them
  - Every variable is declared at most once for every trace
- No implicit initialization: locals start out with arbitrary value of their type

Reasoning about local variable declarations

Operational semantics

$$\frac{v \in \textsf{Values}(\mathsf{T})}{\langle\, \texttt{var}\ \mathsf{x} : \mathsf{T}, \sigma \,\rangle \rightarrow \langle\, \texttt{term}, \sigma[\mathsf{x} := v] \,\rangle}$$

$$\textsf{Values}(\texttt{Int}) = \mathbb{Z}$$
$$\textsf{Values}(\texttt{Bool}) = \{\texttt{true}, \texttt{false}\}$$
$$\ldots$$



"we know nothing about x before declaration"

"x has an arbitrary value right after declaration"

Weakest precondition

$$wp[\![\, \texttt{var}\ \mathsf{x} : \mathsf{T} \,]\!](\mathbf{A}) = \forall\, \mathsf{x} : \mathsf{T} :: \mathbf{A}$$

$[\![\, \texttt{var}\ \mathsf{x} : \mathsf{T} \,]\!] = \mathsf{x} := \mathsf{y}$

where y is a fresh global variable

PL3: Supporting global and local variables

Types

$T ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Rational} \mid \texttt{Real}$

Global declarations

$D ::= \texttt{var}\ x : T$

We assume that all variables, expressions, assertions, and programs are well-typed.

Expressions (executable)

$E ::= c \mid x \mid E + E \mid E * E \mid E - E \mid E < E \mid E \wedge E \mid E \vee E \mid \neg E \mid \ldots$

Assertions (FO logic over suitable theories)

$\mathbf{A} ::= E \mid \forall x : T :: \mathbf{A} \mid \exists x : T :: \mathbf{A} \mid \mathbf{A} \Rightarrow \mathbf{A} \mid \ldots$

PL3 statements

$$
\begin{aligned}
S ::= \; & \texttt{var}\ x : E \\
\mid \; & x := E \\
\mid \; & \texttt{assert}\ \mathbf{A} \\
\mid \; & \texttt{assume}\ \mathbf{A} \\
\mid \; & S ; S \\
\mid \; & \texttt{if}\ (E)\ \{S\}\ \texttt{else}\ \{S\}
\end{aligned}
$$

Exercise: encoding non-deterministic choice

We have seen earlier that we can encode if-statements into non-deterministic choice (using assume statements).

Show that it is also possible to encode non-deterministic choice into PL3.

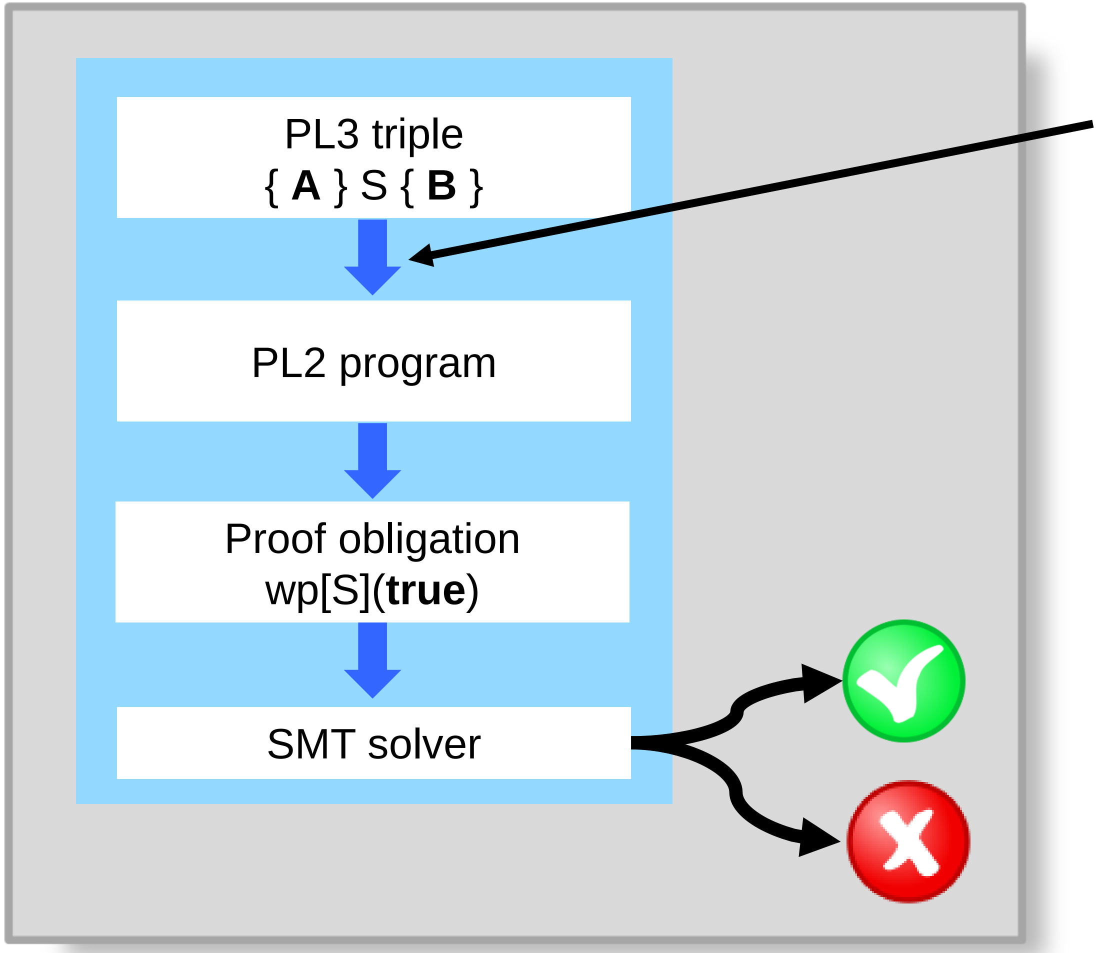
Apply the encoding to the following program and check the verification results.

```
var x: Int

if (*) {
  x := 42
} else {
  x := 23
}

assert x == 42 || x == 23  // succeeds
assert x == 42             // fails
assert x == 23             // fails
```

The tool stack so far



Encode

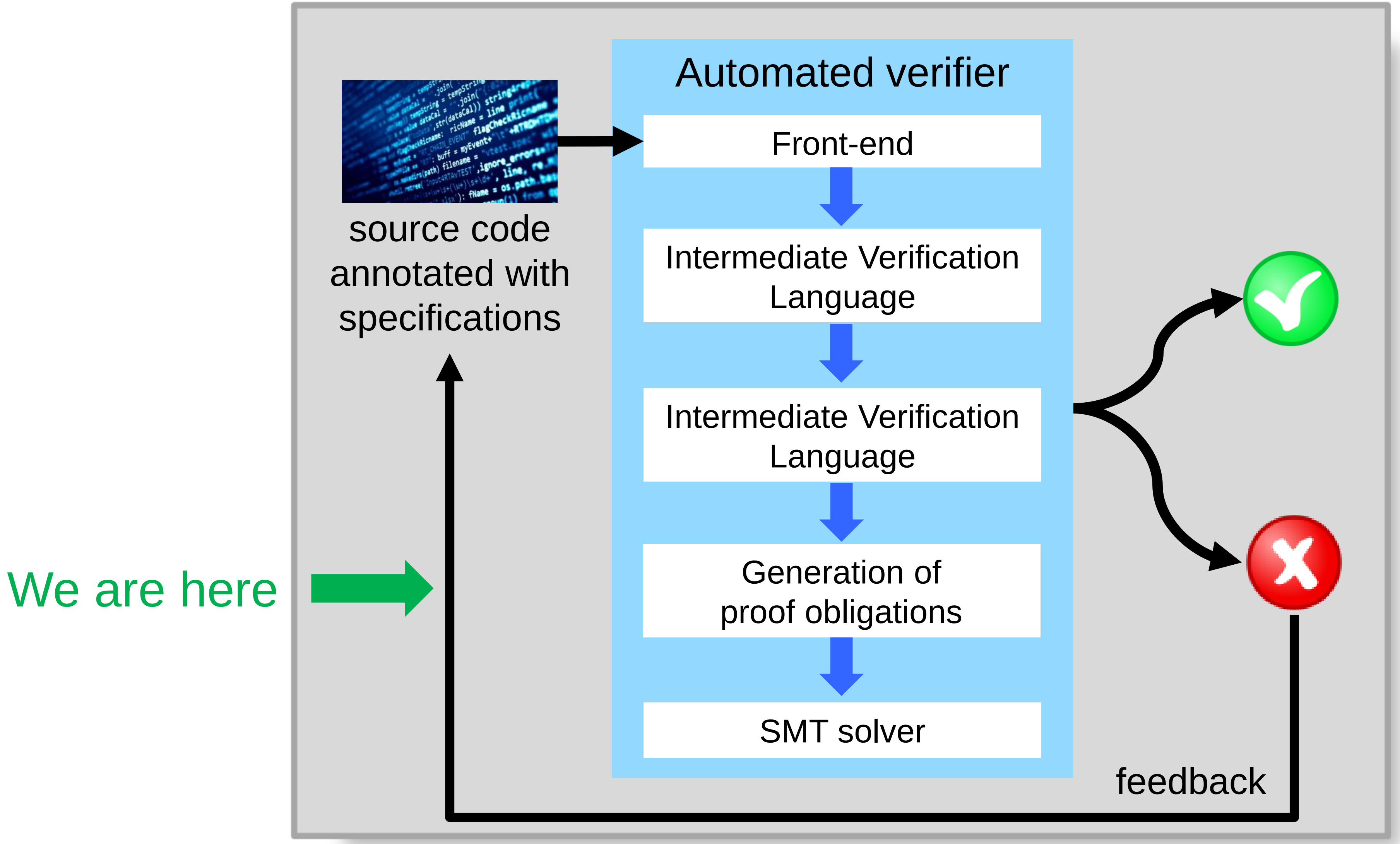
- Pre- and postcondition
- Local variables
- If-statements

Building a first verifier

1. Two intermediate verification languages

2. Error reporting

Roadmap



Automated verifier
Front-end
Intermediate Verification Language
Intermediate Verification Language
Generation of proof obligations
SMT solver
source code annotated with specifications
feedback
We are here

Verification debugging

Verification failures may be caused by:

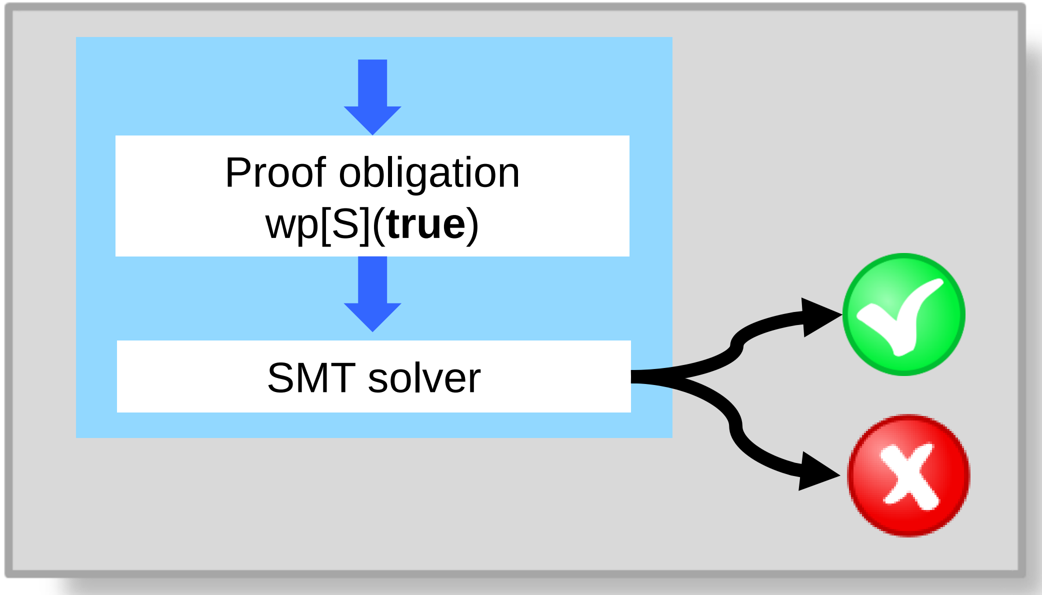
- Errors in the implementation
- Errors in the specification
- Insufficient annotations
  (e.g., missing loop invariants, as we will see later)
- Incompleteness of the verifier
  (spurious errors, false positives)

```
{ 0 ≤ b*b – 4*c }
discriminant := b*b – 4*a*c;
x := (-b + √discriminant) / 2
{ a*x² + b*x + c = 0 }
```

```
// Fermat's last theorem
assert 0<x && 0<y && 0<z ==>
       x*x*x + y*y*y != z*z*z
```

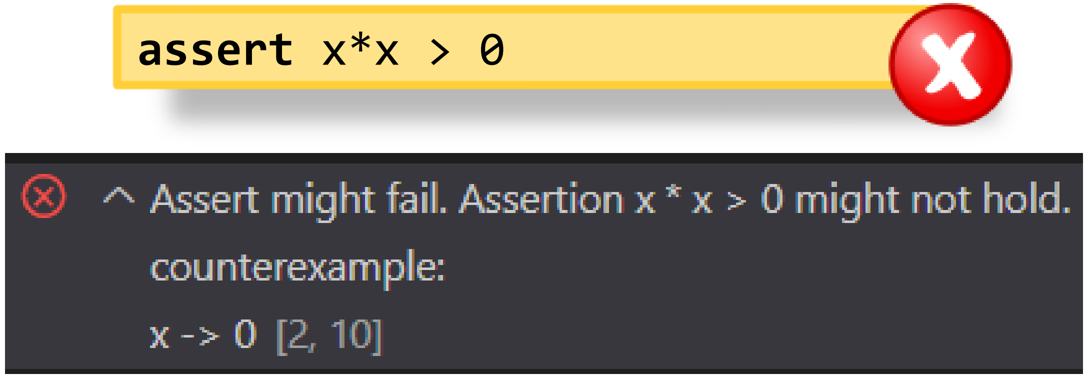
Verifiers should help users to localize and fix verification failures

Counterexamples



- Recall
  - Verifier checks validity of wp[S](true)
  - SMT solver checks the satisfiability of the negation, that is, of ¬wp[S](true)
  - Verification fails if the SMT solver returns sat, together with a model
  - If the verifier returns unknown, it typically provides at least a partial model

- Models contain a value for each variable, such that the proof obligation is not valid
- They are counterexamples to the correctness of the program
- Viper command line option --counterexample variables



Localizing errors

- Realistic programs contain a large number of proof obligations
  - For user-provided specifications such as postconditions
  - For all potential reasons for execution failures, e.g., division by zero, null-pointer dereferencing, out-of-bounds access
  - For other undesirable behaviors, e.g., overflows, data races, deadlocks

- To debug a verification error, it is crucial to know which of these proof obligations failed

- The technique so far checks validity of a single proof obligation wp[S](true), but cannot report which part of this proof obligation is invalid

```
assert MIN_INT <= x + y
assert x + y <= MAX_INT
res := x + y

assert MIN_INT <= x - y
assert x - y <= MAX_INT
d := x - y

assert d != 0
res := res / d
```

Verification failures

PL0 program S	$wp[\![S]\!](\mathbf{B})$
assert A	$\mathbf{A} \wedge \mathbf{B}$
assume A	$\mathbf{A} \Rightarrow \mathbf{B}$
$S_1 ; S_2$	$wp[\![S_1]\!](wp[\![S_2]\!](\mathbf{B}))$
if $(*) \{ S_1 \}$ **else** $\{ S_2 \}$	$(b \Leftrightarrow \mathbf{B}) \Rightarrow wp[\![S_1]\!](b) \wedge wp[\![S_2]\!](b)$
	where b is a fresh Boolean variable

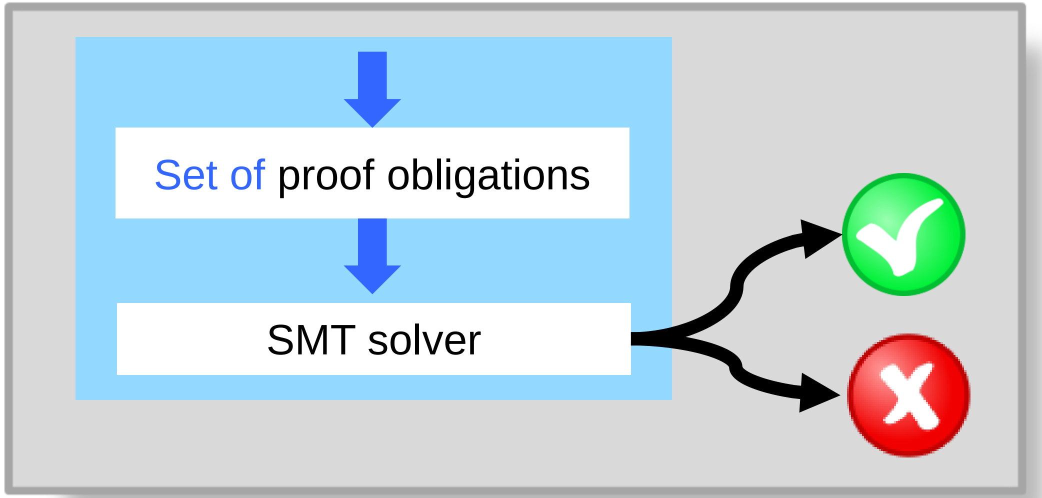
- Each verification error is caused by a failing assertion
- Since we check wp[S](true), assert statements are the only statements that lead to non-trivial proof obligations

- To determine which assertion to blame, we split the program at assertions into multiple verification problems

Computing multiple proof obligations

- *mwp* is a weakest precondition transformer that computes a set *M* of proof obligations
- To verify a statement S, compute *mwp*[S](∅)

PL0 program S	*mwp*$[\![$ S $]\!](M)$
assert A	$M \cup \{ \mathbf{A} \}$
assume A	$\{ \mathbf{A} \Rightarrow \mathbf{B} \mid \mathbf{B} \in M \}$
S_1 ; S_2	*mwp*$[\![S_1]\!]($*mwp*$[\![S_2]\!](M))$
if $(*)$ $\{ S_1 \}$ **else** $\{ S_2 \}$	*mwp*$[\![S_1]\!](M) \cup$ *mwp*$[\![S_2]\!](M)$



- Verification succeeds if all proof obligations are valid
- For each failed proof obligation, report the corresponding assertion

Exercise: error localization

Compute *mwp*[S](∅) for the statement on the right.

Which of the proof obligations are valid?

For each invalid proof obligation, find an initial state such that the corresponding assertion fails

```
if(*) {
  assert x == 7
} else {
  assert x == 2
  assert x > 0
}
```

Verify the example on the right in Viper using the Carbon verifier. How many error messages do you get?

Hint: CTRL+L allows you to choose the verifier.

```
method foo(x: Int, b: Bool) {
  if(b) {
    assert x == 7
  } else {
    assert x == 2
    assert x > 0
  }
}
```

Avoiding masked verification errors

- Both *wp* and *mwp* ignore the order of assertions

$$wp[\![\, \texttt{assert}\ \mathbf{A}_1\ ;\ \texttt{assert}\ \mathbf{A}_2\,]\!](\mathbf{B}) \;=\; \mathbf{A}_1 \wedge \mathbf{A}_2 \wedge \mathbf{B}$$

$$mwp[\![\, \texttt{assert}\ \mathbf{A}_1\ ;\ \texttt{assert}\ \mathbf{A}_2\,]\!](M) \;=\; M \cup \{\,\mathbf{A}_1\,\} \cup \{\,\mathbf{A}_2\,\}$$

```
assert x == 2
assert x > 0
```

```
assert x > 0
assert x == 2
```

- We would like to check the second assertion only for executions that may reach it, that is, in which the first assertion holds

- We achieve this by adding an assumption after each assertion

```
assert A
```

```
assert A
assume A
```

Error reporting in Viper

- Viper has two verification backends

- Carbon
  - Uses weakest preconditions, similarly to the technique taught in this course, but replaces *mwp* by a more efficient approach
  - Counterexamples can be enabled via command line option
  - Reports multiple verification failures

- Silicon
  - Uses symbolic execution
  - Counterexamples can be enabled via command line option
  - Reports only one verification error per method (use command line option to enable multiple errors)
  - Default verifier in the IDE

References

- Weakest preconditions
  - Edsger W. Dijkstra:
    Guarded commands, nondeterminacy and formal derivation of programs. 1975
  - Cormac Flanagan, James B. Saxe:
    Avoiding exponential explosion: generating compact verification conditions. 2001
  - Mike Barnett, K. Rustan M. Leino:
    Weakest-precondition of unstructured programs. 2005

- Error localization (alternative approach)
  - K. Rustan M. Leino, Todd Millstein, James B. Saxe:
    Generating error traces from verification-condition counterexamples. 2005