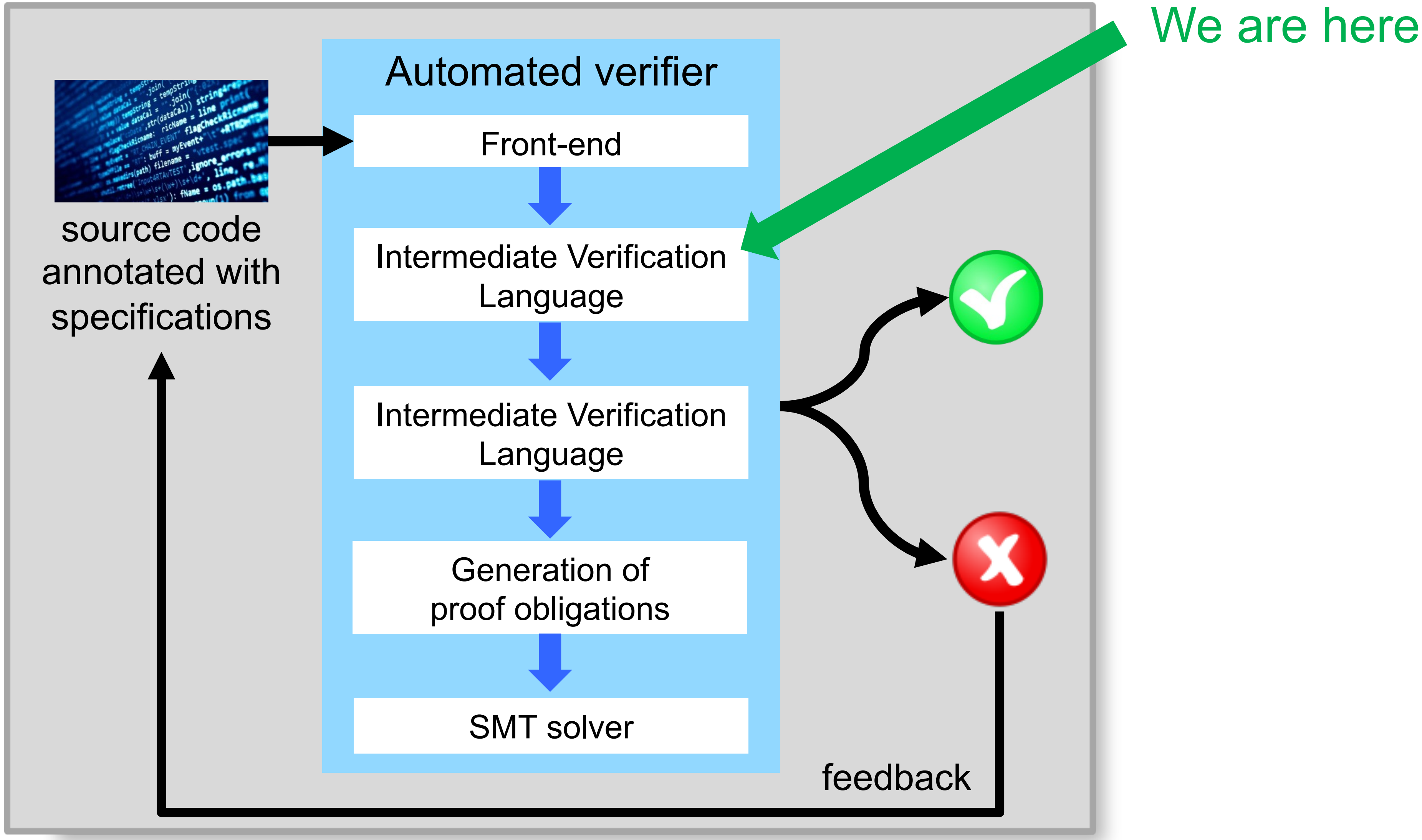
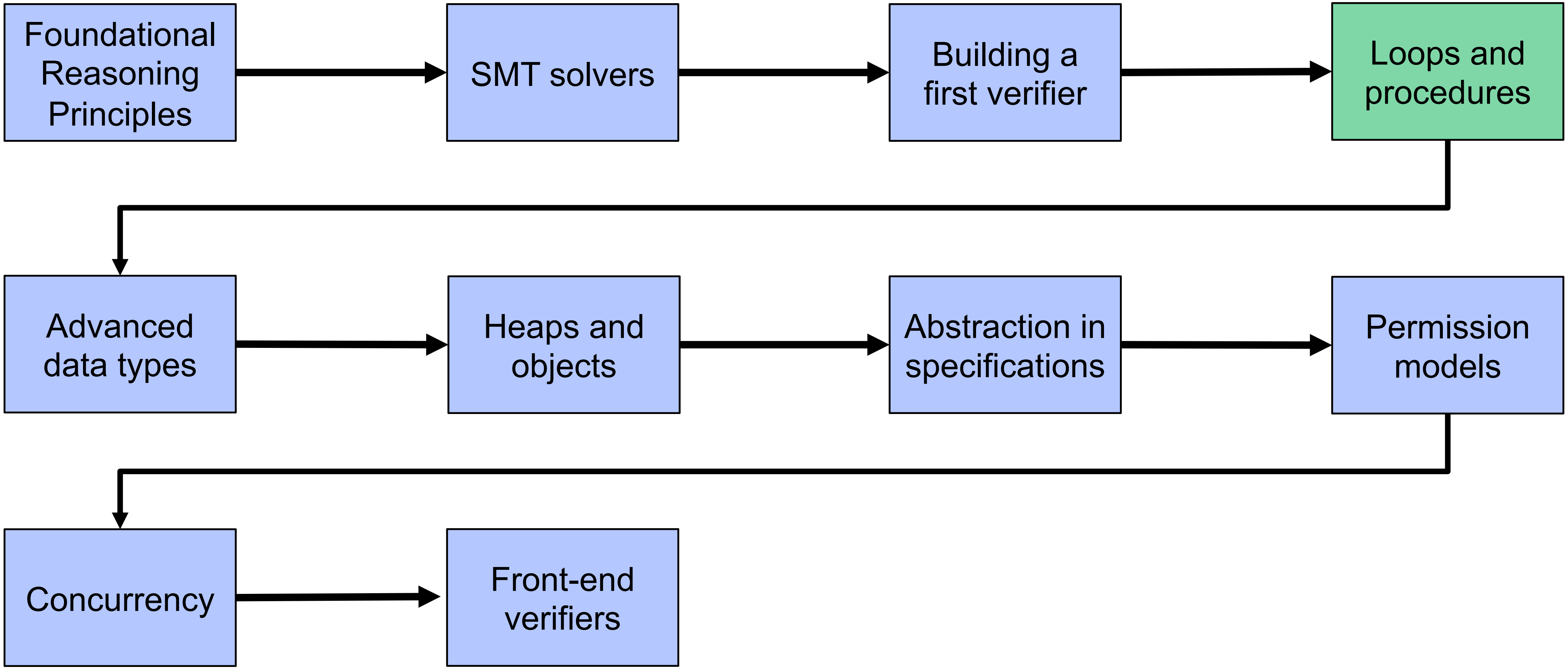
### 02245 – Chapter 4

# LOOPS & PROCEDURES

# Roadmap


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

# Tentative course outline



Foundational Reasoning Principles

SMT solvers

Building a first verifier

Loops and procedures

Advanced data types

Heaps and objects

Abstraction in specifications

Permission models

Concurrency

Front-end verifiers

02245 – Chapter 4.2

# PROCEDURES

# Example – procedure & client

```
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(6)
    assert z == 18

    // z := triple(7) ← FAILS
}
```

- Procedures
  - Define their own scope
  - Specify a contract
  - May be abstract
  - May be recursive
- Modular verification of calls
  - Inspects method contracts
  - Does *not* inspect implementations
  - Avoid client re-verification if implementation changes
  - Respects information hiding

# Example – abstract procedure

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

```
method client()
{
  var i: Int
  i := isqrt(25)
  assert i == 5
}
```

- Abstract procedures
  - Specify a contract
  - Have no implementation
  - Use case: code that cannot be verified
  - Are assumed correct ➔ part of trusted codebase
- Clients of abstract procedures are identical to clients of ordinary procedures

# Example – recursive procedure

```
method factorial(n: Int)
  returns (res: Int)
  requires 0 <= n
  ensures  1 <= res && n <= res
{
  if (n == 0) {
    res := 1
  } else {
    res := factorial(n-1)
    res := n * res
  }
}
```

- Very weak specification
- We will soon consider more intricate contracts

```
method client() {
    var x: Int
    x := factorial(5)
    assert 5 <= x
}
```

# Outline

- Language extension to PL2
- Partial correctness reasoning
- Encoding
- Global Variables
- Termination

# Extending the language (PL2)

## Declarations

D ::= **method** <name>($\overline{x:T}$) (input parameters)

(**returns** ($\overline{y:T}$))? (output parameters)

(**requires** P)* (precondition)

(**ensures** Q)* (postcondition)

({ S })? (method body)

| D;D

distinct sequences

## Statements

S ::= ... (as before)

| $\overline{z}$ := <name>($\overline{e}$) (possibly recursive call)

tuple of expressions with same types as $\overline{x}$

- All statements are placed in methods
- We consider only well-typed programs
- All variables are *local* to a method
- All parameters are *call-by-value*
- Methods are *not* mathematical functions

➔ no method calls in predicates

# Semantics via inlining (sketch)

$\mathbf{method}\ \mathtt{foo}(\overline{\mathtt{x{:}T}})\ \mathbf{returns}\ (\overline{\mathtt{y{:}T}})\ \{\ \mathtt{S}\ \}$

$\overline{\mathtt{z}}\ \mathtt{:=}\ \mathtt{foo}(\overline{\mathtt{a}})\quad \sim \quad \overline{\mathtt{x\ :=\ a}}\ ;\ \mathtt{S}\ ;\ \overline{\mathtt{z\ :=\ y}}$

“semantically equivalent to”

$$\begin{aligned} & \mathit{\mathbf{WP}}(\overline{\mathtt{z}}\ \mathtt{:=}\ \mathtt{foo}(\overline{\mathtt{a}}),\ \mathtt{Q}) \\ = \ & \mathit{\mathbf{WP}}(\overline{\mathtt{x\ :=\ a}}\ ;\ \mathtt{S}\ ;\ \overline{\mathtt{z\ :=\ y}},\ \mathtt{Q}) \end{aligned}$$

may contain other calls to `foo`

```
{ true }
var n: Int := 1
{ n == 1 }
n := double(n)
{ n == 2 }
n := double(n)
{ n == 4 }
```

different postconditions

Semantics again given by fixed points (FP)

- higher-order FP for each procedure

  `FP(foo): Pred → Pred`

- total correctness: least FP
- partial correctness: greatest FP

# Procedure inlining

- One could verify procedure calls like macros by inlining the procedure implementation

- However, inlining has several drawbacks:
  - it does not work for recursive procedures
  - it does not work when the implementation is not known statically (e.g., dynamic binding)
  - it does not support implementations that cannot be verified (e.g., foreign functions, binary libraries, complex code)
  - it increases the program size substantially and slows down verification
  - it is not modular; clients need to be re-verified when the procedure implementation changes

```
method factorial(n: Int)
returns (res: Int) {
  if (n == 0) {
    res := 1
  } else {
    res := factorial(n-1)
    res := n * res
  }
}
```

```
void foo(Collection c) {
  c.add(“Hello”);
}
```

```
void bar(FileOutputStream f) {
  f.write(5);
}
```

```
textEncryptor.encrypt(myText);
```

# Modular reasoning about procedures

- Goal: verify procedures modularly, that is, independently of their callers

- Verify that implementation satisfies the specification
  - Rely on precondition
  - Check postcondition

```
method factorial(n: Int)
returns (res: Int)
  requires 0 <= n
  ensures  1 <= res && n <= res
{
  res := n + 1
}
```

✓

- Verify every caller against the specification
  - Check precondition
  - Rely on postcondition

```
x := factorial(5)

assert 1 <= x   // succeeds
assert x == 6   // fails
```

✗

# Outline

- Language extension to PL2
- Partial correctness reasoning
- Encoding
- Global Variables
- Termination

# Proof obligations

$$\begin{aligned}&\textbf{method}\ \texttt{foo}(\overline{\texttt{x:T}})\\&\quad\textbf{returns}\ (\overline{\texttt{y:T}})\\&\quad\textbf{requires}\ P\\&\quad\textbf{ensures}\ Q\\&\{\ S\ \}\end{aligned}$$

- Procedure implementation satisfies its contract

$$\texttt{valid: } \{\ P\ \}\ S\ \{\ Q\ \}$$

- To handle recursion, proof may assume that all procedures satisfy their contract approximating ***WP***

- Verify caller against contract

consult declared contract

Call rule

$$\frac{\{\ P\ \}\ \textbf{method}\ \texttt{foo}(\overline{\texttt{x:T}})\ \textbf{returns}\ (\overline{\texttt{y:T}})\ \{\ Q\ \}}{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \}}$$

account for arguments (assuming z does not appear in a)

# Procedure framing

- We often need to prove that a property is not affected by a call
  - For loops, the analogous problem was solved by strengthening the loop invariant
  - We cannot strengthen the procedure specification for each call

Call rule

$$\frac{\{\ P\ \}\ \textbf{method}\ \texttt{foo}(\overline{x\texttt{:}T})\ \textbf{returns}\ (\overline{y\texttt{:}T})\ \{\ Q\ \}}{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \}}$$

```
x := 0
z := factorial(5)
assert x == 0
```

- To enable framing, we need a dedicated frame rule for local variables

Frame rule for local variables

$$\frac{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \}}{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \&\&\ R\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \&\&\ R\ \}}$$

where no variable in
$\overline{z}$ appears free in $R$

# Example – modular reasoning and recursion

- To show: implementation satisfies contract

```
{ 0 <= n }
res := factorial(n)
{ 1 <= res && n <= res }
```

- Proof by induction on the number k of calls

```
method factorial(n: Int)
  returns (res: Int)
  requires 0 <= n
  ensures  1 <= res && n <= res
{
  if (n == 0) {
    res := 1
  } else {
    res := factorial(n-1)
    res := n * res
  }
}
```

# Example – modular reasoning and recursion

- To show: implementation satisfies contract

```
{ 0 <= n }
res := factorial(n)
{ 1 <= res && n <= res }
```

- Proof by induction on the number k of calls

- **Base case k `==` `0`:** For every initial state, there is at most one execution without any recursive call

```
{ 0 <= n }
{ n == 0 ==> 1 <= 1 && n <= 1 }
  assume n == 0
{ 1 <= 1 && n <= 1 }
  res := 1
{ 1 <= res && n <= res }
```

✓

```
method factorial(n: Int)
  returns (res: Int)
  requires 0 <= n
  ensures  1 <= res && n <= res
{
  if (n == 0) {
    res := 1
  } else {
    res := factorial(n-1)
    res := n * res
  }
}
```

# Example – modular reasoning and recursion

- To show: implementation satisfies contract

```
{ 0 <= n }
res := factorial(n)
{ 1 <= res && n <= res }
```

- Proof by induction on the number k of calls

- **Induction hypothesis:** assume for all executions with at most k calls that calls satisfy the contract

```
{ 0 <= n }
res := factorial(n)
{ 1 <= res && n <= res }
```

I.H.

```
method factorial(n: Int)
  returns (res: Int)
  requires 0 <= n
  ensures  1 <= res && n <= res
{
  if (n == 0) {
    res := 1
  } else {
    res := factorial(n-1)
    res := n * res
  }
}
```

# Example – modular reasoning and recursion

- To show: implementation satisfies contract

```
{ 0 <= n }
res := factorial(n)
{ 1 <= res && n <= res }
```

- Proof by induction on the number k of calls

- **Induction step:** using the induction hypothesis, show that the implementation satisfies the contract for executions with at most k + 1 calls.

```
{ 0 <= n }
res := factorial(n)
{ 1 <= res && n <= res }
```

I.H.

```
{ 0 <= n }
{ (n == 0 && 0 <= n)
  || (0 <= n && n != 0) }
  if (n == 0) {
    { n == 0 && 0 <= n }
    res := 1
    { 1 <= res && n <= res }
  } else {
    { 0 <= n && n != 0 }
    { 0 <= n && 0 <= n && n != 0 }
    res := factorial(n-1)
    { 1 <= res && n – 1 <= res
      && 0 <= n && n != 0 }
    { 1 <= n * res && n <= n * res }
    res := n * res
    { 1 <= res && n <= res }
  }
{ 1 <= res && n <= res }
```

framing

# Example – partial correctness reasoning

```
method toBinary(d: Int)
    returns (res: Int)
    requires 0 <= d
    ensures  d % 2 == res % 10
{
    res := toBinary(d/2)
    res := res * 10 + (d % 2)
}
```

- Method never terminates
  - Proof argument becomes cyclic
- No induction base!
  - Technically, we reason about a greatest fixed point and do co-induction (think: bisimulation)
- Induction step can be verified

➔ verifies with respect to partial correctness:
whenever execution stops (here: never), the postcondition holds

# Procedures in Viper

```
method divide(n: Int, d: Int)
returns (q: Int, r: Int)
  requires 0 <= n
  requires 1 <= d
  ensures  n == q*d + r
{
  if (n < d) {
    q := 0
    r := n
  } else {
    q, r := divide(n-d, d)
    q := q + 1
  }
}
```

- Multiple pre- / postconditions allowed
  - Will be conjoined
- Calls are statements
  - No calls in (compound) expressions
  - Parallel assignment of return values
- No return statement: final value of result variables will be returned
- All variables are local
  - Framing is straightforward
- Verification is modular, with partial correctness semantics

# Exercise

- Write a recursive method `sum` that yields the sum of the first `n` natural numbers.
- Provide a suitable specification.
- Check whether your specification is strong enough by verifying the client code below.
- Sketch the induction proof justifying why your implementation satisfies the specification

```
method main() {
  var r: Int

  r := sum(10)

  assert r == 55
}
```

- Implement the method below in a language of your choice.
- Run the method on various inputs and form a hypothesis about its behavior.
- Formalize your hypothesis as a Viper specification and verify the method

```
method M(n: Int) returns (r: Int)
{
  if (n > 100) {
    r := n - 10
  } else {
    r := M(n + 11)
    r := M(r)
  }
}
```

# Outline

- Language extension to PL2
- Partial correctness reasoning
- Encoding
- Global Variables
- Termination

# Encoding: procedure bodies

```
method foo(x:T)
  returns (y:T)
  requires P
  ensures Q
{ S }
```

- Procedure implementation satisfies the specification

```
valid: { P } S { Q }
```

  - To handle recursion, proof may assume that all procedures satisfy their specifications
  - Similarly to loops, this is sound as a correct contract is a pre-fixed point

- Generate one proof obligation per method declaration

```
assume P
// encoding of S
assert Q
```

- No proof obligation for abstract methods

# Preliminary encoding

```
method foo(x̄:T)
  returns (ȳ:T)
  requires P
  ensures Q
{ S }
```

## Verify caller against specification

**Call rule**

$$\frac{\{\ P\ \}\ \textbf{method}\ \texttt{foo}(\overline{x:T})\ \textbf{returns}\ (\overline{y:T})\ \{\ Q\ \}}{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \}}$$

**Frame rule for local variables**

$$\frac{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \}}{\{\ P[\ \overline{x}\ /\ \overline{a}\ ]\ \&\&\ R\ \}\ \overline{z} := \texttt{foo}(\overline{a})\ \{\ Q[\ \overline{x}\ /\ \overline{a}\ ][\ \overline{y}\ /\ \overline{z}\ ]\ \&\&\ R\ \}}$$

```
assert P[ x̄ / ā ]

var z̄ // reset all vars in z̄

assume Q[ x̄ / ā ][ ȳ / z̄ ]
```

- Check precondition
- Reset assigned variables
- Assume postcondition

# Encoding of calls: example

```
method foo(p: Int) returns (r: Int)
  requires 0 <= p
  ensures  r == p*p
```

```
x := 4
y := 4

z := foo(x)

assert y + z == 20
```

```
{ 0 <= 4 ∧ ∀z :: z == 4*4 ==> 4 + z == 20 }
x := 4
{ 0 <= x ∧ ∀z :: z == x*x ==> 4 + z == 20 }
y := 4
{ 0 <= x ∧ ∀z :: z == x*x ==> y + z == 20 }
assert 0 <= x
{ ∀z :: z == x*x ==> y + z == 20 }
var z
{ z == x*x ==> y + z == 20 }
assume z == x*x
{ y + z == 20 }
assert y + z == 20
{ true }
```

Framing happens implicitly by not resetting variables that cannot be changed by the call

# Permitting LHS variables in argument expressions

```
method inc(p: Int) returns (r: Int)
  ensures  r == p + 1
```

```
x := 4

x := inc(x)

assert false
```

- So far: LHS of assignments not allowed in arguments

```
{ ∀x :: x == x + 1 ==> false }
x := 4
{ ∀x :: x == x + 1 ==> false }
assert true  // implicit precondition
{ ∀x :: x == x + 1 ==> false }
var x
{ x == x + 1 ==> false }
assume x == x + 1
{ false }
assert false
{ true }
```

✓

- Parameters in the postcondition refer to values past into the call
- If result (LHS variable) of call occurs in actual parameters, the assumption after the havoc conflates the pre-call and post-call values

# Final encoding

```
assert P[ x̅ / a̅ ]
var e̅:̅T̅  := a̅
var z̅ // reset all vars in z̅
assume Q[ x̅ / e̅ ][ y̅ / z̅ ]
```

- Check precondition
- Save pre-call values of arguments
- Reset assigned variables
- Assume postcondition, with actual arguments evaluated in the pre-state

# Example

```
method inc(p: Int) returns (r: Int)
  ensures  r == p + 1
```

```
x := 4

x := inc(x)

assert false
```

```
assert P[ x̄ / ā ]

var e̅:̅T̅  := ā

var z̄ // reset all vars in z̄

assume Q[ x̄ / ē ][ ȳ / z̄ ]
```

```
{ ∀x’ :: x’ == 4 + 1 ==> false }
x := 4
{ ∀x’ :: x’ == x + 1 ==> false }
assert true  // implicit precondition
{ ∀x’ :: x’ == x + 1 ==> false }
e := x
{ ∀x :: x == e + 1 ==> false }
var x
{ x == e + 1 ==> false }
assume x == e + 1
{ false }
assert false
{ true }
```

Note that substituting e by x renames the bound variable from x to x’ to avoid binding the free variable x (capture-avoiding substitution)

# Outline

- Language extension to PL2
- Partial correctness reasoning
- Encoding
- Global Variables
- Termination

# Global variables

- We temporarily re-introduce global variables, such that procedures can have side effects
    - Viper has no global variables, but a global heap (later)
- Specifications of side effects need to relate the state after the call to the state before:

  *“The value of g is one larger than before the call.”*

- Postconditions may include `old(x)` expressions to refer to the pre-state value of global variable x
- Postconditions are two-state predicates
    - Evaluation depends on final and initial state

```
var g: Int // global variable

method inc()
  ensures ??
{
  g := g + 1
}
```

```
var g: Int // global variable

method inc()
  ensures g == old(g) + 1
{
  g := g + 1
}
```

# Exercise (5min)

- Propose an approach that enables framing for method calls in the presence of global variables.
- For example, the assertion on the right should verify when using your approach.

```
var g: Int // global variables
var h: Int
```

```
method inc()
  ensures g == old(g) + 1
{
  g := g + 1
}
```

```
g := 0
h := 0
inc()
assert h == 0
```

# Outline

- Language extension to PL2
- Partial correctness reasoning
- Encoding
- Global Variables
- Termination

# Proving termination

A method **variant** is an an expression V that decreases for every method call (for some well-founded ordering <).

< has no infinite descending chains

# recursive calls

$$V_1 > V_2 > V_3 > V_4 > \ldots > V_k$$

Method terminates because each call decreases a variant that cannot decrease indefinitely

| Well-founded | Not-well-founded |
|---|---|
| < over Nat | < over Int |
| ⊂ over finite sets | < over positive reals |

V decreases

fresh variable

```
         valid: P ==> V >= 0
{ P && V < v } z̄ := foo(ā) { Q && V == v }
-------------------------------------------
         { P } z̄ := foo(ā) { Q }
```

# Proving termination – encoding

```
method factorial(n: Int)
  returns (res: Int)
  requires 0 <= n
  decreases n // variant
{
  if (n == 0) {
    res := 1
  } else {
    res := factorial(n-1)
    res := n * res
  }
}
```

```
method factorial(n: Int)
  returns (res: Int)
  requires 0 <= n
{
  if (n == 0) {
    res := 1
  } else {
    var v: Int := n assert v >= 0
    assert n-1 < v
    res := factorial(n-1)
    assert n == v
    res := n * res
  }
}
```

Program with variant annotation
(not supported by default in Viper)

Encoded program

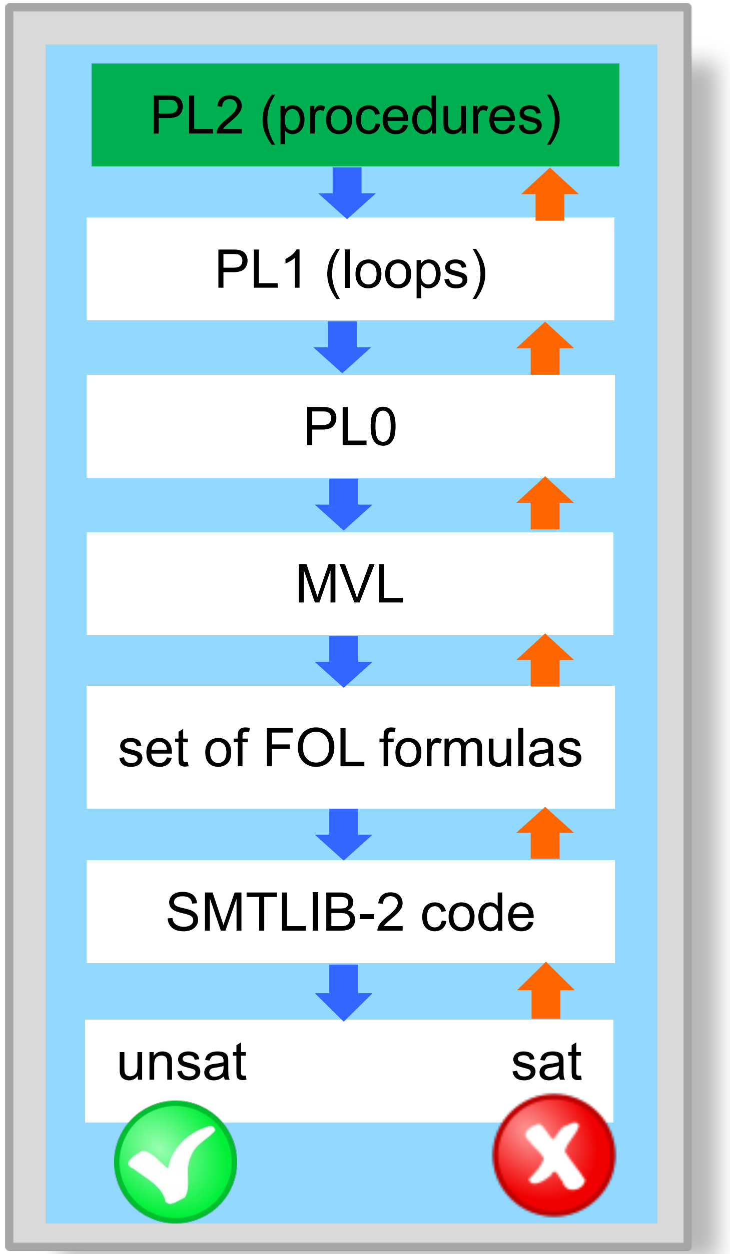
# Exercise

- Use Viper to prove that McCarthy’s 91 function (right) terminates.

```
method M(n: Int) returns (r: Int)
  requires n >= 0
  ensures 100 < n  ==> r == n - 10
  ensures n <= 100 ==> r == 91
{
  if (n > 100) {
    r := n - 10
  } else {
    r := M(n + 11)
    r := M(r)
  }
}
```

# Procedures: wrap-up

- We reason modularly about procedures by choosing suitable procedure specifications
  - Precondition constrains arguments
  - Postcondition constrains results
- Key property: framing
- Modular verification
  - Supports recursion
  - Avoids re-verification of clients after implementation changes
  - Enables reasoning about unverified code (e.g. libraries)
- Procedures can be encoded into PL0



PL2 (procedures)

PL1 (loops)

PL0

MVL

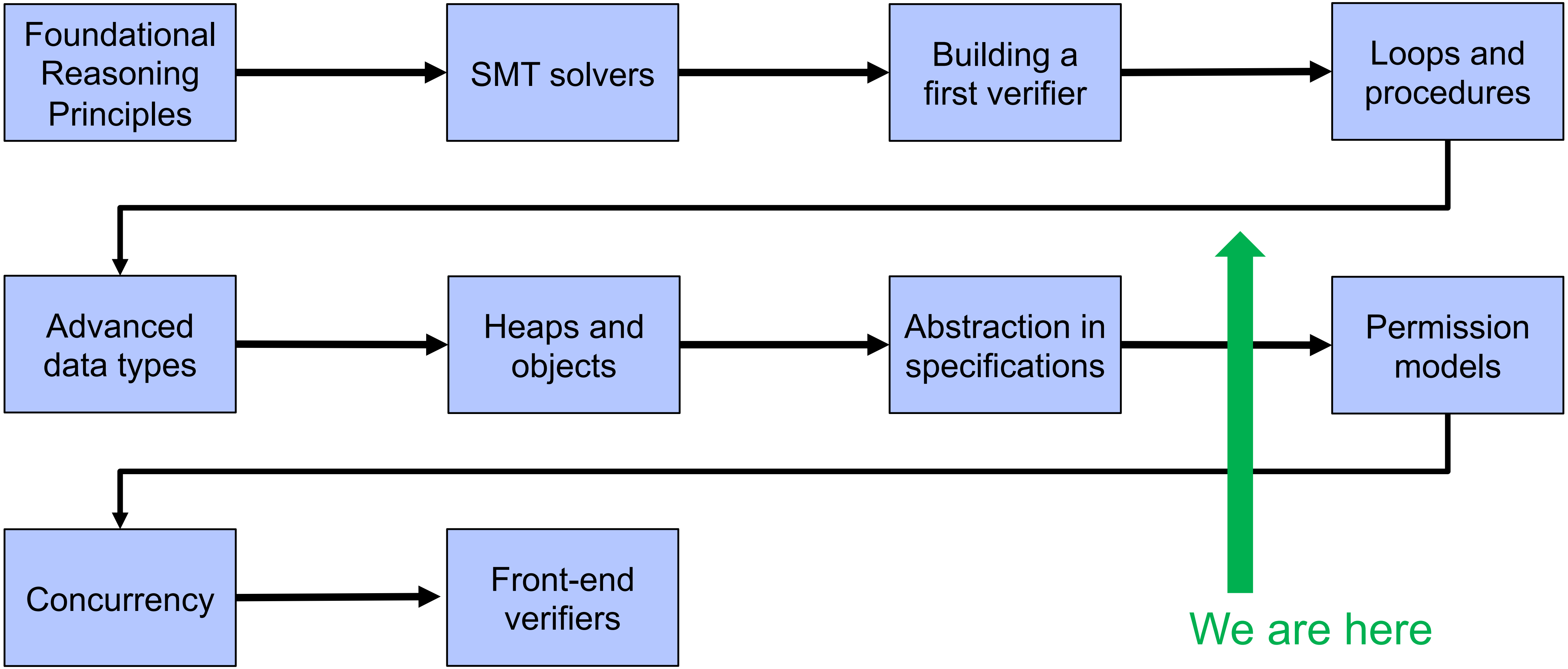
set of FOL formulas

SMTLIB-2 code

unsat ✓

sat ✗

# Tentative course outline



Foundational Reasoning Principles

SMT solvers

Building a first verifier

Loops and procedures

Advanced data types

Heaps and objects

Abstraction in specifications

Permission models

Concurrency

Front-end verifiers

We are here