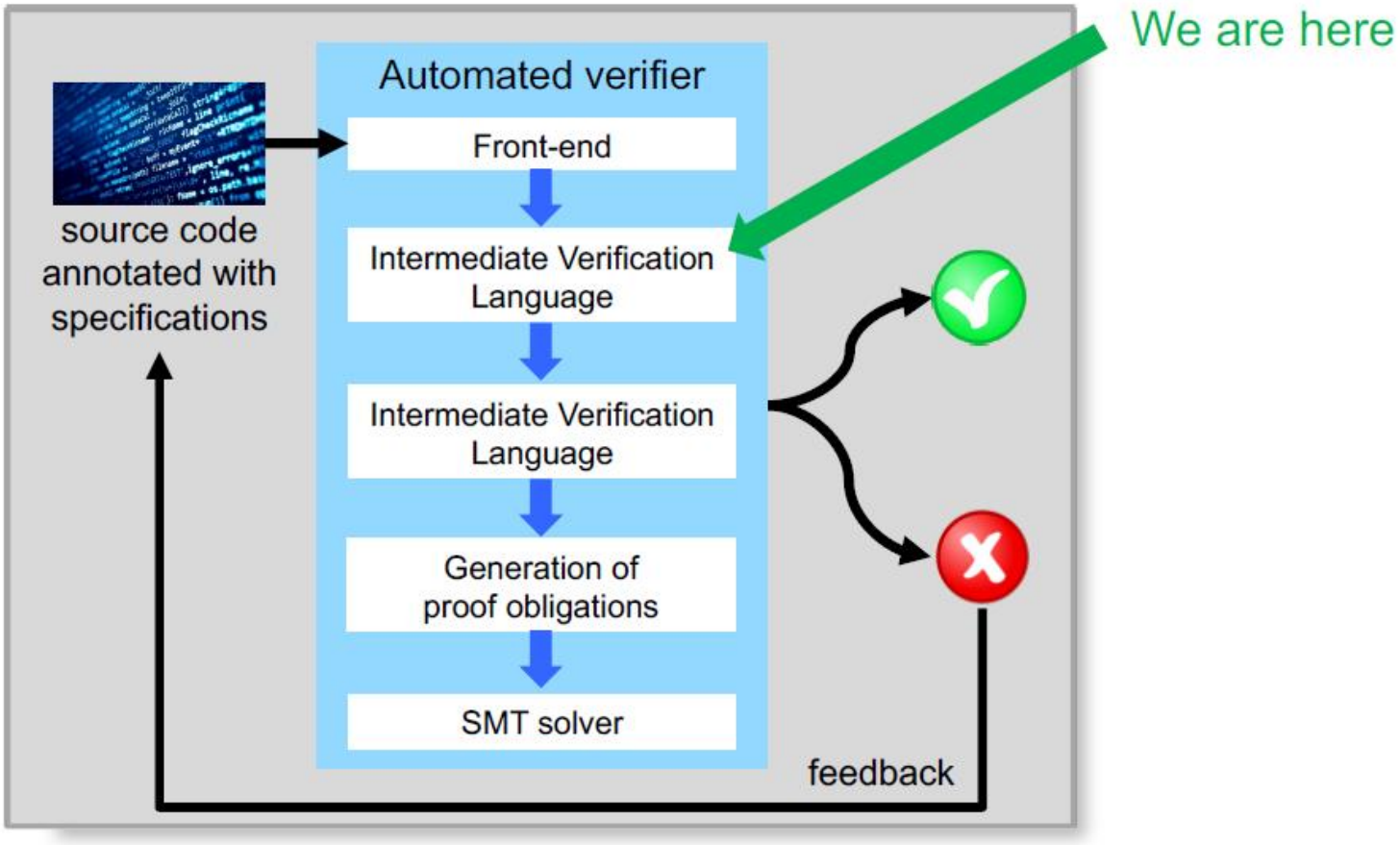
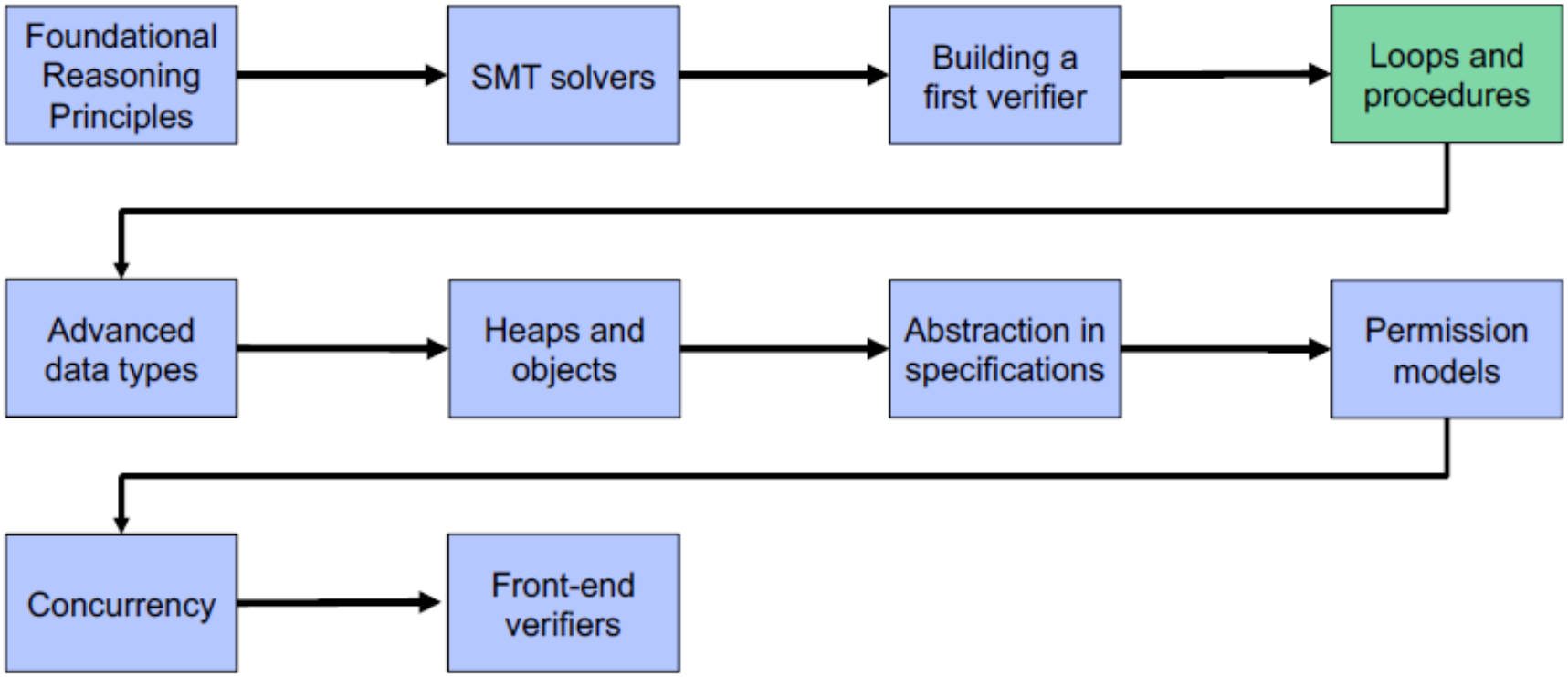
Program Analysis for Software Security

Lecture 7

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



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)
```

- Abstract procedures
  - Specify a contract
  - Have no implementation
  - Use case: code that cannot be verified
  - Are assumed correct ➔ part of trusted codebase

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

- 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>(x̄:T)        (input parameters)
       (returns (ȳ:T))?         (output parameters)
       (requires P)*                 (precondition)
       (ensures Q)*                 (postcondition)
       ({ S })?                      (method body)
    |  D;D
```

distinct sequences

Statements

```
S ::=  ...                                (as before)

    |  z̄ := <name>(ē)   (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)

```
method foo(x̄:T) returns (ȳ:T) { S }
```

```
z̄ := foo(ā)  ~  x̄ := ā ; S ; z̄ := ȳ
```

"semantically equivalent to"

```
WP(z̄ := foo(ā), Q)
= WP(x̄ := ā ; S ; z̄ := ȳ, Q)
```

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

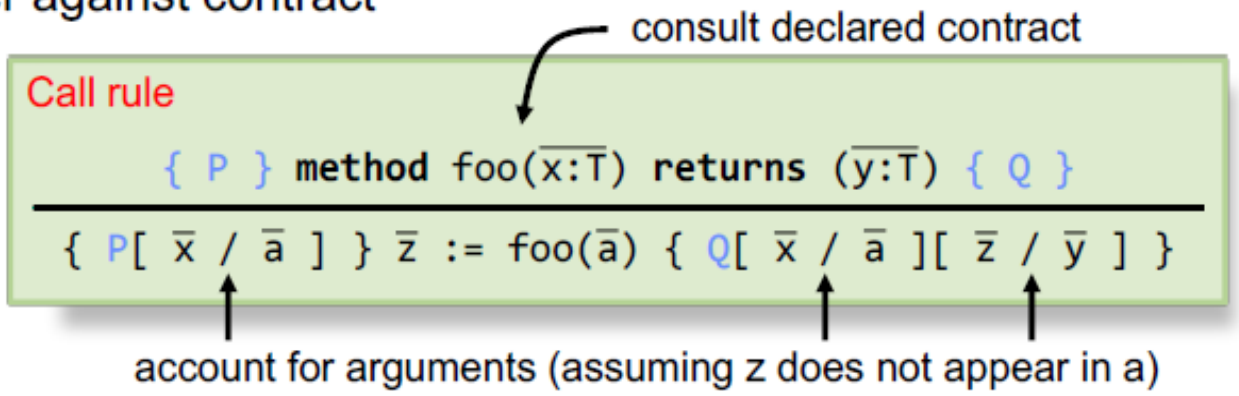
```
method foo(x̄:T)
  returns (ȳ:T)
  requires P
  ensures Q
{ S }
```

- Procedure implementation satisfies its contract

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

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

- Verify caller against contract



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

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

```
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

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

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)          I.H.
{ 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 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

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

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

Frame rule for local variables

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

```
assert P[ x̄ / ā ]
var z̄ // reset all vars in z̄
assume Q[ x̄ / ā ][ z̄ / ȳ ]
```

- 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[ x̄ / ā ]
var e:T := ā
var z̄ // reset all vars in z̄
assume Q[ x̄ / ē ][ ȳ / z̄ ]
```

- 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 e:T̄ := 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
}
```

Framing with global variables: non-solutions

```
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
```

- Bad idea: inspect body of callee to determine which global variables are modified
  - Not modular
  - Does not work for abstract methods
- Bad idea: assume conservatively that all global variables may be modified
  - Callee needs a specification `x == old(x)` for all global variables it does not change
  - Not modular: procedure specifications need to change when a new global variable is declared

Framing with global variables: modifies-clauses

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

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

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

- We (temporarily) introduce one more annotation for each procedure declaration
- A modifies clause lists all global variables that may be modified by the procedure
  - All other global variables must remain unchanged
- A procedure body can be checked syntactically to satisfy its modifies clause

Encoding with old-expressions and modifies clauses

```
var g:T // global variables

method foo(x: Int)
  requires P
  modifies h // subset of g
  ensures Q
```

```
assert P[ x / a ]
var e:T := a
var o:T := h
var h // reset all vars in z
var z
assume Q[ x / e ][ z / y ]
```

- Save pre-state value of modified globals
- reset *potentially* modified global variables
- Assume postcondition, with old-expressions replaced by pre-state values (last substitution is for unmodified variables)

Outline

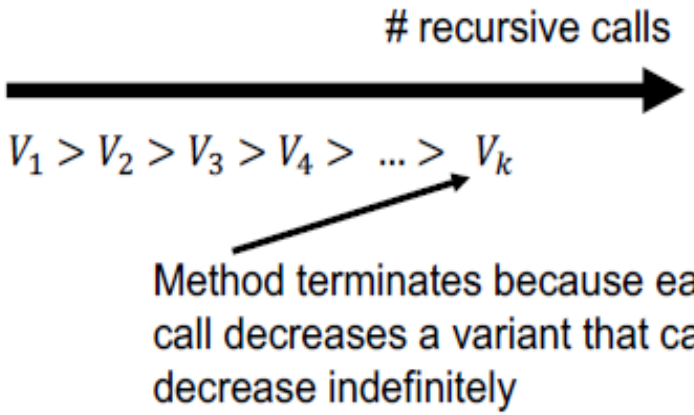
- Language extension to PL2
- Partial correctness reasoning
- Encoding
- Global Variables
- Termination

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

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



Proving termination

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

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

precondition ensures V never becomes negative

Proof obligations for implementation

```
valid: { P && V == v } S { Q }        valid: P ==> V >= 0
```

fresh variable saving initial value of V

recursive calls are verified using the call rule

consult contract with decreasing variant as if already proven

Call rule

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

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
  }
}
```

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

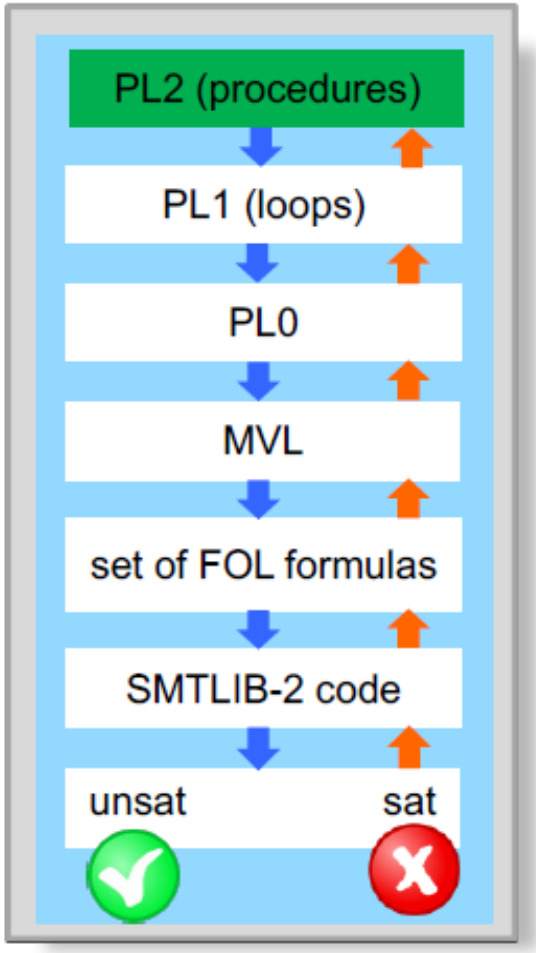
```
define V(m) (m) // variant

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

Encoded program

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



Domain theory

When reasoning about loops and procedures, we typically need to reason about fixed points; a fixed point of a function `f` is some `x` such that `f(x) = x`. This kind of reasoning requires a few definitions and results from domain theory, particularly complete lattices, monotone functions, the Knaster-Tarski fixed point theorem, continuous functions, and Kleene's fixed point theorem.

Partial orders

In the following let `D` be a non-empty set and `⊑: D x D` a binary relation over the elements of `D`.

We call the pair `(D, ⊑) a *partial order* (or a poset: partially ordered set) if for all

- ⊑ is reflexive: for all `a ∈ D`, `a ⊑ a`;
- ⊑ is transitive: for all `a,b,c ∈ D`, if `a ⊑ b` and `b ⊑ c`, then `a ⊑ c`; and
- ⊑ is antisymmetric: for all `a,b,c ∈ D`, if `a ⊑ b` and `b ⊑ a`, then `a = b`.

For example, $(\mathbb{Z}, \leq)$ and $(\mathbb{N}, \leq)$ - the sets of integers and natural numbers with the usual ordering $\leq$ - are partial orders. Another example with perhaps a less obvious order are predicates:

Example. Recall that we initially did not fix a specific syntax for predicates. Instead, we considered the set

```
Pred ::= { P | P: States -> Bool }
```

consisting of arbitrary mappings from states to `Bool = {true, false}`. We can order predicates by pointwise logical implication, i.e. for `⊑ ::= ==>`. More formally, for for two predicates `P, Q in Pred`, we define

```
P ==> Q  iff for all states s, the predicate P(s) ==> Q(s) maps to true.
```

Lemma. `(Pred, ==>)` is a partial order.

We leave the proof, i.e. showing that `==>` is indeed reflexive, transitive, and antisymmetric

Complete lattices

Let $(D, \sqsubseteq)$ be a partial order and let $S \subseteq D$ be any subset of D.

- We call an element $d \in D$ an *upper bound* of S if it is larger than every element of S, that is, iff $s \sqsubseteq d$ holds for all $d \in D$.
- If there is no other upper bound of S that is smaller than d, then d is the *least upper bound* of S; formally: for all $d \in D$, if d' is an upper bound of S, then $d \sqsubseteq d'$.
- We also call the least upper bound (with respect to the considered ordering relation $\sqsubseteq$) the *supremum* of S and denote it by $\sup S$.
- Analogously, $d \in D$ is a *lower bound* of S if it is smaller than every element of S, that is, iff $d \sqsubseteq s$ holds for all $d \in D$.
- If there is no other lower bound of S that is larger than d, then d is the *greatest bound* of S; formally: for all $d \in D$, if d' is a lower bound of S, then $d' \sqsubseteq d$.
- We also call the greatest lower bound (with respect to the considered ordering relation $\sqsubseteq$) the *infimum* of S and denote it by $\inf S$.

For example, consider the partial order `(Pred, ==>)` and the subset

```
S ::= {x == 1, x == 2, x == 3}
```

Then `x >= 1` is an upper bound of `S` since it is implies by each element. However, it is not the least upper bound; that is `x == 1 || x == 2 || x == 3`.

Not every subset of a partial order has a least upper bound or a greatest lower bound. For example, consider the partial order $(\mathbb{N}, \leq)$ and the subset $S ::= \mathbb{N}$. There is no natural number that is larger or equal to every natural number.

Complete lattice. A partial order $(D, \sqsubseteq)$ where every subset $S \subseteq D$ has a least upper bound in D is called a complete lattice.

In fact, if $(D, \sqsubseteq)$ is a complete lattice, then every subset $S \subseteq D$ automatically also has a greatest lower bound in D. Furthermore, it follows that D has a unique smallest element, called bot (bottom), which is given by

$$\mathit{bot} = \sup \{\} = \inf D$$

D also has a unique largest element, called top, which is given by

$$\mathit{top} = \inf \{\} = \sup D$$

Notice that neither $(\mathbb{Z}, \leq)$ nor $(\mathbb{N}, \leq)$ can be complete lattices, since they have no largest element. By contrast, $(\mathbb{N} \cup \{\infty\}, \leq)$ is a complete lattice: for every subset `S`, the supremum of `S` is the maximal element of `S` if such an element exists; otherwise, the supremum `sup S` is ∞.

Monotone functions

Let `(D, ⊑)` be a complete lattice and let `f: D -> D` be a function mapping elements in `D` to elements in `D`. The function `f` is *monotone* if it preserves the ordering ⊑. Formally, `f` is monotone iff

```
f is monotone
iff
for all a,b ∈ D, a ⊑ b implies f(a) ⊑ f(b)
```

For example, the function given by `f(x) ::= x+1` is monotone for the complete lattice $(\mathbb{Z} \cup \{-\infty, \infty\}, \leq)$. By contrast, the function given by `f(x) ::= -x` is not.

Lemma. For every PL0 statement `S`, the mapping from postconditions `Q` to the weakest precondition of `S` and `Q`, that is the fucntion `WP(S, .): Pred -> Pred`, is monotone.

Fixed points of monotone functions

Let `(D, ⊑)` be a complete lattice. Moreover, let `f: D -> D` be a monotone function.

- We call `d ∈ D` a *fixed point* of `f` if `f(d) = d`.
- Every element `d in D` such that `f(d) <= d` holds is called a *pre-fixed point* of `f`.
- Conversely, every element `d in D` such that `d <= f(d)` holds is called a *post-fixed point* of `f`.

Every fixed point of `f` is thus both a pre-fixed and a post-fixed point of `f`.

In general, a function can have no, exactly one, or arbitrarily many fixed points:

- The function `f(x) ::= -x` has no fixed point when considering the complete lattice $(\mathbb{Z} \cup \{-\infty, \infty\}, \leq)$, since the result always oscillates between two possible results.
- For the function `f(x) ::= x`, every element in its domain is a fixed point.
- For the function `f(x) ::= x + 1` and the complete lattice $(\mathbb{N} \cup \{\infty\}, \leq)$, there is only one fixed point: we have `f(∞) = ∞`.

The Tarski-Knaster fixed point theorem ensures that every monotone function `f: D -> D` over a complete `(D, ⊑)` has at least one fixed point. Even better, the theorem gives characterizations of the least fixed point and the greatest fixed point of `f`: the least fixed point is the smallest pre-fixed point and the greatest fixed point is the largest post-fixed point.

Knaster-Tarski Fixed Point Theorem. Let `(D, ⊑)` be a complete lattice and let `f: D -> D` be a monotone function.

Then the least fixed point of `f`, `fix(f)` for short, is given by

`fix(f) = inf { d ∈ D | f(d) ⊑ d }`.

Furthermore, the greatest fixed point of `f`, `FIX(f)` for short, is given by

`FIX(f) = sup { d ∈ D | d ⊑ f(d) }`.

For example, for the identify function `f(x) ::= x`, the least fixed point is

```
fix(f) = inf { d ∈ D | f(d) ⊑ d } = inf { d ∈ D } = bot
```

Fixed points of continuous functions

The Tarski-Knaster theorem gives us a theoretical characterization of least and greatest fixed points. However, it does not explain how we could try to compute them, for example by applying `f` repeatedly until we reach a fixed point. For such a computational characterization, we need a stronger notion than monotonicity, which admits swapping function application and taking the supremum:

A function `f: D -> D` is *continuous* iff for every subset `S ⊆ D`, we have

```
sup { f(d) | d in S } = f( sup S )
```

In particular, every continuous function is also monotonous. Hence, it has a least and greatest fixed point (which may be identical).

Lemma. For every PL0 program `S`, the function `WP(S, .): Pred -> Pred` is continuous.

For continuous functions, Kleene's fixed point theorem allows us to characterize the least and greatest fixed point through iterative application:

Kleene fixed point theorem. Let `(D, <=)` be a complete lattice. Moreover, let `f: D -> D` be a continuous function. Then:

```
fix(f) ::= sup { f^n(bot) | n in Nat }
```

and

```
FIX(f) ::= inf { f^n(top) | n in Nat },
```

where `f^n` denotes the n-fold application of `f`, that is,

`f^0(d) ::= d` and `f^{n+1}(d) ::= f^{n}(f(d))`.

Notice that Kleene's fixed point theorem does *not* mean that we will reach the least or greatest fixed point after finitely many function applications. It is possible, that we might actually have to apply `f` infinitely often and only reach the fixed point in the limit.

This is different from fixed point theorems you might have encountered in program analysis or other courses, where additional constraints (e.g. finite height) guarantee that one reaches the fix point after finitely many steps.

For example, consider the function `f` given by `f(x) ::= x+1` over the complete lattice $(\mathbb{N} \cup \{\infty\}, \leq)$. We can iteratively apply `f` to the least element `bot` to approximate the least fixed point:

```
f^0(bot) = bot = 0
f^1(bot) = f(0) = 0 + 1 = 1
f^2(bot) = f(f(0)) = f(1) = 1 + 1 = 2
f^3(bot) = f(f(f(0)) = f(2) = 3
...
f^n(bot) = f(f^{n-1}(bot)) = f(n-1) = n
```

We can continue apply `f` forever. After each application, we get slightly closer to the function's unique fixed point, ∞, but we never reach it.