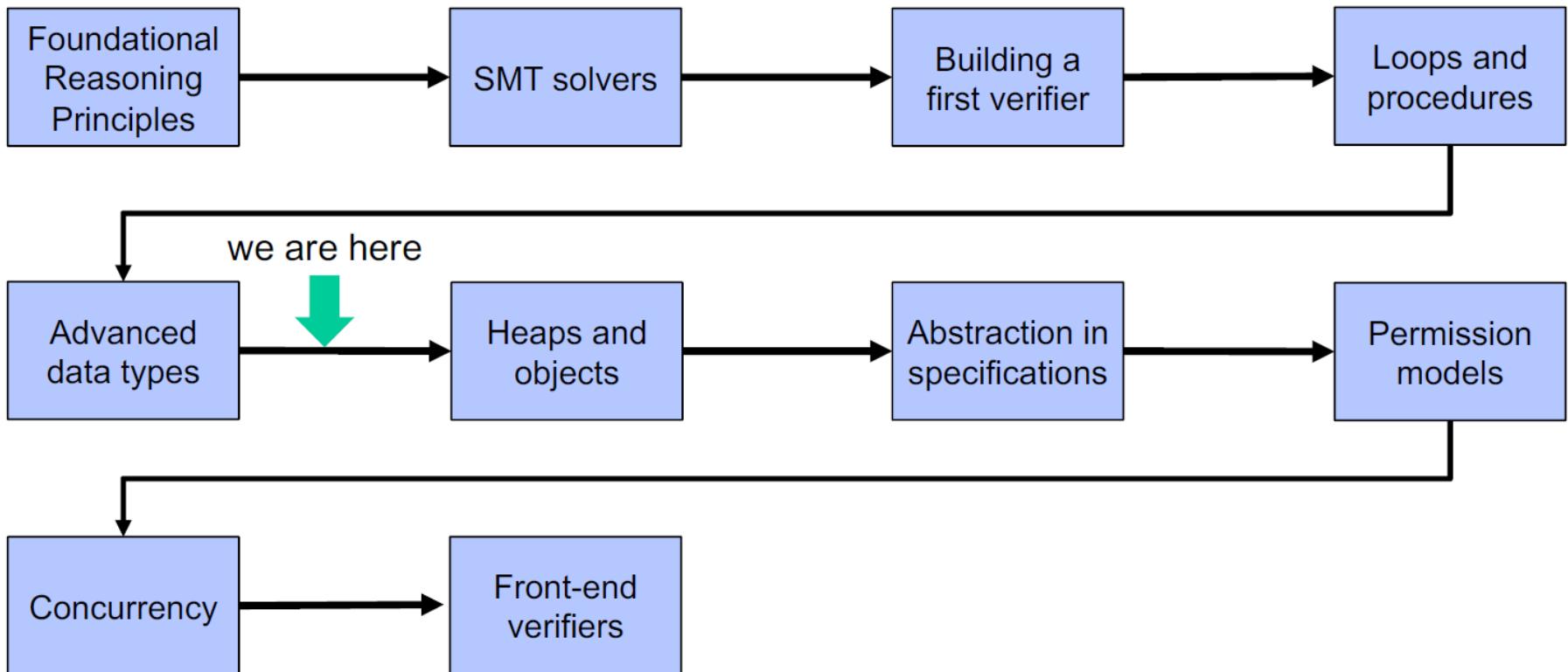


# **Program Analysis for Software Security**

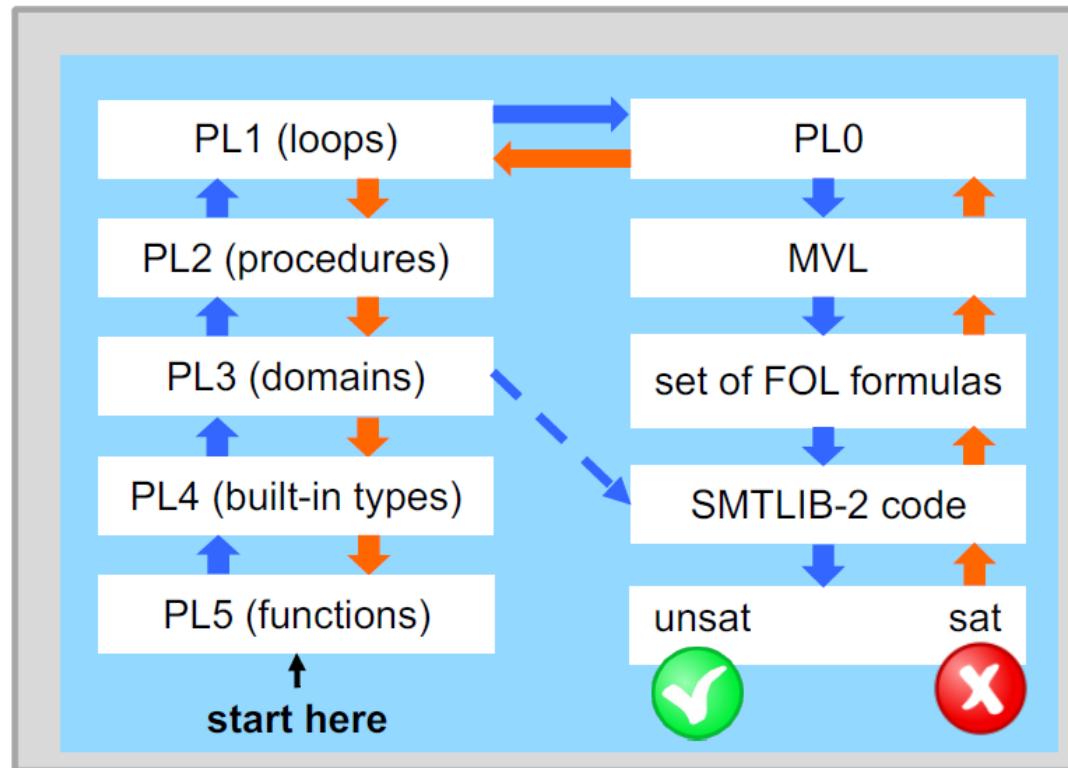
## **Lecture 9**

# **VERIFICATION TACTICS**

# Tentative course outline

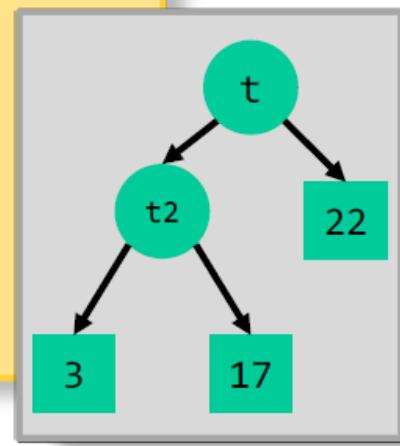


# The language PL5



## Example – summing values in a binary tree

```
method client() {  
    var t: Tree := node(  
        node(leaf(3), leaf(17)),  
        leaf(22)  
    )  
    assert sum(t) == 42  
}  
  
function sum(t: Tree): Int  
  
domain Tree {  
    // ...  
}
```



	Approach	
1	implement function	
2	abstract function with postcondition	
3	definitional axiom using Tree functions	
4	manually written definitional axioms	
5	implement function + assertion in client	

- All approaches are logically equivalent
- Most theories with quantifiers are undecidable
- ➔ To effectively use automated verifiers, we need to understand how tools deal with **quantifiers**

# Outline: verification tactics

- Excursion: quantifiers
- Lemmas & proofs
- Hands-on program verification

# Universal quantifier instantiation

- **Our problem:** Is the **FO** formula  $F$  **unsatisfiable**?
  - equivalent: is  $\neg F$  **satisfiable**?
- To prove **forall**  $x :: G$  **unsat**, we can try out all possible **candidate values** until we find *one* value  $v$  such that  $G[x/v]$  becomes **unsat**
- **Issue 1:** How do we choose *good* candidates?
  - most values may be irrelevant for our VCs
- **Issue 2:** When do we give up trying more values?
  - Logics with quantifiers are often undecidable
  - Better to quickly report that we cannot verify a problem than trying out values indefinitely

**Verification condition:**

$BP \And \neg WP(S, \text{true})$

**unsat**

**forall**  $x :: G$

**unsat**

iff  $\neg (\text{forall } x :: G)$

**sat**

iff **exists**  $x :: \neg G$

**sat**

iff for some value  $v$ ,  $\neg G[x/v]$

**sat**

iff for some value  $v$ ,  $G[x/v]$

**unsat**

**forall**  $x :: G$

$\Leftrightarrow$

$G[x/v_1] \And \dots \And G[x/v_n] \And$   
 $\And \text{forall } x :: G$

# Universal quantifier instantiation – approaches

- Due to undecidability, all approaches are incomplete
  - May return **unknown** or not terminate
- Model-based quantifier instantiation (MBQI)
  - Focuses on proving **satisfiability**
  - Possibly returns **unknown** instead of **unsat**
  - ➔ Not well-suited for our verification problem
- Heuristic quantifier instantiation with E-matching
  - Focuses on proving **unsatisfiability**
  - Possibly returns **unknown** instead of **sat**
  - We may not get a counterexample if verification fails
  - ➔ Most common approach used by verification tools

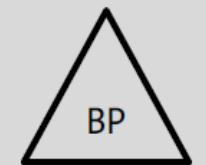
**Verification condition:**  
BP && !WP(S, true) **unsat**

# Heuristic quantifier instantiation

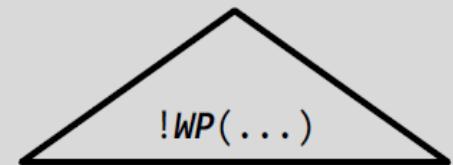
- Main idea: try out a subset  $V$  of all values
  - return **unsat** if  $G[x/v]$  is **unsat** for some  $v$  in  $V$
  - return **unknown** if  $G[x/v]$  is **sat** for all  $v$  in  $V$
- Hypothesis:  $V$  should contain...
  - all **ground terms**
    - terms without quantifier-bound variables
    - “expressions used in program or specification”
    - E.g.,  $0$ ,  $1+2$ ,  $x+2$  (where  $x$  is a free variable)
  - function applications to ground terms
    - “unfolding of function calls”
    - E.g.,  $\text{fib}(\text{fib}(1))$

```
forall x :: x in V ==> G unsat  
iff for some v in V, G[x/v] unsat  
implies forall x :: G unsat
```

## Structure of our VCs



few ground terms



many ground terms

&&

# Heuristic quantifier instantiation loop (for one quantifier)

**Input:** FO formula  $F$  && **forall**  $x :: \underline{G}$

**Output:** **unsat** or **unknown**

**Algorithm:**

$F(0) := F$

$F(0) := h(0) == 1$

for  $i = 0, 1, 2, \dots$

(Choose) pick a ground term  $t$  in  $F(i)$   
or  $G$  such that  $F(i)$  does *not* contain a  
conjunct equal to  $G[x/t]$ ; return  
**unknown** if no such  $t$  exists

(Instantiate)  $F(i+1) := G[x/t] \&& F(i)$

(Check) If  $F(i+1)$  is **unsat**, then  
return **unsat**

$h(0) == 1 \&&$   
**forall**  $x :: \underline{h(x) == 1+h(x-1)} \&& h(x) < 3$

**i = 0:** choose  $t = 1$   
 $G[x/t] = 1 + h(0) \&& h(1) < 3$   
Instantiate:  $F(1) := G[x/t] \&& F(0)$   
Check:  $F(1)$  is **sat** → continue

**i = 1:** choose  $t = 1 + h(0) = 2$   
 $G[x/t] = h(2) == 1+h(2-1) \&& h(2) < 3$   
Instantiate:  $F(2) := G[x/t] \&& F(1)$   
 $= h(2) == 1+h(2-1) \&& h(2) < 3$   
 $\&& h(1) == 1 + h(0) \&& h(1) < 3$   
 $\&& h(0) == 1$   
 $F(2)$  **unsat** → return **unsat**

# E-matching

- Problems with heuristic
  - Formulas may have exponentially many ground terms
  - Function applications admit infinitely many ground terms  
→ Let user determine relevant ground terms
- A **pattern** (or trigger) is a term **p** such that
  - **p** contains all bound variables in the scope of the quantifier
  - **p** contains at least one non-constant uninterpreted function
  - **p** contains at most constant interpreted function
- Consider only ground terms **t** that **e-match** pattern **p**, that is, we can find some ground term **t'** **provably equal** to **p[x / t]**

Predicates

$P ::= \dots \mid \text{forall } x:T :: \{ p \} P$

```
x == f(7) && g(x) == 3 &&
forall y: Int :: 
  { g(f(y)) } g(f(y)) > 5
```

How can we instantiate the above?

# E-matching

- Problems with heuristic
  - Formulas may have exponentially many ground terms
  - Function applications admit infinitely many ground terms  
→ Let user determine relevant ground terms
- A pattern (or trigger) is a term  $p$  such that
  - $p$  contains all bound variables in the scope of the quantifier
  - $p$  contains at least one non-constant uninterpreted function
  - $p$  contains at most constant interpreted function
- Consider only ground terms  $t$  that e-match pattern  $p$ , that is, we can find some ground term  $t'$  provably equal to  $p[x / t]$

$p[x/t] = g(f(7)) = \underline{3} \leftarrow t' \text{ appears above}$   
→ instantiate  $g(f(\underline{7})) > 5$

## Predicates

$P ::= \dots | \text{forall } x:T :: \{ p \} P$

```
x == f(7) && g(x) == 3 &&
forall y: Int :: 
  { g(f(y)) } g(f(y)) > 5
```

How can we instantiate the above?

## Example

```
f(0) != f(1) && g(1) == 0 && f(0) == 1  
&& forall x: Int ::{ p } f(x) == f(g(x))
```

- Patterns are typically terms in the quantified formulas body, e.g.  $p = g(x)$ 
  - e-match 1:  $g(1) == 0$  and 0 is a ground term
  - instantiating  $f(1) == f(g(1))$  makes the whole formula unsatisfiable → return **unsat**
- Too restrictive patterns may often yield unknown, e.g.  $p = g(g(x))$ 
  - No e-matching possible → return **unknown**
- Too permissive patterns may lead to **matching loops**, e.g.  $p = f(x)$ 
  - e-match 0, instantiate  $f(0) == f(g(0))$
  - e-match  $g(0)$ , instantiate  $f(g(0)) == f(g(g(0)))$ , e-match  $g(g(0)) \dots$

# Reasoning about recursive functions

→ 07-factorial.vpr

- Problem: Recursive functions can always be unfolded to instantiate new ground terms
- There is no natural condition for stopping the unfolding, even if the recursive function terminates
- Consequences:
  - Recursive functions lead to matching loops
  - SMT solver may never terminate
- Solution: **limit** the unfolding depth

```
function fac(x: Int): Int  
{ x <= 1 ? 1 : x * fac(x-1) }
```

```
var n: Int; assert fac(n) != 0
```

```
function fac(x: Int): Int  
axiom forall x: Int ::  
    fac(x) == (x <= 1 ? 1 : x * fac(x-1))
```

```
fac(0) == 1 && fac(n) != 0
```

```
fac(1)==1 && fac(0)==1 && fac(n)!=0
```

```
fac(fac(1)) == 1 && fac(1) == 1 && ...
```

```
fac(fac(fac(1))) == 1 && ...
```



# Limited functions

- Goal: encode recursive functions such that they can be unfolded only a limited number of times
- Idea: to stop unfolding, call a different function without a definitional axiom

```
function fac(x: Int): Int
{ x <= 1 ? 1 : x * fac(x-1) }
```

```
var n: Int; assert fac(n) != 0
```



→ 08-factorial.vpr

```
function fac(x: Int): Int
function fac0(x: Int): Int
axiom forall x: Int ::  

  (x <= 1 ==> fac(x) == 1) &&  

  (x > 1 ==> fac(x) == x * fac0(x-1))
```

```
fac(0) == 1 && fac(n) != 0
```

```
fac(1)==1 && fac(0)==1 && fac(n)!=0
```

```
fac(fac(1)) == fac(1) * fac0(fac(1)-1)  

&& fac(1)==1 && fac(0)==1 && fac(n)!=0
```

Since `fac0` is not constrained by axioms, the SMT solver can choose a function for `fac0` such that the formula becomes `unsat`

# Improving limited functions

→ 09-factorial.vpr

- Since limited functions bound the number of unfoldings, the solver cannot find proofs that require more unfoldings
- Can we combine facts to proofs that would require multiple unfoldings?

```
fac(1) == 1 && fac(2) == 2 * fac(2-1)
→ fac(1) == 1 && fac(2) == 2 * fac(2-1)
→ fac(1) == 1 && fac(2) == 2 * 1
→ fac(1) == 1 && fac(2) == 2
```

```
function fac(x: Int): Int
{ x <= 1 ? 1 : x * fac(x-1) }
```

```
assert fac(1) == 1 // one unfolding
```



```
assert fac(2) == 2 // two unfoldings
```



```
assert fac(1) == 1
assert fac(2) == 2
// provable with one unfolding each
```



# Improving limited functions

- Since limited functions bound the number of unfoldings, the solver cannot find proofs that require more unfoldings
- Can we combine facts to proofs that would require multiple unfoldings?

```
fac(1) == 1 && fac(2) == 2 * fac0(2-1)
→ fac(1) == 1 && fac(2) == 2 * fac(2-1)
→ fac(1) == 1 && fac(2) == 2 * 1
→ fac(1) == 1 && fac(2) == 2
```

```
function fac(x: Int): Int
{ x <= 1 ? 1 : x * fac(x-1) }
```

```
assert fac(1) == 1 // one unfolding
```



```
assert fac(2) == 2 // two unfoldings
```



```
assert fac(1) == 1
assert fac(2) == 2
// provable with one unfolding each
```



- Idea: axiomatize  $\text{fac}(x) = \text{fac0}(x)$ 
  - Problem: may reintroduce matching loop

```
axiom {
  forall x: Int :: 
    fac(x) == fac0(x)
}
```

# Improving limited functions

→ 10-factorial.vpr

- Since limited functions bound the number of unfoldings, the solver cannot find proofs that require more unfoldings
- Can we combine facts to proofs that would require multiple unfoldings?

```
fac(1) == 1 && fac(2) == 2 * fac0(2-1)
→ fac(1) == 1 && fac(2) == 2 * fac(2-1)
→ fac(1) == 1 && fac(2) == 2 * 1
→ fac(1) == 1 && fac(2) == 2
```

- Idea: axiomatize  $\text{fac}(x) == \text{fac0}(x)$ 
  - Problem: may reintroduce matching loop
  - Solution: trigger axiom only for function  $\text{fac}(x)$

```
function fac(x: Int): Int
{ x <= 1 ? 1 : x * fac(x-1) }
```

```
assert fac(1) == 1 // one unfolding
```



```
assert fac(2) == 2 // two unfoldings
```



```
assert fac(1) == 1
assert fac(2) == 2
// provable with one unfolding each
```



```
axiom {
  forall x: Int :: { fac(x) }
  fac(x) == fac0(x)
}
```

## Working with limited functions

- To work around unfolding limits, it is often sufficient to mention a ground term that is required for the proof

→ 11-factorial.vpr

```
function fac(x: Int): Int
{ x <= 1 ? 1 : x * fac(x-1) }
```

```
assert fac(2) == 2
```



```
var n: Int := fac(1)
// ground term fac(1) is available
assert fac(2) == 2
```



# Existential quantifier instantiation

→ 12-exists.vpr

- Our problem: Is the FO formula F **unsatisfiable**?
  - equivalent: is  $\neg F$  **satisfiable**?
- To prove **exists**  $x :: G$  **unsat**, we have to show that  $G[x/v]$  becomes **unsat** for all values  $v$
- When aiming to prove **unsatisfiability**, SMT solvers often struggle with existentials

```
assert exists x: Int :: x == 0
```



- Try to avoid existential quantifiers in specifications
- If needed, manually instantiate or introduce existential quantifiers → **user-defined lemmas**

Verification condition:

BP &&  $\neg \text{WP}(S, \text{true})$

unsat

<b>exists</b> $x :: G$	unsat
iff $\neg(\exists x :: G)$	sat
iff <b>forall</b> $x :: \neg G$	sat
iff for all values $v$ , $\neg G[x/v]$	sat
iff for all values $v$ , $G[x/v]$	unsat

# Outline

- Excursion: quantifiers
- Lemmas & proofs
- Hands-on program verification

# Lemmas – in mathematics

- A lemma consists of
  - a premise determining whether the lemma can be used
  - a conclusion stating what property is guaranteed
  - a proof checking that the conclusion indeed always follows from the premise
- To apply a lemma, we **check its premises** and, if yes, can **use its conclusion**
- Lemmas are “subroutines of a larger proof”

**Lemma 1.**

**Premise:**  $n \geq 0$

**Conclusion:**  $\text{fac}(n) > 0$

**Proof:** by induction on  $n$ .

**Theorem.** For all  $x > 0$ ,  
 $\text{fac}(x) + \text{fac}(x) + \text{fac}(x) > 2$ .

**Proof.**

Let  $x > 0$ . Then, since  $x \geq 0$ ,  
Lemma 1 yields  $\text{fac}(x) > 0$ .

Hence,

$\text{fac}(x) + \text{fac}(x) + \text{fac}(x) > 2$ .

# Why do we need lemmas for program verification?

```
function fac(x: Int): Int {  
    x <= 1 ? 1 : x * fac(x-1)  
}  
  
method client(x: Int)  
    returns (y: Int)  
    requires x > 0  
    ensures y > 2  
{  
    var z: Int := fac(x)  
  
    y := z + z + z  
}
```



- SMT solver does not notice that  $\text{fac}(x) > 0$ 
  - No automatic proofs by induction
  - Could be added as postcondition to  $\text{fac}(x)$
- We may not want to add all needed properties as axioms (of functions)
  - Some properties might be specialized and are only useful in very specific cases
  - Many axioms might slow down proof generation

→ 13-factorial-positive.vpr

# Lemmas – as ghost methods

```
method lemma(<arguments>)
  requires Premise
  ensures Conclusion
{
  Proof
}
```

```
lemma(x,y)
assert Conclusion(x,y)
```



- Lemmas are ghost methods
  - They may not affect program execution
  - They can be removed from production code
- Method body represents a correctness proof
  - Abstract methods are trusted (unproven)
- By [invoking a lemma](#), we learn its postcondition only for the supplied the arguments

# Using a lemma in Viper

→ 14-factorial-lemma.vpr

```
function fac(x: Int): Int {  
    x <= 1 ? 1 : x * fac(x-1)  
}  
  
method client(x: Int)  
    returns (y: Int)  
    requires x > 0  
    ensures y > 2  
{  
    var z: Int := fac(x)  
    lemma_fac_pos(x)  
    y := z + z + z  
}
```

we now know  $\text{fac}(x) > 0$

we do *not* know  $\text{fac}(z) > 0$



```
method lemma_fac_pos(n: Int)  
    requires n >= 0  
    ensures fac(n) > 0
```

- By invoking a lemma, we learn its postcondition only for the supplied arguments
- To use a lemma, we just call the method
  - Checks that premise holds for supplied arguments
  - Guarantees that conclusion holds afterward

# Proving lemmas by implementing ghost methods

Statement	Meaning in proofs
<code>x := e</code>	Name an expression
<code>assert P</code>	Make a correct statement (possibly to introduce ground terms)
<code>assume P</code>	Make a (possibly wrong) assumption
<code>if (b) {S1} else {S2}</code>	Case distinction on b
method call	Invoke another lemma
recursive method call (for proofs by induction)	Invoke the induction hypothesis given by the lemma's contract

```
method lemma_fac_pos(n: Int)
  requires n >= 0
  ensures fac(n) > 0
  // decreases n // variant
{
  var v: Int := n; assert v >= 0
  // proof by induction on n
  if (n == 0) { // base case
    assert fac(0) > 0
  } else { // induction step
    assert n-1 >= 0
    // invoke I.H.
    assert n-1 < v
    lemma_fac_pos(n-1)
    assert fac(n-1) > 0
  }
}
```



→ 15-lemma-proof.vpr

# Why we need termination proofs for lemmas

**Claim:** for all integers n, the following triple is valid:

```
{ n >= 0 }  
lemma_fac_pos(n)  
{ fac(n) > 0 }
```

**Induction hypothesis**

(I.H.): for all  $m < n$ , the triple

```
{ m >= 0 }  
lemma_fac_pos(m)  
{ fac(m) > 0 }
```

is valid.

```
method lemma_fac_pos(n: Int)  
  requires n >= 0  
  ensures fac(n) > 0  
{
```

```
  if (n == 0) {
```

induction base: claim holds for  $n == 0$

```
  } else {
```

illegal use of I.H.: argument is not  $< n$

```
    lemma_fac_pos(n)  
  }
```

unsound proof!



→ 15-lemma-proof.vpr

# Why we need termination proofs for lemmas

**Claim:** for all integers n, the following triple is valid:

```
{ n >= 0 }
lemma_fac_pos(n)
{ fac(n) > 0 }
```

**Induction hypothesis**

(I.H.): for all m < n, the triple

```
{ m >= 0 }
lemma_fac_pos(m)
{ fac(m) > 0 }
```

is valid.

```
method lemma_fac_pos(n: Int)
  requires n >= 0
  ensures fac(n) > 0
{
  var v: Int := n; assert v >= 0
  if (n == 0) {
    induction base: claim holds for n == 0
  } else {
    illegal use of I.H.: argument is not < n
    lemma_fac_pos(n)
    assert n < v
  }
}
```

- For terminating proof programs, calls always decrease some variant  
→ only valid proofs verify

variant does not decrease  
→ unsound proof fails

→ 15-lemma-proof.vpr

# Lemmas for existential quantifiers

→ 18-exists-lemmas.vpr

- Sometimes the solver may be unable to deal with parts of a predicate
  - Existential quantifiers, very complex predicates
- We can "hide" such predicates in a boolean function with no definitional axiom
  - The solver has nothing to unfold
  - Function calls are kept around
- We can then introduce abstract lemmas to
  - *unfold* the predicate: make its definition visible, possibly with concrete values for existentials
  - *fold* the predicate: store its definition behind a call, possibly abstracting concrete values

```
function divides(x:Int, y: Int): Bool
  requires x >= 0 && y >= 0
  //ensures result == exists z:Int :: 
  //                                z >= 0 && x * z == y
```

```
method divides_unfold(x: Int, y: Int)
  returns (z: Int)
  requires x > 0 && y > 0
  requires divides(x, y)
  ensures z >= 0 && x * z == y
```

```
method divides_fold(x: Int,
                     y: Int, z: Int)
  requires x > 0 && y > 0
  requires z >= 0 && x * z == y
  ensures divides(x, y)
```