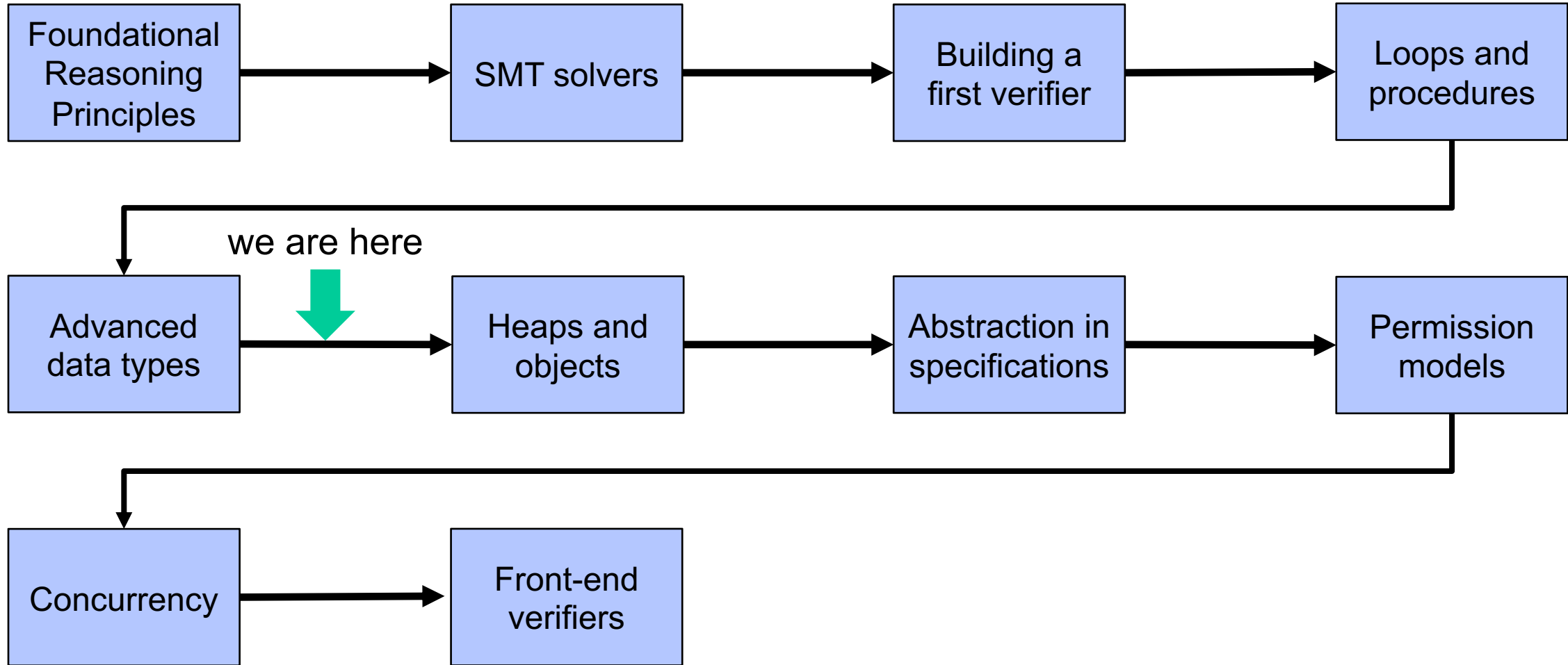


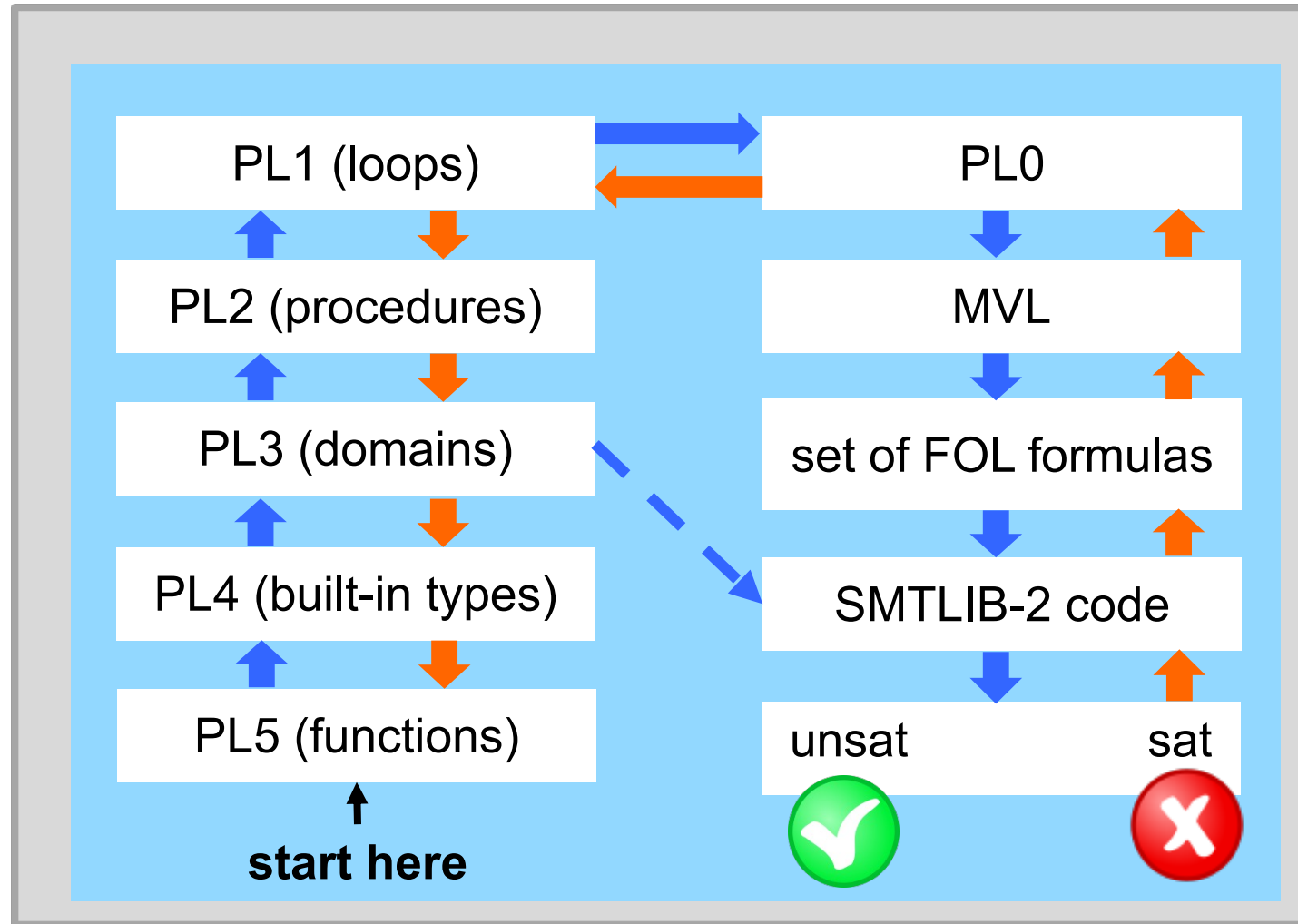
02245 – Module 6

VERIFICATION TACTICS

Tentative course outline

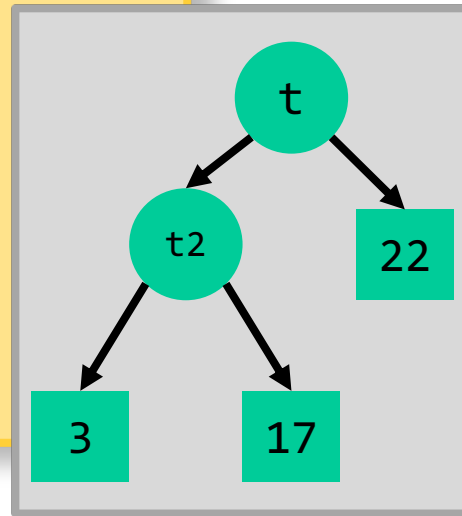


The language PL5



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 {  
  // ...  
}
```



Previous exercise: how to define sum?

Try out variants: 0X-tree-sum.vpr

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 $\text{forall } x :: G$ **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:

$BP \ \&\& \ \neg WP(S, \text{true})$ **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
<==>
G[x/v1] && ... && G[x/vn] &&
  && 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:

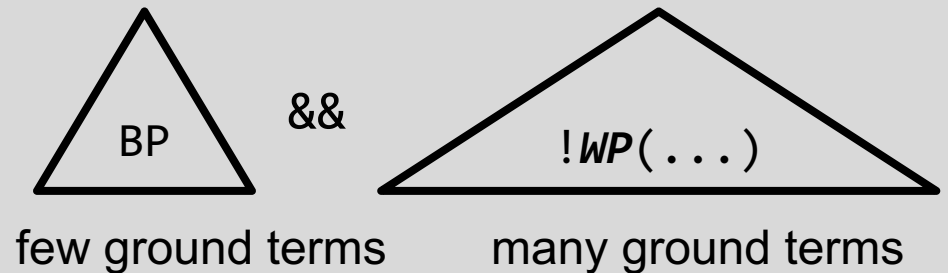
BP && ! $WP(S, true)$ **unsat**

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., \emptyset , $1+2$, $x+2$ (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



- Asserting ground terms can improve quantifier instantiation → [05-tree-sum.vpr](#)

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** \rightarrow 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** \rightarrow 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 a ground term that is **provably equal** to $p[x / t]$

Predicates

```
P ::= ... | 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 $g(g(0)) \dots$
- Viper automatically selects (possibly suboptimal) patterns → **0X-tree-sum.vpr**

Exercise

- Consider axiomatization of 2D points on the right. We added a **function** and an **axiom** for adding two points by adding their components.
- Try out different triggering patterns for the **axiom** on the right and test them for client below. Find patterns such that
 - a) verification succeeds,
 - b) verification fails, and
 - c) verification does not terminate.

```
// file: examples/06-trigger-point.vpr
domain Point {
function cons(x: Int, y: Int): Point
  function first(p: Point): Int
  function second(p: Point): Int

  function add(p: Point, q: Point): Point

  axiom {
    forall p: Point, q: Point ::
      first(add(p,q)) == first(p) + first(q)
      && second(add(p,q)) == second(p) + second(q)
  }
  // ...
}
```

```
method client() {
  var x: Point := add( cons(17, 42), cons(3,8) )
  assert first(x) == 20
  assert second(x) == 50
}
```

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

→ 08-factorial.vpr

- 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
```



- Viper limits recursive functions to one unfolding → [tree-sum-1.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 * 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
```



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
```

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

```
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) == 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

→ 11-factorial.vpr


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

```
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 $\text{exists } x :: G$ **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 && !WP(S, true)
```

unsat

```
exists x :: G
```

unsat

```
iff !(exists x :: G)
```

sat

```
iff forall x :: !G
```

sat

```
iff for all values v, !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 > 3
{
  var z: Int := fac(x)

  y := z + z + z
}
```



→ 13-factorial-positive.vpr

- 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

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 > 3
{
  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 the 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

Exercise

- Use a lemma to verify the following client:

```
// file: 16-exercise.vpr
function foo(x: Int): Int {
  x <= 0 ? 1 : foo(x - 2) + 3
}

method client(r: Int) {
  var s: Int := foo(r)
  var t: Int := foo(s)

  assert 2 <= t - s
}
```

- Prove the following lemma (including termination):

```
// file: 17-commutativity.vpr
function X(n: Int, m: Int): Int
  requires n >= 0 && m >= 0 {
    m == 0 ? 0 : n + X(n, m-1)
  }

method lemma_X_commutative (n: Int, m: Int)
  requires n >= 0 && m >= 0
  ensures X(n, m) == X(m, n) {
    // TODO: show commutativity of
    //      multiplication function X
  }
```

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)
```

Outline

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

Remaining goal for today: verify programs 😊

Rules:

- There are four *verification challenges*, each with a code skeleton
 - You can work on them in groups and in any order
- Some challenges are *hard* → you can always ask for hints, help, or feedback
- Do not modify *executable* source code unless the task is to implement something
 - You can always add ghost code, assertions, and annotations as long as you can justify their soundness (so `assume false` is not allowed, even though it might be useful on the way...).

Challenge 1: Mirroring binary trees

- Skeleton file: `challenges/mirror-tree.vpr`
- Tasks:
 - a. Implement a function that mirrors binary trees, that is, swaps the left and right children of every node in a tree.
 - b. Prove that mirroring a tree does not change the tree's size.
 - c. Prove the provided client to show that mirroring an arbitrary tree twice yields the original tree.

Challenge 2: Insertion sort

- Skeleton file: `challenges/insertion-sort.vpr`
- The skeleton implements a recursive variant of insertion sort
- Tasks:
 - a. Implement the function `sorted` stating that a sequence is sorted in ascending order.
 - b. Prove the given lemma to check if your implementation is sensible.
 - c. Prove (wrt. partial correctness) that the sorting method returns a sorted sequence.
 - d. Prove (wrt. partial correctness) that the sorting method sorts the input sequence.
 - e. Prove that the sorting method terminates.

Challenge 3: Euclid's algorithm

- Skeleton file: `challenges/euclid.vpr`
- Euclid's algorithm is a well-known iterative technique for computing the greatest common divisor (GCD) of two positive integers.
- Tasks:
 - a. Define a function that returns the GCD of two positive integers.
 - b. Prove that Euclid's algorithm indeed returns the GCD of its two arguments.
 - c. Prove that Euclid's algorithm terminates.

Challenge 4: Determining the (clear) winner of an election

- Skeleton file: `challenges/election.vpr`
- The skeleton implements a method that takes a list of votes and attempts to return the candidate who received an absolute majority of votes (if one exists).
 - The algorithm is quite neat, since it runs in linear time.
- Tasks:
 - Verify (wrt. partial correctness) that the search method returns the winner if there is one.
 - Note: we use ghost variables to indicate who should be the winner
 - Show that the search method terminates.

< Discussion >

What next

