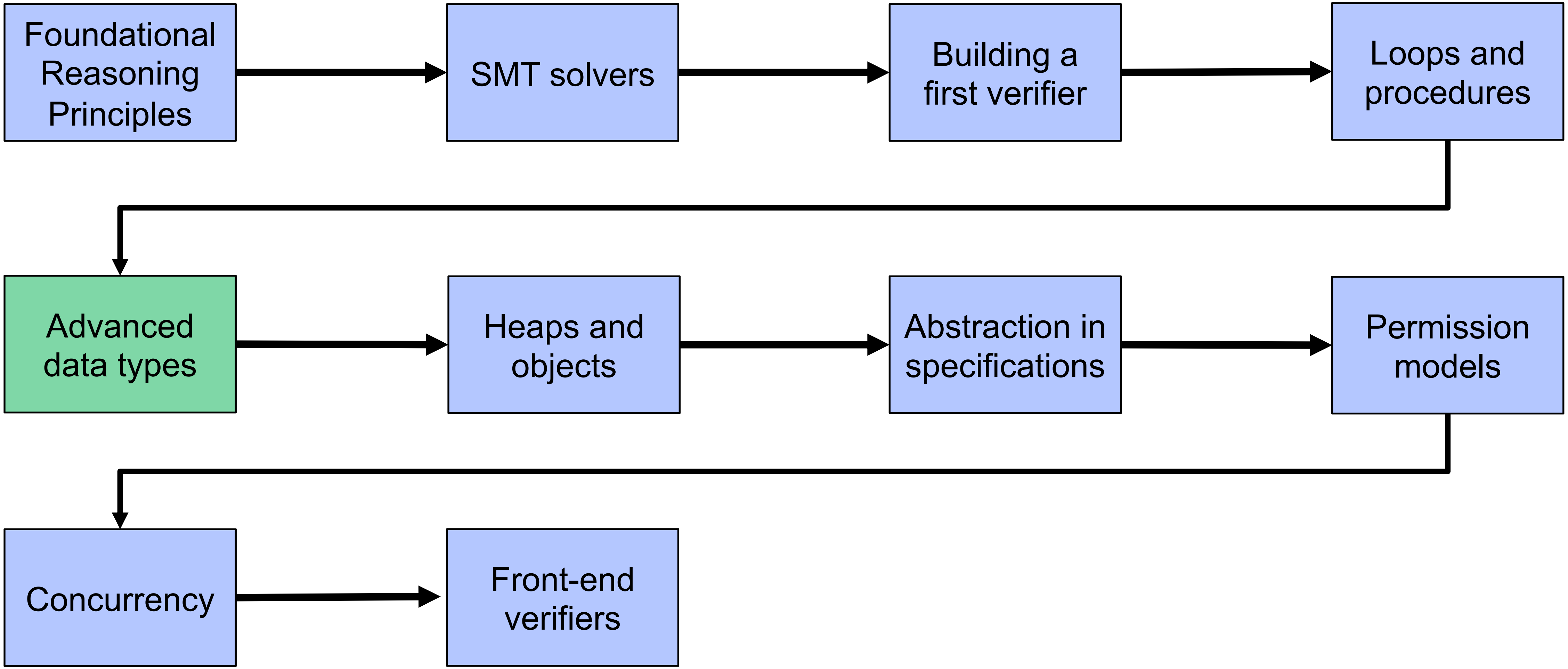
02245 – Module 5

ADVANCED DATATYPES

Tentative course outline



Outline

- Mathematical data types
- User-defined functions
- Function encoding

Mathematical data types

- Our language so far supports only three types

```
Types
T ::= Bool | Int | Rational
```

- Many functional languages feature mathematical data types
  - lists, tuples, sets, trees, etc.

- Subset of abstract data types (ADTs)
  - What are values of a type?
  - What are operations on data of a type?
  - immutable, no side-effects
  - ➔ *"programming & specification vocabulary"*

```
domain Set {
  function empty(): Set
  function add(s: Set, x: Int): Set
  function contains(s: Set, x: Int): Bool
  function union(s: Set, t: Set): Set
  function is_empty(s: Set): Bool
}
```

- Mathematical data types are for specifying imperative code ➔ module 8
  - *"Array sort leaves the multiset of elements unchanged"*
  - *"All implementations of Java's List interface store a sequence of elements"*

Common mathematical data types (PL4)

- We extend our language to support commonly-used data types

- The built-in data types
  - are generic
  - represent immutable, mathematical values
  - represent finite collections
  - are available in Viper

- We use Viper's expression syntax
  - See tutorial for other data types
  - https://viper.ethz.ch/tutorial

Types

```
T ::= Bool | Int | Rational | Set[T]
    | Seq[T] | Multiset[T] | Map[T, T]
```

Expressions

```
e ::= ...                      as before
    | Set[T]()                 empty set
    | Set(ē)                   set literal
    | e union e
    | e intersection e
    | e setminus e
    | e subset e
    | e in e                   membership
    | |e|                      cardinality
```

Example

```
method collect(s: Seq[Int]) returns (res: Set[Int])
  ensures forall j: Int :: 0 <= j && j < |s| ==> s[j] in res
  ensures forall x: Int :: x in res ==> x in s
{
  res := Set[Int]()
  var i: Int := 0
  while (i < |s|)
    invariant 0 <= i && i <= |s|
    invariant forall j: Int :: 0 <= j && j < i ==> s[j] in res
    invariant forall x: Int :: x in res ==> x in s
  {
    res := res union Set(s[i])
    i := i + 1
  }
}
```

Set operations

Sequence operations

Custom data types (PL3)

Declarations

```
D ::= ...                                        as before

    | domain <name> {                      define new type
        (function <name>(x:T): T)*         define function
        (axiom <name> { P })*               define axiom
      }
```

Types

```
T ::= Bool | Int | Rational
    | <name>          defined types
```

Expressions

```
e ::= ...                 as before
    | <name>(ē)      function call
```

```
domain Point {
  function cons(x: Int, y: Int): Point
  function first(p: Point): Int
  function second(p: Point): Int

  axiom destruct_over_construct {
      forall x: Int, y: Int ::
        first(cons(x,y)) == x && second(cons(x,y)) == y
   }
}
```

- Every domain declares a new type and associated functions
- Corresponds to a axiomatizing a new theory

Example: binary trees with values at leafs

```
// Java-like code
interface Tree {
 Tree leaf(int value);
 Tree node(Tree left, Tree right);

 bool is_leaf();
 Tree left();
 Tree right();
 int value();
}
```

```
var t: Tree := node(
  node(leaf(3), leaf(17)),
  leaf(22)
)
assert !is_leaf(t)
var t2: Tree := right(left(t))
assert value(t2) == 17
```

```
domain Tree {

  function leaf(value: Int): Tree
  function node(left: Tree, right: Tree): Tree

  function is_leaf(t: Tree): Bool
  function value(t: Tree): Int
  function left(t: Tree): Tree
  function right(t: Tree): Tree

  axiom value_over_leaf {
    forall x:Int :: value(leaf(x)) == x
  }

  axiom right_over_node {
    forall l:Tree, r:Tree :: right(node(l, r)) == r
  }

  // ... (see 02-tree.vpr)
}
```

Example: binary trees with values at leafs

```
// Java-like code
interface Tree {
 Tree leaf(int value);
 Tree node(Tree left, Tree right);

 bool is_leaf();
 Tree left();
 Tree right();
 int value();
}
```

```
var t: Tree := node(
  node(leaf(3), leaf(17)),
  leaf(22)
)
assert !is_leaf(t)
var t2: Tree := right(left(t))
assert value(t2) == 17
```

constructors

```
domain Tree {
  function leaf(value: Int): Tree
  function node(left: Tree, right: Tree): Tree

  function is_leaf(t: Tree): Bool

           value(t: Tree): Int
           left(t: Tree): Tree
           right(t: Tree): Tree

  axiom value_over_leaf {
    forall x:Int :: value(leaf(x)) == x
  }

  axiom right_over_node {
    forall l:Tree, r:Tree :: right(node(l, r)) == r
  }

  // ... (see 02-tree.vpr)
}
```

Example: binary trees with values at leafs

```
// Java-like code
interface Tree {
 Tree leaf(int value);
 Tree node(Tree left, Tree right);

 bool is_leaf();
 Tree left();
 Tree right();
 int value();
}
```

```
var t: Tree := node(
  node(leaf(3), leaf(17)),
  leaf(22)
)
assert !is_leaf(t)
var t2: Tree := right(left(t))
assert value(t2) == 17
```

```
domain Tree {
  function leaf(value: Int): Tree
  function node(left: Tree, right: Tree): Tree

  function is_leaf(t: Tree): Bool
  function value(t: Tree): Int
           left(t: Tree): Tree
           right(t: Tree): Tree

           ue_over_leaf {
           x:Int :: value(leaf(x)) == x
  }

  axiom right_over_node {
    forall l:Tree, r:Tree :: right(node(l, r)) == r
  }

  // ... (see 02-tree.vpr)
}
```

discriminators

Example: binary trees with values at leafs

```
// Java-like code
interface Tree {
 Tree leaf(int value);
 Tree node(Tree left, Tree right);

 bool is_leaf();
 Tree left();
 Tree right();
 int value();
}
```

```
var t: Tree := node(
  node(leaf(3), leaf(17)
  leaf(22)
)
assert !is_leaf(t)
var t2: Tree := right(left(t))
assert value(t2) == 17
```

```
domain Tree {

  function leaf(value: Int): Tree
  function node(left: Tree, right: Tree): Tree

  function is_leaf(t: Tree): Bool
  function value(t: Tree): Int
  function left(t: Tree): Tree
  function right(t: Tree): Tree

  axiom value_over_leaf {
          x:Int :: value(leaf(x)) == x


        ght_over_node {
          l:Tree, r:Tree :: right(node(l, r)) == r
  }

  // ... (see 02-tree.vpr)
}
```

destructors

Example: binary trees with values at leafs

```
// Java-like code
interface Tree {
 Tree leaf(int value)
 Tree node(Tree left,

 bool is_leaf();
 Tree left();
 Tree right();
 int value();
}
```

Axioms

- Discriminators over constructors
- All trees are built from constructors
- Destructors over constructors

```
var t: Tree := node(
  node(leaf(3), leaf(17)),
  leaf(22)
)
assert !is_leaf(t)
var t2: Tree := right(left(t))
assert value(t2) == 17
```

```
 Tree
right: Tree): Tree

 Bool
nt
ee
  function right(t: Tree): Tree

  axiom value_over_leaf {
    forall x:Int :: value(leaf(x)) == x
  }

  axiom right_over_node {
    forall l:Tree, r:Tree :: right(node(l, r)) == r
  }

  // ... (see 02-tree.vpr)
}
```

Exercise

- The file *`03-trees.vpr`* axiomatizes binary trees with integer values stored in leafs.
- Extend the `Tree` domain by a function `size` that takes a Tree and returns the number of leafs in the tree.
- Extend the `Tree` domain by a function `sum` that takes a Tree and returns the sum of all values stored in the tree.
- Test your domain against the following client (also found in the file but commented out)

```
method client() {
    var t: Tree
    t := node(
            node(
              leaf(3),
              leaf(17)
            ),
            leaf(22)
         )

    assert sum(t) == 42
    assert size(t) == 3
}
```

Solution (see updated *03-trees.vpr*)

- The file *03-trees.vpr* axiomatizes binary trees with integer values stored in leafs.
- Extend the `Tree` domain by a function `size` that takes a Tree and returns the number of leafs in the tree.
- Extend the `Tree` domain by a function `sum` that takes a Tree and returns the sum of all values stored in the tree.
- Test your domain against the following client (also found in the file but commented out)

```
method client() {
    var t: Tree
    t := node(
            node(
              leaf(3),
              leaf(17)
            ),
            leaf(22)
         )

    assert sum(t) == 42
    assert size(t) == 3
}
```

Encoding of custom data types

- We encode custom data types into SMT by axiomatizing them
  - new type ➔ uninterpreted sort
  - new operation ➔ uninterpreted function
  - new axiom ➔ assert axiom (add to BP)

Background Predicate: conjunction of all axioms

Verification condition:

`BP ==> P ==> WP(S, Q)` valid

```
domain Set {
  function empty(): Set
  function card(s: Set): Int
  // ...

  axiom card_empty { card(empty()) == 0 }
  // ...
}
```

Conceptually, data types are encoded to PL0 as `assume BP`; the SMT language also needs declarations which are *not* in PL0.

```
(declare-sort Set)

(declare-const empty Set)
(declare-fun card (Set) Int)
; ...

(assert (= (card empty) 0)) ; axiom
; ...
```

Pragmatically, we can enrich PL0 by a statement for SMT declarations or “inline SMT code”

Encoding of built-in data types

- Built-in data types define domains with carefully crafted axioms and more convenient syntax
- Encoding: PL4 ➔ PL3

- Generics can be handled via monomorphization: generate a separate axiomatization for every instance of a generic type `T` that is used in a given program

Types

```
T ::= ...
    | Set[T]
```

Expressions

```
e ::= ...          as before
    | Set[T]()     empty set
    | |e|          cardinality
```

`T ::= Int`

```
domain IntSet {
  function empty(): IntSet
  function card(s: IntSet): Int
  // ...

  axiom card_empty { card(empty()) == 0 }
  // ...
}
```

Outline

- Mathematical data types
- User-defined functions
- Function encoding

Writing stronger specifications

- The built-in types and operators allow one to specify many interesting properties
- However, there are many methods whose behavior cannot be specified (easily)
- It is often useful to define additional mathematical vocabulary to specify the intended behavior

➔ Axiomatizations have a fixed pattern

➔ Use functional programs

```
method fac(n: Int) returns (res: Int)
  requires 0 <= n
  ensures  res == facDef(n)
{
  res := 1
  var i: Int := 1

  while(i <= n) {
    res := res * i
    i := i + 1
  }
}
```

```
domain X {
 function facDef(n: Int): Int

 axiom {
   forall n: Int ::
     (n <= 1 ==> facDef(n) == 1) &&
     (n > 1 ==> facDef(n)
                == n * facDef(n-1))
}}
```

User-defined functions (PL5)

- Functions abstract over expressions
  - can appear in specifications
  - can be recursive
  - can be uninterpreted (no definition)

- Model of mathematical functions
  - no side-effects
  - must always terminate (*not checked by Viper!*)
  - deterministic
  - well-defined for every input (total)

```
function facDef(n: Int): Int
{
  n <= 1 ? 1 : n * facDef(n-1)
}
```

Declarations

```
D ::= ...
    | function <name>(x̅:̅ ̅T̅): T
        (requires P)*
        (ensures Q)*
       ({ e })?
```

Expressions

```
e ::= ... | <name>(ē)
```

Reasoning about function calls

- Functions generally do not require a specification
  - Postconditions are typically equal the function definition

- We reason about calls by using the function definition

```
function facDef(n: Int): Int
{
  n <= 1 ? 1 : n * facDef(n-1)
}
```

- In contrast to methods, reasoning about function calls is not modular

```
x := facDef(1)
assert x == 1
```

- Non-modularity has drawbacks
  - All callers need to be re-verified when a function definition changes
  - But mathematical vocabulary is typically more stable

Partial functions

- Many operations are inherently partial functions
  - Meaningful only on a subset of the possible arguments
  - Example: division by zero

- Option 1: construct artificially total functions
  - Often leads to awkward function definitions
  - May cause misleading error messages

```
function facDef(n: Int): Int
{ n <= 1 ? 1 : n * facDef(n-1) }
```

```
x := facDef(-1)
```

✓

- Option 2: equip functions with preconditions
  - Needs to be checked for every function call
  - Also called “well-definedness conditions”
  - Supported by Viper

```
function facDef(n: Int): Int
  requires 0 <= n
{ n <= 1 ? 1 : n * facDef(n-1) }
```

```
x := facDef(-1)
```

✗

Exercise

Define a function fib(n) that yields the n^{th} Fibonacci number.

```
fib(0)   = 0
fib(1)   = 1
fib(n+2) = fib(n+1) + fib(n)
```

Provide a suitable precondition.

Verify that the method on the right computes the n^{th} Fibonacci number.

Hint: You can use the skeleton `07-fib.vpr`

```
method iter_fib(n: Int) returns (res: Int)
  requires 0 <= n
  ensures  …
{
  res := 0
  var i: Int := 0
  var next: Int := 1

  while (i < n)
    invariant …
  {
    var t: Int := res
    res := next
    next := t + next
    i := i + 1
  }
}
```

Solution

Define a function fib(n) that yields the n^th^ Fibonacci number.

```
fib(0)   = 0
fib(1)   = 1
fib(n+2) = fib(n+1) + fib(n)
```

Provide a suitable precondition.

Verify that the method on the right computes the n^th^ Fibonacci number.

Hint: You can use the skeleton `07-fib.vpr`

```
function fib(n: Int): Int
  requires 0 <= n
{ n < 2 ? n : fib(n-1) + fib(n-2) }
```

```
method iter_fib(n: Int) returns (res: Int)
  requires 0 <= n
  ensures  res == fib(n)
{
  res := 0
  var i: Int := 0
  var next: Int := 1

  while (i < n)
    invariant 0 <= i && i <= n
    invariant res == fib(i)
    invariant next == fib(i+1)
  {
    var t: Int := res
    res := next
    next := t + next
    i := i + 1
  }
}
```

Function postconditions

- Since reasoning about function calls uses the function definition, functions typically do not have postconditions

- But postconditions are permitted
  - Use keyword `result` to refer to the returned value

```
function facDef(n: Int): Int
  requires 0 <= n
  ensures  1 <= result
{ n <= 1 ? 1 : n * facDef(n-1) }
```

- When reasoning about function calls, Viper uses the function definition and the postcondition

- Postcondition is verified against function definition
  - Assumed for recursive calls
  - Dangerous when functions do not terminate!

```
function f(): Bool
  ensures  false
{ f() }
```

```
x := f()
assert false
```

Use cases for function postconditions

- Abstract functions
  - Shortcut for axiomatizing certain functions
  - In the absence of a function definition, calls are verified using only the postcondition

```
function sqrt(n: Int): Int
  requires 0 <= n
  ensures  0 <= result
  ensures  result * result <= n &&
           n < (result+1) * (result+1)
```

```
c := sqrt(a*a + b*b)
assert a*a + b*b - c*c < 2*c + 1
```

Encode a choose-statement in Viper, which returns an arbitrary integer, as an abstract function.

Use your encoding to choose two values. Can you prove that they are equal or unequal?

Solution

```
function choose(): Int
```

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

  x := choose()
  y := choose()

  assert x == y  // succeeds
  assert x != y  // fails
}
```

Encode a choose-statement in Viper, which returns an arbitrary integer, as an abstract function.

Use your encoding to choose two values. Can you prove that they are equal or unequal?

Use cases for function postconditions

```
function facDef(n: Int): Int
  requires 0 <= n
  ensures  1 <= result
{ n <= 1 ? 1 : n * facDef(n-1) }
```

```
assume 0 <= y
x := facDef(y)
assert 1 <= x // fails without post
```

- Automating induction proofs
  - SMT solvers are generally not able to prove properties about recursive functions using induction
  - By declaring a function postcondition, we provide the necessary induction hypothesis
  - Also works with methods ➔ lemmas

```
function facDef(n: Int): Int
  requires 0 <= n
  ensures  1 <= result
{
    n <= 1
       ? 1
       : n * facDef(n-1)
}
```

```
Induction hypothesis:
for all m < n, 1 <= facDef(m)
```

```
Induction base:
facDef(0) >= 1, facDef(1) >= 1
```

```
Induction step: for n > 1,
facDef(n)
=  n * facDef(n-1)
>= facDef(n-1)          (n > 1)
>= 1                  (by I.H.)
```

Exercise

- Add a function `size(t: Tree): Int` to the skeleton `10-trees.vpr` that counts the number of leafs in the tree t.
- Add a postcondition such that the client in the code skeleton verifies.

```
method client() {
  var t: Tree
  t := node(node(leaf(3), leaf(17)), leaf(22))
  assert size(t) >= 0
}
```

Solution

- Add a function `size(t: Tree): Int` to the skeleton `10-trees.vpr` that counts the number of leafs in the tree t.

- Add a postcondition such that the client in the code skeleton verifies.

```
function size(t: Tree): Int
  ensures result >= 0
{
  is_leaf(t)
    ? 1
    : size(left(t)) + size(right(t))
}
```

```
method client() {
  var t: Tree
  t := node(node(leaf(3), leaf(17)), leaf(22))
  assert size(t) >= 0
}
```

Outline

- Mathematical data types
- User-defined functions
- Function encoding

Simplified encoding of functions

- User-defined functions are encoded into the background predicate as an uninterpreted function and a definitional axiom

```
function f(x: T): TT {
  E
}
```

```
function f(x: T): TT

axiom forall x: T :: f(x) == E
```

- The axiom above is simplified; it omits
  - pre- and postconditions
  - checks that partial expressions are well-defined

Simplified encoding with pre- and postconditions

- Function pre- and postconditions are added to the definitional axiom

```
function f(x: T): TT
  requires P
  ensures  Q
{ E }
```

```
function f(x: T): TT

axiom {
  forall x: T ::
    P ==> f(x) == E && Q[result/f(x)]
}
```

- Sound, but recursive functions may lead to non-termination ➔ next module
- Note that postconditions are encoded in the axiom
  - An inconsistent postcondition can compromise soundness, even if the function is never called!

```
function f(): Bool
  ensures  false
{ f() }
```

~~`x := f()`~~
`assert false`

Well-definedness conditions for partial expressions

- New proof obligation: all expressions are well-defined
  - Example: no division by zero
  - User-defined functions are are called with arguments that satisfy their preconditions

- Well-definedness condition `DEF: Expr → Pred`
  - `DEF(e)` holds in state σ iff expression e can be evaluated in σ

Short-circuit evaluation

Expression e	*DEF*(e)
0, 1, -3, false, … (constants)	true
e1 + e2, e1 < e2, e1 && e2, …	*DEF*(e1) && *DEF*(2)
e1 / e2	*DEF*(e1) && *DEF*(e2) && e2 != 0
foo(e)	*DEF*(e) && “precondition of foo”
e1 ==> e2	*DEF*(e1) && (e1 ==> *DEF*(e2))

Encoding partial expressions

- Every statement first asserts well-definedness of its expressions

```
x := f(y) + 2 * z / n
```

precondition of `f(y)`

```
assert DEF(f(y)) && n != 0
x := f(y) + 2 * z / n
```

```
assume x/y > 1
```

```
assert y != 0
assume x/y > 1
```

- Alternative: redefine ***WP***

```
WP(x := e, Q)      ::= DEF(e) && Q[x / e]
WP(assert P, Q)    ::= DEF(P) && P && Q
WP(assume P, Q)    ::= DEF(P) && P ==> Q
                       ...
```

Wrap-up

- Writing specifications often requires a suitable mathematical vocabulary
  - added via a background predicate BP that axiomatizes uninterpreted sorts and functions
  - Verification condition: `BP ==> P ==> WP(S, Q)`

- Viper's background predicate collects axioms from multiple features
  - Built-in types and their operations
  - User-defined functions
  - Custom axiomatizations via domains

```
method collect(s: Seq[Int])
  returns (res: Set[Int])
  ensures forall j: Int ::
    0 <= j && j < |s| ==> s[j] in res
{ ... }
```

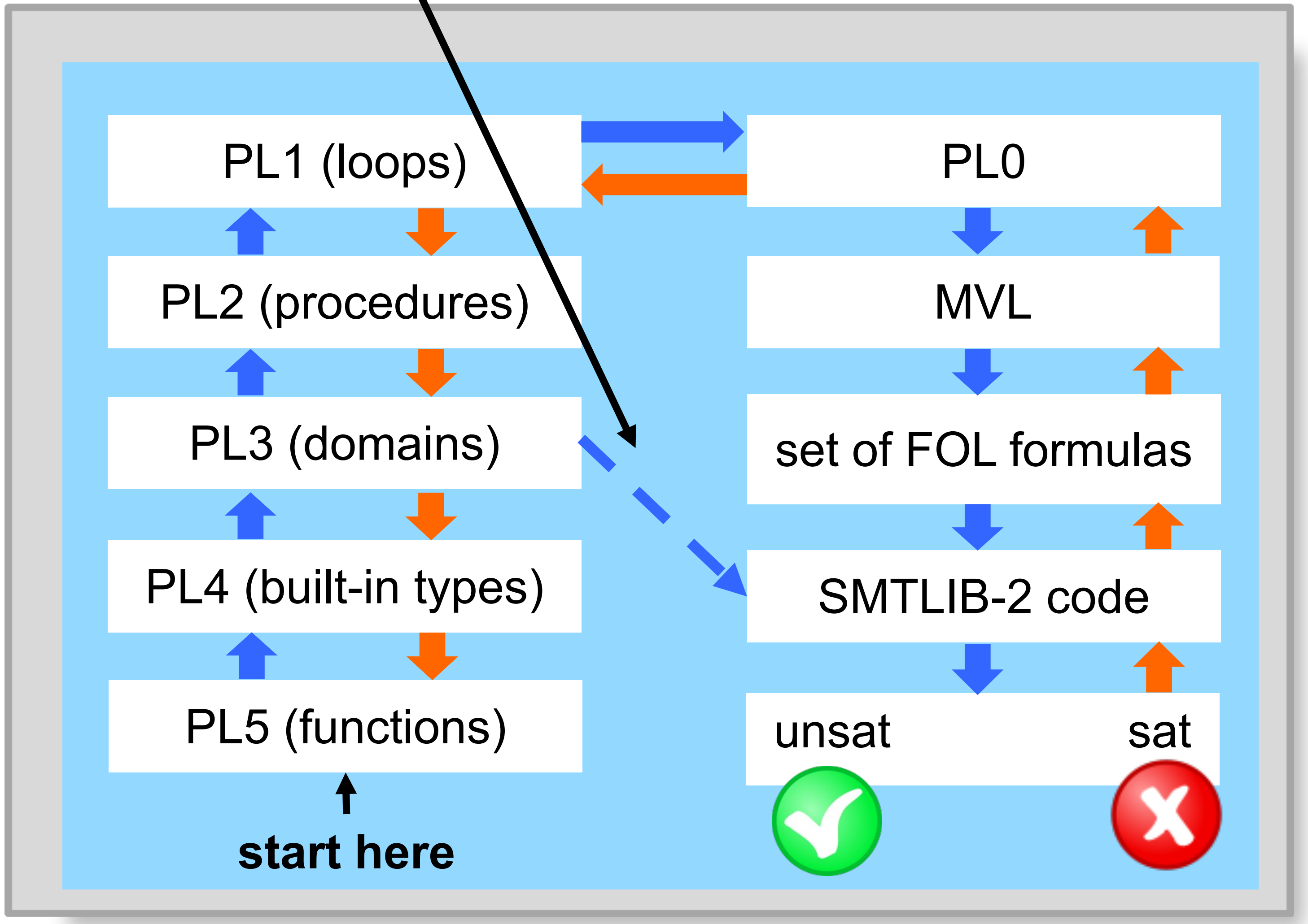
```
function f(n: Int): Int
{ n <= 1 ? 1 : n * f(n-1) }
```

```
domain Set {
  function empty(): Set
  function union(s: Set, t: Set): Set
  // ...
}
```

Wrap-up – Building Verifiers

- We now have all ingredients to implement and verify sequential programs with static memory
- Homework: try to verify some interesting programs ☺
- Next: verification tactics
  - Verifier bottlenecks
  - Pragmatics
  - Verify challenging programs

Conceptually, declarations of sorts, functions, and variables are implicit, i.e. derived from the set of FOL formulas. *In practice,* we need to provide these declarations and thus cannot fully encode PL3 to PL0. We can either directly add the SMTLIB-2 code or enrich every layer with a statement for inline SMT code that is added to the beginning of the generated SMTLIB-2 code.



Tentative course outline