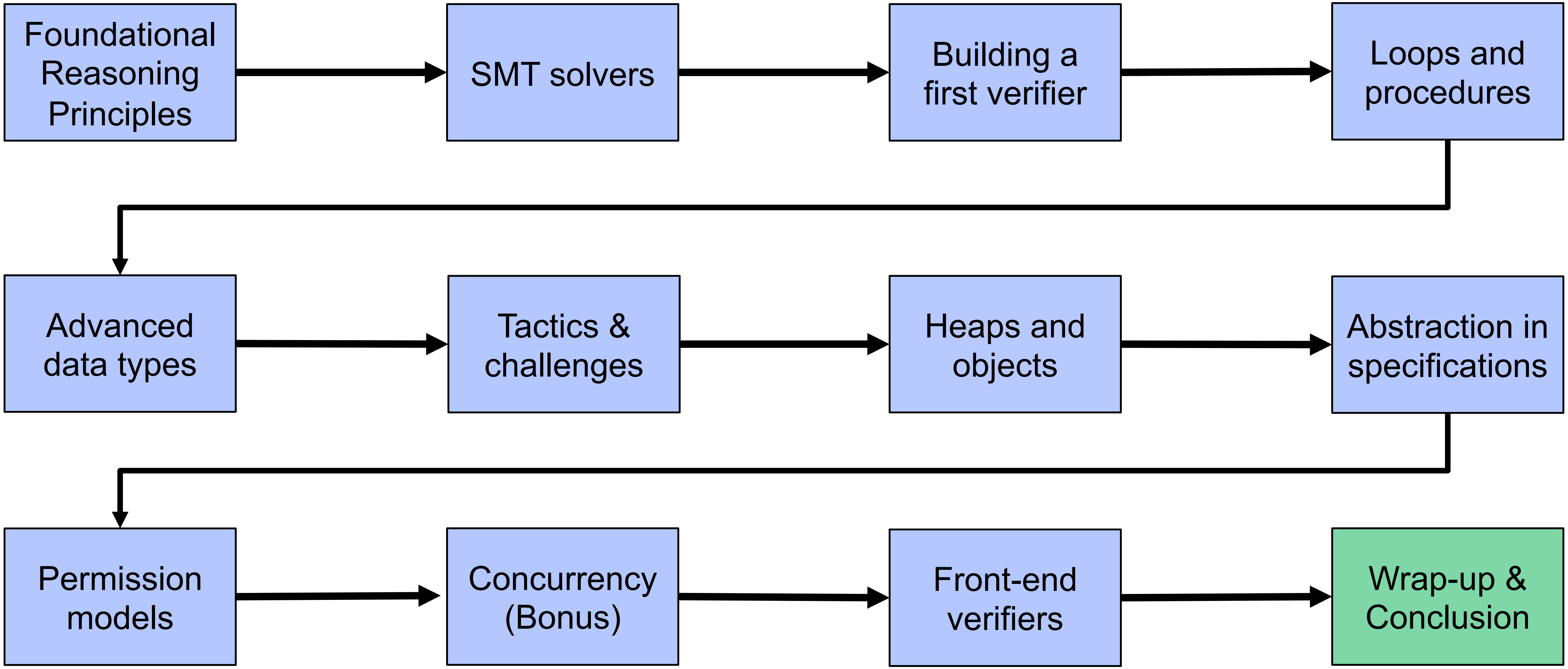
02245

WRAP-UP & CONCLUSION

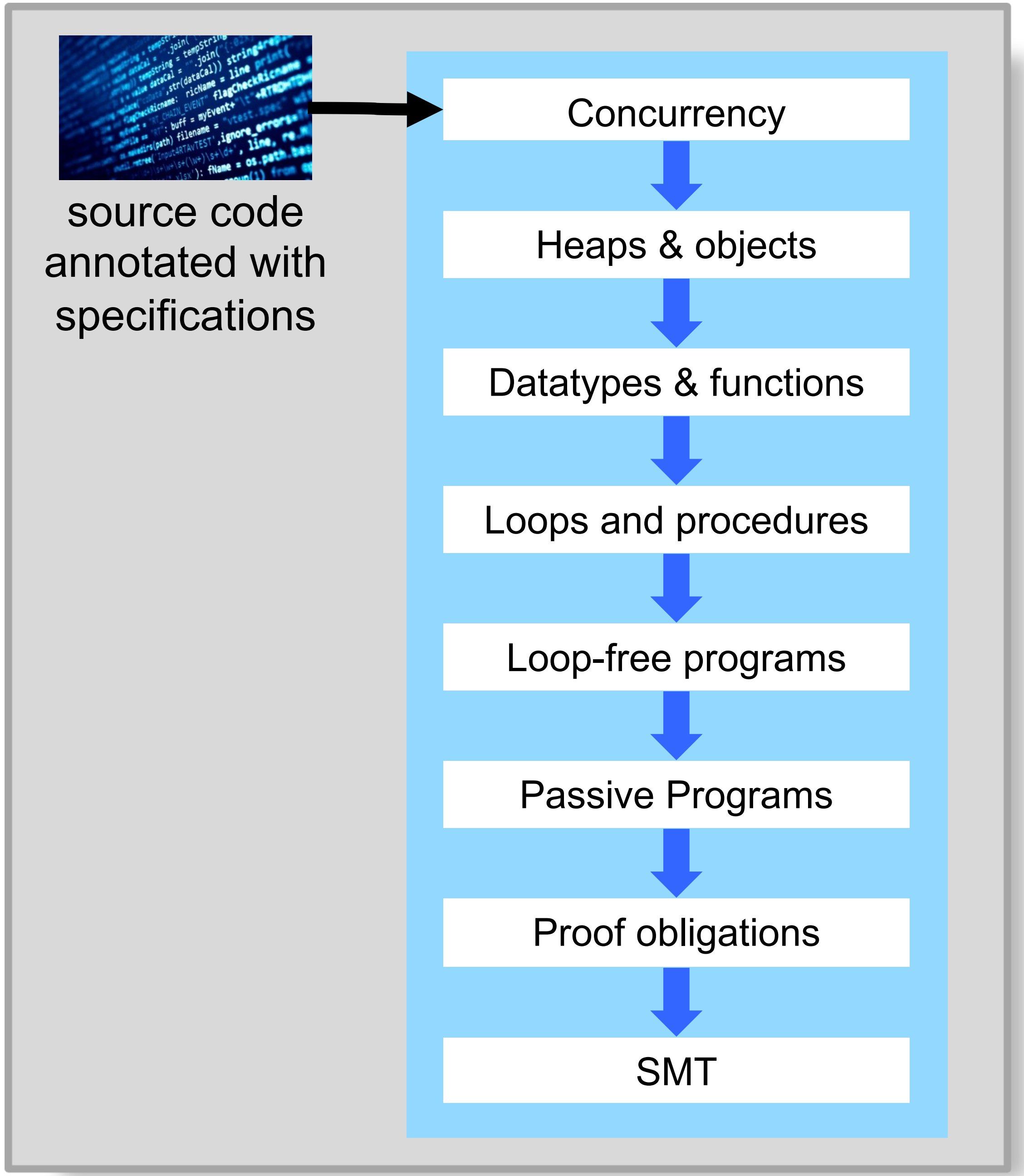
Course outline



Course Summary

Take five minutes to collect the main concepts that you have learned about in the course.

Course Verification Stack



- Techniques and tools for automated program verification
- Wishlist for each translation **A** → **B**
  - **Soundness:** If **B** is valid, then **A** is valid
  - **Completeness:** If **A** is valid, then **B** is valid
  - **Efficiency:** **B**'s size is reasonable wrt. **A**
  - **Explainability:** We can reconstruct errors in **A** from errors in **B**

Developing verification methodologies

Verification Problem

- Programming features
- High-quality specifications

Formalization

- Verification conditions
- Proof rules
- Soundness, completeness

Methodology

- Code annotations
- Modularity and Framing
- Abstraction

Automation

- Encodings
- Ghost code
- Inference

Foundational reasoning principles:

- **Weakest preconditions**
- Floyd-**Hoare logic**
- **permission**-based separation logic

Verifier architecture:

- Modern verification stack
- **Intermediate languages**
- Error reporting

General-purpose tools:

- First-order predicate logic
- SAT/**SMT** solvers
- Patterns, limited functions

Feedback for the first iteration of the course

https://forms.gle/58vES1Vz38qX6jVq5

Satisfiability Modulo Theories

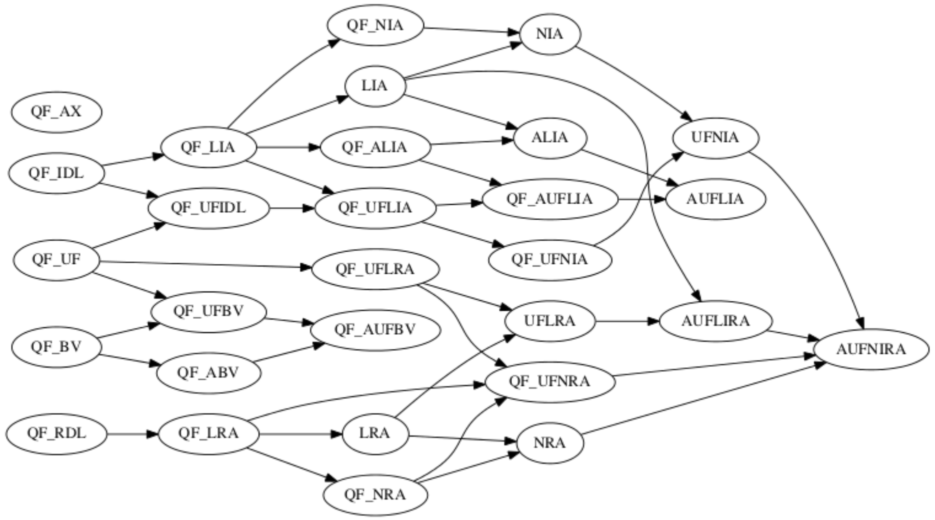
FOL formula F	$\mathfrak{I} = (\mathfrak{A}, \beta) \models$ **F if and only if**
$t_1 = t_2$	$\mathfrak{I}(t_1) = \mathfrak{I}(t_2)$
$R(t_1, \ldots, t_n)$	$\left(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n)\right) \in R^{\mathfrak{A}}$
G $\wedge$ **H**	$\mathfrak{I} \models$ **G** and $\mathfrak{I} \models$ **H**
G $\Rightarrow$ **H**	If $\mathfrak{I} \models$ **G**, then $\mathfrak{I} \models$ **H**
$\exists x$: **T** (**G**)	For some $v \in \mathbf{T}^{\mathfrak{A}}$, $\mathfrak{I}[x := v] \models$ **G**
$\forall x$: **T** (**G**)	For all $v \in \mathbf{T}^{\mathfrak{A}}$, $\mathfrak{I}[x := v] \models$ **G**

A Σ-formula **F** is satisfiable modulo the theory given by the set of axioms **AX** iff there exists a Σ-interpretation $\mathfrak{I}$ such that

- $\mathfrak{I} \models$ **F** , and
- $\mathfrak{I} \models$ **G** for every sentence **G** in **AX**.

- Signature Σ determines available symbols
- Σ-structure $\mathfrak{A}$ assigns meaning to symbols
- Σ-assignment β assigns values to variables

- Decidability of SMT problem depends on
  - the underlying theories
  - the logic fragment (e.g. without quantifiers)



Verification conditions for passive programs

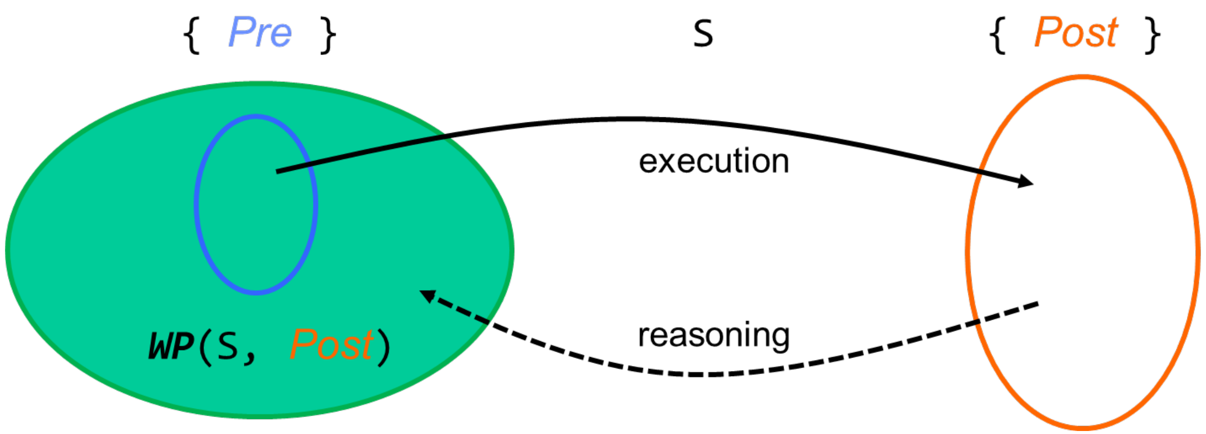
The triple `{ Pre } S { Post }` is **valid**

if and only if

when program `S` is started in any state in `Pre`, then `S` terminates in a state in `Post`

if and only if

`Pre ==> WP(S, Post)` **valid**



S	EWP(S, Q)	MWP(S, M)
assert R	R && Q	M ∪ {R}
assume R	R ==> Q	{P ==> Q | Q ∈ M}
S1; S2	EWP(S1, EWP(S2, Q))	MWP(S1, MWP(S2, M))
S1 [] S2	(B == Q) ==> EWP(S1, B) && EWP(S2, B) where B is fresh	MWP(S1, M) ∪ MWP(S2, M)

We obtain passive programs from loop-free programs by transforming them into dynamic single assignment form

Loops – Partial Correctness

```
{ I && b } S { I }
-----------------------------------------
{ I } while (b) { S } { I && !b }
```

```
while (i <= n)
  invariant 0 <= r && 1 <= i
{
  r := r + i
  i := i + 1
}
```

```
WLP(while (b)
      invariant I { S }, Q) ::= I
```

if I is a loop invariant

Frame rule

```
{ P } S { Q }    S modifies no var. in R
-----------------------------------------
     { P && R } S { Q && R }
```

```
// prior code
assert I
// havoc all loop targets
assume I
{
  assume b
  // encoding of S
  assert I
  assume false
} [] {
  assume !b
}
// subsequent code
```

Proof obligations

- Goal: procedure-modular verification
- Challenge: framing local/global variables
- Procedure implementation satisfies procedure contract

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

- Verify caller against contract

consult declared contract

Call rule

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

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

```
x := 4
y := 4
z := foo(x)
assert y + z == 20
```

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

$\textbf{assert}\ P[\overline{x}/\overline{a}]$

$\textbf{var}\ \overline{e{:}T}\ := \overline{a}$

$\textbf{havoc}\ \overline{z}$

$\textbf{assume}\ Q[\overline{x}/\overline{e}][\overline{y}/\overline{z}]$

Total correctness = Partial Correctness + Termination

A **variant** is an an expression V that decreases in every loop iteration / recursive call (for some well-founded ordering <).

\# iterations / recursive calls

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

```
{ I && b && V == z } S { I && V < z }
-------------------------------------
  { I } while (b) { S } { I && !b }
```

```
while (i <= n) {
  var z: Int := n - i + 1
  assert z >= 0
  r := r + i
  i := i + 1
  assert n - i + 1 >= 0 && n - i + 1 < z
}
```

```
define V(m) (m)

method factorial(n: Int) returns (res: Int)
  requires 0 <= n
  // decreases V(m)
{
  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
  }
}
```

Datatypes

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

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

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

```
(declare-sort Set)

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

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

Background Predicate: conjunction of all axioms

Verification condition:

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

Functions

- Writing specifications often requires a suitable mathematical vocabulary

- Functions are encoded through their 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)]
}
```

- Challenges:
  - Well-definedness conditions for partial functions
  - Recursive functions may lead to non-termination ➔ lemmas, patterns, limited functions
  - May increase trusted code base

Heaps & Objects

- Object based language with field accesses `x.f`

- Implicit garbage collection

- Heaps map references and field names to values

```
type HeapType = Map<T>[(Ref, Field T), T]
```

- Represented as a global variable

```
var Heap: HeapType
```

Heap data structures pose four major challenges for sequential verification:

- Reasoning about aliasing

- Framing, especially for dynamic data structures

- Writing specifications that preserve information hiding

- Data structures with complex sharing

Permission-based separation logic

- Read or write access to memory location `x.f` requires permission `acc(x.f)`
  - Refinement: fractional permissions to distinguish no, read, and write accesses
- Permissions can be transferred, but neither duplicated nor forged
  - **`inhale`** `P`: obtain all permissions required by assertion `P` and assume all logical constraints
  - **`exhale`** `P`: assert all logical constraints, check and remove all permissions required by assertion `P`, and havoc any locations to which all permission is lost
- **Intuition:** permission is held by methods, loop iterations, or predicate instances
- Separating conjunction $P * Q$

```
acc(x.f) * acc(y.f) ==> x != y
```

Frame rule

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P * R\ \}\ S\ \{\ Q * R\ \}}$$

where `S` does not assign to a variable that is free in `R`

Predicates

```
predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}
```

- Predicates enable specifying
  - data structure permissions
  - data structure invariants
  - lock invariants
  - other resources (e.g. runtimes)
- Iso-recursive semantics distinguishes between a predicate instance and its (recursive) body

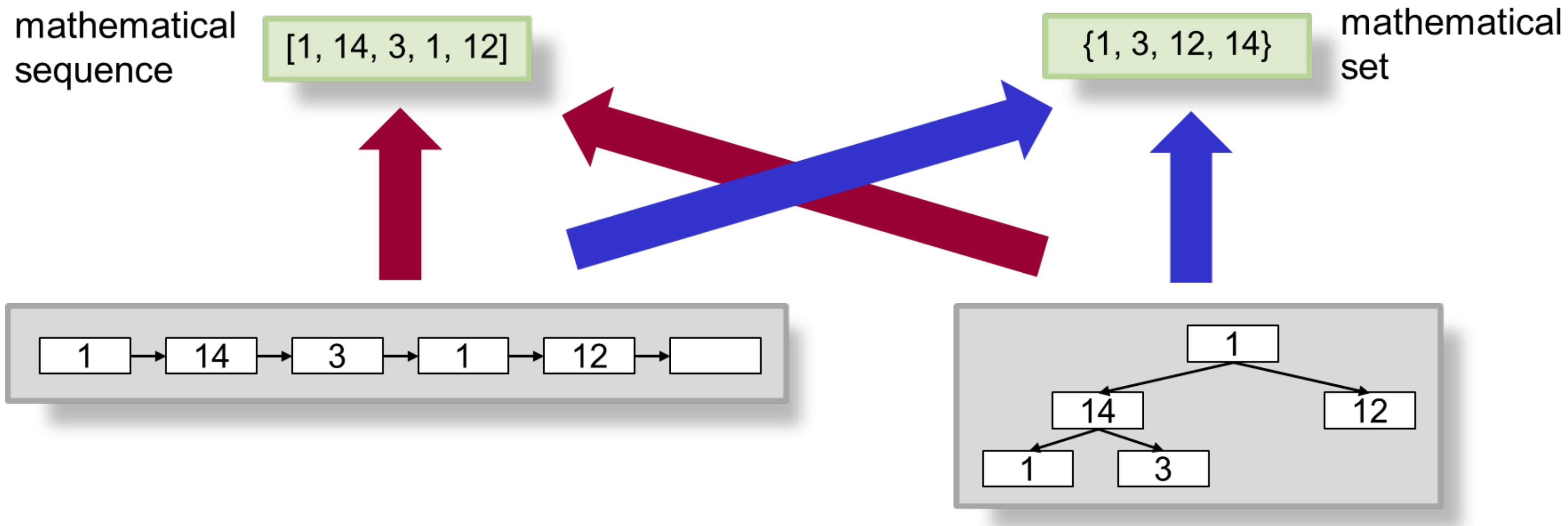
- An unfold statement exchanges a predicate instance for its body

```
inhale list(x)
unfold list(x)
x.next := null
```

- A fold statement exchanges a predicate body for a predicate instance

```
inhale list(x)
unfold list(x)
x.next := null
fold list(x)
exhale list(x)
```

Abstraction



Data abstraction via predicate arguments

```
predicate list(this: Ref, content: Seq[Int]) {
  acc(this.elem) && acc(this.next) &&
  (this.next == null ==> content == Seq[Int]()) &&
  (this.next != null ==> 0 < |content| &&
                         content[0] == this.elem &&
                         list(this.next, content[1..]))
}
```

Data abstraction via abstraction functions

```
function content(this: Ref): Seq[Int]
{
  this.next == null
   ? Seq[Int]()
   : Seq(this.elem) ++ content(this.next)
}
```

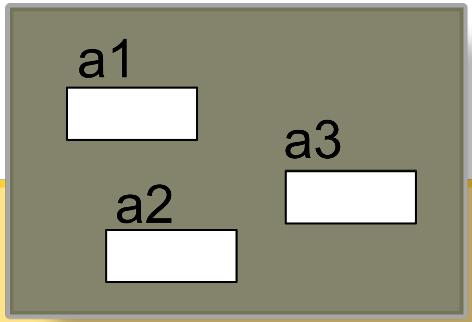
Ghost code

- Code that is needed for verification, but not for the execution of the code
  - Fold and unfold statements
  - Auxiliary variables and fields
  - Abstraction functions
  - Lemmas
  - Assertions for quantifier instantiations
  - Entire methods for internally traversing data structures (`addAtEnd`)

- General rule for ghost code
  The execution of ghost code must not affect the behavior of regular code

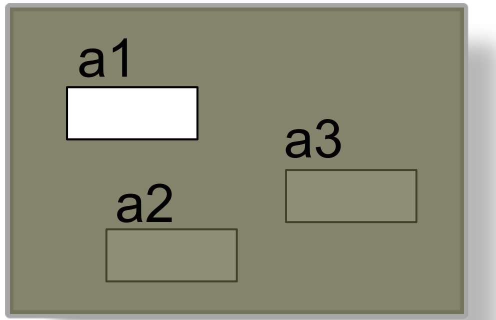
- Examples
  - Ghost variables must not occur in conditions of regular conditionals and loops
  - Ghost statements must not assign to regular variables
  - Ghost code must terminate

Concurrency: Thread-local state



Intuition: permissions are held by methods, loop iterations, predicate instances, and threads

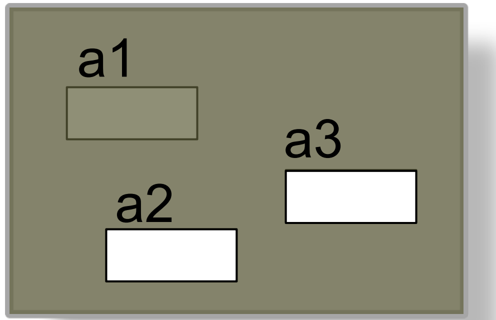
Thread-local state: parallel branches operate on disjoint memory

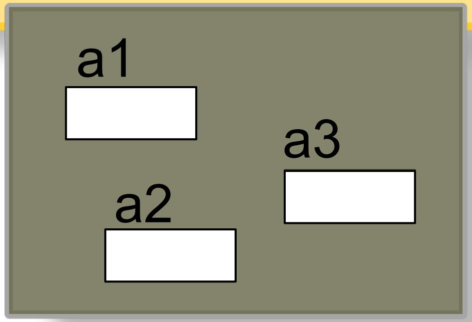



```
            a1 := new(bal)
            a2 := new(bal)
            a3 := new(bal)
            deposit(a2, 150)

deposit(a1, 50)  ||  transfer(a2, a3, 100)

            assert a1.bal == a2.bal
```






```
exhale P1[…]
exhale P2[…]
havoc res1, res2
inhale Q1[…]
inhale Q2[…]
```

Parallel composition rule

$$\frac{\{\ P1\ \}\ S1\ \{\ Q1\ \} \qquad \{\ P2\ \}\ S2\ \{\ Q2\ \}}{\{\ P1 * P2\ \}\ S1 \mid\mid S2\ \{\ Q1 * Q2\ \}}$$

Extension: unstructured fork-join parallelism

Concurrency: shared state and synchronization

Intuition: permissions are held by methods, loop iterations, predicate instances, threads, and locks

```
class Buffer {
  lock invariant acc(this.val)
  Product val;
}
```


```
method produce(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val == null) {
      buf.val := new()
    }
    release buf
  }
}
```

```
assert acc(buf.isLock, wildcard)
inhale acc(buf.val)
```


```
exhale acc(buf.val)
```

```
method consume(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val != null) {
      // consume buf.val
      buf.val := null
    }
    release buf
  }
}
```

Many potential project topics in our group – get in touch!

Open research topics at all levels

- Foundational program logics
- New frontend verifiers
- Tool automation
- Advanced specification features
- Specification inference
- Debugging tools

- We offer student projects on all aspects of program verification & formal methods

- Specification and verification techniques for new program features & properties
- Theoretical foundations
- New proof rules & program logics
- Automation
- Performance optimization
- Case studies
- IDE support

< END >

Thank you for attending the course!

Questions?