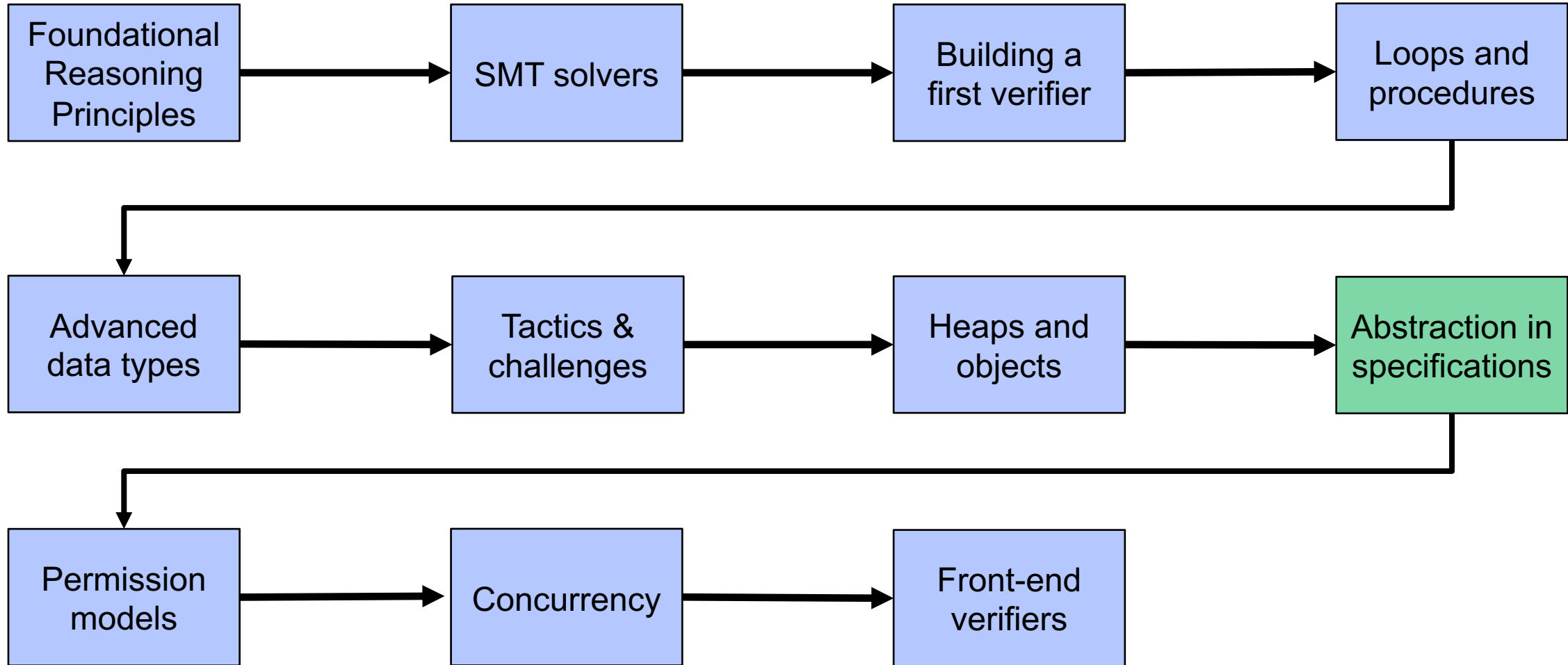


02245 – Module 8

# ABSTRACTION

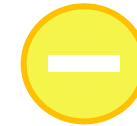
# Tentative course outline



# Challenges revisited

Heap data structures pose three major challenges for sequential verification

- Reasoning about aliasing
  - Permissions and separating conjunction
- Framing, especially for dynamic data structures
  - Sound frame rule, but no support yet for unbounded data structures
- Writing specifications that preserve information hiding
  - Not solved



And additional challenges for concurrent programs, e.g., data races

- Permissions are an excellent basis, but see later

# Running example: linked lists

```
field elem: Int
field next: Ref

method head(this: Ref) returns (res: Int)
  requires acc(this.elem)
  ensures  acc(this.elem)
  ensures  res == this.elem
{
  res := this.elem
}
```

```
method append(this: Ref, e: Int)
  requires // permission to all nodes
  ensures  // list was extended
{
  if(this.next == null) {
    var n: Ref
    n := new(*)
    n.next := null
    this.elem := e
    this.next := n
  } else {
    append(this.next, e)
  }
}
```

- Specification reveals implementation details

- Permissions and behavior cannot be expressed so far

# Abstraction

1. Predicates: abstraction over permissions
2. Representation invariants
3. Data abstraction: abstraction over values
4. Abstraction functions

# User-defined predicates

- User-defined predicates consist of a predicate name, a list of parameters, and a self-framing assertion

## Declarations

```
D ::= ... | predicate U( $\overline{x:T}$ ) { P }
```

```
predicate node(this: Ref) {  
  acc(this.elem) && acc(this.next)  
}
```

- Predicate instances are predicates

## Predicates (or assertions)

```
P ::= ... | U( $\bar{E}$ )
```

```
method head(this: Ref) returns (res: Int)  
  requires node(this)  
  ensures  node(this)  
{ ... }
```

# Recursive predicates

→ 01-lists.vpr

- Predicate definitions may be recursive

## Declarations

```
D ::= ... | predicate U( $\overline{x:T}$ ) { P }
```

## Predicates (or assertions)

```
P ::= ... | U( $\overline{E}$ )
```

- Recursive predicate definitions are interpreted as **least fixed points**
- All instances of the predicate have **finite unfoldings**

- Recursive predicates may denote a statically-unbounded number of permissions

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

- If `list(x)` holds, we have `x != x.next`
- `list` describes a **finite** linked list

# Static verification with recursive predicates

- A program verifier cannot know statically how far to unfold recursive definitions

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

```
inhale list(x)  
y.next := null // do we have permission?
```

- Proving properties of predicates often requires induction proofs

→ difficult to automate

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

```
inhale list(x)  
exhale step(x) // should this succeed?
```

```
inhale step(x)  
exhale list(x) // should this succeed?
```



# Iso-recursive predicates

- We interpret (non-limited) functions **equi-recursively**: function = body
- **Iso-recursive** semantics distinguishes between a predicate instance and its body

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

```
inhale list(x)  
x.next := null // no permission
```



- Intuition: permissions are held by method executions, loop iterations,  
or **predicate instances**

# Folding and unfolding predicates

- Exchanging a predicate instance for its body, and vice versa, is done via extra program statements

## Statements

```
S ::= ...
    | unfold U( $\bar{E}$ )
    | fold U( $\bar{E}$ )
```

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

- 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)
```

# Exercise: list operations

→ 03-list-operations.vpr

Implement additional list operations, such as:


- a. method `create()` returns `(res: Ref)`
- b. method `prepend(this: Ref, e: Int)` returns `(res: Ref)`
- c. method `concat(this: Ref, that: Ref)`
- d. method `reverse(this: Ref)` returns `(res: Ref)`

Hints:

- Write **recursive implementations** (we will discuss iterative implementations shortly)
- Write specifications using the `list` predicate (but **no functional behavior** yet)

# Solution

→ `03-list-operations.vpr`

```
method length(this: Ref) returns (res: Int)
  requires list(this)
  ensures list(this) 
{
  res := 0
  var curr: Ref := this
  unfold list(this)
  while(curr.next != null)
    invariant acc(curr.next)
    invariant (curr.next != null ==> list(curr.next))
    {
      res := res + 1
      curr := curr.next
      unfold list(curr)
    }
}
```

- Postcondition fails
- Loop invariant includes permissions to the list nodes still to be visited ...
- ... but not to the list nodes already visited
- Entire list is transferred into the loop, but not back

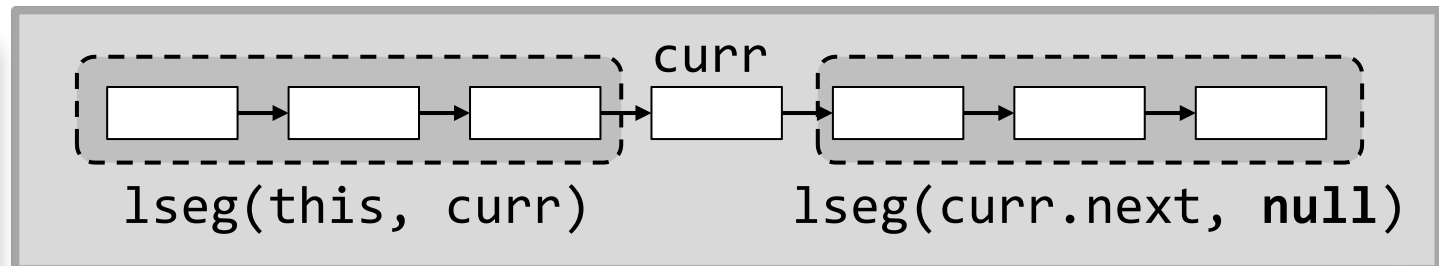
# List segments

- Describing the “list nodes already visited” requires a way to describe a partial list
- The `lseg` predicate describes (possibly empty) list segments

```
predicate lseg(this: Ref, last: Ref) {  
  this != last ==> acc(this.next) &&  
    lseg(this.next, last)  
}
```

- Our loop invariant can now include the nodes still to be visited as well as the nodes already visited

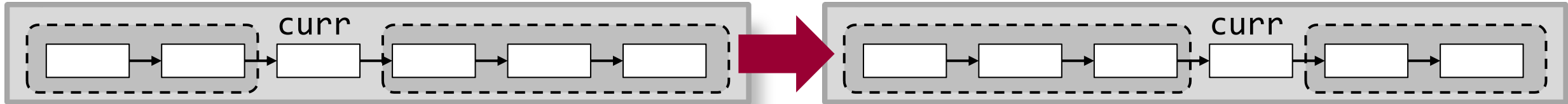
```
while(curr.next != null)  
  invariant acc(curr.next)  
  invariant lseg(curr.next, null)  
  invariant lseg(this, curr)
```



# Manipulating recursive definitions

→ 05-addAtEnd.vpr

- Advancing the `curr` pointer requires extending an `lseg` predicate at the end



- This requires
  - unfolding the recursive predicate instances all the way to the end
  - folding predicate instances for the extended list segment in the reverse order
- Operation is best implemented as a recursive auxiliary method

```
method addAtEnd(this: Ref, n: Ref, nn: Ref)
  requires lseg(this, n)
  requires acc(n.next) && n.next == nn
  ensures lseg(this, nn)
{
  if(this == n) {
    fold lseg(nn, nn) // empty segment
    fold lseg(this, nn)
  } else {
    unfold lseg(this, n)
    addAtEnd(this.next, n, nn)
    fold lseg(this, nn)
  }
}
```

# Iterative data structure traversals revisited

→ [06-length.vpr](#)



```
method length(this: Ref) returns (res: Int)
  requires this != null && lseg(this, null)
  ensures lseg(this, null)
{
  res := 0
  var curr: Ref := this
  unfold lseg(this, null)
  fold lseg(this, curr) // empty segment to establish loop invariant
  while(curr.next != null)
    invariant acc(curr.next) && lseg(curr.next, null) && lseg(this, curr)
    {
      res := res + 1
      var tmp: Ref; tmp := curr.next // read here, before permission is folded
      addAtEnd(this, curr, curr.next) // extend predicate
      curr := tmp
      unfold lseg(curr, null)
    }
  addAtEnd(this, curr, null)
}
```



# Reminder: ghost code

- Many operations of our example are needed for verification, but not for the execution of the code

- Fold and unfold operations
- Entire `addAtEnd` method and calls

```
var tmp: Ref; tmp := curr.next
addAtEnd(this, curr, curr.next)
curr := tmp
unfold lseg(curr, null)
```

- Code that is used for verification only is called **ghost code**

- Statements
- Entire methods
- Variables, fields, functions, ...

- General rule for ghost code

The execution of ghost code must not affect the behavior of regular code

- Examples

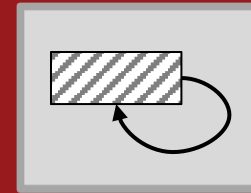
- Ghost variables must not occur in conditions of regular conditionals and loops
- Ghost statements must not assign to regular variables

- Viper does not distinguish regular and ghost code and, thus, does *not* enforce this rule

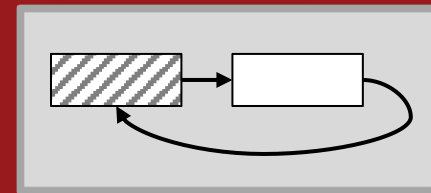
# Exercise: cyclic lists

→ 07-cyclic.vpr

- Write a predicate `lseg(this)` that represents cyclic lists  
Hint: use the `lseg` predicate
- Implement and verify a method that creates an empty list
- Implement and verify a method that inserts an element right after the sentinel node



empty list  
(sentinel only)

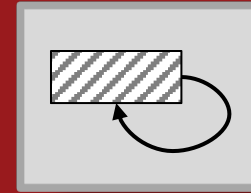


list with one  
element  
(plus sentinel)

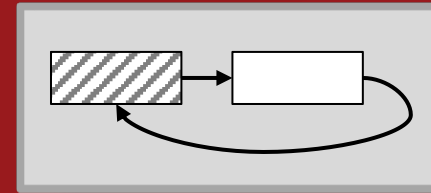
```
predicate lseg(this: Ref, last: Ref) {  
  this != last ==> acc(this.next) &&  
    lseg(this.next, last)  
}
```

# Solution: cyclic lists (a)

- Write a predicate `list(this)` that represents cyclic lists  
Hint: use the `lseg` predicate
- Implement and verify a method that creates an empty list
- Implement and verify a method that inserts an element right after the sentinel node



empty list  
(sentinel only)



list with one  
element  
(plus sentinel)

```
predicate list(this: Ref) {  
  lseg(this, this)  
}
```



predicate is empty

```
predicate list(this: Ref) {  
  acc(this.next) &&  
  lseg(this.next, this)  
}
```



# Solution: cyclic lists (b, c)

- b. Implement and verify a method that creates an empty list

```
method create() returns (res: Ref)
  ensures list(res)
{
  res := new(*)
  res.next := res
  fold lseg(res, res)
  fold list(res)
}
```

- Implement and verify a method that appends an element right after the sentinel node

```
method insert(this: Ref)
  requires list(this)
  ensures list(this)
{
  unfold list(this)
  var n: Ref
  n := new(*)
  n.next := this.next
  fold lseg(n, this)
  this.next := n
  fold list(this)
}
```

# Abstract predicates

- Predicate bodies reveal implementation details

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

- Predicates can also remain abstract

```
predicate list(this: Ref)
```

- Abstract predicates cannot be folded or unfolded

- Abstract predicates and abstract methods allow one to specify interfaces without revealing implementation details

```
predicate list(this: Ref)  
  
method append(this: Ref, e: Int)  
  requires list(this)  
  ensures list(this)
```

- Predicate and method definitions are visible to implementer, but not clients

# Encoding of predicates

- Recall that permissions are tracked in a global permission mask

```
type MaskType = Map<T>[(Ref, Field T), Bool]  
var Mask: MaskType
```

- We use the same mask to track predicate instances
- For this purpose, we map each predicate instance to a field

```
predicate lseg(this: Ref, last: Ref)
```

```
function  
lsegField(this: Ref, last: Ref): Field Int
```

and use that field to index the mask

```
lseg(a, b)
```

```
Mask[null, lsegField(a, b)]
```

- The uninterpreted functions are axiomatized to return unique values

# Encoding of unfold and fold

- An unfold statement exchanges a predicate instance for its body

**unfold**  $U(\bar{E})$

**exhale**  $U(\bar{E})$   
**inhale**  $body(U(\bar{E}))$

- Actual implementation of fold delays havoc until the predicate instance is exhaled

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

**fold**  $U(\bar{E})$

**exhale**  $body(U(\bar{E}))$   
**inhale**  $U(\bar{E})$

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





# Abstraction

1. Predicates: abstraction over permissions
2. Representation invariants
3. Data abstraction: abstraction over values
4. Abstraction functions

# Representation invariants

→ 08-list.vpr

→ 09-field-len.vpr

- Data structures typically maintain several consistency conditions
  - Value constraints, e.g., references being non-null or integers being positive
  - Structural constraints, e.g., a tree being balanced
  
- Such representation invariants are
  - Established by constructors
  - Assumed and preserved by all operations

- Representation invariants can be expressed as part of a predicate

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

```
method append(this: Ref, e: Int)  
  requires list(this)  
  ensures  list(this)  
{  
  unfold list(this) // assume invariant  
  ...  
  fold list(this)   // check invariant  
}
```

- Unfold and fold are statements because they change the state (heap and mask)
- Unfolding-expressions allow one to temporarily unfold a predicate during the evaluation of an expression
- They enable inspecting fields whose permissions are folded inside a predicate

Expressions  
 $E ::= \dots \mid \text{unfolding } U(\bar{E})$

```
predicate list(this: Ref) {  
  acc(this.elem) && acc(this.next) && acc(this.len) &&  
  (this.next == null ==> this.len == 0) &&  
  (this.next != null ==> list(this.next) &&  
    unfolding list(this.next) in this.len == this.next.len + 1)  
}
```

# Exercise: sorted lists

- Write a user-defined predicate `list(this)` that represents sorted integer lists

→ `10-list-sorted.vpr`

# Solution: sorted lists

- Write a user-defined predicate `list(this)` that represents sorted integer lists

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

# Abstraction

1. Predicates: abstraction over permissions
2. Representation invariants
3. Data abstraction: abstraction over values
4. Abstraction functions

# Specifying functional behavior

→ 11-list-fun-spec.vpr

- Using `old`-expressions and `unfolding`-expressions, we can specify some aspects of functional behavior
- But: approach does not work when behavior depends on an unbounded number of fields (e.g., sorting a list)
- And: specifications reveal implementation details

```
predicate list(this: Ref) {  
  acc(this.next) && acc(this.len) &&  
  (this.next == null ==> this.len == 0) &&  
  (this.next != null ==> list(this.next) &&  
    unfolding list(this.next) in  
    this.len == this.next.len + 1)  
}
```

```
method append(this: Ref, e: Int)  
  requires list(this)  
  ensures list(this)  
  ensures (unfolding list(this) in this.len) ==  
    old(unfolding list(this) in this.len + 1)
```

# Challenges revisited

Heap data structures pose three major challenges for sequential verification

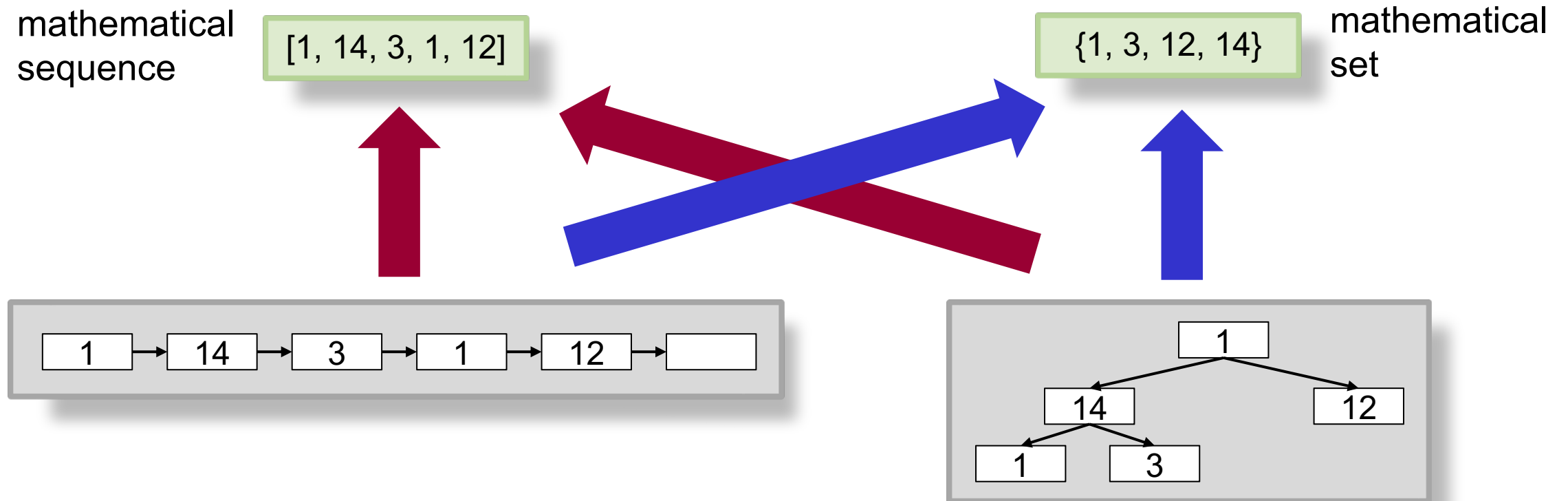
- Reasoning about aliasing
  - Permissions and separating conjunction
- Framing, especially for dynamic data structures
  - Sound frame rule, predicates
- Writing specifications that preserve information hiding
  - Not solved





# Data abstraction

- To write implementation-independent specifications, we map the concrete data structure to mathematical concepts and specify the behavior in terms of those



- Mapping can be a function or a relation

- One data abstraction approach is to store the abstract value in a ghost field and express the mapping as representation invariant

```
field content: Seq[Int] // ghost field

predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) && acc(this.content) &&
  (this.next == null ==> this.content == Seq[Int]()) &&
  (this.next != null ==> list(this.next) &&
    unfolding list(this.next) in
    this.content == Seq(this.elem) ++ this.next.content
  )
}
```

```
method create() returns (res: Ref)
  ensures list(res)
  ensures (unfolding list(res) in res.content) == Seq[Int]()
{
  res := new(*)
  res.next := null
  res.content := Seq[Int]()
  fold list(res)
}
```

- Since the abstract value is stored in a ghost field, mutation of the data structure requires **explicit updates** of these ghost fields
- Specifications need to **unfold predicates** to get access to the ghost field
- **Information hiding is violated** (predicates cannot be abstract)

- An alternative is to pass the abstract value as parameter to the predicate
- A representation invariant expresses the mapping between the argument and the data structure

```
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 predicate arguments (cont'd)

```
method append(this: Ref, e: Int, c: Seq[Int])
  requires list(this, c)
  ensures list(this, c ++ Seq(e))
{
  unfold list(this, c)
  if(this.next == null) {
    var n: Ref
    n := new(*)
    n.next := null
    this.elem := e
    this.next := n
    fold list(n, Seq[Int]())
  } else {
    append(this.next, e, c[1..])
  }
  fold list(this, c ++ Seq(e))
}
```

- Solution supports information hiding
- No unfolding-expressions required in specifications
- Preconditions would ideally quantify existentially over abstract value

```
method append(this: Ref, e: Int)
  requires exists c: Seq[Int] ::
    list(this, c)
  ensures list(this, c ++ Seq(e))
```

- Since existentials are problematic for automation, ghost arguments are required instead (overhead!)

# Exercise: list abstraction

- Extend the prepend method from an earlier exercise with a functional specification using the above predicate (see template).
- You might observe a surprising error message. Why does it occur? Can you fix it without changing the specification?

→ `15-list-abstraction.vpr`

# Solution: list abstraction

Extend the prepend method from an earlier exercise with a functional specification using the above predicate.

You might observe a surprising error message. Why does it occur? Can you fix it without changing the specification?

```
method prepend(this: Ref, e: Int, c: Seq[Int]) returns (res: Ref)
  requires list(this, c)
  ensures  list(res, Seq(e) ++ c)
{
  assert unfolding list(this, c) in true // trivial assertion
  res := new(*)
  res.elem := e
  res.next := this
  fold list(res, Seq(e) ++ c)
}
```

- `this != null` follows from the access predicates inside the list predicate
- But the verifier learns that only once the predicate is unfolded
- The trivial assertion achieves that

# Abstraction

1. Predicates: abstraction over permissions
2. Representation invariants
3. Data abstraction: abstraction over values
4. Abstraction functions



# Avoiding ghost variables to express abstraction

```
field content: Seq[Int]

method create() returns (res: Ref)
  ensures list(res)
  ensures unfolding list(res) in ...
{
  res := new(*)
  res.next := null
  res.content := Seq[Int]()
  fold list(res)
}
```

```
method append(this: Ref, e: Int, c: Seq[Int])
  requires list(this, c)
  ensures list(this, c ++ Seq(e))
```

- Both techniques we have seen so far require ghost variables
- Storing the abstract value in additional variables is unnecessary: **all information is already present** in the data structure
- Access this information analogously to getter-methods in programs: **return the desired information without changing the state**

- For “getter” methods to be meaningful in assertions, they must
  - be side-effect free,
  - terminate, and
  - be deterministic
- Viper provides **heap-dependent functions** for this purpose
- Function bodies are **expressions**
- Functions may be **recursive**, but termination is not checked by default

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

(incomplete declaration)

## Expressions

$E ::= f(\bar{E})$

# Encoding of heap-dependent functions

- Heap-dependent functions are encoded as uninterpreted functions
- Function body is encoded as a **definitional axiom**

```
function f(x: T): T' {  
  E  
}
```

```
function f(x: T, h: HeapType): T'  
axiom forall x: T, h: HeapType :: f(x, h) == [[E]]
```

- `[[_]]` is the encoding function (omitted for types), parametric in the heap
  - Actual definition is more complex and uses limited functions (see Module 5)
  - A proof obligation checks that the function body is well-defined (omitted here)
- Function calls are applications of these functions in the (current or old) heap

```
f(E)
```

```
f([[E]], Heap)
```

# Another frame problem

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

```
// assume we have list(x) && acc(y.f)
tmp := content(x)
y.f := 5
assert tmp == content(x)
```



```
tmp := content(x, Heap)
assert Mask[y, f]
Heap[y, f] := 5
assert tmp == content(x, Heap)
```

- Each heap update modifies the (global) heap
- Any information about heap-dependent functions is lost
- Recovering the information by inspecting the function body would violate information hiding and would not work for abstract functions

# Read effects

- Heap-dependent functions must have a **precondition that frames the function body**, that is, provides all permissions to evaluate the body
- The precondition over-approximates the locations the function value depends on (its **read effect**)
- If permission to a location is not included in the precondition, modifying it cannot affect the function value, which allows framing

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

```
// assume we have list(x) && acc(y.f)
tmp := content(x)
y.f := 5
assert tmp == content(x)
```



# Framing axioms

- The read effect is used to generate a framing axiom for the function
- If two heaps agree on a function's read effect then the function yields the same result in both heaps

```
function get(x: Ref): Int
  requires acc(x.elem)
  { ... }
```

```
function get(x: Ref, h: HeapType): int

axiom forall x: Ref, h1: HeapType, h2: HeapType ::
  h1[x,elem] == h2[x,elem] ==> get(x, h1) == get(x, h2)
```

- How to express framing axioms if the read effect contains an unbounded set of locations?

```
function content(this: Ref): Seq[Int]
  requires list(this)
  { ... }
```

# Predicate versions

- Precisely encoding the set of locations contained in a predicate instance is complex and hard to reason about
- We approximate the values stored in these locations by giving each predicate instance a **version number**
- When two instances of a predicate have the same version, all contained locations have the same value (but not vice versa)
- Version is changed each time one of these locations might change, during unfold

```
function  
listField(this: Ref): Field Int
```

```
Heap[null, listField(a)]
```

```
exhale list(x)  
havoc Heap[null, listField(x)]  
inhale body(list(x))
```

# Framing axioms with predicate versions

- Using predicate versions, fields and predicates are handled analogously in the framing axioms

```
function get(x: Ref): Int
  requires list(x)
{ ... }
```

```
function get(x: Ref, h: HeapType): int

axiom forall x: Ref, h1: HeapType, h2: HeapType ::
  h1[null, listField(x)] == h2[null, listField(x)] ==>
  get(x, h1) == get(x, h2)
```



# Reminder: partial functions

- Preconditions of heap-dependent functions specify the read effect
- Like method preconditions, they may also constrain the function arguments (including the heap)

```
function length(this: Ref): Int  
  requires list(this)  
{ ... }
```

```
function first(this: Ref): Int  
  requires list(this) && 0 < length(this)  
{  
  content(this)[0]  
}
```

- Definitional axioms provide a partial definition of the (total) uninterpreted function

```
function f(x: T): T'  
  requires P  
{ E }
```

```
function f(x: T, h: HeapType): T'  
  
axiom forall x: T, h: HeapType ::  
  [[P]] ==> f(x, h) == [[E]]
```

# Exercise: heap-dependent functions

- a. Express the functional specification of the `prepend` method using the `content` function
- b. Implement the `length` function from the previous slide
- c. Extend the specification of `prepend` with information about the list length

→ `17-list-function.vpr` (bottom)

# Solution: heap-dependent functions

- Express the functional specification of the prepend method using the content function
- Implement the length function from the previous slide
- Extend the specification of prepend with information about the list length

```
function length(this: Ref): Int
  requires list(this)
{
  unfolding list(this) in
    (this.next == null ?
      0 : length(this.next) + 1
    )
}
```

```
method prepend(this: Ref, e: Int) returns (res: Ref)
  requires list(this)
  ensures list(res)
  ensures content(res) == Seq(e) ++ old(content(this))
  ensures length(res) == 1 + old(length(this))
{
  assert unfolding list(this, c) in true
  res := new(*)
  res.elem := e
  res.next := this
  fold list(res)
}
```

# Wrap-up: data abstraction

- Ghost fields + invariants
  - Manual updates of ghost state
  - Predicates cannot be abstract
- Predicate arguments
  - Support information hiding
  - Require ghost parameters
- Heap-dependent functions
  - Support information hiding
  - Typically exist in programs anyway (getters)
  - **Separation of concerns:**  
predicates for permissions and invariants,  
functions for abstraction

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

```
function length(this: Ref): Int  
  requires list(this)  
{ ... }
```

```
function first(this: Ref): Int  
  requires list(this) && 0 < length(this)  
{ ... }
```

```
function content(this: Ref): Seq[Int]  
  requires list(this)  
{ ... }
```

# Challenges revisited

Heap data structures pose three major challenges for sequential verification

- Reasoning about aliasing
  - Permissions and separating conjunction
- Framing, especially for dynamic data structures
  - Sound frame rule, predicates
- Writing specifications that preserve information hiding
  - Data abstraction, heap-dependent functions

