

02245 – Module 9

PERMISSION MODELS

Previously...

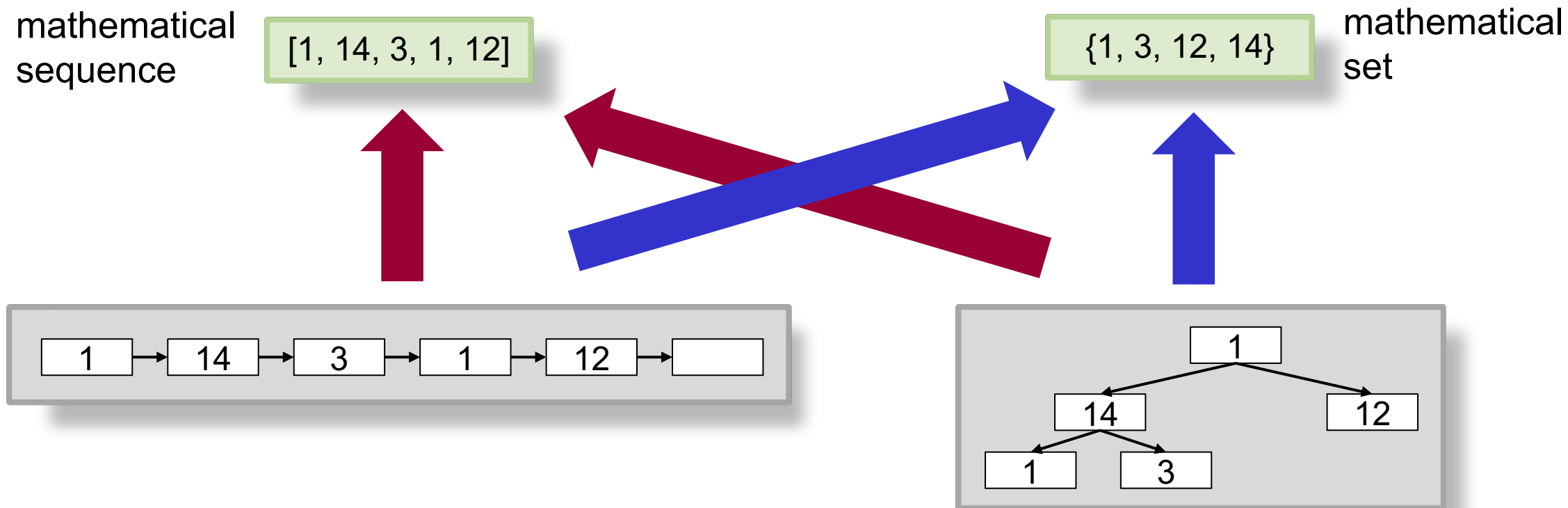
Heap data structures pose three major challenges for sequential verification

- 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



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

Data abstraction via predicate arguments

```
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!)

Data abstraction via abstraction functions

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

(incomplete declaration)

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]  
}
```

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)  
{ ... }
```

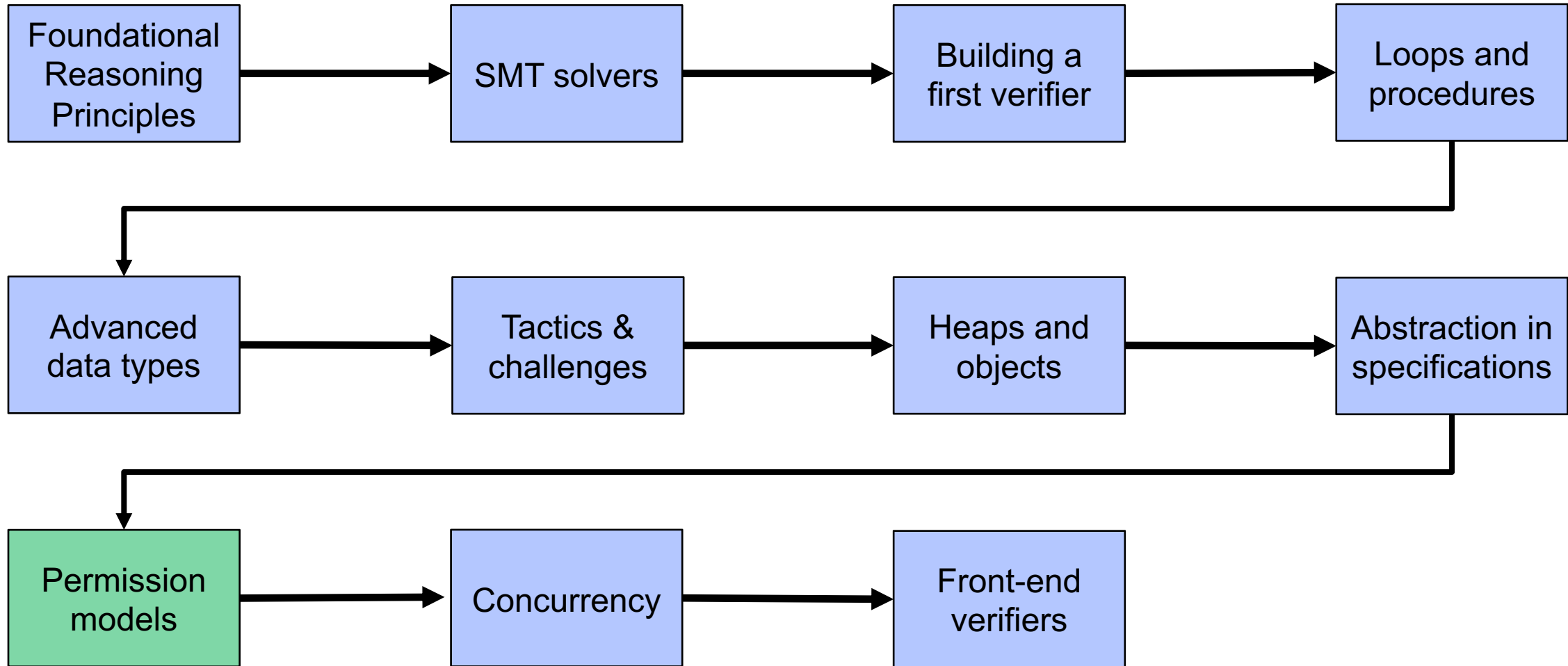
```
function first(this: Ref): Int  
  requires list(this) && 0 < length(this)  
{ ... }
```

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

02245 – Module 9

PERMISSION MODELS

Tentative course outline



Advanced permission models

1. Fractional permissions
2. Quantified permissions

```
method cloneList(this: Ref) returns (res: Ref)
  requires list(this) // read only
  ensures list(this) && list(res)
  ensures content(this) == old(content(this))
{
  res := new(*)
  unfold list(this)
  if(this.next == null) {
    res.next := null
  } else {
    var tmp: Ref
    tmp := cloneList(this.next)
    res.elem := this.elem
    res.next := tmp
  }
  fold list(this)
  fold list(res)
}
```

- Methods that only read a data structure must specify that each abstraction remains **unchanged**
- Adding an abstraction requires changes to existing specifications (**non-modular**)
- Possible solution: specify that predicate version remains unchanged (not possible in Viper)
- We introduce a more expressive solution in the following

Fractional permissions

- To **distinguish read and write access**, permissions can be split and re-combined
- A **permission amount π** is a rational number in $[0,1]$
- Viper syntax
 - Permissions are fractions n/d
 - **write** for $1/1$ and **none** for $0/1$
 - **acc(E.f)** is a shortcut for **acc(E.f, write)**
 - **P(E)** is a shortcut for **acc(P(E), write)**
- **Field read** requires a **non-zero** permission
- **Field write** requires **full (write)** permission

Predicates (or assertions)

```
P ::= ...  
    | acc(E.f,  $\pi$ )  
    | acc(P( $\bar{E}$ ),  $\pi$ )
```

```
inhale acc(x.f, 1/2)  
v := x.f
```



```
inhale acc(x.f, 1/2)  
x.f := v
```



Manipulating fractional permissions

- Separating conjunction sums up permissions of the conjuncts

`acc(x.f, 1/2) && acc(x.f, 1/2)`

is equivalent to

`acc(x.f, 1/1)`

- `inhale` *adds* permissions
- `exhale` *subtracts* permissions and `havocs` only when *all* permission to a location or predicate instance is removed
- Values are *framed* as long as *some* permission is held

```
method cloneList(this: Ref) returns (res: Ref)
  requires acc(list(this), 1/2) // read only
  ensures  acc(list(this), 1/2) && list(res)
  { ... }
```

```
method frameList(this: Ref) returns (l: Ref)
  requires list(this)
  {
    var tmp1: Seq[Int]
    tmp1 := content(this)
    l := cloneList(this) // no havoc of version
    assert tmp1 == content(this)
  }
```

Predicates and fractional permissions

→ 02-clone.vpr
→ 02-fractions.vpr

- Predicates may contain fractional permissions, e.g. to permit sharing
- Field locations with more than full permission are infeasible (magic)
- Predicate instances with more than full permission are feasible (no magic)
- **Unfold and fold multiply** the fraction of the predicate with the fractions in the predicate body

```
predicate readCell(this: Ref) {  
  acc(this.cell) && acc(this.cell.val, 1/2)  
}
```

```
predicate P(this: Ref) {  
  acc(this.val, 1/2)  
}
```

```
inhale acc(x.val)  
fold P(x)  
fold P(x)  
exhale P(x) && P(x) // not false
```

```
inhale acc(readCell(x), 1/4)  
unfold acc(readCell(x), 1/4)  
exhale acc(x.cell.val, 1/8)
```



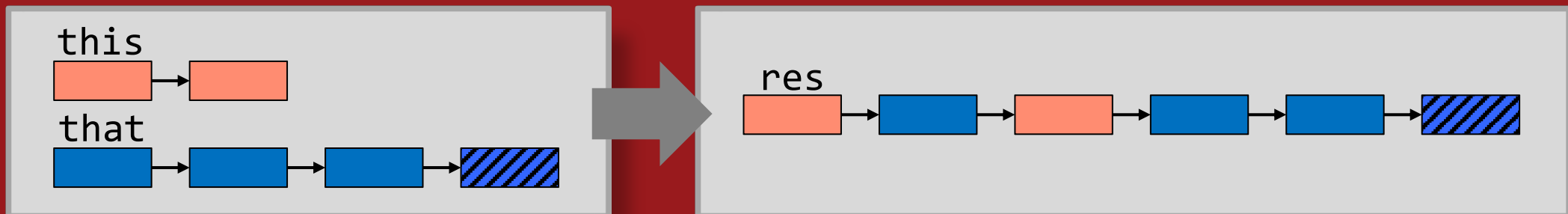
Exercise: fractional permissions

→ 03-list-zip.vpr

- Implement a method that creates a new list by zipping together two lists (see diagram):
`method zip(this: Ref, that: Ref) returns (res: Ref)`
- Write a reasonable specification such that the method verifies and returns all permissions it holds; use full permissions only.
- Adjust your specification and implementation to use fractional permissions where possible.
- Can you verify a client that zips a list with itself?

■ Hints:

- Do not write a functional specification (yet)
- You can use method `cloneList` for the case that one of the two lists is empty.
- You may swap the arguments for the recursive call.



Solution: fractional permissions

```
method zip(this: Ref, that: Ref) returns (res: Ref)
  requires acc(list(this), 1/2) && acc(list(that), 1/2)
  ensures  acc(list(this), 1/2) && acc(list(that), 1/2)
  ensures  list(res)
{
  unfold acc(list(this), 1/2)
  if(this.next == null) {
    res := cloneList(that)
  } else {
    res := new(*)
    res.elem := this.elem
    var rest: Ref
    rest := zip(that, this.next)
    res.next := rest
    fold list(res)
  }
  fold acc(list(this), 1/2)
}
```

```
method client(this: Ref) returns (res: Ref)
  requires list(this)
  ensures  list(this) && list(res)
{
  res := zip(this, this)
}
```



Heap-dependent functions

- Heap-dependent functions may only read the heap
- Hence, an arbitrarily small fraction would be sufficient
- Problem: we don't know how often permissions are split
- Possible solution: use **wildcard** to avoid concrete fraction

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

```
inhale acc(list(this), 1/2)
x := length(this)
```



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

Adjusted encoding: permissions and field access

- Permissions are tracked in a global permission mask

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

- Convention: $\text{Mask}[\text{null}, f] == 0.0$ for all fields f

- Field access

```
v := x.f
```

```
assert Mask[x, f] > 0.0  
v := Heap[x, f]
```

```
x.f := E
```

```
assert Mask[x, f] == 1.0  
Heap[x, f] := E
```

- Field access requires permission!

Adjusted encoding: inhale

- **inhale** A means:
 - obtain all permissions required by assertion A
 - assume all logical constraints
- Encoding is defined recursively over the structure of A

inhale acc(E.f, π)

Mask[[[E]],f] := Mask[[[E]],f] + π
assume Mask[[[E]],f] <= 1.0

Reaching more than full permission for a **field** location goes to magic

inhale acc(P(E), π)

Mask[null,PField([[E]])] := Mask[null,PField([[E]])] + π

inhale A && B

[[**inhale** A]]; [[**inhale** B]]

Separating conjunction:
add sum of permissions

- The encoding also asserts that E and π are well-defined (omitted here)

Adjusted encoding: exhale

- **exhale** A means:
 - assert all logical constraints
 - check and remove all permissions required by assertion A
 - havoc any locations to which all permission is lost
- Encoding is defined recursively over the structure of A

```
exhale acc(E.f,  $\pi$ )
```

```
assert Mask[[[E]],f]  $\geq \pi$   
Mask[[[E]],f] := Mask[[[E]],f] -  $\pi$ 
```

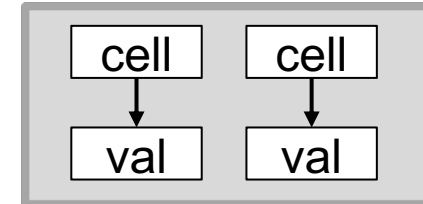
```
exhale A
```

```
var oldMask: MaskType  
var newHeap: HeapType  
oldMask := Mask  
[[exhale A]]  
assume forall y,g :: Mask[y,g] > 0.0 ==> newHeap[y,g] == Heap[y,g]  
Heap := newHeap // effectively havocs all locations to which all  
permission was lost
```

Sharing in data structures

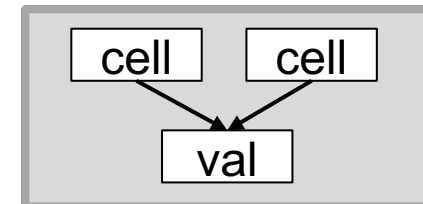
- Full permissions can describe tree-shaped data structures only

```
predicate exclusiveCell(this: Ref) {  
  acc(this.cell) && acc(this.cell.val)  
}
```



- Fractional permissions allow sharing

```
predicate sharedCell(this: Ref) {  
  acc(this.cell) && acc(this.cell.val, 1/2)  
}
```

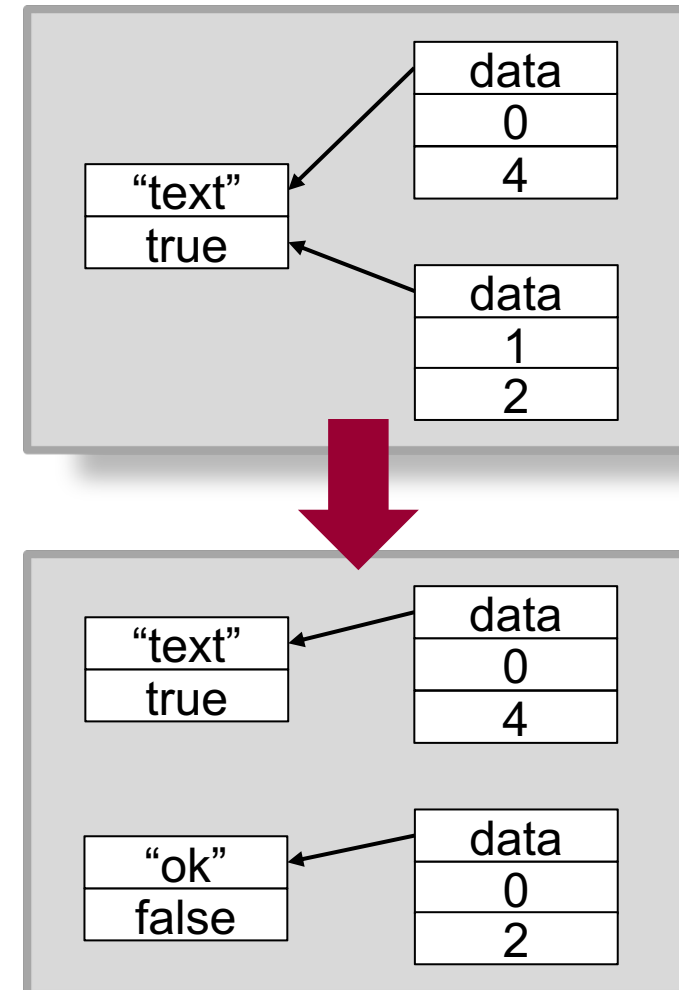


- Sharing is used in many data structures
 - Examples: doubly-linked lists, global data, caches, graphs, ...

Case study: binary reference counting

- Binary reference counting optimizes code that uses immutable data
- Keep track whether the data is shared
- Updates on shared data perform a copy (“copy on write”)
- Updates on unshared data perform a destructive update
- Once shared, the data does not go back to unshared (unlike with full reference counting)

- Example: text segments



The Perm type

- The demo uses a ghost field of type `Perm`
- Values of type `Perm` include:
 - constants `none`, `write`, `wildcard`, and fractions
 - expressions, e.g., `write - x.frac` or `2*write`
- `Perm` is typically used for ghost variables
 - Parameterize methods that require read permission
 - Perform permission accounting when permissions are distributed and later re-collected
- Type `Perm` is encoded as a real

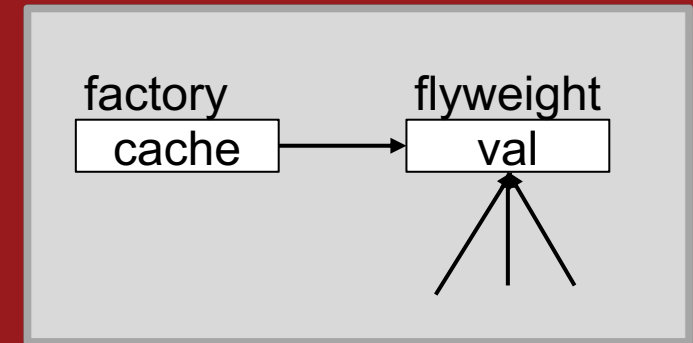
```
field frac: Perm
```

```
method cloneList(this: Ref, p: Perm)
                    returns (res: Ref)
  requires acc(list(this), p)
  ensures  acc(list(this), p) &&
           list(res)
```

Exercise: sharing

→ 05-flyweight.vpr

- Implement a simplified version of the Flyweight pattern with the following properties:
- A flyweight object has a single field `val`.
- The factory manages only one object.
- The factory's `get` method returns a flyweight object and provides read access to its `val` field.
- It obtains this flyweight object from a cache, and creates it if the cache is empty.



Solution: sharing

```
// Flyweight
field val: Int

// Factory
field cache: Ref
field frac: Perm

predicate factory(this: Ref) {
  acc(this.cache) && acc(this.frac) &&
  none < this.frac &&
  (this.cache != null ==>
    acc(this.cache.val, this.frac))
}
```

```
method get(this: Ref) returns (fly: Ref, p: Perm)
  requires factory(this)
  ensures factory(this)
  ensures none < p && acc(fly.val, p)
{
  unfold factory(this)
  if(this.cache == null) {
    fly := new(val)
    this.cache := fly
    this.frac := write
  }
  fly := this.cache
  this.frac := this.frac / 2
  p := this.frac
  fold factory(this)
}
```

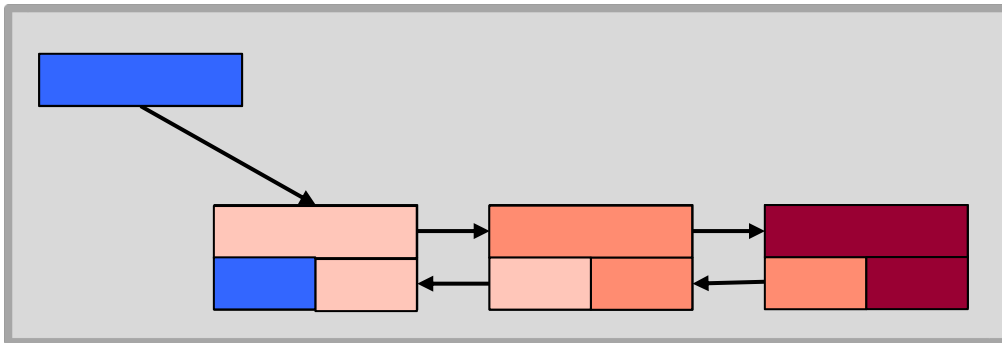
Sharing in mutable data structures

- Previously: immutable shared objects
- To specify mutable data structures with sharing, we arrange fractional permissions such that they can be combined to obtain a full permission

- Example: doubly-linked list

```
predicate nodes(this: Ref) {  
  acc(this.next) && acc(this.prev, 1/2) &&  
  (this.next != null ==>  
    acc(this.next.prev, 1/2) &&  
    this.next.prev == this &&  
    nodes(this.next)  
  )  
}
```

```
predicate dlist(this: Ref) {  
  acc(this.head) &&  
  (this.head != null ==>  
    acc(this.head.prev, 1/2) &&  
    this.head.prev == null &&  
    nodes(this.head)  
  )  
}
```



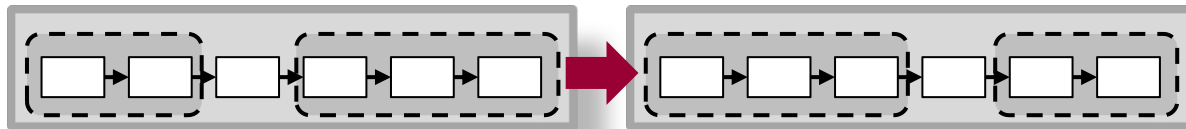
Advanced permission models

1. Fractional permissions

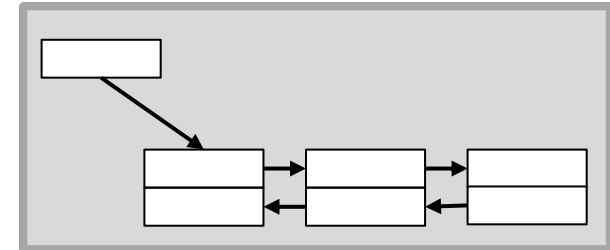
2. Quantified permissions

Limitations of recursive predicates

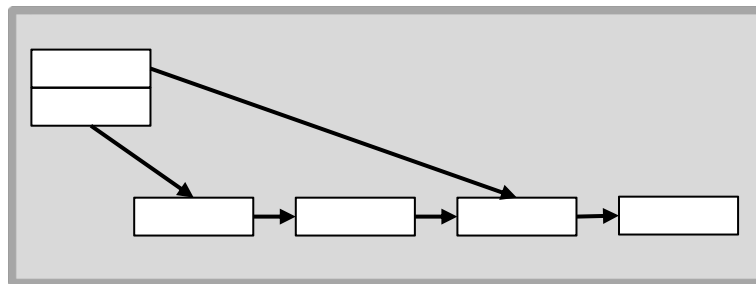
- Recursive predicates allow one to specify unbounded data structures
 - Traversals happen **in the order** in which the predicate needs to be unfolded
- Predicates are not ideal for many other use cases



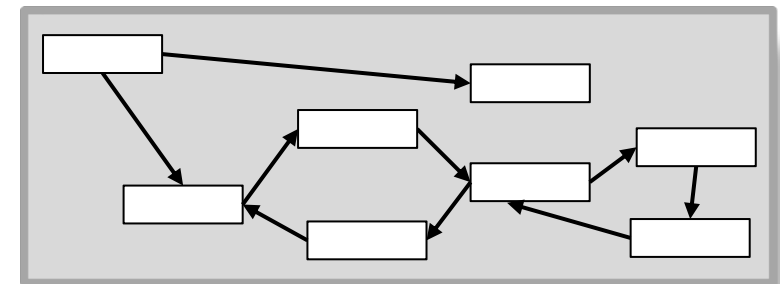
Iterative traversals



Other traversal orders



Random-access data structures



Arbitrary cyclic data structures

Quantified permissions

- To denote permission to an unbounded set of locations without prescribing a traversal order, we allow permissions and predicates in universal quantifiers

Permissions

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

- Universal quantifiers can be thought of as a possibly-infinite iterated conjunction

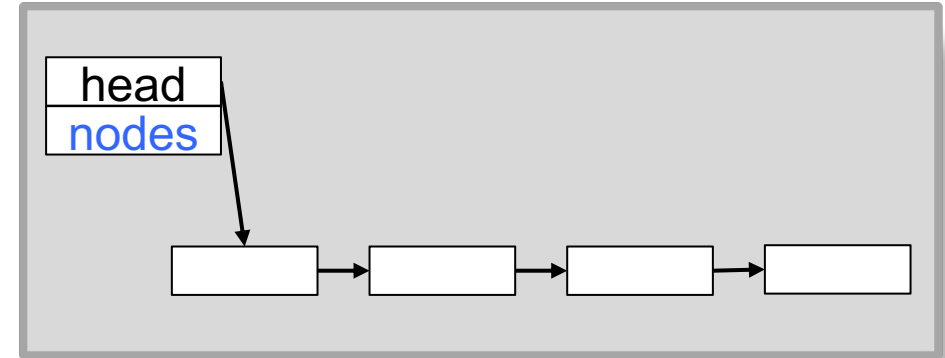
$\text{forall } x:T :: P \iff P[x/v1] \wedge P[x/v2] \wedge \dots$

- Viper's forall represents a possibly-infinite **iterated separating conjunction**

$\text{forall } x:T :: P \iff P[x/v1] * P[x/v2] * \dots$

Explicit footprints

- As alternative to predicates, we can specify permission to an unbounded set of locations by
 - maintaining an explicit set of references as ghost state (the explicit footprint)
 - quantifying over the set elements in specifications

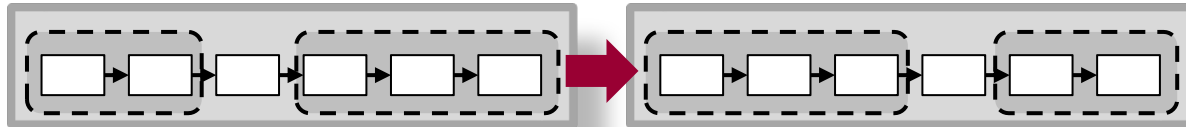


```
field head: Ref
field nodes: Set[Ref] // explicit footprint
```

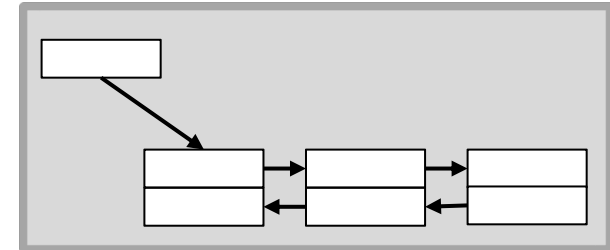
```
predicate list(this: Ref) {
  acc(this.head) && acc(this.nodes) &&
  (forall n: Ref :: n in this.nodes ==> acc(n.elem) && acc(n.next) &&
    (n.next != null ==> n.next in this.nodes)) &&
  (this.head != null ==> this.head in this.nodes)
}
```


Limitations of recursive predicates

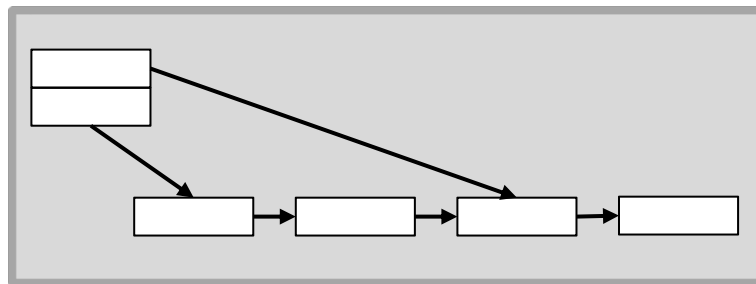
- Recursive predicates allow one to specify unbounded data structures
 - Traversals happen **in the order** in which the predicate needs to be unfolded
- Predicates are not ideal for many other use cases



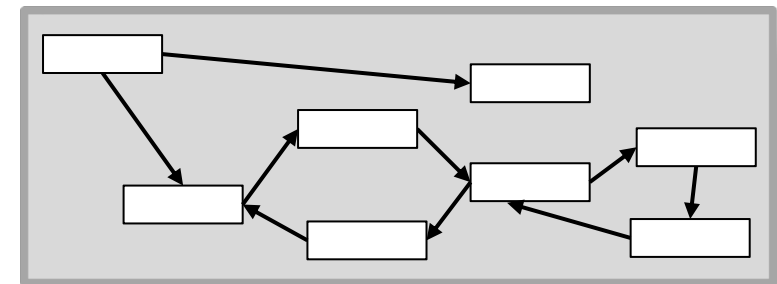
Iterative traversals



Other traversal orders



Random-access data structures



Arbitrary cyclic data structures

Well-formed quantified permissions

- Viper requires for each assertion $\text{acc}(E.f)$ under a **forall** $x:T$ that E is **injective** for all instantiations x_1, x_2 :

$$x_1 \neq x_2 \implies E[x/x_1] \neq E[x/x_2]$$

- Analogous rule applies to predicates (for parameter tuples)
- Examples

```
forall x: Ref :: x in s ==> acc(x.f) // s has type Set[Ref]
```



```
forall x: Ref :: acc(y.f) && (y.f != x ==> P(x))
```

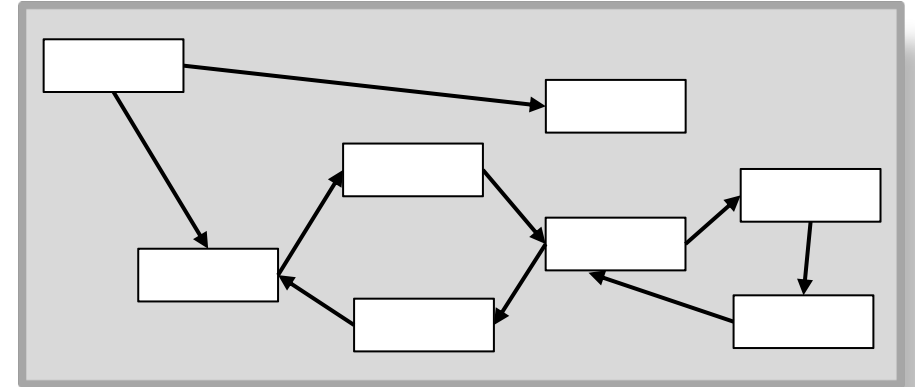


```
forall i: Int :: 0 <= i && i < |t| ==> acc(t[i].f) // t has type Seq[Ref]
```

well-formed if t contains no duplicates

Complex sharing: graph marking

- We represent a graph as a set of nodes
- Each node stores a (possibly empty) set of successors
- Each node contains a flag that is set during marking



```
field next: Set[Ref]
```

```
field flag: Bool
```

```
define graph(nodes) (
```

```
  forall n: Ref :: n in nodes ==> acc(n.next) && acc(n.flag) && (n.next subset nodes)
)
```

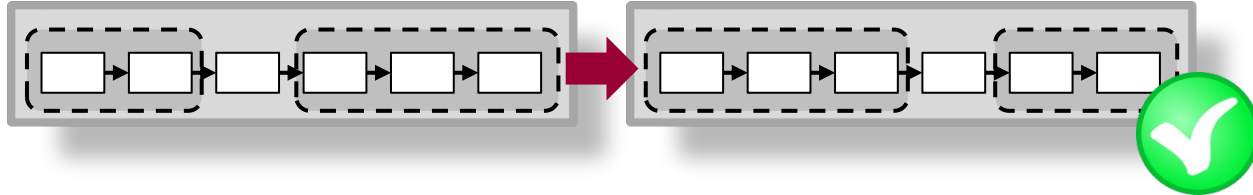
Exercise: cycle detection in lists

→ 8-is-cyclic.vpr

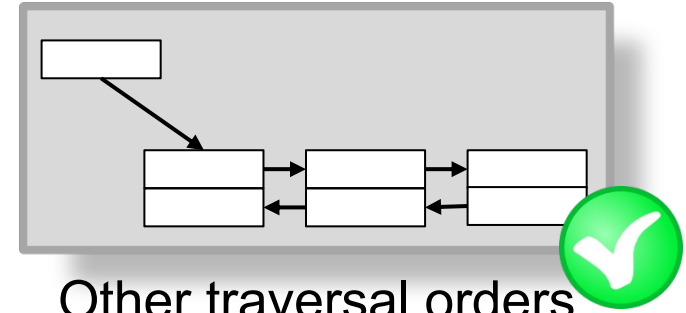
- Implement and verify a method
method `isCyclic(nodes: Set[Ref], root: Ref)` returns `(res: Bool)`
- that returns whether a singly-linked list starting at `root` is cyclic.
- Hints:
 - Represent the list as set of nodes
 - Use a variable to keep track of nodes already traversed
 - Verify memory safety, but not functional correctness

< Solution >

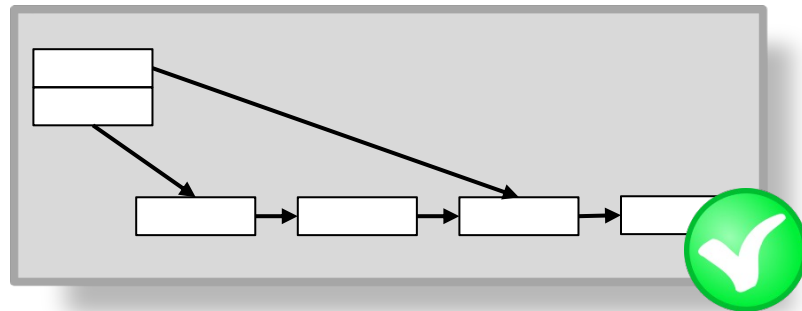
Quantified permissions address the limitations of predicates



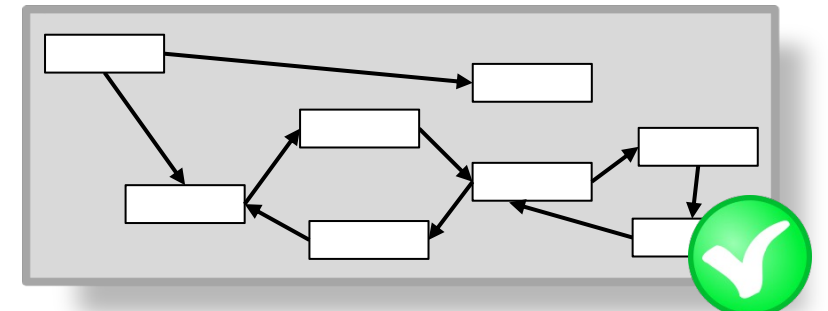
Iterative traversals



Other traversal orders



Random-access data structures



Arbitrary cyclic data structures

Arrays

- Viper does not have built-in arrays
- In contrast to sequences, arrays are mutable heap data structures
- We model arrays by a set of disjoint references that can be accessed via an index
- `loc(a, i).val` models `a[i]`
- More-dimensional arrays can be encoded analogously

```
field val: Int // for integer arrays

domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int
  function first(r: Ref): Array
  function second(r: Ref): Int

  axiom injectivity {
    forall a: Array, i: Int :: {loc(a, i)}
      first(loc(a, i)) == a &&
      second(loc(a, i)) == i
  }

  axiom length_nonneg {
    forall a: Array :: len(a) >= 0
  }
}
```

Accessing array locations

- Arrays are random-access data structures
- We can express permissions using quantified permissions

```
forall i: Int :: 0 <= i && i < len(a) ==> acc(loc(a, i).val)
```

- Similarly for sub-ranges of the array

- We define macros for convenient access

```
define lookup(a, i)  
  loc(a, i).val
```

```
define update(a, i, e) {  
  loc(a, i).val := e  
}
```

- Bounds are checked implicitly via permissions

Wrap-up: advanced permission models

- Fractional permissions
 - Distinguish between read and write permission
 - Are useful to express sharing, to strengthen framing, and for concurrency (see later)
- Quantified permissions
 - Complement predicates for the specification of unbounded data structures
 - Are especially useful for random-access structures, complex sharing, and flexible traversals
 - Inherit challenges of quantification (controlling instantiations, performance)
- Other permission models exist
 - Magic wands (permission-aware implication): useful to specify partial data structures
 - Counting permissions are related to fractional permissions, but use units

Exercise: two-dimensional arrays

→ 10-2d-array.vpr

- Encode two-dimensional arrays, including the domain and access macros.
- Write a method `reset` that sets all array elements to zero

< Solution >

Master thesis projects on verification (in a broad sense)

- Extending & improving a [source code verifier for Rust](#)
- Developing new verification methodologies
- Reasoning about [probabilistic](#) programs
- Formalizing meta theory
- Developing new decision procedures
- Verification case studies
- < your proposal >

Contact me after class or
write to chmat@dtu.dk