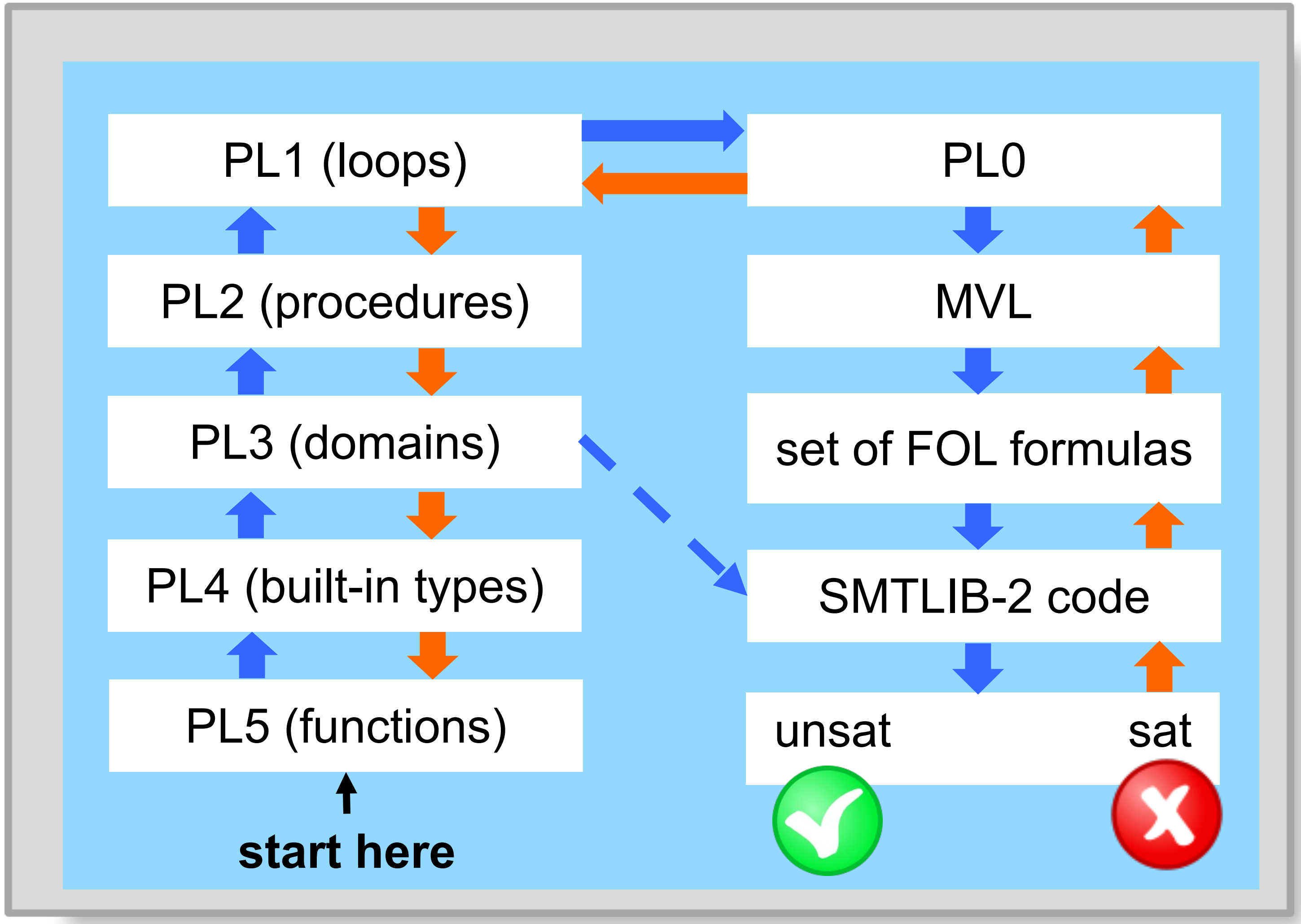
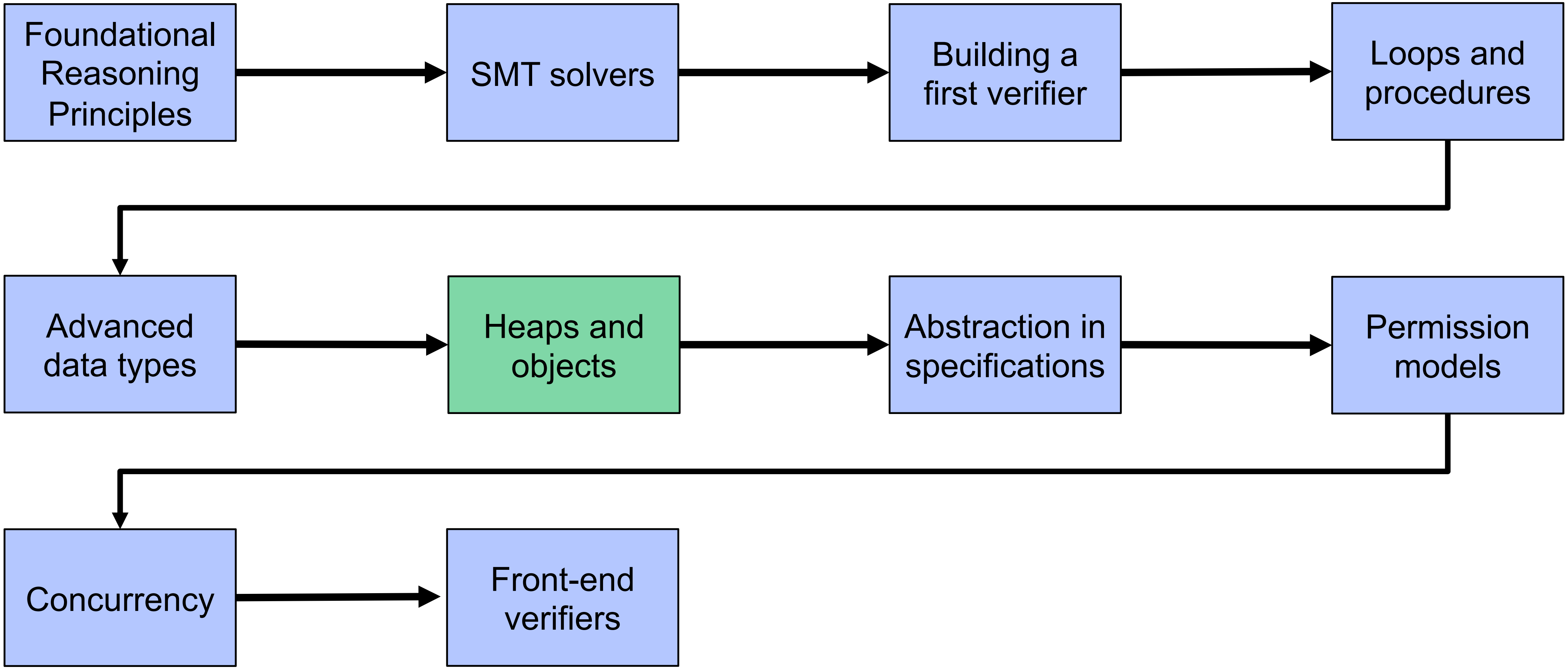
02245 – Module 7

HEAPS AND OBJECTS

Previously...


PL1 (loops)
PL0
PL2 (procedures)
MVL
PL3 (domains)
set of FOL formulas
PL4 (built-in types)
SMTLIB-2 code
PL5 (functions)
unsat
sat
start here

Tentative course outline



Why objects and heap-based data structures?

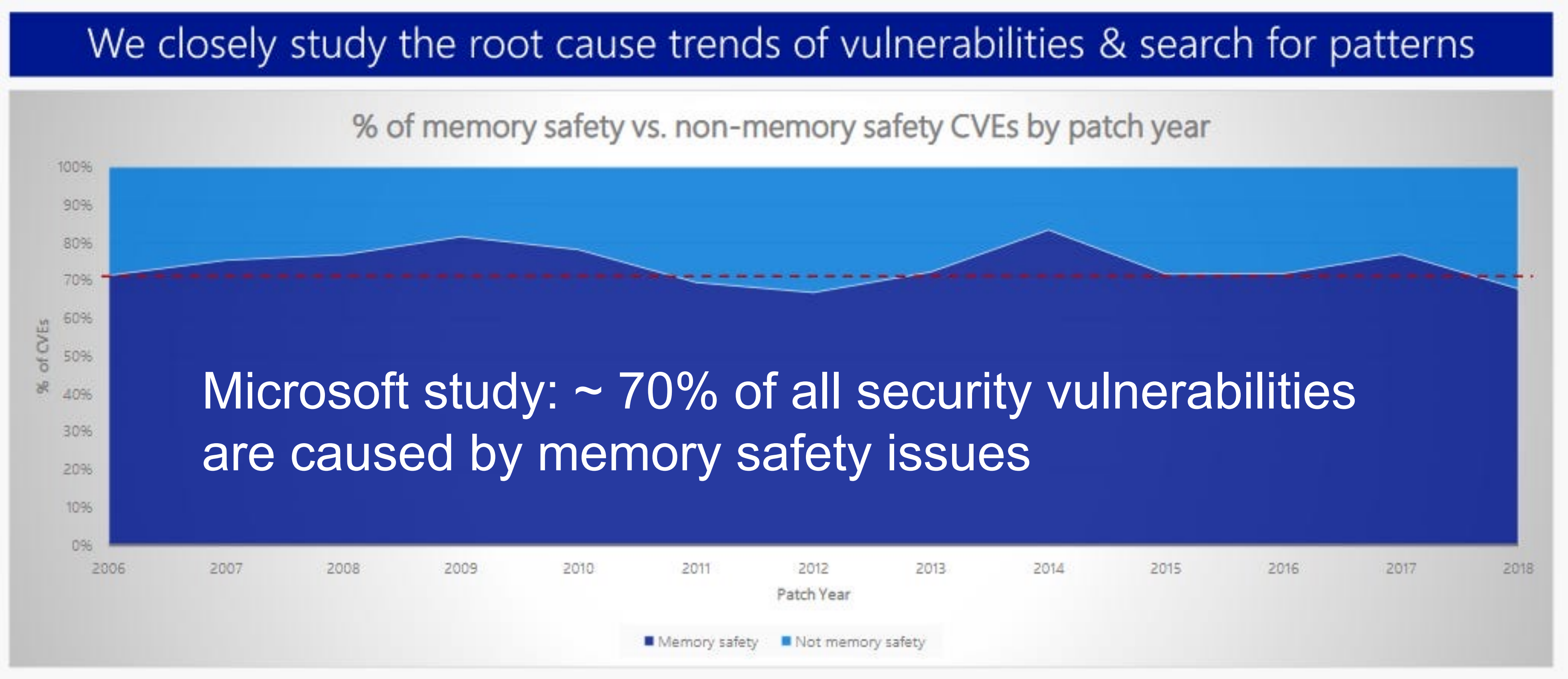
- Static data structures
  - Examples: arrays, all mathematical data structures from module 5
  - Fixed size, stack-allocated
  - Immutable, no memory reuse
  - To update the data structure we create an updated copy

```
// static array A = [0,0,0]
A := cons(3, 0)

// create updated copy
B := set(A, 1, 17)

assert lookup(A, 1) == 0
```

- Dynamic data structures
  - Examples: resizable arrays, linked lists or trees, object graphs, ...
  - Dynamic size, heap-allocated
  - Mutable
  - To up update the data structure, we efficiently change it in-place

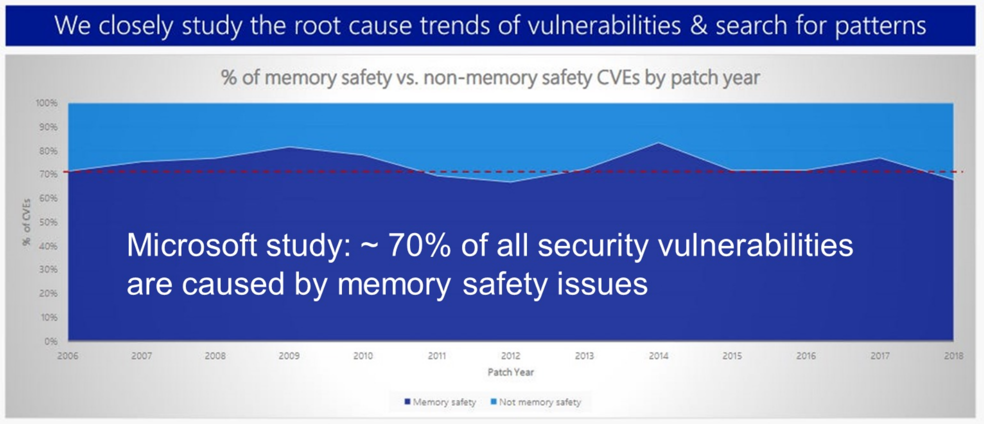
```
// dynamic array A = [0,0,0]
A := new Array(3, 0) // not Viper!

B := A // A, B reference same array
B[1] := 17 // in-place mutation

assert A[1] == 17
```

Why verification of heap-manipulating programs?



Why verification of heap-manipulating programs?

- Memory safety is the absence of errors related to memory accesses
  - dereferencing null-pointers
  - accessing unallocated (heap) memory
  - accessing dangling pointers
  - double-free bugs
  - use-after-free bugs



- Heap-manipulating programs are a prime target for program verification
  - Efficient algorithms need efficient data structures
  - Device drivers, embedded systems, ...

- Same concepts apply to concurrent programs

Objects and the heap

1. Heap model
2. Reasoning about objects and references
3. Ownership and access permissions
4. Encoding

Heap model: an object-based language

➔ 00-heap.vpr

```
field val: Int

method foo() returns (res: Int)
{
  var cell: Ref

  // create object with field val
  cell := new(val)

  cell.val := 5
  res := cell.val
}
```

- A heap is a set of objects
- No classes: each object can have all fields declared in the entire program
  - Type rules of a source language can be encoded
  - Memory consumption is not a concern since programs are not executed
- Objects are accessed via references
  - Field read and update operations
  - No information hiding
- No explicit de-allocation (garbage collector)
  - Conceptually, objects could remain allocated

Extended programming language (PL6)

Declarations

```
D ::= ... | field f: T
```

Fields are declared globally

Types

```
T ::= ... | Ref
```

Only one type of references

Expressions

```
E ::= ... | null | E.f
```

Pre-defined null-reference

Field read expression

Statements

```
S  ::=  ...
      | x := new(f̄)
      | x := new(*)
      | x.f := E
```

Allocation with given fields or with all fields

Field update of **Ref**-typed var.

Objects and the heap

1. Heap model
2. Reasoning about objects and references
3. Ownership and access permissions
4. Encoding

Proof rule for field read

→ `01-field-read.vpr`

- Idea: treat field accesses like variable assignment

Field read

```
———————————————————————————————————————————————
{ E != null && Q[x / E.f] } x := E.f { Q }
```

- Additional well-definedness condition prevents null-dereferencing

```
{ true }
assume r != null && r.val == 5
{ r != null && r.val == 5 }
x := p.val
{ x == 5 }
assert x == 5
{ true }
```

Exercise: Naïve proof rule for field update

- Idea: treat field accesses like variable assignment

Field update

```
---------------------------------------------
{ x != null && Q[x.f / E] } x.f := E { Q }
```

- Additional well-definedness condition prevents null-dereferencing

- The above rule for field update is *unsound*. Give an example that illustrates that.

Solution: Naïve proof rules for field update

→ `02-field-update.vpr`

Field read

```
{ E != null && Q[x / E.f] } x := E.f { Q }
```

Field update

```
{ x != null && Q[x.f / E] } x := E.f { Q }
```

- Aliasing: two references that point to the same object in memory

```
field val: Int

method foo(x: Ref)
{
 // ...
 { true }
  assume x != null && x.val == 5
  { x != null &&
     y != null && x.val == 5 }
  y := x // create an alias
  { x != null
     && y != null && y.val == 5 }
  x.val := 7
  { y != null && y.val == 5 }
  assert y.val == 5
}
```

should not verify!

Field access: candidate proof rules with aliasing

- Idea: reflect potential aliasing in precondition of field-update rule

Field update (informal!)

```
{ x != null && Q[E2.f / (E2==x) ? E : E2.f] } x.f := E { Q }
```

"substitute field access for all objects E2 equal to x"

```
method foo(x: Ref)
{
  var y: Ref
  assume x != null && x.val == 5
  { x != null && x != null && (x==x ? 7 : x.val) == 5 }
  y := x
  { x != null && y != null && (y==x ? 7 : y.val) == 5 }
  x.val := 7
  { y != null && y.val == 5 }
  assert y.val == 5
}
```

- Adjusted rule correctly accounts for aliasing

Shortcomings of candidate proof rule for field update

- Size of assertions grows exponentially in the worst case

```
{ x != null && y != null && t != null && x.val == 5 && y.val == 7 }
{ ... && (x==y ? (t == x ? (...) : (...)) : (x==x ? (...) : (...))) == 7 && ... }
t.val := x.val
{ ... && x==y ? (t==x ? y.val : t.val) : (x==x ? y.val : x.val)) == 7 && ... }
x.val := y.val
{ ... (x==y ? t.val : x.val) == 7 && ... }
y.val := t.val
{ x.val == 7 && ... }
```

- Rule requires explicit syntactic occurrence of field locations in the assertion, but properties may depend on unboundedly many field locations
  - Example: a linked list is sorted (how many `node.next` do we need?)

Reminder: method framing with global variables

- Method specification declares which variables may get modified

```
var x, y: Int

method set(v: Int)
  modifies x
  ensures x == v
{ … }
```

```
y := 7
set(5)

assert x > 0 && y == 7
```

- Frame rule (for any statement S)

Frame rule

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

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

- Encoding

```
y := 7
var x   // havoc vars in mod-clause
assume x == 5
assert x > 0 && y == 7
```

Method framing with heap locations: modifies clause

- Idea: method specification declares which locations may get modified

```
method set(x: Ref, v: Int)
  modifies x.f
  ensures x.f == v
{ … }
```

Frame rule

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

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

- Two ways to adapt the frame rule
  - «variable» means local or global variable, or «field»
  - «variable» means local or global variable, but not «field»

Method framing with heap locations: naïve approach

```
method set(x: Ref, v: Int)
  modifies x.f
  ensures x.f == v
{ … }
```

Frame rule

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

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

«variable» may mean «field»

```
assume y != z
y.f := 7
set(z, 5)
assert y.f == 7
```

- Incomplete: framing is very weak, as information about all objects is lost

«variable» does not mean «field»

```
assume y == r
y.f := 7
set(z, 5)
assert y.f == 7
```

- Unsound: this interpretation of the frame rule ignores aliasing!

Shortcomings of naïve method framing approach

- Sound encoding needs to consider aliasing
  - Inherits shortcomings of candidate rule for field updates
  - Explosion of cases
  - Treatment of assertions that depend on heap locations implicitly

```
y.f := 7
// encoding of set(z, 5)
var tmp: Int
z.f := tmp  // considers aliasing
assume z.f == 5
assert y.f == 7
```

- Many methods modify a statically-unknown set of heap locations
  - Locations cannot be listed explicitly in a modifies clause

```
method sort(list: Ref)
  modifies list.val, list.next.val, list.next.next.val, …
{ … }
```

- Listing modified heap locations violates information hiding

Summary of challenges

Heap data structures pose three major challenges for sequential verification

- Reasoning about aliasing
- Framing, especially for dynamic data structures
- Writing specifications that preserve information hiding

Additional challenges for concurrent programs, e.g., data races

Objects and the heap

1. Heap model
2. Reasoning about objects and references
3. Ownership and access permissions
4. Encoding

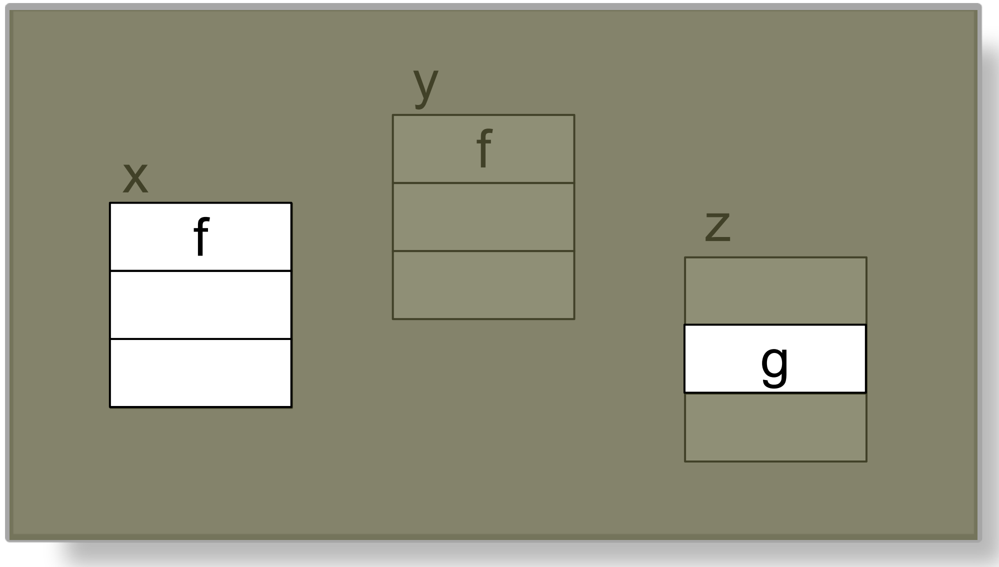
Access permissions

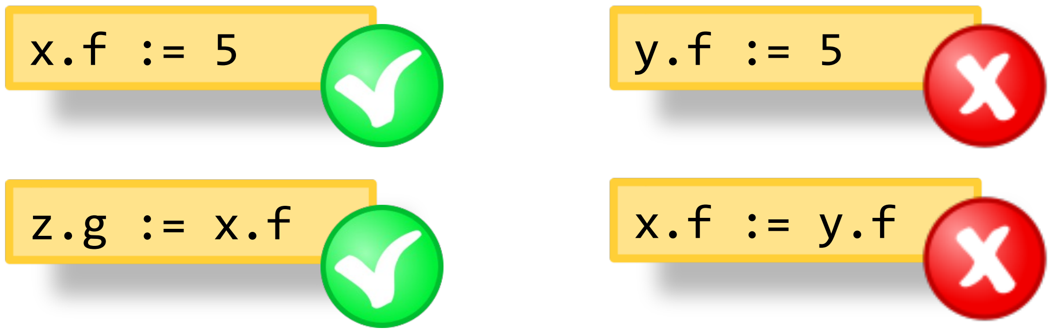
- Associate each heap location with *at most one* permission

- Read or write access to a memory location requires permission

- Permissions are created when the heap location is allocated

- Permissions can be transferred, but not duplicated or forged





Permission assertions

➔ `03-object.vpr`

- Permissions are denoted by access predicates
  - Access predicates are *not* permitted under negations, disjunctions, and on the left of implications
- Predicates may contain both permissions and value constraints
- Predicates must be self-framing, that is, include all permissions to evaluate their heap accesses
- An assertion that does not contain access predicates is called pure or heap independent

Predicates

```
P  ::=  ... | acc(E.f)
```

```
acc(p.f) && p.f > 0
```

```
requires p.f > 0
```

✗

Exercise: swapping the fields of two objects

→ `04-swap.vpr`

- Implement a swap method that exchanges the field values of two objects.
- Specify its functional behavior.
- Write a client method that creates two objects and calls swap on them. Include an assertion to check that swap's specification is strong enough.
- Change your client method such that it calls swap, passing the same reference twice.

```
field f: Int

method swap(a: Ref, b: Ref)
{ … }
```

Solution: swapping the fields of two objects

- Implement a swap method that exchanges the field values of two objects.
- Specify its functional behavior.
- Write a client method that creates two objects and calls swap on them. Include an assertion to check that swap's specification is strong enough.

```
field f: Int

method swap(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
  ensures  acc(a.f) && acc(b.f)
  ensures  a.f == old(b.f) && b.f == old(a.f)
{
  var tmp: Int
  tmp := a.f
  a.f := b.f
  b.f := tmp
}
```

```
method client1()
{
  var x: Ref
  var y: Ref
  x := new(f)  // get permission for f
  y := new(f)
  x.f := 5     // initialize f
  y.f := 7
  swap(x, y)
  assert x.f == 7 && y.f == 5
}
```

Solution: swapping the fields of two objects

- Implement a swap method that exchanges the field values of two objects.
- Specify its functional behavior.
- Change your client method such that it swaps an object with itself.

```
field f: Int

method swap(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
  ensures  acc(a.f) && acc(b.f)
  ensures  a.f == old(b.f) && b.f == old(a.f)
{
  var tmp: Int
  tmp := a.f
  a.f := b.f
  b.f := tmp
}
```

```
method client2()
{
  var x: Ref
  x := new(f)  // get permission for f
  x.f := 5     // initialize f
  swap(x, x)   // precondition violation
}
```

Permission assertions and aliasing

➔ 05-alias.vpr

Reminder:

- There is *at most one* permission for every heap location
- Permissions can be transferred, but not duplicated or forged

If we have two permissions `acc(a.f)` and `acc(b.f)`, can a and b be aliases?

```
field f: Int

method alias(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
{
  a.f := 5
  b.f := 7
  assert a.f == 5
}
```

✔

```
field f: Int

method alias2(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
{
  assert a == b
}
```

✘

➔ How do we justify this?

Permission assertions, more formally

- We extend states to stack-heap pairs $\sigma = (s, h)$

- The stack s: $\mathbf{Var} \rightarrow \mathbf{Value}$ assigns values to variables
  - We used this as the full state state used in all previous classes

- The heap h assigns values to object-field pairs

$$h: \mathbf{Objects} \times \mathbf{Fields} \xrightarrow{\text{finite partial}} \mathbf{Value}$$

  - $dom(h)$ is the set of all object-field pairs for which h is defined

  - $(\text{obj}, f) \in dom(h)$ means we have permission to field f of object obj

Alternative: $permMask: \mathbf{Objects} \times \mathbf{Fields} \xrightarrow{\text{finite partial}} \mathbf{Bool}$

Predicates over extended states

Predicate P	$\mathfrak{I} = (\mathfrak{A}, s, h) \vDash$ P if and only if
`acc(`t`.f)`	$(\mathfrak{I}(t), f) \in dom(h)$
$t_1 = t_2$	$\mathfrak{I}(t_1) = \mathfrak{I}(t_2)$
$R(t_1, \ldots, t_n)$	$(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n)) \in R^{\mathfrak{A}}$
Q ∧ **R**	$\mathfrak{I} \vDash$ **Q** and $\mathfrak{I} \vDash$ **R**
Q ⇒ **R**	If $\mathfrak{I} \vDash$ **Q**, then $\mathfrak{I} \vDash$ **R**
$\exists x$: **T** (**Q**)	For some v ∈ $\mathbf{T}^{\mathfrak{A}}$, $\mathfrak{I}[x := \mathrm{v}] \vDash$ **Q**
$\forall x$: **T** (**Q**)	For all v ∈ $\mathbf{T}^{\mathfrak{A}}$, $\mathfrak{I}[x := \mathrm{v}] \vDash$ **Q**

$\mathfrak{I}(t)$ is the value obtained from evaluating term t in interpretation $\mathfrak{I}$

Examples:

$\mathfrak{I}(x) = s(x)$

$\mathfrak{I}(x + 17) = s(x) +^{\mathfrak{A}} 17^{\mathfrak{A}}$

$\mathfrak{I}(x.f) = h(s(x), f)$

$\mathfrak{I}(x.f.g) = h(h(s(x), f), g)$

- Self-framing predicates are always well-defined

Assume s`(a) ==` s`(b)` and h`(a.f) ==` s`(c)`

Does $\mathfrak{I} = (\mathfrak{A}, s, h) \vDash$ `acc(a.f)` ∧ `acc(b.f)` ∧ `b.f == c` hold?

Handling aliasing

- Problem: having permissions `a.f` and `b.f` should mean `a` and `b` are no aliases

- We introduce a new connective: the separating conjunction `P ∗ Q`
  - `P ∗ Q` partitions the heap h into two chunks
  - Every permission assertion `acc(E.f)` is evaluated in its own heap chunk
  - All other predictes are evaluated in the full heap

$(\mathfrak{A}, s, h) \models_h$ `acc(x.f) * acc(y.g) ?`

$(\mathfrak{A}, s, h) \models_{h1}$ `acc(x.f)`

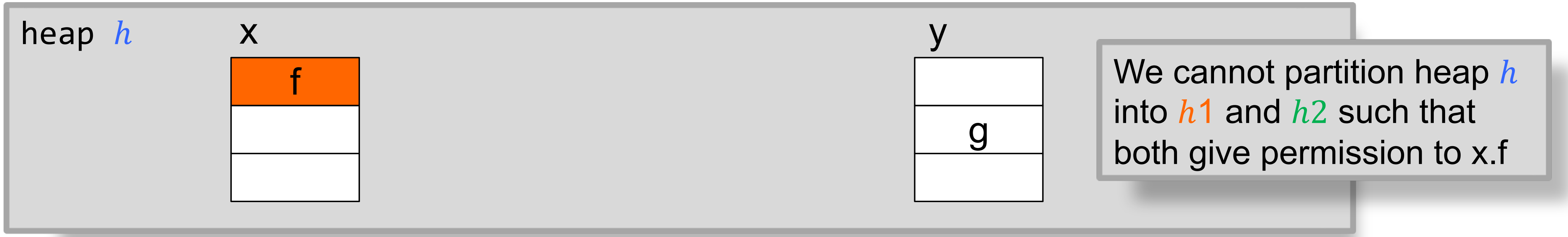
$(\mathfrak{A}, s, h) \models_{h2}$ `acc(y.g)`

Handling aliasing

- Problem: having permissions `a.f` and `b.f` should mean `a` and `b` are no aliases

- We introduce a new connective: the separating conjunction $P * Q$
  - $P * Q$ partitions the heap h into two chunks
  - Every permission assertion `acc(E.f)` is evaluated in its own heap chunk
  - All other predictes are evaluated in the full heap

$(\mathfrak{A}, s, h) \vDash_h$ `acc(x.f) * acc(x.f)` ?



$(\mathfrak{A}, s, h) \vDash_{h1}$ `acc(x.f)`

$(\mathfrak{A}, s, h) \vDash_{h2}$ `acc(x.f)`

Predicates with separating conjunction

Predicate P	$\mathfrak{I} = (\mathfrak{A}, s, h) \vDash_{h'}$ **P if and only if**
`acc(t.f)`	$(\mathfrak{I}(t), f) \in dom(h')$
$t_1 = t_2$	$\mathfrak{I}(t_1) = \mathfrak{I}(t_2)$
$R(t_1, \ldots, t_n)$	$(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n)) \in R^{\mathfrak{A}}$
Q $\wedge$ **R**	$\mathfrak{I} \vDash_{h'}$ **Q** and $\mathfrak{I} \vDash_{h'}$ **R**
Q $*$ **R**	exists partition of h' into $h1$, $h2$ such that $\mathfrak{I} \vDash_{h1}$ **Q** and $\mathfrak{I} \vDash_{h2}$ **R**
...	...

evaluate access permissions in current heap chunk h' (initially h)

split current heap chunk into two

- **Q** $*$ **R** and **Q** $\wedge$ **R** are equivalent if **Q** and **R** are pure
- Holding permission to `x.f` and `y.f` implies that x and y are no aliases

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

Separating Conjunction in Viper

➔ 04-swap.vpr
➔ 05-alias.vpr

- Viper's && is the separating conjunction $*$

- Viper has no ordinary conjunction $\wedge$

- **`Q`** $*$ **`R`** and **`Q`** $\wedge$ **`R`** are equivalent if **`Q`** and **`R`** are pure (heap independent)

- For the call `swap(x, x)`, the precondition is equivalent to false

```
method swap(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
```

Exercise

- Reconsider the method on the right.

- Change the precondition such that we can call the method by passing both aliasing references and non-aliasing references to it as arguments without violating the precondition.

- Does the assertion still hold?
  Why (not)?

```
method alias(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
{
  a.f := 5
  b.f := 7
  assert a.f == 5
}
```

Solution

➔ 06-alias.vpr

- Reconsider the method on the right.
- Change the precondition such that we can call the method by passing both aliasing references and non-aliasing references to it as arguments without violating the precondition.
- Does the assertion still hold?
  Why (not)?

```
method alias(a: Ref, b: Ref)
  requires acc(a.f)
    && (b != a ==> acc(b.f))
{
  a.f := 5
  b.f := 7
  assert a.f == 5
}
```

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
- Writing specifications that preserve information hiding

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

Field access: proof rules with permissions

Field read

```
─────────────────────────────────────────────────────────
{ acc(E.f) * P[x / E.f] } x := E.f { acc(E.f) * P }
```

Field update

```
─────────────────────────────────────────────────────────────────
{ acc(x.f) * x.f == N } x.f := E { acc(x.f) * x.f == E[x.f / N] }
```

- Each field access requires (and preserves) the corresponding permission
- Permission to a location implies that the receiver is non-null
- Substitution with logical variable N in the field-update rule is needed to handle occurrences of `x.f` inside E (e.g., `x.f := x.f + 1`)

Framing

Frame rule

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

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

Unsound if S assigns to heap locations constrained by R

Framing

Frame rule

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

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

- The frame R must be self-framing
  - If heap locations constrained by R are disjoint from those modified by S, R is preserved
  - Otherwise, the precondition is equivalent to false (the triple holds trivially)
- Example

$$\frac{\overline{\{\ \mathsf{acc}(x.f) * x.f = N\ \}\ \ x.f := 5\ \ \{\ \mathsf{acc}(x.f) * x.f = 5\ \}}}{\{\ \mathsf{acc}(x.f) * x.f = N * \mathsf{acc}(y.f) * y.f = 7\ \}\ \ x.f := 5\ \ \{\ \mathsf{acc}(x.f) * x.f = 5 * \mathsf{acc}(y.f) * y.f = 7\ \}}$$

Framing (cont'd)

- The following proof derives an incorrect triple. Why is it not a valid proof?

$$\dfrac{\overline{\{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = \mathtt{N} \,\}\ \ \mathtt{x.f} := 5\ \ \{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 \,\}}}{\{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = \mathtt{N} * \mathtt{x.f} = 1 \,\}\ \ \mathtt{x.f} := 5\ \ \{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 * \mathtt{x.f} = 1 \,\}}$$

- Recall that the frame must be self-framing, which is not the case here
- Making the frame self-framing yields a valid (but vacuous) proof

$$\dfrac{\overline{\{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = \mathtt{N} \,\}\ \ \mathtt{x.f} := 5\ \ \{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 \,\}}}{\{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = \mathtt{N} * \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 1 \,\}\ \ \mathtt{x.f} := 5\ \ \{\, \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 * \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 1 \,\}}$$

Framing for method calls

```
method set(p: Ref, v: Int)
  requires acc(p.f)
  ensures  acc(p.f) && p.f == v
{
  p.f := v
}
```

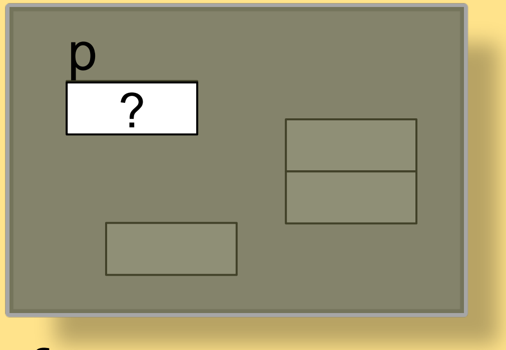
```
// assume we have acc(x.f) && acc(y.f)
assume y.f == 7
set(x, 5)
assert x.f == 5 && y.f == 7
```

$$\dfrac{\dfrac{\{\,\mathbf{acc}(\mathtt{p.f})\,\}\ \ \mathbf{method}\ \mathtt{set(p,\ v)}\ \ \{\,\mathbf{acc}(\mathtt{p.f}) * \mathtt{p.f} = \mathtt{v}\,\}}{\{\,\mathbf{acc}(\mathtt{x.f})\,\}\ \ \mathtt{set(x,\ 5)}\ \ \{\,\mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5\,\}}}{\{\,\mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\,\}\ \ \mathtt{set(x,\ 5)}\ \ \{\,\mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\,\}}$$

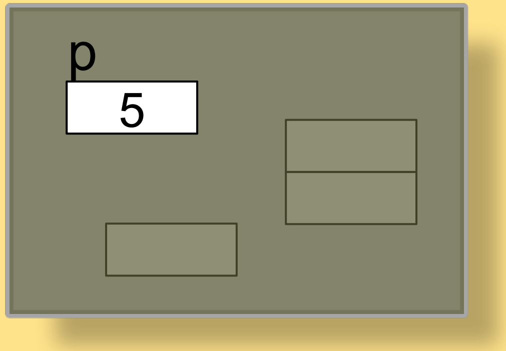
- Frame rule enables framing without modifies clauses
- A method may modify only heap locations to which it has permission

Permission transfer

```
method set(p: Ref, v: Int)
  requires acc(p.f)
  ensures  acc(p.f) && p.f == v
{
```

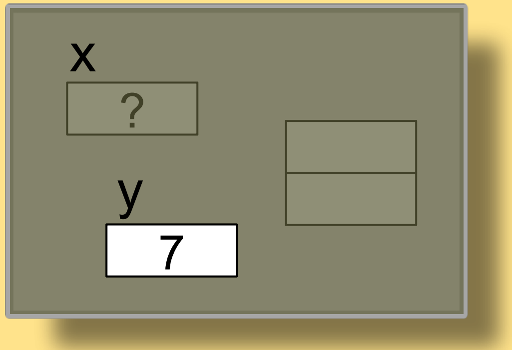



```
  p.f := v
```




```
}
```

```
// assume we have acc(x.f) && acc(y.f)
assume x.f == 2 && y.f == 7
```




```
set(x, 5)
```

Framing!

```
assert x.f == 5 && y.f == 7
```

Permission transfer for method calls

$$\dfrac{\dfrac{\{\,\mathbf{acc}(\mathtt{p.f})\,\}\ \mathbf{method}\ \mathtt{set(p,\ v)}\ \{\,\mathbf{acc}(\mathtt{p.f}) * \mathtt{p.f} = \mathtt{v}\,\}}{\{\,\mathbf{acc}(\mathtt{x.f})\,\}\ \mathtt{set(x,\ 5)}\ \{\,\mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5\,\}}}{\{\,\mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\,\}\ \mathtt{set(x,\ 5)}\ \{\,\mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5 * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\,\}}$$

- Permissions are held by method executions or loop iterations
- Calling a method transfers permissions from the caller to the callee (according to the method precondition)
- Returning from a method transfers permissions from the callee to the caller (according to the method postcondition)
- Residual permissions are framed around the call

Framing for loops

```
// assume we have acc(x.f) && acc(y.f)
x.f := 0
y.f := 7
while (x.f < 10)
  invariant acc(x.f)
{
  x.f := x.f + 1
}
assert y.f == 7
```

$$\dfrac{\dfrac{\{\ \mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} < 10\ \}\ \mathtt{x.f := x.f + 1}\ \{\ \mathbf{acc}(\mathtt{x.f})\ \}}{\{\ \mathbf{acc}(\mathtt{x.f})\ \}\ \mathbf{while}(\mathtt{x.f < 10})\ \{\ \ldots\ \}\ \{\ \mathbf{acc}(\mathtt{x.f}) * \neg \mathtt{x.f} < 10\ \}}}{\{\ \mathbf{acc}(\mathtt{x.f}) * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\ \}\ \mathbf{while}(\mathtt{x.f < 10})\ \{\ \ldots\ \}\ \{\ \mathbf{acc}(\mathtt{x.f}) * \neg \mathtt{x.f} < 10 * \mathbf{acc}(\mathtt{y.f}) * \mathtt{y.f} = 7\ \}}$$

Permission transfer for loops

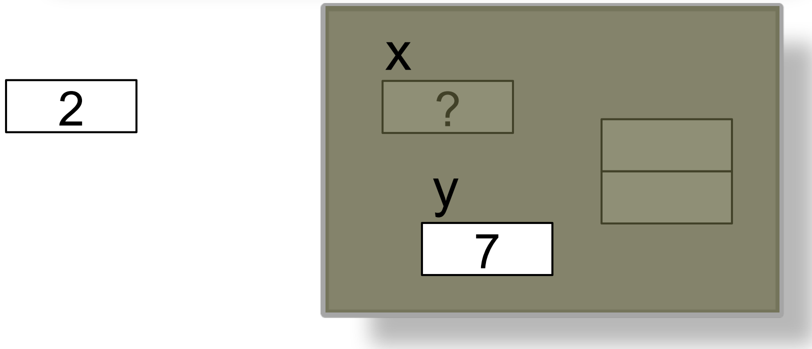
$$\frac{\dfrac{\{\,\texttt{acc}(\texttt{x.f}) * \texttt{x.f} < 10\,\}\ \ \texttt{x.f} := \texttt{x.f + 1}\ \ \{\,\texttt{acc}(\texttt{x.f})\,\}}{\{\,\texttt{acc}(\texttt{x.f})\,\}\ \ \texttt{while(x.f < 10) \{ ... \}}\ \ \{\,\texttt{acc}(\texttt{x.f}) * \neg \texttt{x.f} < 10\,\}}}{\{\,\texttt{acc}(\texttt{x.f}) * \texttt{acc}(\texttt{y.f}) * \texttt{y.f} = 7\,\}\ \ \texttt{while(x.f < 10) \{ ... \}}\ \ \{\,\texttt{acc}(\texttt{x.f}) * \neg \texttt{x.f} < 10 * \texttt{acc}(\texttt{y.f}) * \texttt{y.f} = 7\,\}}$$

- Permissions are held by method executions or loop iterations
- Entering a loop transfers permissions from the enclosing context to the loop (according to the loop invariant)
- Leaving a loop transfers permissions from the loop to the enclosing context (according to the loop invariant)
- Residual permissions are framed around the loop

Permission transfer: inhale and exhale operations

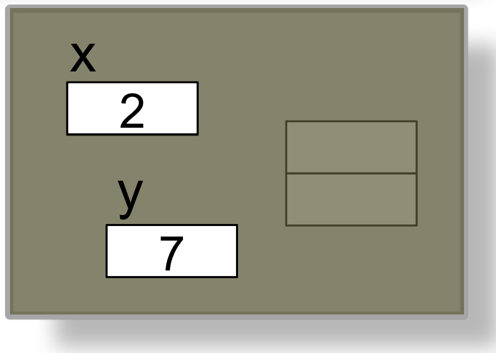
- `inhale` P means:
  - obtain all permissions required by assertion P
  - assume all logical constraints

```
inhale acc(x.f) && x.f == 2
```



- `exhale` P means:
  - assert all logical constraints
  - check and remove all permissions required by assertion P
  - havoc any locations to which all permission is lost

```
exhale acc(x.f) && x.f == 2
```



Encoding of method bodies and calls

```
method foo() returns (…)
  requires P
  ensures  Q
{ S }
```

```
x := foo()
```

- Encoding without heap and globals
  - Body
    ```
    assume P
    // encoding of S
    assert Q
    ```
  - Call
    ```
    assert P[…]
    havoc x
    assume Q[…]
    ```
- Encoding with heap
  - Body
    ```
    inhale P
    // encoding of S
    exhale Q
    ```
  - Call
    ```
    exhale P[…]
    havoc x
    inhale Q[…]
    ```
- **`inhale`** and **`exhale`** are permission-aware analogues of **`assume`** and **`assert`**

Exercise: definition of exhale

- **`exhale`** `P` means:
  - assert all logical constraints
  - check and remove all permissions required by `P`
  - havoc (reset) any locations to which all permission is lost

- Write an example that demonstrates that omitting the havoc from the exhale encoding would be unsound

Solution: definition of exhale

- **`exhale`** `P` means:
  - assert all logical constraints
  - check and remove all permissions required by `P`
  - havoc (reset) any locations to which all permission is lost

```
method reset(p: Ref)
  requires acc(p.f)
  ensures  acc(p.f) && p.f == 0
{
  p.f := 0
}
```

- Write an example that demonstrates that omitting the havoc from the exhale encoding would be unsound

```
var x: Ref
inhale acc(x.f) && x.f == 5
reset(x)
assert x.f == 5   // would verify
assert false      // would verify
```

- Before the call, we have `acc(x.f) && x.f == 5`
- exhale without havoc would retain `x.f == 5`
- We assume `x.f == 0` through the method call
- We reached a contradiction!

Encoding of loops

```
while(b)
  invariant I
{ S }
```

- Reminder: encoding without heap

```
assert I
havoc targets
assume I
if(*) {
  assume b
  // encoding of S
  assert I
  assume false
} else {
  assume !b
}
```

- Encoding with heap

```
exhale I
havoc targets
inhale I
if(*) {
  assume b
  // encoding of S
  exhale I
  assume false
} else {
  assume !b
}
```

Encoding of allocation

- new-expression specifies the relevant fields

```
x := new(f, g)
```

- Encoding chooses an arbitrary reference and inhales permissions to relevant fields

```
var x: Ref
inhale acc(x.f) && acc(x.g)
```

- Incomplete information about freshness of new object

```
x := new(f)
y := new(f)
assert x != y
```

```
method foo(y: Ref)
{
  var x: Ref
  x := new(f)
  assert x != y
}
```

Exercise: working with permissions

➔ `07-account.vpr`

- Implement, specify, and verify a class for bank accounts with the following methods:
  - `create` returns a fresh account with initial balance 0
  - `deposit` deposits a non-negative amount to an account
  - `transfer` transfers a non-negative amount between two accounts
  - Account balances are integers.
- Verify the client program on the right.

```
method client()
{
  var x: Ref
  var y: Ref
  var z: Ref
  x := create()
  y := create()
  z := create()
  deposit(x, 100)
  deposit(y, 200)
  deposit(z, 300)
  transfer(x, y, 100)
  assert x.bal == 0
  assert y.bal == 300
  assert z.bal == 300
}
```

Solution: working with permissions

```
field bal: Int

method create() returns (n: Ref)
  ensures acc(n.bal) && n.bal == 0
{
  n := new(bal)
  n.bal := 0
}

method deposit(to: Ref, amount: Int)
  requires acc(to.bal) && 0 <= amount
  ensures  acc(to.bal)
  ensures  to.bal == old(to.bal) + amount
{
  to.bal := to.bal + amount
}
```

```
method transfer(
            from: Ref, to: Ref, amount: Int)
  requires acc(from.bal) && acc(to.bal)
  requires 0 <= amount && amount <= from.bal
  ensures  acc(from.bal) && acc(to.bal)
  ensures  to.bal == old(to.bal) + amount
  ensures  from.bal == old(from.bal) - amount
{
  to.bal := to.bal + amount
  from.bal := from.bal - amount
}
```

Verifying memory safety

➔ 08-memory-safety.vpr

- Memory safety is the absence of errors related to memory accesses, such as, null-pointer dereferencing, access to un-allocated memory, dangling pointers, out-of-bounds accesses, double free, etc.

- Using permissions, Viper verifies memory safety by default

```
var x: Ref
x.f := 5
```
✗

```
var x: Ref
x := null
x.f := 5
```
✗

See module 8 for arrays

```
method free(p: Ref)
  requires acc(p.f)
```

model de-allocation via method call

```
free(x)
x.f := 5
```
✗

```
free(x)
free(x)
```
✗

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

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

Objects and the heap

1. Heap model
2. Reasoning about objects and references
3. Ownership and access permissions
4. Encoding

Heaps

- Encode references and fields

```
type Ref              // type for references
const null: Ref       // null references

type Field T          // polymorphic type for field names
```

```
field f: Int
field g: Ref
```

```
const f: Field int
const g: Field Ref
```

- Heaps map references and field names to values

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

- Represent the program heap as one global variable

```
var Heap: HeapType
```

Permissions and field access

- Permissions are tracked in a global permission mask

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

- Convention: ¬`Mask[null, f]` for all fields `f`

- Field access

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

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

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

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

  - Field access requires permission!

Inhale

→ `09-inhale.vpr`

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

`inhale E`	`assume [[E]]`	[[.]] encoding
`inhale acc(E.f)`	`assume ¬Mask[ [[E]], f ]` `Mask[ [[E]], f ] := true`	Reaching more than full permission goes to magic
`inhale E => P`	`if([[E]]) { [[inhale P]] }`	
`inhale P && Q`	`[[inhale P]]; [[inhale Q]]`	Separating conjunction: add sum of permissions

- The encoding also asserts that E is well-defined (omitted here)

Exhale (1st attempt)

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

`exhale` `E`	**`assert`** `[[E]]`
`exhale acc``(E.f)`	**`assert`** `Mask[ [[E]], f ]` `Mask[ [[E]], f ] :=` **`false`** **`havoc`** `Heap[ [[E]], f ]`
`exhale` `E => P`	**`if`**`([[E]]) { [[`**`exhale`**` P]] }`
`exhale` `P && Q`	`[[`**`exhale`**` P]]; [[`**`exhale`**` Q]]`

havoc e.g. by assigning to a fresh variable

Separating conjunction: remove sum of permissions

- The encoding also asserts that `E` is well-defined (omitted here)

Example

```
inhale acc(x.f) && x.f == 5
```

```
assume ¬Mask[x,f]
Mask[x,f] := true

assert Mask[x,f]  // well-definedness check
assume Heap[x,f] == 5
```

✓

```
exhale acc(x.f) && x.f == 5
```

```
assert Mask[x,f]
Mask[x,f] := false
havoc Heap[x,f]

assert Mask[x,f]  // well-definedness check
assert Heap[x,f] == 5
```

✗

Exhale (fixed)

- Conceptually, permissions should be removed after checking logical constraints
- Adapt encoding
  - Check well-definedness against mask at the beginning of the exhale
  - Delay havoc until the end of the exhale

```
exhale P
```

```
var oldMask: MaskType
var newHeap: HeapType
oldMask := Mask
[[exhale P]]        // like before, but no havoc and with
                       well-definedness check on oldMask
assume forall y,g :: Mask[y,g] ==> newHeap[y,g] == Heap[y,g]
Heap := newHeap  // effectively havocs all locations to which
                   permission was lost
```

Exercise: encoding of exhale

- Encode the operation (on paper, not using Viper)

```
exhale acc(x.f) && x.f == 5
```

with the fixed encoding.

Solution: encoding of exhale

- Encode the operation (on paper, not using Viper)

  exhale acc(x.f) && x.f == 5

  with the fixed encoding.

```
var oldMask: MaskType
var newHeap: HeapType
oldMask := Mask

assert Mask[x,f]
Mask[x,f] := false

assert oldMask[x,f]  // well-definedness check
assert Heap[x,f] == 5

assume forall y,g :: Mask[y,g] ==> newHeap[y,g] == Heap[y,g]
Heap := newHeap
```

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, but see next module

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

- Permissions are an excellent basis, but see later