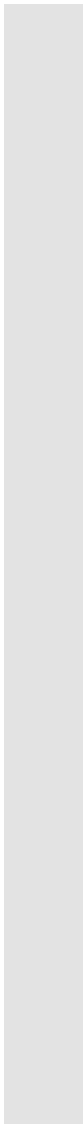


Reasoning about allocation

K. Rustan M. Leino

Amazon Web Services

20 May 2025
IFIP WG 2.3 meeting
Athens, Greece



Modeling allocation, and why

new is new

```
procedure P(y: C, s: set<C>)  $\triangleq$   
  var x := new C  
  assert x  $\neq$  y  $\wedge$  x  $\neq$  y.next  $\wedge$  x  $\notin$  s
```

```
class C  $\triangleq$   
  var data: int  
  var next: C
```

Modification in a heap with shared mutable objects

```
procedure P(y: C)  
  modifies {y}
```

$\triangleq$

```
y.data := 10
```

Modification justified
by **modifies** clause



modifies s

SOURCE PROGRAM

....>

MEANING

ensures $\forall r \cdot H[r] = H'[r] \quad \vee \quad r \in s$

Modification in a heap with shared objects

```
procedure P(y: C)  
  modifies {y}
```

$\triangleq$

```
y.data := 10
```

```
var x := new C
```

```
x.data := 12
```

Modification justified
by **modifies** clause

Modification justified
by freshness

```
modifies s
```

....>

```
ensures  $\forall r \cdot H[r] = H'[r] \quad \vee \quad r \in s \quad \vee \quad r \text{ fresh}$ 
```

Possible encoding: alloc field

$x := \text{new } C$

....>

$x : | \neg H[x].\text{alloc}$

$H[x].\text{alloc} := \text{true}$

$\text{modifies } s$

....>

$\text{ensures } \forall r \cdot H[r] = H'[r] \quad \vee \quad r \in s \quad \vee \quad \neg H[r].\text{alloc}$

Monotonicity

```
procedure P()  $\triangleq$   
  Q(...)  
  var x := new C  
  x.data := 12
```

Modification to be
justified by freshness.
But how?

....>

```
ensures  $\forall r \cdot H[r].\text{alloc} \Rightarrow H'[r].\text{alloc}$ 
```

For each object, the number of applications of Monotonicity is linear in the number of preceding calls

Monotonicity (again)

```
procedure P(y: C)  
  modifies {y}
```

$\triangleq$

```
  y.data := 10
```

```
  Q(...)
```

```
  assert y.data = 10
```



Needs to know y was allocated
before call to Q()

Closure properties

```
axiom  $\forall H: \text{Heap}, c: C \cdot$   
  IsAlloc(c, H) = H[c].alloc
```

Closure properties

axiom $\forall H: \text{Heap}, c: C \cdot$
 $\text{GoodHeap}(H) \Rightarrow \text{IsAlloc}(c, H) = H[c].\text{alloc}$

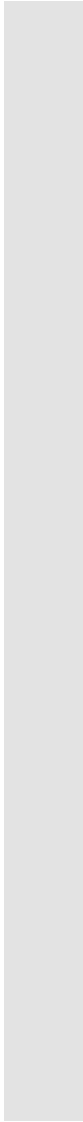
axiom $\forall H: \text{Heap}, s: \text{set}\langle C \rangle, c: C \cdot$
 $\text{GoodHeap}(H) \wedge \text{IsAlloc}(s, H) \wedge c \in s$
 $\Rightarrow \text{IsAlloc}(c, H)$

axiom $\forall H: \text{Heap}, r: C \cdot$
 $\text{GoodHeap}(H) \wedge \text{IsAlloc}(r, H)$
 $\Rightarrow \text{IsAlloc}(H[r].\text{next}, H)$

axiom $\forall H: \text{Heap}, w: W \cdot$
 $\text{GoodHeap}(H) \wedge \text{IsAlloc}(z, H)$
 $\Rightarrow \text{IsAlloc}(F(w), H)$

Summary so far

- Meaning of **modifies**
- Monotonicity
- Closure properties



Functions

Functions take an implicit heap argument

```
function F(a: A): B
```

....>

```
function F(H: Heap, a: A): B
```

Framing for functions

```
procedure P(a: A, c: C)  
  modifies {c}
```

$\triangleq$

```
var b := F(a)
```

```
c.data := 10
```

```
assert b = F(a)
```

Requires knowing
whether or not F
depends on state of c

```
function F(a: A): B  
  reads s
```

Frame axiom

```
function F(a: A): B  
  reads s
```

....>

```
axiom  $\forall H: \text{Heap}, H': \text{Heap}, a: A \cdot$ 
```

$H \approx_s H'$

$\Rightarrow F(H, a) = F(H', a)$

(This axiom gives rise to a number of instantiations proportional to the square of the number of heaps. 🙄)

`const` fields don't affect `reads` clause

```
function G(c: C): int  
  reads {}
```

$\triangleq$

```
3 * c.cdata
```

```
class C  $\triangleq$   
  var data: int  
  var next: C  
  const cdata: int
```




Quantification

Memory Safety Principle

A memory-safe programming language guarantees

- No dangling pointers
- No wild pointers (e.g., non-allocated object references)

So, does

$$\forall c: C \cdot c.data \% 2 = 0$$

implicitly range over just allocated objects?

Design A: $\forall c: C \cdot H[c].alloc \Rightarrow H[c].data \% 2 = 0$

Design B: $\forall c: C \cdot H[c].data \% 2 = 0$

Design A:
Follow the
Memory
Safety
Principle for
quantifier
ranges

```
function DataIsEven(): bool  
  reads ...
```

$\triangleq$

$$\forall c: C \cdot c.cd\text{ata} \% 2 = 0$$

- Implicitly reads a `ll oc` field of all (allocated and unallocated) `C` objects, so **reads** clause would need to include all those `C` objects
- Or: ...at least all unallocated `C` objects (thanks to Monotonicity)
- Or: In a function, don't allow quantifications over types that involve references
- Or: Let user opt out of having a frame axiom for this function

Design B:
Ignore the
Memory
Safety
Principle for
quantifier
ranges

```
function DataIsEven(): bool  
  reads {}
```

$\triangleq$

```
 $\forall c: C \cdot c.cd\text{ata} \% 2 = 0$ 
```

- **reads** clause can be empty
- (Without further user-supplied antecedents, like $c \in s$) quantifications/comprehensions can be used only in ghost contexts (i.e., non-compiled code, like specifications)
- But...

Design B:
Ignore the
Memory
Safety
Principle for
quantifier
ranges

```
ghost var s := { c: C | P(c) }  
if s ≠ {} {  
  ghost var g :| g ∈ s  
}
```

- So, ghost variables can hold non-allocated objects!
- Then:

```
var c := new C(data := 20)  
ghost var g := c  
for i := 100 to 200  
  invariant g.data % 2 == 0 // error: proof fails  
{  
  c := new C(data := i)  
  if i % 2 == 0 {  
    g := c;  
  }  
}
```

(additional slides)



Allocation times

(as were used in ESC/Java, 1997-2001)

Every object has a premeditated *allocation time*

```
type Time = int
function atime(o: object): Time

// Global variables
var H: Heap
var time: Time
```

```
c := new C
```

....>

```
c :| time ≤ atime(c)
time :| atime(c) < time
```

- Monotonicity: **ensures** $\text{time} \leq \text{time}'$
One linear chain per path, not per path per object
- Freshness: $\text{time} \leq \text{atime}(c)$

Closure properties (allocation time)

```
function GoodHeap(H: Heap, t: Time): bool
```

```
axiom  $\forall H: \text{Heap}, t: \text{Time}, c: C.$ 
```

```
    GoodHeap(H, t)  $\Rightarrow$  IsAlloc(c, H) = H[c].alloc
```

```
axiom  $\forall H: \text{Heap}, t: \text{Time}, s: \text{set}\langle C \rangle, c: C.$ 
```

```
    GoodHeap(H, t)  $\wedge$  IsAlloc(s, H)  $\wedge$   $c \in s$   
     $\Rightarrow$  IsAlloc(c, H)
```

```
axiom  $\forall H: \text{Heap}, t: \text{Time}, r: C.$ 
```

```
    GoodHeap(H, t)  $\wedge$  IsAlloc(r, H)  
     $\Rightarrow$  IsAlloc(H[r].next, H)
```

```
axiom  $\forall H: \text{Heap}, t: \text{Time}, w: W.$ 
```

```
    GoodHeap(H, t)  $\wedge$  IsAlloc(z, H)  
     $\Rightarrow$  IsAlloc(F(w), H)
```

Summary

- Keeping track of allocation is needed
 - to know that `new` returns something new
 - to justify modifications of newly allocated objects
- Encodings
 - `alloc` field
 - allocation time
- Quantifications/comprehensions over reference types
 - follow Memory Safety Principle
 - Limitations on using quantifications in function bodies
 - ignore Memory Safety Principle
 - Surprising behavior for ghost variables