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>) ≜
  var x := new C
  assert x ≠ y  ∧  x ≠ y.next  ∧  x ∉ s
```

```
class C ≜
  var data: int
  var next: C
```

Modification in a heap with shared mutable objects

```
procedure P(y: C)
  modifies {y}
≙
  y.data := 10
```

Modification justified by `modifies` clause

```
modifies s
```

SOURCE PROGRAM

⇢

MEANING

```
ensures ∀r· H[r] = H′[r]  ∨  r ∈ s
```

Modification in a heap with shared objects

```
procedure P(y: C)
  modifies {y}
≙
  y.data := 10
  var x := new C
  x.data := 12
```

Modification justified by `modifies` clause

Modification justified by freshness

```
modifies s
⇢
ensures ∀r· H[r] = H′[r]  ∨  r ∈ s  ∨  r fresh
```

Possible encoding: `alloc` field

```
x := new C
```

⇢

```
x :| ¬H[x].alloc
H[x].alloc := true
```

```
modifies s
```

⇢

```
ensures ∀r· H[r] = H′[r]  ∨  r ∈ s  ∨  ¬H[r].alloc
```

Monotonicity

```
procedure P() ≜
  Q(…)
  var x := new C
  x.data := 12
```

Modification to be justified by freshness. But how?

⋯→

```
ensures ∀r· H[r].alloc  ⟹  H′[r].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}
≜
  y.data := 10
  Q(…)
  assert y.data = 10
```

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

Closure properties

```
axiom ∀ H: Heap, c: C·
  IsAlloc(c, H) = H[c].alloc
```

Closure properties

```
axiom ∀ H: Heap, c: C ·
  GoodHeap(H) ⟹   IsAlloc(c, H) = H[c].alloc

axiom ∀ H: Heap, s: set<C>, c: C ·
  GoodHeap(H) ∧ IsAlloc(s, H) ∧ c ∈ s
  ⟹  IsAlloc(c, H)

axiom ∀ H: Heap, r: C ·
  GoodHeap(H) ∧ IsAlloc(r, H)
  ⟹  IsAlloc(H[r].next, H)

axiom ∀ H: Heap, w: W ·
  GoodHeap(H) ∧ IsAlloc(z, H)
  ⟹  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}
≜
  var b := F(a)
  c.data := 10
  assert b = F(a)


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

Requires knowing whether or not F depends on state of c

Frame axiom

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

⋯→

```
axiom ∀ H: Heap, H′: Heap, a: A ·
  H ≈s H′
  ⟹  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 {}
≜
  3 * c.cdata
```

```
class C ≜
  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

```
∀c: C·  c.data % 2 = 0
```

implicitly range over just allocated objects?

Design A:
```
∀c: C·  H[c].alloc ⟹  H[c].data % 2 = 0
```

Design B:
```
∀c: C·  H[c].data % 2 = 0
```

Design A: Follow the Memory Safety Principle for quantifier ranges

```
function DataIsEven(): bool
  reads …
≜
  ∀c: C·  c.cdata % 2 = 0
```

- Implicitly reads `alloc` 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 {}
≜
  ∀c: C·  c.cdata % 2 = 0
```

- `reads` clause can be empty
- (Without further user-supplied antecedents, like `c ∈ 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 time ≤ time'`
  One linear chain per path, not per path per object
- Freshness: `time ≤ atime(c)`

Closure properties (allocation time)

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

```
axiom ∀ H: Heap, t: Time, c: C ·
  GoodHeap(H, t) ⟹   IsAlloc(c, H) = H[c].alloc

axiom ∀ H: Heap, t: Time, s: set<C>, c: C ·
  GoodHeap(H, t) ∧ IsAlloc(s, H) ∧ c ∈ s
  ⟹  IsAlloc(c, H)

axiom ∀ H: Heap, t: Time, r: C ·
  GoodHeap(H, t) ∧ IsAlloc(r, H)
  ⟹  IsAlloc(H[r].next, H)

axiom ∀ H: Heap, t: Time, w: W ·
  GoodHeap(H, t) ∧ IsAlloc(z, H)
  ⟹  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