Garbage Collection in Bertinoro (Alternative form)

Reggio

A Region System for Controlling Memory Management Costs Enforced by an Ownership Types System


Ellen Arvidsson


Elias Castegren


Tobias Wrigstad


James Noble


Sophia Drossopoulou


Matthew Parkinson


Sylvan Clebsch

Wanted: Controllable and Predictable Memory Management Costs

- Mix-and-Match Memory Management

  Ability to freely mix multiple ways to manage memory in the same program

- Incremental Memory Management

  Only pay for memory management in a subset of the heap

- Concurrent Memory Management

  Program work should not have to wait for GC work, and conversely

- Zero-Copy Ownership Transfer

  Moving obligation to manage memory between different parts of a program should not come with a cost

- Safe Concurrency

  Freedom from data races

with Regions

Wanted: Controllable and Predictable Memory Management Costs

- Mix-and-Match Memory Management

  Ability to freely mix multiple ways to manage memory in the same program — at region granularity

- Incremental Memory Management

  Only pay for memory management in a subset of the heap — in the active region granularity

- Concurrent Memory Management

  Program work should not have to wait for GC work, and conversely — program thread opening region can choose to GC

- Zero-Copy Ownership Transfer

  i.e. change region topology

  Moving obligation to manage memory between different parts of a program should not come with a cost

- Safe Concurrency

  Freedom from data races

if a thread can reference an object O, it can safely access O synchronously at no synchronisation cost

Choose your Preferred Memory Management Strategy (out of these three)

- **Arena allocation**

  Group objects with same life time, a region is like a lifetime

  (Cheaply) deallocate lots of objects

  Bump-pointer allocation

- **Reference counting**

  Record interest in an object: while interest > 0, keep alive

  Minimise floating garbage

  Deal with cycles e.g. through trial deletion(?)

- **Tracing GC**

  Scan memory from roots to determine determine liveness

  Free unused memory, currently not using a moving collector

ISMM 2019

snmalloc: A Message Passing Allocator

Paul Liétar
Microsoft Research, UK
paul@lietar.net

Theodore Butler*
Drexel University, USA
theodore.j.butler@drexel.edu

Sylvan Clebsch
Microsoft Research, UK
Sylvan.Clebsch@microsoft.com

Sophia Drossopoulou
Imperial College London, UK
s.drossopoulou@imperial.ac.uk

Juliana Franco
Microsoft Research, UK
Juliana.Franco@microsoft.com

Matthew J. Parkinson
Microsoft Research, UK
mattpark@microsoft.com

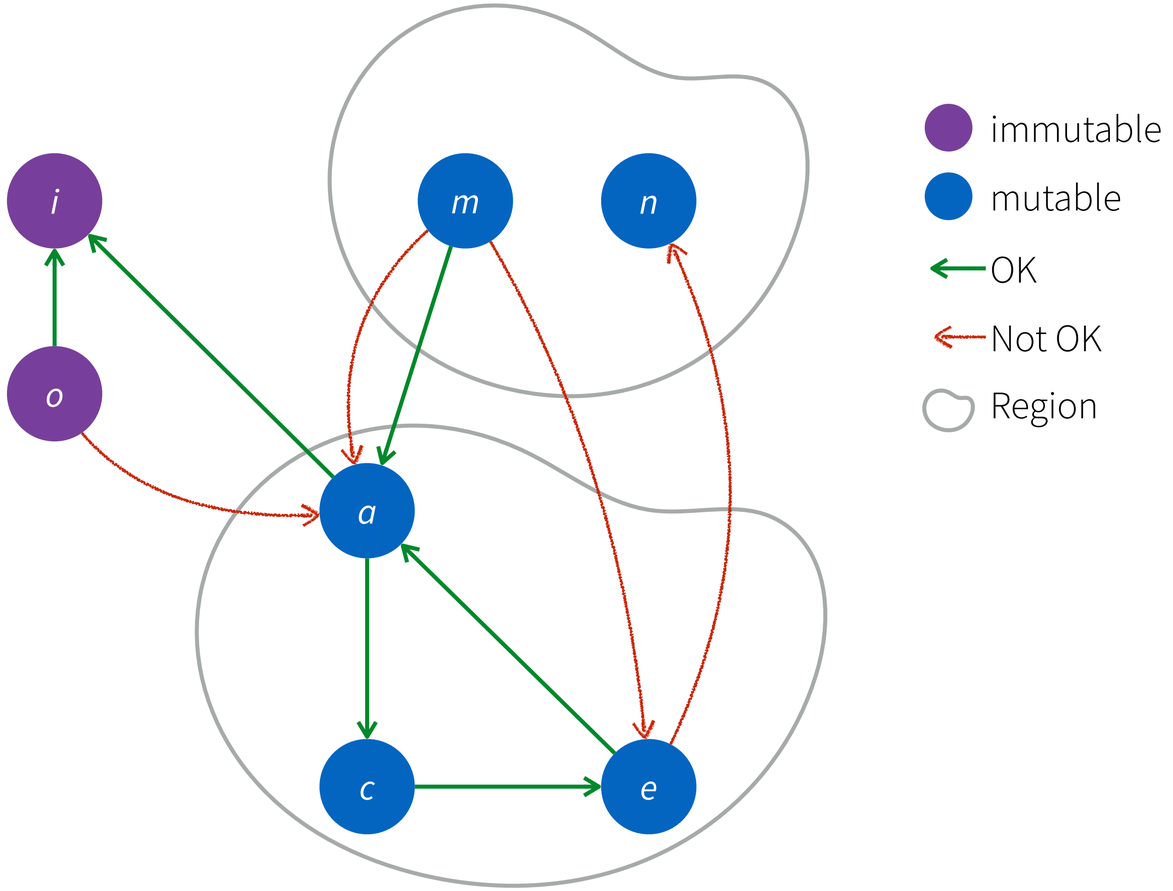
Alex Shamis
Microsoft Research, UK
Imperial College London, UK
alexsha@microsoft.com

Christoph M. Wintersteiger
Microsoft Research, UK
cwinter@microsoft.com

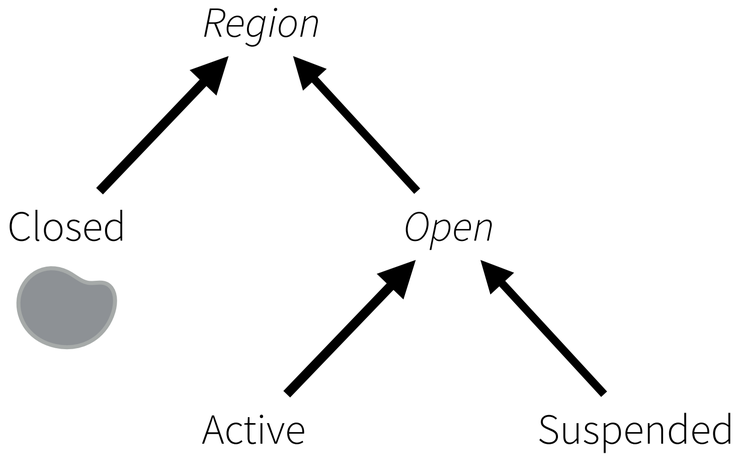
David Chisnall
Microsoft Research, UK
David.Chisnall@microsoft.com

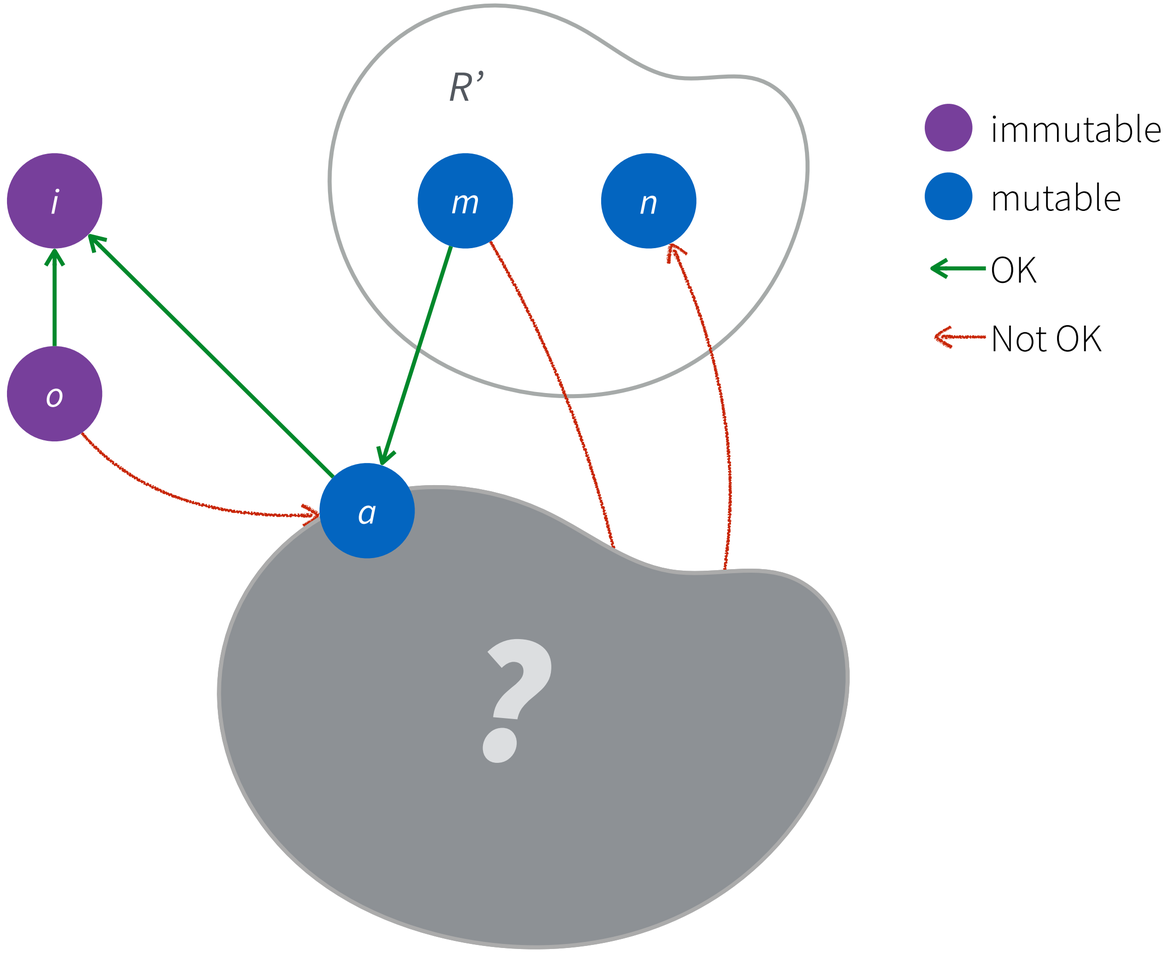
Key Concepts

- We are dividing the *mutable heap* into regions
- Mutable objects
- Immutable objects
- Regions
- Bridge objects
- External uniqueness
- Temporary immutability (coming up)

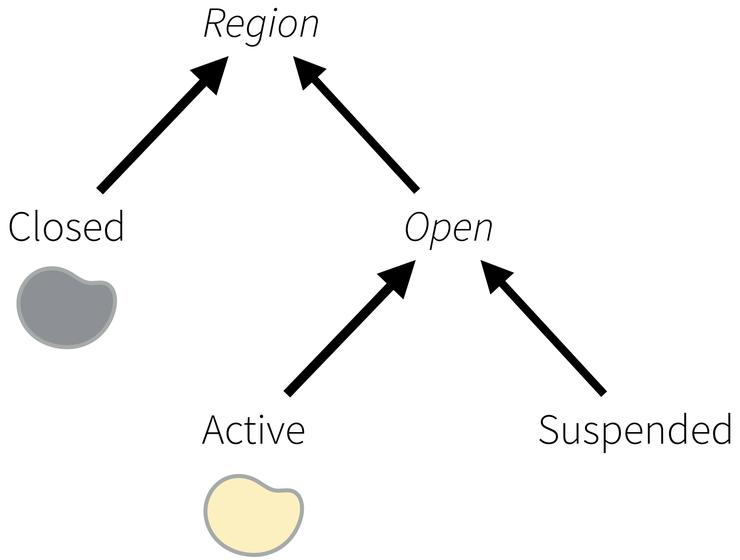


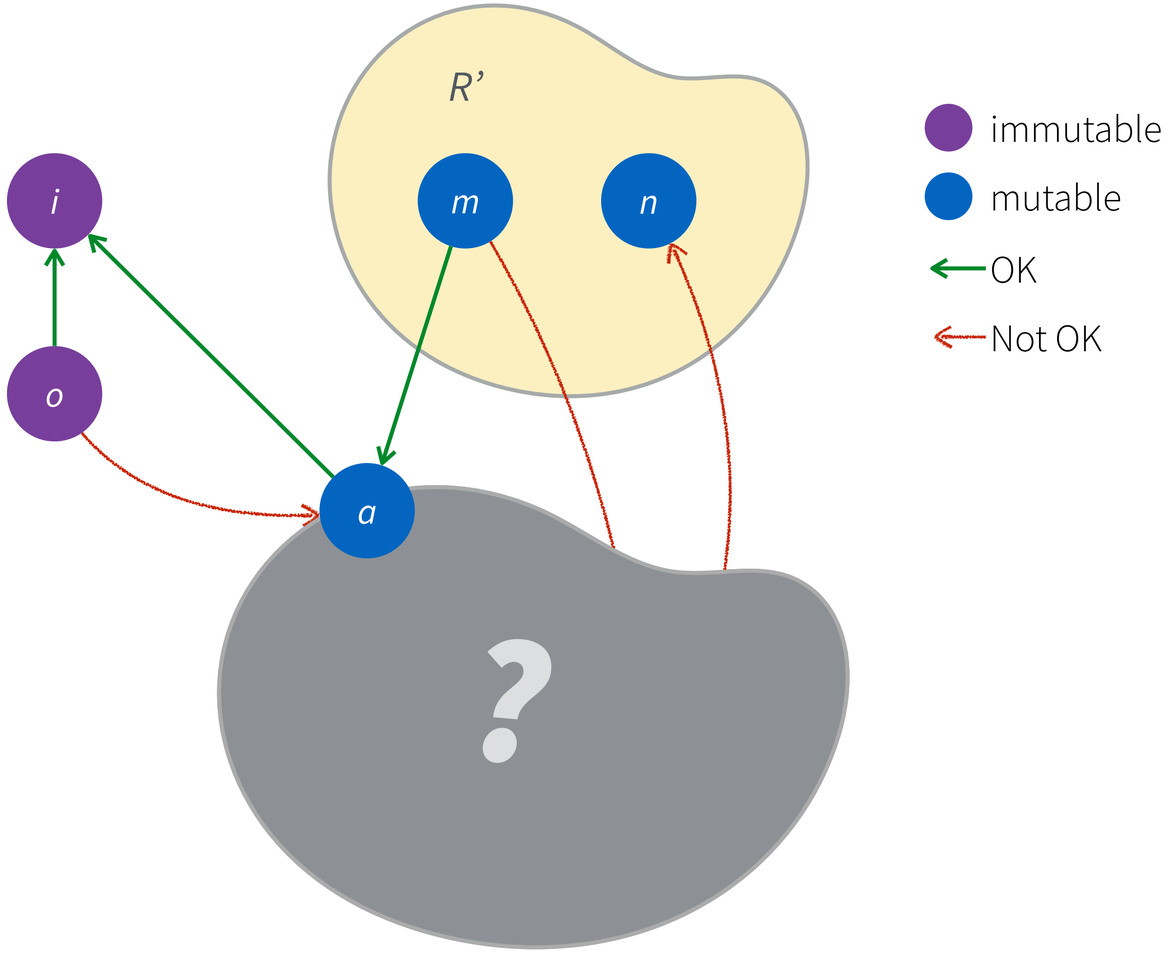
Region States: closed regions' objects are inaccessible (except ptr to bridge obj)





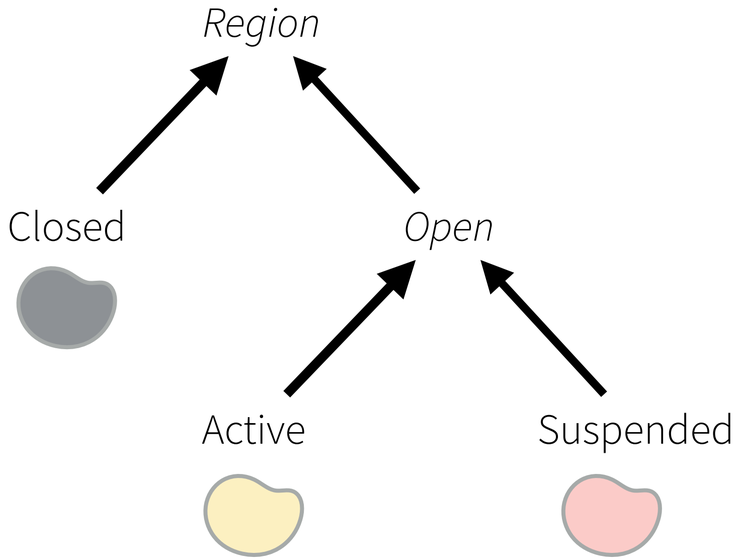
Region States: the active regions' objects are accessible and mutable

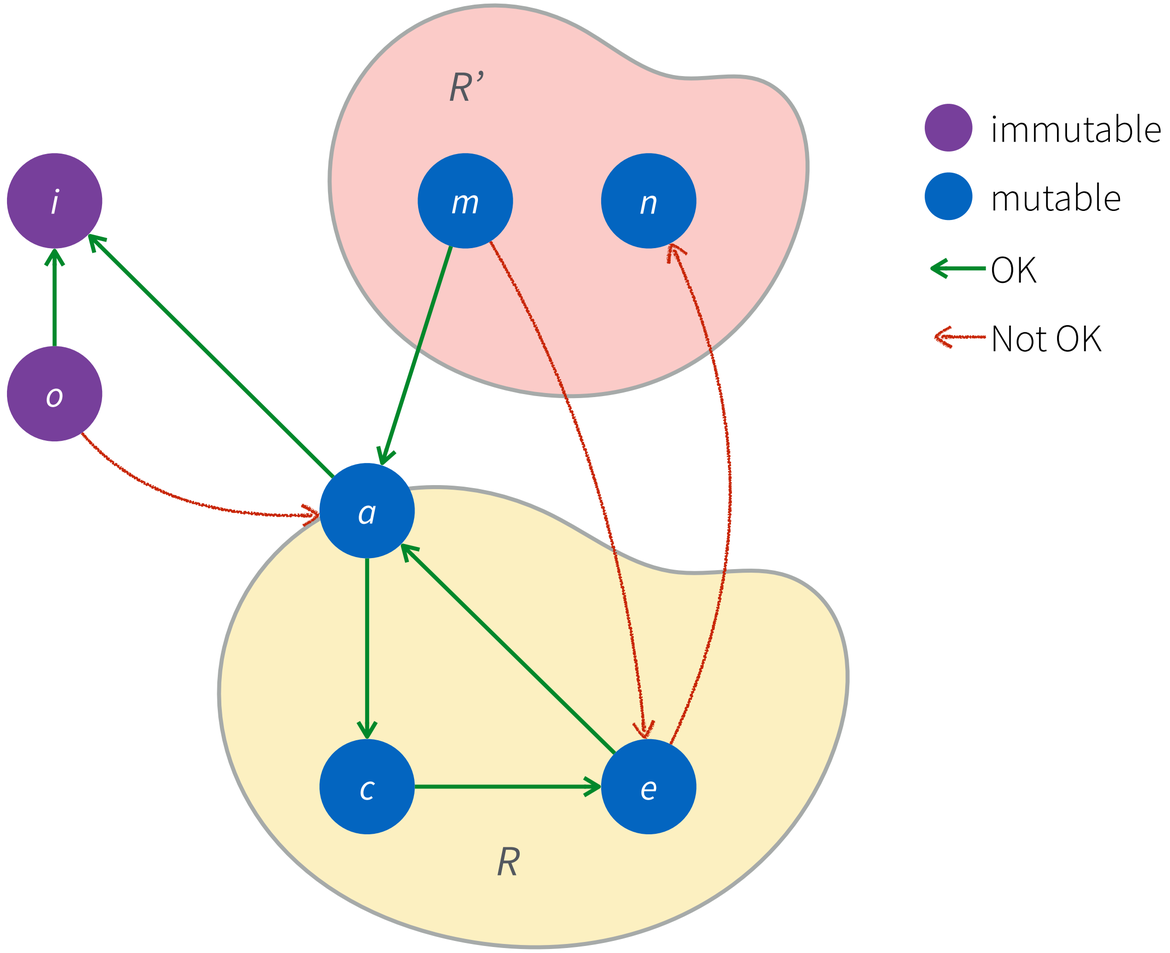




Note: all allocations happen in the active region

Region States: suspended regions' objects are temporarily immutable



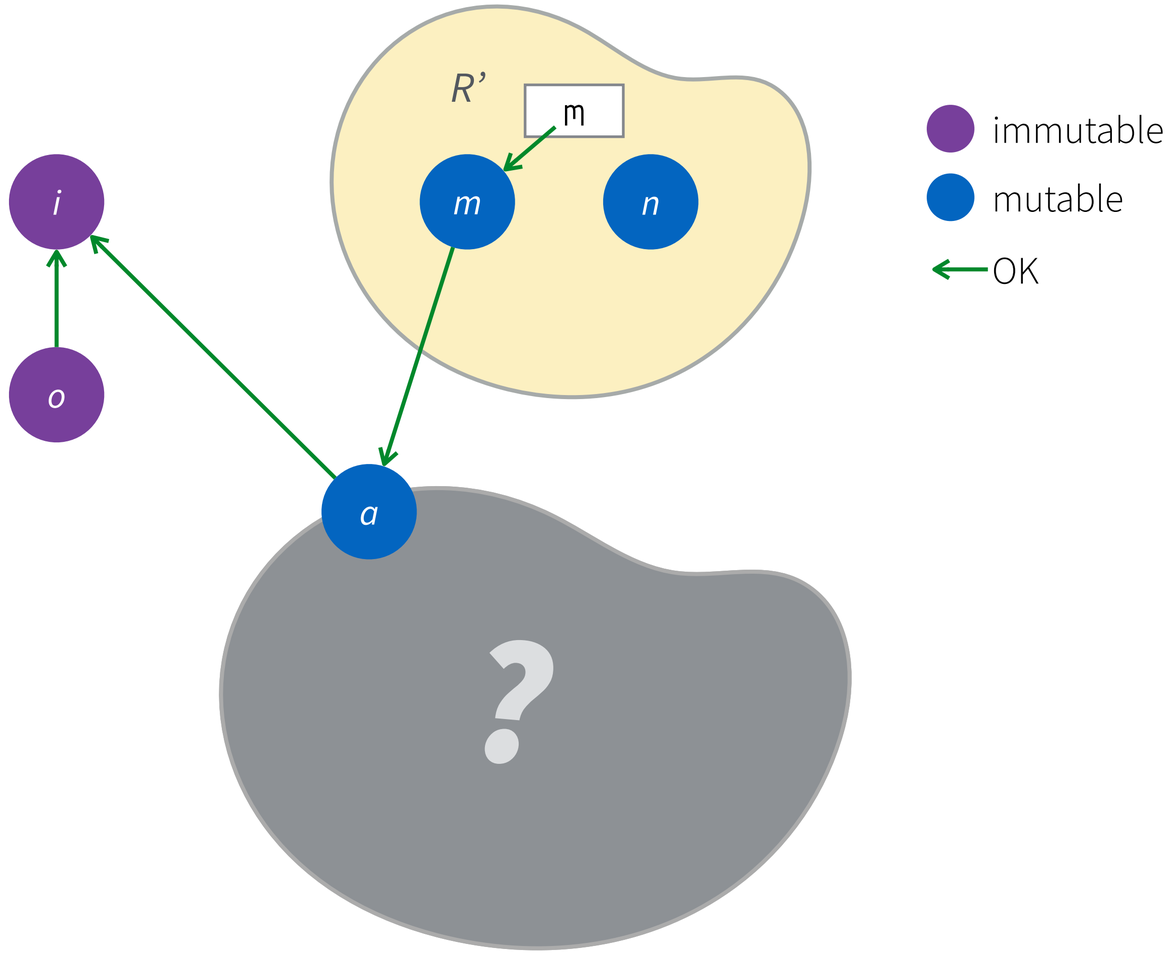


To access R, we must open it — this suspends R'

Explicitly moving the window of mutability from *R'* to *R*

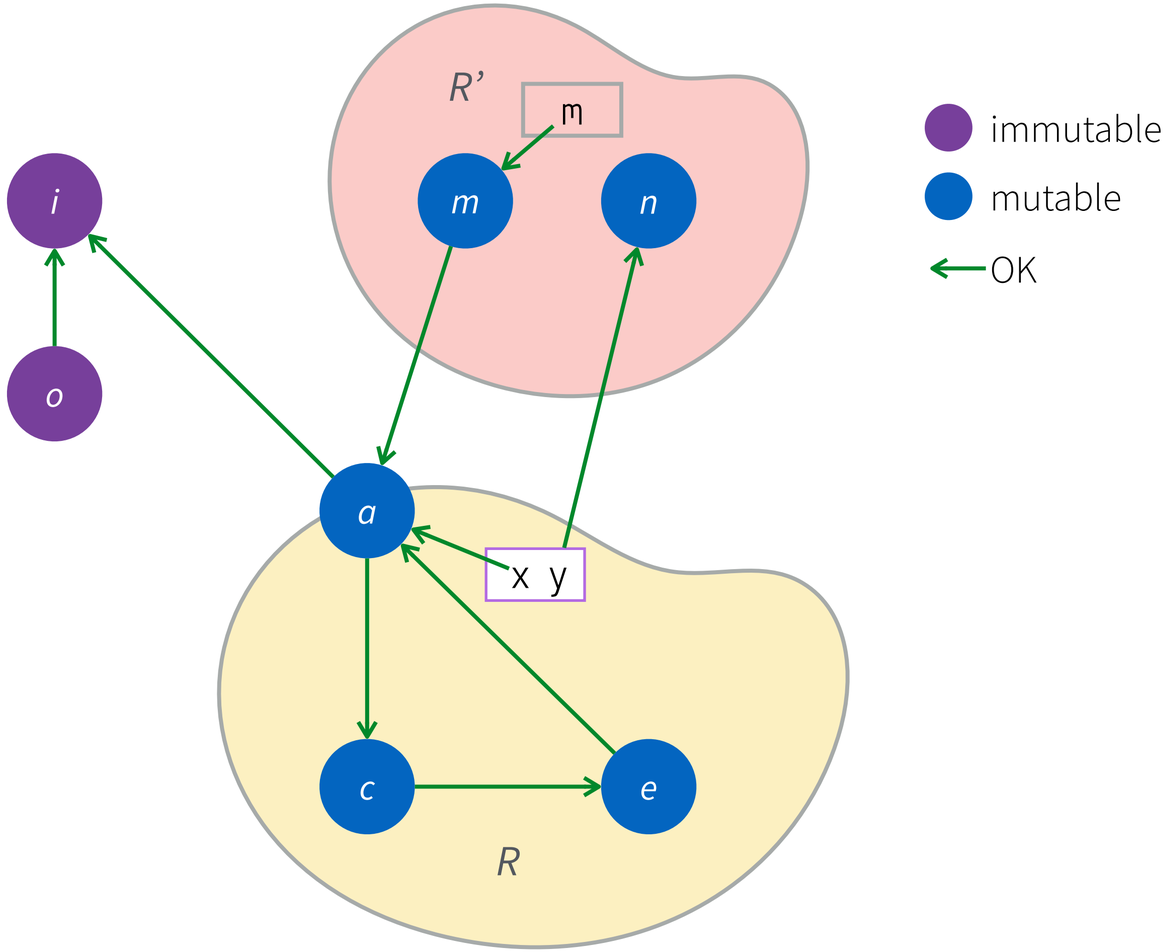
```
// Assume we are in R'
enter m.f x => {
  // Now we are in R
  …

}
```



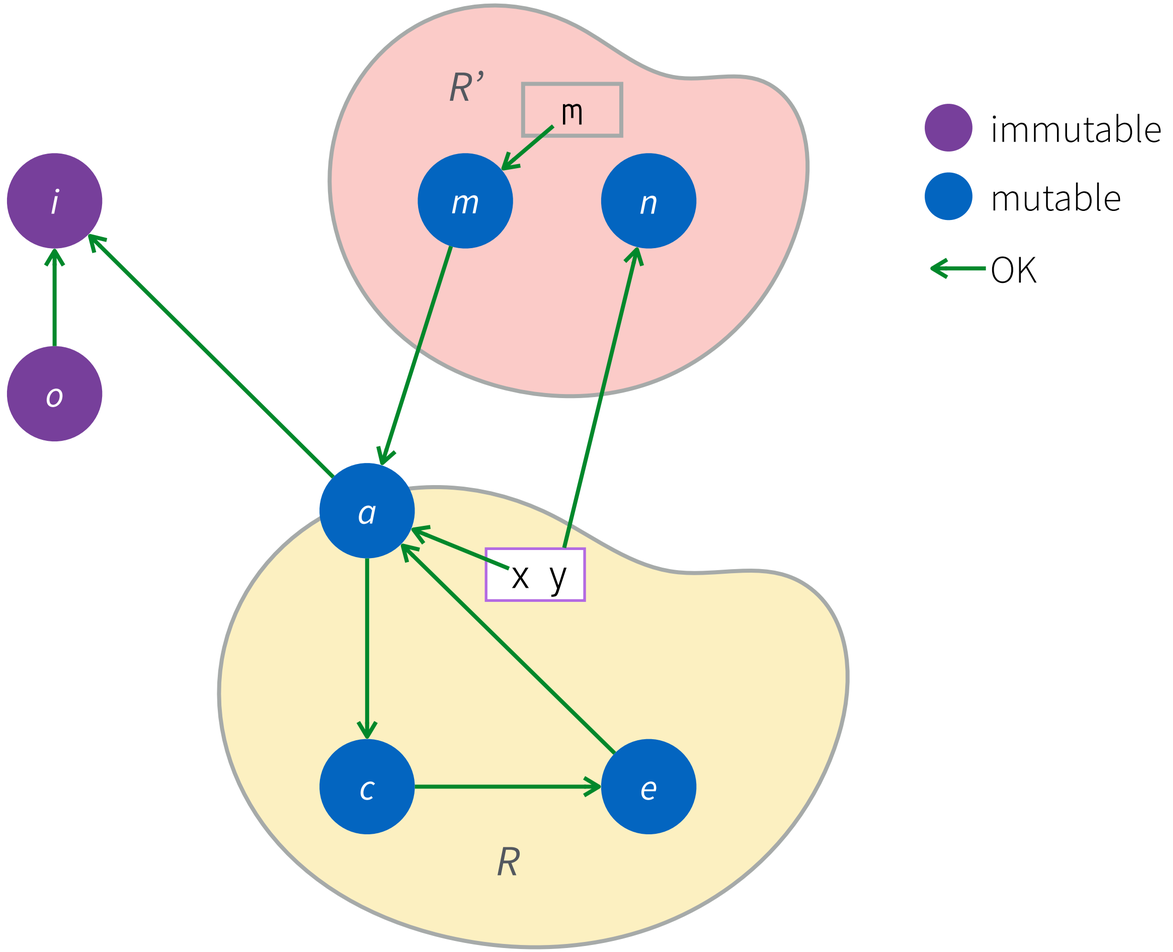
Explicitly moving the window of mutability from *R'* to *R*

```
// Assume we are in R'
enter m.f x => {
  // Now we are in R
  // m = … illegal
  …
}
```



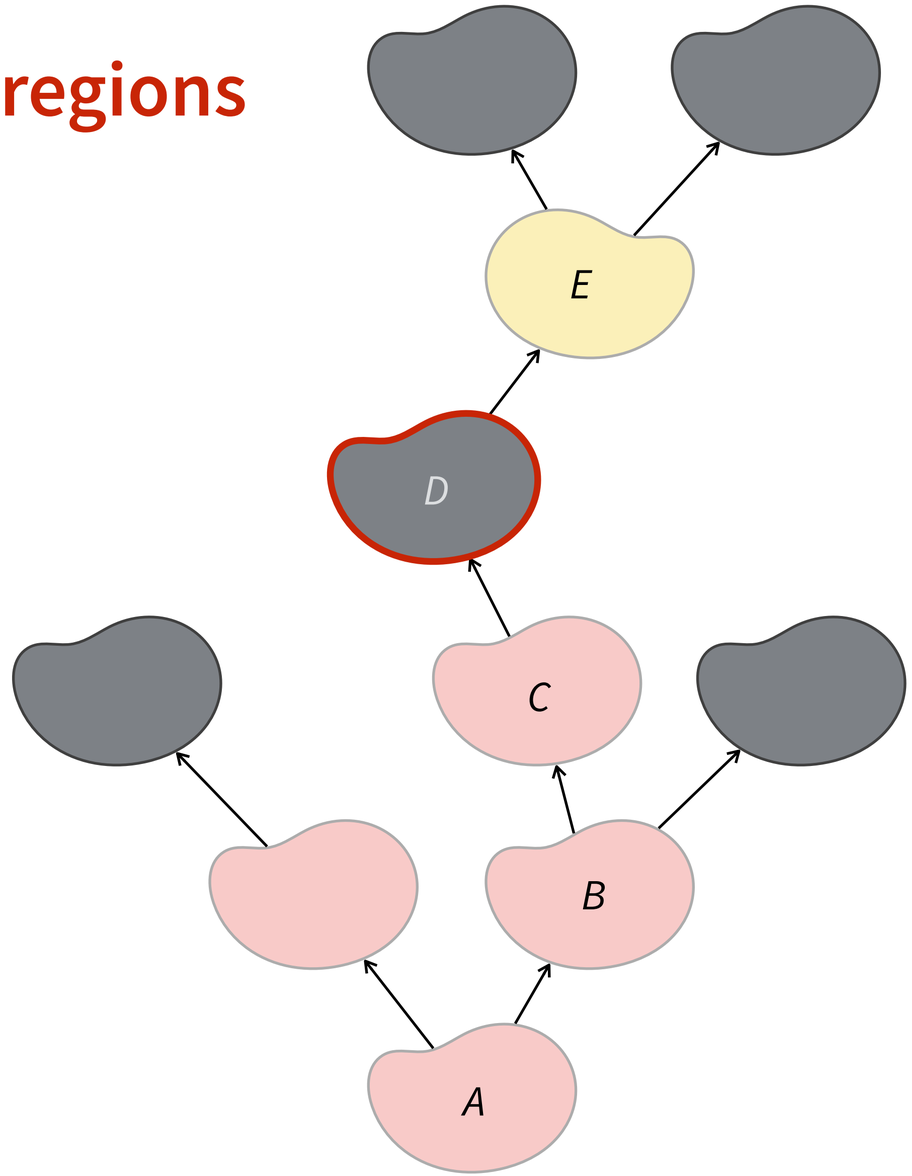
Explicitly moving the window of mutability from *R'* to *R*

```
// Assume we are in R'
enter m.f x => {
  // Now we are in R
  m.foo() // OK
  …
}
```



Closed regions cannot reach open regions

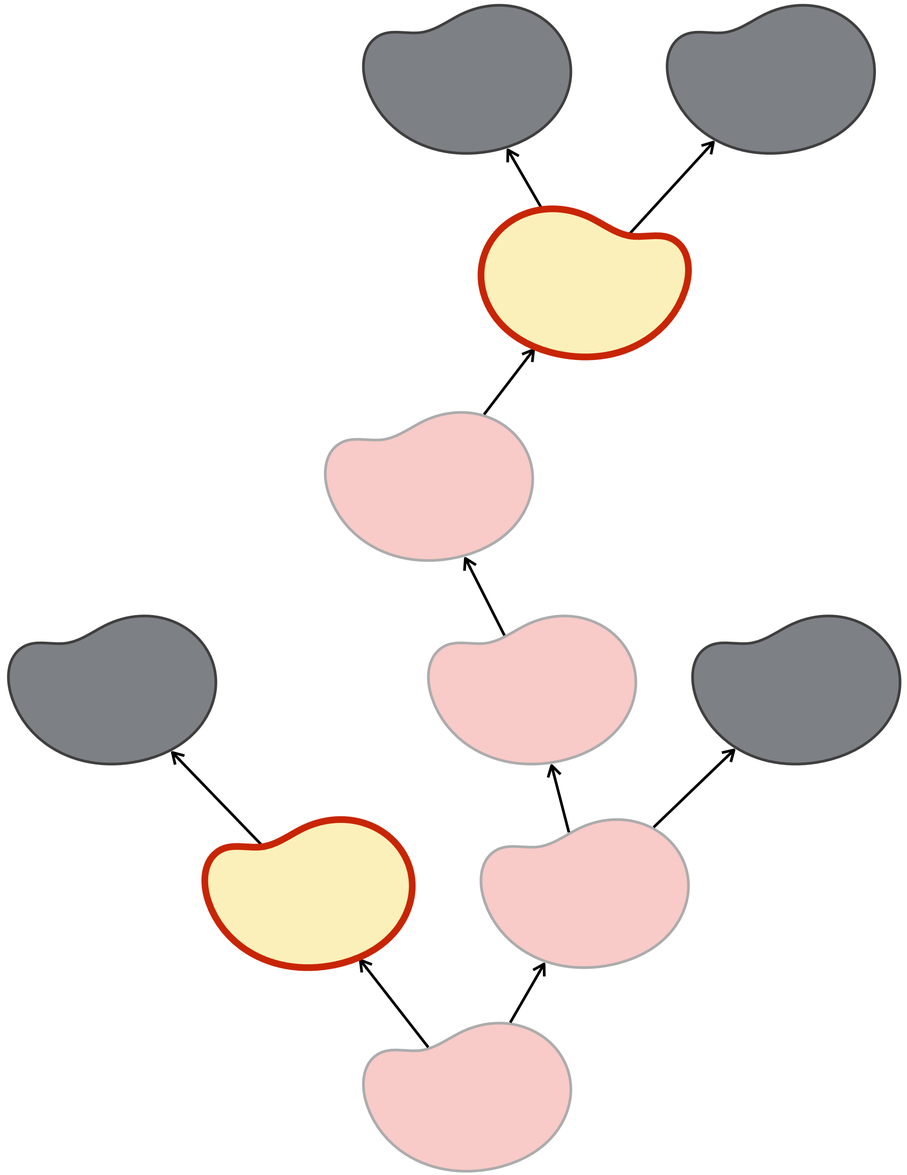
```
enter … A => {
  enter A.f B => {
    enter B.f C => {
      enter C.f D => {
        enter D.f E => {
          …
        }
      }
    }
  }
}
```



Not a legal program!

At <u>most</u> one window of mutability

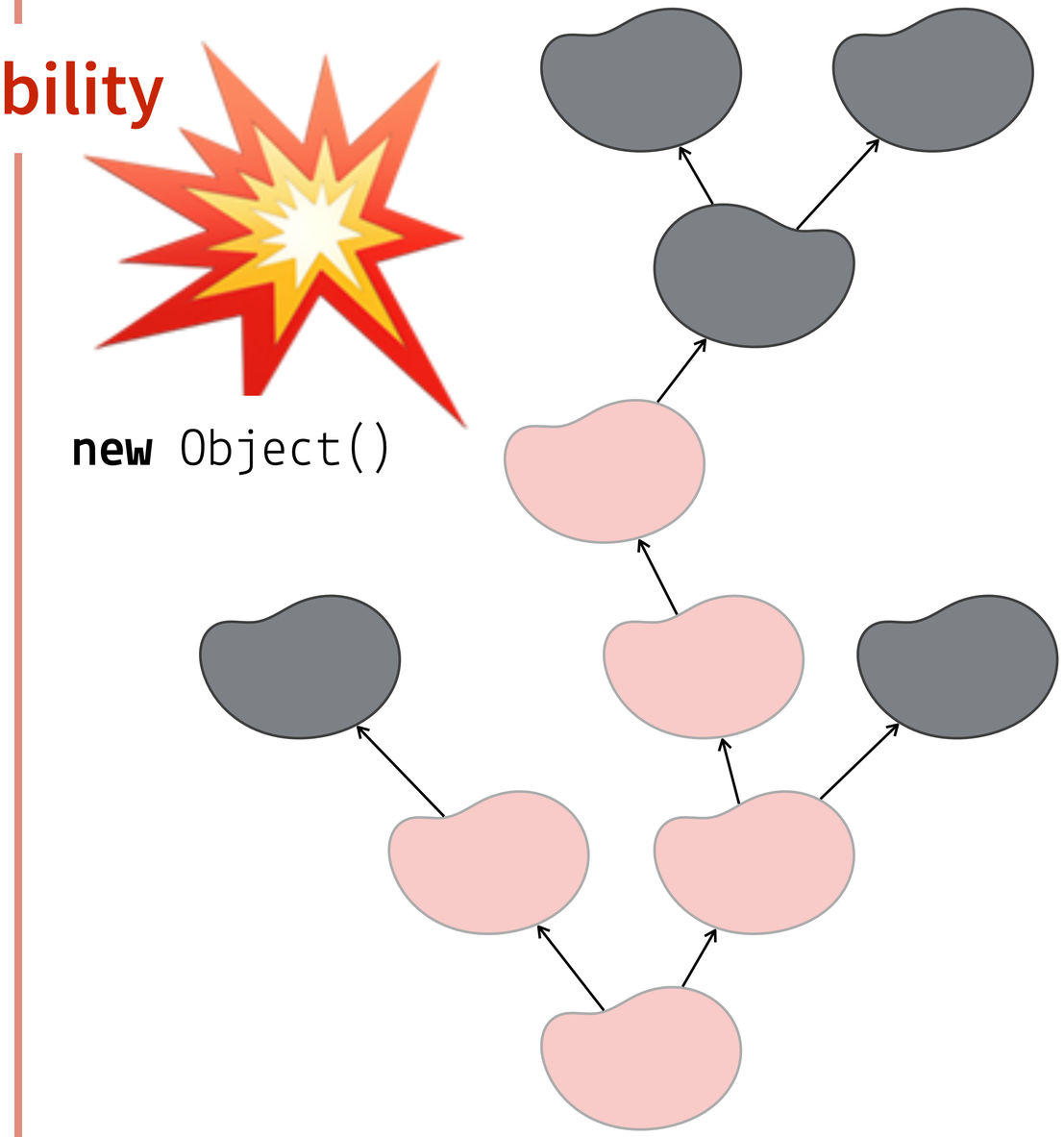
```
enter … A => {
  enter A.f B => {
    // suspends A
    …
  }
}
```



Not a legal program!

At <u>least</u> one window of mutability

```
enter … A => {
  enter A.f B => {
    // suspends A
    …
  }
  // re-activates A
}
```



Not a legal program!

A Type System to Enforce the Reggio design

- Based on reference capabilities

  Dictate what holder can do and what possible aliases can do

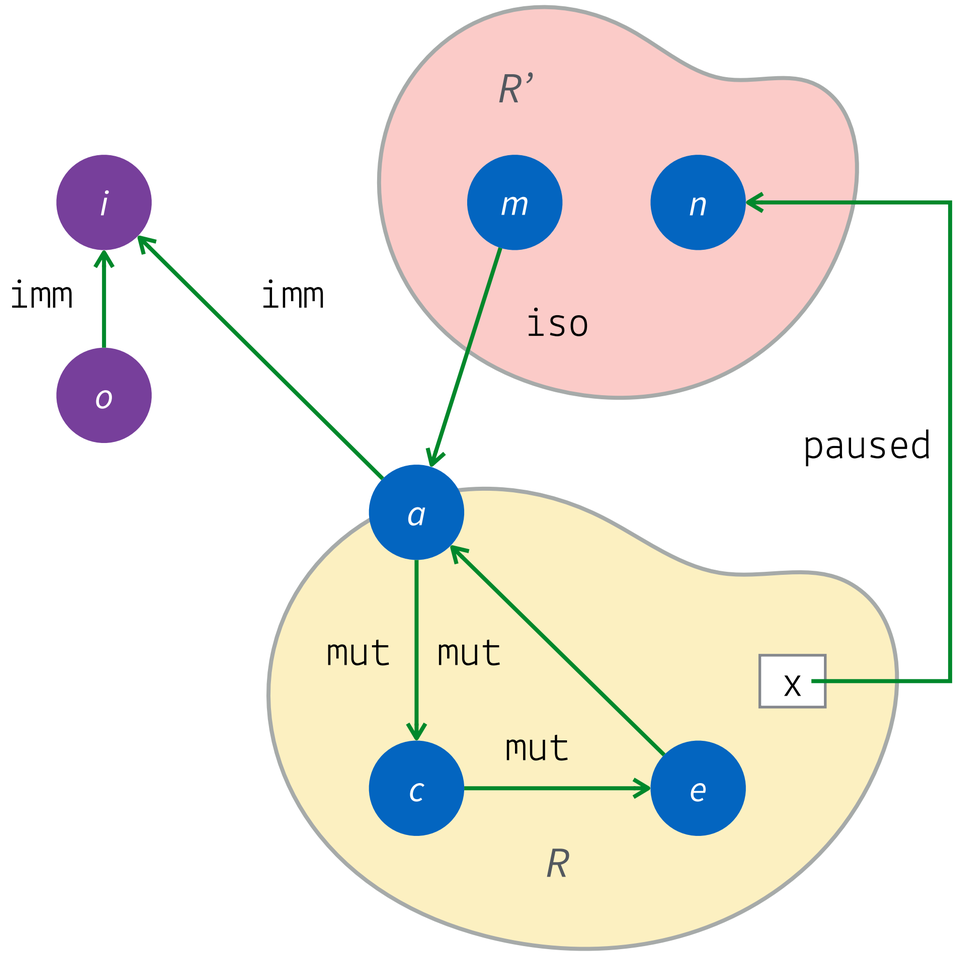
  Type = **`capability`** `& base_type` e.g., **`mut`** `& String`

- No need to distinguish different regions

  Only one mutable region

  Multiple suspended regions can be thought of as the same region

  Can only see bridge objects of closed regions which are necessarily stored in separate variables / fields



A Type System to Enforce the Reggio design

- Based on reference capabilities

  Dictate what holder can do and what possible aliases can do

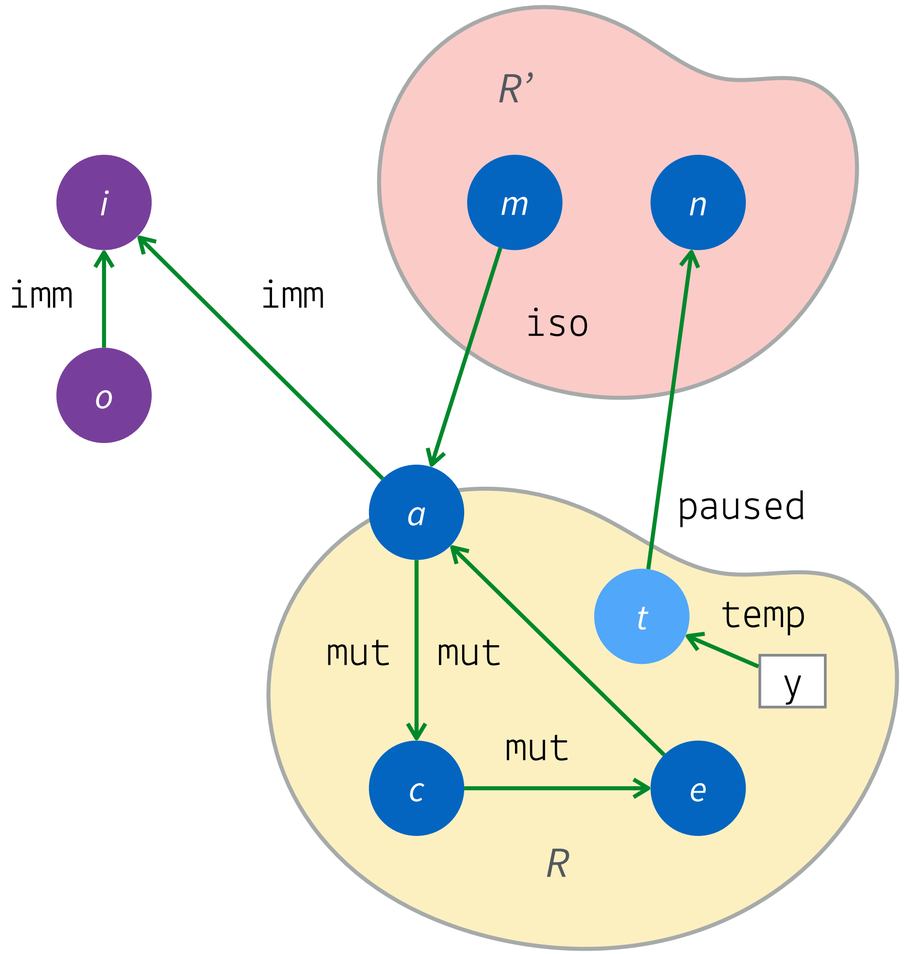
  Type = **`capability`** `&` `base_type` e.g., **`mut`** `&` `String`

- No need to distinguish different regions

  Only one mutable region

  Multiple suspended regions can be thought of as the same region

  Can only see bridge objects of closed regions which are necessarily stored in separate variables / fields



A Type System to Enforce the Reggio design

- Based on reference capabilities

  Dictate what holder can do and what possible aliases can do

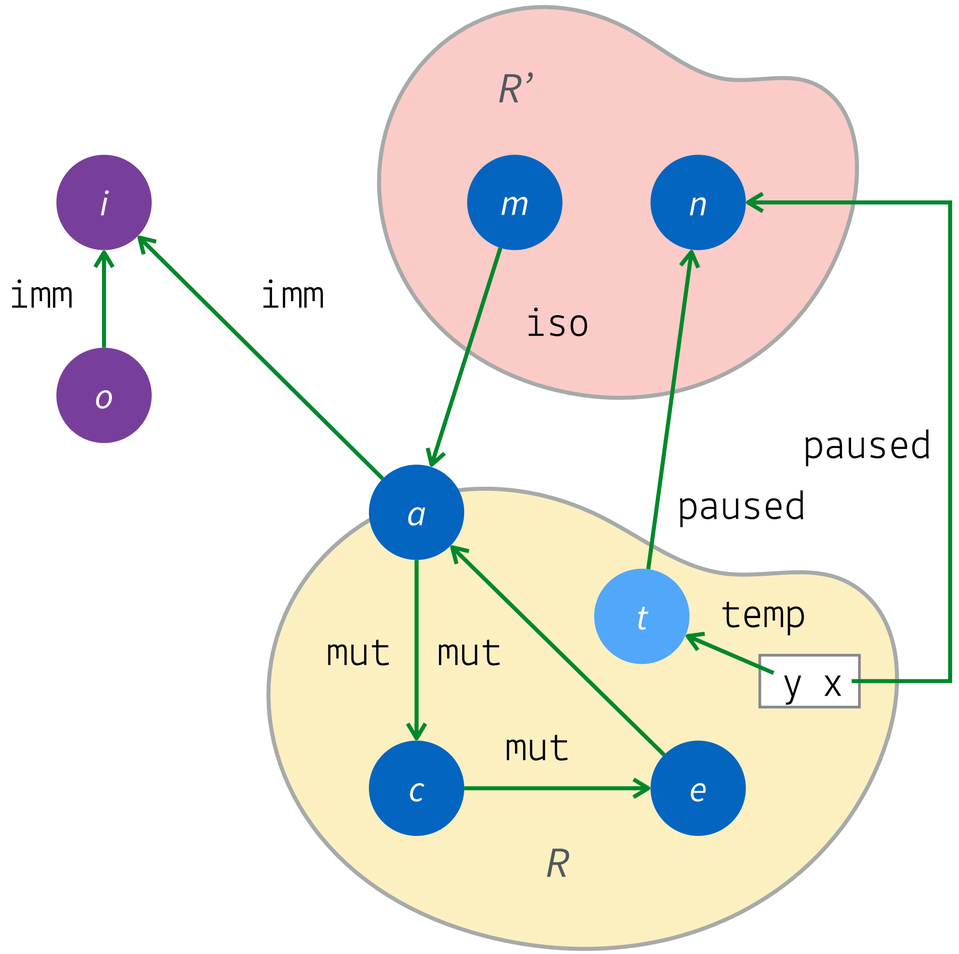
  Type = **`capability`** `& base_type` e.g., **`mut`** `& String`

- No need to distinguish different regions

  Only one mutable region

  Multiple suspended regions can be thought of as the same region

  Can only see bridge objects of closed regions which are necessarily stored in separate variables / fields



A Type System to Enforce the Reggio design

- Based on reference capabilities

  Dictate what holder can do and what possible aliases can do

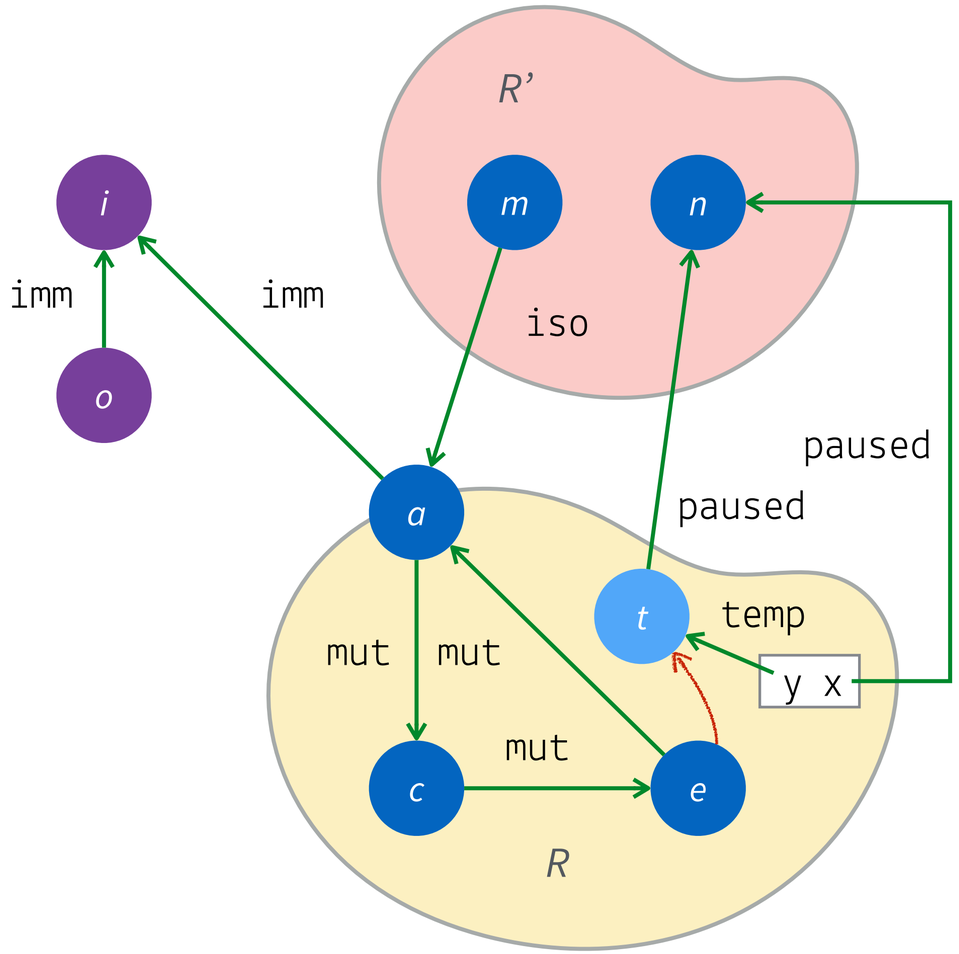
  Type = **`capability`** `& base_type` e.g., **`mut`** `& String`

- No need to distinguish different regions

  Only one mutable region

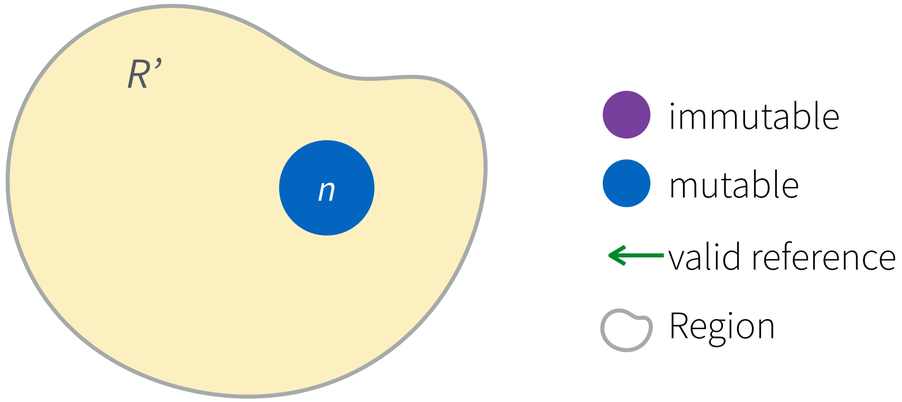
  Multiple suspended regions can be thought of as the same region

  Can only see bridge objects of closed regions which are necessarily stored in separate variables / fields



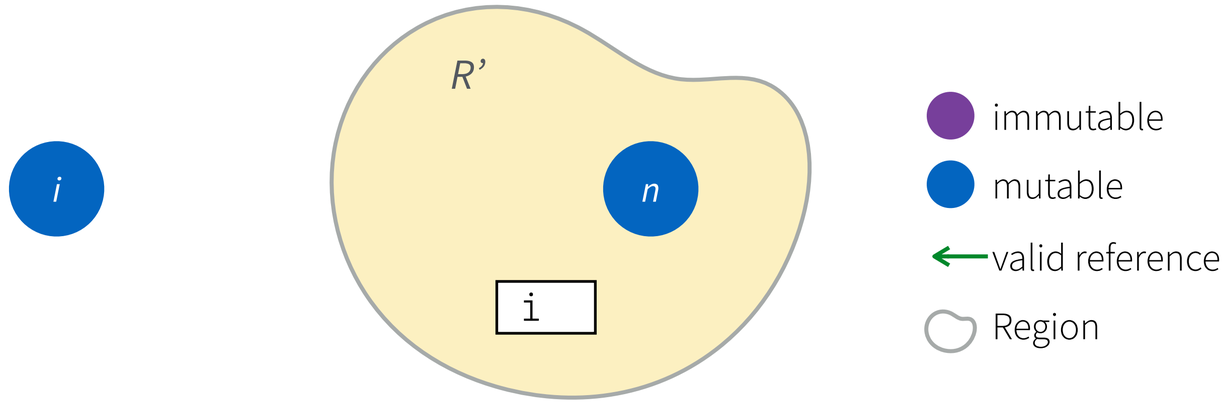
Creating our initial example

```
// We are inside R'
```



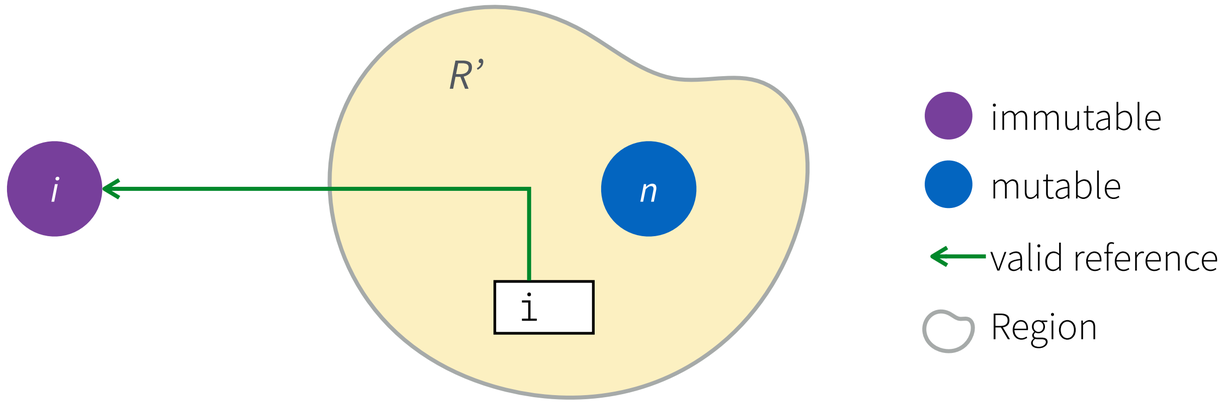
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
```



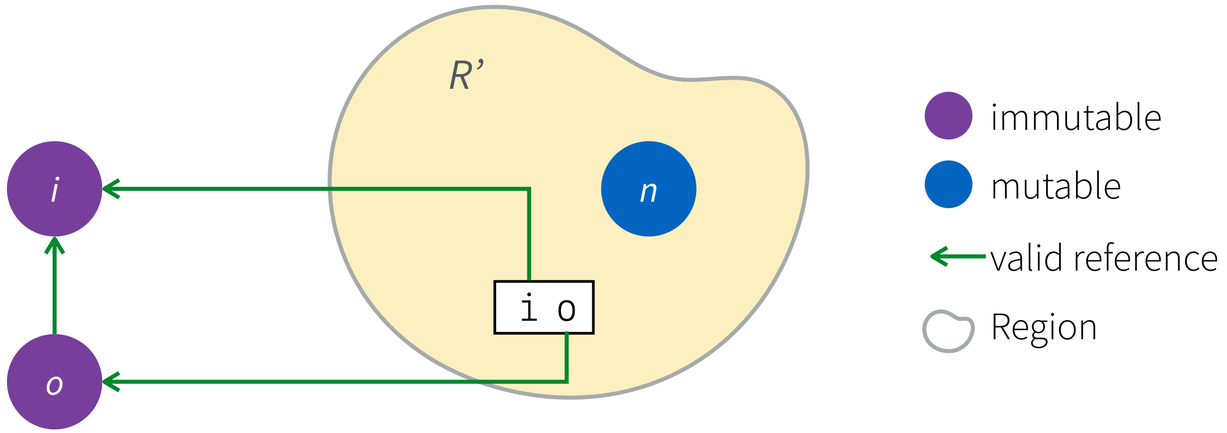
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
```



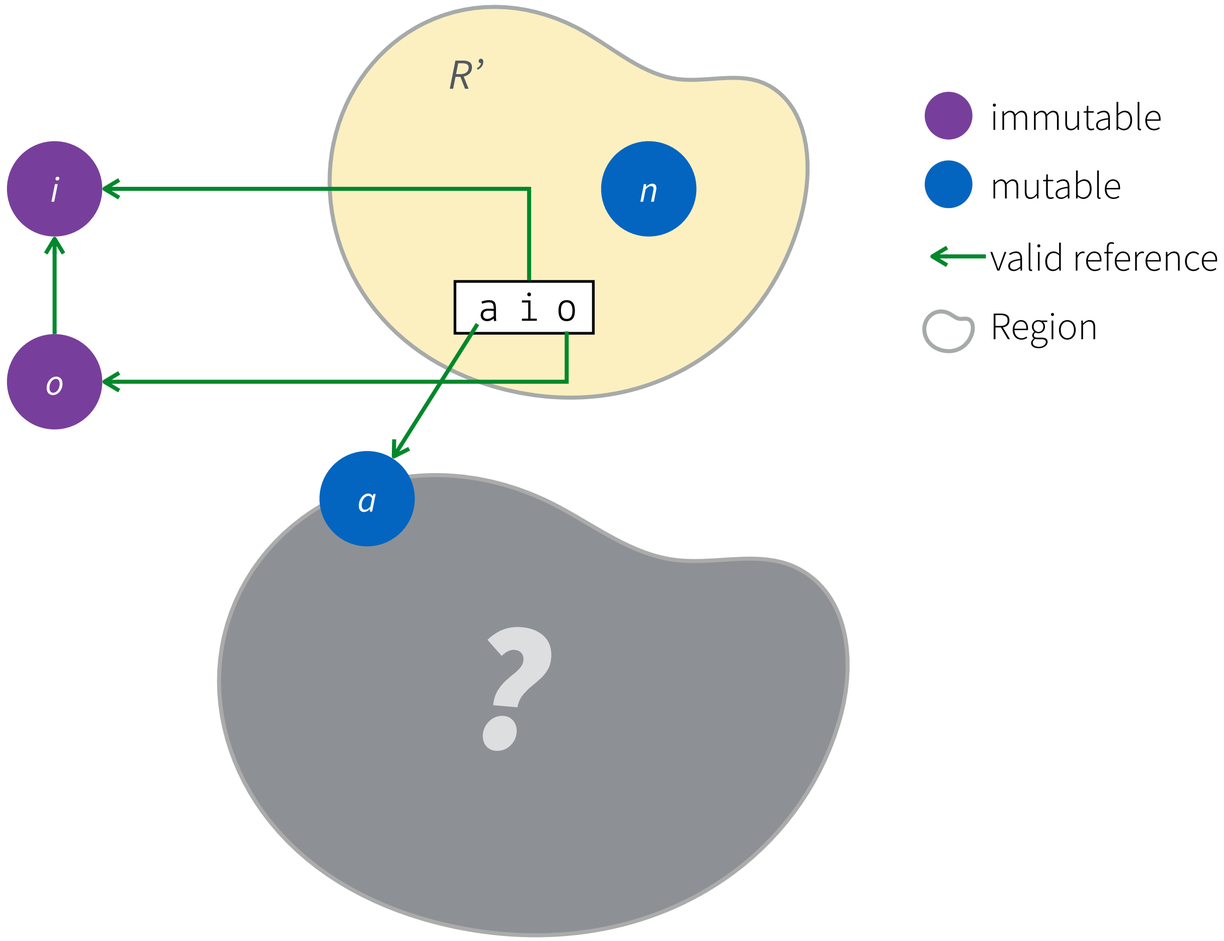
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
```



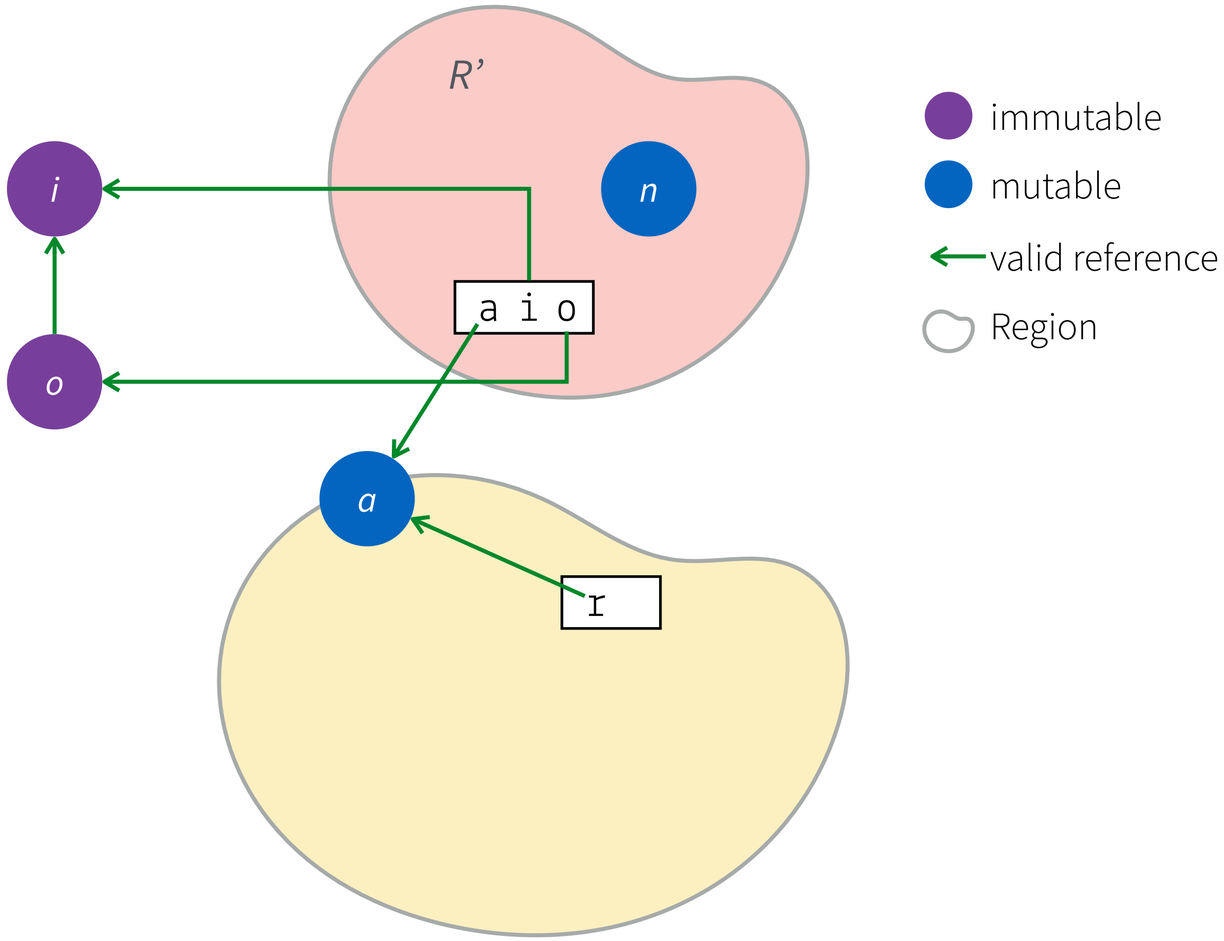
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
```



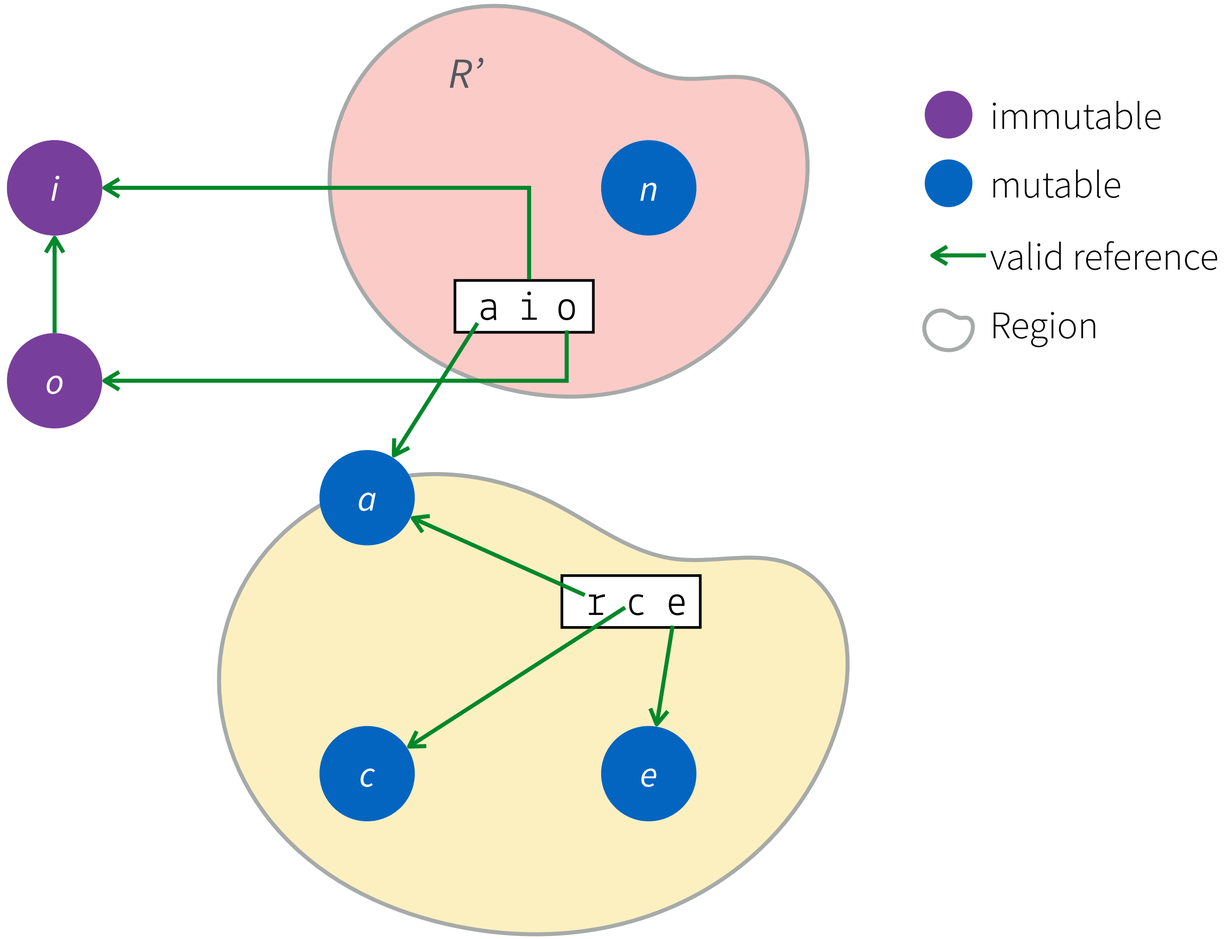
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
enter a r => {
```



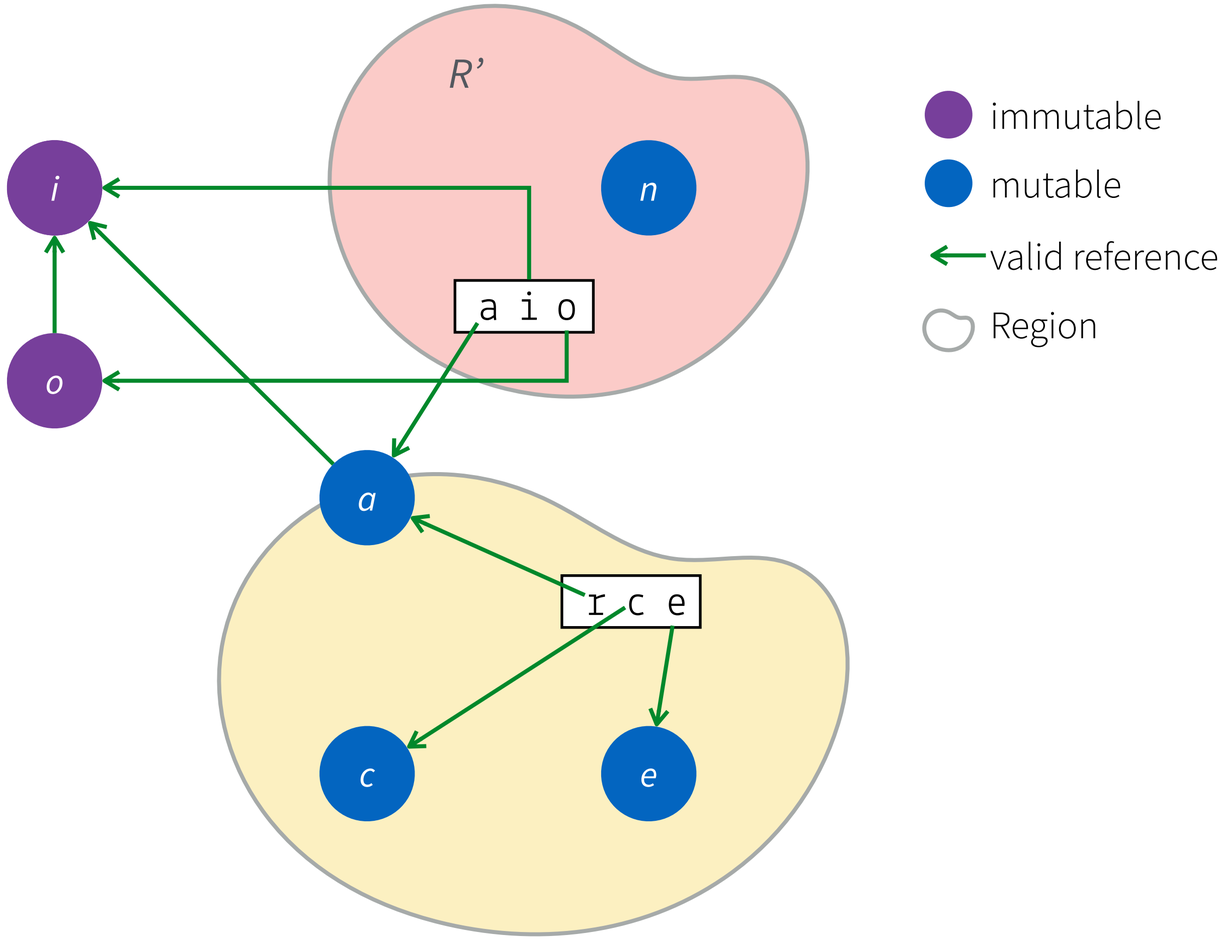
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
enter a r => {
  // We are inside R
  let c = new mut Link
  let e = new mut Link
```



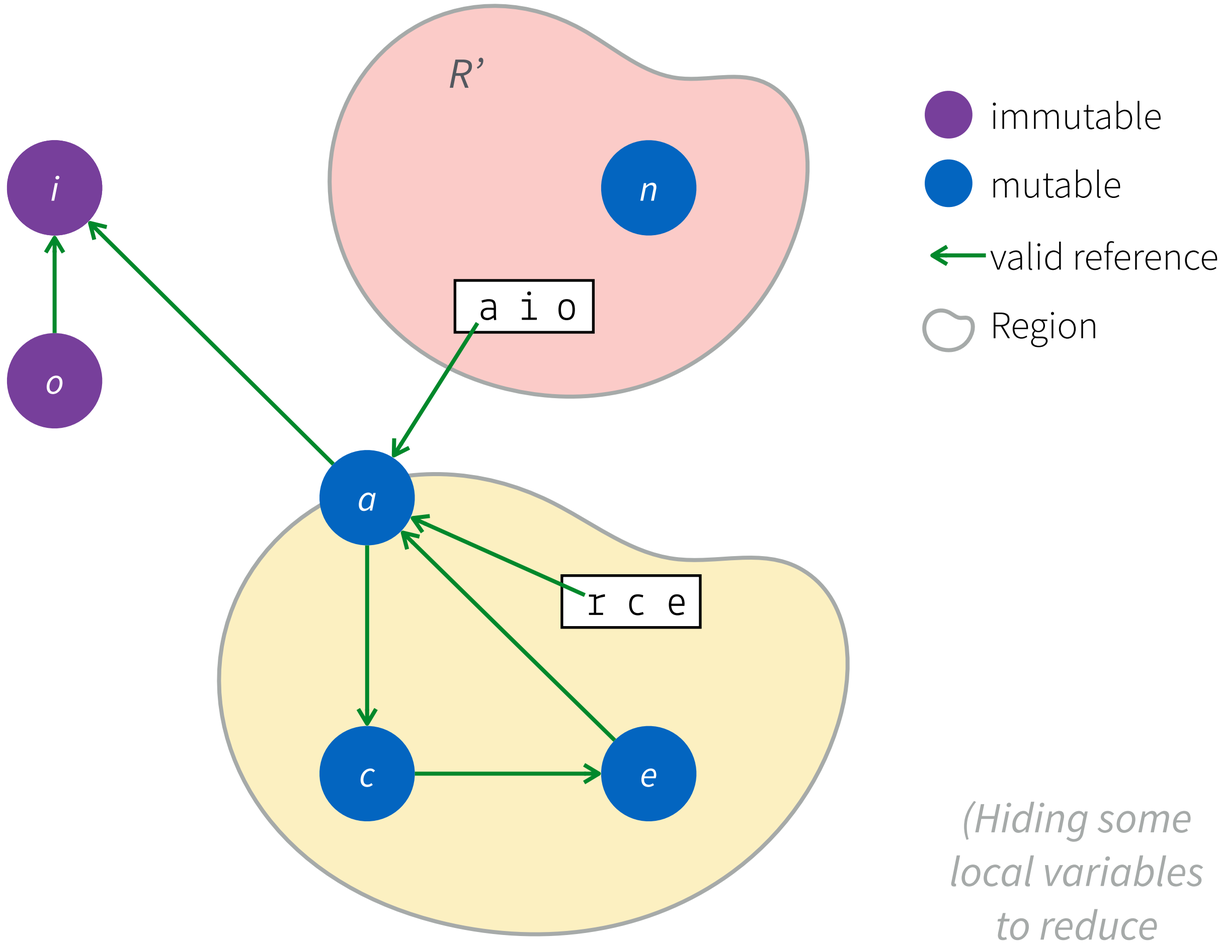
Creating our initial example

```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
enter a r => {
  // We are inside R
  let c = new mut Link
  let e = new mut Link
  r.elem := i
```



Creating our initial example

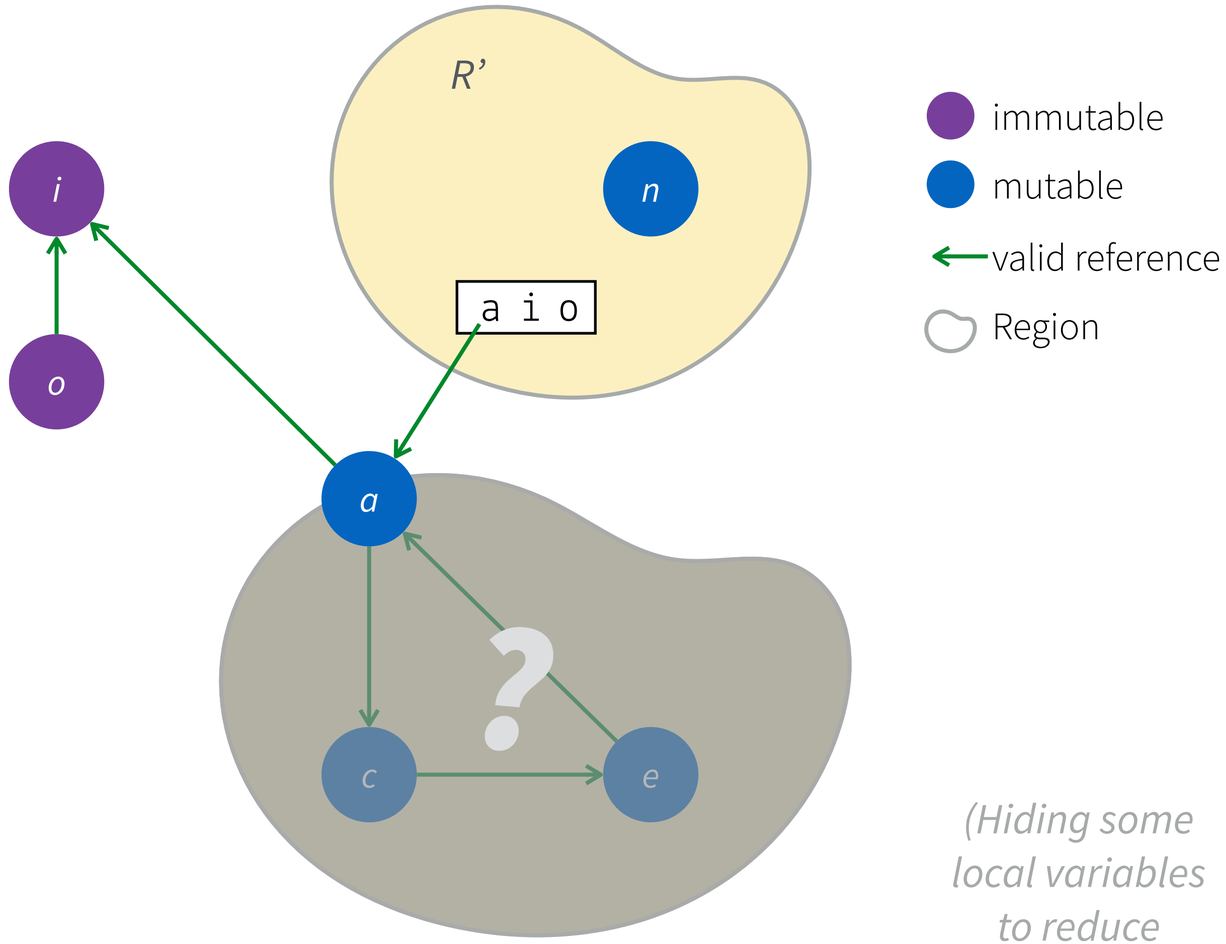
```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
enter a r => {
  // We are inside R
  let c = new mut Link
  let e = new mut Link
  r.elem := i
  r.next := c
  c.next := e
  e.next := r
```



(Hiding some local variables to reduce clutter)

Creating our initial example

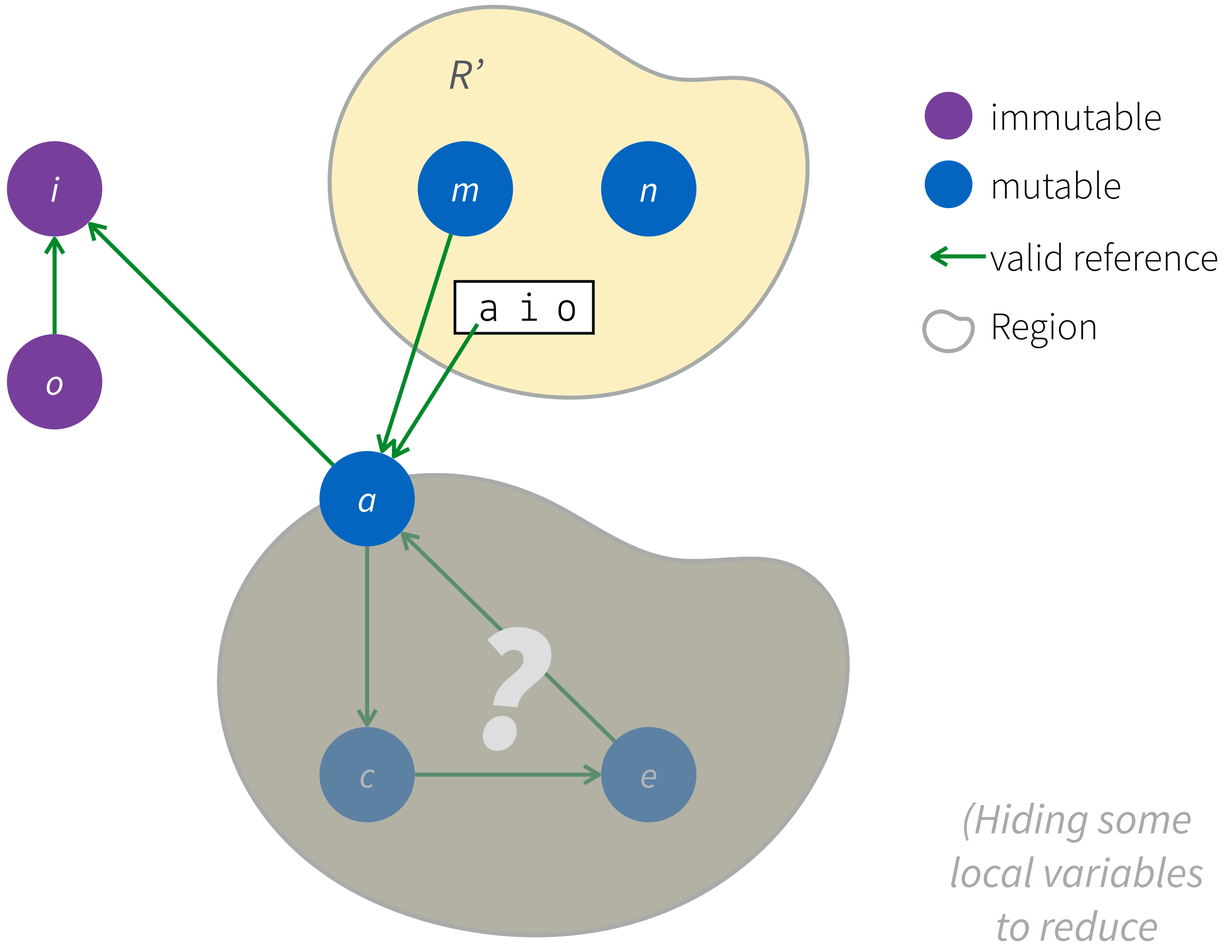
```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
enter a r => {
  // We are inside R
  let c = new mut Link
  let e = new mut Link
  r.elem := i
  r.next := c
  c.next := e
  e.next := r
}
```



(Hiding some local variables to reduce clutter)

Creating our initial example

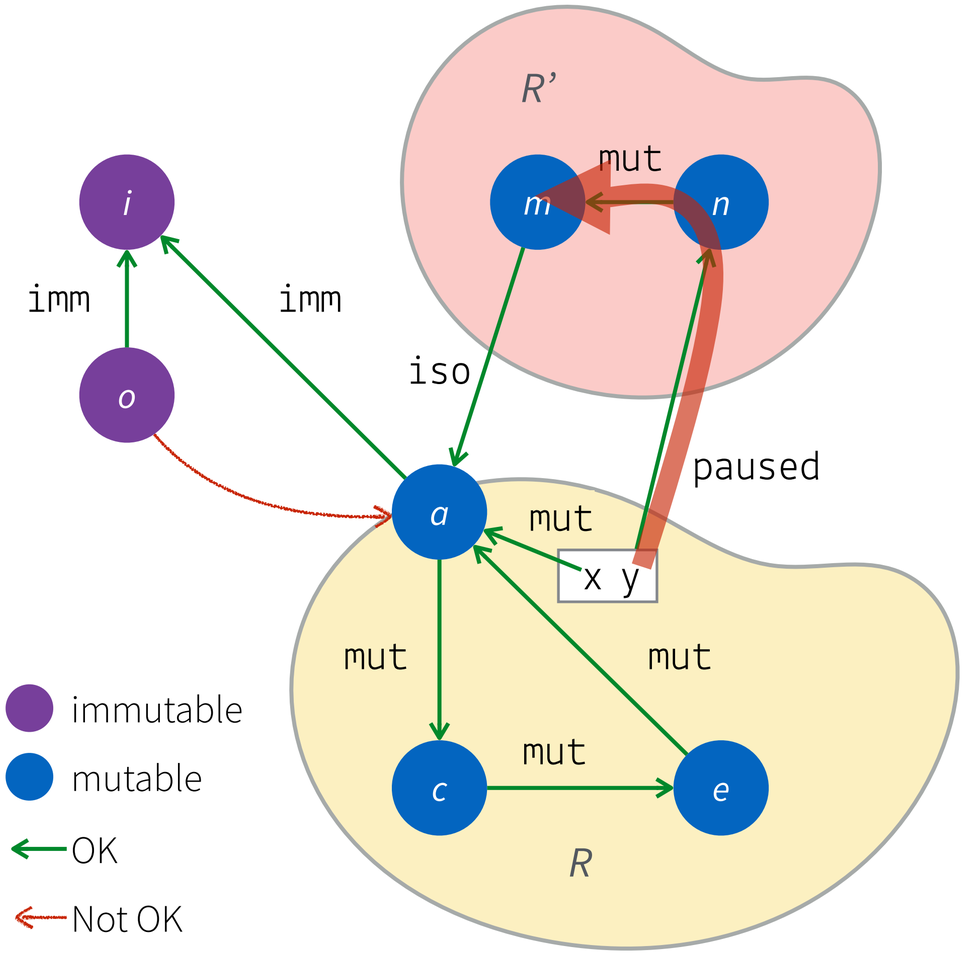
```
// We are inside R'
let i = freeze new iso Cell(42)
let o = freeze new iso Cell(i)
let a = new iso Link
enter a r => {
  // We are inside R
  let c = new mut Link
  let e = new mut Link
  r.elem := i
  r.next := c
  c.next := e
  e.next := r
}
let m = new mut Cell(a)
```



(Hiding some local variables to reduce clutter)

Viewpoint Adaptation

- Even though two paths alias, they need not have the same type

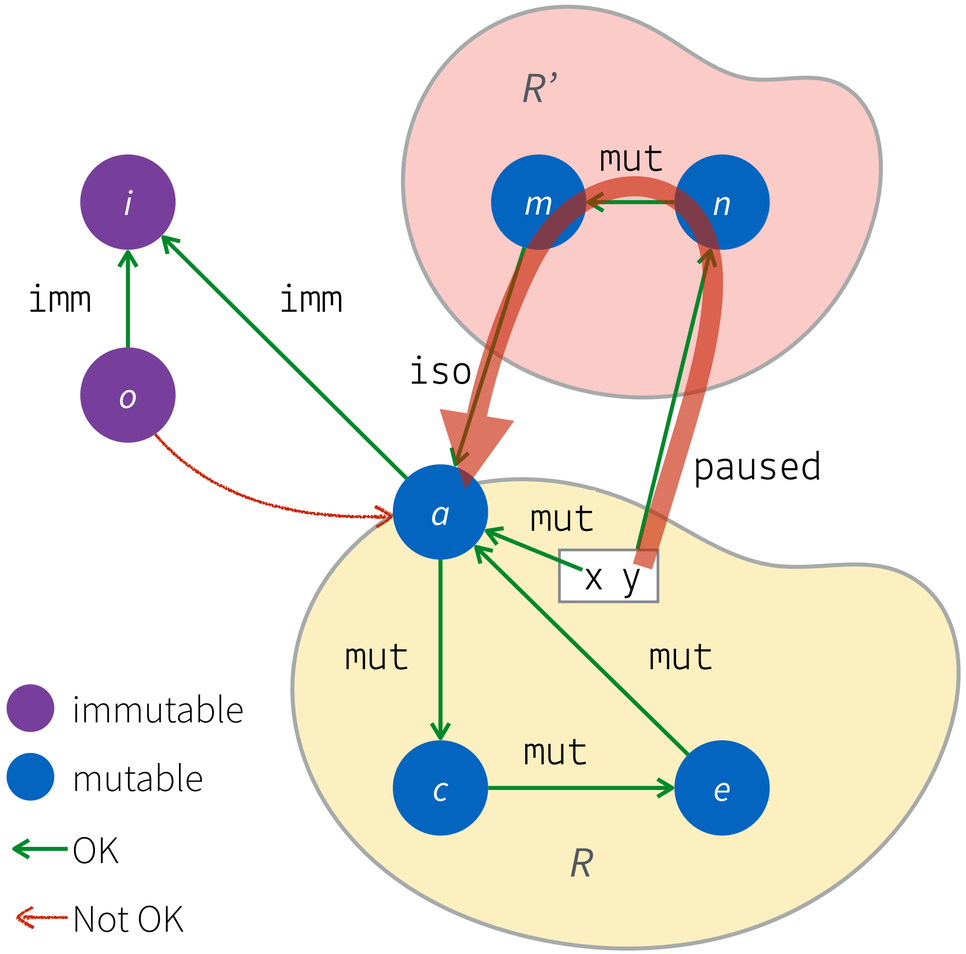


`y.n.m` : **paused**

Capability	mut	temp	imm	iso	paused
mut	mut	⊥	imm	iso	⊥
temp	mut	temp	imm	iso	paused
imm	imm	imm	imm	imm	imm
iso	⊥	⊥	⊥	⊥	⊥
paused	paused	paused	imm	iso	paused

Viewpoint adaptation: ”paused sees mut as paused”

Double Dipping?

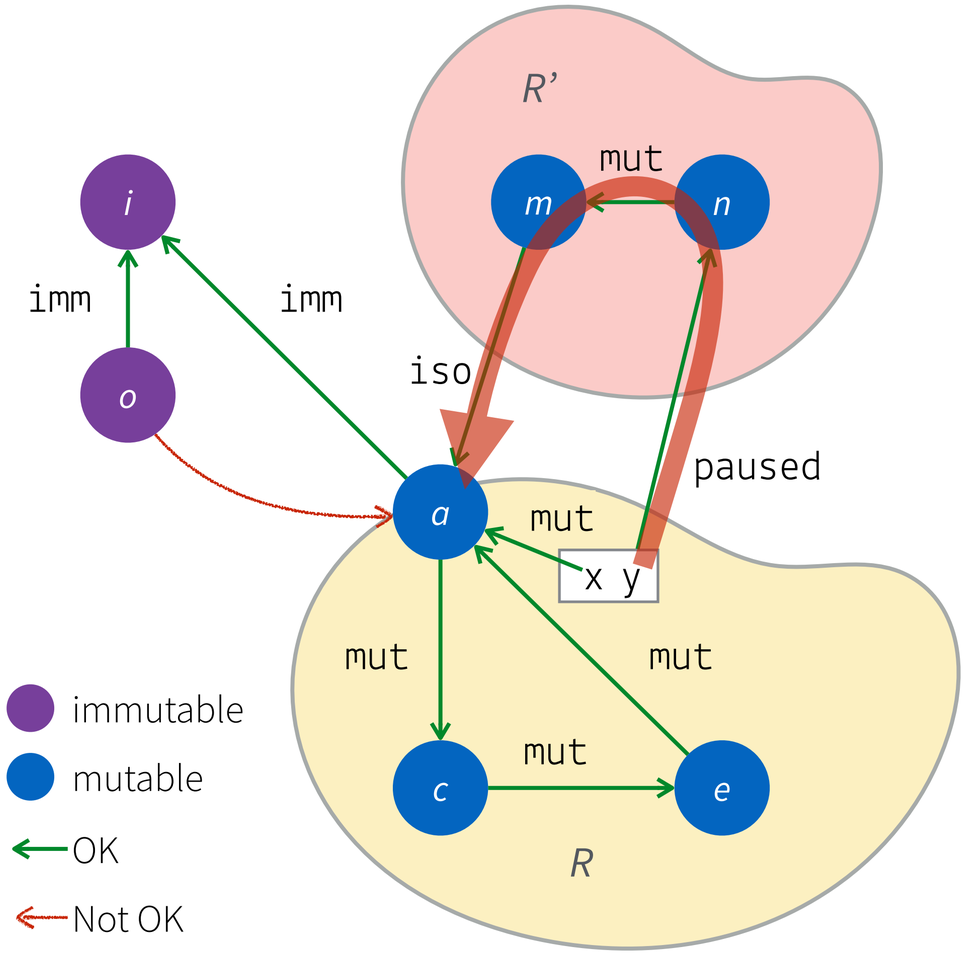


`y.n.m.a : ?`

Capability	mut	temp	imm	iso	paused
mut	mut	⊥	imm	iso	⊥
temp	mut	temp	imm	iso	paused
imm	imm	imm	imm	imm	imm
iso	⊥	⊥	⊥	⊥	⊥
paused	paused	paused	imm	iso	paused

Viewpoint adaptation: ”paused sees mut as paused”

Double Dipping?



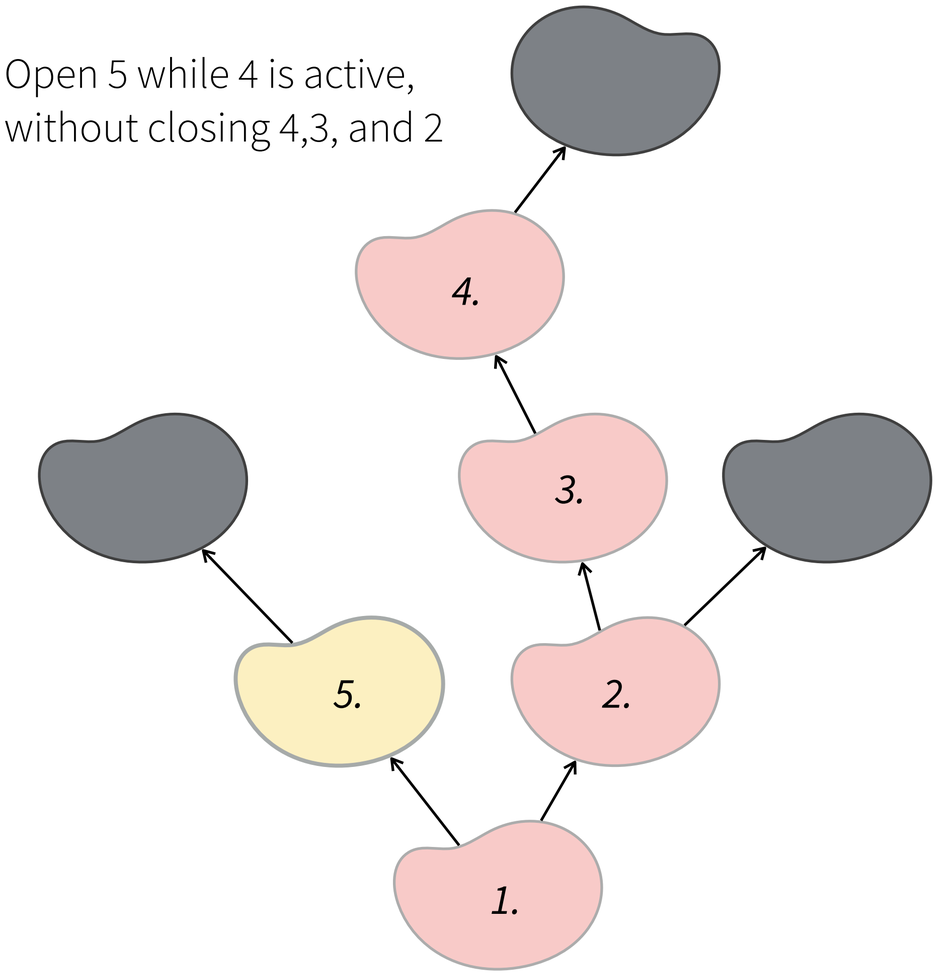
`y.n.m.a : iso`

Capability	mut	temp	imm	iso	paused
mut	mut	⊥	imm	iso	⊥
temp	mut	temp	imm	iso	paused
imm	imm	imm	imm	imm	imm
iso	⊥	⊥	⊥	⊥	⊥
paused	paused	paused	imm	iso	paused

Viewpoint adaptation: ”paused sees mut as paused”

Enter an `iso` via a paused needs a dynamic check (to trap double-dipping)

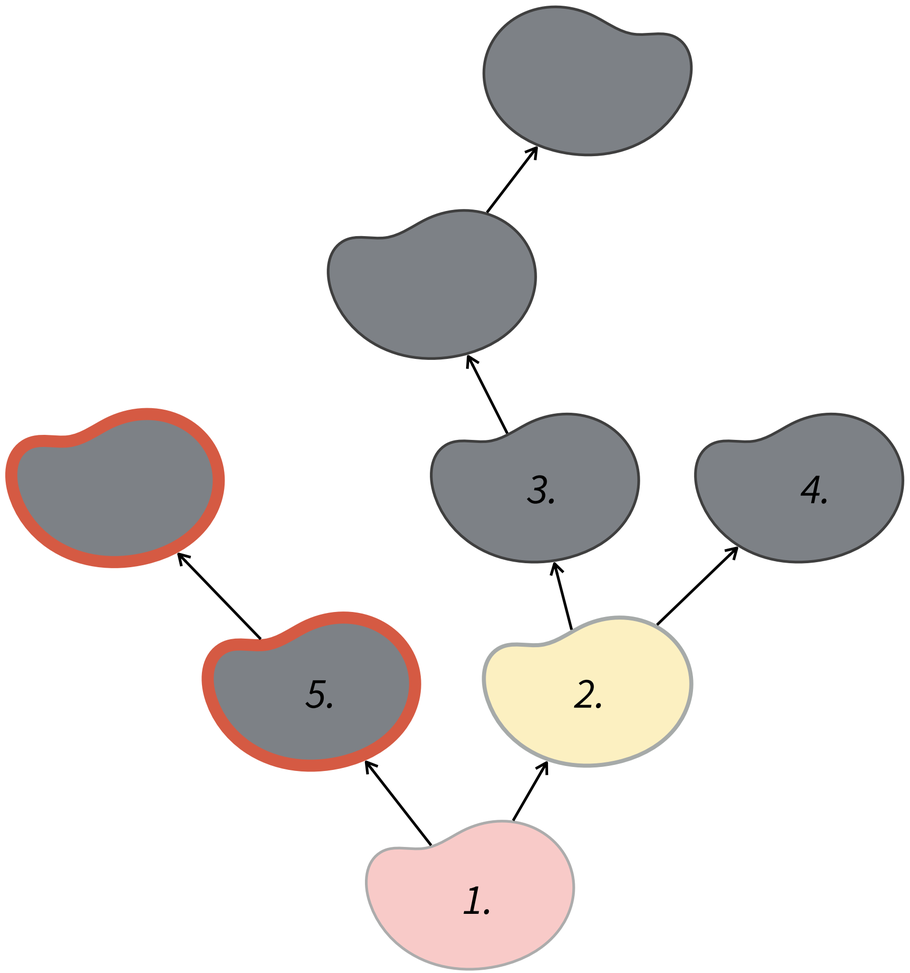
Open 5 while 4 is active,
without closing 4,3, and 2



Capability	mut	temp	imm	iso	paused
mut	mut	⊥	imm	iso	⊥
temp	mut	temp	imm	iso	paused
imm	imm	imm	imm	imm	imm
iso	⊥	⊥	⊥	⊥	⊥
paused	paused	paused	imm	iso	paused

Viewpoint adaptation: ”paused sees mut as paused”

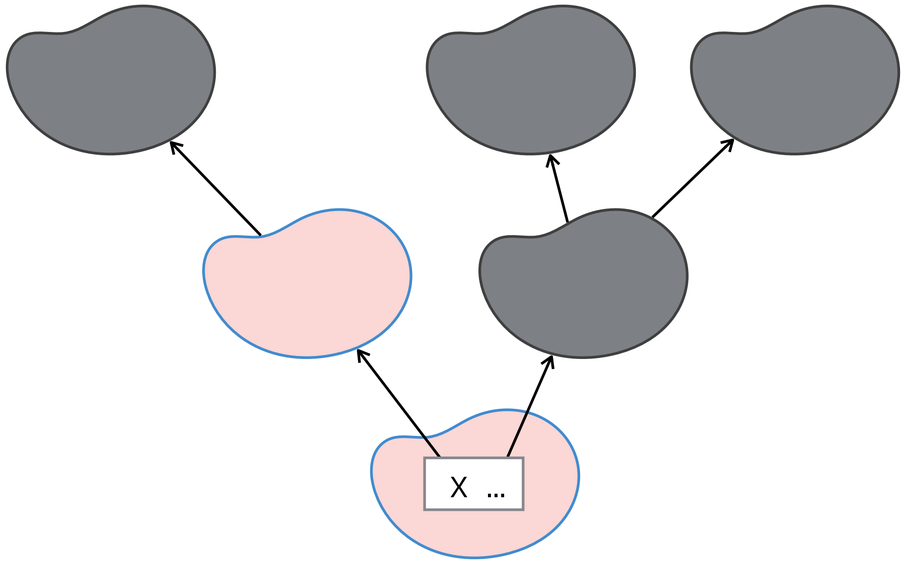
Rejected point in the design space



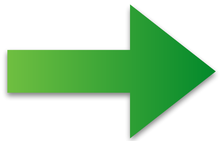
Capability	mut	temp	imm	iso	paused
mut	mut	⊥	imm	iso	⊥
temp	mut	temp	imm	iso	paused
imm	imm	imm	imm	imm	imm
iso	⊥	⊥	⊥	⊥	⊥
paused	paused	paused	imm	⊥	paused

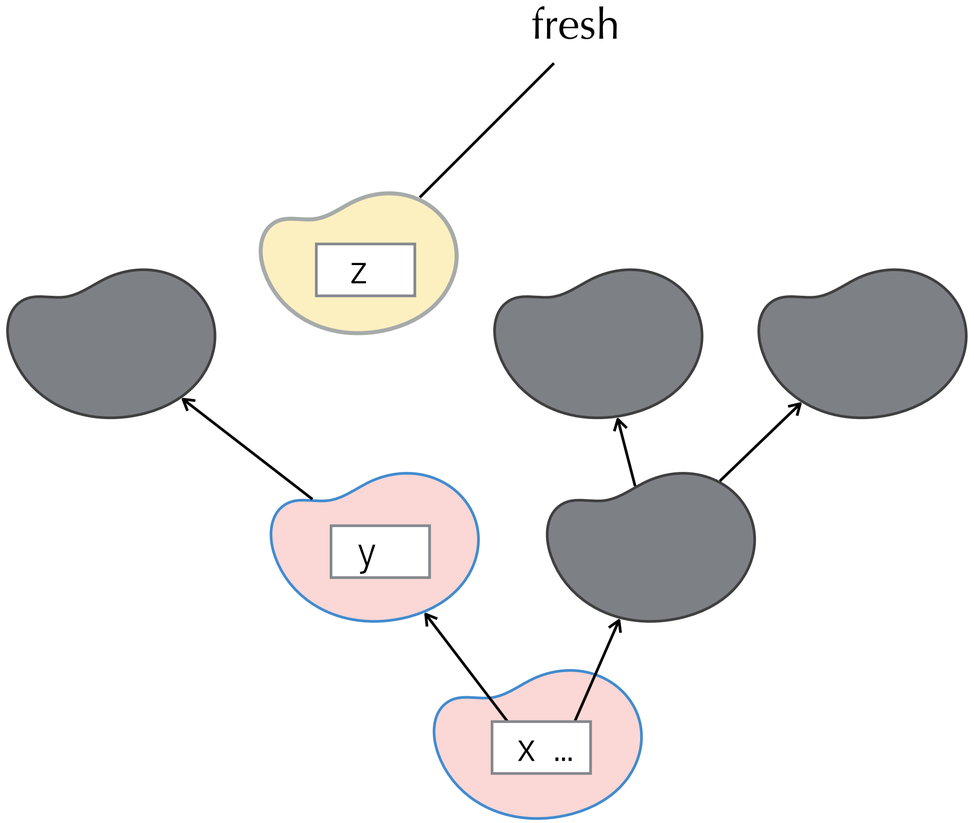
Viewpoint adaptation: ”paused sees iso as ⊥”

Explore




```
explore x y => {
  // Open y as paused
  …
}
```






```
enter x y => {
  // No other code in this block
  enter (new iso Object) z => {
    … // y: paused
  }
}
```

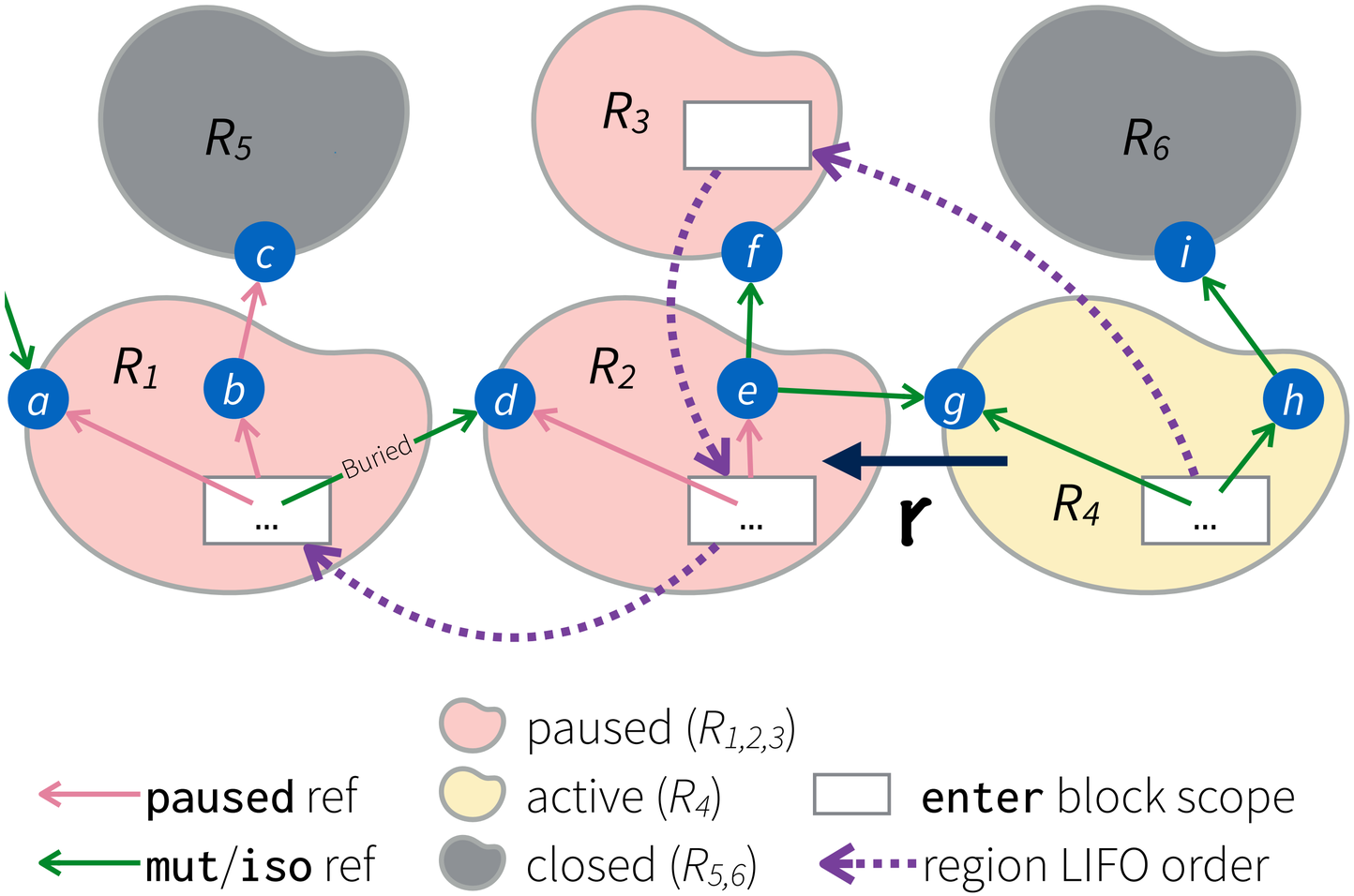
Computation in active regions can ignore objects in suspended regions

- **Guarantee 1**: when a suspended region is reactivated, it's objects will have exactly the same incoming references as when it became suspended
- **Guarantee 2**: no pointers from objects in a suspended region to objects in an active region
- G1 + G2 means no need to manipulate RC's or trace through pointers into a suspended region

r

Ergo: you only pay for memory managed in the active region

Partitioning controls costs



Key proof: the topology invariant

$$\forall ref_1, ref_2 \in \text{references}(\langle RS; H_{op}; H_{cl}; H_{fr} \rangle).$$

$$\bigvee \begin{cases} ref_1 = ref_2 & (1) \\ \text{reg}(\text{dst}(ref_1)) \neq \text{reg}(\text{dst}(ref_2)) & (2) \\ \text{reg}(\text{src}(ref_1)) = \text{reg}(\text{dst}(ref_1)) \vee \\ \quad \text{reg}(\text{src}(ref_2)) = \text{reg}(\text{dst}(ref_2)) & (3) \\ \text{reg}(\text{dst}(ref_1)) \in \text{regions}(H_{fr}) \vee \\ \quad \text{reg}(\text{dst}(ref_2)) \in \text{regions}(H_{fr}) & (4) \\ RS \vdash \text{dst}(ref_1) \preceq \text{src}(ref_1) \vee \\ \quad RS \vdash \text{dst}(ref_2) \preceq \text{src}(ref_2) & (5) \end{cases}$$

For any two references ref_1 and ref_2 in a well-formed configuration, either they…

(1) …are the same reference *(n.b. not aliases)*

(2) …point into different regions

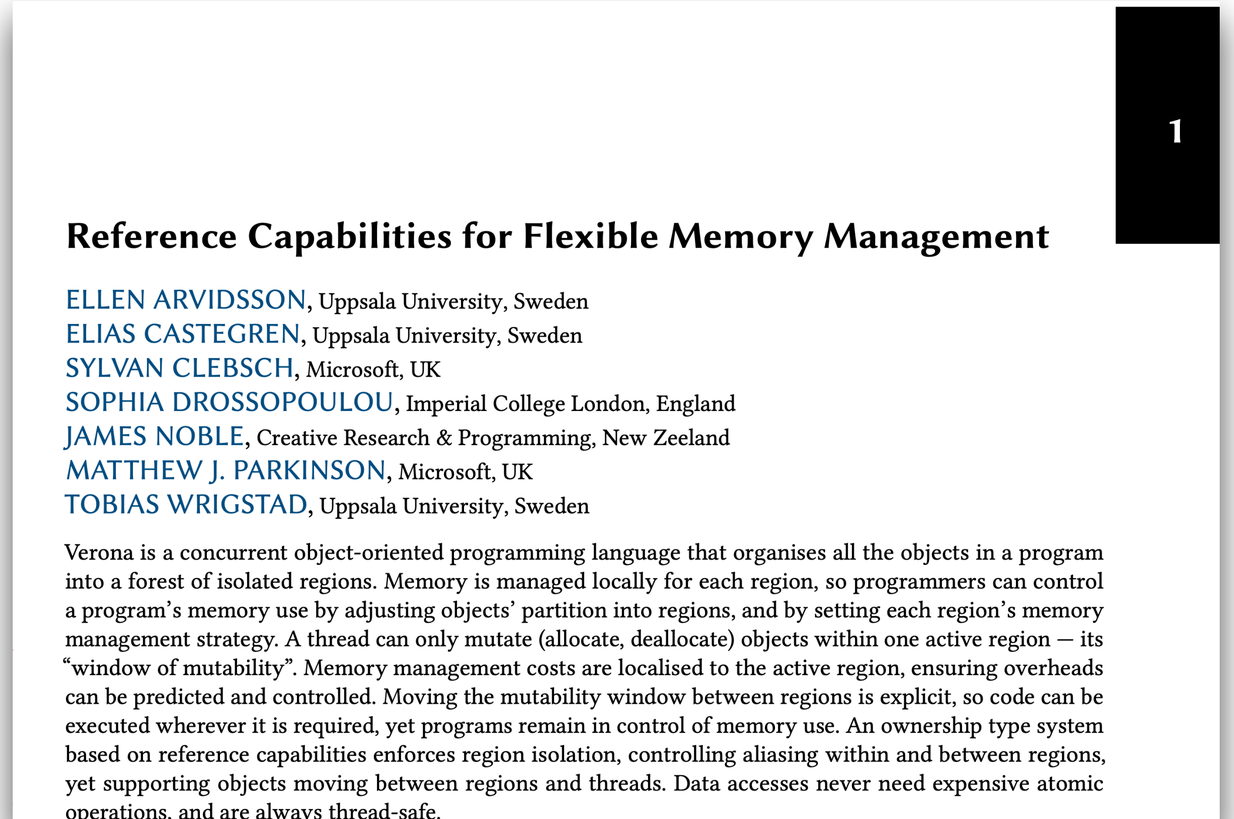
(3) …point between objects in the same region(s)

or at least one of them

(4) …points to an immutable object

(5) …points to a region higher up the region stack

Will appear at OOPSLA / SPLASH in October



1

Reference Capabilities for Flexible Memory Management

ELLEN ARVIDSSON, Uppsala University, Sweden
ELIAS CASTEGREN, Uppsala University, Sweden
SYLVAN CLEBSCH, Microsoft, UK
SOPHIA DROSSOPOULOU, Imperial College London, England
JAMES NOBLE, Creative Research & Programming, New Zeeland
MATTHEW J. PARKINSON, Microsoft, UK
TOBIAS WRIGSTAD, Uppsala University, Sweden

Verona is a concurrent object-oriented programming language that organises all the objects in a program into a forest of isolated regions. Memory is managed locally for each region, so programmers can control a program's memory use by adjusting objects' partition into regions, and by setting each region's memory management strategy. A thread can only mutate (allocate, deallocate) objects within one active region — its "window of mutability". Memory management costs are localised to the active region, ensuring overheads can be predicted and controlled. Moving the mutability window between regions is explicit, so code can be executed wherever it is required, yet programs remain in control of memory use. An ownership type system based on reference capabilities enforces region isolation, controlling aliasing within and between regions, yet supporting objects moving between regions and threads. Data accesses never need expensive atomic operations, and are always thread-safe.

with Regions

Wanted: Controllable and Predictable Memory Management Costs

- Mix-and-Match Memory Management

  We don't care about how suspended regions manage their memory

  Ability to freely mix multiple ways to manage memory in the same program at region granularity

- Incremental Memory Management

  We only need to manipulate RC's/trace refs. in the active region

  Only pay for memory management in a subset of the heap — in the active region granularity

- Concurrent Memory Management

  All regions are thread-local (but cheaply transferrable)

  Program work should not have to wait for GC work, and conversely — program thread opening region can choose to GC

- Zero-Copy Ownership Transfer

  i.e. change region topology Falls out of region isolation + uniqueness

  Moving obligation to manage memory between different parts of a program should not come with a cost

- Safe Concurrency

  Falls out of region isolation + uniqueness

  Freedom from data races; if thread T can reference object O, then T may access O synchronously at *no* synchronisation cost

  i.e. a region is only visible and access

...but is it unobtrusive enough?