Applied Unified Ownership

or

Capabilities for Sharing Across Threads

Elias Castegren Tobias Wrigstad

DRF

transfer

parallel programming

Applied Unified Ownership

memory management

placement in pools
(previous talk)

pool dependencies
(previous talk)

GC
(coming up)

many forms
of ownership

many forms
of uniqueness

capabilities

Applied Unified Ownership

substructural types

behavioural types

regions

effects

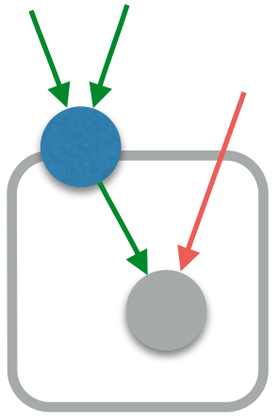
read-only

coordination

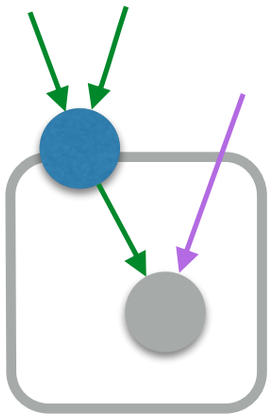
lock-freedom

Ownership Variants

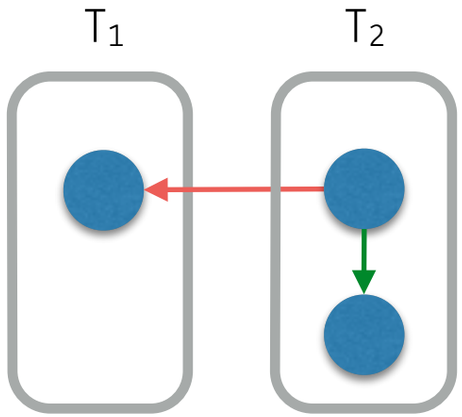
dominators



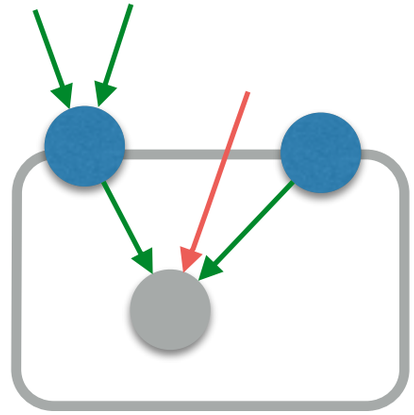
modifiers



threads

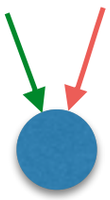


ombudsmen

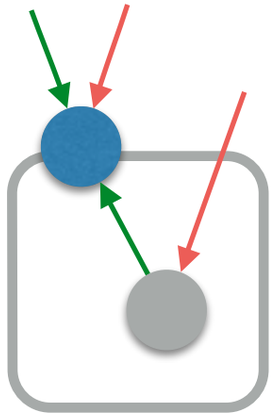


Uniqueness Variants

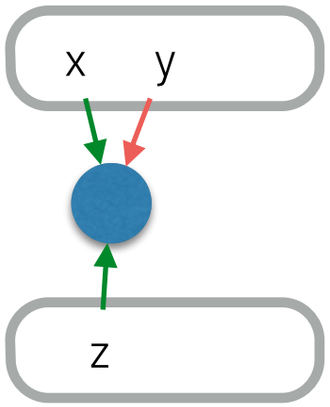
classical



external

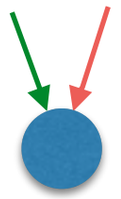


local / refined ownership

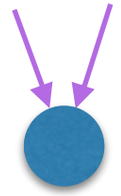


”Mix”

fractional permissions



or



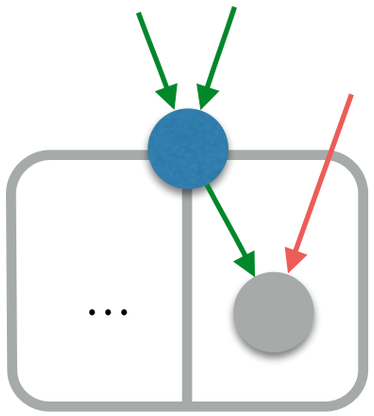
normal

read-only

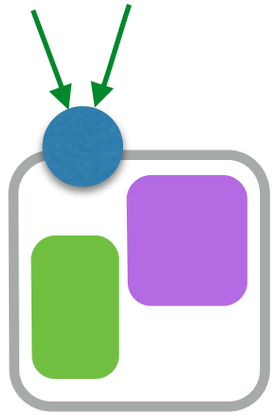
disallowed

Static Control of Side-Effects

regions

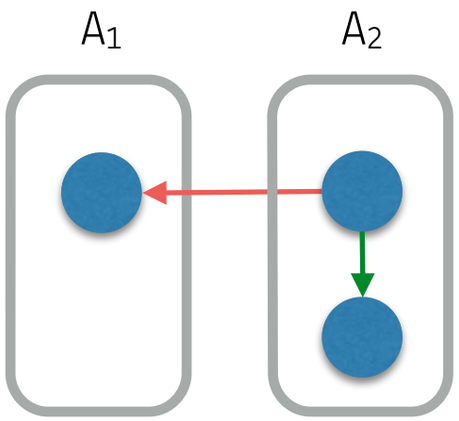


effect systems

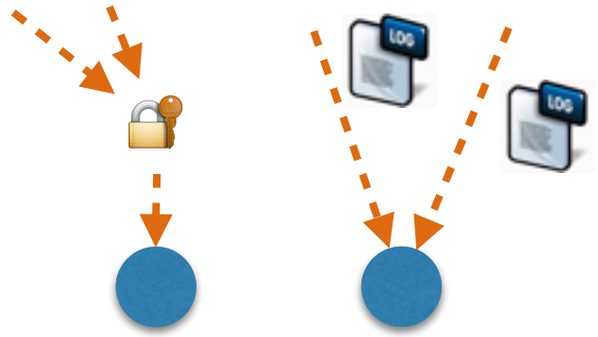


Dynamic Control of Side-Effects

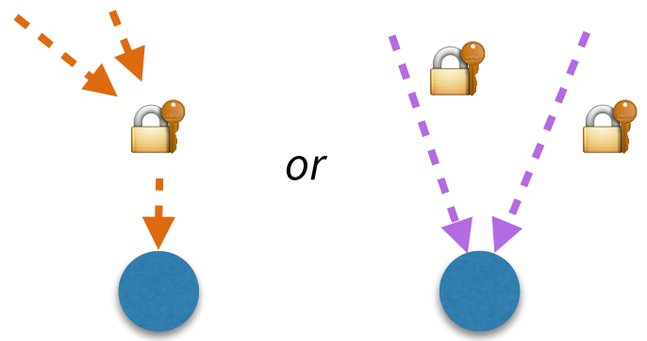
active object isolation



locks or transactions



readers-writer lock



normal

read-only

disallowed

access

read access

Capabilities Replaces References

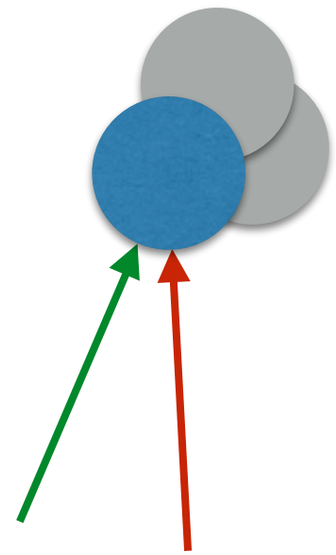
- A capability is a token governing access to a particular resource

  >1 capability governs access to single resource = aliasing / sharing

- Capabilities control their own flow through a system

  Copy semantics : aliasing of resources

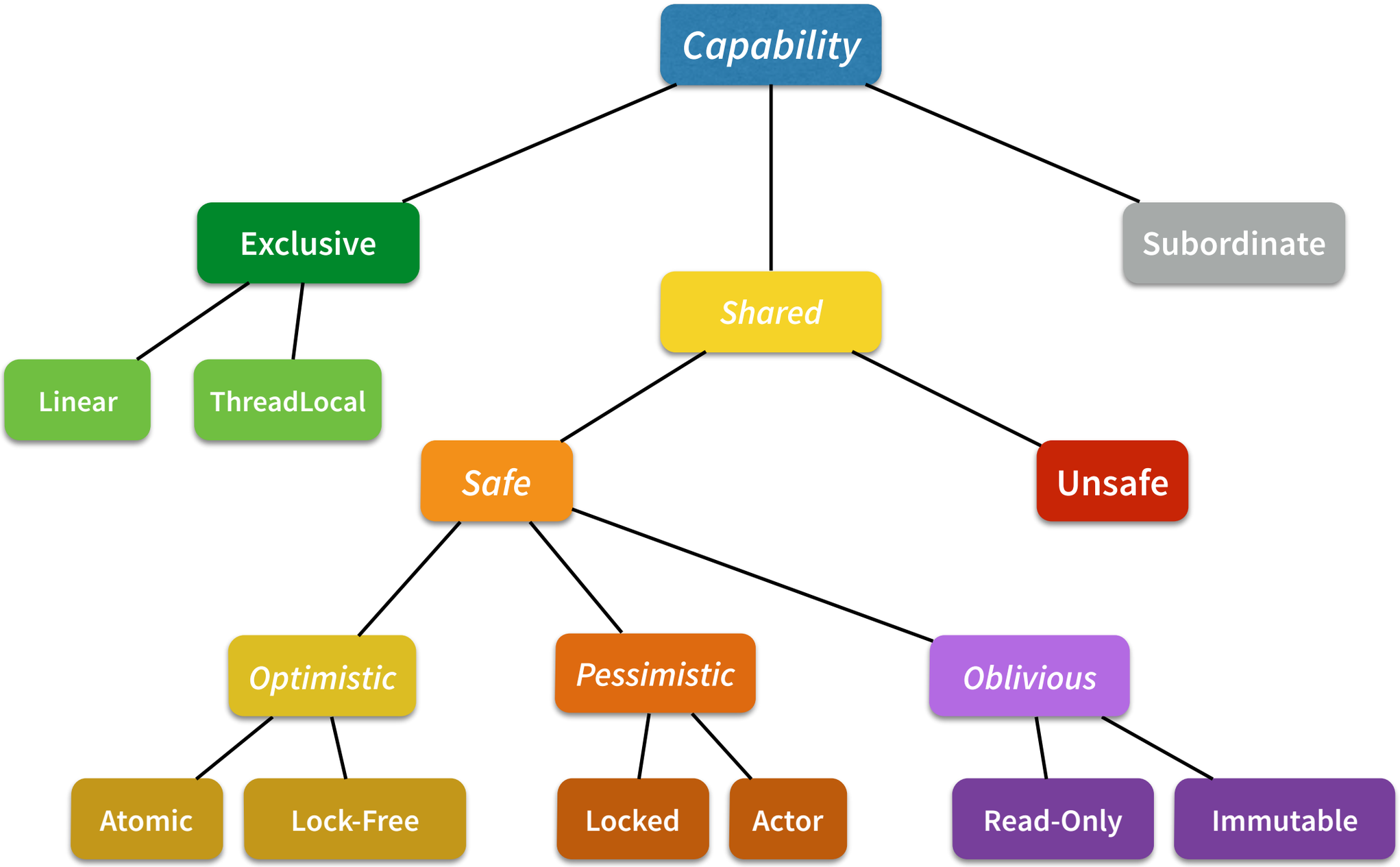
  Transfer semantics : linear access to resources



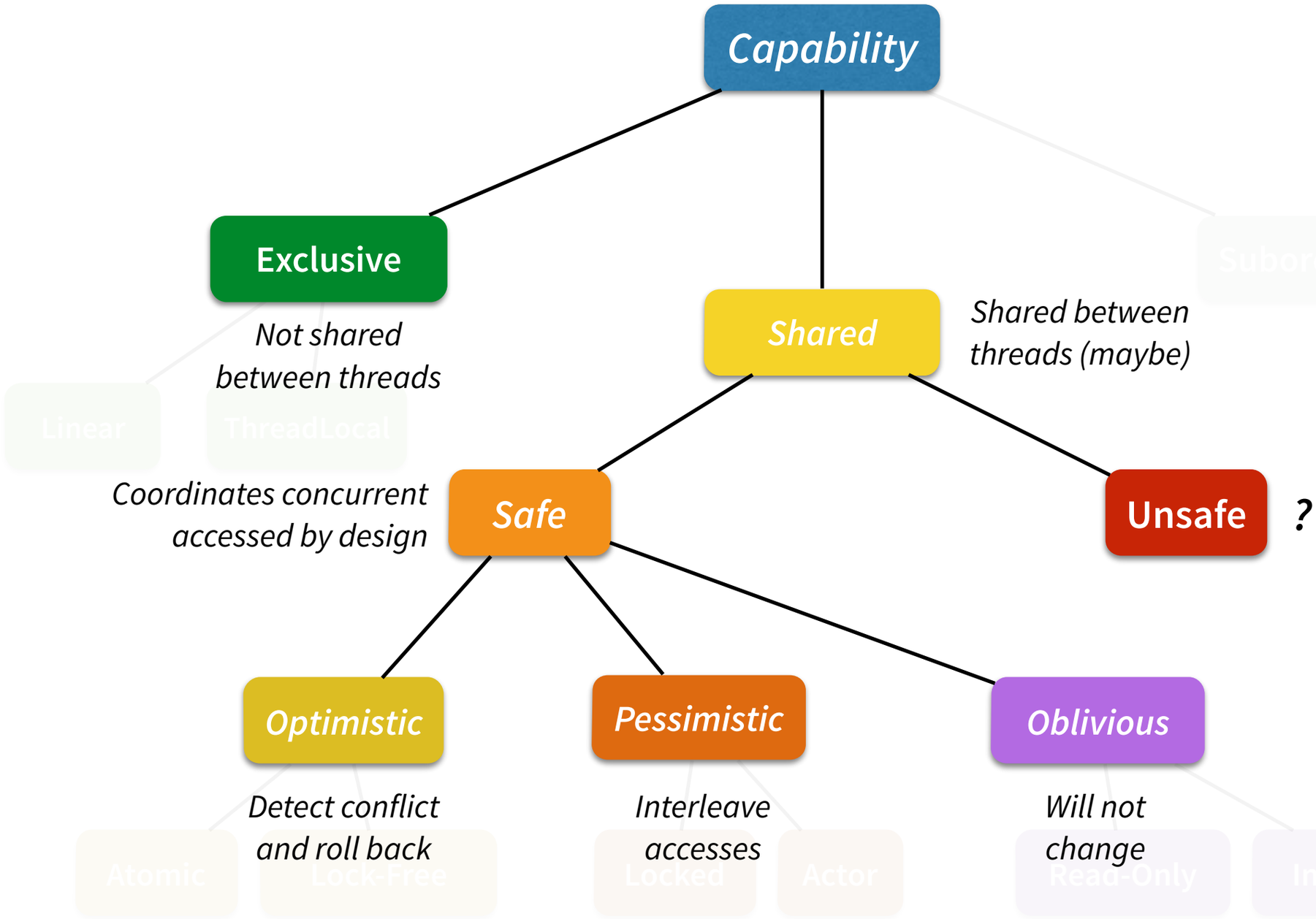
- Proliferation of ways in which resources are accessed

  Focus on interaction with objects shared across **multiple threads of control**

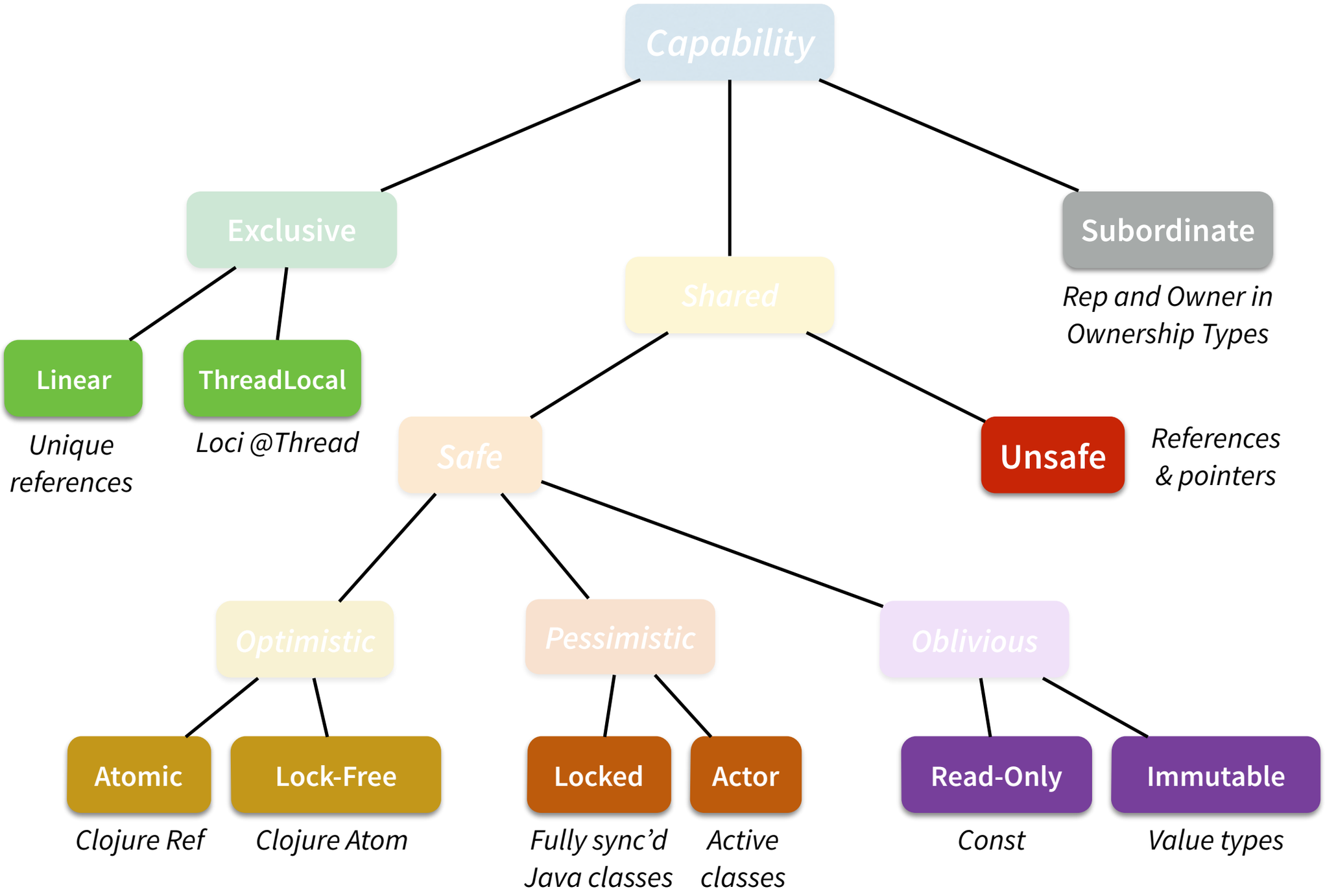
Hierarchy of Capabilities for (Non)Sharing



Hierarchy of Capabilities for (Non)Sharing



Hierarchy of Capabilities for (Non)Sharing



Exclusive Capabilities

```
x = y.f;      // assign from
x.bar();      // dereference
y.f = this;   // self type
```

- *Denote objects exclusive to a thread*

- **Linear**

  Only one active/usable reference to an object at any point in the program

  Strong properties (see previous talk)

- **Thread-Local**

  All references to an object are only reachable from one thread

  Weaker, but simpler to program with

Shared Capabilities (1/2)

- *Denote objects (that can be) shared across multiple threads*

```
x = y.f;      // assign from
x.bar();      // dereference
y.f = this;   // self type
```

- **Optimistic** — detect conflicts and roll back

  Atomic wrap operations in transactions

  Lock-Free enforce a protocol that gives rise to exclusivity

- **Pessimistic** — enforce interleaved accesses

  Locked require (some kind of) lock to be acquired before access

  Actor only allow asynchronous communication (object processes messages)

Shared Capabilities (2/2)

- *Denote objects (that can be) shared across multiple threads*

- **Oblivious** — object cannot change, so sharing is safe (wrt DRF)

  Read-Only a reference that cannot be used to observe/trigger mutation

  Immutable a reference to an object that cannot change

```
x = y.f;      // assign from
x.bar();      // dereference
y.f = this;   // self type
```

Misc. Capabilities

- *Denote objects (that can be) shared across multiple threads*

- **Subordinate** a reference to an object governed by another capability

  Inside exclusive or shared — DRF

  Inside unsafe — **?**

- **Unsafe**

  Alt. 1: move coordination to use-site

  Alt. 2: escape hatch to allow data races

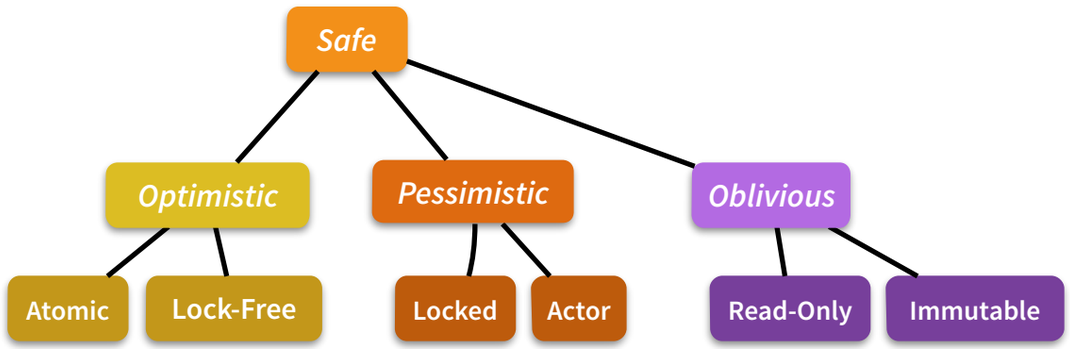
```
x = y.f;      // assign from
x.bar();      // dereference
y.f = this;   // self type
```

Polymorphic Concurrency Control [work in progress]

- *Safe* *require DRF, don't care how it is achieved*

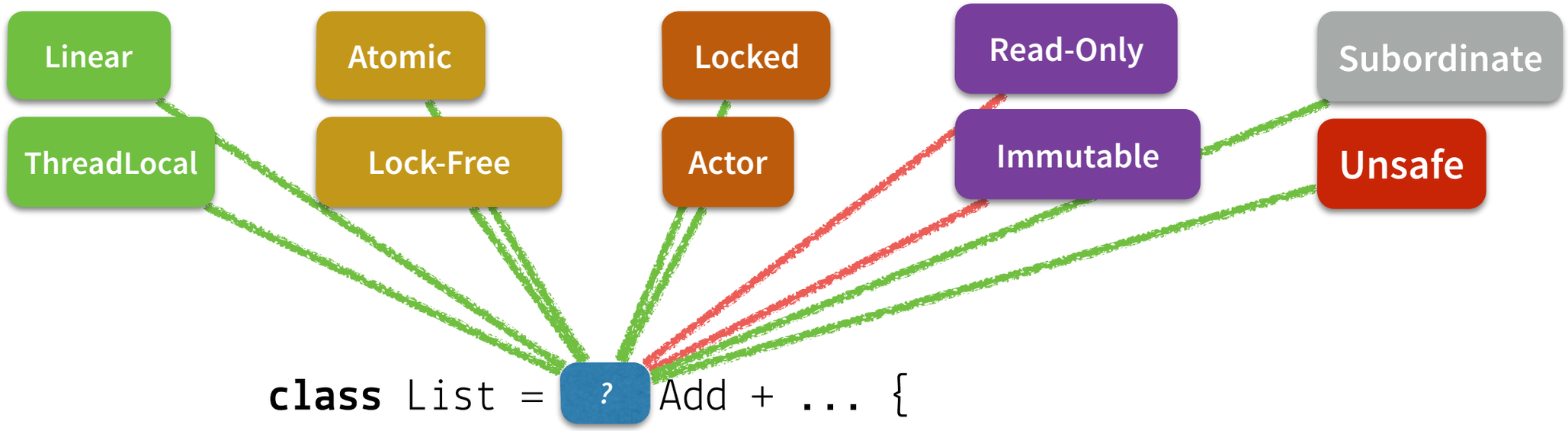
```
def summarise(es:[safe T]) : int
  let
    sum = 0
in {
  repeat i <- |es|
    sum += es[i].operation();
  sum;
}
```

Some interesting cases involving actors due to changing to async computations.



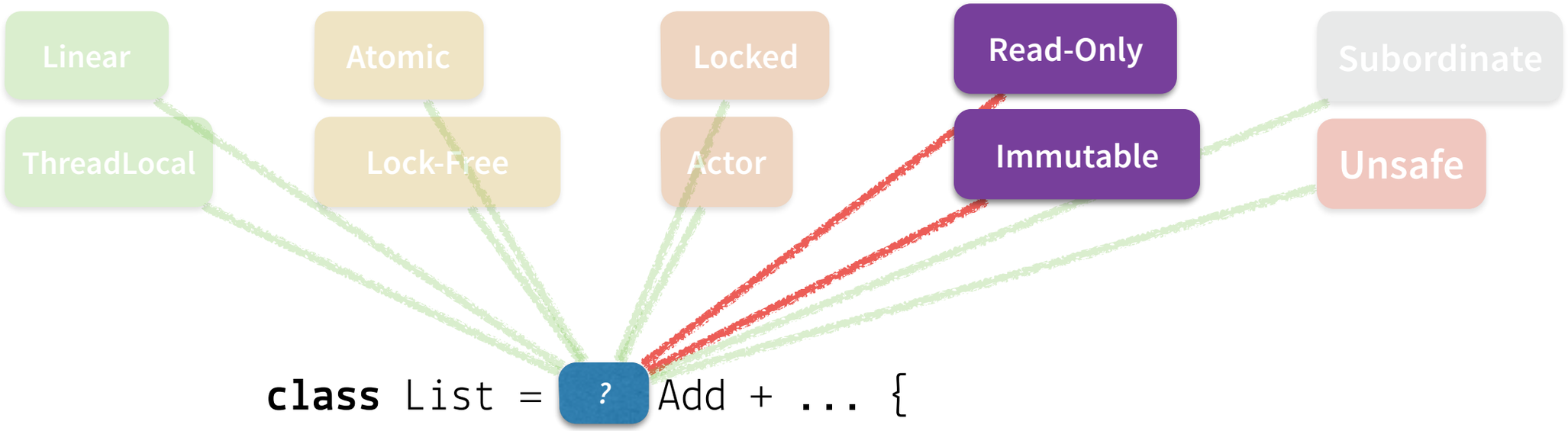
Capability = Trait + Mode

```
trait Add {
  require var first : Link;
  require var last : Link;
  def prepend(o:T) : void this.first = new Link(o, this.first);
  def append(o:T) : void this.last = new Link(o, null);
}
```



Capability = Trait + Mode

```
trait Add {
  require var first : Link;
  require var last : Link;
  def prepend(o:T) : void this.first = new Link(o, this.first);
  def append(o:T) : void this.last = new Link(o, null);
}
```



Traits Assume Race-Freedom

- Every trait may safely assume race-freedom

  How it is achieved is controlled by its *mode*

- Extending trait reuse to concurrent & parallel programming

  Creating classes from the same set of traits with different modes

  Cf. ArrayList (unsafe/linear/local) vs. Vector (safe, possibly read-locked) in Java APIs

- Capability composition using ⊕ and ⊗ follow simple rules to exclude races

Classes are Built from Traits

```
trait Add {
  require var first : Link;
  require var last : Link;
  ...
}
```

```
trait Remove {
  require var first : Link;
  require var last : Link;
  ...
}
```

OK!

```
class List = Add ⊕ Remove {
  var first : Link;
  var last : Link;
}
```

Classes are Built from Traits

```
trait Add {
  require var first : Link;
  require var last : Link;
  ...
}
```

```
trait Remove {
  require var first : Link;
  require var last : Link;
  ...
}
```

Rejected at compile-time

```
class List = Add ⊗ Remove {
  var first : Link;
  var last : Link;
}
```

Classes are Built from Traits

```
trait Left {
  require var left : Tree;
  ...
}
```

```
trait Right {
  require var right : Tree;
  ...
}
```

OK!

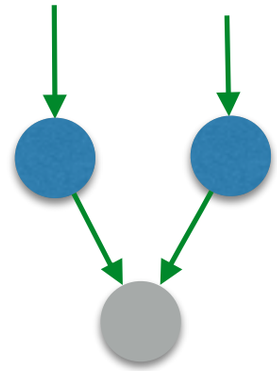
```
class Tree = Left ⊗ Right ⊗ ... {
  var left  : Tree;
  var right : Tree;
  ...
}
```

Capability Composition and State Sharing

prevents indirect sharing

T-COMPOSITION

$$\frac{\textbf{may-alias}(Fd_1, Fd_2) \implies \odot = \oplus \qquad \vdash \mathtt{K}_1 \qquad \vdash \mathtt{K}_2 \qquad \exists\, k\ .\ k(\mathtt{K}_1 \odot \mathtt{K}_2) \qquad \forall\, Fd_1 \in \textbf{fields}\,(\mathtt{K}_1), Fd_2 \in \textbf{fields}\,(\mathtt{K}_2)\ .\ Fd_1 \odot Fd_2}{\vdash \mathtt{K}_1 \odot \mathtt{K}_2}$$



$\boxed{Fd_1 \odot Fd_2}$ *(sharing fields across traits)*

C-DETERMINISTIC

$$\frac{\textbf{stable}(t)}{\mathtt{val}\, f : t \otimes \mathtt{val}\, f : t}$$

C-NONDETERMINISTIC

$$\frac{\textbf{safe}(t) \qquad \neg\, \textbf{stable}(t)}{\mathtt{val}\, f : t \otimes \mathtt{val}\, f : t}$$

C-VAL-VAL

$$\frac{\vdash \mathtt{K}_1 \otimes \mathtt{K}_2}{\mathtt{val}\, f : \mathtt{K}_1 \odot \mathtt{val}\, f : \mathtt{K}_2}$$

C-VAR-VAL

$$\frac{t_1 <: t_2}{\mathtt{var}\, f : t_1 \oplus \mathtt{val}\, f : t_2}$$

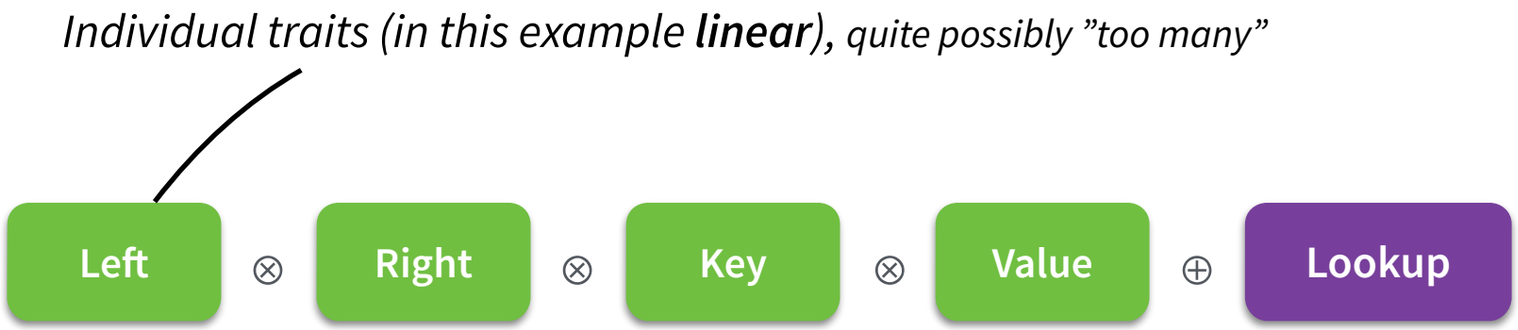
C-DISJOINT

$$\frac{f_1 \neq f_2}{mod_1\, f_1 : t_1 \odot mod_2\, f_2 : t_2}$$

C-DISJUNCTION

$$\frac{}{mod_1\, f : t \oplus mod_2\, f : t}$$

Constructing A Type: Tree Example




```
class Tree = (Left ⊗ Right ⊗ Key ⊗ Value) ⊕ Lookup { … }
```

- Possible to operate on left and right subtrees, key and value ***in parallel***

  E.g., `t : Tree` allows `a : Left + b : Right = t;`

- Mediate between mutable (unaliased) and read-only (aliased) views (cf. fractional perms.)

  E.g., `t : Tree` allows `a : 𝔍(Left ⊗ Right ⊗ Key ⊗ Value) + b : Lookup = t;`

Jail

Co-Encapsulation

```
class Tree<K> = (Left<K> ⊗ … ⊗ Value<K>) ⊕ Lookup<K> { … }
```

- Exposing nested capabilities in type, allows top-level operations on them

  E.g. `t : Tree<Person>` where `Person = Name ⊗ Age` allows

```
l : Lookup<Person> ⊗ tmp : ℑ(…) = t;
nl : Lookup<Name> ⊕ al : Lookup<Age> = l;
t = nl ⊕ al ⊕ tmp;
```

- Two forms of unpacking, depending on mode of co-encapsulating capability

  readonly `Lookup<K>` — *only external ops allowed by "reverse borrowing"*

  class `Tree<Person>` =
  … **linear** `Lookup<Name>` ⊗ **linear** `Lookup<Age>` … — *internal ops allowed*

Unpacking and Packing

(what I omitted on the previous slide)

Must keep track of forgotten parts of type!

Use "jail" to re-pack type

```
class List<T> = Take<T> ⊕ Put<T> ⊕ Look<T> {
  Link<T> first;
}

List<Pair> a;

𝔍[Take<Pair> ⊕ Put<Pair>] j, Look<Pair> b = consume a;
j.foo(); // rejected @ CT!
Look<Pair> c = b; // rejected @ CT

Look<Cell> d, e = consume b;

finish {
  async { operate on d }
  async { operate on e }
}

b = d + e;
a = b + j;
```

Structured (Scoped) Equivalent

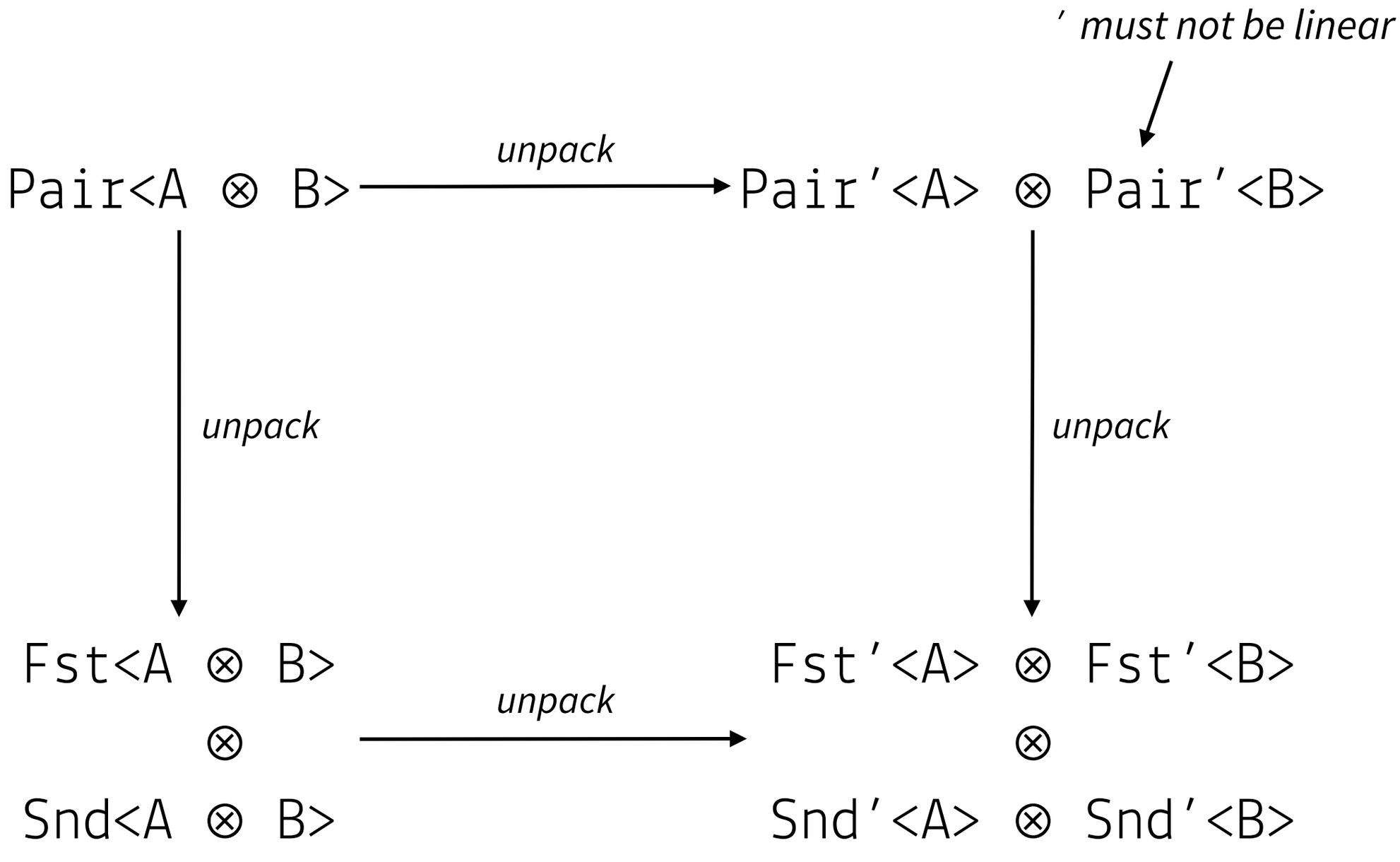
(logically desugars to the code on the previous slide)

```
class List<T> = Take<T> ⊕ Put<T> ⊕ Look<T> {
  Link<T> first;
}

List<Pair> a;

finish(d, e : Look<Cell> = a) {
  async { operate on d }
  async { operate on e }
}
```

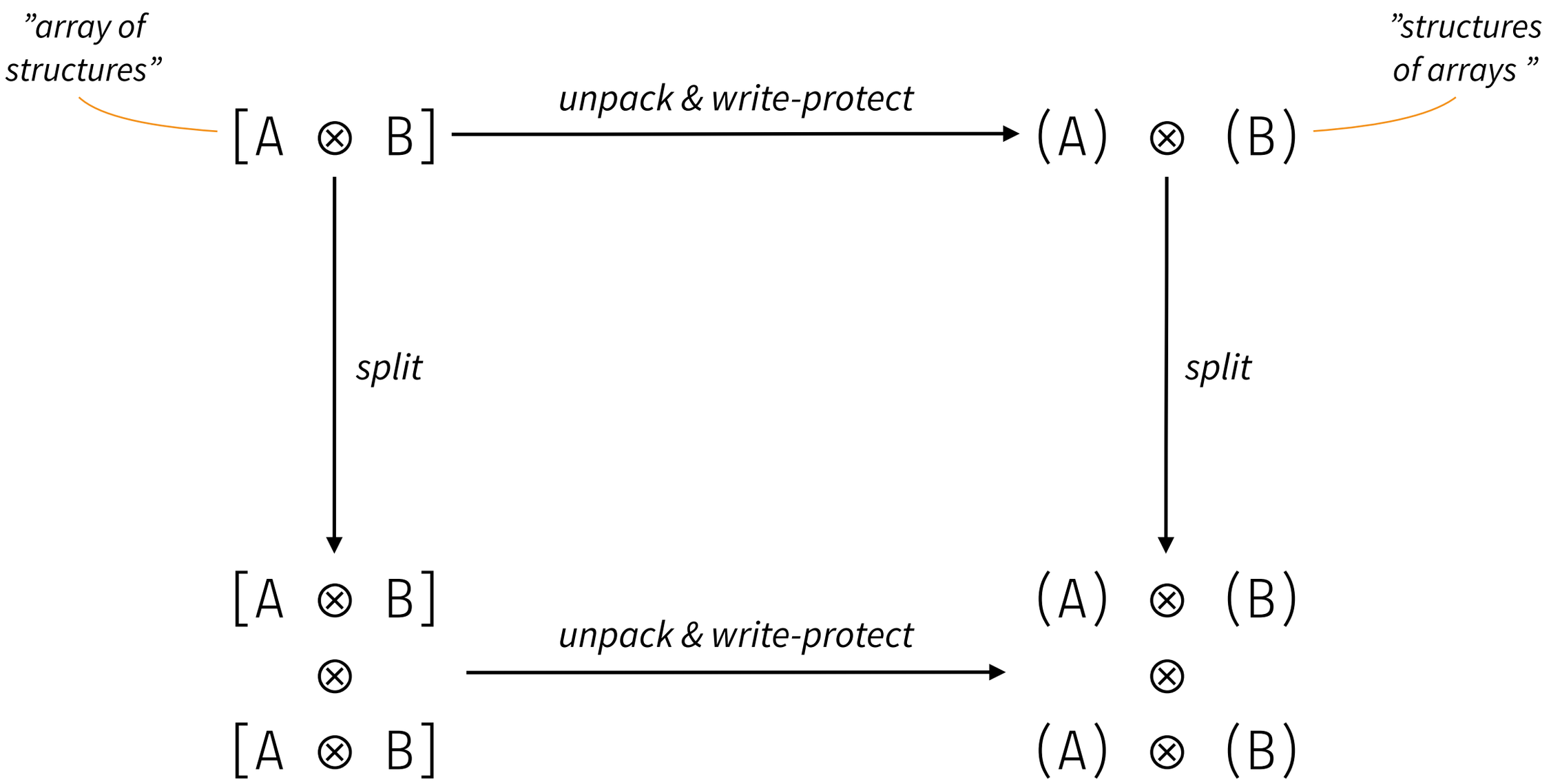
temporary unpacking → `finish(d, e : Look<Cell> = a) {`

implicit repacking → `}`

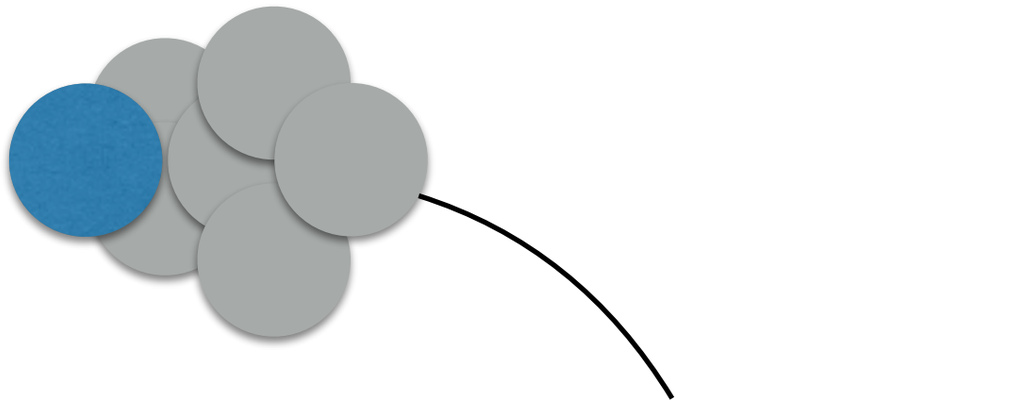
Unpacking Composite Capabilities



[Arrays] and (Tuples)

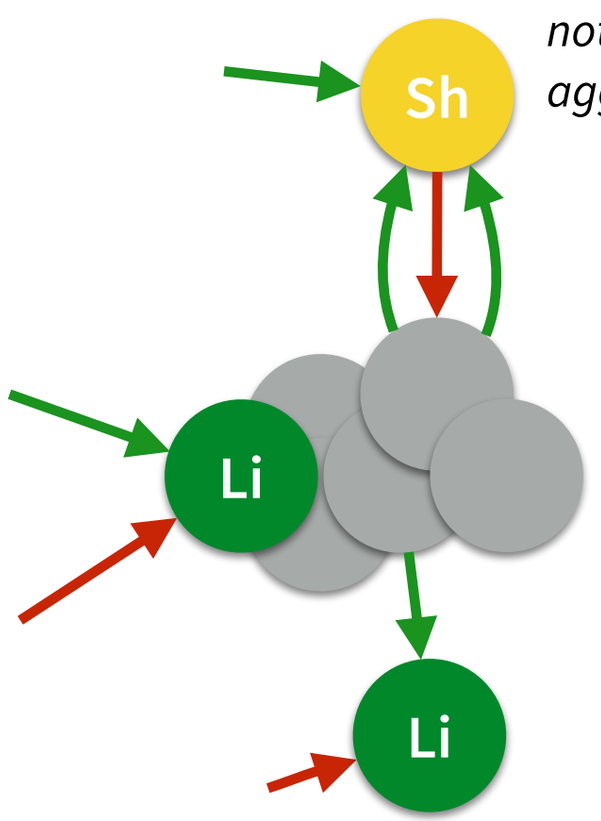


Capabilities are Dominators



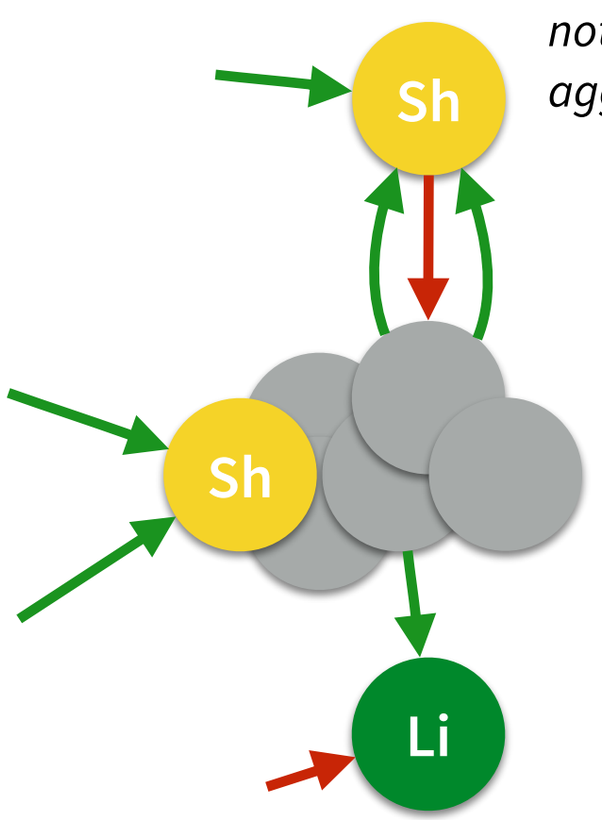
Subordinate objects are dominated by the exclusive ”bridge”

Encapsulation of State under Bridge



not part of aggregate

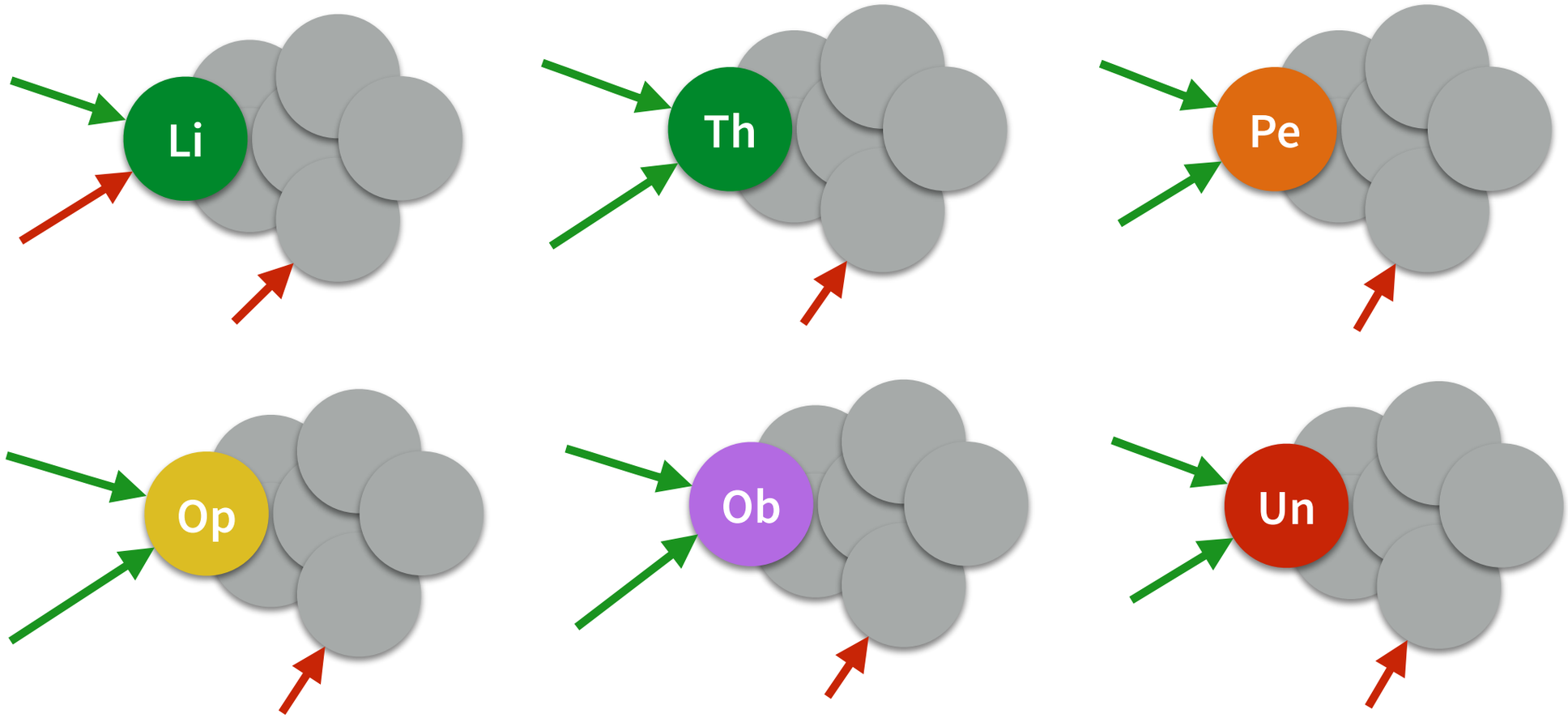
effectively part of the aggregate



not part of aggregate

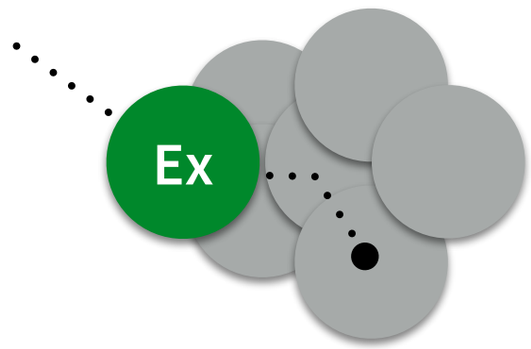
effectively part of the aggregate

Aliasing of Bridge Objects

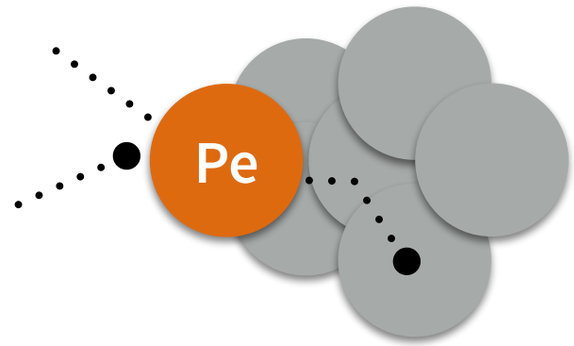


DR(F) under Bridge

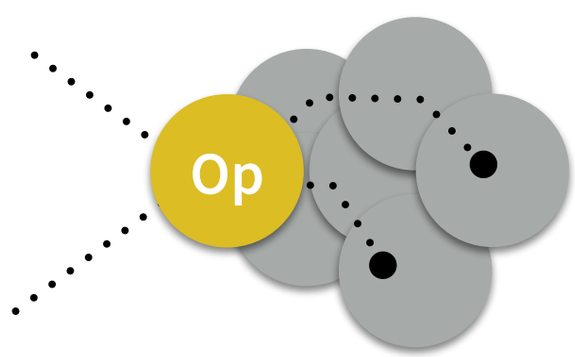
only one thread has access



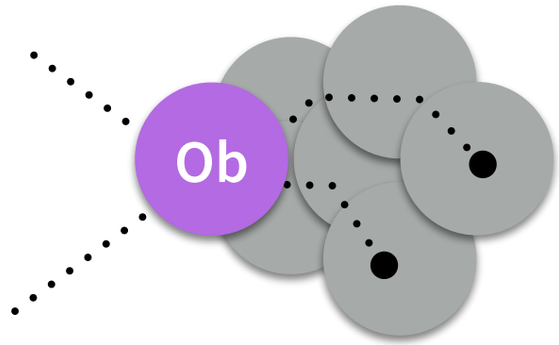
interaction serialised



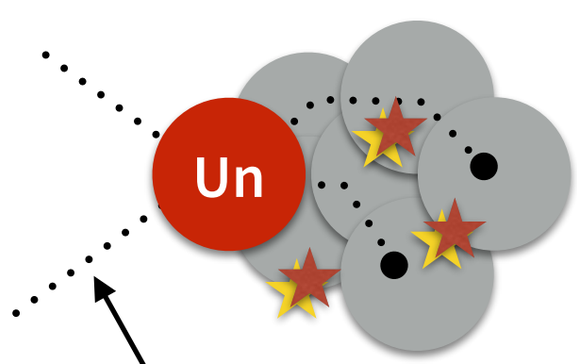
racy



conflicts rolled-back

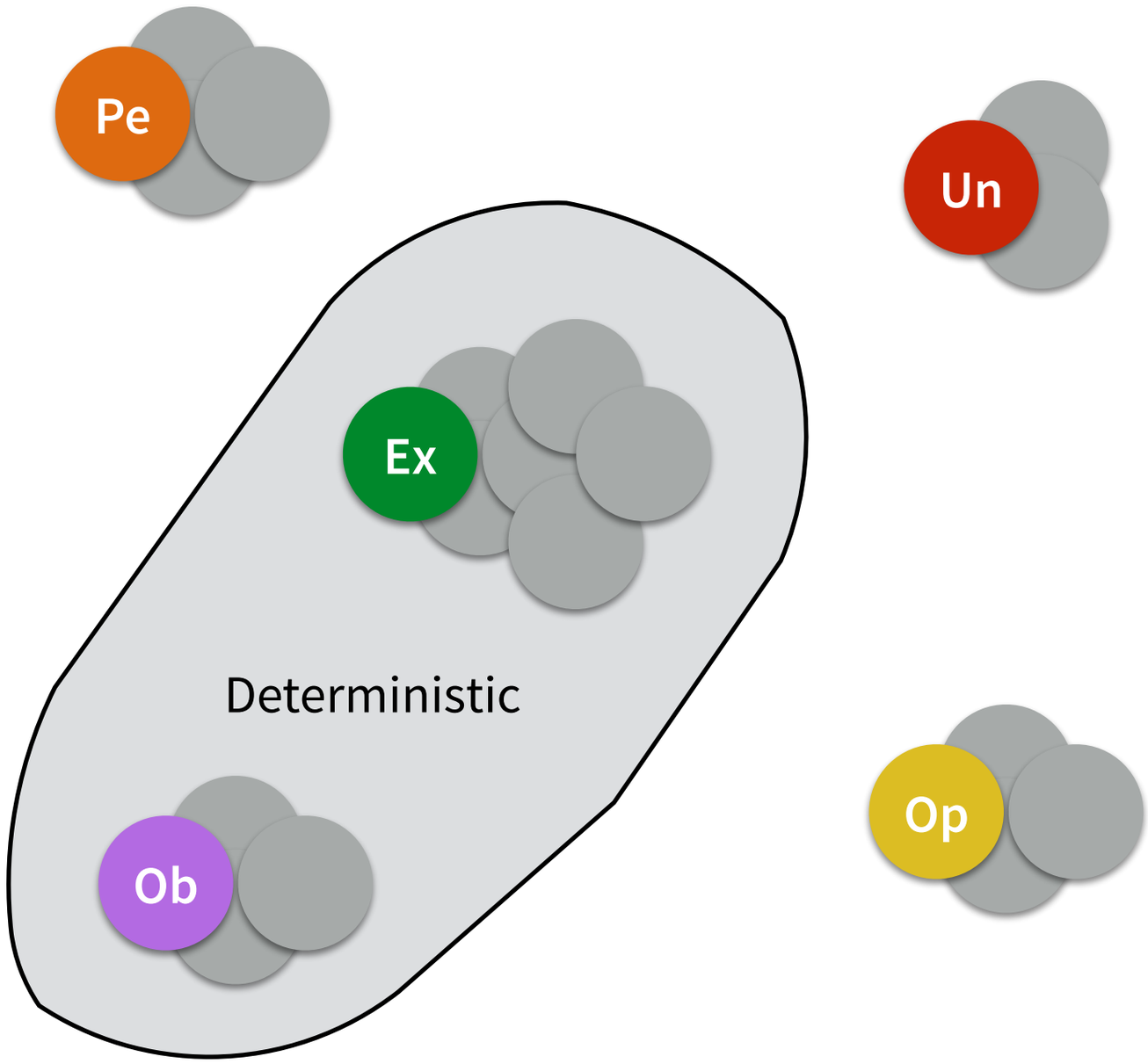


no mutable state



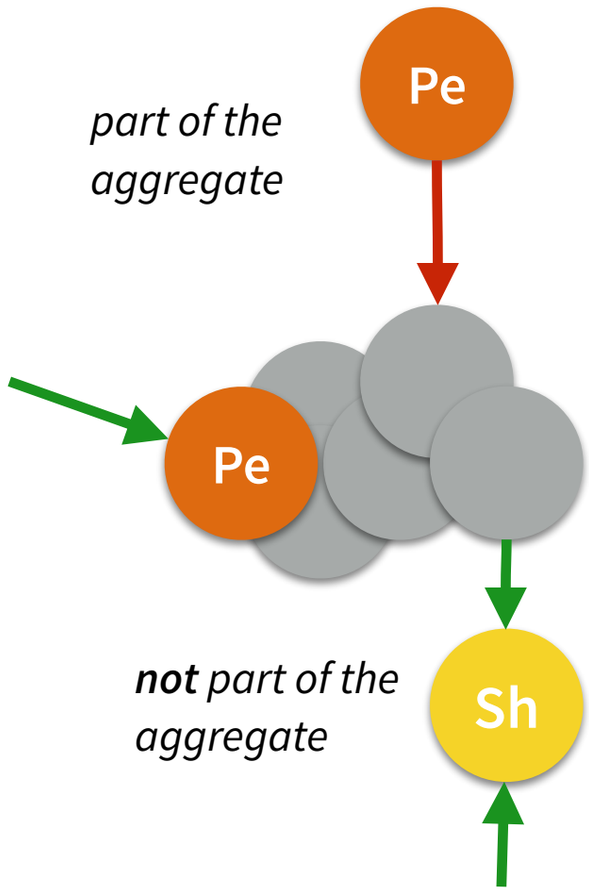
(or require sync at use-site)

Identifiable (Non-)Determinism



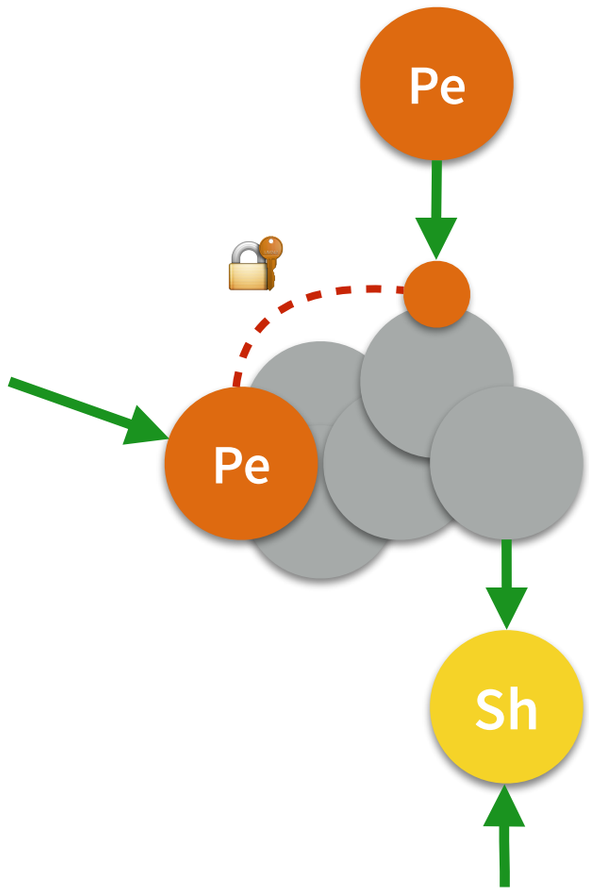
Ownership & Synchronisation

owners-as-dominators



all sharing immediate from code

owners-as-coordinators

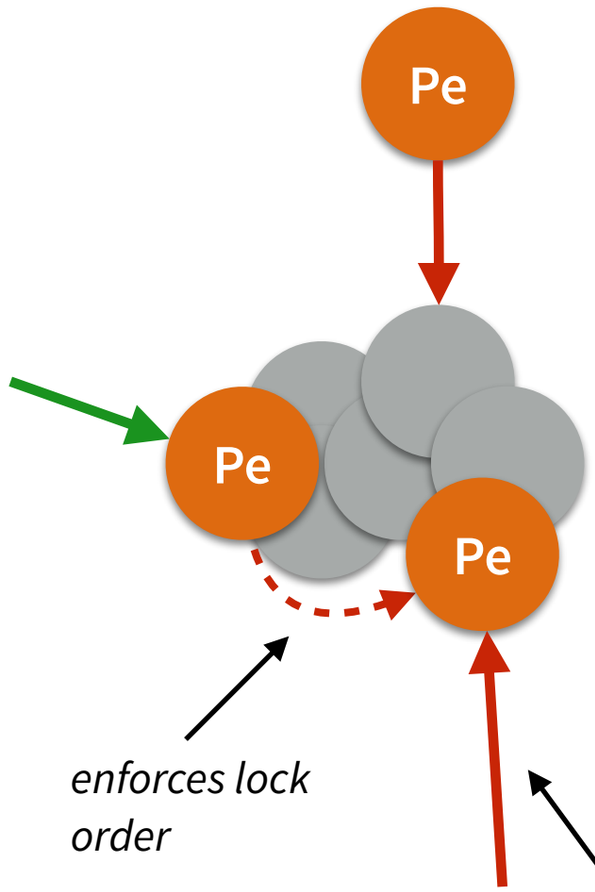


leaked internal pointers become synchronised

works for all shared capabilities

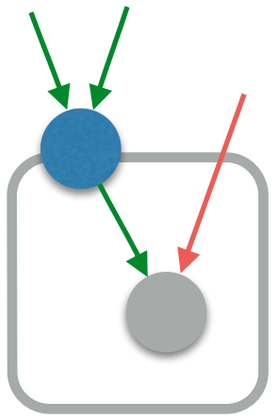
sharing less immediate in code

full ownership model

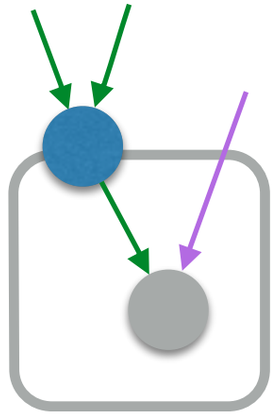


Ownership Variants

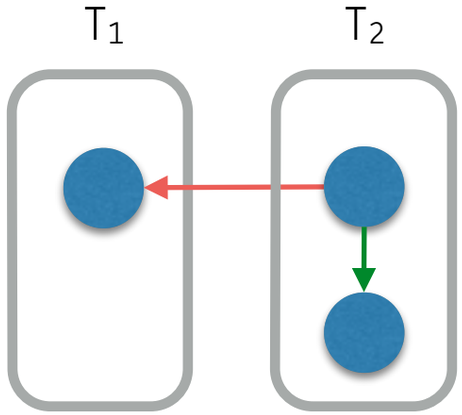
dominators



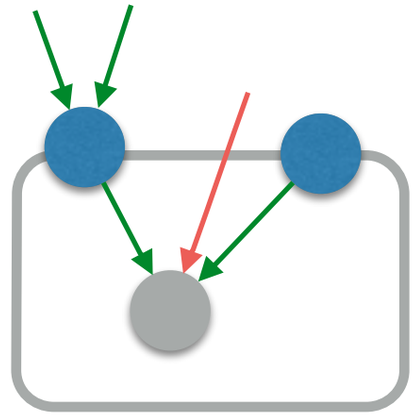
modifiers



threads

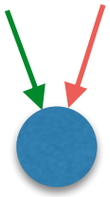


ombudsmen

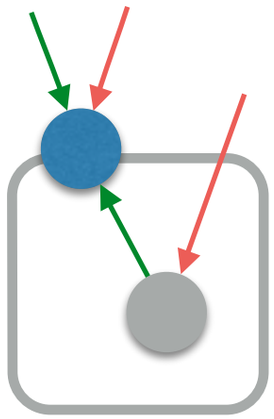


Uniqueness Variants

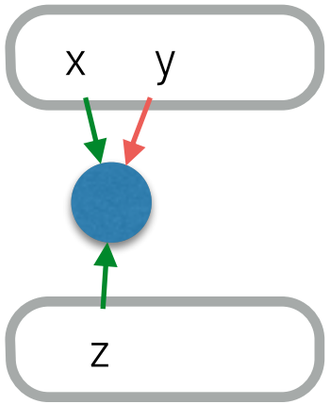
classical



external

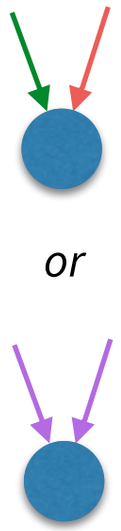


local / refined ownership



"Mix"

fractional permissions



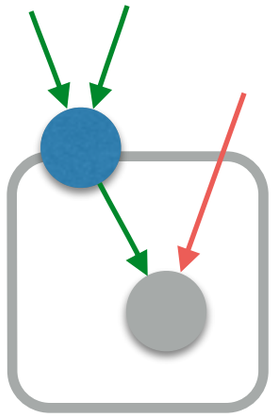
normal

read-only

disallowed

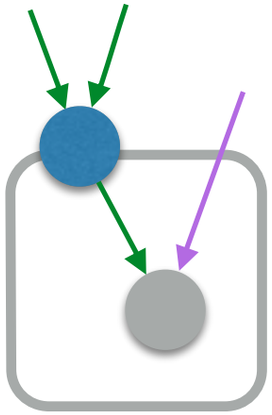
Ownership Variants

dominators



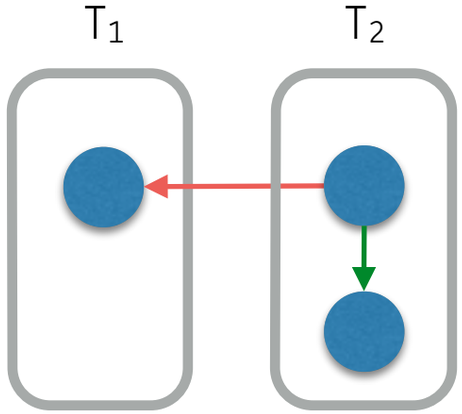
… Subordinate

modifiers



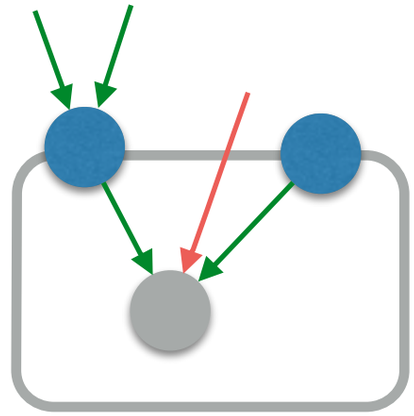
Subordinate + 🔑

threads



Thread Local

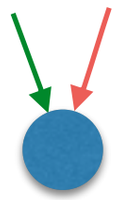
ombudsmen



… ⊗ …

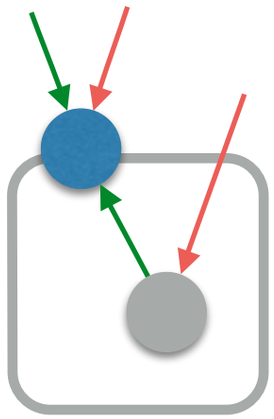
Uniqueness Variants

classical



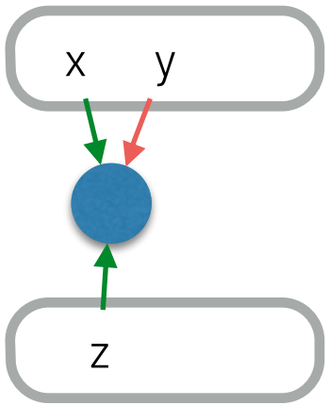
Linear

external



Linear Subordinate

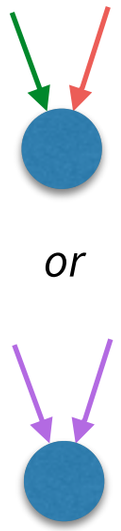
local / refined ownership



Linear ∨

"Mix"

fractional permissions

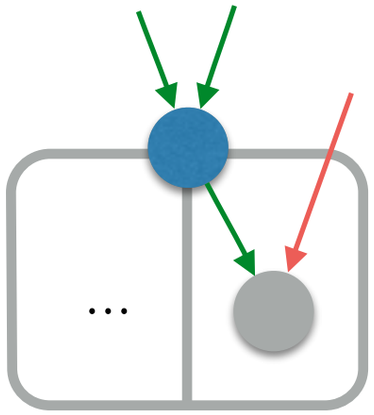


Linear ⊕ *Oblivious*

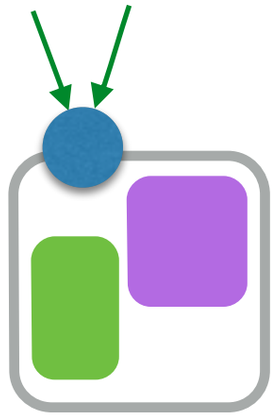
or

Static Control of Side-Effects

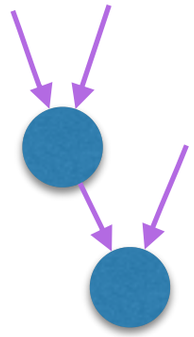
regions

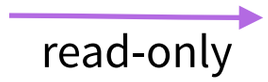


effect systems



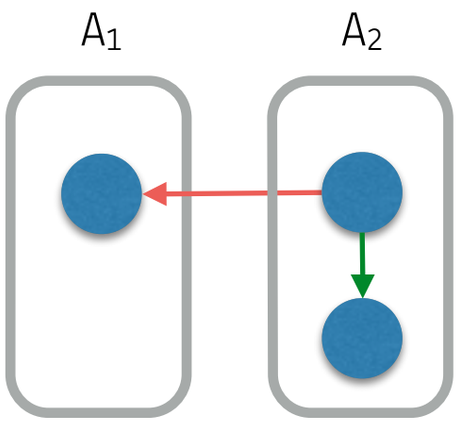
immutability



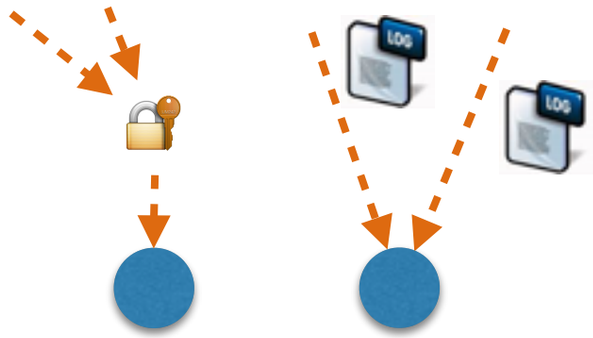


Dynamic Control of Side-Effects

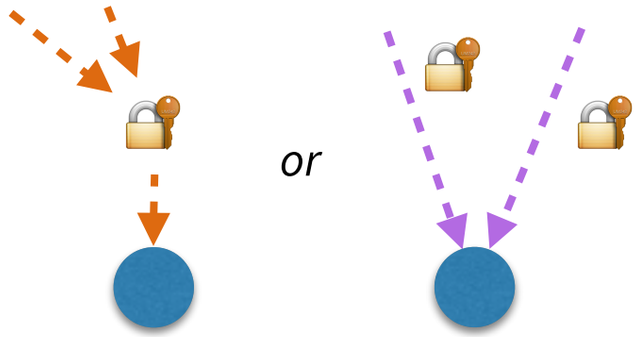
active object isolation



locks or transactions

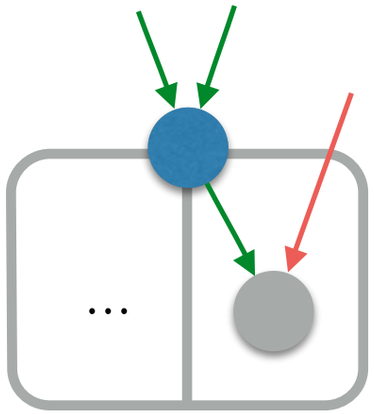


readers-writer lock



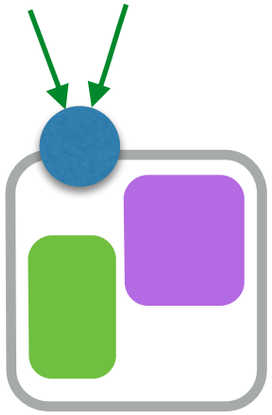
Static Control of Side-Effects

regions



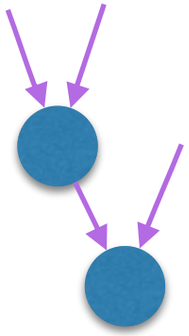
⊗

effect systems



capabilities…

immutability



Immutable

normal

read-only

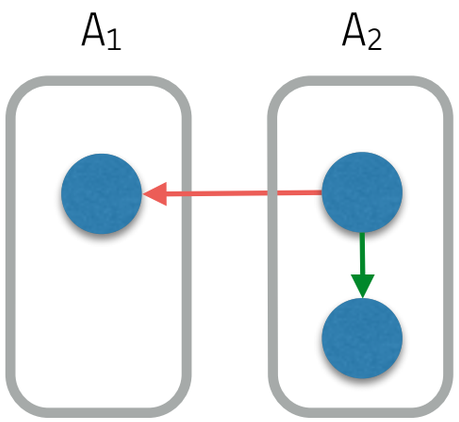
disallowed

access

read access

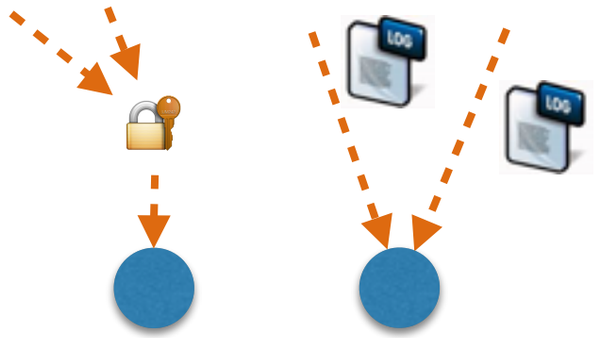
Dynamic Control of Side-Effects

active object isolation



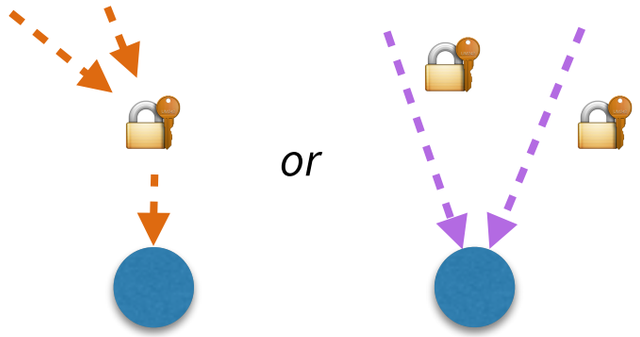
Actor

locks or transactions



Locked **Atomic**

readers-writer lock



Locked ⊕ ***Oblivious***

Thank you.

Questions?