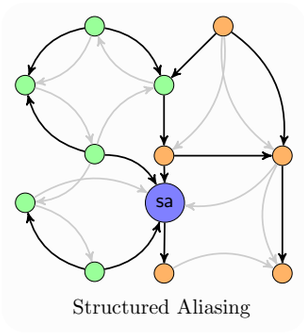
Reference Capabilities for Concurrency Control

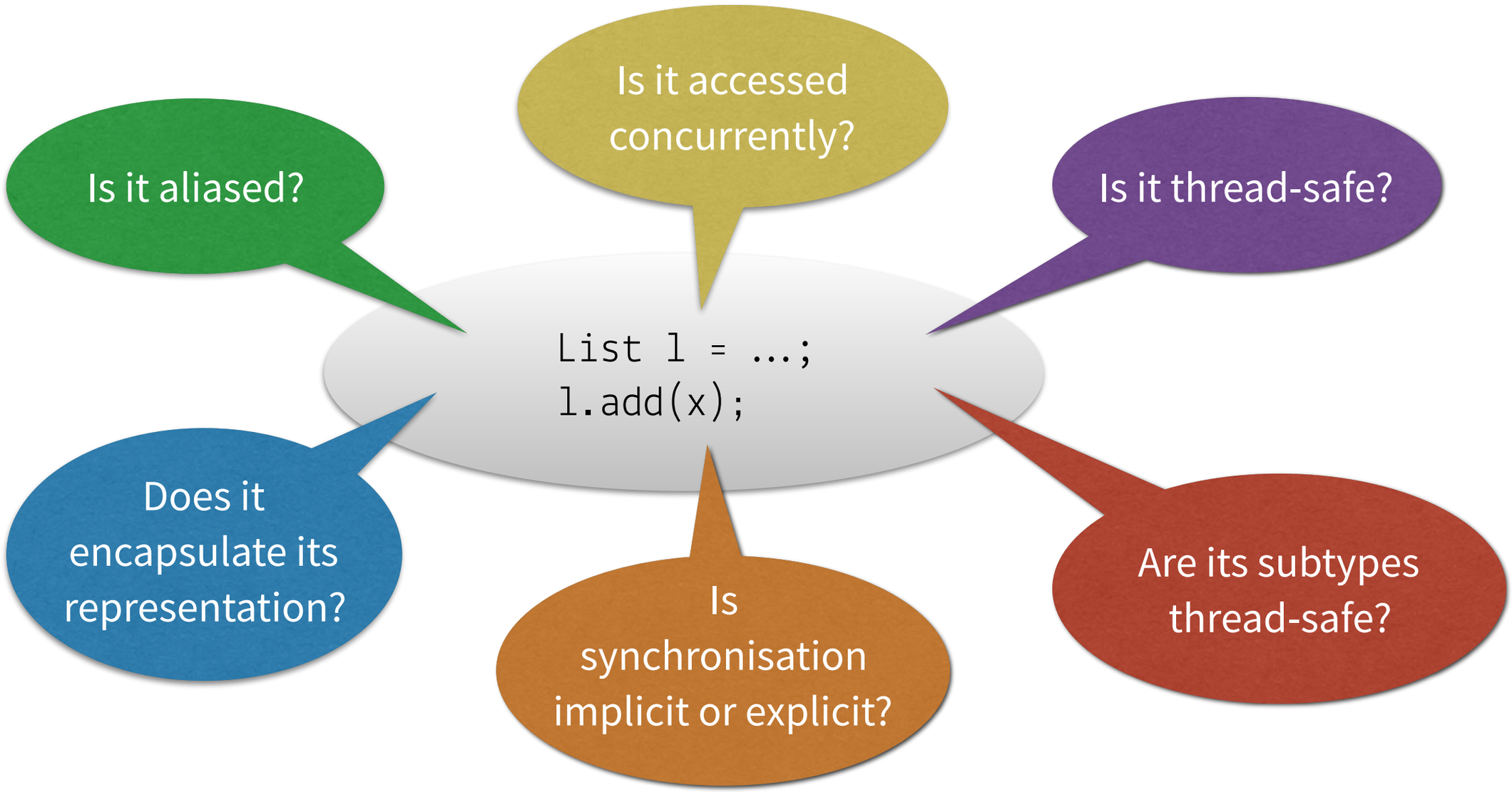
Elias Castegren, Tobias Wrigstad

ECOOP'16

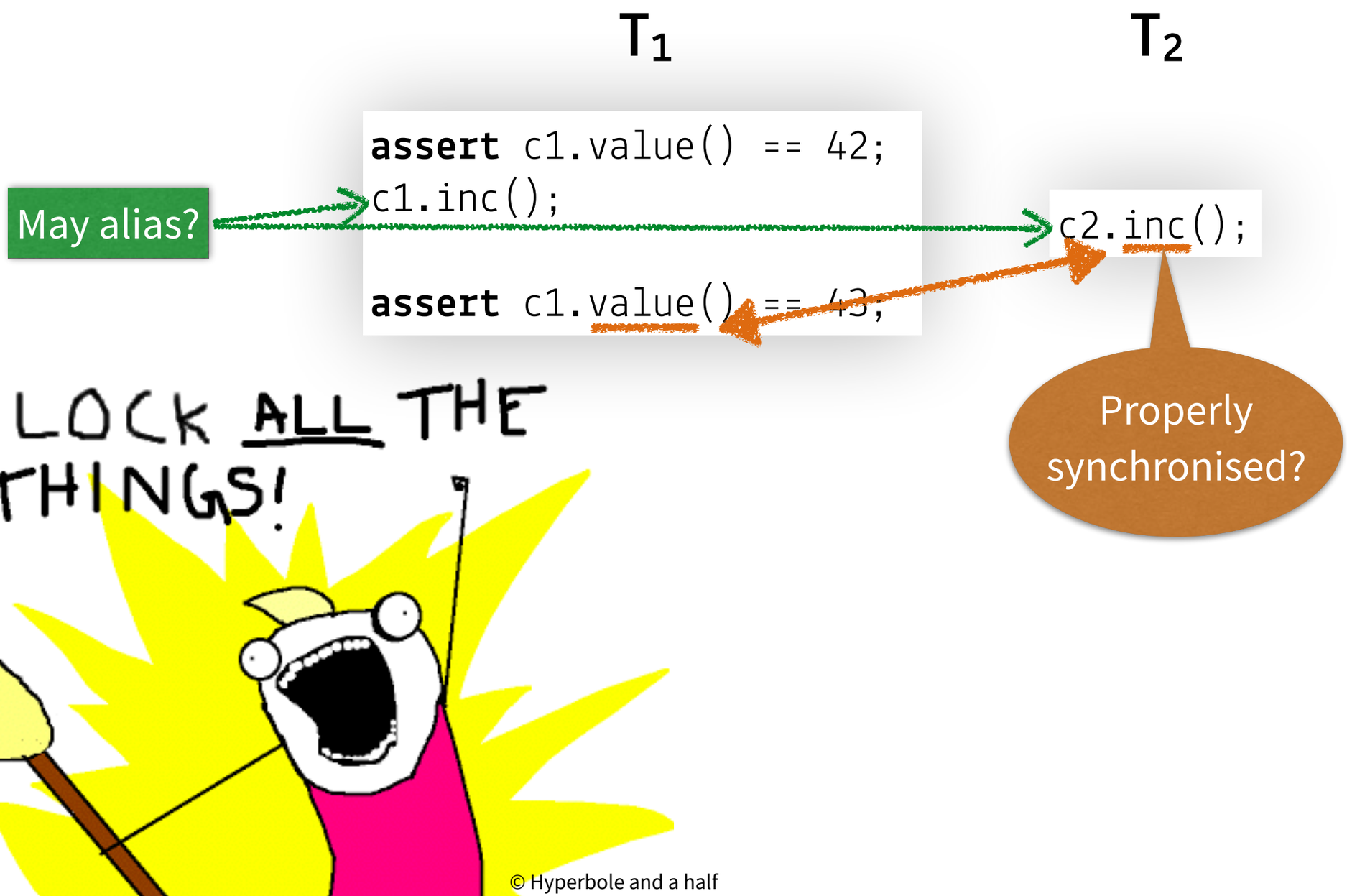


Structured Aliasing

Concurrency Imposes Many Concerns



Aliasing and Concurrency Control



© Hyperbole and a half

Aliasing and Concurrency Control

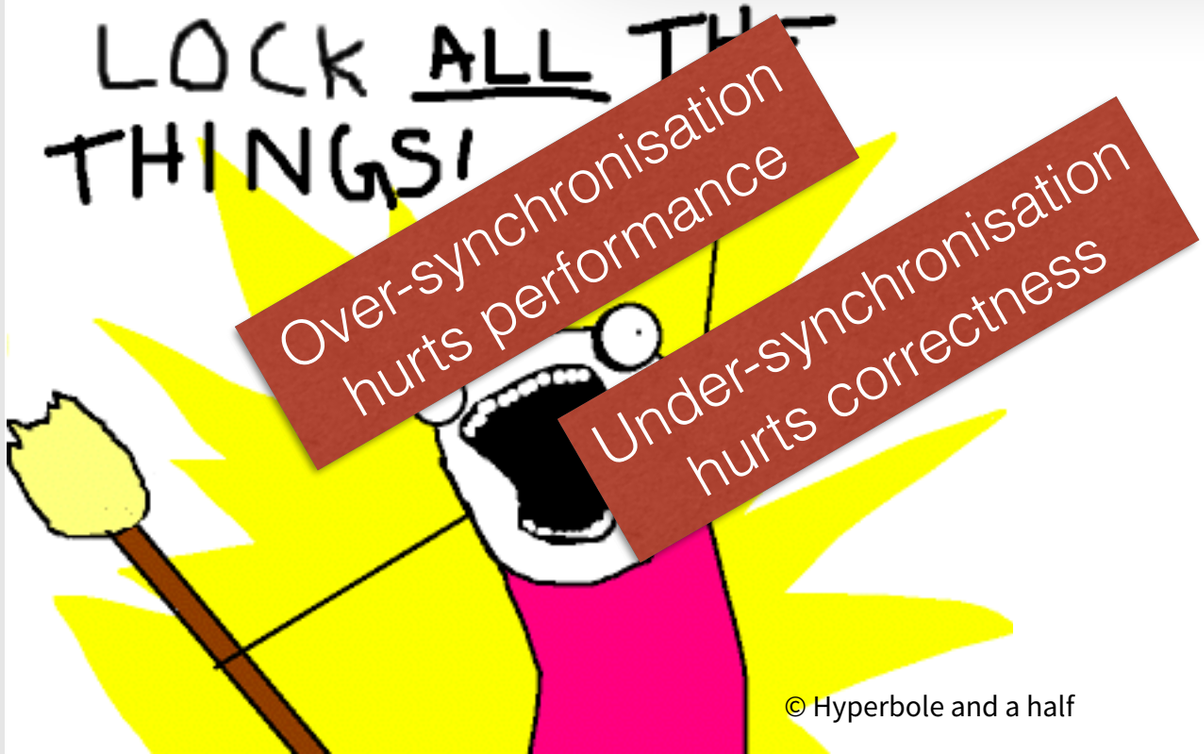
T_1 T_2

```
assert c1.value() == 42;
c1.inc();
assert c1.value() == 43;
```

```
c2.inc();
```

May alias?

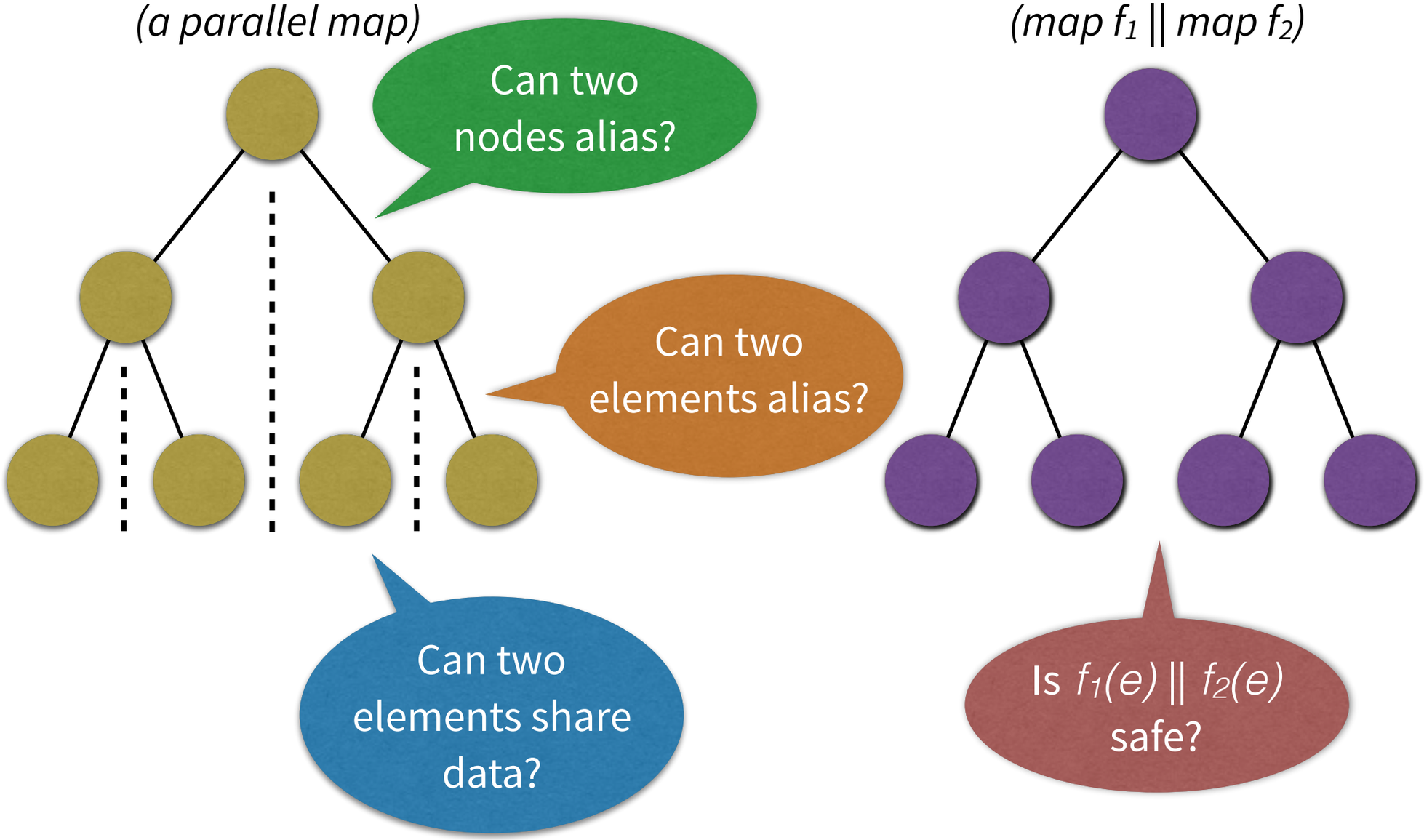
Properly synchronised?



© Hyperbole and a half

Aliasing and Parallelism

Data parallelism
(a parallel map)



Task parallelism
(map f_1 || map f_2)

Concurrency and Code Reuse

```
class List<T>
  var first : Link<T>

  def add(elem : T) : void
    ...



def remove(i : int) : T
  ...



def lookup(i : int) : T
  ...

...
```

```
class SynchronizedList<T>
  var list : List<T>

  def add(elem : T) : void
    lock();
    this.list.add(elem);
    unlock();

  def remove(i : int) : T
    lock();
    tmp = this.list.remove(i);
    unlock();
    return tmp;
  def lookup(i : int) : T
    lock();
    tmp = this.list.remove(i);
    unlock();
    return tmp
  ...
```

Summary of Motivation

- Code that needs concurrency control is indistinguishable from code that does not

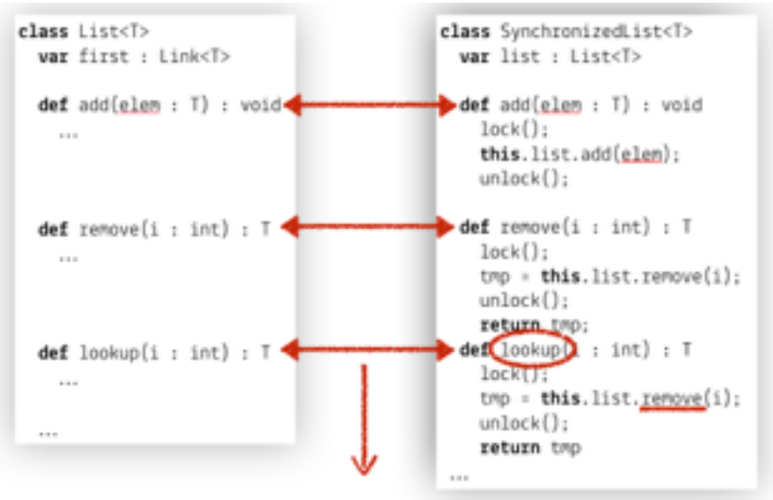
  Correct synchronisation warrants program-wide aliasing analysis

```
c2.inc();
```

Thread-safe?

- Concurrency control varies across different usage scenarios

  Building concurrency control into a data-structure generates overhead



- Business logic and concurrency control are often orthogonal concerns

```
lock(); ...; unlock();
```

Reference Capabilities

- A capability grants access to some ~~resource~~ object

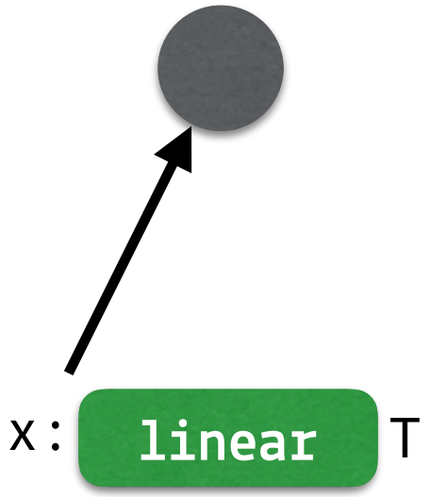
reference

- The type of a capability defines the interface to its object

- A capability assumes exclusive access

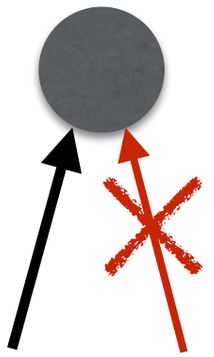
  Thread-safety ⇒ No data-races

- How thread-safety is achieved is controlled by the capability's *mode*

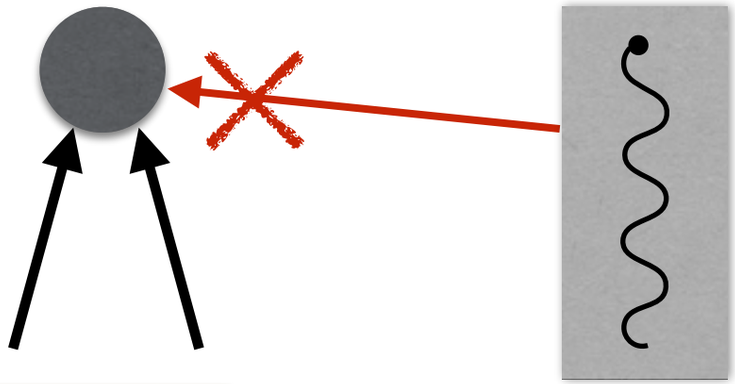


Modes of Concurrency Control

- *Exclusive* modes

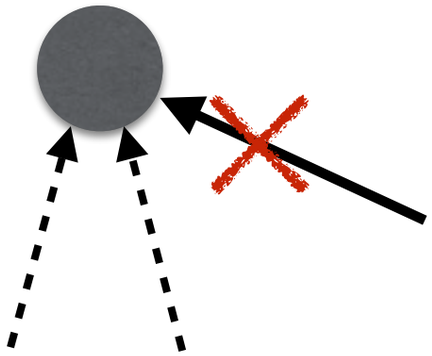


`linear` Globally unique



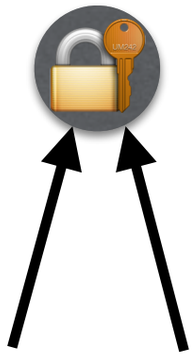
`thread` Thread-local

- *Safe* modes



`read`

Precludes mutating aliases



`locked`

Implicit locking

Modes of Concurrency Control

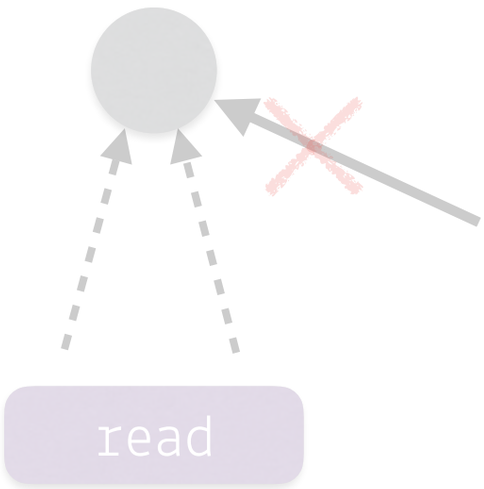
Dominating modes

`linear`

`thread`

`locked`

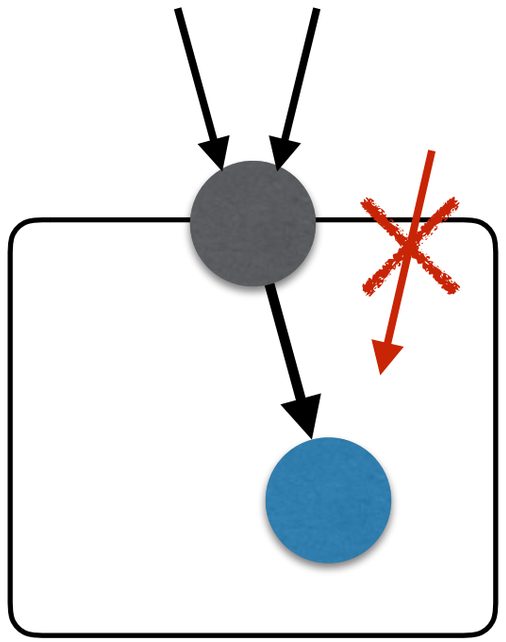
Guarantees
mutual exclusion



`read`

Precludes mutating aliases

Subordinate mode



`subordinate`

Encapsulated

Capability = Trait + Mode

- Capabilities are introduced via traits

```
trait Inc
  require var cnt : int
  def inc() : void
    this.cnt++;
```

If included in a class with a field `cnt : int`...

...I will provide a method `inc()`

```
trait Get
  require val cnt : int
  def value() : int
    return this.cnt;
```

- Modes control *why* they are safe

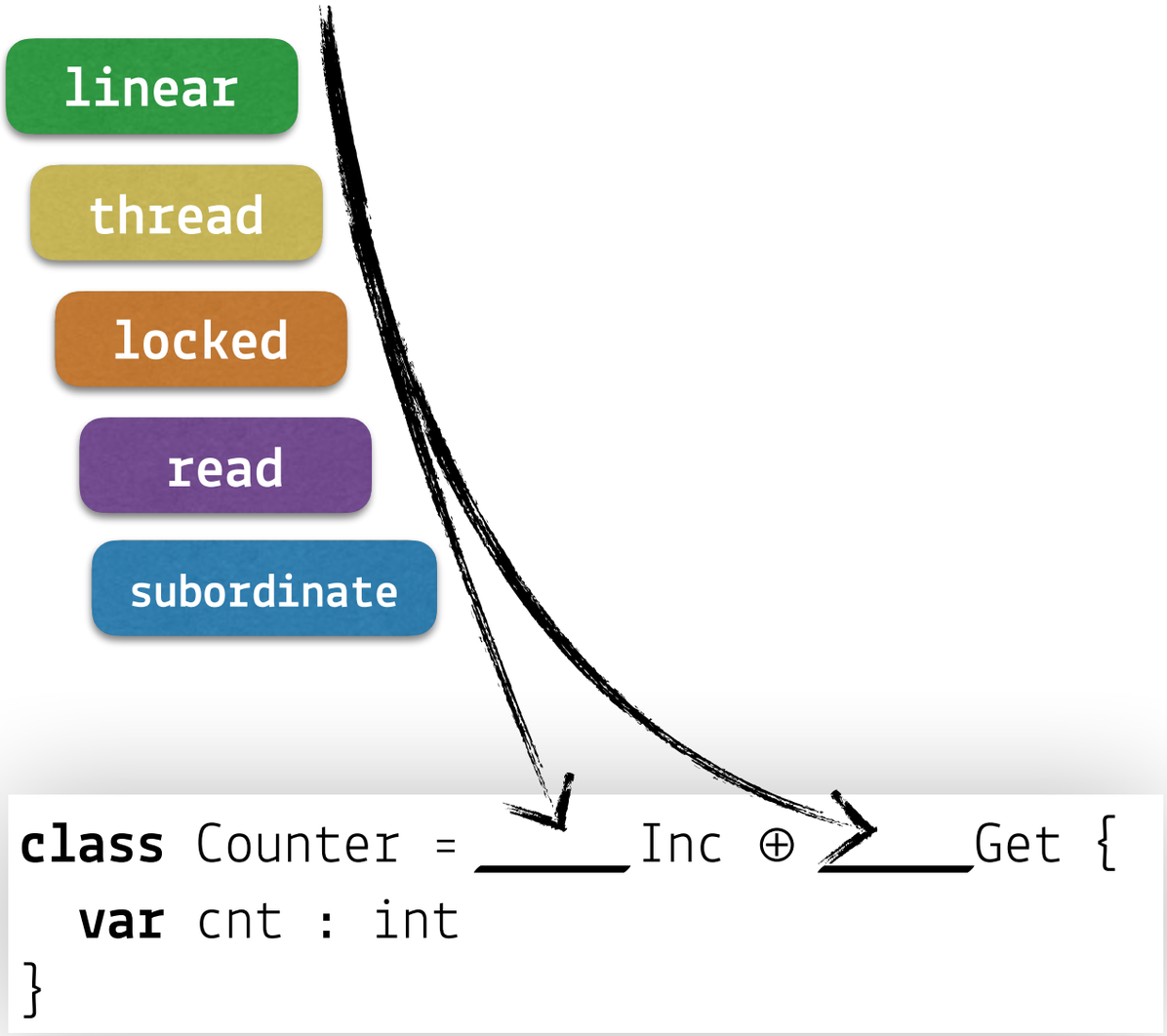
linear `Inc` — Globally unique increment capability

locked `Inc` — Implicitly synchronised increment capability

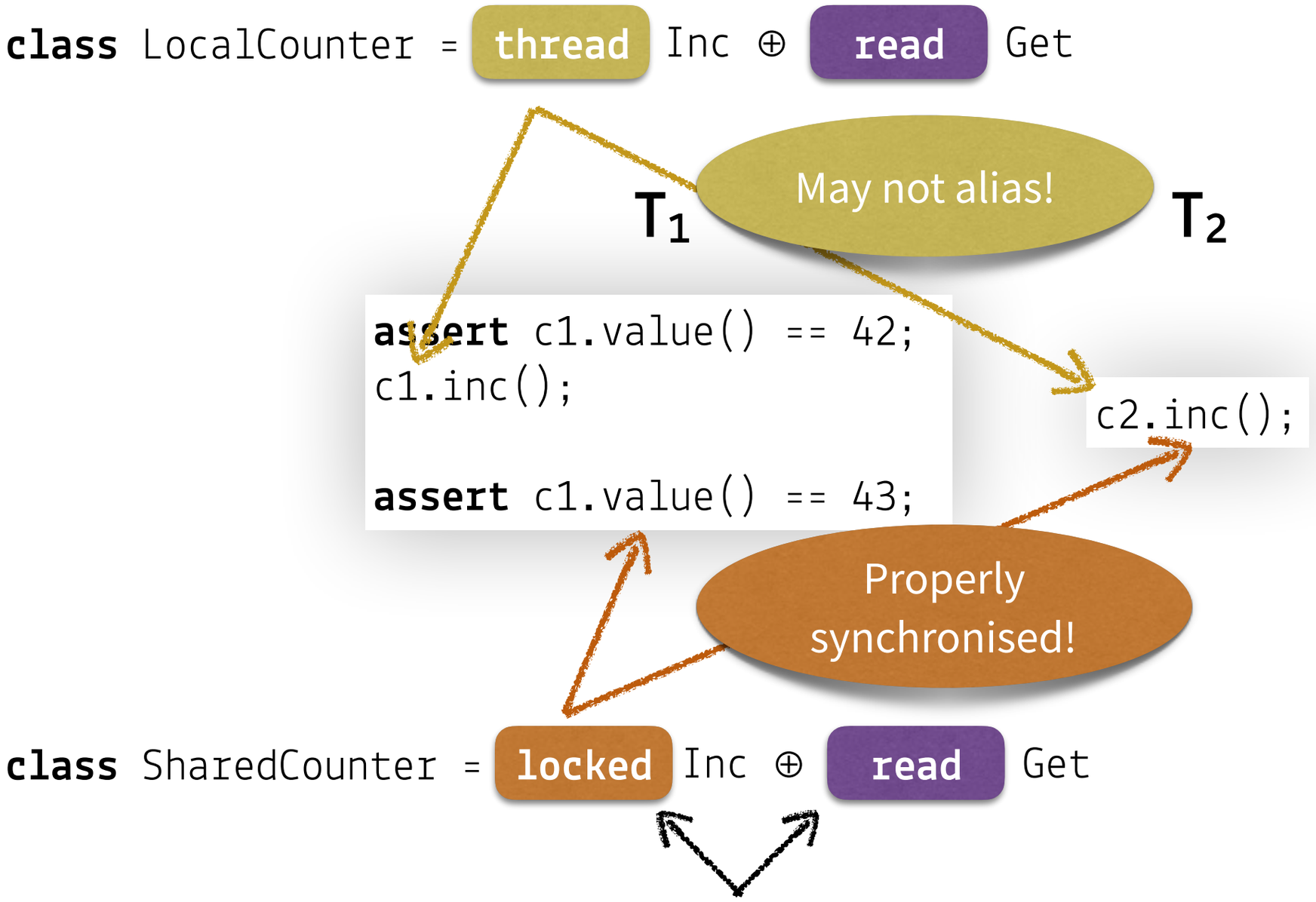
~~**read** `Inc` — A read-only increment capability~~

read `Get` — A read-only capability for getting the value

Classes are Composed by Capabilities

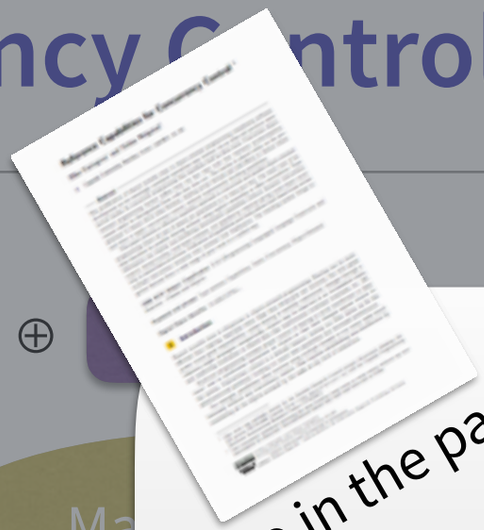


Aliasing and Concurrency Control (revisited)

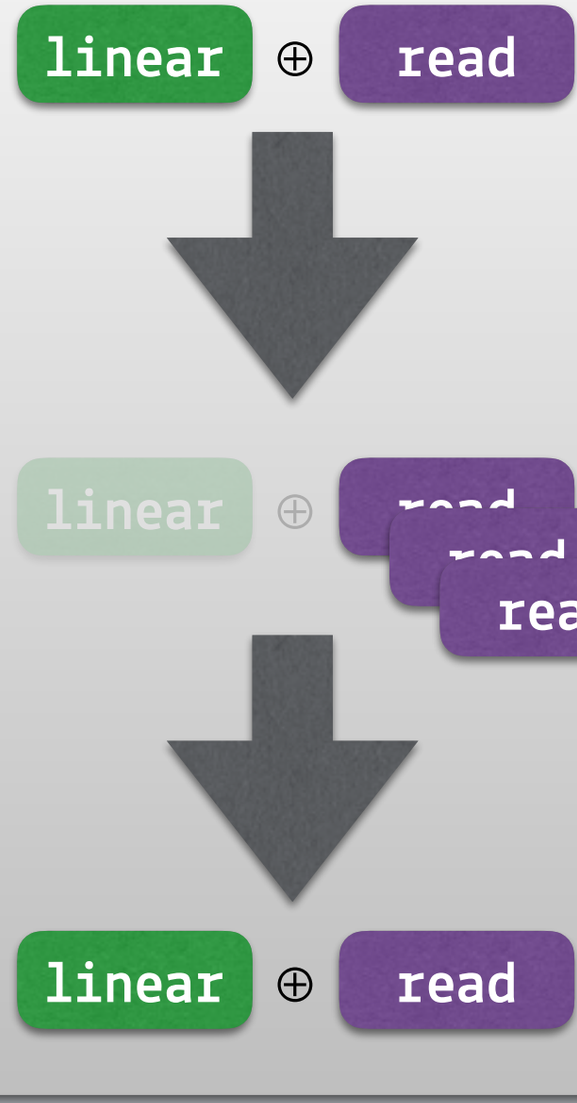


Aliasing and Concurrency Control (revisited)

```
class LocalCounter = thread Inc ⊕
```



Also in the paper!



Ma

T_1

```
assert c1.value() ==
c1.inc();

assert c1.value() ==
```

sy

```
class SharedCounter = locked Inc ⊕
```

Implemented by a read

Composite Capabilities

- A capability *disjunction* A ⊕ B can be used as A *or* B, but not in parallel
- Capabilities that do not share data should be usable in parallel…

```
trait Fst {
  require var fst : int
  ...
}
```

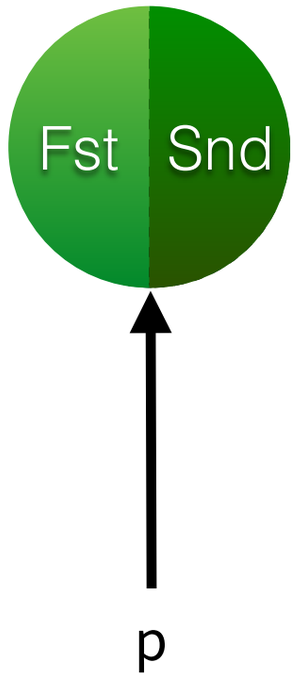
```
trait Snd {
  require var snd : int
  ...
}
```

```
class Pair = linear Fst ⊗ linear Snd {
  var fst : int
  var snd : int
}
```

- A capability *conjunction* A ⊗ B can be used as A *and* B, possibly in parallel

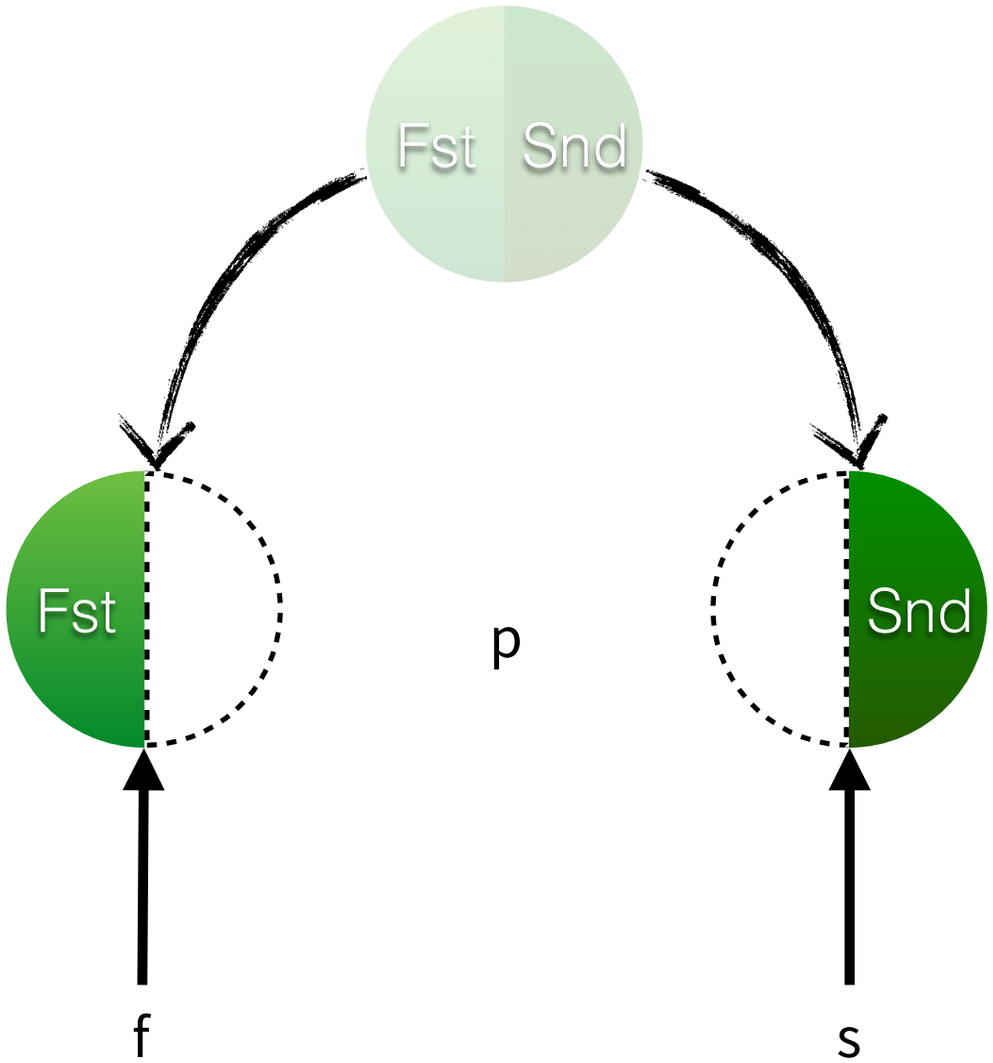
Packing and Unpacking

```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
p = consume f + consume s
```



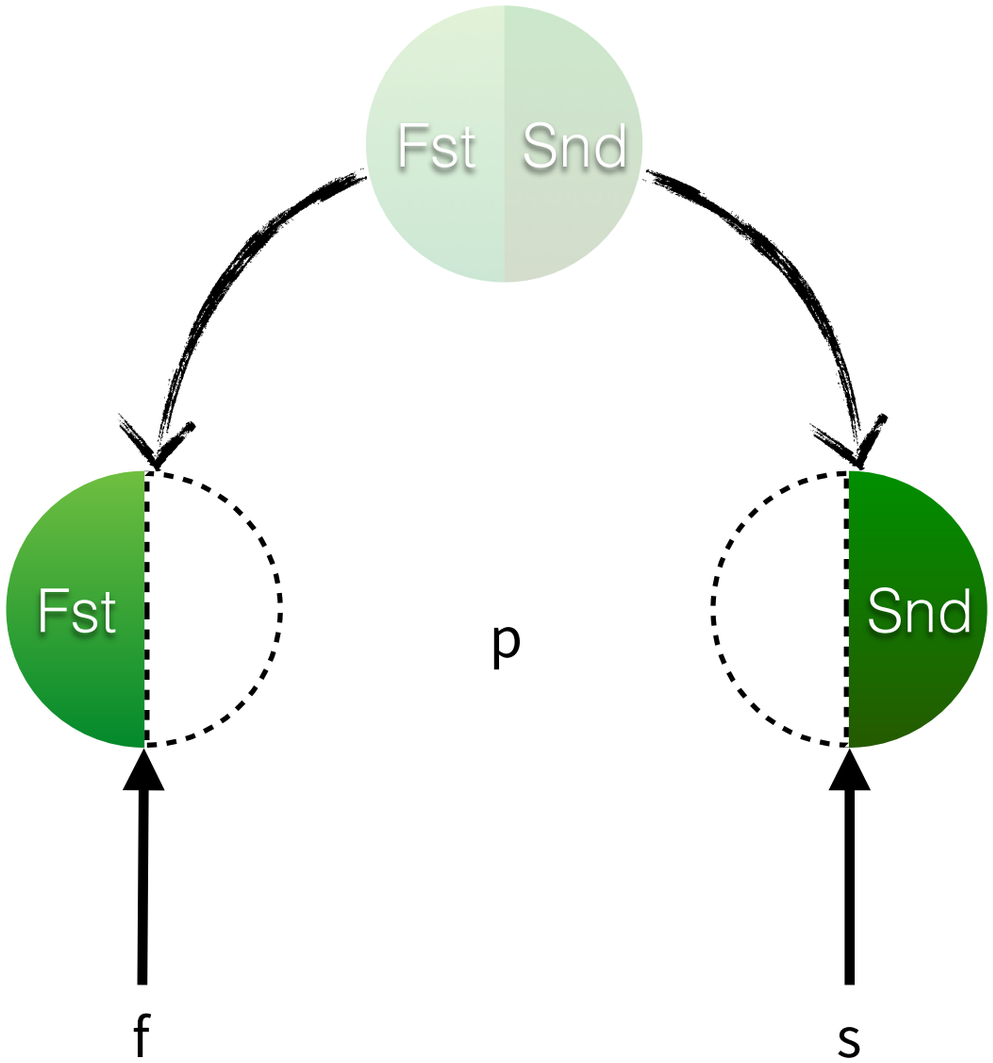
Packing and Unpacking

```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
p = consume f + consume s
```



Packing and Unpacking

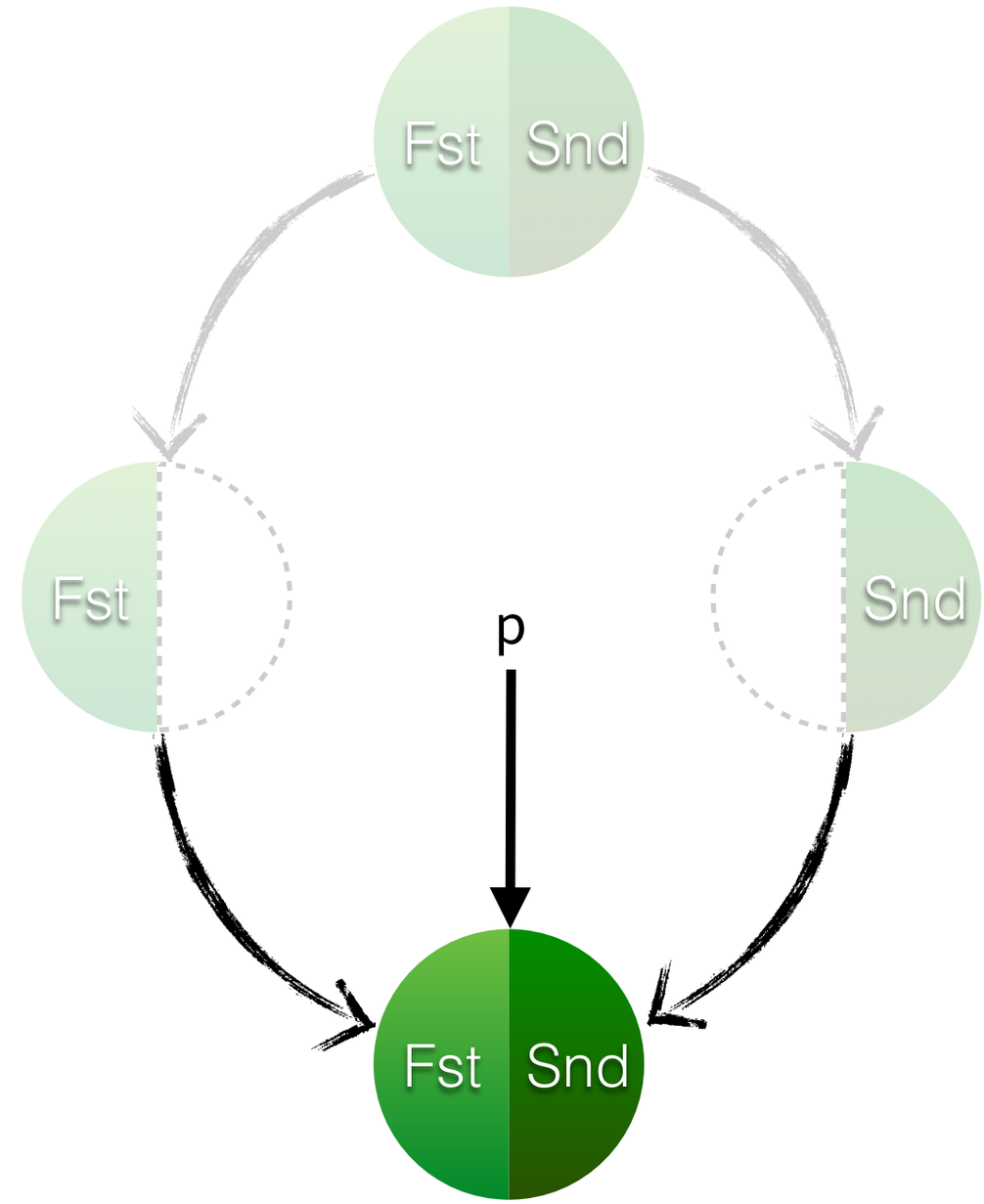
```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
p = consume f + consume s
```



Packing and Unpacking

```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
```

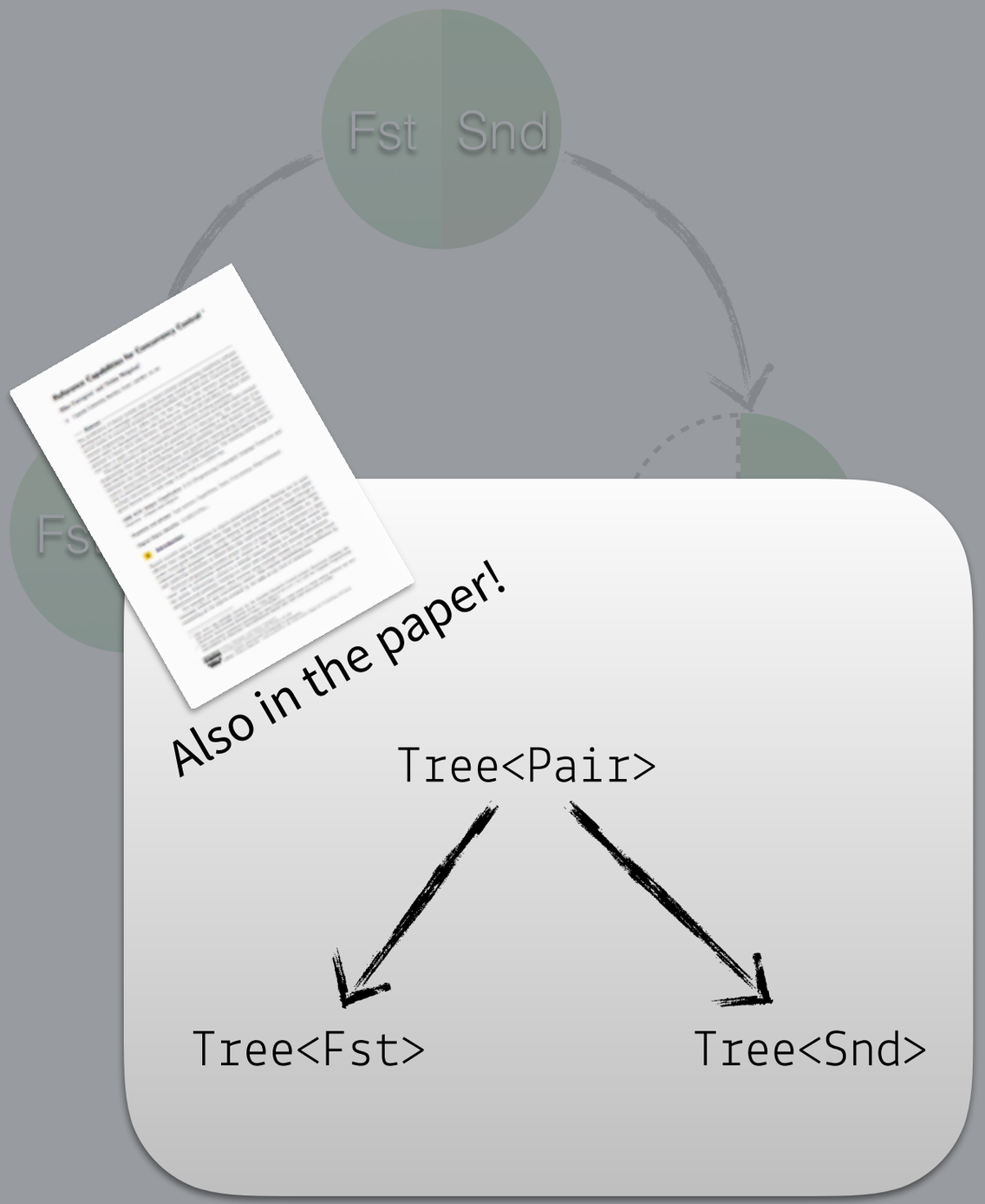
```
p = consume f + consume s
```



Packing and Unpacking

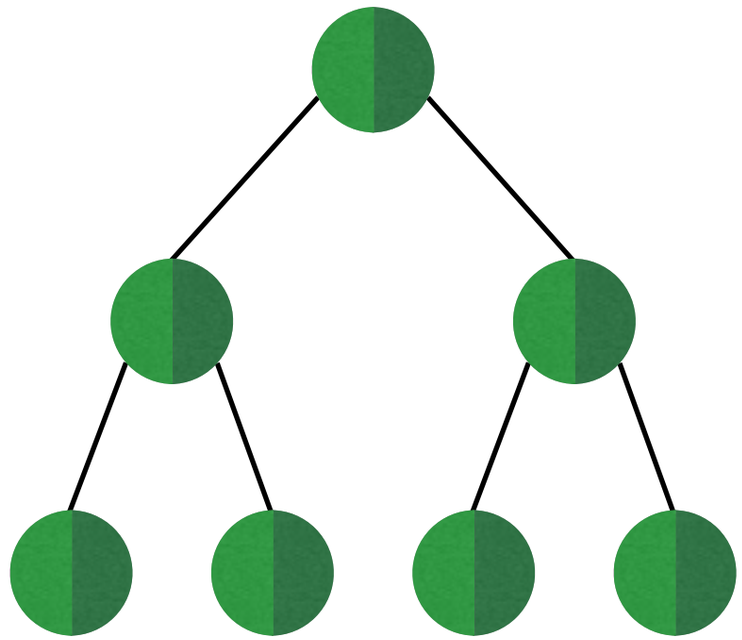
```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
```

```
p = consume f + consume s
```



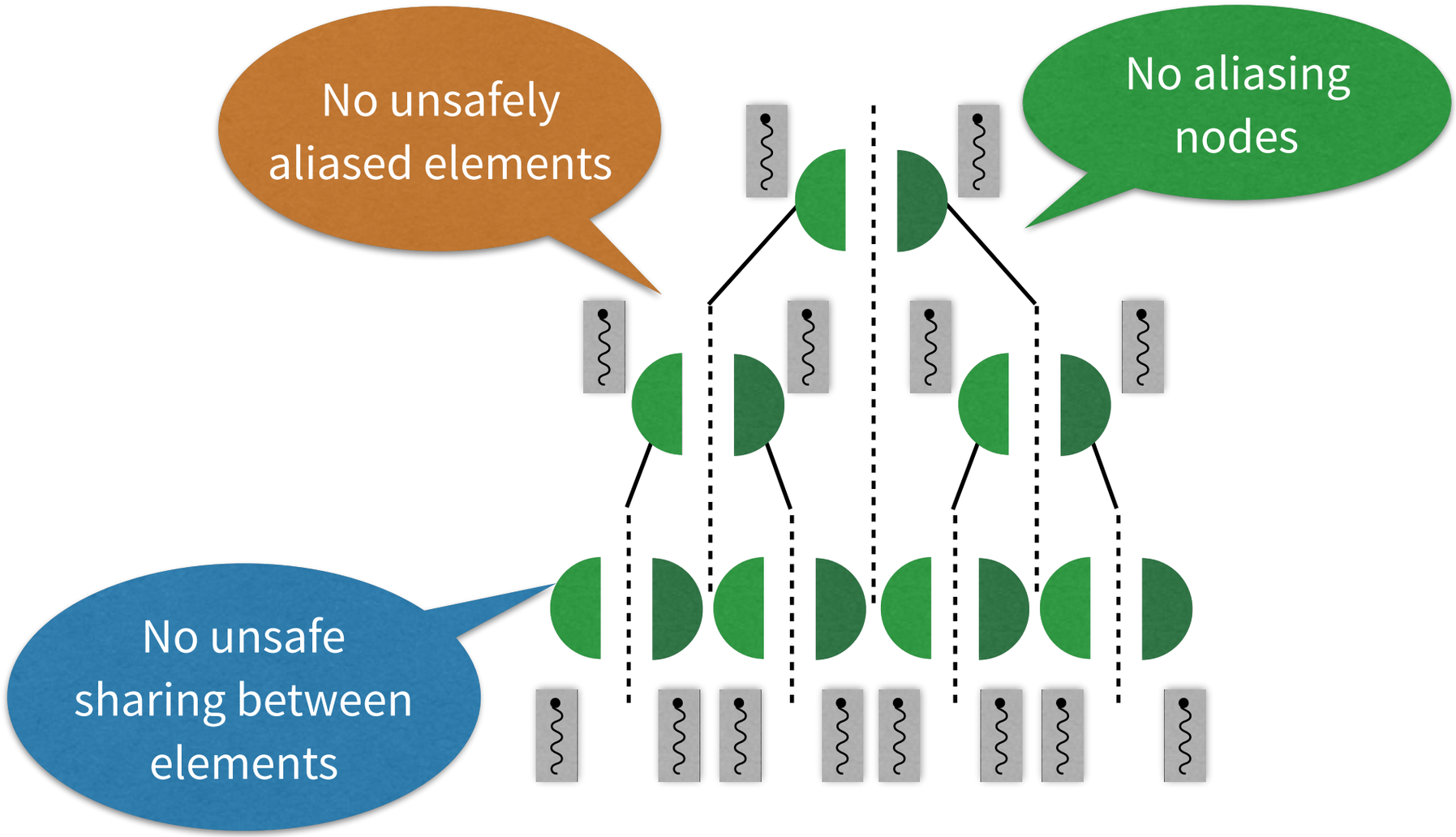
Aliasing and Parallelism (revisited)

class Tree = **linear** Left ⊗ **linear** Right ⊗ ...



Aliasing and Parallelism (revisited)

```
class Tree = linear Left ⊗ linear Right ⊗ ...
```

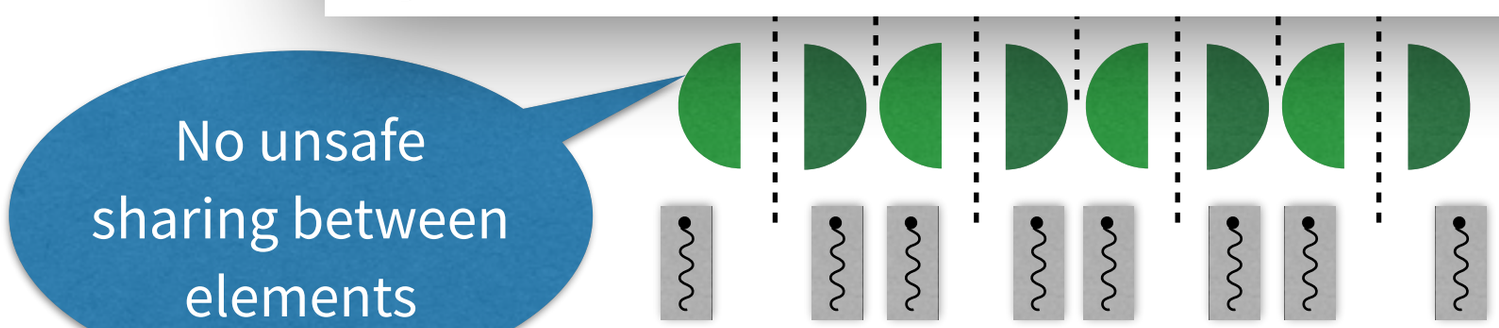


Aliasing and Parallelism (revisited)

class Tree = **linear** Left ⊗ **linear** Right ⊗ ...

(borrowing)

```
def foreach(t : S(Tree<T>), f : T -> T) : void
  let l : S(linear Left<T>)
    , r : S(linear Right<T>)
    , e : S(linear Element<T>) = consume t;
finish {
  async { foreach(l.getLeft(), f) }
  async { foreach(r.getRight(), f) }
  e.apply(f);
}
```



No unsafe sharing between elements

Concurrency and Code Reuse (revisited)

```
trait Add<T> {
  require var first : Link<T>
  def add(elem : T) : void {
    ...
  }
}
```

- Reuse traits across different concurrency scenarios
- Separate business logic from concurrency concerns

Can assume exclusive access

```
class List<T> = thread Add<T> ⊕ ... {
  var first : Link<T>
}
```

```
class SynchronizedList<T> = locked Add<T> ⊕ ... {
  var first : Link<T>
}
```

Annotations in type declarations only

No effect tracking or ownership types

Meta-Theoretic Results

- Type system formalised on top of a simple object-oriented language

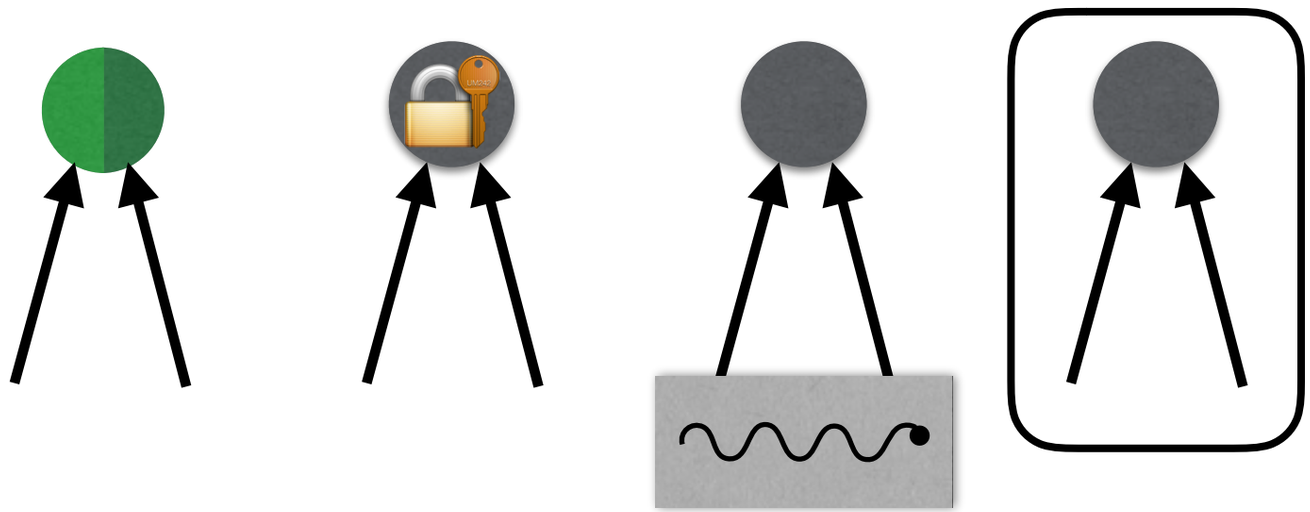
  Finish-Async style parallelism

  Reentrant readers-writer locks

- Key result: **Well-formed programs preserve safe aliasing**

  Any two aliases are e.g. composable, synchronised, thread-local, subordinate

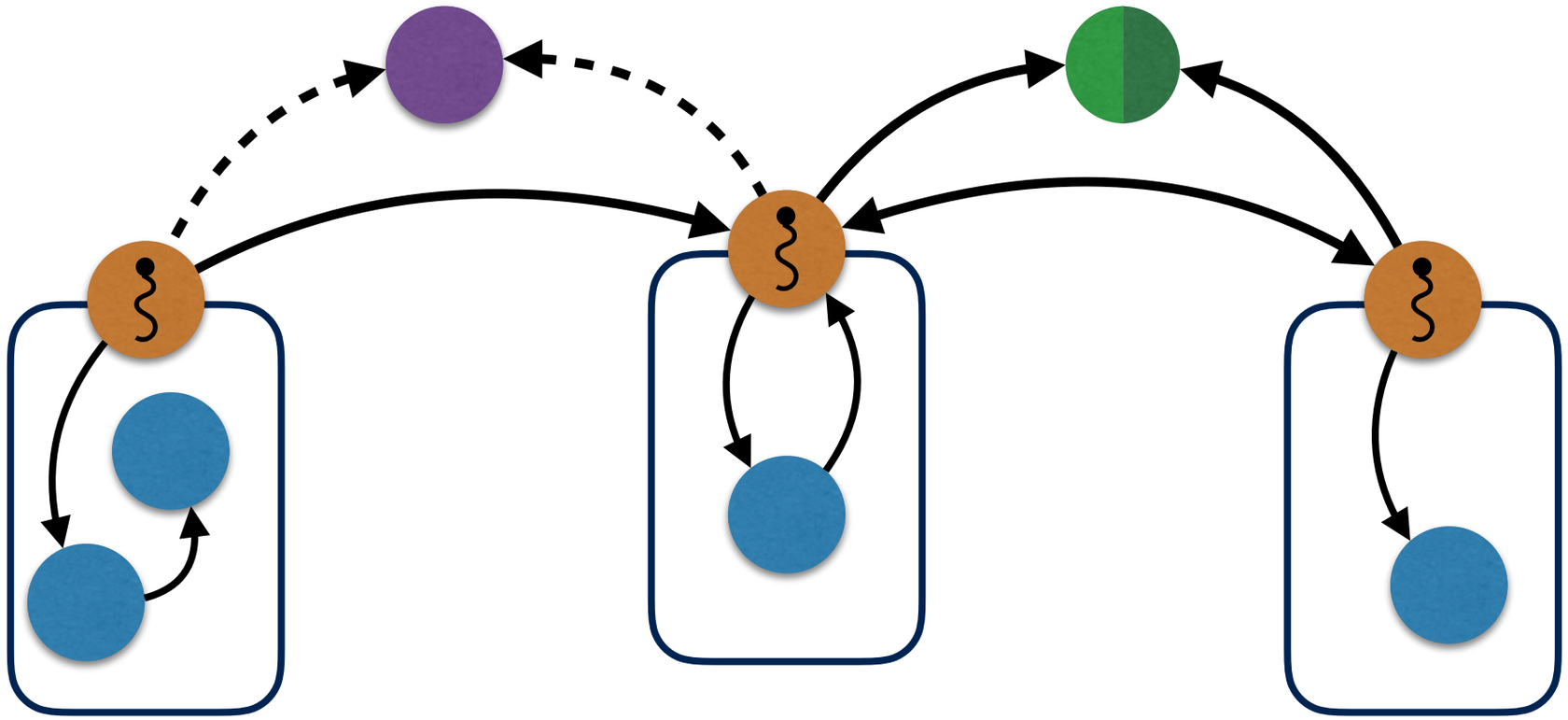
  $\Longrightarrow$ No data-races



Current and Future Work

- Implementation underway in the context of Encore

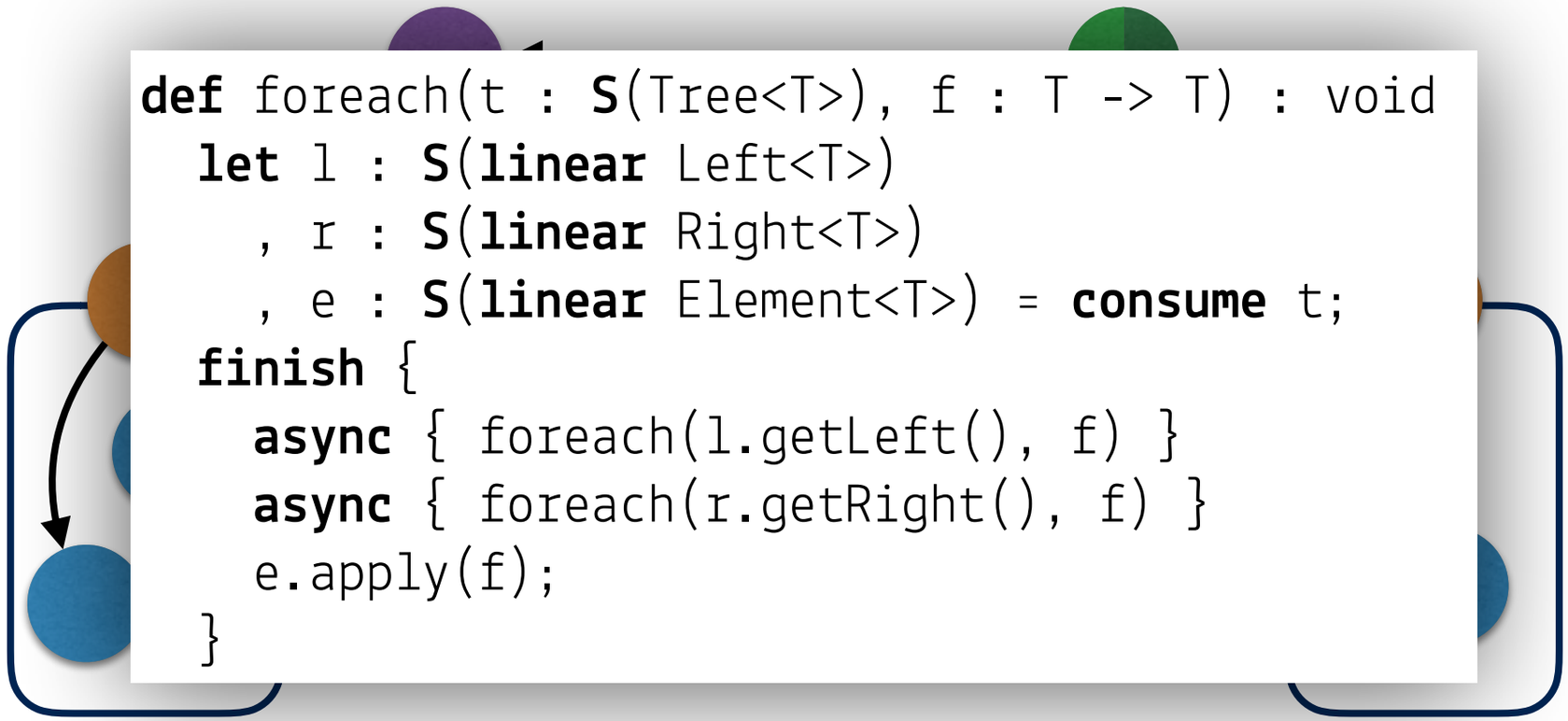
  Reference capabilities for safe sharing between active objects



Current and Future Work

- Implementation underway in the context of Encore

  Reference capabilities for safe sharing between active objects

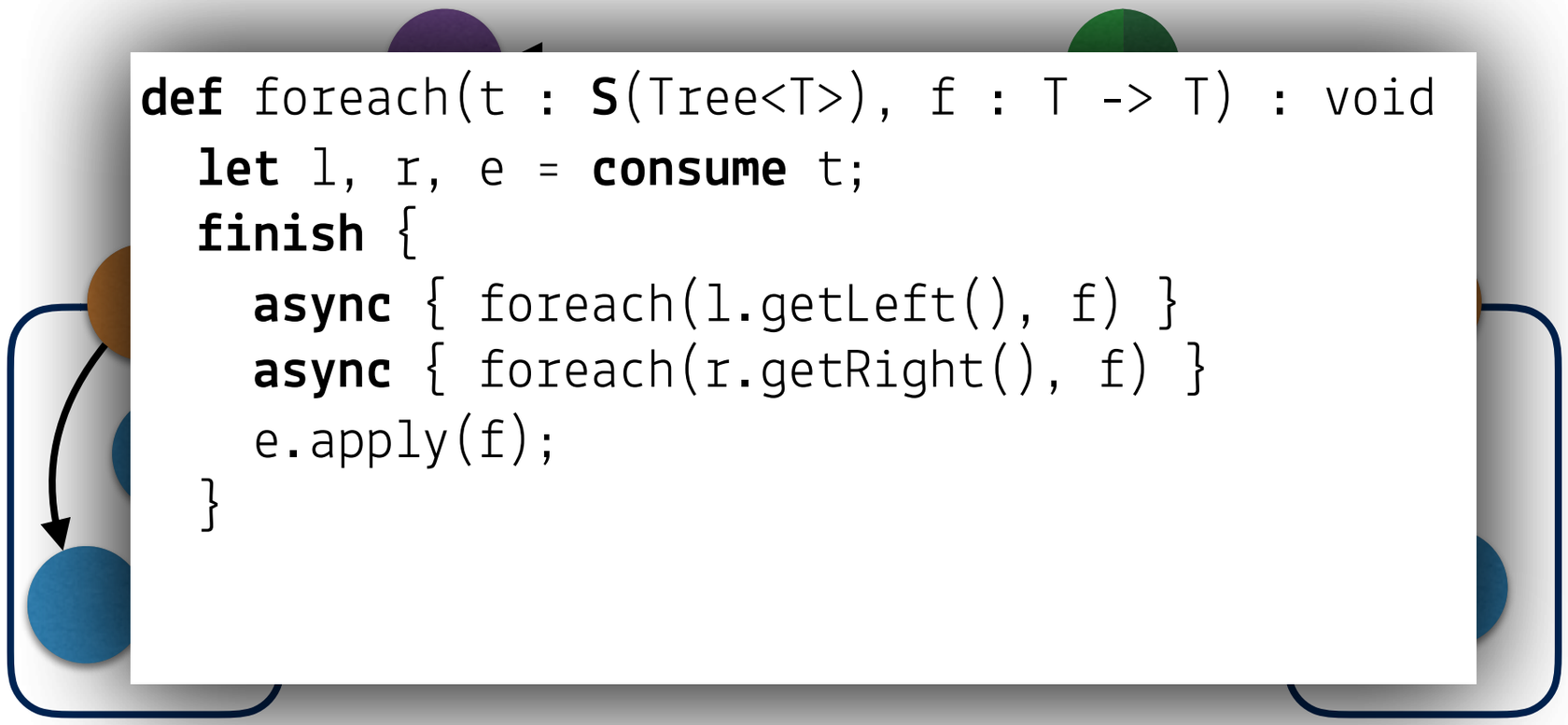


```
def foreach(t : S(Tree<T>), f : T -> T) : void
  let l : S(linear Left<T>)
    , r : S(linear Right<T>)
    , e : S(linear Element<T>) = consume t;
  finish {
    async { foreach(l.getLeft(), f) }
    async { foreach(r.getRight(), f) }
    e.apply(f);
  }
```

Current and Future Work

- Implementation underway in the context of Encore

  Reference capabilities for safe sharing between active objects

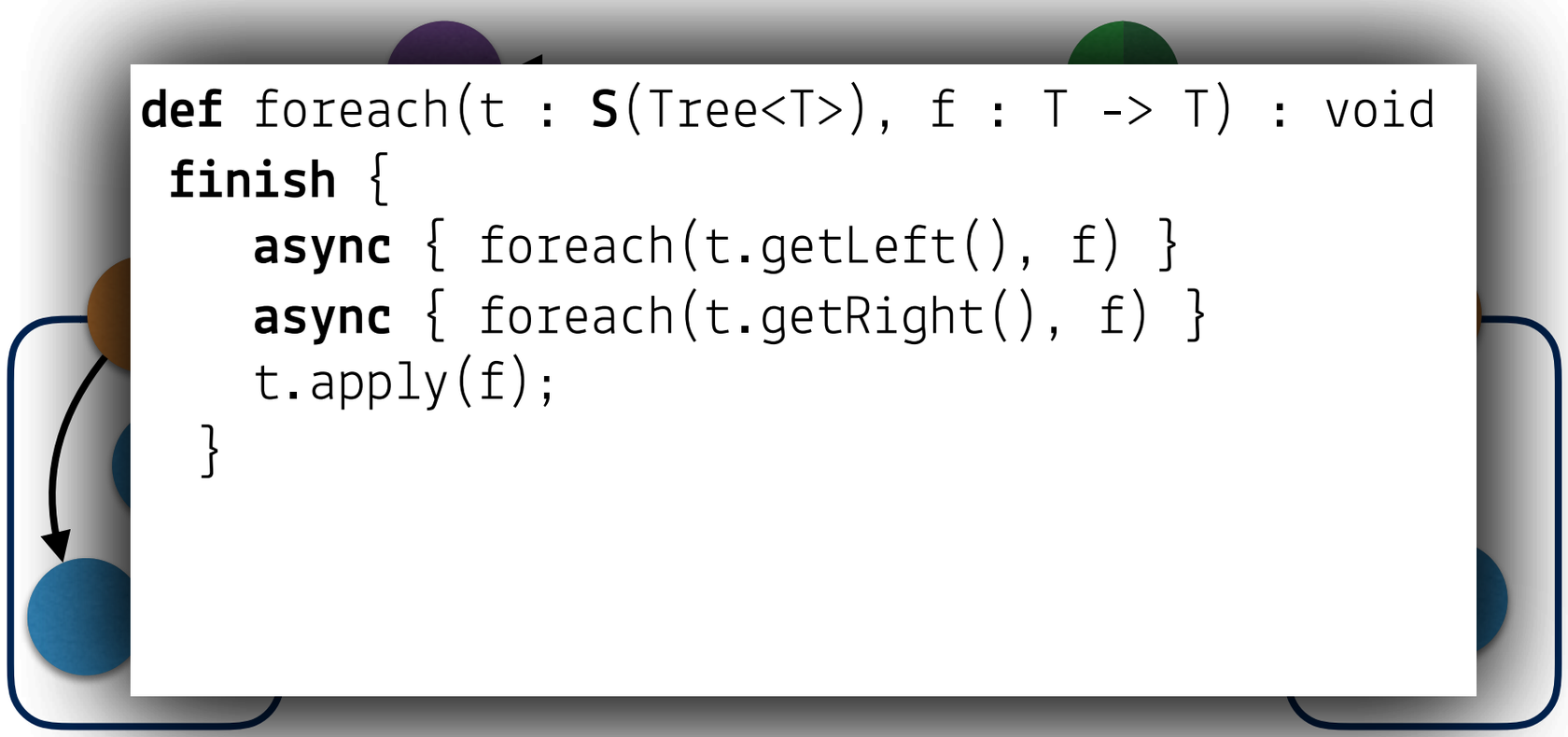


```
def foreach(t : S(Tree<T>), f : T -> T) : void
  let l, r, e = consume t;
  finish {
    async { foreach(l.getLeft(), f) }
    async { foreach(r.getRight(), f) }
    e.apply(f);
  }
```

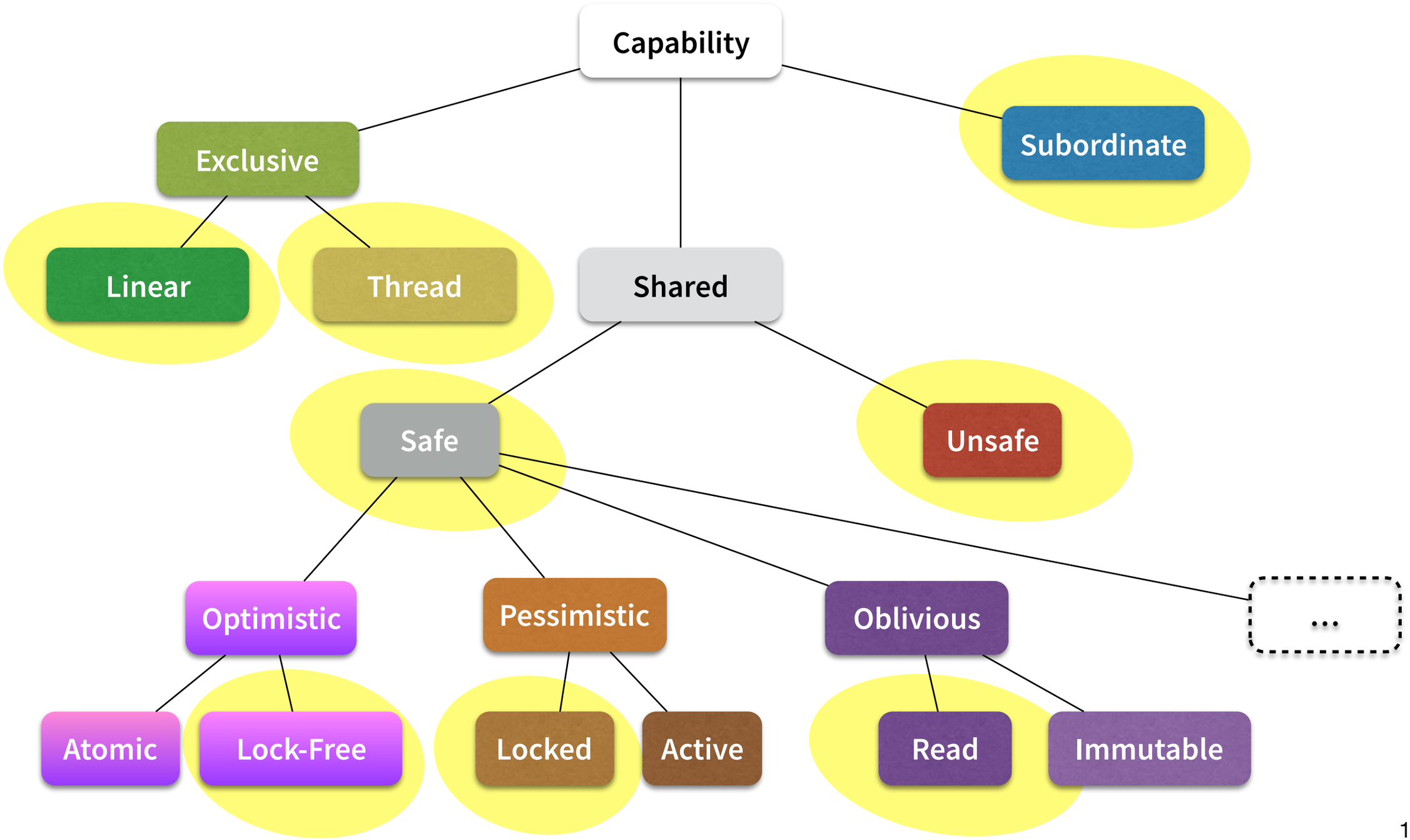
Current and Future Work

- Implementation underway in the context of Encore

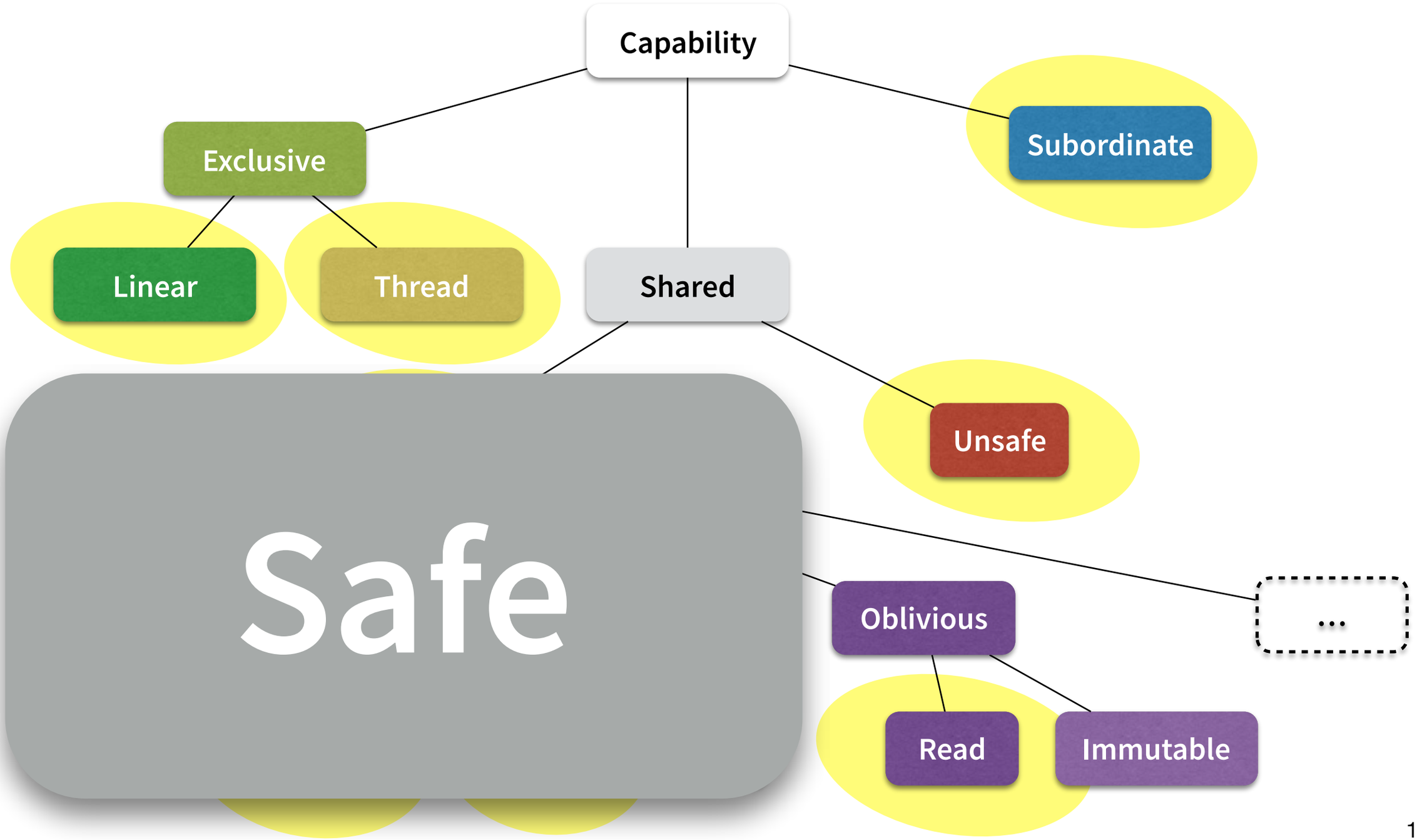
  Reference capabilities for safe sharing between active objects



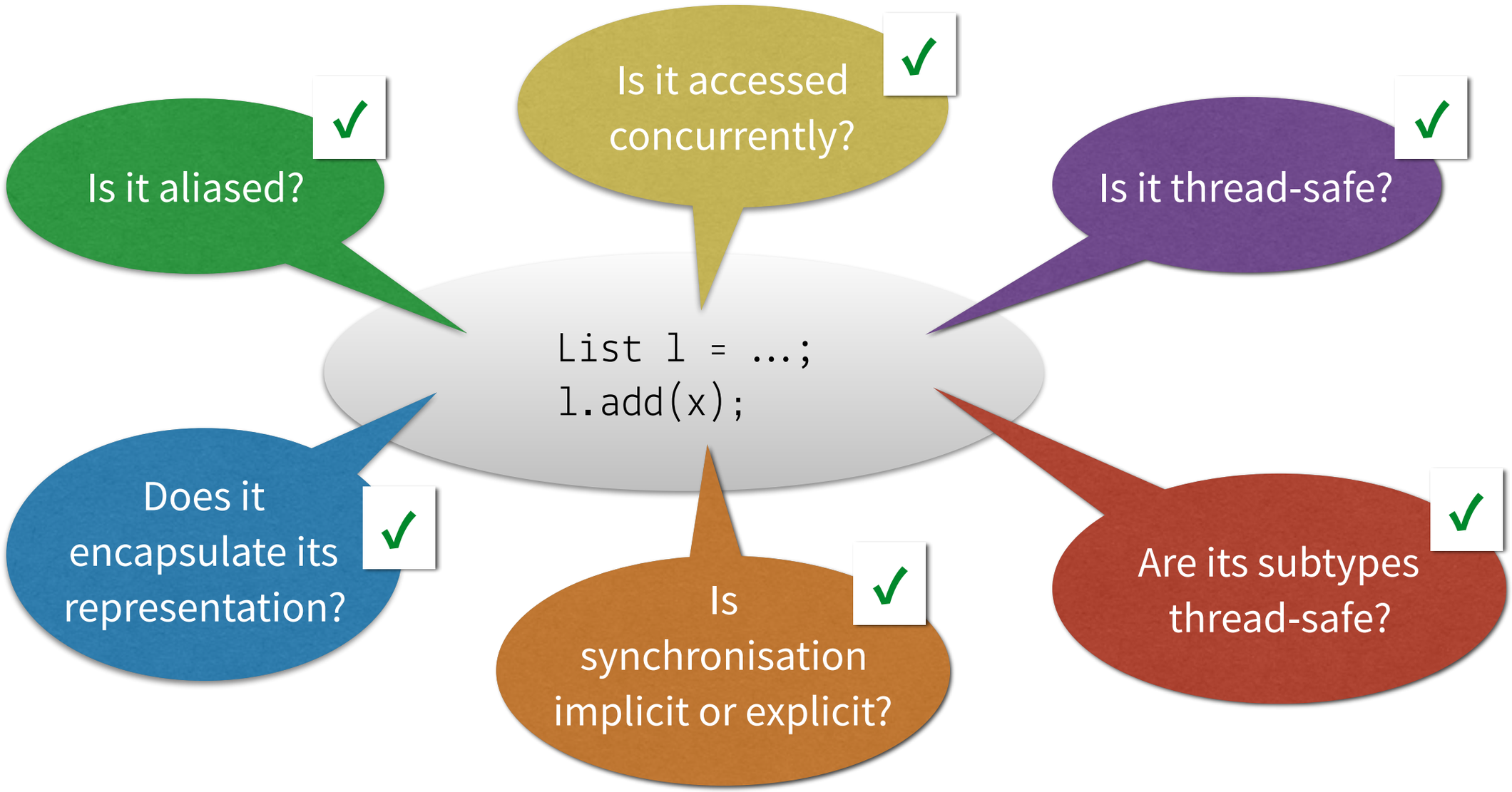
A Hierarchy of Capabilities



A Hierarchy of Capabilities



Reference Capabilities Address Many Concerns



More examples, full proofs etc. in the technical report!