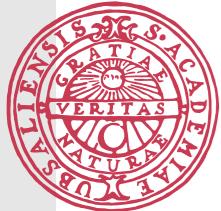


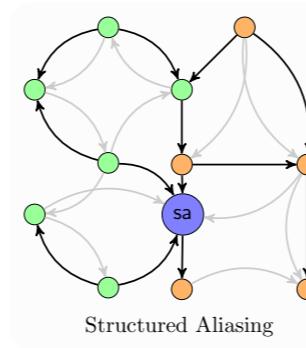
Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16



UPMARC



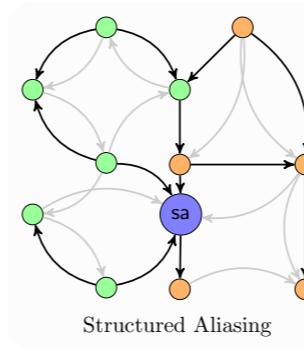
Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

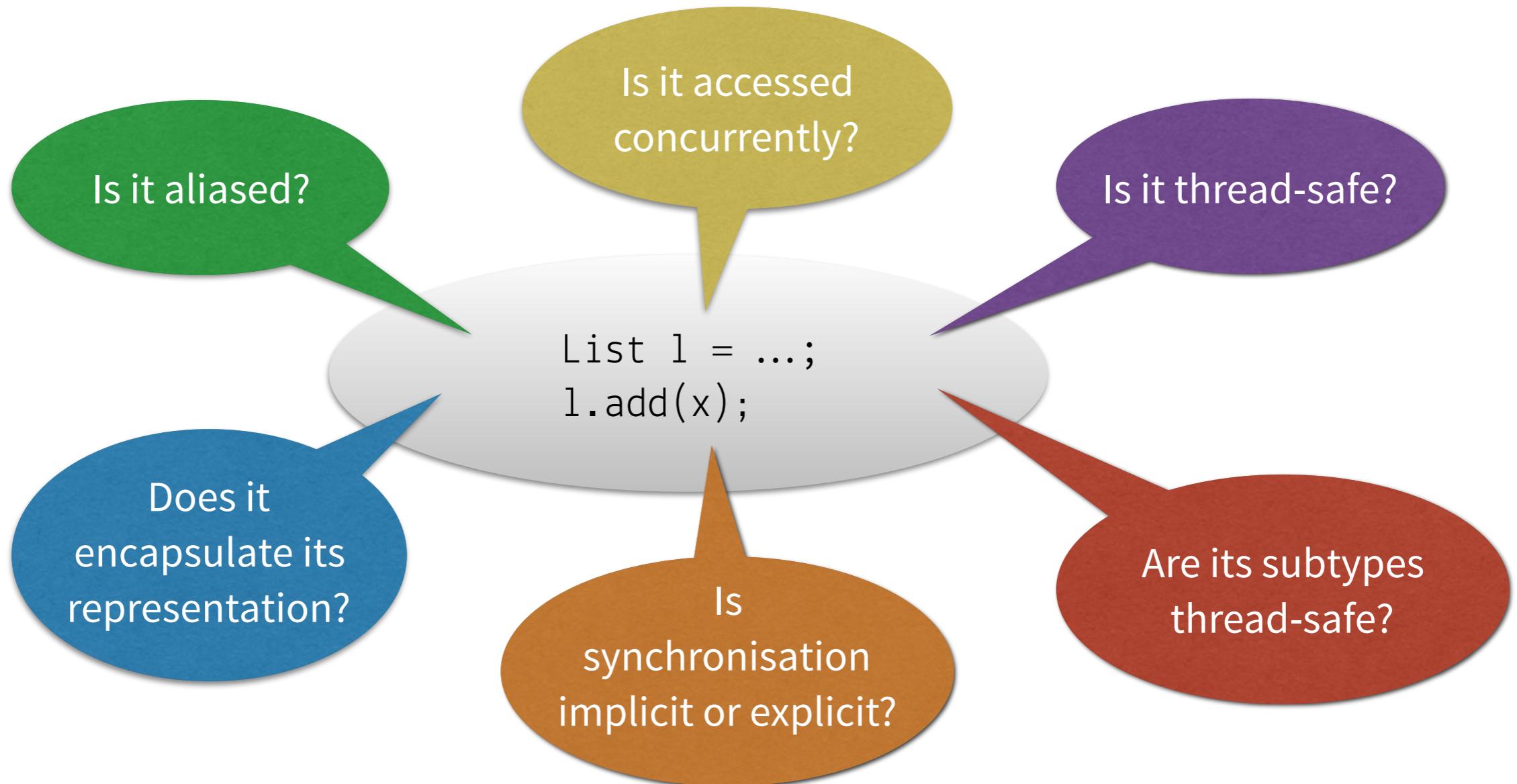
IWACO'16



UPMARC



Concurrency Imposes Many Concerns

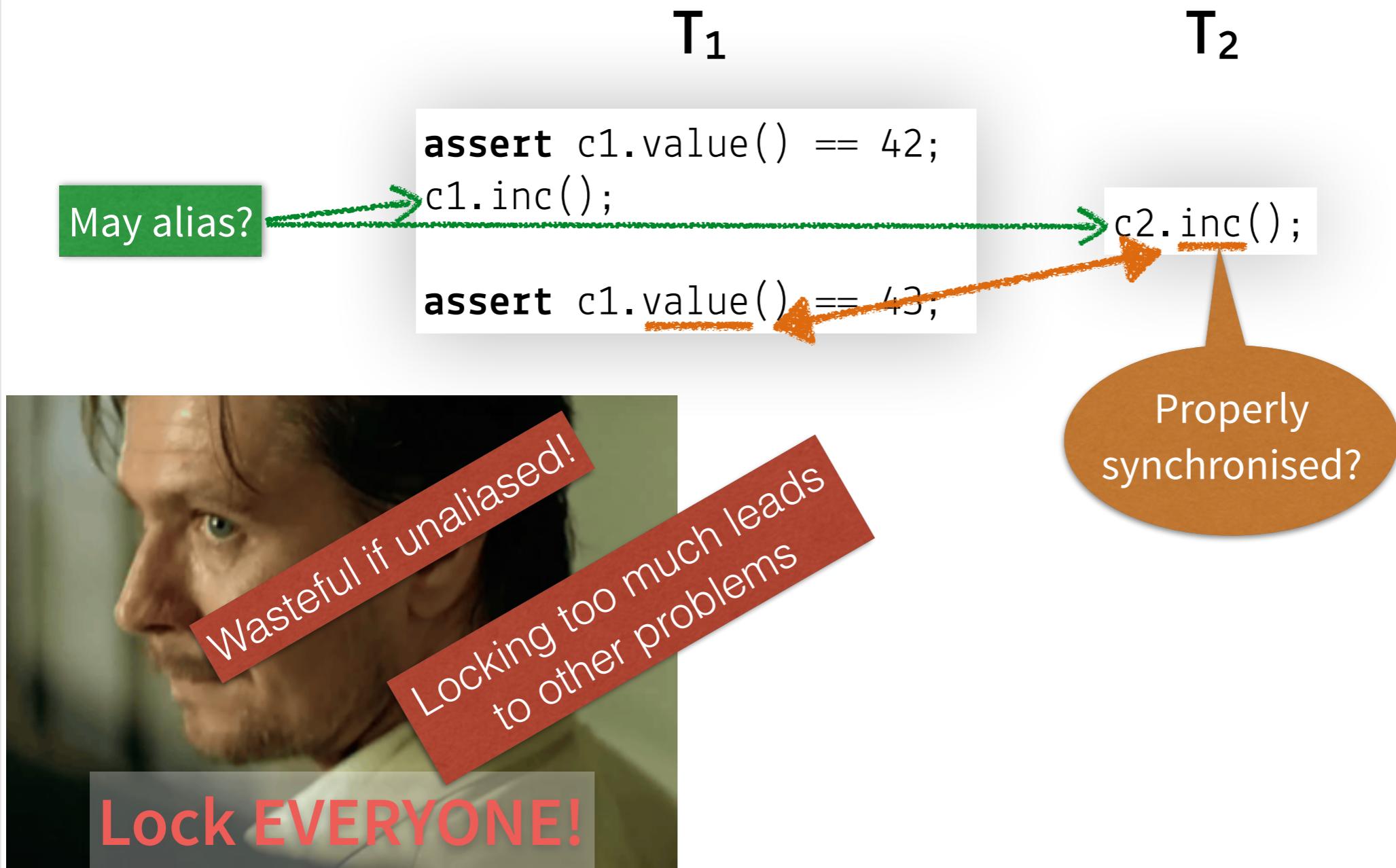


Aliasing and Concurrency Control

May alias?

```
assert c1.value() == 42;  
c1.inc();  
c2.inc();  
assert c1.value() == 43;
```

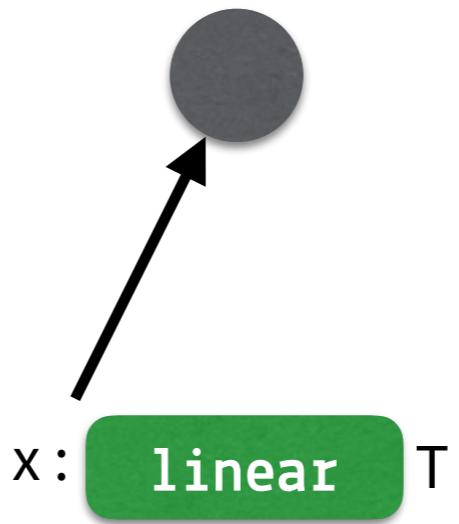
Aliasing and Concurrency Control



Reference Capabilities

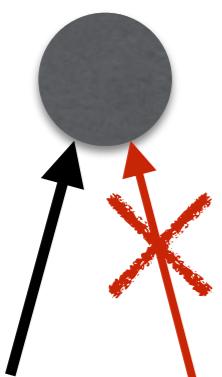
- A capability grants access to some ~~resource~~ object

- The type of a capability defines the interface to its object
- A capability assumes exclusive access
Thread-safety \Rightarrow No data-races
- How thread-safety is achieved is controlled by the capability's mode



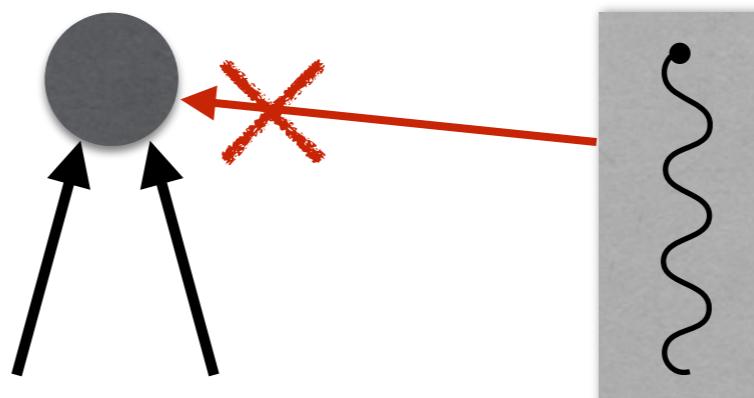
Modes of Concurrency Control

- *Exclusive modes*



linear

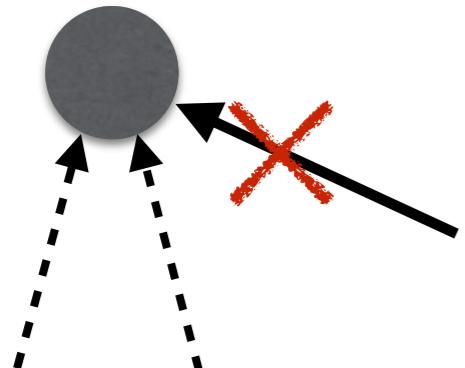
Globally unique



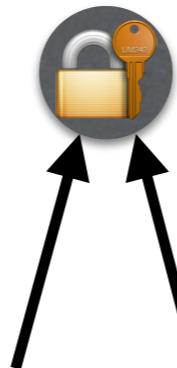
thread

Thread-local

- *Safe modes*



read



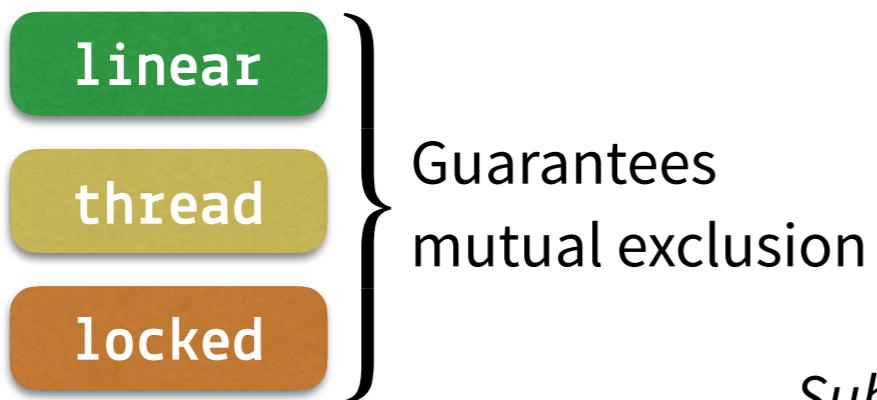
locked

Precludes mutating
aliases

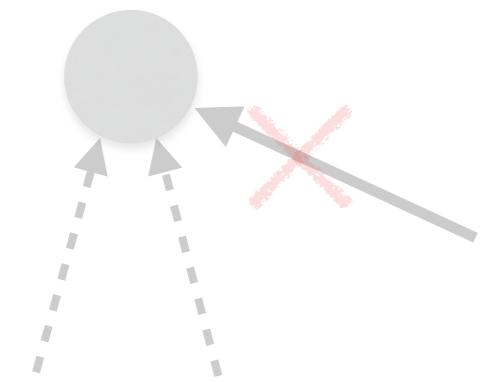
Implicit locking

Modes of Concurrency Control

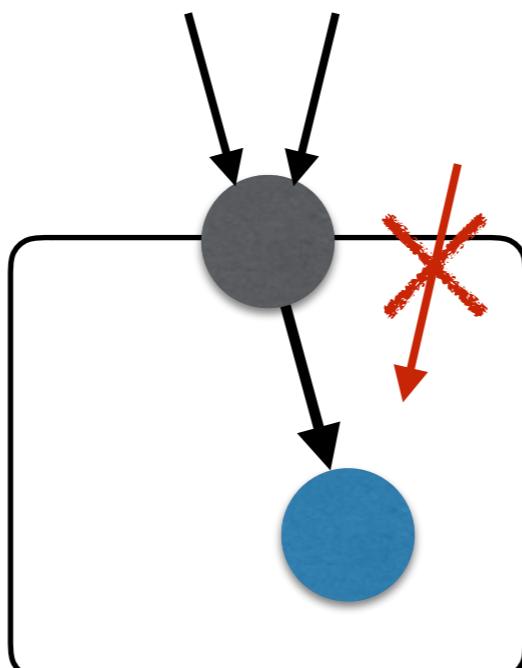
Dominating modes



Subordinate mode



Precludes mutating
aliases



Encapsulated

Capability = Trait + Mode

- Capabilities are introduced via traits

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

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

- Modes control *why* they are safe

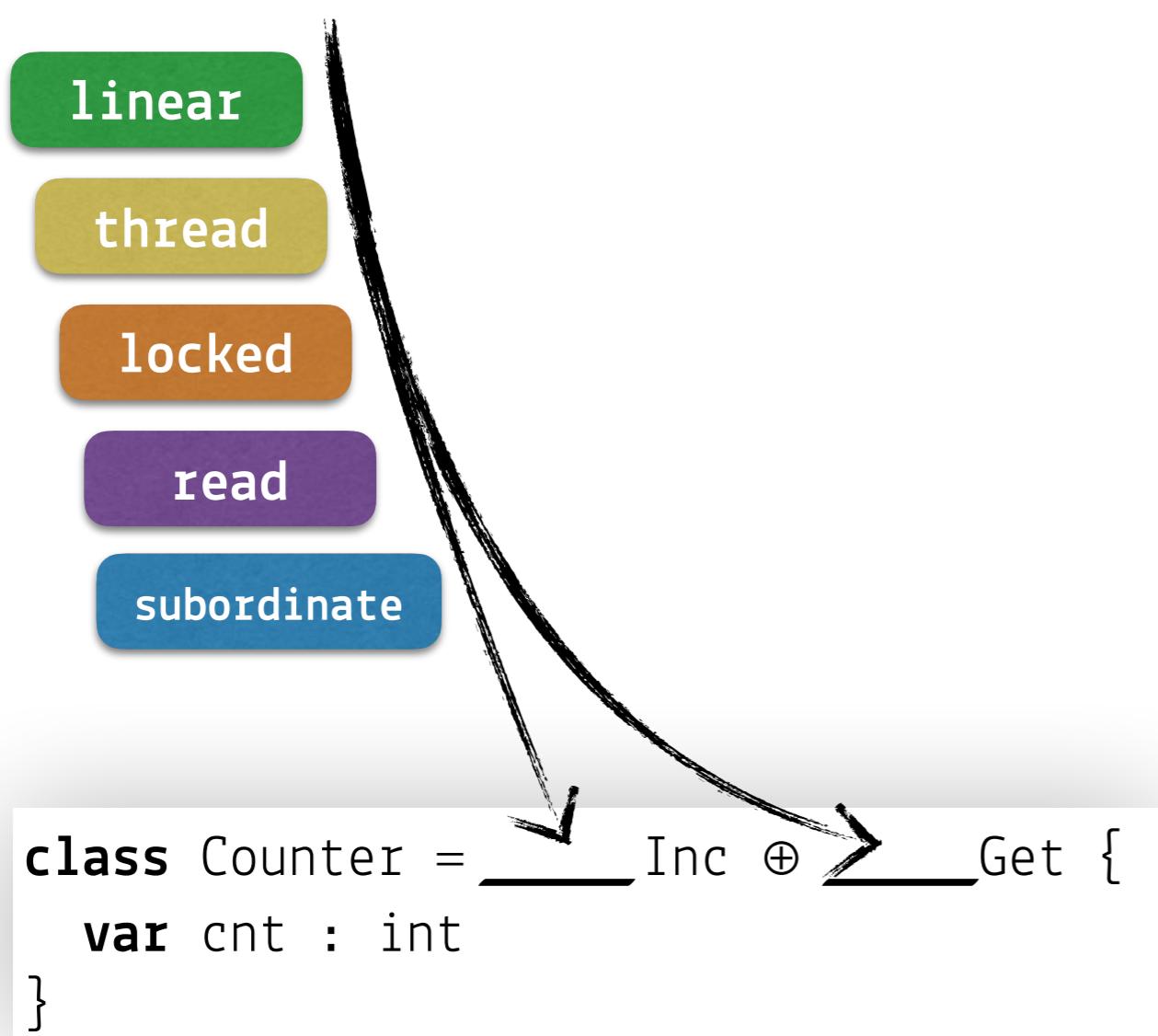
linear Inc — Globally unique increment capability

locked Inc — Implicitly synchronised increment capability

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

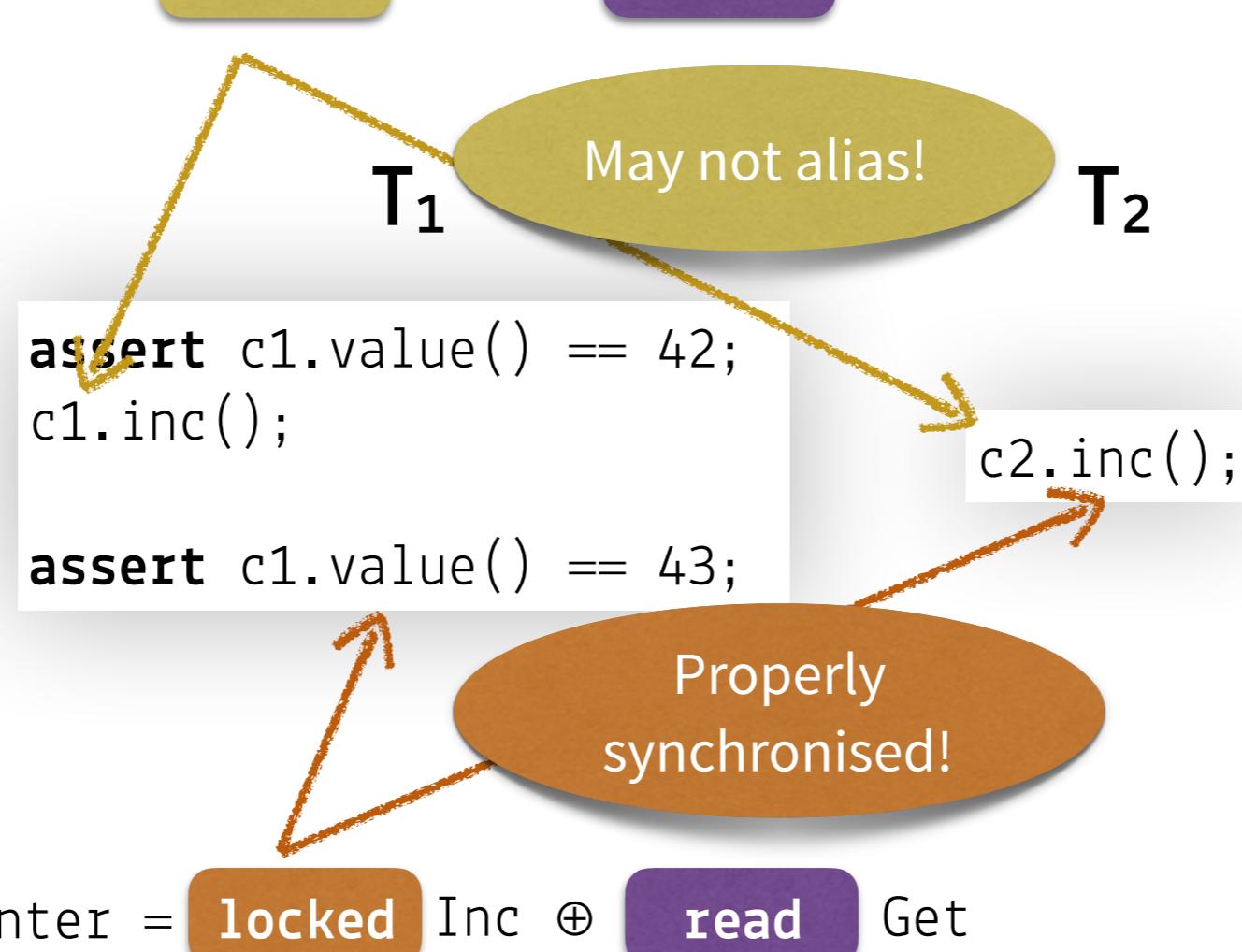
read Get — A read-only capability for getting the value

Classes are Composed by Capabilities



Aliasing and Concurrency Control (revisited)

class LocalCounter = **thread** Inc ⊕ **read** Get



class SharedCounter = **locked** Inc ⊕ **read** Get

Implemented by a readers-writer lock

Composite Capabilities

- A capability *disjunction* $A \oplus B$ can be used as A or B , but not at the same time
- Capabilities that do not share data should be usable in parallel...

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

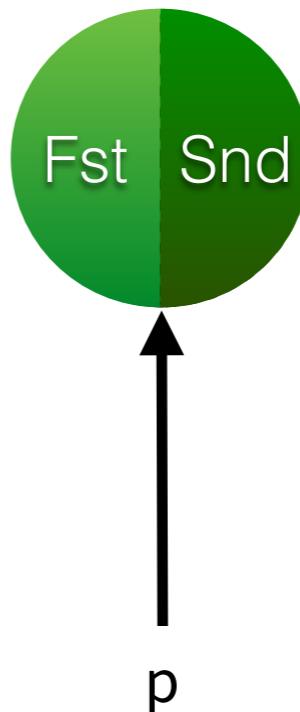
```
class Pair = linear Fst ⊕ linear Snd {  
    var fst : int  
    var snd : int  
}
```



- A capability *conjunction* $A \otimes B$ can be used as A and B , possibly in parallel

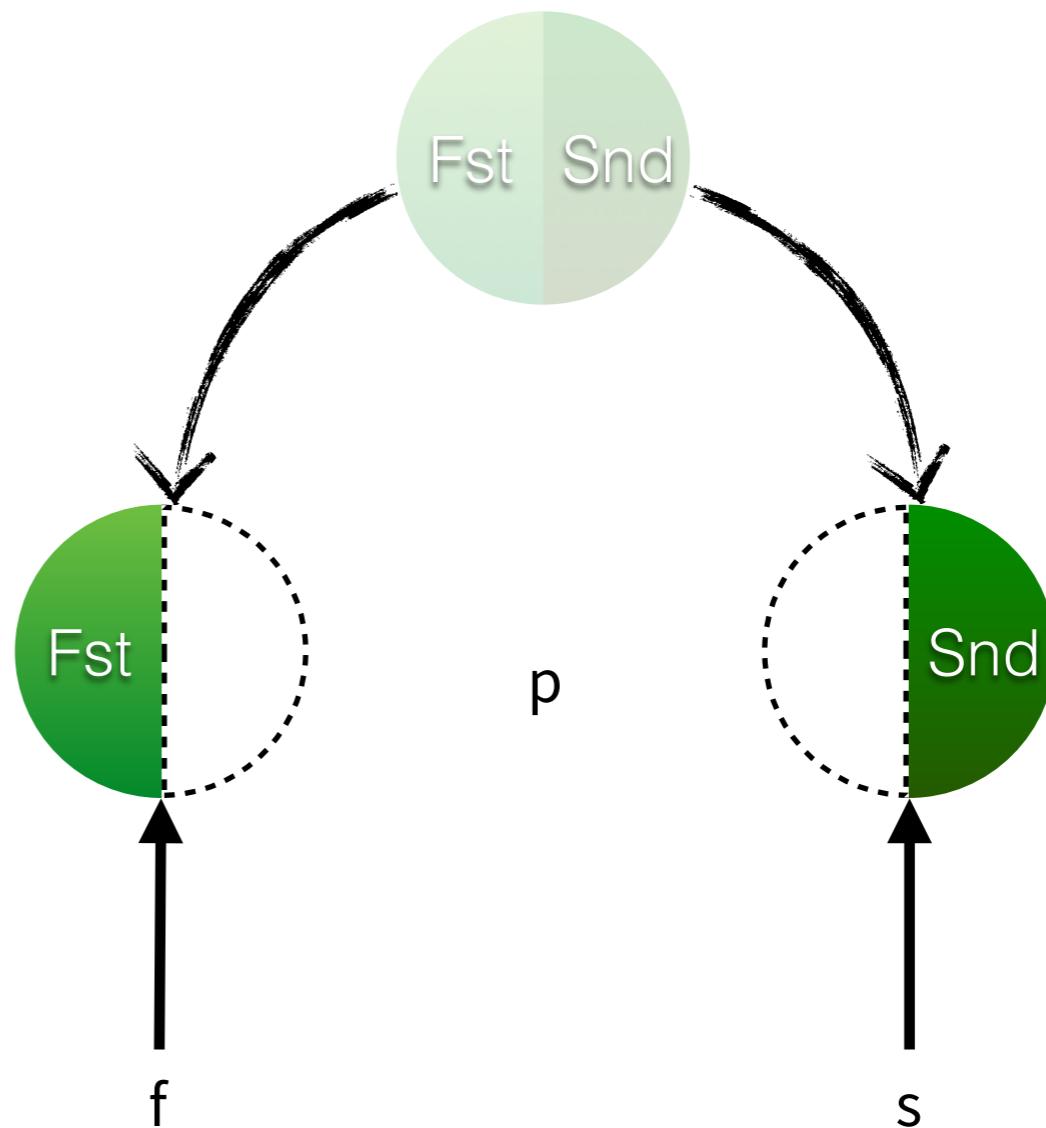
Packing and Unpacking

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



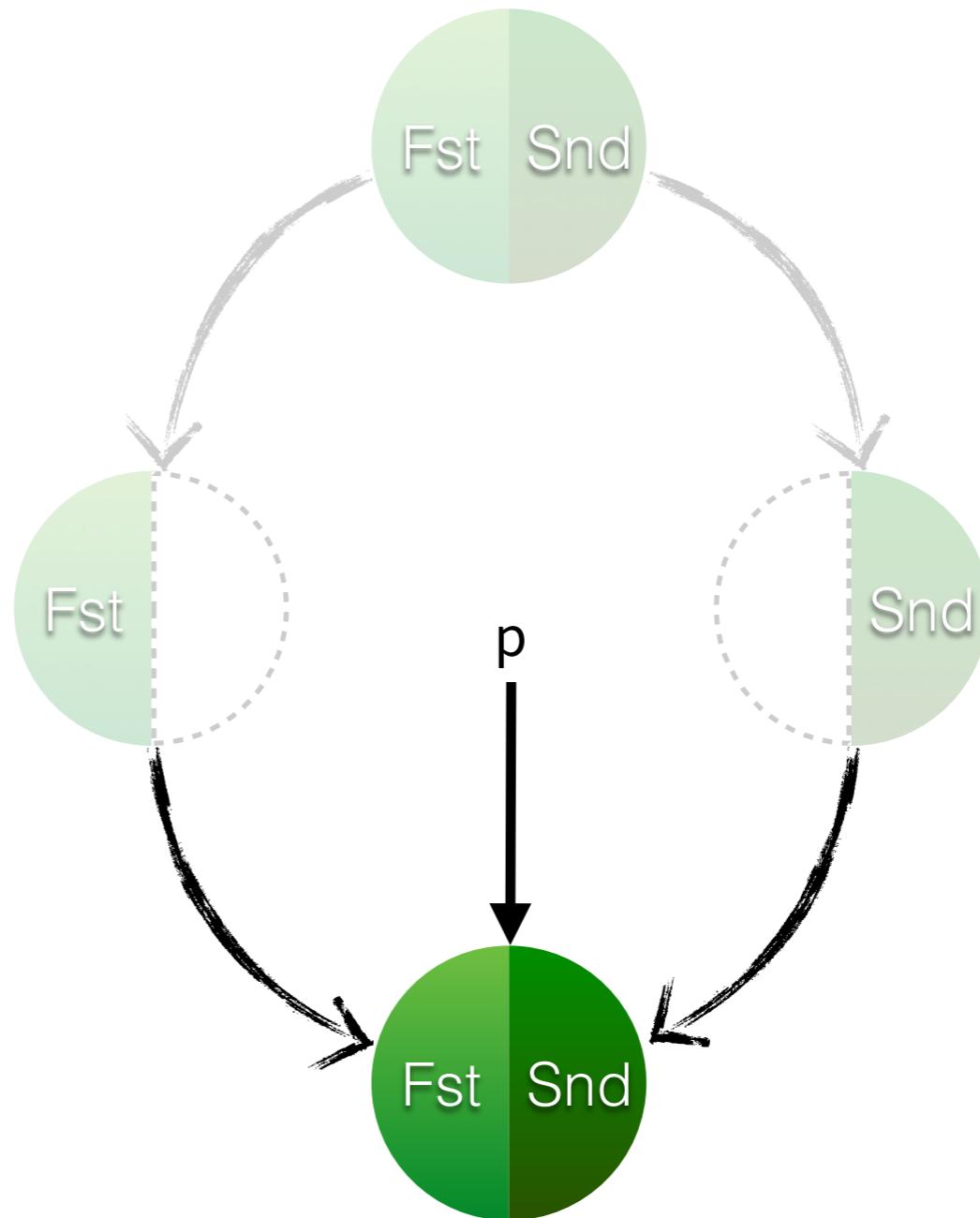
Packing and Unpacking

```
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

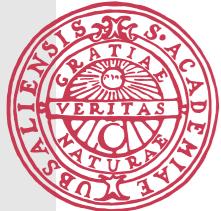
```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
p = consume f + consume s
```



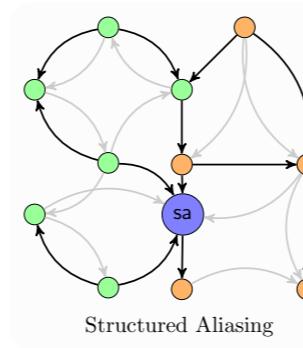
Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16



UPMARC



Subordination and Trait-Based Reuse

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

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

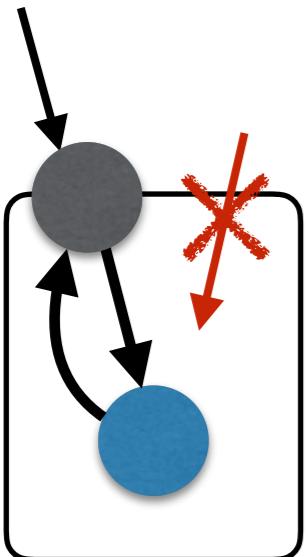
Can assume
exclusive access

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

Annotations in type declarations only
No effect tracking
or ownership types

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

Reference Capabilities as Primitives



Ownership

External uniqueness

Single writer,
multiple readers

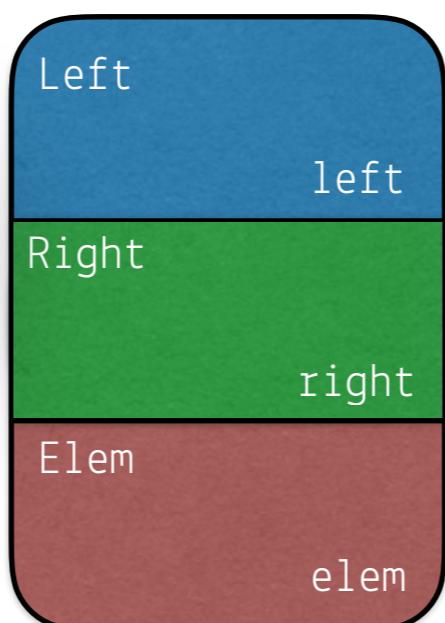
linear Inc \oplus read Get

linear Inc \oplus read Get

linear Inc \oplus read Get

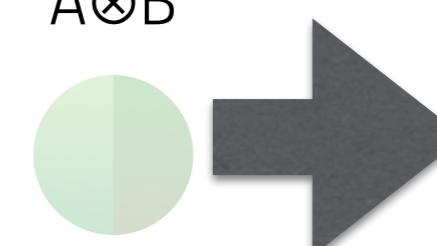
Regions and effects

```
class Tree = Left ⊗ Right ⊗ Elem  
var left : Tree  
var right : Tree  
var elem : int
```



...
multiple
disjoint
writers

$A \otimes B$



A

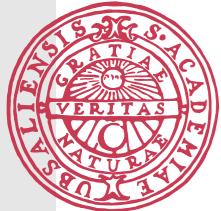


B

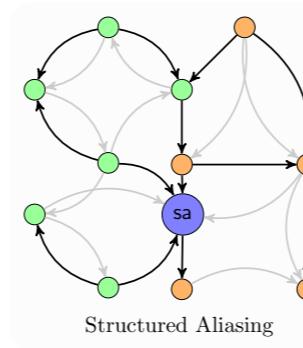
Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16



UPMARC



Active Objects as a Mode of Synchronisation

- The message queue of an active object can replace the synchronisation of locks

```
class ActiveCounter
  var cnt : int

  def inc() : void
    this.cnt++

  def get() : int
    return this.cnt
```

Active by default

```
class ActiveCounter = active Inc ⊕ active Get
  var cnt : int
```

Active Objects as a Mode of Synchronisation

- Opens up for new combinations

active

⊕

linear

Actor with unsynchronised initialisation methods

active

⊕

locked

Actor with priority channel

active

⊕

subord

Actor nested in another actor

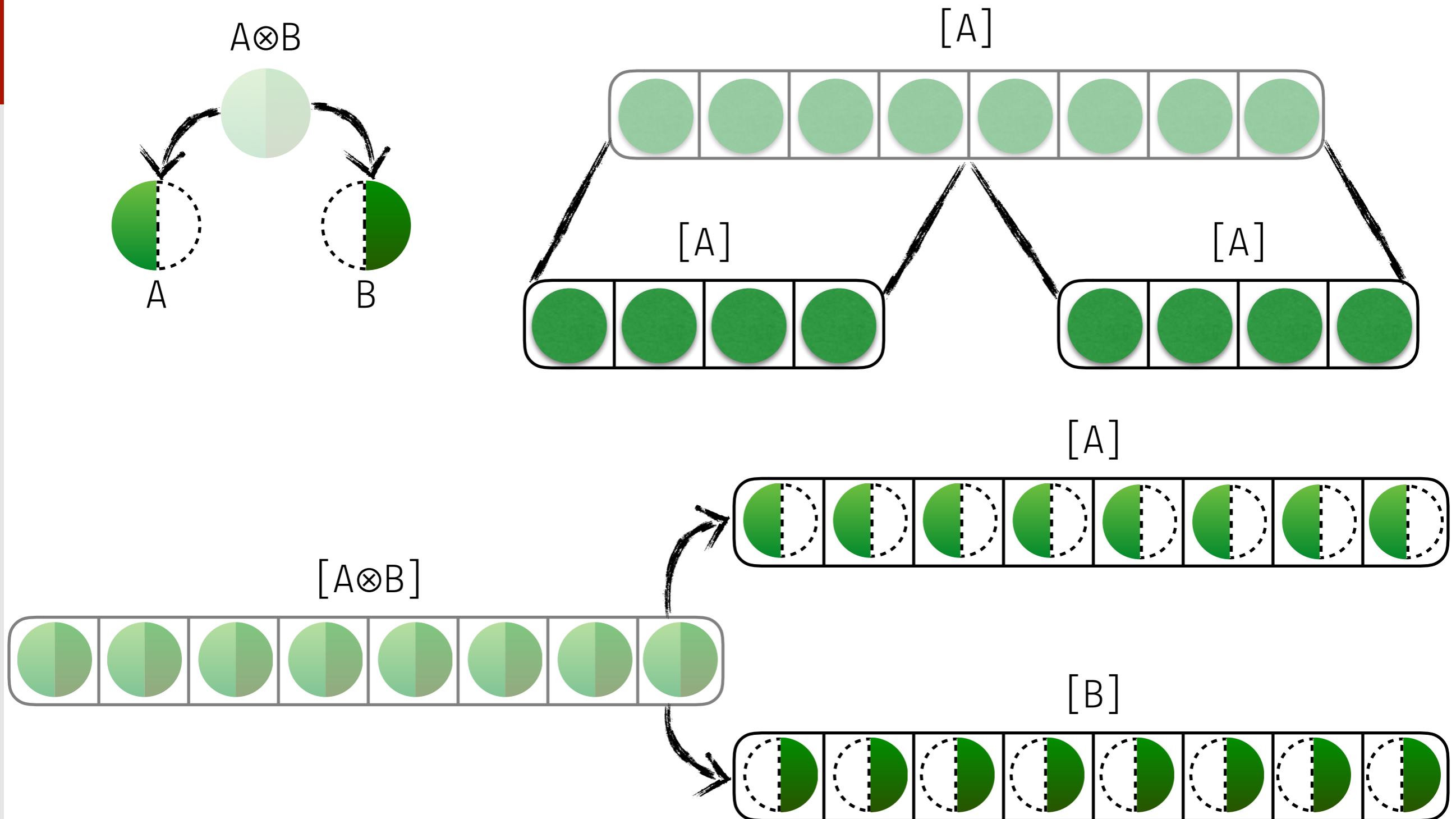
active

⊗

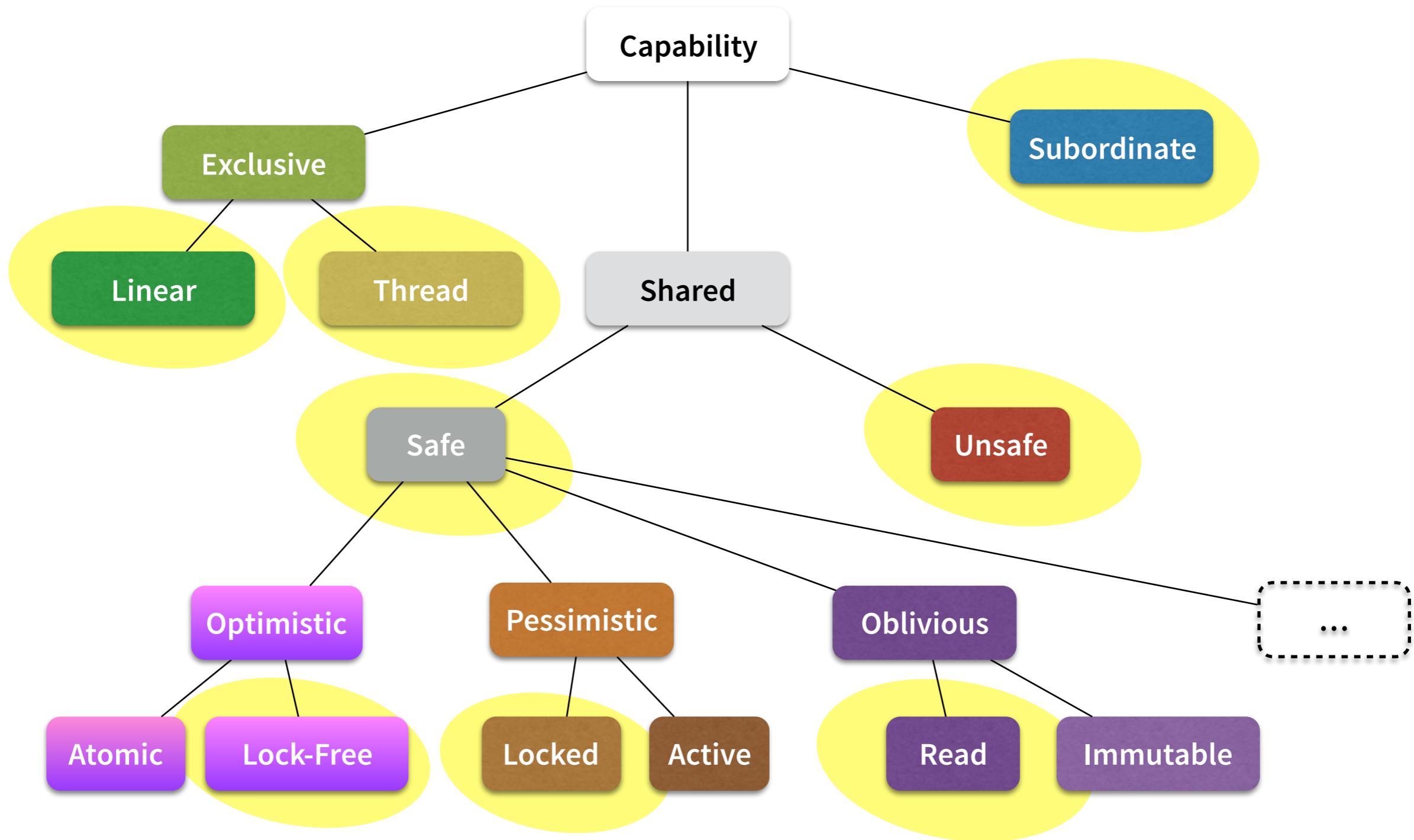
active

Actor with parallel message queues

Array Capabilities



A Hierarchy of Capabilities



Conclusions

- Reference capabilities is a promising approach for thread-safe OO programming
- Brings together ideas from a wide variety of work in a unified system

Ownership/Universe types

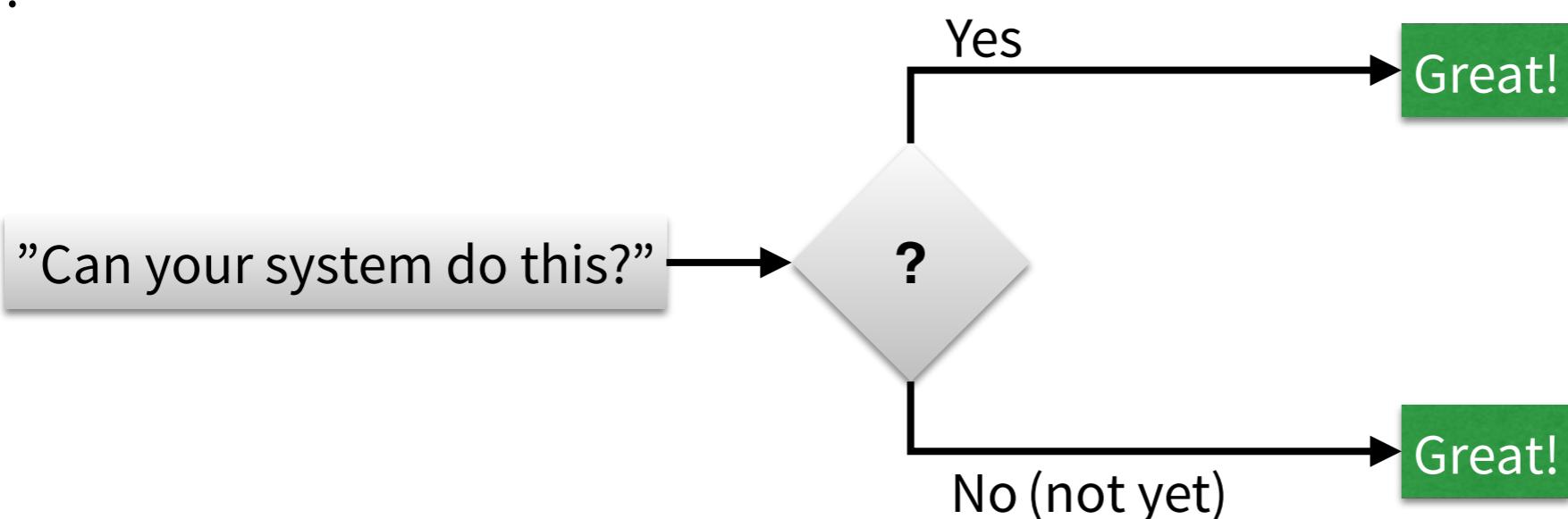
Linear/Unique references and external uniqueness

Read-only/Immutability

Regions and effects

Fractional permissions

...



Thank you!

Reference Capabilities for Concurrency Control

LUDVÍK VÁCLAVÍK
LUDVÍK VÁCLAVÍK
LUDVÍK VÁCLAVÍK

UPMARC
Uppsala Programming on Multicore
Academic Research Center

Concurrency imposes many concerns

- Is aliased?
- Accessed concurrently?
- Is thread-safe?
- Implicit/explicit synchronisation?
- Are subtypes thread-safe?

Composition \Rightarrow Parallelism

```
class Pairs = linear fst & linear snd
let f, s = consume p;
  finish(
    async{f.set(x)}
    async{s.set(y)})
]
```

Same code template safe for different use cases

```
trait Add
var first : Link
def add(e : T) : void { ... }
```

```
class List = > Add < ...
  mode
    linear Globally unique
    thread Thread-local
    locked Synchronised
    read Read-only
    subscribe Encapsulated
    unsafe Requires synch.
```

Reuse traits across different concurrency scenarios

```
class ArrayList = unsafe Add < ...
class Vector = locked Add < ...
```

Different combinations of modes express different patterns

- readers-writer locks
- regions and effects
- fractional permissions
- external uniqueness
- substitute
- And more!

✓ No data-races
Traits can assume exclusive access to the underlying object.

✓ No code duplication
Separate business logic from concurrency concerns.

✓ No effect system
References can always be used to the full extent of their types

Elias Castegren Tobias Wrigstad

Let's talk more at the poster session!



UPMARC

