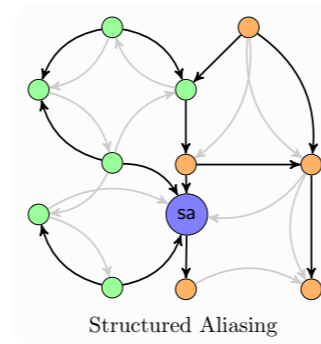


# Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16



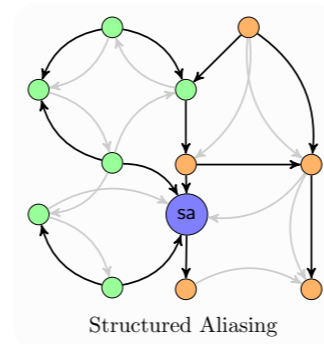
# Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16

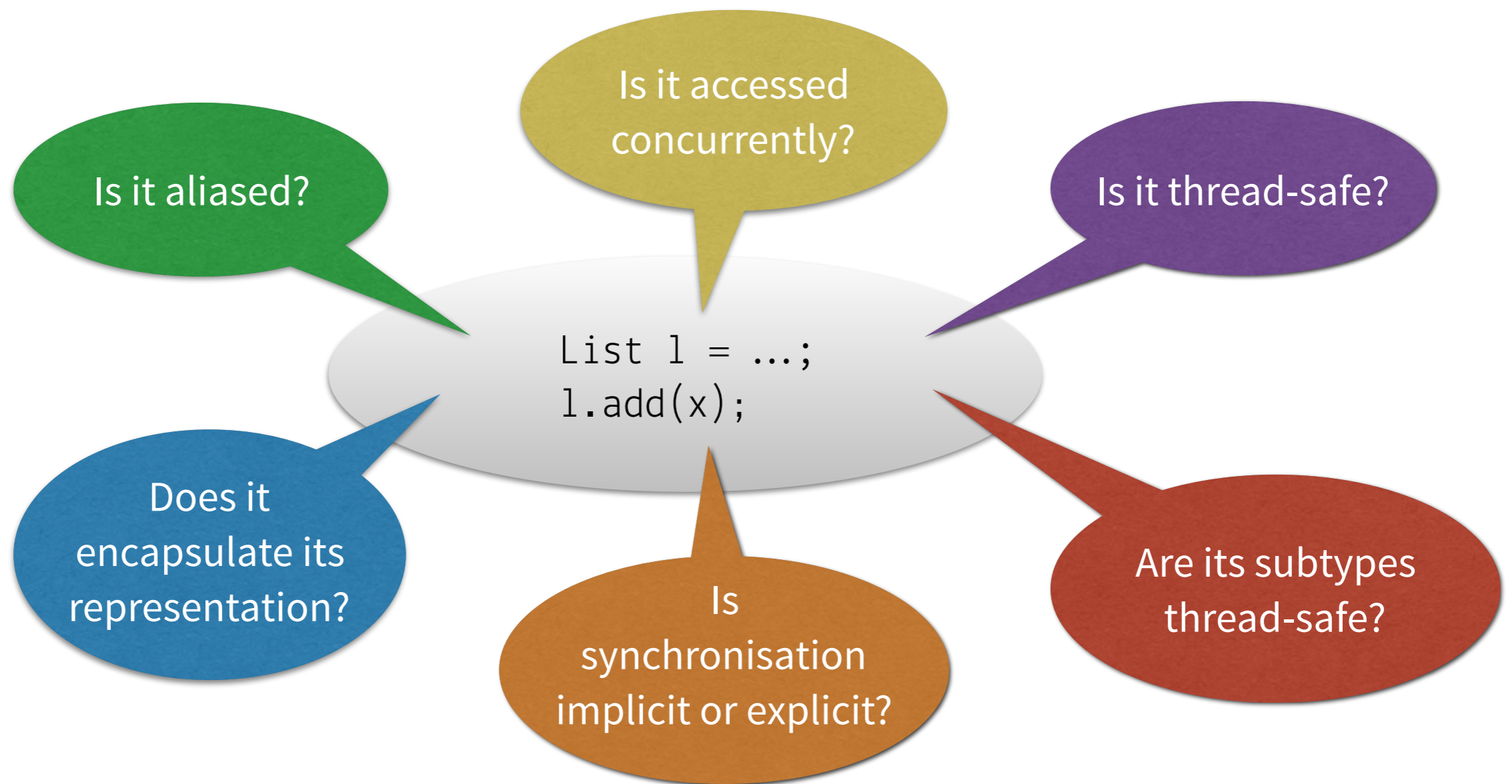


UP/MARC



# Concurrency Imposes Many Concerns

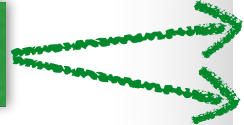
---



# Aliasing and Concurrency Control

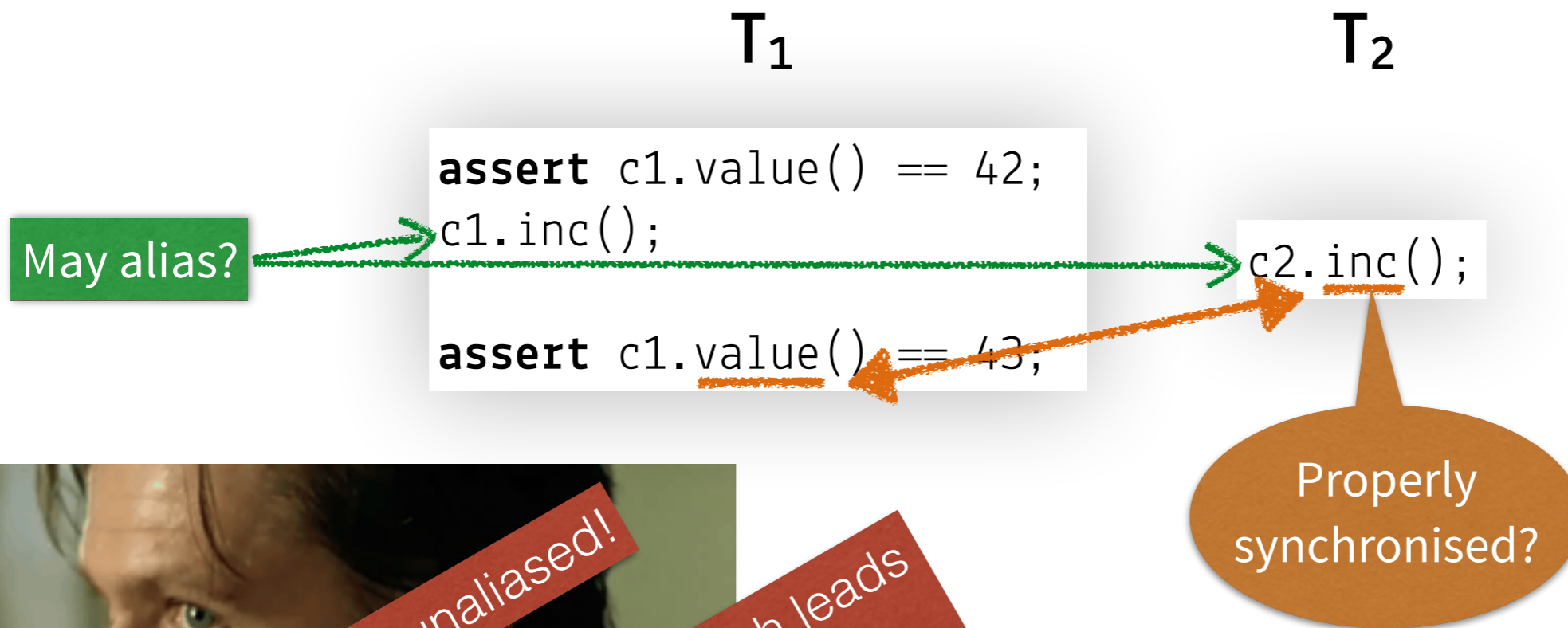
---

May alias?



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

# Aliasing and Concurrency Control



Wasteful if unaliased!

Locking too much leads to other problems

Lock EVERYONE!

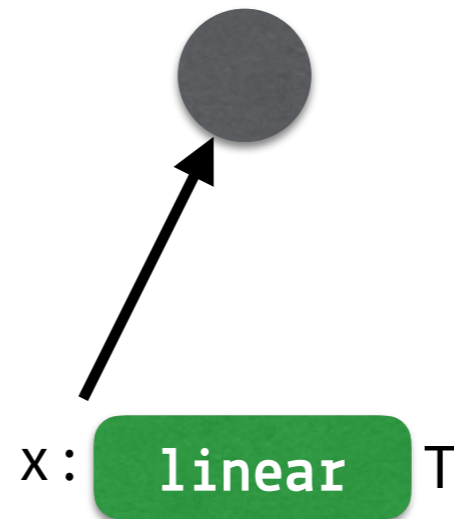
# Reference Capabilities

- 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

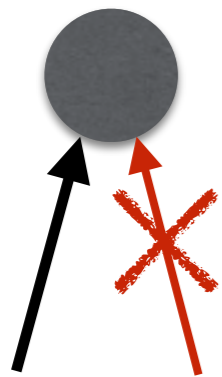
Thread-safety  $\Rightarrow$  No data-races

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



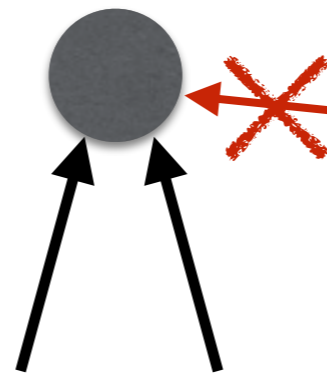
# Modes of Concurrency Control

- *Exclusive modes*



**linear**

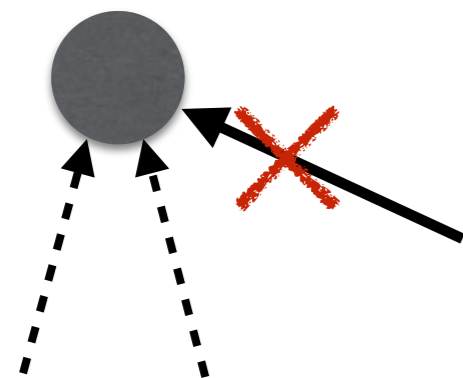
Globally unique



**thread**

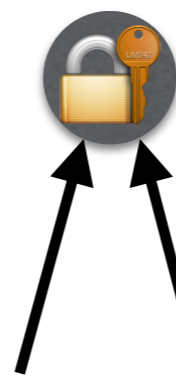
Thread-local

- *Safe modes*



**read**

Precludes mutating  
aliases



**locked**

Implicit locking

# Modes of Concurrency Control

*Dominating modes*

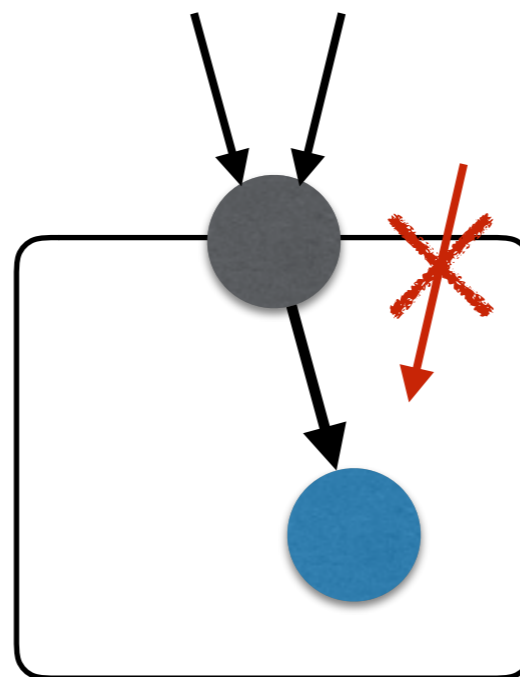
linear

thread

locked

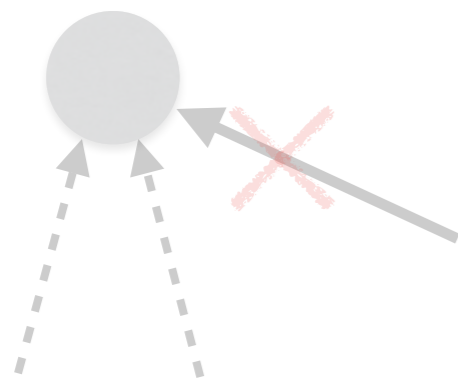
} Guarantees  
mutual exclusion

*Subordinate mode*



subordinate

Encapsulated



read

Precludes mutating  
aliases

# 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

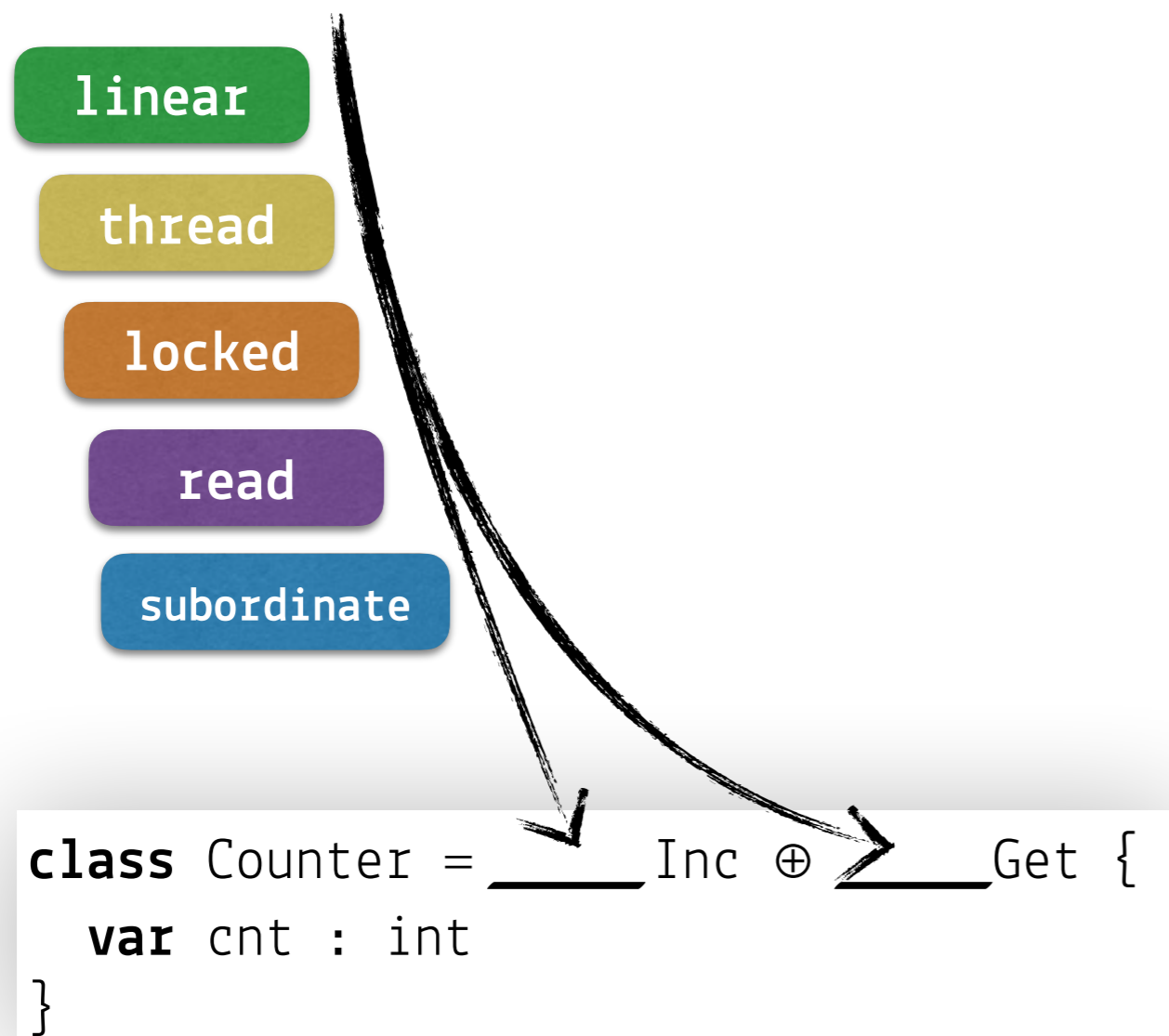
**linear** Inc — Globally unique increment capability

**locked** Inc — Implicitly synchronised increment capability

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

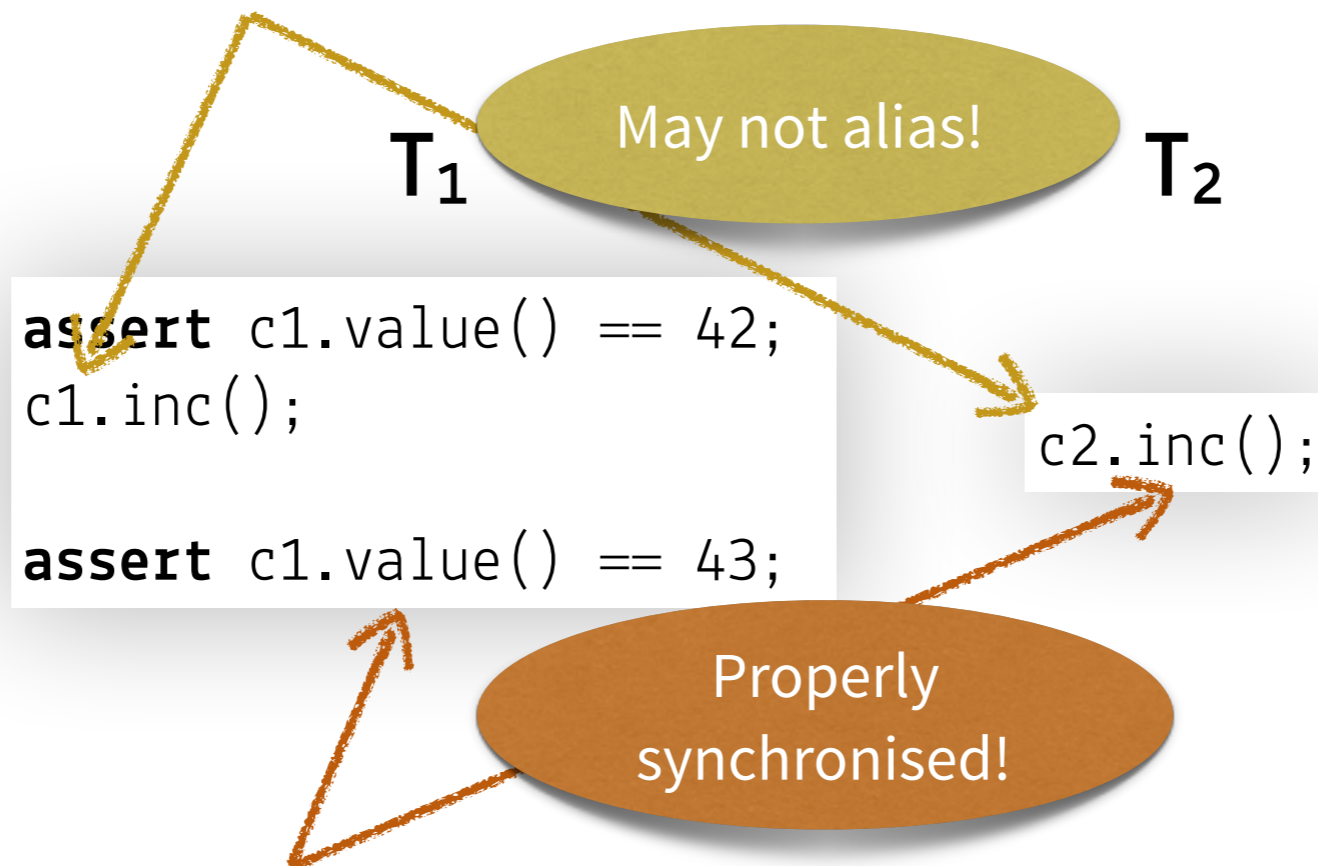
**read** Get — A read-only capability for getting the value

# Classes are Composed by Capabilities



# Aliasing and Concurrency Control (revisited)

**class** LocalCounter = **thread** Inc  $\oplus$  **read** Get



**class** SharedCounter = **locked** Inc  $\oplus$  **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  $\otimes$  linear Snd {  
  var fst : int  
  var snd : int  
}
```

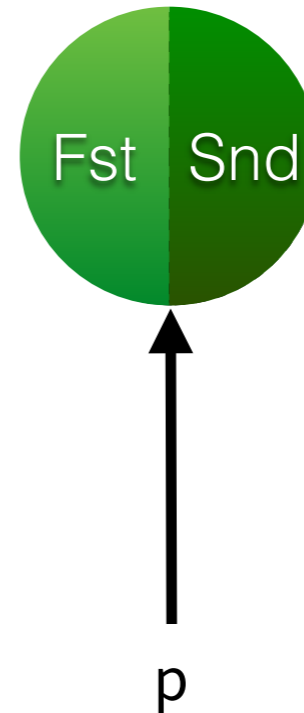


- A capability *conjunction*  $A \otimes B$  can be used as *A and B*, possibly in parallel

# Packing and Unpacking

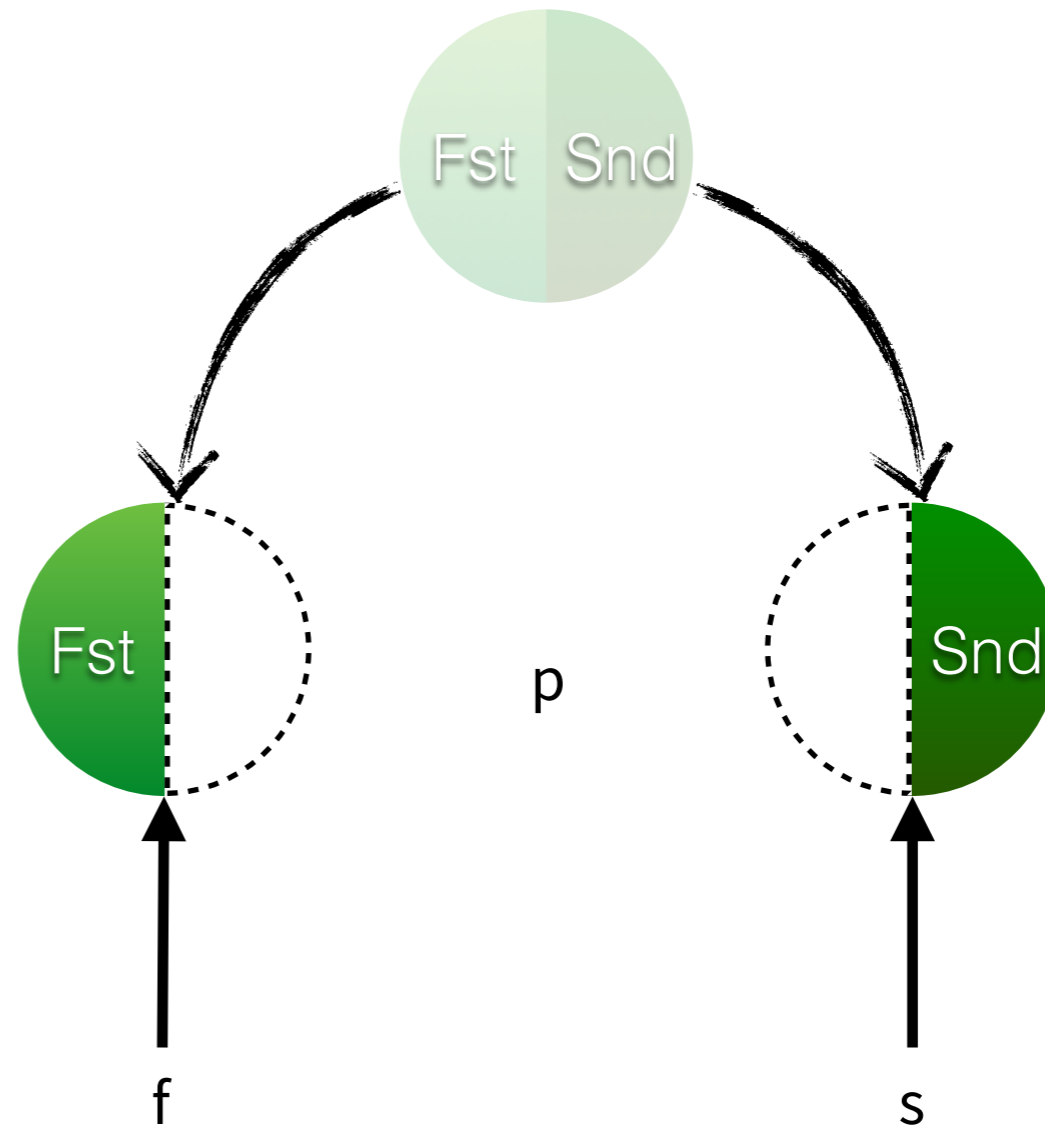
---

```
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

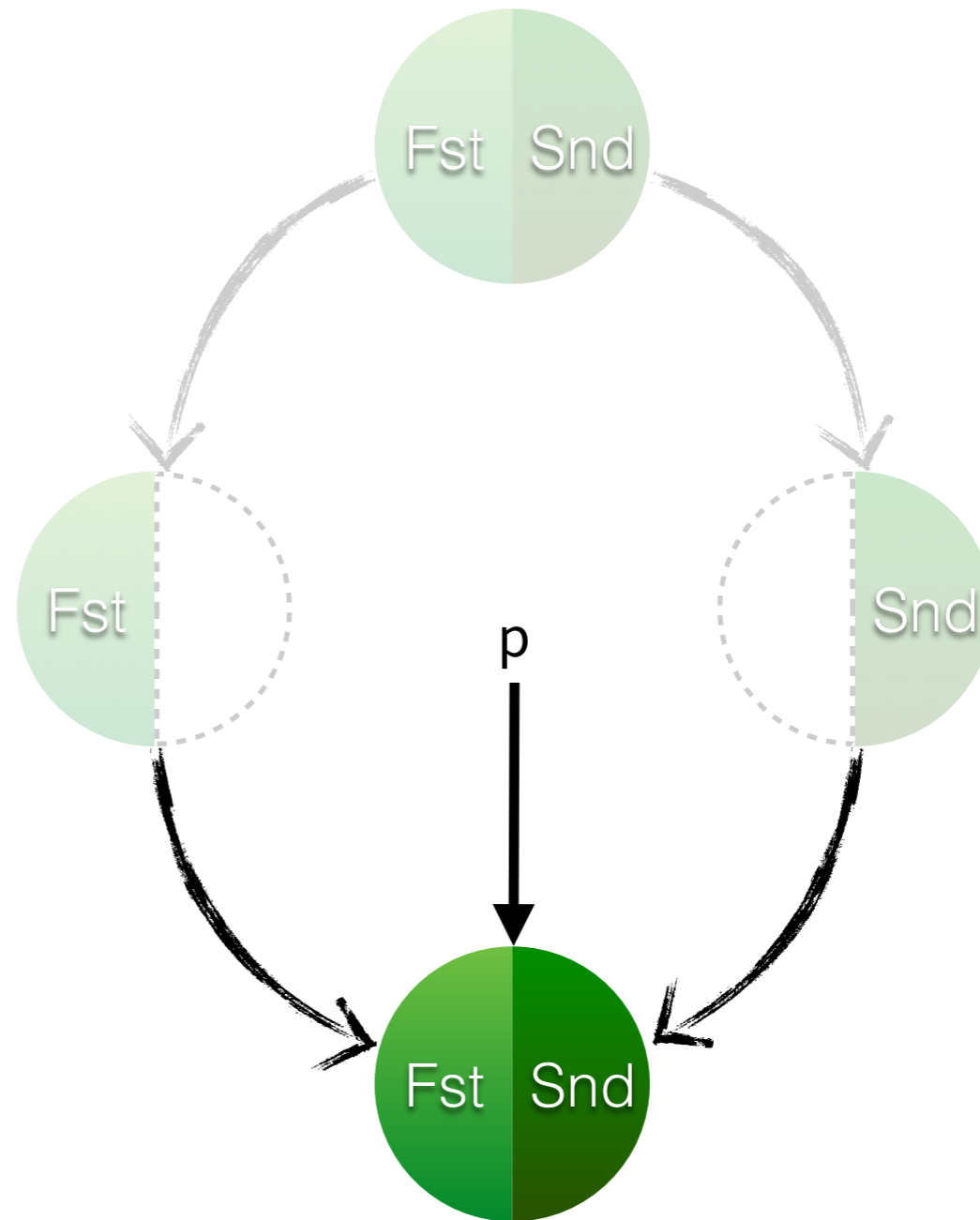
```
let p = new Pair();  
let f, s = consume p;  
finish{  
  async{f.set(x)}  
  async{s.set(y)}  
}  
p = consume f + consume s
```



# 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**



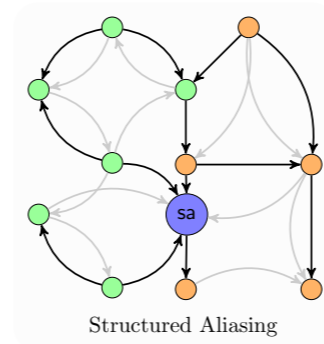
# Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16



UP/MARC



# 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

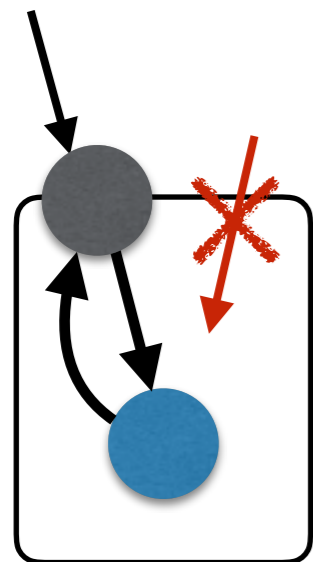
```
class List<T> = thread Add<T> ⊕ ...
  var first : Link<T>
```

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

Annotations in type declarations only

No effect tracking  
or ownership types

# Reference Capabilities as Primitives



Ownership

External uniqueness

Single writer,  
multiple readers

linear

Inc  $\oplus$

read

Get

linear

Inc  $\oplus$

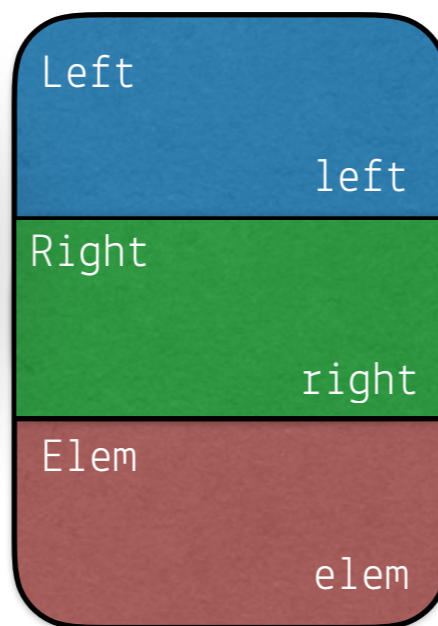
read

Get

read

Regions and effects

```
class Tree = Left  $\otimes$  Right  $\otimes$  Elem
  var left : Tree
  var right : Tree
  var elem : int
```



...  
multiple  
disjoint  
writers

linear

Inc  $\oplus$

read

Get

$A \otimes B$

A

B

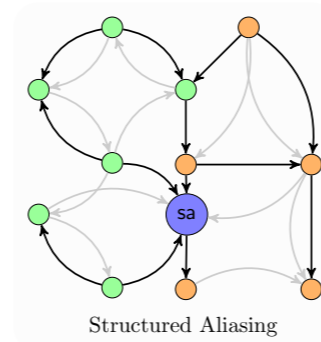
# Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO'16




UP/MARC



# 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  $\oplus$  active Get
  var cnt : int
```

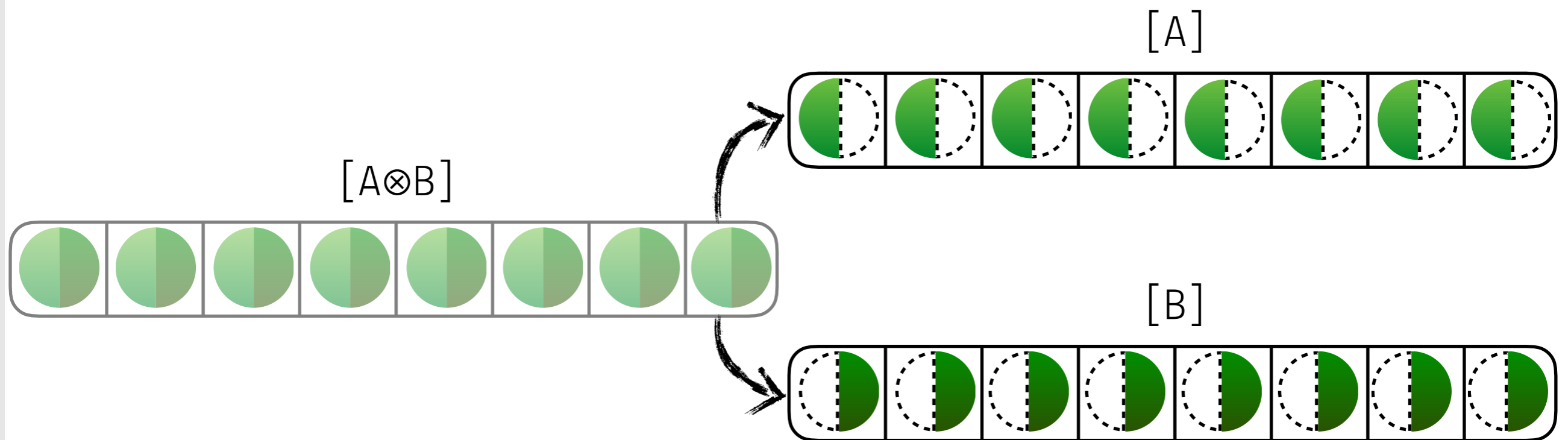
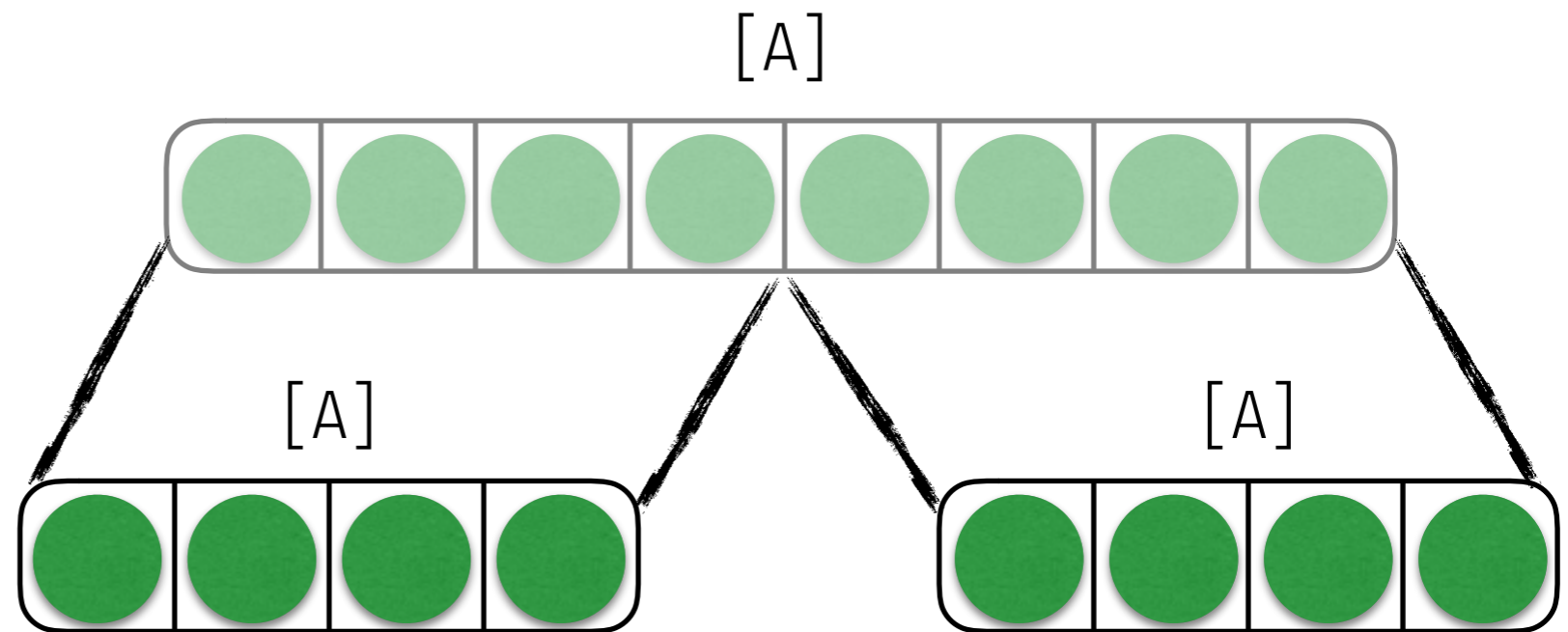
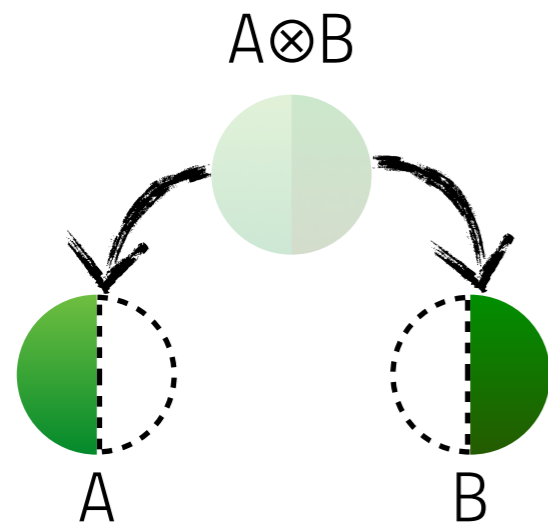
# Active Objects as a Mode of Synchronisation

---

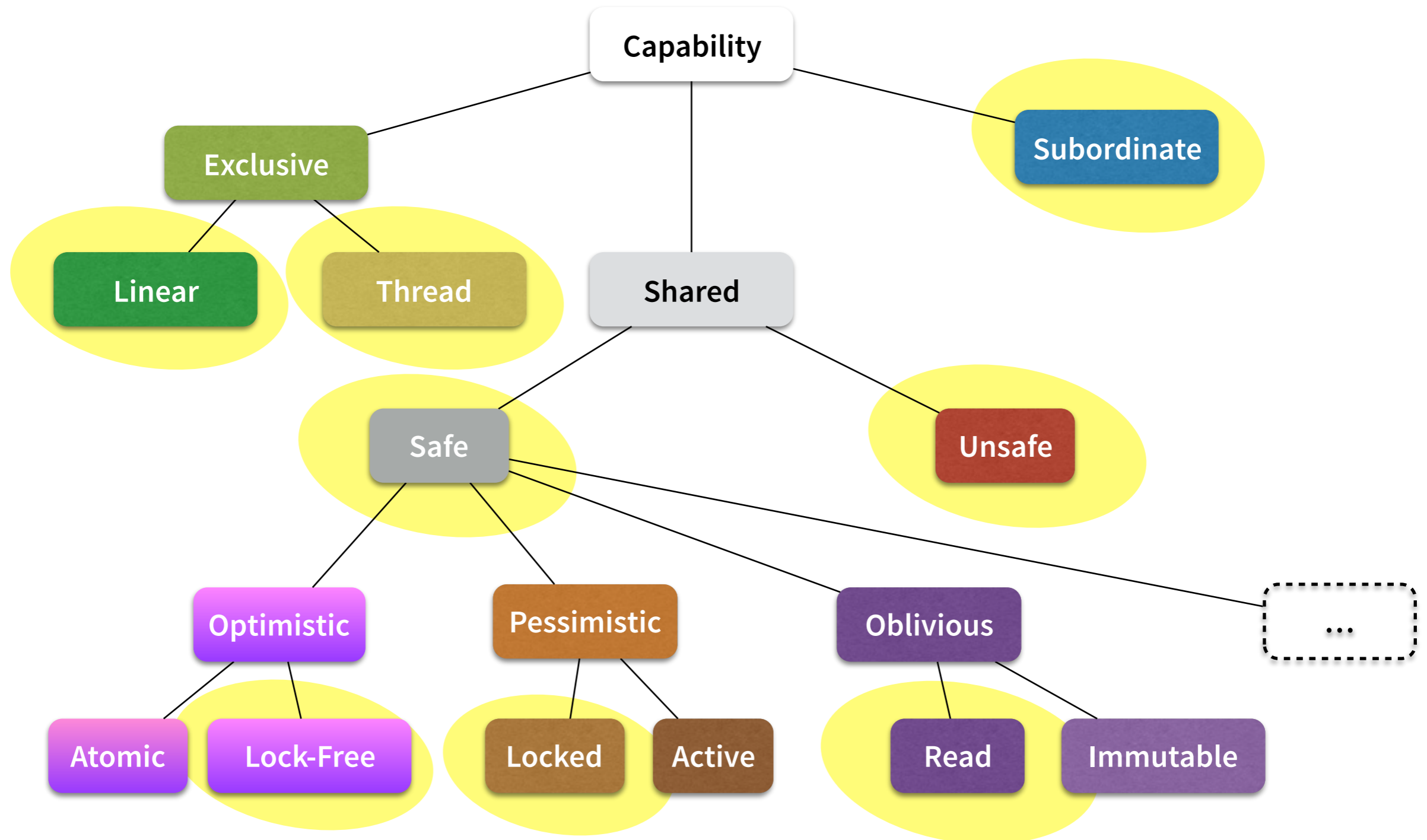
- Opens up for new combinations

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

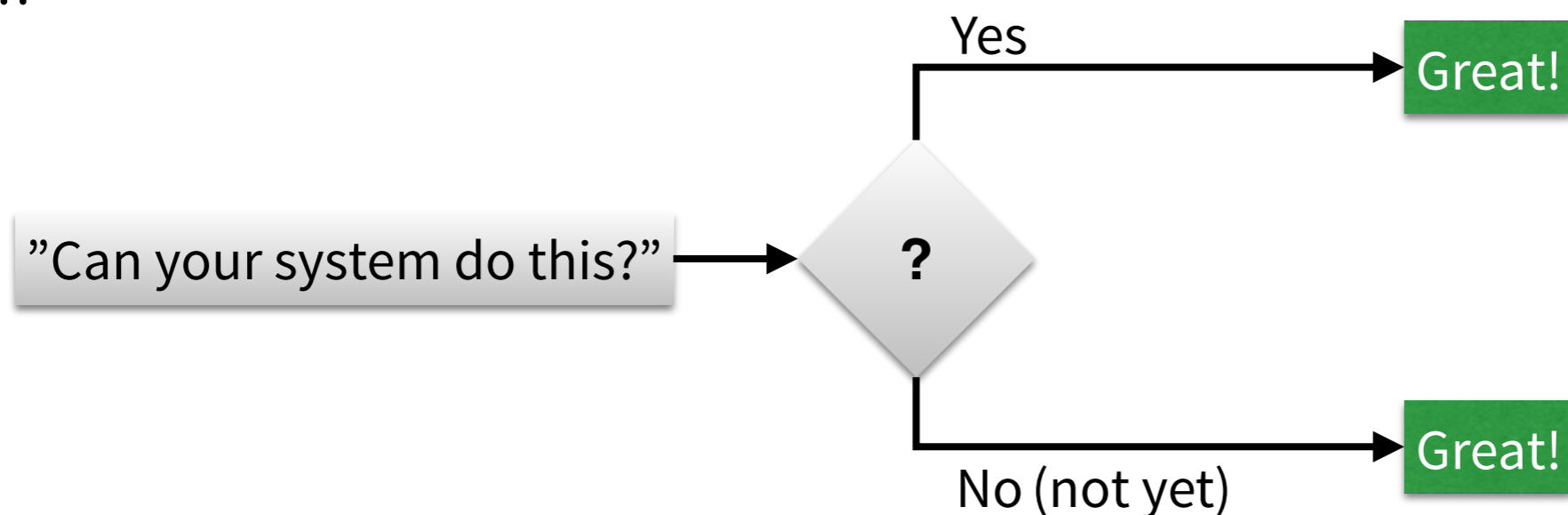
Linear/Unique references and external uniqueness

Read-only/Immutability

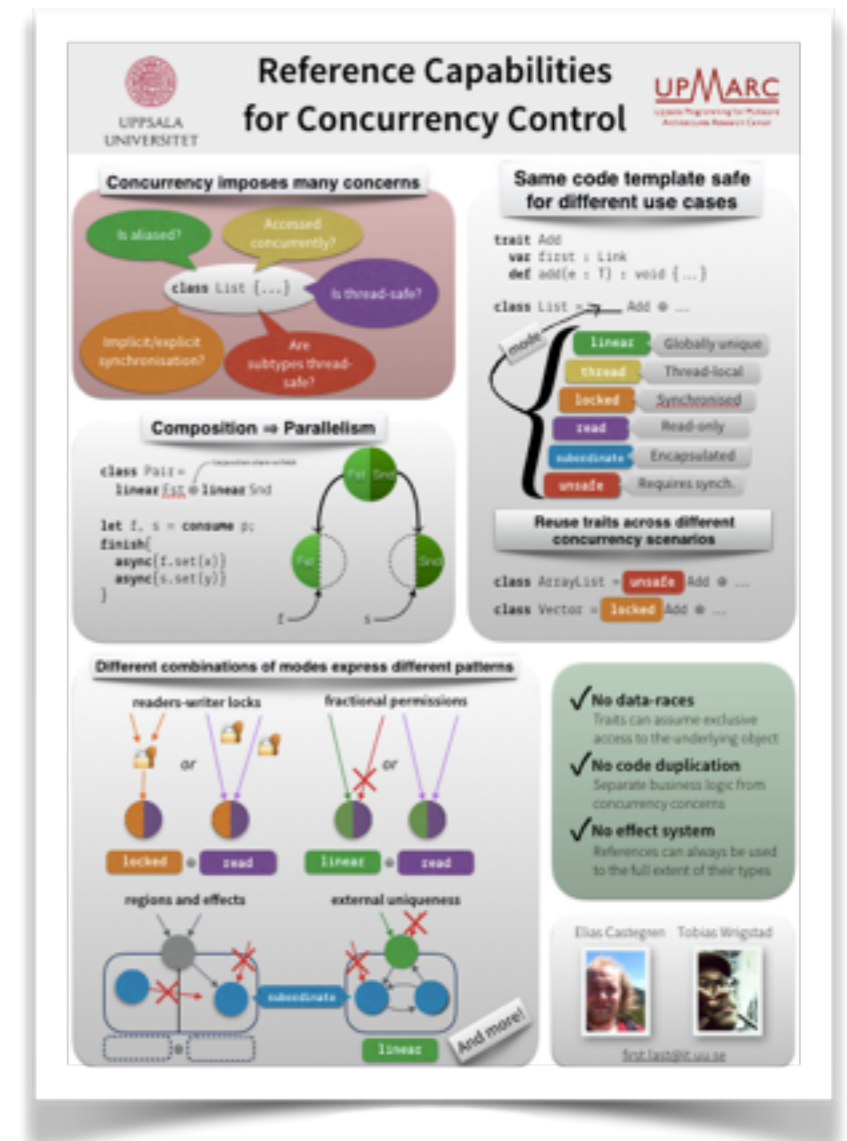
Regions and effects

Fractional permissions

...



Thank you!



Let's talk more at the poster session!

UP/MARC

