Capable: Capabilities for Scalability

Current state of design

Elias Castegren, Tobias Wrigstad
`forename.surname@it.uu.se`

IWACO 2014, Uppsala

UPPSALA
UNIVERSITET

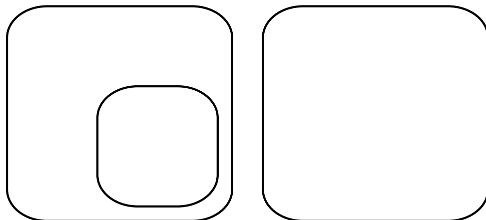
UPMARC

Introduction

- Safe parallel programming using capabilities
- Scalability and performance rather than verification

Background

- Ownership types prescribe structure

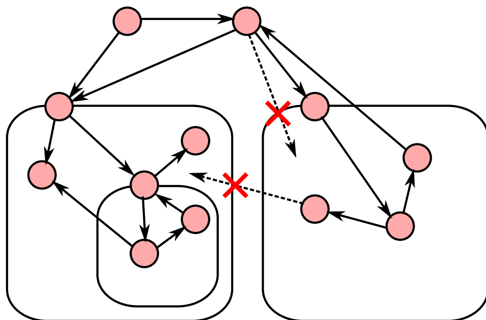


Background

- Ownership types prescribe structure

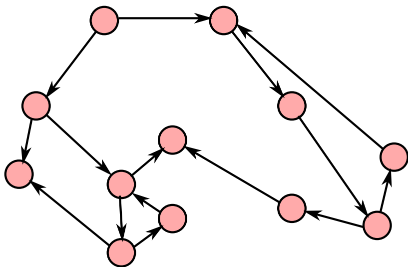
Background

- Ownership types prescribe structure



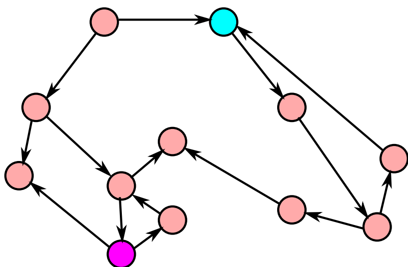
Background

- Ownership types prescribe structure
- Effect systems describe usage



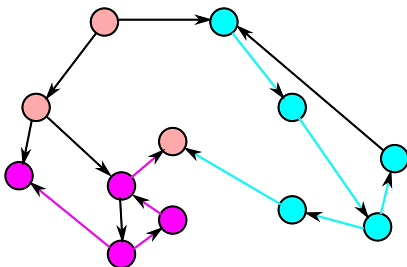
Background

- Ownership types prescribe structure
- Effect systems describe usage



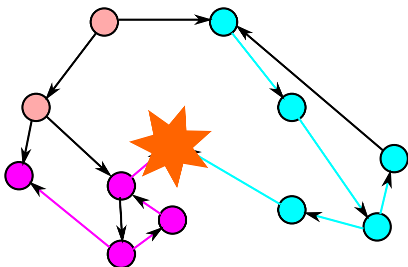
Background

- Ownership types prescribe structure
- Effect systems describe usage



Background

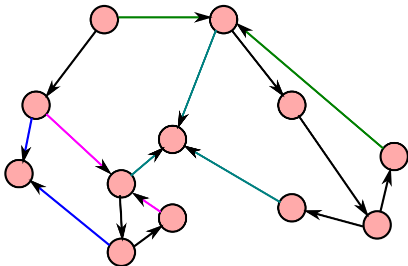
- Ownership types prescribe structure
- Effect systems describe usage



Background

- Ownership types prescribe structure
- Effect systems describe usage
- We're aiming somewhere in-between:

  Any structure is allowed as long as all aliases are non-interfering.



Outline

- Introduction
- Background
- Capabilities
- Traits and classes
- Composition, splitting and merging
- Nesting and parametricity
- Composition as abstract memory layout specification

Capabilities

- A capability governs access to some resource

Capabilities

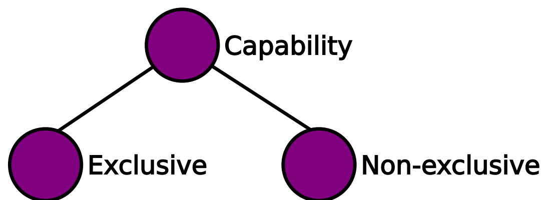
- A capability governs access to some resource
- Capability $\simeq$ (Reference, $\{$allowed operations$\}$)

Capabilities

- A capability governs access to some resource
- Capability $\simeq$ (Reference, $\{$allowed operations$\}$)

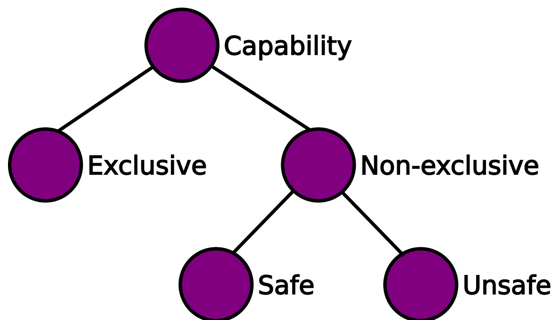
Capabilities

- A capability governs access to some resource
- Capability $\simeq$ (Reference, $\{$allowed operations$\}$)



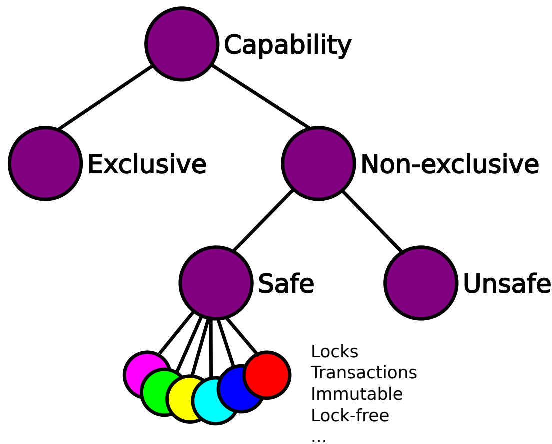
Capabilities

- A capability governs access to some resource
- Capability $\simeq$ (Reference, $\{$allowed operations$\}$)



Capabilities

- A capability governs access to some resource
- Capability $\simeq$ (Reference, $\{$allowed operations$\}$)



Safety

- Orthogonal to the design of our system: Plug in your favorite synchronization mechanism!

Safety

- Orthogonal to the design of our system: Plug in your favorite synchronization mechanism!
- Baseline:
  - No write-write conflicts outside of unsafe capabilities
  - Exclusive capabilities are (and remain) exclusive

Safety

- Orthogonal to the design of our system: Plug in your favorite synchronization mechanism!
- Baseline:
  - No write-write conflicts outside of unsafe capabilities
  - Exclusive capabilities are (and remain) exclusive
- Our focus: Lock-free capabilities with static support for speculation and publication of values (see paper for more details)

Traits

- Capabilities are introduced using traits:

```
safe trait Get{
  require int value;
  int get(){
    return this.value;
  }
}
```

Traits

- Capabilities are introduced using traits:

```
safe trait Get{
  require int value;
  int get(){
    return this.value;
  }
}
```

- Traits are exclusive by default:

```
trait Set{
  require int value;
  void set(int val){
    this.value = val;
  }
}
```

Classes

```
trait Set{
  require int value;
  void set(int val)...
```

```
safe trait Get{
  require int value;
  int get()...
```

- Classes are formed by composing traits:

```
class Cell = Set ⊕ Get*{
  provide int value;
}
```

Classes

```
trait Set{
  require int value;
  void set(int val)...
```

```
safe trait Get{
  require int value;
  int get()...
```

- Classes are formed by composing traits:

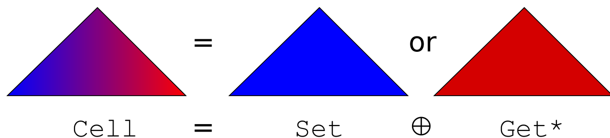
```
class Cell = Set ⊕ Get*{
  provide int value;
}
```

- Class types are composite capabilities:

```
Cell c = new Cell;
c.set(42);
c.get();   // = 42
```

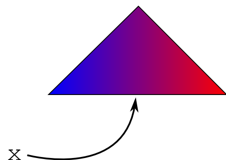
Composition and splitting

- A *disjunction* can be split into *either* of its components:



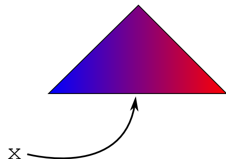
Composition and splitting (`Cell = Set ⊕ Get*`)

```
x = new Cell;
```

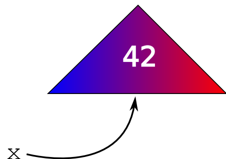


Composition and splitting (`Cell = Set` $\oplus$ `Get`*)

`x = new Cell;`

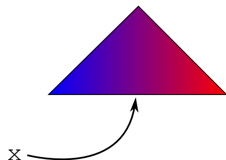


`x.set(42);`

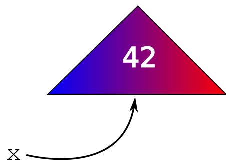


Composition and splitting (`Cell = Set` $\oplus$ `Get`*)

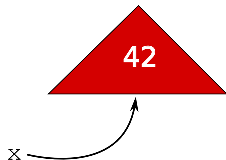
```
x = new Cell;
```




```
x.set(42);
```

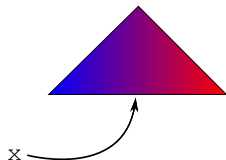



```
x = (Get*) consume x;
```

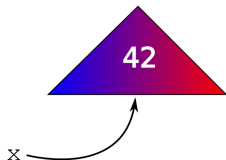


Composition and splitting (`Cell` = `Set` $\oplus$ `Get`*)

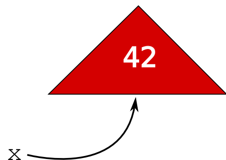
```
x = new Cell;
```

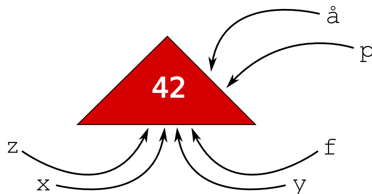



```
x.set(42);
```



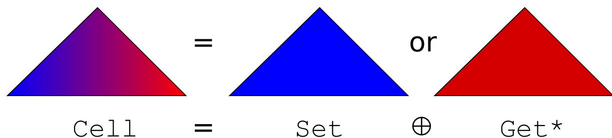

```
x = (Get*) consume x;
```



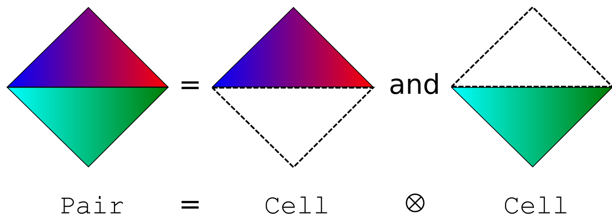


Composition and splitting

- A *disjunction* can be split into *either* of its components:

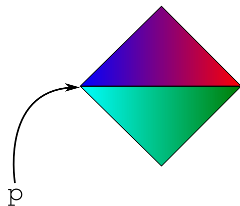


- A *conjunction* can be split into *both* of its components:



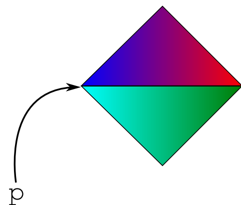
Composition and splitting (Pair = Cell ⊗ Cell)

```
p = new Pair;
```

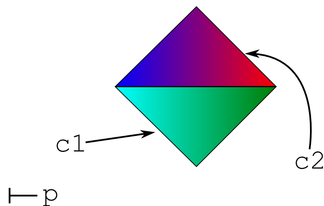


Composition and splitting (`Pair = Cell` $\otimes$ `Cell`)

```
p = new Pair;
```

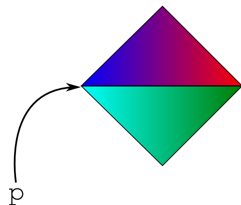



```
Cell c1, c2 = consume p;
```

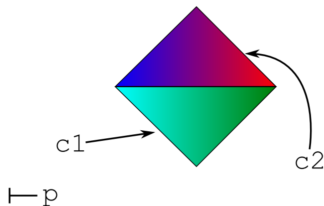


Composition and splitting (`Pair = Cell` ⊗ `Cell`)

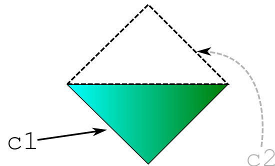
```
p = new Pair;
```



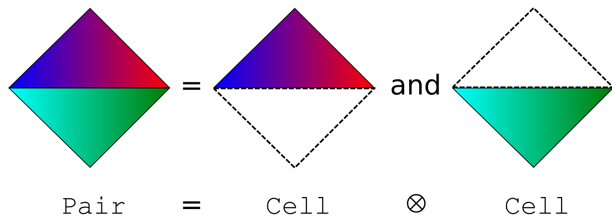

```
Cell c1, c2 = consume p;
```



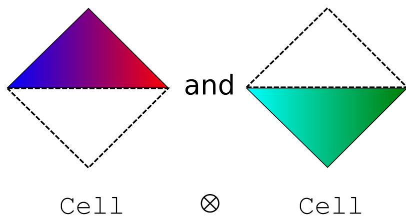
`c1` and `c2` are aliases, but can only access "their half" of the `Pair`:



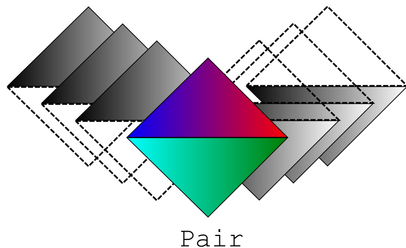
Splitting and merging



Splitting and merging



Splitting and merging



Merging

- Structured split and merge

```
Pair p = ...;
split p into Cell c1, c2 in{
  ...  // p is invalidated
}
...  // p is reinstated
```

Merging

- Structured split and merge

```
Pair p = ...;
split p into Cell c1, c2 in{
  ...  // p is invalidated
}
...  // p is reinstated
```

- Unstructured split and merge

```
Pair p = ...;
Cell c1, c2 = consume p;
...
p = consume c1 ⊗ consume c2
...  // c1 and c2 are invalidated
```

Merging

- Structured split and merge

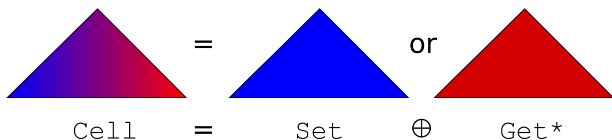
```
Pair p = ...;
split p into Cell c1, c2 in{
  ...  // p is invalidated
}
...  // p is reinstated
```

- Unstructured split and merge

```
Pair p = ...;
Cell c1, c2 = consume p;
...
p = consume c1 ⊗ consume c2 ← dynamic alias check!
...  // c1 and c2 are invalidated
```

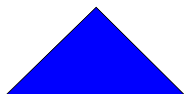
Merging

- What about disjunction?



Merging

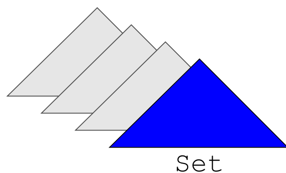
- What about disjunction?



Set

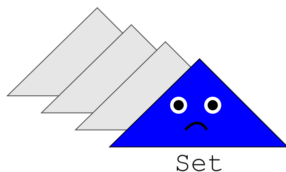
Merging

- What about disjunction?



Merging

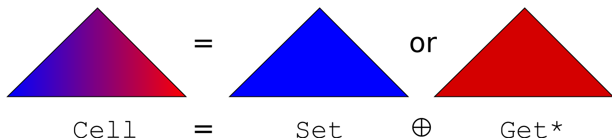
- What about disjunction?



- The `Get` capability is lost!

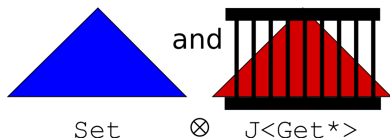
Splitting with Jails

- `Cell = Set` $\oplus$ `Get`* means `Set` and `Get`* may not be used in parallel



Splitting with Jails

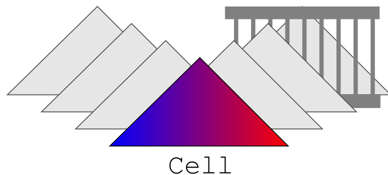
- `Cell = Set` $\oplus$ `Get`* means `Set` and `Get`* may not be used in parallel



- A *jailed* capability is an alias with an empty interface

Splitting with Jails

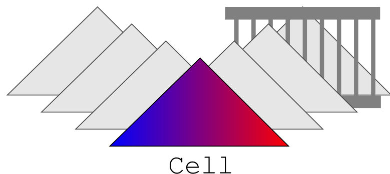
- `Cell = Set` $\oplus$ `Get`* means `Set` and `Get`* may not be used in parallel



- A *jailed* capability is an alias with an empty interface

Splitting with Jails

- `Cell = Set` $\oplus$ `Get`* means `Set` and `Get`* may not be used in parallel



- A *jailed* capability is an alias with an empty interface
- Jails can turn any disjunction $\mathtt{c}_1 \oplus \mathtt{c}_2$ into a conjunction $\mathtt{c}_1 \otimes \mathtt{J}\langle\mathtt{c}_2\rangle$, which can be split and merged in a non-lossy way

Recap

- Capabilities are either *exclusive*, *safe* or *unsafe*

Recap

- Capabilities are either *exclusive*, *safe* or *unsafe*
- *Traits* specify behaviour and introduce capability types.

Recap

- Capabilities are either *exclusive*, *safe* or *unsafe*
- *Traits* specify behaviour and introduce capability types.
- *Classes* introduce composite capability types

Recap

- Capabilities are either *exclusive*, *safe* or *unsafe*
- *Traits* specify behaviour and introduce capability types.
- *Classes* introduce composite capability types
  - A *disjunction* $c_1 \oplus c_2$ can be split into *either* c_1 or c_2

Recap

- Capabilities are either *exclusive*, *safe* or *unsafe*
- *Traits* specify behaviour and introduce capability types.
- *Classes* introduce composite capability types
  - A *disjunction* $c_1 \oplus c_2$ can be split into *either* c_1 or c_2
  - A *conjunction* $c_1 \otimes c_2$ can be split into *both* c_1 and c_2 which may be used in parallel

Recap

- Capabilities are either *exclusive*, *safe* or *unsafe*
- *Traits* specify behaviour and introduce capability types.
- *Classes* introduce composite capability types
  - A *disjunction* $c_1 \oplus c_2$ can be split into *either* c_1 or c_2
  - A *conjunction* $c_1 \otimes c_2$ can be split into *both* c_1 and c_2 which may be used in parallel
- Merging can be used to regain composite capabilities in both structured and unstructured ways

Co-encapsulation through nesting

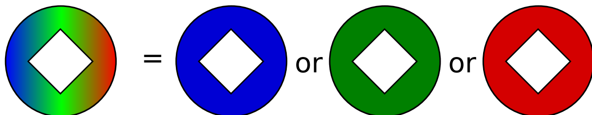
- Parametricity exposes internal details about a nested capability

```
class List⟨T⟩ = Add⟨T⟩ ⊕ Del⟨T⟩ ⊕ Nth*⟨T⟩{
  provide Link⟨T⟩ first;
}
```

Co-encapsulation through nesting

- Parametricity exposes internal details about a nested capability

```
class List⟨T⟩ = Add⟨T⟩ ⊕ Del⟨T⟩ ⊕ Nth*⟨T⟩{
  provide Link⟨T⟩ first;
}
```

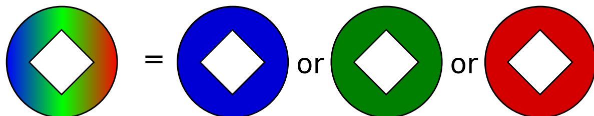


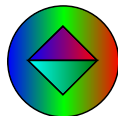
List<T> = Add<T> ⊕ Del<T> ⊕ Nth*<T>

Co-encapsulation through nesting

- Parametricity exposes internal details about a nested capability

```
class List⟨T⟩ = Add⟨T⟩ ⊕ Del⟨T⟩ ⊕ Nth*⟨T⟩{
  provide Link⟨T⟩ first;
}
```

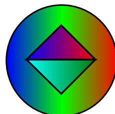




Nesting and splitting

$$\texttt{List}\langle\texttt{T}\rangle = \texttt{Add}\langle\texttt{T}\rangle \oplus \texttt{Del}\langle\texttt{T}\rangle \oplus \texttt{Nth}^{*}\langle\texttt{T}\rangle$$

$$\texttt{Pair} = \texttt{Cell} \otimes \texttt{Cell}$$

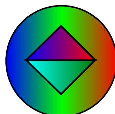


List<Pair>

Nesting and splitting

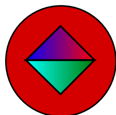
$$\texttt{List}\langle\texttt{T}\rangle = \texttt{Add}\langle\texttt{T}\rangle \oplus \texttt{Del}\langle\texttt{T}\rangle \oplus \texttt{Nth}^{*}\langle\texttt{T}\rangle$$

$$\texttt{Pair} = \texttt{Cell} \otimes \texttt{Cell}$$



List<Pair>

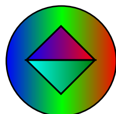
Outer split:



Nth*<Pair>

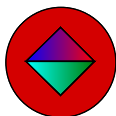
Nesting and splitting

$$\texttt{List}\langle \texttt{T}\rangle = \texttt{Add}\langle \texttt{T}\rangle \oplus \texttt{Del}\langle \texttt{T}\rangle \oplus \texttt{Nth}^{*}\langle \texttt{T}\rangle$$
$$\texttt{Pair} = \texttt{Cell} \otimes \texttt{Cell}$$



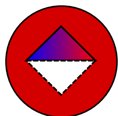
List<Pair>

Outer split:

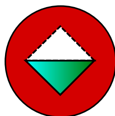


Nth*<Pair>

Inner split:



and



Nth*<Cell> ⊗ Nth*<Cell>

Composition as abstract memory layout specification

- Cache locality is good! Data accessed together by a single thread should be on the same cache line

Composition as abstract memory layout specification

- Cache locality is good! Data accessed together by a single thread should be on the same cache line
- False sharing is bad! Disjoint data accessed in parallel should be on separate cache lines

Composition as abstract memory layout specification

- Cache locality is good! Data accessed together by a single thread should be on the same cache line
- False sharing is bad! Disjoint data accessed in parallel should be on separate cache lines

- Different splitting semantics suggest different access patterns:

  $\texttt{c} = \texttt{c}_1 \oplus \texttt{c}_2 \Rightarrow \texttt{c}_1$ and $\texttt{c}_2$ accessed together or separately

Composition as abstract memory layout specification

- Cache locality is good! Data accessed together by a single thread should be on the same cache line
- False sharing is bad! Disjoint data accessed in parallel should be on separate cache lines

- Different splitting semantics suggest different access patterns:

  $\texttt{c} = \texttt{c}_1 \oplus \texttt{c}_2 \Rightarrow \texttt{c}_1$ and $\texttt{c}_2$ accessed together or separately
  Keep $\texttt{c}_1$ and $\texttt{c}_2$'s resources on the same cache line!

Composition as abstract memory layout specification

- Cache locality is good! Data accessed together by a single thread should be on the same cache line
- False sharing is bad! Disjoint data accessed in parallel should be on separate cache lines

- Different splitting semantics suggest different access patterns:

  $c = c_1 \oplus c_2 \Rightarrow c_1$ and c_2 accessed together or separately
  Keep c_1 and c_2's resources on the same cache line!

  $c = c_1 \otimes c_2 \Rightarrow c_1$ and c_2 may be accessed in parallel

Composition as abstract memory layout specification

- Cache locality is good! Data accessed together by a single thread should be on the same cache line
- False sharing is bad! Disjoint data accessed in parallel should be on separate cache lines

- Different splitting semantics suggest different access patterns:

  $c = c_1 \oplus c_2 \Rightarrow c_1$ and c_2 accessed together or separately
  Keep c_1 and c_2's resources on the same cache line!

  $c = c_1 \otimes c_2 \Rightarrow c_1$ and c_2 may be accessed in parallel
  Keep c_1 and c_2's resources on different cache lines!

What else?

- Merging non-exclusive capabilities
- Aliasing exclusive capabilities
- Lock-free capabilities

What else?

- Merging non-exclusive capabilities
- Aliasing exclusive capabilities
- Lock-free capabilities

- See paper for more details

Summary

- Safe aliasing through capability splitting (and merging)
- Capability composition hints efficient memory layout
- Lock-free capabilities for lock-free data structures

Summary

- Safe aliasing through capability splitting (and merging)
- Capability composition hints efficient memory layout
- Lock-free capabilities for lock-free data structures

What now?

- Implementation and evaluation
- Extend the support for lock-free data structures
- Add high level abstractions (e.g. Reagents [Turon 12])

Summary

- Safe aliasing through capability splitting (and merging)
- Capability composition hints efficient memory layout
- Lock-free capabilities for lock-free data structures

What now?

- Implementation and evaluation
- Extend the support for lock-free data structures
- Add high level abstractions (e.g. Reagents [Turon 12])

Thank you!