Attached and Detached Closures in Actors

Elias Castegren

KTH Royal Institute of Technology

Dave Clarke, Kiko Fernandez-Reyes, Tobias Wrigstad, Albert Mingkun Yang

Uppsala University

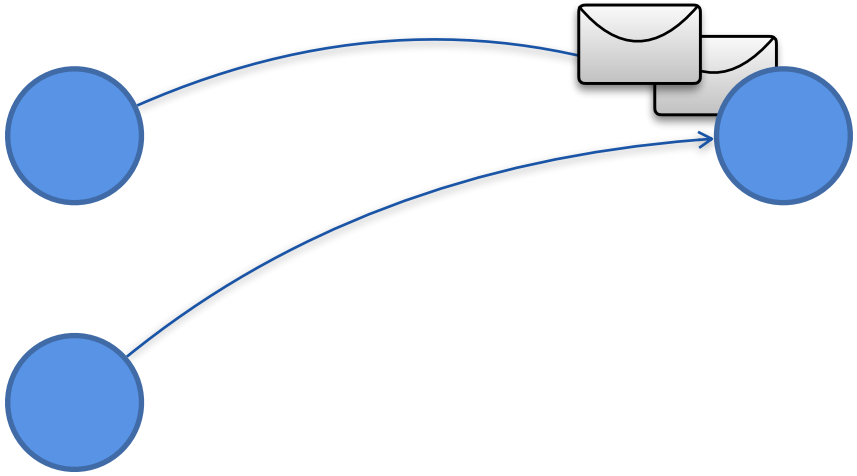
What Tobias Said

- The Encore programming language
  - Object orientation + actors
  - Guarantees safe sharing of objects between actors
- Handling both concurrency and parallelism in the actor model
- Lessons learned & Open questions

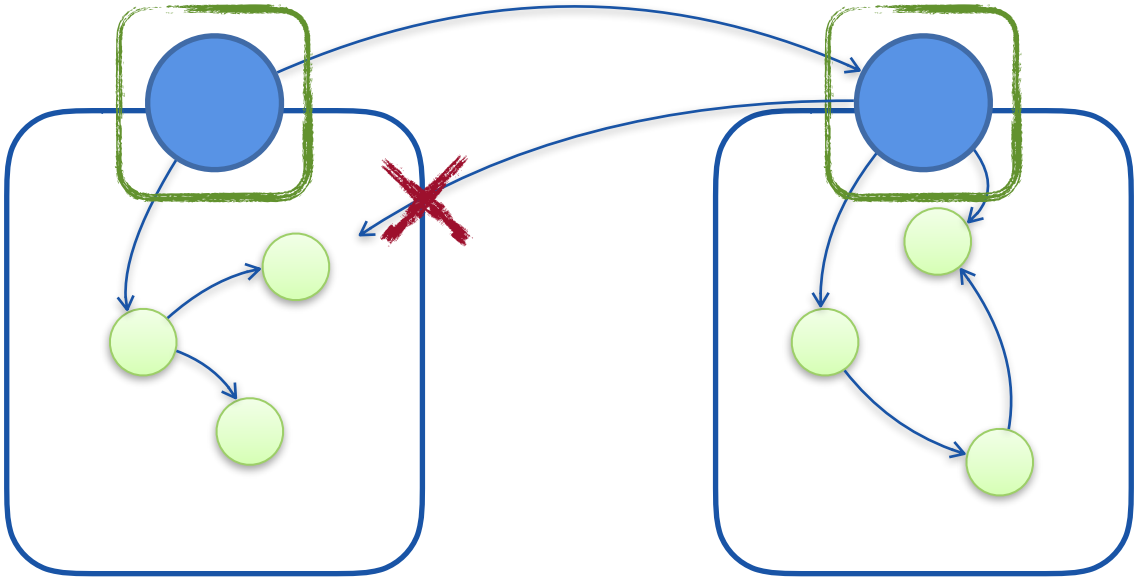
This Talk

- State-capturing closures in an actor-setting
- Current and future solutions in Encore
- Terminology for discussing closure semantics

We All Like Actors



We All Like Actors



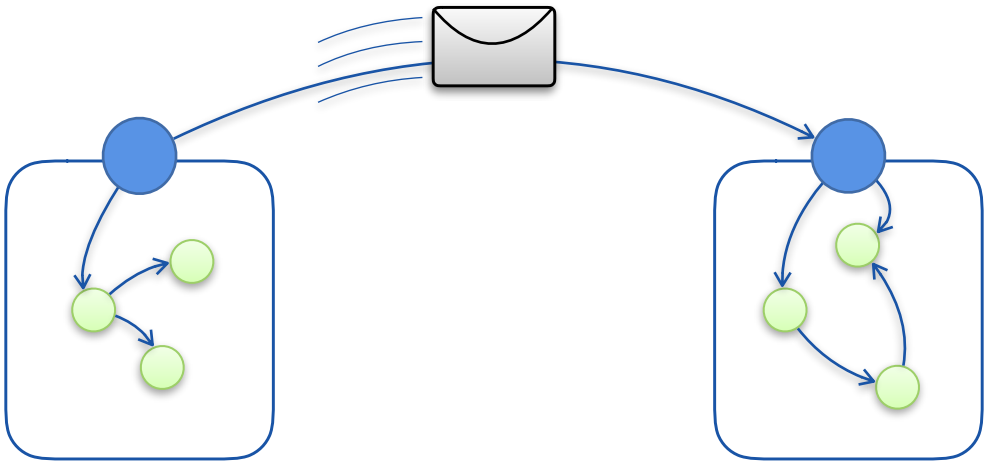
Some of Us Like Functional Programming

- Functional programming plays nicely with the actor model
  - Algebraic data-types
  - Immutability
  - Higher-order functions
  - …

```
data List a =
            | Nil
            | Cons a (List a)
```

$\lambda\, x : t.\, x + 42$

- Examples include Erlang and Elixir



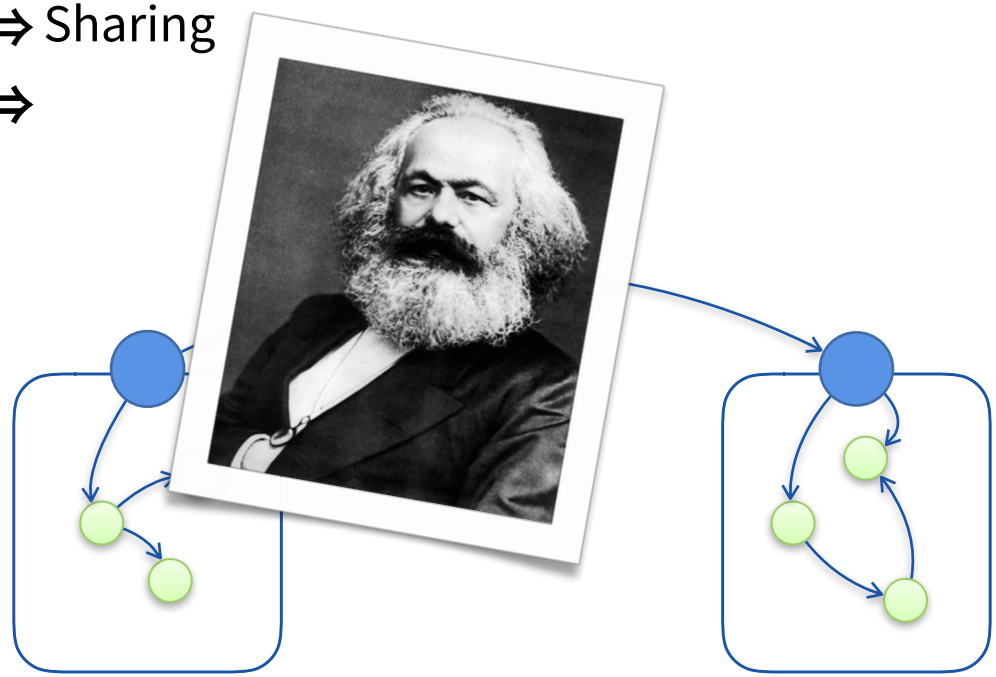
Some of Us Also Like Object Orientation

- Actor programming is familiar to OO programmers
  - Actors can be thought of as ”active” objects
  - Sending Messages ≈ Calling Methods

- OO relies heavily on mutable state and aliasing

  ⇒ Sharing

  ⇒



Some of Us Also Like Object Orientation

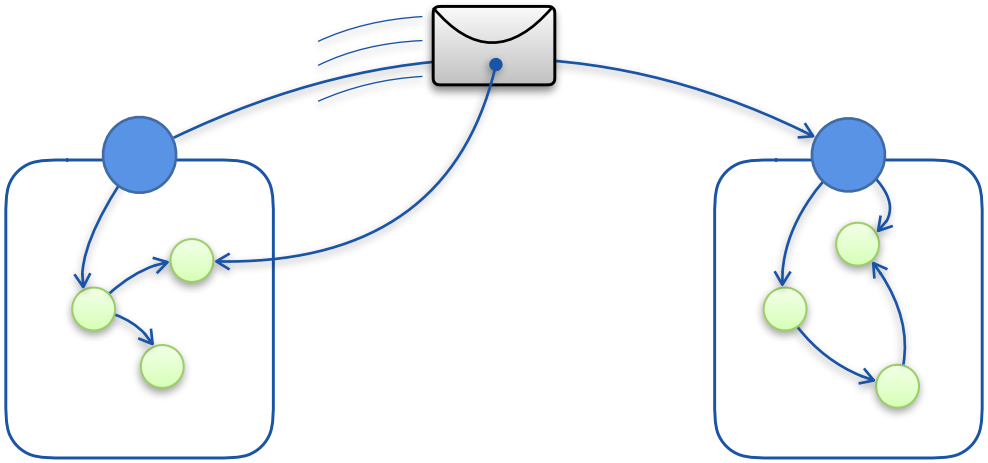
- Actor programming is familiar to OO programmers
  - Actors can be thought of as ”active” objects
  - Sending Messages ≈ Calling Methods

- OO relies heavily on mutable state and aliasing

  ⟹ Sharing

  ⟹ Data-races

  ⟹ Loss of actor isolation!



Making Actors and OO Play Nice

- Capability-based languages/systems, type systems
  - Encore
  - Pony [Clebsch et al.]
  - LaCasa (for Scala) [Haller & Loiko]
  - Joelle [Östlund et al.]
- Relying on delegation of method calls
  - e.g. far references in AmbientTalk [Dedecker et al.]
- Relying on copying of (passive) objects
  - e.g. Proactive [Caromel et al.]

Encore Primer/Reminder

```
active class Actor
  var count : int
  val other : Actor

  def work() : unit
    val fut = this.other ! compute()

    val result = get fut
    this.print(result)
  end

  def print(v : Data) : unit
    this.count += 1
    ... // Print the value
  end
  ...
end
```

Actors introduced via classes

Message passing

Synchronisation via futures

Capabilities for Concurrency Control

- Every reference carries a capability (tracked by the type system)
  - **`linear`** — No aliases, transfer semantics
  - **`local`** — Local to its creating actor
  - **`read`** — Read-only reference (no mutable aliases)
  - **`active`** — Actor reference (asynchronous communication)
  - …

Capabilities for Concurrency Control

- Every reference carries a capability (tracked by the type system)
  - **`linear`** — No aliases, transfer semantics
  - **`local`** — Local to its creating actor
  - **`read`** — Read-only reference (no mutable aliases)
  - **`active`** — Actor reference (asynchronous communication)
  - …

```
local class Counter
  var cnt : int
  ...
end

linear class List
  var first : Node
  ...
end
```

```
var c = new Counter
actor ! foo(c)
```

Can't share local object

```
var l = new List
actor ! bar(consume l)
```

Avoiding Blocking on Futures (Chaining)

```
active class Actor
  var count : int
  val other : Actor

  def work() : unit
    val fut = this.other ! compute()
```

Induces waiting times

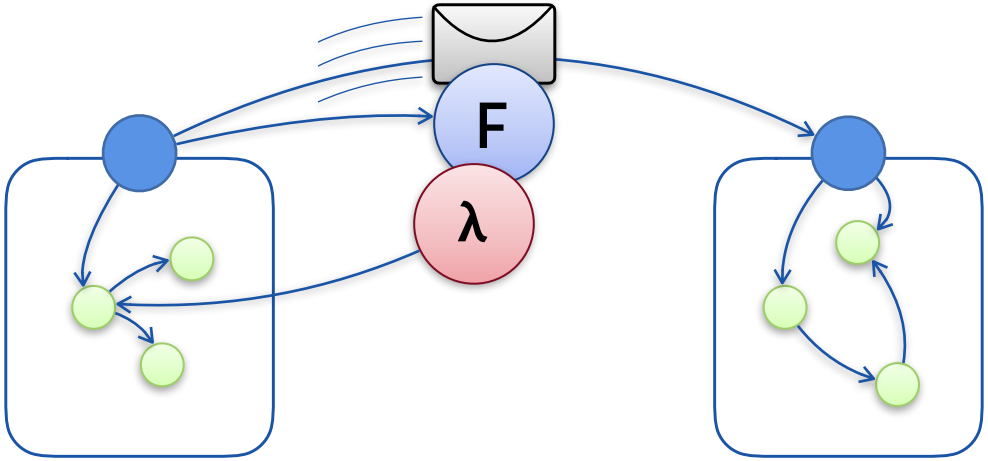
```
    val result = get fut
    this.print(result)
  end

  def print(v : Data) : unit
    this.count += 1
    ... // Print the value
  end
  ...
end
```

```
def work_noblock() : unit
  val fut = this.other ! compute()

  fut ~~>
    fun (v : Data) => this.print(v)
end
```

Who runs this closure?

Who Runs a Closure?

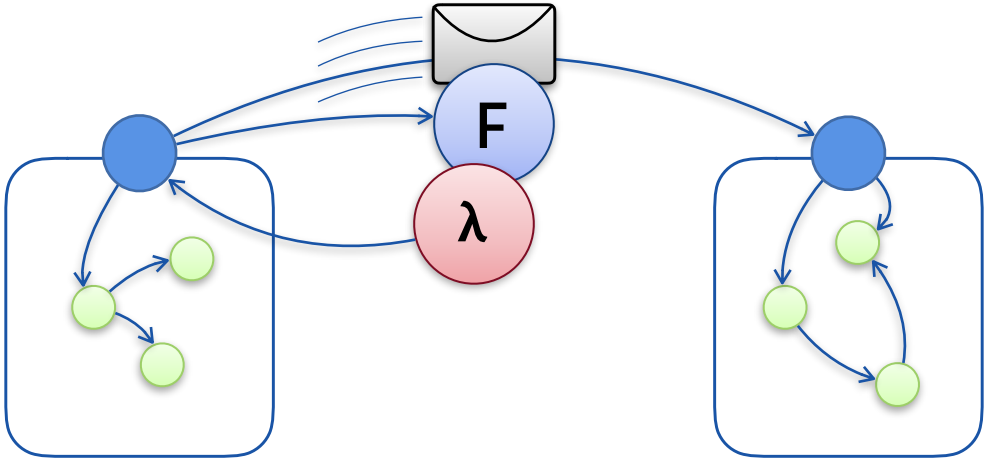
```
def work_noblock() : unit
  val fut = this.other ! compute()

  fut ~~>
    fun (v : Data) => this.print(v)
end
```



Who Runs a Closure?

```
def work_noblock() : unit
  val fut = this.other ! compute()

  fut ~~>
    fun (v : Data) => this.print(v)
end
```

```
def work_noblock2() : unit
  val fut = this.other ! compute()

  fut ~~>
    fun (v : Data) => this ! print(v)
end
```

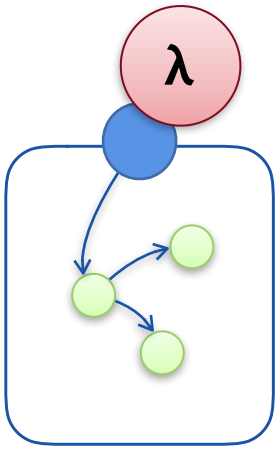


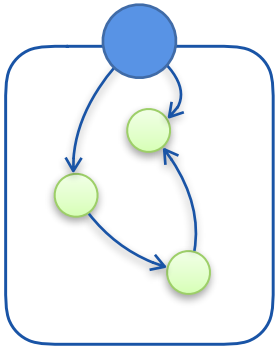
Attached and Detached Closures

- An **attached closure** is always run by its creating actor
- A **detached closure** can be run by any actor

Attached and Detached Closures

- An **attached closure** is always run by its creating actor
- A **detached closure** can be run by any actor





Attached and Detached Closures

- An **attached closure** is always run by its creating actor
- A **detached closure** can be run by any actor


```
fun (v : Data) => this.print(v)
```

Closures and Capabilities in Encore

- A closure mirrors the (non-sharable) capabilities it captures

```
fun (v : Data) => this.print(v) :        (Data -> unit)
```

local

Closures and Capabilities in Encore

- A closure mirrors the (non-sharable) capabilities it captures

```
fun (v : Data) => this.print(v) : local (Data -> unit)
```

Closures and Capabilities in Encore

- A closure mirrors the (non-sharable) capabilities it captures

```
fun (v : Data) => this.print(v) : local (Data -> unit)
```

```
fun (v : Data) => this!print(v) :        (Data -> unit)
```

active

Closures and Capabilities in Encore

- A closure mirrors the (non-sharable) capabilities it captures

```
fun (v : Data) => this.print(v) : local (Data -> unit)
```

```
fun (v : Data) => this!print(v) : active (Data -> unit)
```

Closures and Capabilities in Encore

- A closure mirrors the (non-sharable) capabilities it captures

```
fun (v : Data) => this.print(v) : local (Data -> unit)
```

```
fun (v : Data) => this!print(v) :       (Data -> unit)
```

Labeling Closures as Attached/Detached

```
def work_noblock() : unit
  val fut = this.other ! compute()

  fut ~~>
    fun (v : Data) => this.print(v)
end
```

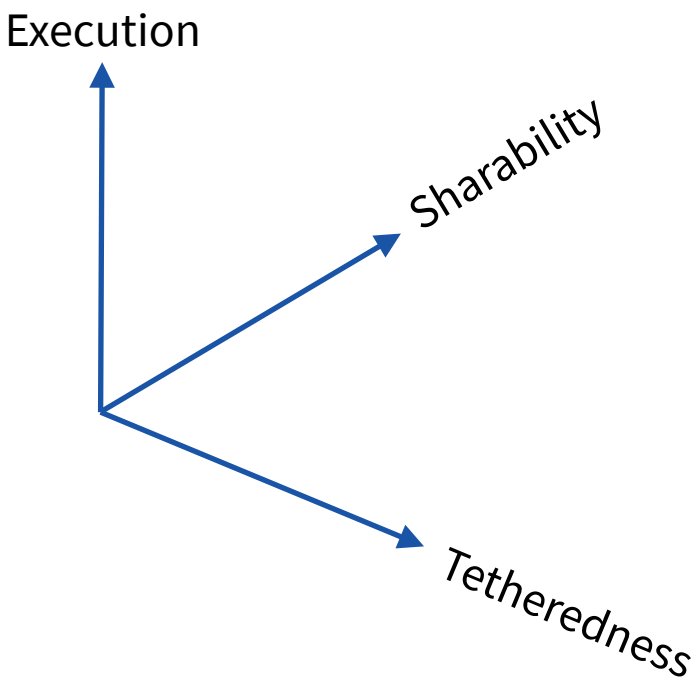
Captures local state: must be attached!

```
def work_noblock2() : unit
  val fut = this.other ! compute()

  fut ~~>
    fun (v : Data) => this ! print(v)
end
```

Only captures safe state: can be detached!

Categorising Closures

- Tetheredness ∈ {attached, detached}
- Execution ∈ {synchronous, asynchronous}
- Sharability ∈ {sharable, unsharable}



Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

```
fun (v : Data) => v.foo() + 1
```

```
fun (v : Data) => this ! print(v)
```

Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

```
fun (v : Data) => this.print(v)
```

Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe "normal" closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

```
fut ~~>
  fun (v : Data) => this.print(v)
```

Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

```
async (x.foo())
```

Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

Categorising Closures

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

Related Work (closures)

Scala/Akka	All closures detached, synchronous and sharable (unsafe)
Pony	Synchronous, detached/sharable or attached/unsharable
AmbientTalk	All closures attached, far references are asynchronous
ProActive	Attached, synchronous and sharable (deep copy)
Erlang	No mutable state
ABS	No closures (functions passed by name)

Open Questions

- Sharing attached closures

```
def run(fn : int -> int) : int
  fn(42)
end
```

- Deadlocking on attached closures

```
def deadlock(a : Actor) : unit
  var fut = a ! msg() ~~> fun(v) => ...
  var value = get fut
end
```

- Reasoning about timing and scheduling

Open Questions

- Sharing attached closures

```
def run(fn : int -> int) : int
  fn(42)
end
```

active

- Deadlocking on attached closures

```
def deadlock(a : Actor) : unit
  var fut = a ! msg() ~~> fun(v) => ...
  var value = get fut
end
```

- Reasoning about timing and scheduling

Open Questions

```
def nondeterministic(a : Actor) : unit
  val oldCount = this.count
  var fut = a ! msg()

  fut ~~>
    fun (v : Data) => this.count += 1

  if oldCount == this.count then
    ...
  end
end
```

- Reasoning about timing and scheduling

Conclusion

- Closures capturing state can be made to play nicely with actors
  - Attached closures must be run by their creating actor
  - Detached closures can be run by anyone
  - Some closures must be run asynchronously
- Encore's existing type system can express both kinds of closures
- More work needed to reason about runtime behaviour

Attached and Detached Closures in Actors

Thank you!

Attached and Detached Closures in Actors

Tetheredness	Execution	Sharability	Comment
Attached	Synchronous	Sharable	Explicitly pass back closure to owner
Attached	Synchronous	Unsharable	Current Encore implementation
Attached	Asynchronous	Sharable	Encore, when chaining
Attached	Asynchronous	Unsharable	Delaying operations
Detached	Synchronous	Sharable	Safe ”normal” closures in Encore
Detached	Synchronous	Unsharable	Not useful?
Detached	Asynchronous	Sharable	Task paralellism
Detached	Asynchronous	Unsharable	Not useful?

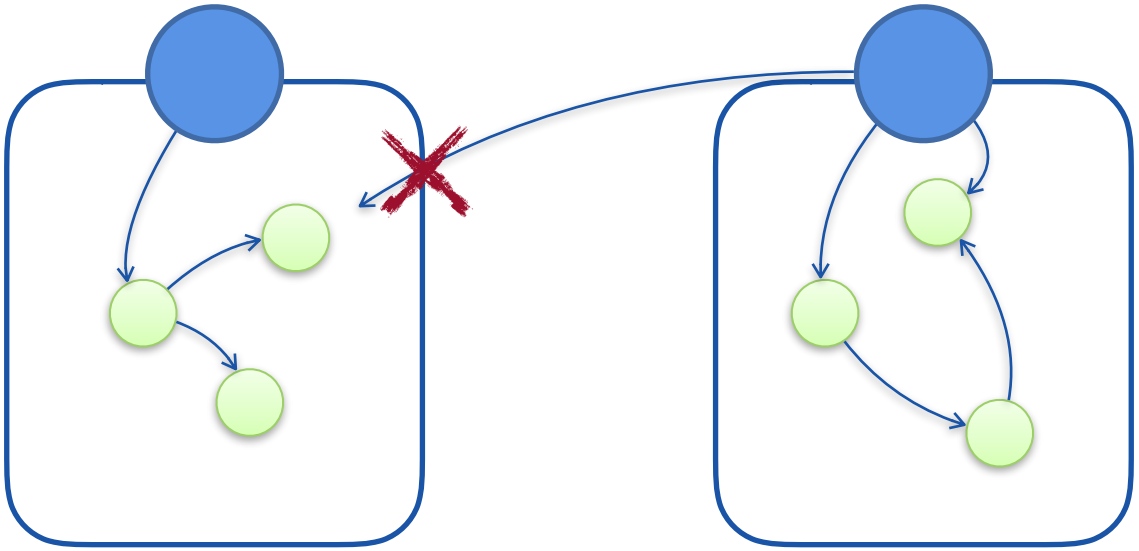
Capturing Linear Capabilities

```
var x = new LinearThing()
var f = fun () => x
var x1 = f()
var x2 = f()
```

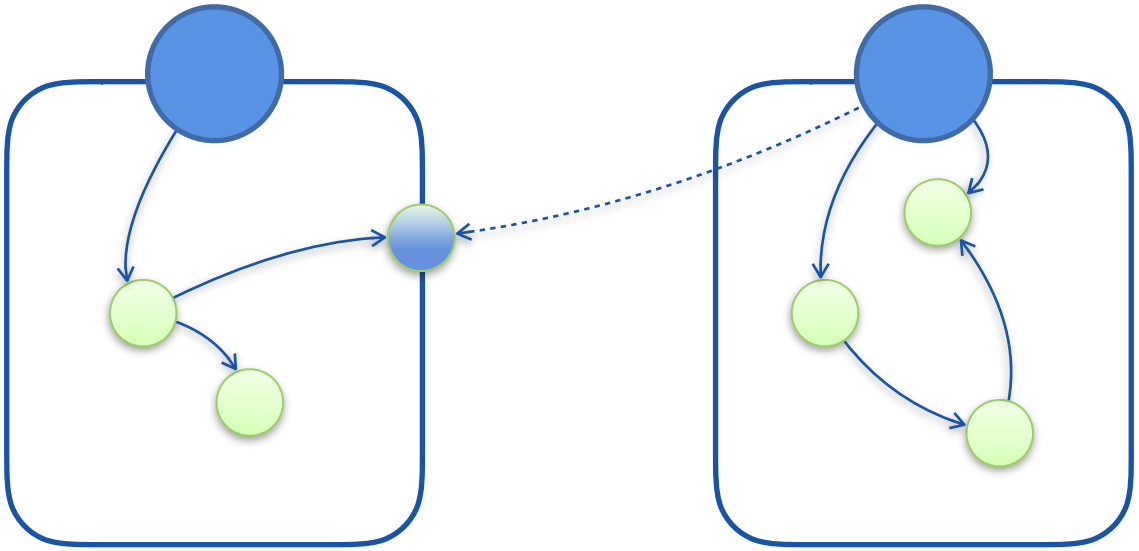
```
var x = new LinearThing()
var f = fun () => x.foo()
async f()
async f()
```

```
var x = new LinearThing()
var a = new Actor()
var f = fun () => a ! send(x)
f()
f()
```

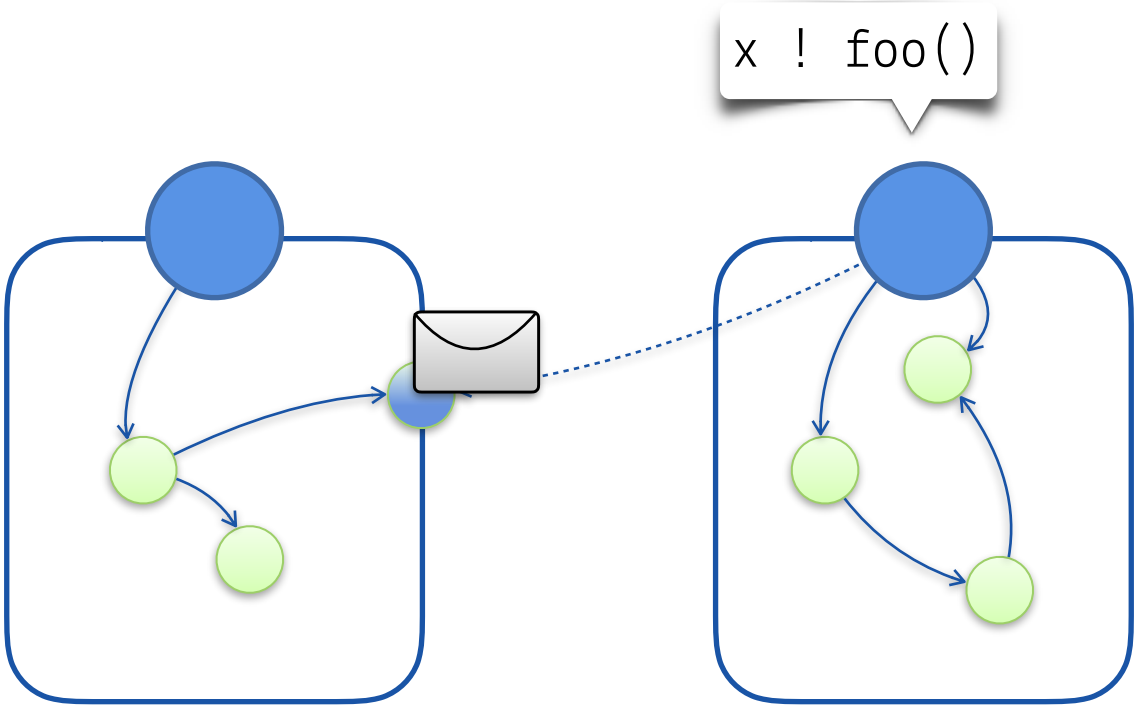
Bestowed References (Far References)



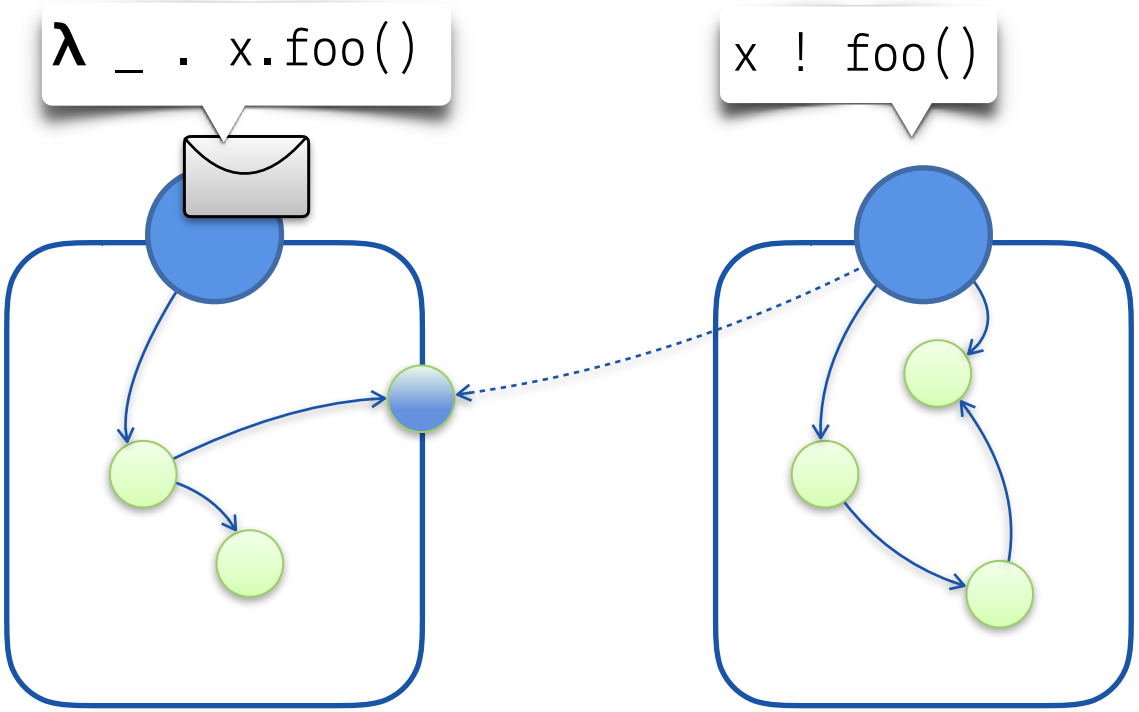
Bestowed References (Far References)



Bestowed References (Far References)



Bestowed References (Far References)



Await and Continuations

```
def foo(a : Actor) : unit
  var fut = a ! compute()
  fut ~~>
    fun (result : Data) => this.print(result)
end
```

```
def foo(a : Actor) : unit
  var fut = a ! compute()
  var result = await fut
  this.print(result)
end
```