Attached and Detached Closures in Actors

Elias Castegren*
KTH Royal Institute of Technology
Sweden
eliasca@kth.se

Dave Clarke
Information Technology
Uppsala University
Sweden
dave.clarke@it.uu.se

Kiko Fernandez-Reyes
Information Technology
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Tobias Wrigstad
Information Technology
Uppsala University
Sweden
tobias.wrigstad@it.uu.se

Albert Mingkun Yang
Information Technology
Uppsala University
Sweden
albert.yang@it.uu.se

Abstract

Expressive actor models combine aspects of functional programming into the pure actor model enriched with futures. Such functional features include first-class closures which can be passed between actors and chained on futures. Combined with mutable objects, this opens the door to race conditions. In some situations, closures may not be evaluated by the actor that created them yet may access fields or objects owned by that actor. In other situations, closures may be safely fired off to run as a separate task.

This paper discusses the problem of who can safely evaluate a closure to avoid race conditions, and presents the current solution to the problem adopted by the Encore language. The solution integrates with Encore's capability type system, which influences whether a closure is *attached* and must be evaluated by the creating actor, or whether it can be *detached* and evaluated independently of its creator.

Encore's current solution to this problem is not final or optimal. We conclude by discussing a number of open problems related to dealing with closures in the actor model.

CCS Concepts • **Computing methodologies → Parallel programming languages**; **Concurrent programming languages**; *Concurrent computing methodologies*;

Keywords closures, parallel programming, concurrent programming, type systems

*Work done while at Uppsala University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AGERE '18, November 5, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6066-1/18/11…$15.00
https://doi.org/10.1145/3281366.3281371

ACM Reference Format:
Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Tobias Wrigstad, and Albert Mingkun Yang. 2018. Attached and Detached Closures in Actors. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3281366.3281371

1 Introduction

The actor model has been around for decades [1, 3, 4], but has recently seen renewed interest as it offers a solid programming model for both distributed and concurrent systems. The feature of actors responsible for its renewed popularity is its concurrency model, which is based on isolating mutable state within actors and commmunicating between actors using message passing. Actors thus offer encapsulation and data race freedom, while remaining close enough in spirit to the object-oriented paradigm. The actor model is a language abstraction while threads are an implementation detail.

Modern actor-based programming languages, like AmbientTalk [10], Scala (via Akka) [14], Pony [11], Encore [5], ABS [15], Creol [16], Proactive [6], and deviate from the pure actor model by offering language features that originated in other paradigms, including sharing of immutable objects, algebraic data types, futures and/or promises, and higher-order and anonymous functions (*closures*).

The Encore [5] programming language combines many of these features in a single language and offers a capability system for programmers to express their intentions with respect to how data is transferred, shared and interacted with. The capability system ensures statically that actor's data are manipulated in a data race-free fashion. Because closures capture variables from the lexical scope in which they are defined, they are particularly subtle to deal with in a context where concurrent operations are frequent. Closures can be passed between actors or chained on futures, meaning that they will be run when the future is eventually fulfilled. In both cases, the closure's creator and the eventual evaluator(s) of the closure may not be the same actor, which introduces

the opportunity for data races through variables and objects captured from the original scope.

Encore addresses this problem by distinguishing two kinds of closures: *attached* closures, which must be run by the actor who created them, and *detached* closures, which can be run by any actor. Encore's capability type system guarantees that a closure which modifies state owned by an actor will always be run by the actor itself (as an attached closure), and that running a detached closure will never lead to data races.

This paper focuses on Encore (Section 2), though the ideas are applicable to other actor languages. Problems associated with closures in an actor-based programming languages are first reviewed (Section 3), before giving Encore's approach to these problems in two parts: changes to the runtime (Section 4) and support from Encore's capability-based type system (Section 5). Some remaning open problems are discussed (Sections 6), and the paper concludes by discussing how closures are treated in similar languages (Section 7).

2 Encore Primer

This section gives an overview of the Encore actor model and its capability-based type system.

2.1 Overview

Encore [5] is an actor language in the active object tradition [17, 22, 23]. Programs are made up of active objects (from now on called actors) and passive objects (typical objects in object-oriented programming, but without synchronisation). Encore actors can be dynamically created, have a logical thread of control and communicate with other actors through asynchronous message sends. Asynchronous message sends use the `!` notation (*e.g.,* Listing 1, Line 14) and synchronous calls use the `.` notation (*e.g.,* Listing 1, Line 15). Messages to an actor are enqueued in its private mailbox and executed in FIFO order by the actor's own thread by executing a corresponding method on the actor. An actor's fields can only be read and written by the actor itself. Scheduling is cooperative, and there is no notion of preemption.

The results of asynchronous messages sends are communicated using futures. Futures are fulfilled at most once, always implicitly, either by returning from the method executed as the result of recieving an asynchronous message, or by delegating the fulfillment to another actor through an asynchronous message send [12]. Listing 1 shows an example of an actor declaration (Line 1) with three methods: `block()`, `noblock()`, and `print()` (Lines 5, 13, and 18). The `block()` method illustrates the typical use of futures. The actor sends the (asynchronous) message `compute` to the `other_actor` and immediately gets a *future*, `fut`, from `other_actor`. This future will eventually be fulfilled by `other_actor` with the return value from its `compute()` method.

```
1  active class Actor
2    var count : int
3    var other_actor : Actor
4
5    def block() : unit
6      -- bang = async msg
7      var fut = this.other_actor ! compute()
8
9      var value = get fut -- blocks!
10     this.print(value)
11   end
12
13   def noblock() : unit
14     this.other_actor ! compute() ~~>
15       fun (v) => this.print(v) -- dot = sync call
16   end
17
18   def print(v : Obj) : unit
19     this.count += 1
20     ... -- some action (printing)
21   end
22   ...
23 end
```

Listing 1. Running example.

Encore also has a notion of tasks, denoted **async** *e* **end**. Tasks are asynchronous one-shot actors running the expression *e*. Spawning a task immediately returns a future.

Futures' main operations are **get** and future chaining.

The **get** operation blocks the current actor until the future is fulfilled (receives its value) and then returns that value. If **get** is immediately applied to the future to get the result of the message send (Line 9 in `block()`), then message sends can be structured like method calls and returns in object-oriented programming. That is, the operation **get** `fut` blocks until the value is computed, after which the local method `print()` is called, synchronously, with this value as parameter.

The `noblock()` method gives an example of future chaining (the `~~>` operator, on Line 14) which attaches a callback to the future return from `compute()` to be evaluted on fulfilment. The callback is the closure in Line 15. Any number of callbacks can be registered on a future, and each callback registration returns a new future, a handle to *its* future result. When a future is fulfilled by a value `v`, `v` is passed to each of the registered callbacks. In `noblock()`, this means evaluating the closure which calls **this**`.print(v)`. In this particular example, the (future) result of the printing operation is ignored, but had it not, it would have been the `unit` value returned from `print()`.

Effectively, the implementation of `noblock()` makes the printing method (Line 15) a continuation of `compute` without

requiring the `compute` method to be aware of this fact. Thus, future chaining allows the construction of pipelines of asynchronous processes in a program.

2.2 Encore's Capability-Based Type System

Encore associates each reference with a capability that defines which operations are available on the referenced object (*e.g.,* which methods are available to call) as well as the reference itself (*e.g.,* may this reference be copied). A capability can be thought of as a slice of an object. Capabilities are tracked by the type system, which prevents the creation of a capability which could cause a data race. If two actors have references to the same object, the type system ensures that the capabilities of these references do not allow concurrent write/write or read/write accesses to (the same parts of) the underlying object [8].

The operations available on an object through a reference is defined by the type of the capability, just as in any object-oriented language. Additionally, each capability is annotated with a *mode* which defines how data race freedom is achieved; either by restricting how the capability may be copied, or by constraining the type of the capability. Encore currently supports five different capability modes:

`linear` — a capability that may not be copied. A **`linear`** capability is safe to use because no alias to the same object (or part of object) may exist.

`local` — a capability that may not be passed between actors. A **`local`** capability is exclusively owned by the owner that created it.

`subord` — a *subordinate* capability which is strongly encapsulated within an object. A **`subord`** capability can only be accessed by going via the capability of the encapsulating object[1].

`read` — a capability that does not allow mutating operations. A **`read`** capability is always safe to pass between actors.

`active` — a capability through which all accesses are asynchronous (*i.e.,* message sends). All actor references have **`active`** capabilities.

Types are introduced via classes and traits, and the same type can be reused with different modes depending on the intended usage (with the exception of **`read`** capabilities, which can only be used when the constitutent parts do not allow mutating operations). For example, a **`local`** `List` is a list that is intended to be used locally by a single actor, whereas a **`linear`** `List` is a list that can be transferred between actors (but never shared). It is also possible to compose and decompose capabilities, and for example turn a mutable **`linear`** capability into an immutable **`read`** capability which may be shared freely. We refer to prior work for details [8].

[1]For example, subordinate capabilities inside a linear capability are safe from data-races because the linear capability cannot be shared across actors.

When defining the class of an actor, the capability of **`this`** is interesting. For example, when the actor accesses its own fields (*e.g.,* Line 19 in Listing 1), **`this`** is a **`local`** capability — this makes sense, since only the actor itself may access its own fields. On the other hand, when the actor shares itself with other actors, or sends itself messages, **`this`** is an **`active`** capability. This distinction turns out to be important when reasoning about how closures can be shared (*cf.,* Section 5).

3 Problem Space

Actor programs require careful consideration when handling closures that capture mutable state. If these stateful closures are evaluated by other actors, then their evaluation can introduce data races. This section uses Listing 1 to motivate the discussion of the problems that arise when combining actors and closures.

As explained in the previous section, `Actor` has three methods: `block()`, `noblock()`, and `print()`. The `block()` and `noblock()` methods use `print()` in a synchronous and asynchronous way, respectively.

The `print` method counts the number of printed items (Line 19) and to this end reads and writes a field in the actor.

No state is captured in the `block` method — there are no closures. The closure in the `noblock()` method is chained on the future, thus capturing the state of the `print` method.

The last scenario is subject to data-races as the thread of the *fulfiller* of the future executes the **`this`**`.count += 1` operation (Line 19). Multiple futures fulfilled by different actors could be racing on incrementing the same print counter, or racing with the thread of the actor who supposedly has solitary access to `count`.

Clearly, the use of future chaining on a closure can cause data races in the program. This problem is the same regardless of whether the closure (Line 15) is exposed to the world outside the actor using future chaining (as in Line 14) or returned from a method.

In summary, programming with closures in actor programs requires careful consideration of how to handle *closures that capture state* as it can break actor isolation.

To ensure safety and be able to leverage the expressive power of closures and the future chaining operation, we have identified the following contraints on closures.

3.1 Capturing Local State

A closure chained on a future can be thought of as the continuation of the method — what the method does with the result in the future. As such, closures should be able to capture the local state, including **`this`**, of the surrounding actor.

3.2 Sharing Closures Among Actors

It should be possible to create a closure and pass it from one actor to another, who will then evaluate it. This will facilitate

higher-order programming patterns, such as asynchronous pipelines and exception handling.

3.3 Data Race-Freedom

Closures that capture local actor state while being shared among actors are subject to race conditions, wherein two actors try to write (or read and write) the same field of the same object at the same time. Attaching a closure to a future or returning a closure should not violate the data race freedom provided by actors.

3.4 Expose Parallelism

Closures that do not capture local state in a way that could lead to data races can be run independently of the actor that created them. Such closures can be used as a source of parallelism For example, Encore's parallel combinators [13] uses such closures to express asynchronous, speculative parallel pipelines. Ideally, in order to reason about parallelism, the programmer needs to know whether a closure can be run independently.

3.5 Deadlock Avoidance

As performing a **get** on a future blocks the actor in which the **get** is performed, a deadlock can result if the fulfillment of the future depends on a computation performed by the current actor, not yet performed. Deadlock avoidance should be as simple as possible.

3.6 Reasoning about Timing and Scheduling

Chaining closures on futures results in computations whose timing are not easy to reason about, because the running of the closure depends on whichever computation is fulfilling the future and when the scheduler schedules that computation and the chained computation. This problem is exacerbated when chaining multiple closures to form pipelines.

4 The Encore Solution: Attached and Detached Closures

Encore has made some progress towards addressing the problems described in the previous section, though the current solution is not complete and needs to be refined. We present the solution in two parts. The first part, treated in this section, deals with the changes to the runtime behaviour of Encore, and the second part, treated in Section 5, describes how Encore's capability system is employed to ensure data race freedom.

Encore's runtime handle *attached* and *detached* closures separately. Attached closures outside their creating actor are evaluated by being passed to their creating actor and executed there, thus serialising its execution with any other activity in the creator. Detached closures may be passed around freely and additionally be evaluated by any thread.

Since a detached closure can be evaluated by any thread of control, it can always legally be turned into an attached closure, at the risk of increasing latency through the addition of an additional pipeline stage in the computation.

Consider the following snippet from the running example:

```
this.other_actor ! compute() ~~>
  fun (v) => this . print(v) -- sync call
```

Because the closure synchronously calls `print()` of the surrounding actor, which modifies its state, it *must be attached*.

Consider the alternative implementation which uses an asynchronous message send, which places a message in the surrounding actor's queue rather than performing modifications in the current thread:

```
this.other_actor ! compute() ~~>
  fun (v) => this ! print(v) -- async call
```

This implementation allows the closure to be *detached*, as it neither access the fields of the surrounding actor nor captures its state. This solution avoids data races as `count` is only accessed using the actor's own thread.

4.1 Who Runs the Runnables?

In principle, detached closures can be run by any thread. As detached closures are typically attached to futures as callbacks, it is at the time the future is fulfilled that the closure becomes runnable, and the runtime needs to decide who runs the closure based on whether it is attached or detached.

If the closure is attached, the thread that fulfils the future sends the attached closure, and the value it will be applied to, back to the closure's owner.

If the closure is detached, the thread that fulfils the future will evalute the closure. This is a pragmatic design choice in Encore, made to reduce overall latency. When an actor fulfils a future, it also executes all callbacks on that future. By similar reasoning, registering a callback on an already fulfilled future immediately runs the callback in the thread of the registering actor.

There is plenty of scope to allow different actors to run the closures, or to even run them independently of actors by passing them to a task.

4.2 Supporting Sharing of Attached Closures

Chaining of closures is permitted, regardless of the closure's attached/detached status. It would be possible to allow attached closures to be passed around freely — but not evaluated except in the context of their creating actor. This, however, causes typing to become problematic. If an attached closure gets to its owning actor, the actor evaluates the closure. If an attached closure gets to a non-owning actor, this actor would send a message back to the owning actor, to do its evaluation, but its return type would be a future. A simple solution is to use the **get** operation on this future.

This violates a design principle of Encore that opts for a clear demarkation of asynchronous operations.

4.3 Summary

The key challenge for programming with closures in actor programs is that a closure may capture variables and fields from the context in which it is created. Encore distinguishes closures whose captured variables or fields give it the capability to mutate the state of its creating actor, and categorises these as *attached*. All other closures are *detached*.

Closures of either category can be created and chained on futures, but only detached closures can be passed in messages between actors. When executing closures, attached closures must always be executed by their creating actors, to ensure that the execution is interleaved — never overlapping — with that actor's other operations. Detached closures can be executed freely by any actor at any time.

The next section explores how Encore's capability system preserves data race freedom in the presence of attached and detached closures.

5 The Encore Solution: Closures and Capabilities

This section explains how Encore's capability system deals with closures, and how it can be used to reason about attached and detached closures. The key to preventing data races through variables captured by a closure is to look at the capabilities of these variables.

Recall that the only capabilities that allow unsynchronised mutation are **linear**, **local**, and **subord** capabilities. However, if a closure could capture such a capability and then pass it to another actor, running that closure would not be safe. On the other hand, capturing a **read** or **active** capability (or some variable of a primitive type) is always safe: running such a closure cannot cause data races.

Encore tracks what a closure captures by also annotating closure types with modes when needed: a closure that captures a **local** or a **subord** capability will get that same mode. This means that a closure that captures a **local** capability will itself be **local**, and can therefore not be passed to a different actor (and similarly for subordinate capabilities). A closure that does not capture **local** or **subord** capabilities remains mode-less.

Without the concept of attached closures, it would not be safe to do future chaining with a closure which captures **local** or **subord** capabilities, since the closure may be run by a different actor. On the other hand, with the attached/-detached distinction in the language, the type of the closure unambigously shows if a closure needs to be attached for safety (*i.e.,* it captures **local** or **subord** state) or not. For example, the closure on Line 15, Listing 1, captures the **local** capability **this**, and thus gets the type "**local** `(t -> unit)`" (where `t` is the type of `v`). This means that the closure must be attached so that the creating actor is the one that will run the closure. On the other hand, if the body of the closure is changed to "**this** `! print(v)`" so that **this** is used as an **active** capability, the type of the closure is just "`t -> unit`", meaning that it can safely be used as a detached closure and be run by any actor.

Closures that capture **linear** capabilities are more complicated. These closures must be treated linearly, *i.e.,* never have more than one reference to them. If calling the closure also consumes the linear capability (*e.g.,* by returning it), care must be taken to ensure that the closure cannot be called more than once. For simplicity, Encore does not currently support capturing **linear** capabilities.

In summary, a closure that captures **local** or **subord** capabilities must be attached, while a closure that does not can be detached. Since this information is already present in the type of the closure, correctly identifying attached closures can be done automatically. Notably, no additional annotations are required for this; the code in Listing 1 can remain unchanged, and the closure on Line 15 would still be correctly identified as attached.

6 Open Problems

Some open problems remain. We discuss these in the current section and describe some possible solutions.

6.1 Sharing Attached Closures

As outlined in Section 4.2, it would be possible to pass around attached closures on their own, without chaining them to futures. This has the downside that it is harder to reason about the latency of calling a closure, and requires that all closure calls are preceded by an ownership check. By tracking which closures are attached,we could restrict this downside to only these closures.

One way to track this would be to annotate closures with the **active** mode, just as closures which capture local state are given the **local** mode (*cf.* Section 5). A closure of type "**active** `(t1 -> t2)`" would then be known to be attached and asynchronous, meaning that the programmer knows that calling it may require synchronisation with the creator of the closure. Additionally, it would be possible to convert an unsharable closure of type "**local** `(t1 -> t2)`", which captures local state, to a sharable attached closure of type "**active** `(t1 -> t2)`", analogously to how the capability of **this** can be seen as both **local** and **active** depending on usage (*cf.* Section 2.2).

6.2 Deadlocking on Attached Closures

Performing a **get** on a future can possibly lead to a deadlock if the fulfiller of the future is the current actor or depends on the current actor assisting in the computation — recall that **get** blocks the current actor. The problem is also present with closures chained on futures, and it is particularly subtle.

If the closure must be run by the current actor, doing a **get** on its future will result in deadlock. Deadlock detection is either statically undecidable or dynamically expensive. Thus the programmer needs to be able to reason about which actor will evaluate a closure to avoid directly deadlocking.

Lines 6–7 of Listing 1 shows an actor sending a message to an `other_actor` and blocking on the future result. If `other_actor` happens to be the current actor, this will invariably deadlock, as Line 7 prevents the actor from picking up the message and computing the result it is blocking on. This is a well-known artefact in actor systems, and an indication of a high-level design problem if it happens in practice.

Attached and detached closures could be argued to exacerbate this problem in combination with future chaining. Consider the following lines of Encore:

```
1 var fut = a ! msg() ~~> fun (v) => ...
2 var value = get fut
```

On Line 1, we chain a closure on the future resulting from the `msg()` message send. If the closure on Line 1 is attached, it will be sent back to the current actor to be computed, but will never be able to pick this message up because it is blocking on Line 2 above.

Static deadlock detection in Encore is complicated by its support for delegating the task of fulfilling a future to another actor using the `forward` construct [12] which essentially allows a single future to move across an asynchronous pipeline, to be fulfilled at its end.

6.3 Reasoning about Timing and Scheduling

As long as all closures are executing synchronously in a pipeline *e.g.,* function composition, timing is easy to predict. However, when building pipelines of asynchronous operations, a move forward to the next pipeline stage will depend on how long it takes for the next actor in the pipeline getting to the head of its scheduler queue.

Encore's scheduler is not currently able to observe topologies and *e.g.,* organise objects in a pipeline in a suitable fashion on the hardware. Encore's support for off-loading computation from an actor via detached closures is relatively weak. A detached closure will be executed by the actor that fulfils the future it is chained on. In many cases, it is reasonable to want to attach a closure even if it could technically be detached, *e.g.,* in a distributed system where the cost of safely moving captured data to a remote node would be much higher than moving computation to the data.

The design decision in Encore to make the fulfiller of a future run the detached closures chained on that future has the possible side-effect of moving work out of the current actor. Furthermore, many different closures may be chained on that future, and a fulfiller needs to evaluate all of them before the method returns. This makes reasoning about timing in such situations difficult.

On the other hand, if the future is already fulfilled, then the actor doing the chaining runs the closure. Thus, the timing of future chaining depends heavily on whether or not the future is fulfilled.

7 Related Work and Discussion

Several existing actor languages support closures, as summarised in Table 1.

Closures in Pony must declare which variables they capture (similar to Spores [18]). If a closure captures a reference that it uses for mutation, *i.e.,* a `ref` capability, the closure itself must be declared as `ref`, meaning it cannot be passed between actors. Pony does not support chaining of attached closures on promises (analogous to futures) and having them passed back to the closure's owner for evaluation. Pony's promises are implemented as actors, and thus have a logical thread of control. This is an interesting design choice with respect to executing closures chained on futures (*cf.,* Section 4.1).

The AmbientTalk language only supports attached closures, meaning a closure is always passed back to its creating actor for evaluation. This is in line with its concurrency model which is based on E's [19], which partitions an application into different "vats" and makes method calls across vat-boundaries asynchronous. Encore has a similar feature, `bestow`, that allows references across actors' heaps, but operations on those references are made asynchronous by the compiler [7]. In some ways, `bestow` can be thought of as implicitly wrapping operations in attached closures, which are sent back to the actor whose state it captures on execution.

Scala/Akka does not directly support the notion of attached or detached closures, although both concepts can be implemented manually. This is in line with the lack of support for actor isolation of Scala/Akka, which is on the level of Java, *i.e.,* none. It is also notable that it is not uncommon for actors in Scala to break the actor concurrency model [20].

The ABS language does not support closures. ABS has a functional sublanguage which does not include objects — the only objects in ABS are the concurrent actor objects. ABS supports passing functions by name, but because functions cannot capture mutable variables (including **this**), ABS avoids the problem (but is also less expressive in this respect). ABS does offer some type system support that is close to

Table 1. Handling of closures in actor languages.

Language	Strategy
Pony [9, 11]	All closures are detached
AmbientTalk [10]	All closures are attached
Scala/Akka [14]	Unsafe sharing of closures allowed
Erlang [2]	No mutable state
ABS [15]	No closure support
Proactive [6]	Closures are deep-copied

Table 2. Instantiating the design space combinations.

no.	**Tetheredness**	**Execution**	**Sharability**	**Comment**
1	attached	synchronous	sharable	Requires explicit passing back to creator to evaluate
2	attached	synchronous	unsharable	Current Encore implementation
3	attached	asynchronous	sharable	Supported by AmbientTalk, and Encore with future chaining
4	attached	asynchronous	unsharable	Delaying operations
5	detached	synchronous	sharable	Supported by Encore and Pony
6	detached	synchronous	unsharable	Probably not useful
7	detached	asynchronous	sharable	Task parallelism
8	detached	asynchronous	unsharable	Probably not useful

what is required to track attached and detached closures. Location Types [21], a pluggable type system for the ABS programming language, track whether a reference is `near`, on the same core, concurrent-object group, machine, ..., `far`, on another core, concurrent-object group, machine, ..., or `somewhere`, meaning location unknown. Near objects can be interacted with using synchronous method call, whereas calling far objects must use asynchronous message sends.

The Erlang language mimics the implementation of stateful actors using tail-recursive processes that pass around the current values of their state, rather than allowing mutable state. Capturing mutable state in a closure is therefore not possible.

Finally, the Proactive language enforces the deep copying of passive objects when passing objects between actors. Modelling closures is possible using objects with a single method, but such closures will always be effectively detached due to the deep copying. Implementing attached closures manually is therefore not possible.

Taking a step back, there are essentially three dimensions to the design space of closures in actors:

1. Tetheredness ∈ { attached, detached }
2. Execution ∈ { synchronous, asynchronous }
3. Sharability ∈ { sharable, unsharable }

Tetheredness concerns *who can apply the closure*. An attached closure can only be executed by its creating actor, whereas a detached closure can be executed by any actor (thread of control). Execution concerns *whether the closure is executed immediately by (synchronously) the current thread of control, or in an asynchronous operation in some other actor.* Sharability concerns *whether the closure can be passed around (*sharable*) or not (*unsharable*).*

Table 2 overviews all combinations of the properties, and some programming languages that support the combinations.

Combination 1, { attached, synchronous, sharable }, is interesting. It describes a design where attached closures can be shared freely, but not applied outside of their creating actor. This implementation requires the ability to inquire the owner of a closure and compare with the current actor, or alternatively a static type system which is able to capture ownership precisely. Combination 2 avoids the latter by forbidding sharing. Combination 3 makes attached closures safe to share by passing them back to their creating actor, making the application asynchronous. This is the approach taken in AmbientTalk [10] and future chaining in Encore.

The most permissive is Combination 5: { detached, synchronous, sharable }. Such closures can be passed around freely and anyone can apply them using their own thread of control. Two examples of such closures in Encore would be:

```
fun (v:int) => v + 1
fun (v:int) => this ! action(v)
```

The first is side-effect free (including read effects on captured state). The latter only reads an **`active`** capability, which is safe.

The Encore **`async`** construct is commonly used to create closures in the style of Combination 7: detached, asynchronous. These are exactly the properties needed for parallelism inside an actor's method:

```
val fut = async fib(40) end -- { detached, async }
```

The closure above calculates *fib*(40) in a separate logical thread. In Encore, such a closure must not capture the creating actor's state or parallelism is lost. Removing the shareability destroys parallelism, but lets the actor put off running tasks to the future, interleaving them with other operations, thus safely allowing access to state (Combination 4).

Combinations 6 and 8 are probably not useful given the paradoxical combination of detachment (can be run by anyone) and unsharable (cannot be shared with anyone).

8 Concluding Remarks

This paper introduces the dichotomy of attached and detached closures. (Table 2 deepens this analysis a little further.) Attached closures must effectively be run by their creating actor when executed, whereas detached closures must not. Thus, executing the former outside of the creating actor must run (logically in the very least) as an asynchronous activity.

Attached and detached closures exist before in different languages, but not as explicitly identified concepts, and not in a single language.

The Encore programming language supports both attached and detached closures, although attached closures can only be passed between actors implicitly via future chaining. We outline a way to extend this support using existing Encore concepts at the type-level.

Acknowledgments

This work was partially supported by the Swedish Research Council through the UPMARC Programming for Multicore Architecture Research Centre and the *SCADA* project.

References

[1] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT Press.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1993. *Concurrent programming in Erlang.* Prentice Hall.

[3] Henry G. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes. *SIGART Newsletter* 64 (1977), 55–59. https://doi.org/10.1145/872736.806932

[4] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5, Article 76 (Oct. 2017), 39 pages. https://doi.org/10.1145/3122848

[5] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming, SFM 2015*, M Bernardo and E B Johnsen (Eds.). Lecture Notes in Computer Science, Vol. 9104. Springer, 1–56.

[6] Denis Caromel, Christian Delbe, Alexandre Di Costanzo, and Mario Leyton. 2006. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology* 12 (2006), issue 1. https://hal.archives-ouvertes.fr/hal-00125034

[7] Elias Castegren, Joel Wallin, and Tobias Wrigstad. 2018. Bestow and Atomic: Concurrent Programming Using Isolation, Delegation and Grouping. *Journal of Logical and Algebraic Methods in Programming* 100 (2018), 130–151. https://doi.org/10.1016/j.jlamp.2018.06.007

[8] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[9] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816

[10] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science)*, Dave Thomas (Ed.), Vol. 4067. Springer, 230–254. https://doi.org/10.1007/11785477_16

[11] Pony Developers. 2018. Pony – High-Performance Actor Programming. http://www.ponylang.org. (2018).

[12] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a Promising Future. In *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings (Lecture Notes in Computer Science)*, Giovanna Di Marzo Serugendo and Michele Loreti (Eds.), Vol. 10852. Springer, 162–180. https://doi.org/10.1007/978-3-319-92408-3_7

[13] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. 2016. ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science)*, Alberto Lluch-Lafuente and José Proença (Eds.), Vol. 9686. Springer, 101–120. https://doi.org/10.1007/978-3-319-39519-7_7

[14] Lightbend Inc. 2018. AKKA – Scala actor library. http://akka.io/. (2018).

[15] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2010. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers (Lecture Notes in Computer Science)*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.), Vol. 6957. Springer, 142–164. https://doi.org/10.1007/978-3-642-25271-6_8

[16] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. 2006. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.* 365, 1-2 (2006), 23–66. https://doi.org/10.1016/j.tcs.2006.07.031

[17] R. Greg Lavender and Douglas C. Schmidt. 1996. Pattern Languages of Program Design 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, 483–499. http://dl.acm.org/citation.cfm?id=231958.232967

[18] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 8586. Springer, 308–333.

[19] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[20] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. 2013. Why do scala developers mix the actor model with other concurrency models?. In *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 7920. Springer, 302–326.

[21] Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. 2013. Location Types for Safe Programming with Near and Far References. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 471–500. https://doi.org/10.1007/978-3-642-36946-9_16

[22] Yasuhiko Yokote and Mario Tokoro. 1987. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, Akinori Yonezawa and Mario Tokoro (Eds.). MIT Press, 129–158.

[23] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. 1986. Object-Oriented Concurrent Programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings.*, Norman K. Meyrowitz (Ed.). ACM, 258–268. https://doi.org/10.1145/28697.28722