UPPSALA
UNIVERSITET

IT 17 039
Examensarbete 15 hp
Juli 2017

Implementing Safe Sharing Features for Encore

Joel Wallin

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Implementing Safe Sharing Features for Encore

Joel Wallin

Actor isolation is an important property in parallel and concurrent programs that utilize the actor model. However, when expressing certain patterns isolation can sometimes be too strong and forces complexity on actors. To address this problem, two new language constructs have been introduced: Bestow allows an actor to delegate a part of its interface to its internal objects; Atomic enables grouping of messages which requires them to be handled in sequence. This thesis discusses several valid designs which are compared in the context of an object-oriented and actor based language called Encore. Bestow and atomic have proven to simplify several patterns and to minimize the complexity of actors by decoupling classes and allowing for granular interfaces. This additional abstraction comes with some overhead at run-time which there is plenty of room for reducing in future work. Equipped with these new tools a programmer can simplify complex concurrency patterns, allowing them to focus on the main task at hand.

Handledare: Elias Castegren
Ämnesgranskare: Tobias Wrigstad
Examinator: Olle Gällmo
IT 17 039
Tryckt av: Reprocentralen ITC

Acknowledgments

I would like to thank the people involved in the Upscale project and Encore for all their help and feedback, mainly Elias, Tobias and Albert. Special thanks to Jonas for keeping me company and being great to bounce ideas off of.

Contents

1 Introduction **1**
- 1.1 Goal and Purpose . . . 1
- 1.2 Outline . . . 2

2 Background **2**
- 2.1 The Actor Model . . . 2
- 2.2 Encore . . . 3
- 2.3 Sharing Data in Concurrent Programs . . . 5

3 Problem Description **6**
- 3.1 Motivation: Breaking Isolation . . . 6

4 Design **8**
- 4.1 Bestow . . . 8
  - 4.1.1 Thread Safe Bestowed Objects . . . 9
  - 4.1.2 Synchronization . . . 10
- 4.2 Atomic . . . 12
  - 4.2.1 Semantics . . . 12
  - 4.2.2 Enforcing Non-Interleaving . . . 13
  - 4.2.3 Message Sends . . . 15

5 Implementation **16**
- 5.1 Bestow . . . 17
  - 5.1.1 Typechecking . . . 18
  - 5.1.2 Optimizer . . . 18
  - 5.1.3 Garbage Collection of Bestowed Objects . . . 19
- 5.2 Atomic . . . 20
  - 5.2.1 Typechecking . . . 20
  - 5.2.2 Optimizer . . . 21
  - 5.2.3 Code Generation . . . 21
  - 5.2.4 Run-Time . . . 23
  - 5.2.5 Scheduling the Atomic Target . . . 23

6 Evaluation **24**
- 6.1 Expressive Power . . . 25
  - 6.1.1 Sleeping Barber . . . 25
  - 6.1.2 Iterator . . . 26
- 6.2 Performance . . . 28
  - 6.2.1 Bestow Overhead . . . 28
  - 6.2.2 Atomic Overhead . . . 30
  - 6.2.3 Concurrent Sorted Linked-List . . . 32
- 6.3 Discussion . . . 33

7 Related Work **34**

8 Conclusion **35**
- 8.1 Future Work . . . 35

Bibliography **38**

1 Introduction

The ever ongoing race for more computing power has lead to the inception and exploration of parallel programming [1]. The inevitable problem that parallel systems have to solve is how to synchronize when multiple entities want to access the same data. Languages in use today were often originally designed to be sequential, without parallel execution in mind [1, 2], thus requiring the programmer to identify where synchronization is needed. This is not an easy task as deadlocks and data races can be incredibly difficult to detect. A programmer who is too liberal with synchronization also risks creating bottle-necks that could seriously decrease the performance gain of parallel execution [1, 2].

With a high level of abstraction the programmer is able to more easily focus on solving the problem at hand. The actor model is a style of programming that employs this philosophy by isolating data within actors [3]. The result is total data race freedom, as only the actor itself can directly access and modify its internal state. However, there are situations where isolation is too restrictive to express certain patterns, leading to an inefficient implementation [4, 5]. Although additional abstraction may come with increased run-time, it could be worth it if patterns can be simplified.

Aliasing is a powerful tool in imperative languages, but is dangerous in a parallel setting. A more nuanced approach for actor-based languages is to extend an actor's interface by enabling it to expose its internal objects and implicitly require that any interaction still goes via the actor [4]. External entities can then be allowed to access the actor's internal structure while still guaranteeing synchronization. An external entity may also specify a sequence of operations to be performed on an internal object or actor without any interleaving. This thesis explores these two concepts including their design, implementation, and real world applications.

1.1 Goal and Purpose

The goal of this project is to implement two language constructs in an actor-based and object-oriented language called Encore [6]. These constructs enable safe interaction with another actor's internal objects and non-interleaved operations. There are two main challenges when extending a compiler with new language constructs. First, the solution is going to need functionality that is not present in the compiler. Second, integrating the solution into the rest of the language seamlessly by having it coupled as loosely as possible with other unrelated parts of the compiler. The purpose of implementing these two constructs is to evaluate whether they are useful in an actor-based language. While proven sound, they have not yet been proven useful.

1.2 Outline

The general structure of this thesis is as follows: First is an introduction to the actor model and Encore, followed by common methods for sharing data in a concurrent setting. The next item on the list is a more intricate description of the problem of sharing data in an actor-based language. After that, the design aspects of the constructs and their implementation into Encore are presented. Finally, the evaluation is done by finding patterns where the constructs can be used in a set of predefined benchmarks. Both the expressive power and performance aspects of the constructs are explored in the evaluation.

2 Background

2.1 The Actor Model

In recent years increasing clock speed of processors by adding more and smaller transistors has become increasingly difficult. Most modern systems instead utilize multi-core processors [1]. To harness this added computing power, software needs to actively use the available cores. Most programming languages were not designed with parallel execution in mind and can only utilize multiple cores by explicitly creating new threads or processes. Methods that achieve this passively have been explored, such as the actor model [3].

Since its introduction in 1973, the actor model has been relevant and influential in both research and industry [1, 3, 7]. It is a model for concurrency where actors are isolated entities which communicate with each other asynchronously in order to minimize common concurrency issues such as deadlocks and data races. Before delving into specifics, a common set of terminology must first be established:

Message: The unit of communication between different actors are messages. A message is a tuple of an identifier which defines the message's type and an optional payload that contains data.

Mailbox, aka. Message Queue: The messages that an actor receives are stored in its mailbox.

Interface: The types of messages that an actor understands and can process are defined by its interface. Passive objects can also have their own interface, which usually only their owner can directly interact with.

State: An actor's state is the data that is stored inside the actor, and which can be accessed synchronously by that actor alone.

Passive Object: Objects that are not actors are passive objects. Depending on the implementation, a passive object may be mutable or immutable, and isolated or shared between actors.

Actor, aka. Active Object: An actor can be defined as a three-tuple of its mailbox, interface, and state. Actors handle messages from its mailbox which are filtered through its interface. An actor may also interact with its own state and respond to other actor's messages.

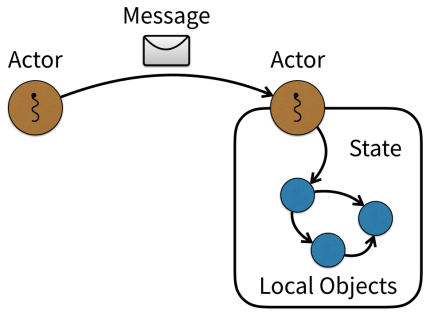

Figure 1: The actor model visualized.

There exists many variations of the actor model which are implemented in a wide array of languages and libraries [3]. These can be broken down into four distinct categories: *The Classic Actor Model, Processes, Active Objects,* and *Communicating Event-Loops* [3]. In this thesis the two latter categories are of interest.

Active Objects (AO) takes advantage of object-oriented design and applies it to the actor model [3]. Actors are called active objects and normal local objects are called passive objects. The interface of active objects are its methods, just like in any other object-oriented setting. Communicating Event-Loops (CEL) has a wider definition of actors: Each actor contains a heap with a set of objects and a single thread of execution, where the interface of any object in the actor's heap can be used as the actor's interface [3]. Two examples of such languages are E and AmbientTalk [3, 8].

The interface and isolation properties of actors are the two main areas of interest in this thesis. Systems using AO and CEL handle these a bit differently:

Interface: As described earlier, any object in an actor's heap can be used as an interface to the actor in CEL, while the methods of actors are the only valid interface in AO.

Isolation: When the state of an actor is only directly accessible and can be modified by the actor itself, then the actor's state is considered to be isolated. This is not always guaranteed in some actor-based languages, but usually both AO and CEL languages have this guarantee.

2.2 Encore

Encore is a general purpose object-oriented language which achieves parallelism mainly through the actor model [6], and can be categorized to be in the AO

subset. Its compiler is written in Haskell and translates Encore code to C code which is then compiled by a C compiler. As of the writing of this thesis, Encore is open-source[1] and details about its current syntax, capabilities, and features can be found in its documentation[2]. While the language contains many features, this section focuses on a few core properties.

```
active class Main
  def main() : unit
    println("Hello World!")
  end
end
```

```
def loop(n : int) : unit
  var numLoops = 0
  while (numLoops < n) do
    numLoops += 1
  end
end
```

Listing 1: Encore Hello World program and a loop method.

Like other actor based languages which use active objects, Encore's objects can either be active or a subclass of passive objects [6, 9]. Message sends may return *future* values which makes the act of sending messages asynchronous [6]. Future values serve as containers where the result of the computation will be stored eventually. An actor is thus not blocked after sending a message. Retrieving the result is however blocking if the result has yet to be computed.

Passive objects are the building blocks of active objects. They have reference semantics and are only supposed to be handled by a single actor at a time. Because reference semantics are supported it is still possible for an inexperienced (or malicious) programmer to write a program with data races. In Encore this problem is addressed with a type system called *Kappa* [9] that enforces synchronization and data race freedom, unless the programmer actively takes steps to get around Kappa by using the `unsafe` tag.

Encore's syntax has a mix of imperative, functional, and object-oriented flavour. A message send is performed with the `!` bang operator, while for synchronous method calls on passive objects the standard `.` dot operator is used. The future returned by a message send can be used as an argument for the `get` operation which retrieves the result from a future.

Defining whether an object is active or passive is done in its class definition. The `active` keyword is used for active objects and there are a plethora of keywords for different flavours of passive objects. For example `local`, which requires that the object is local to a single actor and thus cannot be shared between actors. There are also tags such as `sharable` for objects that can be shared between actors.

Encore also features polymorphism, and another important concept called *traits* which in Encore corresponds to abstract classes and interfaces. Lastly, there is

[1] https://github.com/parapluu/encore
[2] https://www.gitbook.com/book/stw/the-encore-programming-language/details

also support for closures which are similar to anonymous functions. The main difference is that an anonymous function is just a function with no name, while a closure also captures the state of the surrounding environment.

2.3 Sharing Data in Concurrent Programs

In imperative programming languages like C, C++ and Java, access to shared data is synchronized with mutex locks or semaphores [1, 2]. This needs to be done in order to avoid race conditions that can otherwise occur. Locks and semaphores do however come with additional overhead, as threads might need to wait for a resource to become available. This overhead does not scale well as more threads wants to use the same resource. The easiest way to reduce these negative effects is to write code that requires as little synchronization as possible [2]. It is however not always feasible and becomes increasingly difficult to avoid as programs become larger and more complex [2, 3, 7].

Some languages, like Java for example, support exclusive execution with the `synchronize` keyword, which ensure that only one thread can execute a given block of code at a time. In languages without similar features however, support for non-interleaved operations on data needs to be implemented by the programmer. Most commonly it is accomplished through locks or by making asynchronous operations synchronous [1, 2, 3].

An approach coming from the database world is the idea of transactions [10]. Where each operation must either complete in its entirety or have no effect in the system. Software transactional memory (STM) is one implementation that provide this functionality, where instead of lock-based synchronization, operations are allowed to execute in parallel [10]. Each memory access is monitored so that if a conflict which would lead to a data race is detected, it will cause one of the operations to abort and rollback to the beginning of the operation which will then be restarted. To the programmer and the system it will be as if no conflict had ever happened.

Actor-based programming languages like Erlang and Scala tries to solve both the synchronization and scaling problem with the actor model. In Erlang everything is immutable and sharing data is done by performing a deep copy of an object and sending that in a message to another actor [3]. Scala supports both mutable and immutable data, and the same pattern for sharing data in Erlang is possible in Scala, but immutable data does not need to be copied. A common pattern in both Erlang and Scala is to have one actor host the mutable data and have other actors send messages to retrieve and update it [3]. Transferring ownership is another alternative if both actors do not need to access the resource at the same time [1, 3, 6].

In Encore, an actor can not directly access an internal object of another actor, the actor must instead use the interface of the object's owner. Expressing certain patterns can thus be difficult due to the isolation property of actor-based

languages. Conventional methods described in this section could simplify these patterns to some degree, but a method more intimately tailored to the actor model is desired.

3 Problem Description

When an actor wants to interact with the internal objects of another target actor, it must go through the target's interface. To enable this interaction, the target actor must also extend its interface with methods that interact with its internal objects. The problem with this is that the target actor gets coupled with its internal objects and its interface gets bloated. By safely breaking isolation both of these problems can be solved.

3.1 Motivation: Breaking Isolation

Aliasing is widely used in imperative languages. Partly because aliasing is often a more efficient way to share data than copying, and in an object-oriented setting the programmer can directly interact with the shared object's interface, rather than going through the owner's interface. Sharing data in a parallel setting is however not always safe if the data is modified via aliasing, as this could expose the program to data races if there is no synchronization mechanism.

Listing 2 shows a list and iterator implementation in Encore, for simplicity the exact implementation of some methods are omitted and active objects are allowed to return values instead of futures. There are two problems with this program, disregarding that it is not a legal Encore program. Firstly, this implementation breaks encapsulation by having the iterator alias a node in the list without having to go through the list actor's interface. Secondly, there is no guarantee that the list is not changed between the client's method calls. When the client calls `hasNext` the result might be `True`, but the list could change between the `hasNext` and `getNext` calls.

A non-interleaved version of `hasNext` and `getNext` is needed to guarantee that the list cannot change between methods calls. This could be implemented in the list actor as a single method, although encapsulation would still be violated as a node in the list is still being aliased in the iterator. Integrating the iterator into the list actor's interface is another option, although this strongly couples the two classes. An iterator trait could also be considered in order to implement iterator functionality. However, both the trait and integrating the iterator into the list interface has the same problem that only one actor can iterate through a list at a time. If multiple actors want to iterate over the same list, the list would need to map different iterators to different actors.

Efficiently implementing this list and iterator pattern is challenging in Encore. By introducing new constructs that safely break isolation, the implementation of

this pattern can be simplified. A *bestowed object* is a safe reference to a passive object which can be shared between any number of actors, enabling another actor to safely interact with the passive object's interface without the risk of data races. An *atomic* block forces the target actor of the block to handle the messages sent to it within the block in sequence without interleaving for other messages.

With these tools, the synchronization, interleaving, and bloated interface problem can be solved. The list actor now instead creates a `Bestowed` object containing a local `Iter` to decouple the list and iterator interfaces, and uses an `atomic` block for non-interleaving. Listing 3 shows a safe version with both of these properties – `getIter` now returns a `Bestowed` object and an `atomic` block which operates on `iter` makes sure that only the client can modify the list object when performing operations within that block.

```
local class Node[t]
  var elem : t
  var next : Node[t]
end

active class List[t]
  var head : Node[t]
  def iter(): Iterator
end

local class Iterator
  var curr : Node[t]
  def getNext() : t
  def hasNext() : bool
end
```

```
active class Client
  var list : List[int]

  def iterate() : unit
    val iter = list ! iter()
    while (iter ! hasNext())
      val e = iter ! getNext()
      transmogrify(e)
    end
  end
end
```

Listing 2: A list and iterator, some method implementations are omitted.

```
active class List[t]
  var head : Node[t]

  -- Iter == Iterator
  def iter() : Bestowed[Iter]
end
```

```
def iterate() : unit
  val iter = list ! iter()
  atomic iter
    while (iter ! hasNext())
      val e = iter ! getNext()
      transmogrify(e)
    end
  end
end
```

Listing 3: data race free and non-interleaved iterator, Iter is the same as the Iterator class.

4 Design

The language constructs *bestow* and *atomic* make it possible to interact with internal objects of an actor while still guaranteeing that every operation is synchronized. They originate from Castegren and Wrigstad [4], who describe a use case similar to the one in Section 3.1, and define semantics for the constructs. The design of the constructs take advantage of functionality that is already present in the Encore compiler and tries to minimize additions to the run-time. This is because a solution generated by the compiler is more resilient to future changes in the run-time, than a run-time implementation is.

4.1 Bestow

The bestow operation lets an actor create a safe reference of one of its passive objects called a *bestowed object*. Isolation prohibits another actor from accessing an internal object of another actor, and bestowed objects are a thread safe way to break isolation. Figure 2 illustrates this, an actor tries to interact with the internal `foo` object of another actor, this interaction is possible by bestowing `foo`. A bestowed object looks and acts like an actor, and every message sent to it is relayed to its owner. This guarantees that every operation is synchronized, and enables a bestowed object to be shared between any number of actors without the risk of data races. There are two relevant entities of bestowed objects:

Bestowed Target: The passive object that is the target of bestow.

Bestowed Owner: The actor that performs bestow and carries out all of the operations on the bestowed target.

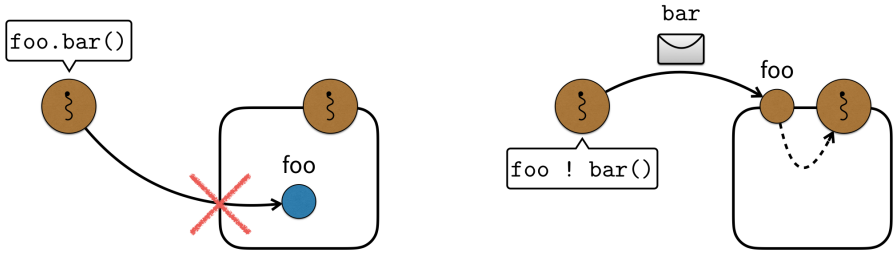

Figure 2: An actor can interact with the internal foo object of another actor by having its owner bestow it.

Requiring that every operation on a bestowed object is synchronized with the owner is not the same as aliasing an object like in an imperative language. This is because we can only interact with the bestowed target's interface and its state is inaccessible (private), but in an imperative setting the aliased object's state may not be inaccessible. One could argue that this forced isolation promotes

better coding conventions by disallowing patterns that could break encapsulation and couple classes.

4.1.1 Thread Safe Bestowed Objects

A bestowed object has the same interface as the bestowed target, and any interaction with it is asynchronous. This makes a bestowed object analogous to an active object, hence message sends instead of method calls should be used for interaction. Because not all passive objects are bestowed objects like in CEL languages, there should also be a tag on those objects. A new type for bestowed objects could be used for this purpose.

The bestowed type tracks whether an object is bestowed. This is used in the compiler to typecheck message sends on bestowed objects and to translate certain operations on bestowed objects into different ones. The bestowed type is also visible to the programmer, which makes it possible to determine whether or not an object is bestowed.

Not all Encore objects can be bestowed. Active objects already have a built-in synchronization mechanism and the bestow construct is therefore not needed for those objects. To adhere to the constraints posed by the Kappa typesystem in Encore, some subclasses of passive objects are also disallowed. One such subclass are `linear` objects, they are guaranteed to only have one reference existing in the entire program and allowing a linear object to be bestowed breaks that contract.

At run-time a physical representation of bestowed objects is needed so the compiler can generate code to interact with them. A bestow class in the standard library is a flexible option because its run-time representation will be generated by the compiler. Listing 4 is an example implementation. An `unsafe` class has no safety mechanisms in the Kappa type system, which allows us to share this object between multiple actors without any complaints from the compiler. Ideally the `target` and `owner` fields would also be private so that they are not accessible by the programmer (this is not yet possible to do in Encore). The `owner` has the `Actor` object type which is actually a trait, this will be explained in Section 4.1.2. Implementing bestowed objects directly in the run-time is also a viable option.

So how is the bestow class a safe reference that can be used to synchronize with the owner? Assuming that its fields are private, the compiler can extract the target and the owner, and redirect operations on the bestowed object to the owner. A bestowed object cannot be modified by an actor as its state is not accessible, making it immutable. Any number of actors can thus simultaneously interact with the same bestowed object without having to worry about synchronization.

```
unsafe class Bestow[unsafe t]
  private val target : t
  private val owner  : Actor

  def init(target : t, owner : Actor) : unit
    this.target = target
    this.owner  = owner
  end
end
```

Listing 4: An implementation of the bestow class.

4.1.2 Synchronization

With the bestowed type and the bestow class there is an interface for the compiler to extract the bestowed owner and target. Operations performed on the bestowed object can be any valid method in its interface, and the mechanism for synchronizing these operations with the owner rely on closures (Figure 3).

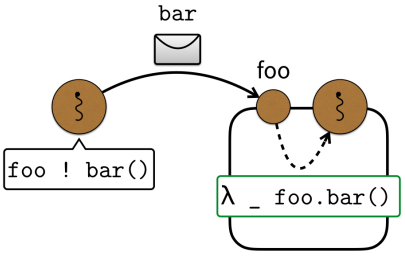

Figure 3: Closures are used to relay messages to the owner.

If the semantics of communicating with a bestowed object are identical to an active object, then when someone performs a message send on a bestowed object, it needs to be able to tell the owner to call that method on the target. The target might not necessarily be a member of the owner, as it might just have been created within that context. Therefore the owner might not have a reference to the object. To support both of these cases, the expression could be wrapped inside of a closure and sent to the owner to execute. Closures are convenient in Encore as the variables it interacts with from its current environment are copied into the closure's environment. Thus a local variable can be used inside of a closure, but cannot be reassigned.

To enable the bestowed target to run a generic closure, a new message type in its mailbox can be added in the run-time. This message has a closure as its payload, and when it is handled by the actor it will extract the closure and run it. Enabling this functionality globally is unsafe, as it might break the isolation property of actors or risk that a malicious programmer runs compromising code inside an actor (Listing 5).

```
def malicious_example(actor : active) : unit
  val closure = λ _ {while true do
                       println(42)
                     end}
  actor ! run_closure(closure)
end
```

Listing 5: Making an actor run an infinite loop.

Another alternative is to add the `Actor` trait in Listing 6, which includes a method for running an arbitrary closure. An active class which wants to use bestow can be extended with this trait. The trait is also active, which means that only active classes can be extended with the trait. The new `perform` method is polymorphic and takes a closure as an argument which returns a `sharable` type `t`. Sharable types are types that are allowed to be shared between actors in the Kappa type system.

```
active trait Actor
  def perform[sharable t](f : () -> t) : t
    f()
  end
end
```

Listing 6: An implementation of the Actor trait.

The trait design does not enable global support for running generic closures, but it restricts it by requiring the `Actor` trait. Programmers could still implement this trait themselves if they wanted to, so restricting the running of generic closures by requiring the `Actor` trait is not as intrusive as global support. Restricting the running of generic closures further is possible by making the `Actor` trait inaccessible to the programmer and have the compiler add it to any active class that uses bestow.

A small overhead is introduced by using the trait design when compared to the run-time implementation, as code for the perform method would need to be generated for every class that is extended with the trait. The main advantage with the trait over the run-time implementation, is that it is more future proof due to resilience of changes in the run-time.

Bestowed objects are safe to share between any number of actors, and with the `Actor` trait actors can also run generic closure. There is now a synchronization mechanism between bestowed objects and their owner, allowing an actor to safely interact with the internals of another actor.

4.2 Atomic

The atomic construct lets an actor specify a sequence of messages that need to be handled by another actor without interleaving for other messages. Atomic also allows intermediate results from these messages to be handled. The atomic construct is not necessarily tied with the bestow construct and can be used in other situations when performing multiple asynchronous operations which needs to not be interleaved.

Atomic is a block-expression which operates on a single actor, there are two primary parts of the atomic construct:

Atomic Target: The active object that is the target of the atomic block. It is this actor that has to handle the messages in sequence without interleaving for other messages.

Atomic Body: The operations within the atomic block.

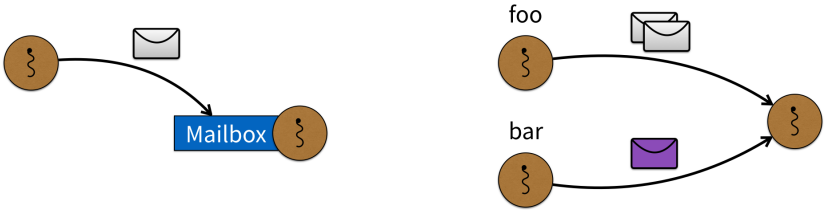

Figure 4: Actors only have one mailbox, atomic provides exclusive access.

Atomic gives an actor atomic (exclusive) access to another actor. In Figure 4, `foo` has atomic access and the messages from `bar` can thus not be handled before `foo` is finished running its atomic block. Unlike bestow, there are no problems with any internal state that arises from this as both of the actors are still isolated from each other. The main problem to solve is how to achieve exclusive access for one specific actor and still allow other actors to send messages to its mailbox.

Enforcing non-interleaving of atomic messages can be accomplished in a multitude of ways, and two different designs are discusses in this thesis. The first design groups messages together into a single message using closures, and the second design utilizes multiple mailboxes to sort atomic messages from other messages. These designs have different semantics, but enforces non-interleaving nonetheless.

4.2.1 Semantics

An important question to address, is whether to make atomic a blocking operation or not. Both alternatives are valid but lead to slightly different functionality. Depending on which option is chosen, message sends inside the atomic body

performed on the target can be converted from asynchronous message sends to synchronous method calls.

If atomic is blocking, message sends can be translated to method calls and there would thus be no need for using `get` to retrieve a value from a future. Using method call syntax instead of message send syntax is then a valid option. It would also be worth considering whether to disallow results from message sends on the atomic target to be used as the argument of expressions that operate on futures.

If atomic is not blocking however, then every message send on the atomic target would still need to be a message send. The semantics for operations inside the atomic block would thus be identical to the rest of the language (except for the exclusive access to the atomic target).

4.2.2 Enforcing Non-Interleaving

The messages sent to the atomic target needs to be handled without interleaving for other messages. The atomic target could be bombarded by messages from other actors, but it still needs to guarantee that atomic messages are handled first. The two main options that have been considered for enabling this are as follows:

Atomic Closure: Wrap the atomic body inside of a single closure, which can then be sent to and run by the atomic target. Effectively grouping multiple messages into one.

Atomic Mailbox: Actors can switch which mailbox it reads messages from. Allowing actors that want atomic access to exclusively write to a shared mailbox, while other actors still write to the old mailbox.

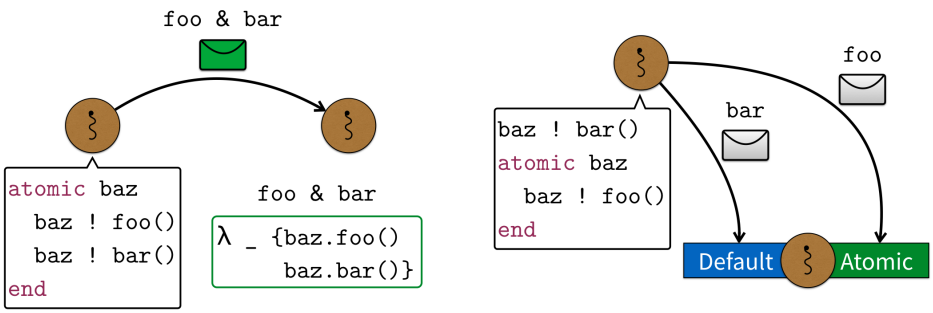

Figure 5: Atomic closure on the left, and atomic mailbox on the right.

Atomic Closure bundles the whole body of the atomic block into a single message by wrapping it inside of a closure (Figure 5). The `Actor` trait from Listing 6 can be utilized to enable the atomic target to run a generic closure. This allows a programmer to interact with intermediate values inside the atomic

body, but requires that atomic is a blocking operation if data declared outside the atomic body is to be mutable without any race conditions. This is because the closure will be run by the atomic target, and if the other actor with atomic access continues beyond its atomic block, then both actors could access and modify the same object at the same time. To eliminate this race condition, atomic closure thus has to be blocking. The blocking nature of atomic closure also leads to a new scenario where deadlocks can occur. This scenario is when the actor with atomic access and the atomic target are blocking on each other at the same time. Although, this scenario is fairly obvious if a programmer knows that atomic is a blocking operation.

Atomic Mailbox gives an actor atomic access by putting atomic messages in their own message queue (Figure 5). To enable the use of multiple message queues, two new pointers to message queues called `read` and `write` are introduced (Figure 6). All messages sent to the actor are placed where `write` points to, and the actor itself reads messages from wherever `read` points to. When not in an atomic block, both `read` and `write` will point to the `Default` message queue like the two left most actors in Figure 6. But whenever an atomic block is entered, a new message queue called `Atomic` is created, and the target of the atomic block is sent a message to switch so that its `read` pointer points to the `Atomic` message queue. This can be seen in the two right most actors in Figure 6. Any message sent to the atomic target is still sent to wherever `write` points. But the actor with atomic access has exclusive access to the `Atomic` message queue and thus to the atomic target. When the body of the atomic block has finished running, another message is sent to the target to switch back so that `read` points to the `Default` message queue again. This design does not require that atomic is a blocking operation and its semantics are identical to those described by Castegren and Wrigstad [4].

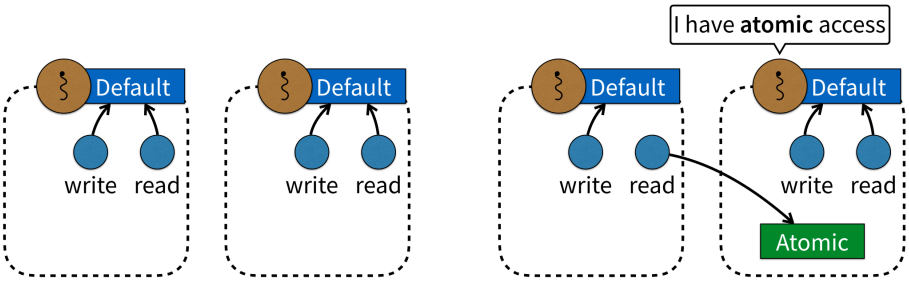

Figure 6: The read and write pointers of atomic mailbox.

With the atomic closure design there is opportunity for performance optimizations by translating certain asynchronous message sends to synchronous method calls. The main disadvantage is that the atomic block has to be a blocking operation and that closures would need to be extended in order to allow for mutation of local data declared outside of the atomic block.

Although the atomic mailbox design requires modification of the actor implemen-

tation in the run-time, the semantics are much clearer and there are fewer risks of deadlocking as atomic would not have to be a blocking operation. It is also less complex conceptually for a programmer, as the only semantic difference is exclusive access to the atomic target.

Both atomic closure and atomic mailbox have been partially implemented as they are interesting to contrast with each other in practice.

4.2.3 Message Sends

All message sends on the atomic target needs to be sent to the correct queue. But detecting that a message send is performed on an atomic target is not always straightforward. Consider the example in Listing 7. Two message sends on the atomic target are performed outside of the body in two functions. Logically these operations are inside of the atomic body and should thus be sent to the atomic message queue.

The scope of the atomic block could be limited to make code like in Listing 7 not valid atomic message sends. Or the scope could be encompass the whole actor. There are three options for this problem:

Lexical Scope: An atomic block only affects statements that are in its immediate lexical scope.

Actor Wide: An atomic block affects all statements in the current actor that are executed between its start and finish.

Atomic Handle: An atomic block gives an additional handle to an actor and only operations on the handle are affected.

```
atomic this.a
  this.foo()
  this.bar(this.a)
end
```

```
def foo() : unit
  this.a ! baz()
end

def bar(a : active) : unit
  a ! baz()
end
```

Listing 7: Atomic target used outside of the atomic body. What scope does atomic operations have?

Relating to Listing 7, under lexical scope neither of the message sends will be atomic, under actor wide all of the message sends will be atomic, and under atomic handle only `bar` will be atomic.

Lexical Scope puts unnecessary constraints on valid atomic messages. And it can lead to messy code if every atomic message send has to be concentrated

directly inside of the atomic block. Its strength is that it is simple to implement and for a programmer to understand.

Actor Wide allows for more flexible code by widening the scope of atomic operations to encompass the entire actor. However, it is a bit more complex conceptually for the programmer to understand and implementing actor wide is neither straightforward. One implementation is to search the arguments and bodies of all message sends and method calls inside the atomic block in order to identify every atomic operation. Another implementation is to save all atomic targets in some data structure in the actor with atomic access, and perform a check during every message send if the target is atomic. Both of these options are quite difficult to implement efficiently.

Atomic Handle makes it clear for the programmer that they are interacting with an atomic target and the compiler can statically check if a message send is performed on an atomic target. Atomic handle is about as flexible as actor wide, but require that the programmer passes the atomic handle as an argument to any method that wants to perform an atomic operation.

The choice for this implementation of atomic would have been atomic handle, as it is easier to implement efficiently than actor wide and code is still easy to reason about. But due to time constraints, lexical scope has been implemented.

5 Implementation

The Encore compiler is written in Haskell and generates C code which is compiled to an executable by a C compiler and linked with a run-time written in C. There are five distinct steps in the Encore compiler which are performed in the following order:

1. **Parser:** The Encore input files are parsed and then an abstract syntax tree (AST) is created.
2. **Desugarer:** Replaces an AST node with a different one, for example, a for loop may be rewritten as a while loop.
3. **Typechecker:** Annotates all the types of every expression and checks that they have the correct type. It both catches and eliminates execution errors during compilation that can otherwise occur during run-time.
4. **Optimizer:** The second desugaring phase with type information.
5. **Code Generation:** Translates the annotated AST into C code.

In the generated C code there are a few interesting things to note. Actors have a message queue in the run-time where all messages sent to it are put. When these messages are eventually handled by an actor, they are filtered through its *dispatcher*, the C interface of an actor. Here all of its methods and necessary language constructs are represented. Another point of interest

is closures, which are static functions (Listing 8) that have an environment (C struct) which is dynamically created whenever a closure is run. This means that a new environment needs to be created for every closure call.

To implement the bestow and atomic language constructs in Encore, the compiler has to be extended in all its different parts. For bestow and atomic there already is support for closures in Encore which can be utilized. Bestow can thus mostly rely on functionality that is already in the compiler. Atomic on the other hand needs to be able to switch between mailboxes and allow for closure mutability, which are new concepts in the language.

Adding functionality in the Parser and new nodes in the AST is straightforward, as both bestow and atomic are orthogonal to everything else in the Parser and AST, meaning that there are no conflicts to resolve. The rest of the stages in the compiler are more interesting however.

```
// the environment
struct env_closure
{
  int i;
};

// the closure, only showing the relevant argument
int closure(struct env_closure* env)
{
  // extract variables out of the environment
  int i = env->i;
  // the body of the closure
  return i;
}
```

Listing 8: Simplification of the generated code for a closure. This closure returns the value of an integer variable.

5.1 Bestow

Bestow enables an actor to safely interact with the interface of an internal object of another actor, this is a different pattern than having to go through an actor's interface in order to interact with its internal objects. The implementation outline of bestow is to add parsing, a new AST node, a new type called `Bestowed`, and perform necessary typechecking on the argument of the bestow expression. The bestow expression and performing a message send on a bestowed object will also be desugared in the Optimizer.

5.1.1 Typechecking

Remember that the building blocks of active objects are built with the passive subclasses. As discussed in Section 4.1.1, some subclasses are not valid targets for bestow. Checking that these conditions are satisfied is done by confirming that the target expression is a passive object which is either `local` or `subord`.

The bestowed construct can in the current implementation only be used by an active object. It does not make sense to limit so that actors only can use the bestow construct, as this makes it difficult for separate libraries to use bestow. Allowing passive objects to use bestow is a fairly trivial change that just require that the bestowed owner is assigned differently by the generated C code. The owner must also include the `Actor` trait, otherwise it is not possible to send a generic closure to the owner via the perform method.

An operation on a bestowed object must be well-typed. For example, the method called with a message send on a bestowed object must also exist and match the definition in the target's interface. The typechecking for such an expression is therefore done by checking the well-typedness in relation to the target. This is important also for generating good error messages in the compiler, like when the wrong number of arguments are used.

With the constraints posed by their design, bestowed objects behave almost semantically the same as active objects. But in order to allow the use of the message send syntax on bestowed objects they need to be defined as active objects in the type system. This circumvents having to implement a special case for message sends on bestowed types by simply extending the definition of active objects.

5.1.2 Optimizer

Simply extending the definition of active objects to include bestowed objects only solves the typechecking part of a bestowed expression. The compiler still has no idea what a message send on a bestowed object is supposed to be translated into. In the Optimizer it is possible to desugar any part of the AST. There are two types of expressions that need to be desugared, both of which can be seen in Listing 9.

```
1 val obj = bestow target
2 obj ! foo()
```

Listing 9: Bestow expressions which need to be desugared.

The first item in Listing 9 is where a bestowed object is created, it needs to be desugared into creating a new `Bestow` object (from Listing 4) and the type of the

expression is set to `Bestowed`. The second item needs to transform the message send to a method call and apply it on the bestowed target. This method call then has to be wrapped inside of a closure which is used as an argument of a message send with the `perform` method of the owner.

The desugared versions of the two expressions can be seen in Listing 10, where the left hand side corresponds to the first item and the right side the second item in Listing 9. Note that for every message send a new closure environment (Listing 8) is created, which has to be allocated on the heap, have its members assigned, and eventually be garbage collected.

```
-- 1. desugared to:
new Bestow(target, this)
```

```
-- 2. desugared to:
val target = obj.target
val owner = obj.owner
val closure = λ _ target.foo()
owner ! perform(closure)
```

Listing 10: Desugaring the two bestow expressions in the Optimizer.

5.1.3 Garbage Collection of Bestowed Objects

Bestowed objects are unlike any other type of object in Encore, and a special case is needed in order to properly garbage collect them. In Encore, referencing counting is used to track the number of references to a particular object. This partly enables the garbage collector to determine if an object is alive by checking if an object's reference count is greater than zero. In order to count these references, objects are traced by a tracing function that recursively traces any reference to other objects (Figure 7). A bestowed object's tracing function needs to trace both its owner and target fields like any other type of object in Encore. But because bestowed objects can be shared between multiple actors, this recursive tracing is not a safe thing to do. The bestowed target is local to the owner, and only the owner has access to meta information of the target which is needed during tracing. Recursively tracing the bestowed target will thus cause the program to crash.

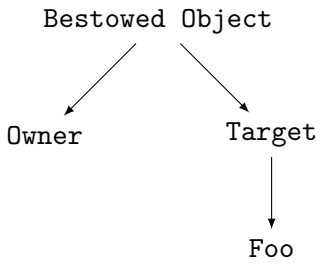

Figure 7: Example structure of a bestowed object.

The first step in this solution to the garbage collection problem, is to not recursively trace the bestowed target. This guarantees that the owner is kept alive, but subobjects of the target do not have the same guarantee. In the example in Figure 7, `foo` is a subobject of the bestowed target. The bestowed owner does not have a reference to either the target or `foo`. Thankfully the bestowed object's tracing function will guarantee that target is kept alive, but `foo` will be garbage collected because no recursive tracing is done.

When the owner performs garbage collection it needs to be able to recursively trace the objects it has bestowed. The owner might not have a reference these objects, so a reference will need to be saved somewhere in the owner. During the tracing phase, these objects can then be retrieved and recursively traced. Now `foo` is also guaranteed to be kept alive, and when a bestowed object goes out of scope it can be garbage collected properly.

5.2 Atomic

Atomic enables an actor to send a series of messages to an actor which are guaranteed to be handled in sequence without interleaving for other messages. The implementation outline of atomic is to first add parsing and a new AST node. The implementations of the two atomic designs diverge at this point in the compiler. The closure version of atomic require mutability of external variables inside closures. The Optimizer is partly used for this purpose by desugaring message sends on the atomic target in the atomic body, and in the code generation closures can be enabled to modify external variables. For atomic mailbox the actor implementation in the run-time needs to be modified, and there are several concurrency challenges to solve.

5.2.1 Typechecking

In a well-typed atomic block the only restrictions are related to the target, which must always be an active or bestowed object. The scope of atomic operations can vary depending on the implementation (Section 4.2.3). If a new atomic reference is created at the start of an atomic block, then this reference has some limitations on its use. The reference has to be local to the current actor, cannot be assigned to a field, or put in a data structure for example. This is because the scope of atomic operations are restricted to only the atomic body, and the atomic reference's scope is similar to a stack variable which can only be used within its immediate scope.

Atomic closure performs synchronous method calls on the atomic target, which needs to be enabled in the typechecker in order to allow synchronous semantics. Futures are not returned by such message sends on the atomic target, so the typechecker also has to catch so that operations on futures like `get` are not applied on the return value.

5.2.2 Optimizer

In this implementation of atomic closure, the message sends on the atomic target inside the atomic body are desugared to method calls, and the desugared body is then wrapped inside of a closure. The desugared atomic closure block can be seen in Listing 11.

```
-- expression:                    -- desugared to:
atomic obj as baz in              val baz = obj
  baz ! foo()                     val closure = λ _ {baz.foo();
  baz ! bar()                                        baz.bar()}
end                               baz ! perform(closure)
```

Listing 11: Desugaring atomic closure in the Optimizer.

5.2.3 Code Generation

External variables inside of a closure's environment are represented as primitives or pointers to objects in the generated C code (Listing 8). To enable an atomic closure block to mutate external variables, the objects can be modified to double pointers, and primitives to single pointers. This enables reassignment of variables by dereferencing them in every operation inside the closure. The actor that performs the atomic operation will be blocked while it waits for the atomic target to run the closure. Because it is blocked, it is safe to dereference the variables when the atomic target runs the closure, as the variables are guaranteed to still be alive.

Enabling closure mutability via the method explained above is easier said than done, the simplest route is to add a new AST node or a type that serves as a flag in the code generation to dereference these variables. Listing 12 is an example of the code that will be generated to enable closure mutability. The closure tries to modify an external integer `i`, and by having its address in the environment, it can be dereferenced and reassigned inside the closure. To the left, the reassignment of `i` will only be local to the closure itself, while to the right, the reassignment of `i` will propagate to the original owner of `i`.

An atomic mailbox block must be initialized before its body can be executed, all the required steps can be seen in Listing 13. A simplified version of this, is to first create a new alias for the atomic target (if applicable), and a new atomic message queue. An atomic start message then needs to be sent to the target. This message contains the new message queue as its payload, which the atomic target will start reading messages from after handling the message. All the messages sent to the atomic target inside the body now needs to be sent to the new atomic queue. To do this, the write message queue of the atomic target

```
// no closure mutability
struct env
{
  int64_t i;
  Foo_object* foo;
};

void closure(struct env* env)
{
  int64_t i = env->i;
  Foo_object* foo = env->foo;
  i = baz(foo);
}
```

```
// closure mutability
struct env
{
  int64_t* i;
  Foo_object** foo;
};

void closure(struct env* env)
{
  int64_t* i = env->i;
  Foo_object** foo = env->foo;
  *i = baz(*foo);
}
```

Listing 12: Enabling closure mutability by using the address of variables and dereferencing them inside the closure.

is reassigned locally to the atomic queue. The body can then be executed. At the end of the body an atomic end message is sent to signal the end of the atomic block, allowing the atomic target to start reading from its default message queue again.

The code generation for atomic mailbox is a bit more involved in the actual implementation. For example, reassigning the write message queue like in Listing 13, would be a race condition as isolation is broken. To allow for this lazy reassignment of the write message queue, a shallow copy on the stack can be created. This shallow copy can be modified without subjecting the implementation to race conditions.

```
-- The atomic mailbox block
atomic obj as baz in
  baz ! foo()
end
```

```
// initialize atomic
actor* baz = obj;
msgq* q = new_messageq();
atomic_start_msg(baz, q);
baz->write = q;

// atomic body
send_foo_message(baz);

// end of atomic
atomic_end_msg(baz);
```

Listing 13: Simplification of the generated C code for an atomic mailbox block.

5.2.4 Run-Time

The added run-time functionality for atomic mailbox includes creating two new types of messages which signify the start and end of an atomic block, and modifying the actor implementation. Actors also need to be able to handle such messages, which is done by adding a case for each new message in an actor's dispatcher. The actor implementation is modified with a *read* and a *write* pointer to two message queues. Everything that interacts with an actor's message queue needs to be modified so it references the right queue. Fortunately, there is only one instance where the write queue needs to be used, which is when an actor wants to send a message to an actor. But modifying the actor implementation affects many places in the run-time, and some of the introduced problems are not trivial to solve. These problems will be discussed in Section 5.2.5.

5.2.5 Scheduling the Atomic Target

A non-trivial concurrency problem arises when modifying the actor implementation with a read and write pointer in atomic mailbox. There are a finite number of cores in a computer, and an actor can only run on a single core at a time. A scheduling algorithm is used in order to determine when an actor gets to run on one of the cores. How this scheduling algorithm works is not important, but atomic mailbox needs to modify when actors are scheduled in order to avoid the atomic target being scheduled multiple times.

In the example in Listing 14, the `size` message will be sent to the default message queue and the `append` message will be sent to the atomic message queue. If an actor's message queue is empty it will be scheduled by the Encore scheduler, and if both of these message queues are empty, the actor will be scheduled twice. It is possible that an actor will run on two cores at the same time if it is scheduled twice. This can lead to messages being handled out of order, and if we are (un)lucky the whole program could crash.

```
active class List
  var size : int
  var head : Node

  def size() : int
  def append(i : int) : unit
end
```

```
def newList() : unit
  val l = new List()
  var s = l ! size()
  atomic l
    l ! append(42)
  end
end
```

Listing 14: Exemplifies the scheduling problem, some method implementation are omitted.

There are a lot of assumptions in the run-time that an actor only has one message queue. The sections of code where multiple message queues can cause problems

are thus numerous. A lock could be used whenever an actor wants to perform an operation on one of its message queues, however this comes with a significant overhead. A more complex scheme can be employed to make sure that an actor does not get scheduled again if it is already scheduled. This involves atomically reading the address where the read pointer points to in case it has be updated, and resolving race conditions when an actor is flagged as being scheduled.

An example of a race condition that is particularly difficult to resolve without the use of locks can be seen in Listing 15. Enqueuing a message and determining whether an actor's message queue is empty is not subject to any race condition. However with multiple message queues, the if condition at the first comment may be `True`, but the actor may be scheduled by another actor before the function call at the second comment has completed. This is because multiple actors with different empty atomic message queues may be trying to schedule the same actor.

```
void send_message(actor* a, message* m)
{
  bool is_empty = messageq_enqueue(a, m);
  if (is_empty)
  {
    if (!actor_is_scheduled(a)) // 1.
      schedule_actor(a);        // 2.
  }
}
```

Listing 15: A race condition when sending messages.

6 Evaluation

A set of benchmarks for actor-based languages called *Savina* [11] are partially implemented in Encore, and have been used as inspiration to evaluate whether any patterns can be improved by using bestow or atomic. These benchmarks tackle common concurrency problems which measures different aspects in an actor-based system. For example: message passing overhead, message throughput, and mailbox contention.

A small set of the Savina benchmarks, and a continuation of the iterator in Section 3.1 will be discussed. The Savina benchmarks includes:

1. **Ping:** Message delivery overhead.
2. **Concurrent Sorted Linked-List:** Reader-writer concurrency and interacting with a linear time data structure.
3. **Sleeping Barber:** Inter-process communication and state synchronization.

Ping is used for measuring the overhead of message sends when using both of the constructs. It is a slightly modified version of the *Ping Pong* benchmark in Savina that only performs a ping message with no response pong message, as only the act of sending a message is of interest and a response pong message does not tell us anything about the message send overhead. The concurrent list is used to contrast any overhead that bestow has by performing a linear time operation for every message. The sleeping barber benchmark along with the iterator is used to evaluate the expressiveness and use-cases of bestow and atomic.

6.1 Expressive Power

The use-case of bestow described in Section 3.1 lays out a scenario where bestow simplifies patterns. This scenario is when performing a task where the internal object is the only entity of interest, and without bestow the owner's interface would need to be extended with methods of its internal objects. Atomic on the other hand proves to be a natural way of resolving the non-interleaved pattern. When only considering the expressiveness of atomic, it is difficult to imagine a scenario where atomic is a bad choice of achieving non-interleaving of messages. Determining whether there are any patterns where bestow or atomic prove to be especially useful is the purpose of this section, and their limitations are also explored.

6.1.1 Sleeping Barber

This benchmark is a simulation of a barbershop where the barber cuts customers' hair at a constant speed, but if there are no customers he falls asleep. Customers arrive at a random rate to the barbershop, if there is a queue and it is not full the customer waits for its turn, otherwise the customer leaves. If there is no queue, the customer wakes up the barber to receive a haircut.

Actors need to be able to communicate with each other in order to synchronize. One could consider using bestow for extracting the barber's state (awake or asleep), but it is a bit of an anti-pattern as it would require a new passive class for that purpose. The barber class itself could be converted to a passive class and bestow itself. But a more reasonable option is to make the queue that customer's wait in a passive class and a field in the barber. This makes it clear that the barber owns the queue, which it then can bestow and give out to customers.

There is no other great candidate for bestow, however atomic can be used to create a more streamlined interface for the barber. In Listing 16, the different required functionality of the barber's interface can effectively be spread out and used by a customer to check if a barber is asleep, wake him up, and receive a haircut. Without atomic, a queue would be needed to synchronize so that only one customer could communicate with the barber at a time, but with atomic all of

their messages are non-interleaved. Now all customers instead can communicate simultaneously.

```
active class Barber
  var isAsleep : bool

  def work() : unit
    -- cut customer's hair
  end

  def checkOnBarber() : unit
    if this.isAsleep then
      this.isAsleep = false
      println("Barber awake")
      this.work()
    end
  end

  def sleep() : unit
    println("Barber asleep")
    this.isAsleep = true
  end
end
```

```
active class Barber
  var isAsleep : bool

  def work() : unit
    -- cut customer's hair
  end

  def isAsleep() : bool
    this.asleep
  end

  def sleep() : unit
    println("Barber asleep")
    this.isAsleep = true
  end

  def wake() : unit
    println("Barber awake")
    this.isAsleep = false
    this.work()
  end
end
```

Listing 16: The Barber's interface using atomic on the right, and without atomic on the left.

This allows for more fine-grained concurrency as actors have to wait less and methods can be simplified so that an external actor has more control in-between message sends. A customer can now for example choose to leave if the barber is asleep. Atomic is also particularly useful in this scenario, as a programmer does not need to implement a synchronization mechanism of their own (like a queue).

6.1.2 Iterator

The interface aspect of bestow has previously been highlighted with a list iterator in Section 3.1. A more in depth example is explored in this section to showcase how tricky this pattern is to implement without bestow. This also allows us to identify more instances where bestow can be used to simplify patterns, and what a clever programmer can do to control the level of concurrency in a program.

Having a separate class for the iterator becomes tricky as it cannot directly alias any of the collection's internals. It would then require that the collection can keep track of multiple states when multiple actors want to iterate at the same

time. A hash map could be used for this purpose, and map different actors to different iterator states. Another option is converting the iterators into actors. This would double the overhead of interacting with the collection, as messages sent to the iterator would need to be sent by the iterator to the collection as well. The interface of the collection would also need to be larger, as the collection must mirror the iterator's interface. Merging the collection and the iterator thus makes more sense in order to eliminate this overhead.

```
local class Node[t]
  -- omitted
end

-- extended with Iterable
active class List[t]
    : Iterable
  val head : Node[t]
  val map  : HashMap[t]

  def iter() : int
    val key = map.hash()
    map.add(key, this.head)
    key
  end

  def next(key : int) : t
    val tmp = map.get(key)
    map.replace(key, tmp.next)
    tmp.elem
  end
end
```

```
local class Node[t]
  -- omitted
end

active class List[t]
  val head : Node[t]

  def iter() : Bestowed[Iter]
    bestow new Iter(this.head)
  end
end

local class Iter[t]
  val current : Node[t]

  def init(h : Node[t]) : unit
    current = h
  end

  def getNext() : t
    val tmp = current
    current = tmp.next
    tmp.elem
  end
end
```

Listing 17: List and Iter can be decoupled by using bestow, and bestowed objects can also be used for synchronization.

Bestow allows the iterator to become a passive internal object of the collection, and both the collection and the iterator can be decoupled from each other. In Listing 17, the `List` class' interface when using bestow no longer needs to be cluttered with the iterator's interface. This cluttering becomes more problematic the larger the iterator class' interface is. Neither does a synchronization mechanism need to be implemented by the programmer to correctly map different iterator states to different actors.

The bestow construct enables more fine-grained concurrency patterns. Consider the difference between creating and bestowing an iterator inside of a list (like in Listing 17), versus creating an iterator outside the list and bestowing each

individual node. In the first case every individual operation on the iterator will be done without any interleaving. While in the second case only operations on individual nodes will be done without any interleaving. This gives a clever programmer the ability to control the level of concurrency and synchronization that is needed in different situations.

6.2 Performance

Bestow and atomic are tools which can be used to interact with an actor at a higher level of abstraction. While abstraction is often useful in simplifying the implementation of patterns for a programmer, it can come at the cost of additional overhead at run-time. Since Encore translates into C code, a C profiling tool can be used to quantify the overhead. The profiling tool of choice is *gprof* and the benchmarks were performed on a machine equipped with an Intel Core i5-5200U (4 cores, 3MB L3 Cache) processor running at 4×2.20 Ghz combined with 8GB of RAM, which runs on Ubuntu 14.04 LTS.

Each benchmark has been run 10 times and their gprof output has been averaged. In gprof's output, the displayed time is the amount of CPU time spent in user-mode code (outside the kernel) within the process. As a result, on a multi-core system the CPU time can actually exceed the real time. The output generated by gprof has been condensed down to highlight the interesting parts. The function names has been slightly simplified to increase readability. There is no meta information other than self and cumulative seconds for run-time functions, which have been prefixed with `enc`. Regular methods have their class name as an identifier followed by the method name and whether if it is a synchronous method call or asynchronous message send.

6.2.1 Bestow Overhead

The Ping benchmark is used to measure the overhead of performing message sends on a bestowed object with and without atomic access. This benchmark (Listing 18) represents the maximal overhead of bestow over normal actor messages. The bestow version of the Ping benchmark converts the active `Ping` class into a `local` passive class, and instead has a new active wrapper class `bestow` a `Ping` object with is used in `sendPing`.

In Figure 8 both of the cases, with and without atomic can be seen. The left graph is comparing message sends on actors versus bestowed objects. To no surprise, bestow comes with an overhead, but the size of the overhead is quite staggering. In the right graph the message sends are wrapped inside of an atomic closure block which means that only one message has to be sent to the target. Everything inside this block can also be performed synchronously by the target. Using atomic like this, the overhead can be transformed into a performance improvement instead.

```
active class Ping
  def ping() : unit
    () -- does nothing
  end
end
```

```
def sendPing(n : int) : unit
  val ping = new Ping()
  var numMsgs = 0
  while (n > numMsgs) do
    ping ! ping()
    numMsgs += 1
  end
end
```

Listing 18: The Ping benchmark.

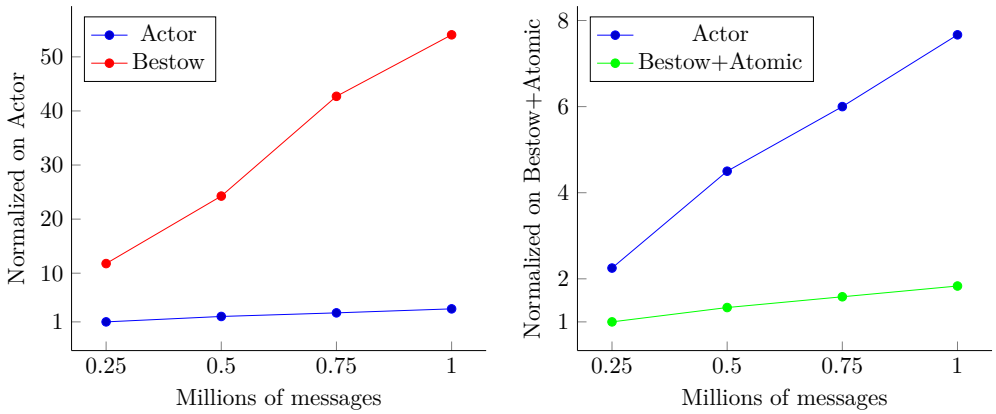

Figure 8: Overhead of message sends on bestowed objects, and using atomic closure to reduce the impact of the overhead.

To figure out what is the cause of the overhead the output of gprof can be analyzed. Table 1 is Ping on an actor, and Table 2 is Ping with a bestowed object, each has performed one million messages. The main difference is that message sends on bestowed objects needs to perform a lot more costly operations in the run-time. Specifically, for every message send a new closure environment is allocated and has to be traced for garbage collection purposes. When performing ping on a normal actor, none of these things needs to be done. In bestow's case a lot more time is hence spent in the run-time to handle these one million new closure environments that have to be created. Allocating, tracing, and freeing these closures environments is the cause of the overhead of bestow. The overhead of the tracing scales proportional to the structure of the bestowed target, for example the cost of recursively tracing a linked list grows linearly with the list's size.

When wrapping the pings inside an atomic closure block, this overhead can be drastically reduced. Using atomic in Table 3, only one message send to the owner has to be sent, allowing it to run everything else synchronously. Run-time specific functions barely register in the profiling because of this. Using atomic mailbox instead of atomic closure would be negligibly slower than bestow, as two messages signaling the start and end of an atomic block has to be sent.

% time	cumulative seconds	self seconds	calls	name
28.57	0.08	0.08		enc_messageq_push
14.29	0.12	0.04	1000000	Ping_ping_async
9.52	0.14	0.02	1000000	Ping_ping_sync
4.76	0.15	0.01		enc_alloc_msg
	==== The rest is mostly irrelevant ====			

Table 1: Ping with message sends on an active object.

% time	cumulative seconds	self seconds	name
21.64	0.45	0.45	enc_hashmap_get
6.25	0.58	0.13	enc_opt_next
5.29	0.69	0.11	encore_trace_object
4.57	0.79	0.10	enc_messageq_push
	==== Mostly more GC and tracing ====		

Table 2: Ping with message sends on a bestowed object.

% time	cumulative seconds	self seconds	calls	name
100.00	0.02	0.02	1	fun_closure0
0.00	0.02	0.00	1	Ping_perform_async
0.00	0.02	0.00	1000000	Ping_ping_sync

Table 3: Ping with message sends on a bestowed object, wrapped inside an atomic block.

6.2.2 Atomic Overhead

The overhead of both atomic versions is measured using the same Ping benchmark as in Listing 18. Like bestow, the atomic benchmark also represents the maximal overhead of atomic over normal actor messages. The atomic version of the Ping benchmark wraps each ping message inside an atomic block (Listing 19). It should be noted that atomic mailbox does work properly in the Ping benchmark.

```
while (n > numMsgs) do
  atomic ping as p in
    p ! ping()
  end
  numMsgs += 1
end
```

Listing 19: The atomic version of the Ping benchmark.

In Figure 9 the overhead for both atomic closure and atomic mailbox can be seen. Both versions have an overhead compared to message sends on actors. Atomic closure's overhead is substantially larger than atomic mailbox's, and atomic mailbox's overhead is smaller than bestow's. Analyzing the generated C code for atomic closure reveals that it is mostly identical to bestow. The difference is that atomic closure is a blocking operation and will thus have to wait for the atomic target to finish running the closure. In the case of atomic mailbox, the overhead quite clearly comes from handling new message queues at run-time, in addition to the two extra messages at the start and end of every atomic block.

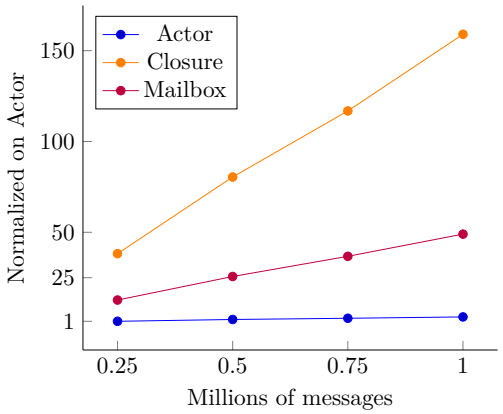

Figure 9: Comparing Ping with versions using atomic.

The gprof output for atomic closure is in Table 4, where one million messages have been performed. It looks similar to the output of bestow, the main difference is the time that each operation takes. As mentioned before, the root cause of this is the blocking nature of atomic closure. One of the actors in the program will thus be blocked for an extended period of time. Another message will neither be sent before the earlier message is completed. This means that the target actor will be descheduled as its message queue will be empty. The scheduler will then also have to work extra by rescheduling the two actors. As a result the two actors are spending a long time idle compared to bestow.

The cause of the overhead of atomic mailbox can be seen in Table 5, where one million messages have been performed. The culprits are indeed the new message queues and extra messages. To a lesser extent atomic mailbox also suffers from idling like atomic closure. After each atomic mailbox block the atomic target's atomic message queue will be empty, the Encore scheduler will then deschedule the target even though its default message queue may be full of messages. Due to the sheer number of messages sent to the atomic target, the time being spent idle by the target is not as severe.

An intuitive observation is that the overhead of both atomic closure and atomic mailbox scales proportionally to the number of message sends inside of the atomic block, and the number of individual atomic blocks. If many messages can be

concentrated inside of a singular atomic block, the overhead can be negligible. However, the more individual atomic blocks that exists, the more atomic messages will need to be sent. And the larger the discrepancy between individual atomic blocks and number of message sends within them are, the larger the overhead will be. There thus exists a break-even point between the overhead of atomic closure and the performance improvement of using method calls instead of message sends.

% time	cumulative seconds	self seconds	name
19.87	1.21	1.21	enc_mpmcq_pop
10.18	1.83	0.62	enc_hashmap_get
8.13	2.33	0.50	enc_messageq_markempty
8.05	2.82	0.49	enc_cpu_tick
5.58	3.16	0.34	enc_messageq_push
	==== Mostly more GC and tracing ====		

Table 4: Ping using atomic closure.

% time	cumulative seconds	self seconds	name
8.11	0.15	0.15	enc_messageq_push
8.11	0.30	0.15	enc_messageq_markempty
7.57	0.44	0.14	enc_mpmcq_pop
7.57	0.58	0.14	enc_hashmap_get
7.30	0.72	0.14	enc_pool_alloc
	==== Mostly more GC and tracing ====		

Table 5: Ping using atomic mailbox.

6.2.3 Concurrent Sorted Linked-List

In the Concurrent Sorted Linked-List benchmark, a set of actors generates random integers which they send to a linked-list actor. The linked-list actor inserts the integer by the order of increasing value. This operation has an amortized worst-case time complexity of $\mathcal{O}(n)$. As a result, every insertion in the linked-list is going to progressively take more time.

The purpose of performing this benchmark is to contrast the overhead of bestowed objects in a program that runs a comparatively expensive operation for every message send. This will reveal whether or not there are scenarios where the overhead of bestowed objects is negligible in the overall performance of a program. In Listing 20 is a simplification of the benchmark.

In this iteration of the benchmark only one actor is spawned to send insert messages to the list actor. The bestow version of the benchmark (Listing 20) bestows the internal `List` object of the `ListActor`, which the `inserter` function

```
def inserter(n : int) : unit
  val l = new ListActor()
  var numMsgs = 0

  while (n > numMsgs) do
    l ! insert(random())
    numMsgs += 1
  end
end
```

```
local class List
  -- omitted
end

active class ListActor
  list : List
  def insert(i : int) : unit
end
```

Listing 20: Inserting random integers into a list actor, some method implementations are omitted.

instead sends messages to. In Figure 10 the result of the benchmark can can be seen, where each instance has been run 10 times and averaged.

The bestow version of the benchmark does on average run slower. This slow-down is proportional to the number of messages, but the overall performance of the benchmark is not severely affected by the use of bestowed objects. There is no break-even point for bestow like there is for atomic, but the overhead of bestow can become an issue in some scenarios. One such scenario arises when a larger number of messages are sent, and the work done by the target actor for each message is small.

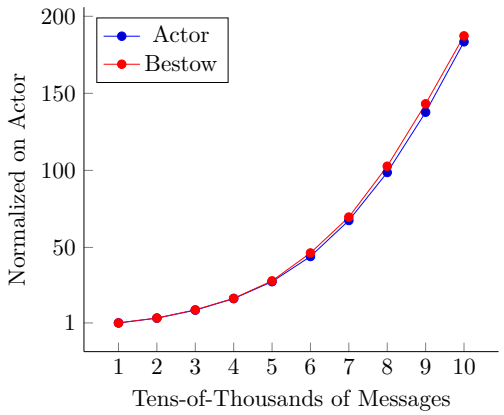

Figure 10: Comparing the Concurrent Sorted Linked-List benchmark with a version using bestow.

6.3 Discussion

Bestow is most useful to decouple an actor's interface with its internal objects and to produce more fine-grained interfaces. A weakness is when several internal objects needs to be bestowed. It could easily clutter the interface of the actor if

every internal object needs a method that returns a specific bestowed object. An array, heap, or some other convenient data structure could of course be used to improve this. Another weakness is if the interfaces of an actor and its bestowed objects overlaps. This conflict the programmer would have to resolve by renaming methods.

It is always obvious when atomic can be used to synchronize multiple messages. A lot of the times it can also improve patterns by replacing complicated synchronization mechanisms of the programmer by a simple atomic block on an actor. An example could be to use atomic to check that a list actor is not empty, and if so, retrieve the first element in the list. Perhaps surprising, is the fact that atomic can be used to produce more fine-grained interfaces as well. This can be accomplished by splitting up large methods into smaller parts, and using atomic allows an actor to interact with these new methods without any interleaving. As a result, atomic also makes it possible to react to intermediate results.

As for performance, abstraction often comes at the cost of additional overhead, yielding worse performance. In its current implementation and on the tested machine, atomic closure has a break-even point where it can yield better performance in cases where there are few individual atomic blocks and more message sends on the atomic target inside them. Testing has not yielded any case where bestow or atomic mailbox can be used to improve performance. Though the overhead of bestow can be negligible in computation intensive scenarios.

7 Related Work

Achieving actor isolation is commonly done by restricting aliasing across actors. Data is then shared by for example performing a deep copy or transferring the data's ownership [5, 12, 13, 14]. Enabling an external actor to safely interact with internal data of another actor allows for sharing of data and its internal structure without sacrificing isolation.

Bestowed objects are close in nature to the extension of an actor's interface in CEL languages. In E an actor is called a Vat, which has its own thread of execution called an event loop [3, 8]. All objects encapsulated inside the Vat can be used as an interface for interacting with the Vat. Bestowing an object creates a new point of entry for interacting with an actor, similar to eventual references in E. AmbientTalk also has event-loops for its actors, exporting an internal object of an actor is similar to eventual references in E and bestowed objects [3]. Eventual references in E and exporting an internal object in Ambient talk are different than bestowed objects in that they are not visibly bestowed to the programmer.

Another technique similar to bestow is based on function passing, where instead of passing data between entities, functions are sent to collections of data called silos where they are executed [15]. Bestowed objects could be used as an interface

to an internal part of the silo. This is similar to performing a message send on a bestowed object, as it is equivalent to sending a function (closure) to the owner to execute.

X10 introduces a language construct that is also called atomic for executing multiple operations in sequence without interleaving [16]. But the semantics are slightly different as no expression with side effects may be used, such as a blocking operation. And only local variables and methods can be accessed.

8 Conclusion

The integrated implementation of bestow into Encore shows that it is a useful tool when simplifying certain programming patterns by allowing an actor's internal objects to serve as an extension to its interface. This means that the interface of the actor and its internal objects can be decoupled and kept separate. The additional abstraction that bestow provides comes with a large overhead in its current form, fortunately there are ways to reduce this.

The two atomic implementations offer a built-in synchronization mechanism for the programmer that can both be used to simplify programming patterns and sometimes improve performance. The patterns that are improved are those where messages need to not be interleaved, and when splitting large methods and interacting with intermediate values is desired. Like bestow, both versions of atomic come with an overhead, but the testing suggests that atomic closure has a break-even point where the performance improvement of translating certain message sends to method calls outweigh the overhead of an atomic block. Reducing this overhead is also possible, which would increase the cases where atomic closure could yield a performance improvement.

8.1 Future Work

One major weak point in the bestow implementation is the large overhead of using the construct. There are multiple avenues to decrease the overhead, one strategy is to reuse closure environments if they are in the current scope, or move certain patterns inside the target actor. Because the closure environments of bestow operations are immutable, it is also possible to statically assign some variables.

Memory allocation and garbage collection works differently for certain parts of Encore. Messages are one such part, and elements like closure environments, and message queues could be allocated inside messages to reduce the garbage collection overhead. Bundling messages together is also a possibility if the number of messages can be specified or calculated. This bundling of messages can be used for both constructs to decrease the number of messages which need to be sent.

What the scope of atomic operations should be is not obvious. Three alternatives have been presented in this thesis: lexical scope, atomic reference, and actor wide. Only lexical scope has been implemented, but the two other alternatives would probably prove to be more useful in practice.

Both of the constructs have problems in their current implementation. The two major problems are the tracing and garbage collection algorithm for bestowed objects, and resolving race conditions in atomic mailbox when an actor is to be scheduled to run. The tracing problem has potentially been resolved, but due to time constraints its correctness has not been tested as rigorously as it could have been. A first step to solving the scheduling problem is to have the programmer specify the maximum number of atomic messages. This number could be stored as a counter in a message which gets decremented for each message handled, and if the counter is greater than zero it is moved to the front of the actor's message queue after it gets descheduled. This would resolve one type of race condition where messages sent to the atomic message queue no longer need to schedule the actor.

The work of Castegren and Wrigstad [4] showed that the idea behind bestow and atomic was sound. This thesis has built upon their work and both implemented the constructs and proved their usefulness in practice. Addressing a few problems of bestow and atomic would allow for their integration into the main branch of Encore, where more patterns where the constructs are useful can be discovered.

List of Listings

1 Hello World! . . . 4
2 List iterator breaking isolation . . . 7
3 Bestowed iterator safely breaking isolation . . . 7
4 Bestowed object implementation . . . 10
5 Sending a malicious closure . . . 11
6 Actor trait implementation . . . 11
7 The scope of atomic operations . . . 15
8 Generated C code for a closure . . . 17
9 Bestow expressions . . . 18
10 Desugaring bestow expressions . . . 19
11 Desugaring atomic closure . . . 21
12 Enabling closure mutability . . . 22
13 Generating C code for atomic mailbox . . . 22
14 Scheduling problem: example . . . 23
15 Scheduling problem: run-time race condition . . . 24
16 The sleeping barber . . . 26
17 Bestowing an iterator to improve patterns . . . 27
18 Ping benchmark . . . 29
19 Ping benchmark using atomic . . . 30
20 Concurrent Sorted Linked-List benchmark . . . 33

List of Figures

1 The Actor Model . . . 3
2 Actor isolation vs. bestowed objects . . . 8
3 Synchronization of bestowed objects . . . 10
4 Exclusive access with atomic . . . 12
5 Atomic closure and atomic mailbox . . . 13
6 Atomic mailbox's read and write pointers . . . 14
7 Example structure of a bestowed object . . . 19
8 Comparison: Ping benchmark with bestow and atomic . . . 29
9 Comparison: Ping benchmark with atomic . . . 31
10 Comparison: Concurrent Sorted Linked-List benchmark . . . 33

List of Tables

1 Gprof: Ping benchmark . . . 30
2 Gprof: Ping using bestow . . . 30
3 Gprof: Ping using bestow and atomic . . . 30
4 Gprof: Ping using atomic closure . . . 32
5 Gprof: Ping using atomic mailbox . . . 32

Bibliography

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer. "A View of the Parallel Computing Landscape". In: *Communications of the ACM* (2009). DOI: 10.1145/1562764.1562783.

[2] B. Goetz. *Java Concurrency in Practice*. Reading, Massachusetts, USA: Addison-Wesley Professional, 2006.

[3] J. De Coster, T. Van Cutsem, W. De Meuter. "43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties". In: *AGERE* (2016). DOI: 10.1145/3001886.3001890.

[4] E. Castegren and T. Wrigstad. "Actors without Borders: Amnesty for Imprisoned State". In: *PLACES* (2017). DOI: 10.4204/EPTCS.246.4.

[5] S. Srinivasan and A. Mycroft. "Kilim: Isolation-Typed Actors for Java". In: *ECOOP* (2008). DOI: 10.1007/978-3-540-70592-5_6.

[6] S. Brandauer et al. "Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore". In: *Formal Methods for Multicore Programming* (2015). DOI: 10.1007/978-3-319-18941-3_1.

[7] J. Armstrong. *Programming Erlang: Software for a Concurrent World (Pragmatic Programmers)*. Raleigh, North Carolina, USA: Pragmatic Bookshelf, 2013.

[8] M. Miller. "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control". In: *PhD thesis, John Hopkins University, USA* (2006).

[9] E. Castegren and T. Wrigstad. "Reference Capabilities for Concurrency Control". In: *ECOOP* (2016). DOI: 10.4230/LIPIcs.ECOOP.2016.5.

[10] M. Herlihy, J. Eliot B. Moss. "Transactional Memory: Architectural Support for lock-free Data Structures". In: *International Symposium on Computer Architecture* (1993). DOI: 10.1145/165123.165164.

[11] S. Imam, V. Sarkar. "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries". In: *AGERE* (2014). DOI: 10.1145/2687357.2687368.

[12] J. Armstrong. "A History of Erlang". In: *HOPL III* (2007). DOI: 10.1145/1238844.1238850.

[13] S. Clebsch, S. Drossopoulou, S. Blessing and A. McNeil. "Deny Capabilities for Safe, Fast Actors". In: *AGERE* (2015). DOI: 10.1145/b2824815.2824816.

[14] P. Haller and M. Odersky. "Capabilities for Uniqueness and Borrowing". In: *ECOOP* (2010). DOI: 10.1007/978-3-642-14107-2_17.

[15] H. Miller, P. Haller, N. Müller and J. Boullier. "Function Passing: A Model for Typed, Distributed Functional Programming". In: *Onward!* (2016). DOI: 10.1145/2986012.2986014.

[16] V. A. Saraswat, V. Sarkar, C. von Praun. "X10: Concurrent Programming for Modern Architectures". In: *Principles and Practice of Parallel Programming* (2007). DOI: 10.1145/1229428.1229483.