DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2018

Object Capabilities and Uniqueness for Isolating Actors in Akka

FREDRIK SOMMAR

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Object Capabilities and Uniqueness for Isolating Actors in Akka

FREDRIK SOMMAR

Master in Computer Science
Date: November 28, 2018
Supervisor: Philipp Haller
Examiner: Mads Dam
Swedish title: Objektförmågor och unikhet för isolering av aktörer i Akka
School of Electrical Engineering and Computer Science

Abstract

Large-scale concurrent systems need to ensure that the number of bugs is as low as possible, especially since the symptoms may appear far from the cause. Data races, for instance, is caused by the lack of isolation between systems. Akka is the de facto actor library, which is a common way to write large-scale concurrent systems, and is written for the statically typed language Scala. LaCasa is a compiler plugin for Scala that introduces object capabilities and uniqueness to Scala's type system, providing isolation and thus preventing data races. This thesis investigates compile-time isolation for actors by designing and implementing an adapter for LaCasa in Akka.

The adapter was developed as a set of alternative versions of Akka interfaces created in a similar package hierarchy, each wrapping their Akka equivalent. That way, the concepts from Akka carry over, while a new, *safer*, API is exposed. Users can either ensure all their messages are `Safe`, which is a marker that signifies deeply immutable classes, since deeply immutable objects have isolation properties by default, or resort to using boxes – LaCasa's core concept, providing object capabilities and uniqueness to isolate references. The adapter supports receiving both `Safe` messages and boxes from the same actor, where `Safe` messages are treated similarly to how regular Akka handles messages.

To evaluate the adapter, a selection of programs was picked from the Savina actor benchmark suite, capturing different kinds of actor applications. The results show that it is feasible to use the adapter for existing applications – when all messages are `Safe`, which was true for 6 out of 7 applications. It exceeds expectations set in the hypothesis; requiring on average 10% of code to be modified for `Safe` messages, and with an insignificant change in total lines of code. Furthermore, the performance impact was also shown to be insignificant with the introduction of LaCasa.

During the migration, several patterns were observed and documented in Chapter 6. Future work should be put into supporting more Akka patterns, automate the marking of Safe types and increase the convenience when using boxes.

Sammanfattning

Storskaliga parallella system måste se till att antalet buggar minimeras, speciellt eftersom symptom kan dyka upp långt ifrån orsakerna. I system som saknar isolering kan det lätt hända att processer tävlar om samma resurs. Akka är programmeringsvärldens de facto aktörsystem (ett vanligt sätt att skriva parallella program på) och är skrivet för det statiskt typade språket Scala. LaCasa är en insticksmodul för Scalas kompilator som introducerar objektförmågor och unikhet, och kan genom det bistå med isolering av objekt, och på så sätt förhindra det ovannämnda resurstävlandet redan innan ett program kör. Det här examensarbetet undersöker isolering för aktörer under kompilering genom att utforma och implementera en adapter för LaCasa i Akka.

Adaptern utvecklades som en uppsättning alternativa versioner av Akkas gränssnitt, där varje version sparar en dold instans av Akkas motsvarighet till gränssnittet. Gränssnitten är skapade i en liknande pakethierarki, och på så sätt överförs koncepten från Akka, medan ett nytt, säkrare programmeringsgränssnitt exponeras. Användaren kan antingen försäkra sig om att alla dess meddelanden är `Safe` (klasser med oföränderliga data), eller utnyttja lådor – LaCasas kärnkoncept som tillhandahåller objektförmågor och unikhet. Objekt med oföränderlig data har inbyggda isoleringsegenskaper, medan lådor bistår med ett mer strikt system för isolering. I en aktör går det att ta emot både `Safe` meddelanden och lådor. Meddelanden som är `Safe` behandlas på samma sätt som ordinarie Akka hanterar meddelanden.

För att utvärdera adaptern valdes ett urval av program från Savina jämförelseindex, som inkluderar olika varianter av aktörprogram. Resultaten visar att svaret på problemformuleringen är ja: det är möjligt att använda adaptern för befintliga applikationer – när alla meddelanden är `Safe`, vilket är fallet för sex av sju program. Det överträffade förväntningarna i hypotesen; i genomsnitt krävs det att 10% av koden ändras i de fall alla meddelanden är `Safe`, samt en obetydlig förändring av antalet rader kod. Utöver det visades även att prestandan förändrades obetydligt med introduktionen av LaCasa.

Under migreringen observerades flera programmeringsmönster, som dokumenterats i kapitel 6. Framtida arbete bör fokusera på att stödja fler av Akkas programmeringsmönster, automatiskt markera meddelanden som är `Safe`, samt att öka användarvänligheten när det kommer till användning av lådor när meddelanden inte är `Safe`.

Contents

Listings

List of Figures

List of Tables

Chapter 1

Introduction

An important aspect when writing code is to limit the number of bugs. Static typing (checking the types of variables at compile time) is a feature in many popular languages and prevents *some* classes of bugs at compile time. Limiting the number of bugs is especially important in concurrent programs where the symptoms may appear far from the cause [25]. A common way to write concurrent programs is with actor libraries, where Akka [17] is the most well known, made for the statically typed language Scala, and is used in production at large multinational companies, including Amazon, Ebay, Blizzard, and Cisco.

Consider the concurrent actor program in Listing 1.1, written in Scala using Akka; it has a bug that cannot be found with conventional type checking. References to the `Data` object are sent to the two actors, *aliasing* the object (more than one reference pointing to the same object). Since the object is mutable, both actors can freely mutate it simultaneously. This breaks the expected behavior of actors, since mutability should not be observable outside an actor. The symptom is a data race due to the actors not being properly *isolated*.

Conventional statically typed programming languages, including Scala, cannot enforce compile-time isolation. (However, some newer languages, e.g., Rust, can – see Chapter 3.) LaCasa [12] is a compiler plugin and library for Scala that provides functionality for compile-time isolation, using object capabilities and uniqueness (more on that in Section 2.3). A trivial way to provide isolation is to use deep-copy semantics (i.e., copy all of the underlying data when sending it) whenever references are passed around, but that is not performant, since deep copies are expensive [23, 20].

```
case class Message(var counter: Int)
class A(n: Int) extends Actor {
  def receive: Receive = {
    case d: Message =>
      d.counter += 1
      println(s"actor$n: ${d.counter}")
  }
}
def main(args: Array[String]): Unit = {
  val system = ActorSystem("app")
  val msg = Message(0)
  system.actorOf(Props(new A(1))) ! msg
  system.actorOf(Props(new A(2))) ! msg
}
```

Listing 1.1: A non-isolated actor example using Akka in Scala. The output is non-deterministic due to the data race when incrementing the counter, and can print, e.g., that the counter is 2 from both actors.

One goal of this thesis is to enable compile-time isolation, that is performant, for Akka actors by designing and implementing an adapter for LaCasa in Akka, and to minimize the changes necessary to use the adapter in existing Akka applications. With isolation, classes of memory-related bugs, including data races, can be eliminated. Moreover, scaling actors from a local to distributed level is trivial, as their behaviors are united with isolation, but that is outside the scope of this thesis. Another goal is to provide an empirical study that evaluates the effort required to apply LaCasa to actor-based programs.

Preliminary results of this thesis have been published in the proceedings of the *Tenth Workshop* on *Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES@ETAPS 2017)* [14].

1.1 Problem statement

The thesis aims to answer the research question: *is it possible to design and implement an adapter that integrates LaCasa into Akka, and thus enable isolation for Akka actors?* If the answer is yes, *what effort is required, in terms of code changes, to convert existing Akka programs to*

use the LaCasa-Akka adapter? And, *what is the performance impact of using the adapter instead of pure Akka?*

Hypothesis The hypothesis is that for most projects it will be necessary to do major refactorings to support the new requirements that LaCasa will exert on the surrounding code. For some projects, however, the necessary changes might be smaller in scope, e.g., if a strict coding standard was followed in an attempt to minimize memory-related bugs.

Evaluation The answer to the research question will be evaluated by creating an adapter connecting LaCasa and Akka. An empirical study will be conducted on existing Akka applications by migrating them to use the adapter and thereby examine the effort required and whether it is feasible to use the LaCasa-Akka adapter. Its performance will be evaluated to determine if and when it can be used as a replacement for pure Akka.

1.2 Contributions

This thesis contributes a system for enforcing isolation in Akka, and more generally for an existing actor library. Users of Akka can take advantage of a compile-time isolation to eliminate classes of memory-related bugs and thus have a more stable system that should be easier to reason about.

Another contribution is an empirical study evaluating the effort of introducing isolation to the type system in existing applications, which is new. The migration effort is studied on 7 existing Akka programs from the de facto standard actor benchmark suite, Savina [15]. The work builds on LaCasa, a system that extends Scala with isolation types, and together with Akka, it enables a new type of empirical study. Previous work are either new languages or new actor libraries (see Chapter 3), and therefore a study on both an existing language (Scala) *and* an existing actor library (Akka) has not been done before. Furthermore, the performance impact of the migration is measured, and the programs are well suited for it since they are from a benchmark suite.

Since the initial results were published [14], more programs have been added and the convenience has been improved (in terms of required changes on an existing code base) much thanks to the new concept *Typed Signature Adapter*, TSA. Additionally, the performance has been evalu-

ated (see Section 5.3), and migration patterns have been identified (see Chapter 6).

1.3 Sustainability, ethics and societal aspects

The content presented in this thesis does not provide anything additional on top of LaCasa nor Akka that can be considered changes to their ethical or societal impacts. It could be argued that the combination of removing classes of memory-related bugs, and simplifying actors to reduce the risk that a system uses resources in vain, provides a positive impact to sustainability in terms of using computing resources; an automatically scaling system with memory leaks could potentially ask for increasingly more resources that it would not utilize. The results presented in this thesis heavily reduce that risk.

1.4 Outline

In Chapter 2, the necessary basics for Scala, LaCasa and Akka are presented. Chapter 3 covers existing systems, including languages, with actor libraries or similar concurrency models, that enable language-wide static guarantees (for isolation and more). Details of the LaCasa-Akka adapter implementation are found in Chapter 4, the empirical study as well as performance benchmarks in Chapter 5 and identified patterns that can simplify migrating existing applications in Chapter 6. Finally, Chapter 7 highlights future improvements and summarizes the thesis with a conclusion.

Chapter 2

Background

This chapter is split into three sections, the first describing Scala and its type system with the knowledge required to understand coming parts. Following Scala, there is a section on actor systems and Akka. Finally, the third section introduces LaCasa and describes how it enforces uniqueness and isolation in a language without built-in support for it.

2.1 Scala

Scala is an object-functional language that runs on the JVM, with Java interopability [21]. Scala makes heavy use of traits, which are similar in nature to interfaces (found in, e.g., Java). Traits provide a way to define templates that other types can implement.

2.1.1 Syntax

The Scala syntax is familiar to anyone with experience in a C-based language. However, there are still many differences; semicolons are optional and methods can be called in an operator-styled infix way. Thus, the `+` operator is simply defined as `def +(other: Int): Int` for `Int`, and similarly for other types. If the only argument is a function, it can be defined as a block following the method call, ignoring the parentheses all together.

Scala uses `val` and `var` to declare variables, with the former indicating a shallowly immutable variable. Because of local type inference, – i.e., the compiler infers what type a variable should be based on how it is

used – types rarely need to be explicitly declared. If they do, they are in a suffix position, following the variable name with a colon and the type.

When defining and using generic type parameters, square brackets are used, as opposed to angle brackets which are normally used in C-based languages. Corollary, accessing items in a list uses normal parentheses instead of square brackets. Any class can make use of the parentheses access method by overriding the `apply` method.

Since classes carrying only data is a common, and encouraged, pattern in Scala (cf. *Plain Old Java Objects* in Java), there is special syntax sugar for them: case classes. Fields in a case class are public and **`val`** by default, i.e. immutable. Comparisons between two case classes are done on structure (the data itself) and not reference, and they provide default implementations for the aforementioned `apply` method.

A collection of usages with the mentioned syntax can be found in Listing 2.1.

```
case class User(name: String, age: Int)

def main(args: Array[String]) = {
  var unitialized: Int
  unitialized = 5
  val u = User("Fredrik", 25) // using case class apply
  val someValue = 1 + unitialized // 1.+(unitialized)
  val myList = List("a", "b", "c")
  println(myList(1)) // b
}
```

Listing 2.1: A declaration of the main method with example expressions demonstrating Scala's syntax.

2.1.2 Type system

Scala's traits are very similar to classes as they can contain fields and implementation of methods. Like interfaces in Java, multiple traits can be extended by a class, but traits are more expressive than interfaces.

A trait or class is normally invariant with respect to its generic type parameters. Consider a container type `C[T]` and the class `Animal` with a subtype `Dog`, expressed as `Dog <: Animal`. If `T` is invariant, there is no relation between `C[Dog]` and `C[Animal]`. However, when the container

is covariant with respect to `T` – indicated as `C[+T]` at point of declaration – the relation `C[Dog] <: C[Animal]` is implied. Corollary, if the container is contravariant with respect to `T` – indicated as `C[-T]` at point of declaration – the relation `C[Animal] <: C[Dog]` is implied.

Implicits A method can declare implicit parameters [21]. Calling such a method will automatically resolve the implicit argument if an implicit value of that type exists in scope. An explicit argument can be provided, and will override the implicit value in scope. It is important to know that there is an order in which implicits are resolved, thus determining which implicit shadows which.

Using implicit parameters, and implicits in general, is mostly a matter of convenience, to reduce the number of parameters that need to be passed - especially if there usually is a reasonable default value. Implicits are realized solely based on their types, and therefore using primitives and other common types, e.g. `String`, for implicits is discouraged. Instead, consider wrapping the common types in a case class – see Listing 2.2 for an example usage.

```
case class Sender(name: String)

implicit val sender = Sender("Alice")

def postMessage(msg: String)(implicit from: Sender) = {
  println(s"${from.name}: $msg")
}

postMessage("hello, implicits") // "Alice: hello, implicits"
```

Listing 2.2: Wrapping a value in a case class for use with implicits.

Implicits can be used to evidence a fact about a type [21]. If a trait is declared with a single generic parameter, e.g. `Safe[T]`, and a method requires `Safe[T]` to exist for some `T`, an implicit parameter to the method can be added, requesting a value of type `Safe[T]` (see Listing 2.3). This is normally referred to as a *context bound*.

```
def tIsSafe[T](value: T)(implicit ev: Safe[T])
```

Listing 2.3: Requesting an implicit evidence of a type.

Because this use case is rather wide spread and has many uses in Scala, there is an alternative syntax sugar version which is arguably easier to understand – especially for end users of a library that do not want to be exposed to implicits. See Listing 2.4 for how the previous method would look like using this sugar.

```
def tIsSafe[T: Safe](value: T)
```

Listing 2.4: Requesting an implicit evidence of a type, using syntax sugar.

Partial functions Pattern matching is an integral part of Scala [21]. With pattern matching it is possible to deconstruct the values being matched by, e.g., extracting values from case classes. As such, matches are a very powerful tool. A partial function often leverages pattern matching to simplify its definition.

```
def foo: PartialFunction[Any, Unit] = {
  case User(name, age) =>
    println(s"$name is $age years old")
  case x: Int if x > 0 =>
    println(s"$x is a positive integer")
}
```

Listing 2.5: An example declaration of a partial function accepting `Any` values.

Consider the partial function in Listing 2.5 which accepts values of type `Any`, the superclass of all classes in Scala. The function is only defined for the `User(name, age)` case class, and positive integers. For input that is not defined in the match-like construct, the partial function returns `false` for its `isDefinedAt` method. Partial functions can be combined, using the `orElse` method. It accepts a function as argument, and is run when the partial function is undefined for the input values.

Type members Classes and traits may declare type members, which are an expressive complement to generics [21]. In an abstract position, i.e., in an abstract class or trait, type members can have type bounds, either upper, lower or equal. In the case of equality, the type member is a type alias. Listing 2.6 gives an example declaration of a trait with two type members, where one has an upper bound.

```
trait Graph {
  type Node <: Number
  type Edge

  def neighbors(n: Node): Iterator[Edge]
}
```

Listing 2.6: Example usage of type members.

In non-abstract classes extending from a trait or abstract class with abstract type members, the type members need to be assigned. For instance, a subclass of `Graph` could have `type Node = Int` and `type Edge = (Node, Node)` or any other types, as long as they oblige to the type constraints.

Type members without constraints in a non-abstract class are dependent on the instance they are contained in, meaning that an instance of a class with unconstrained type member `C`, does not have the same `C` as another instance of the same class [21].

In type position, type members can be specified via refinement as follows: `CanAccess { type C = box.C }`. Refinement indicates that the type member of the passed class needs to be equal to the provided type. In this case, `C` is dependent on the type member `C` of value `box`, and is an example of a path-dependent type.

2.2 Actor systems

The actor model formally describes how actor systems, a concurrent way of programming, can and should behave. Fundamentally, the model works by having processing units, called actors, send and receive messages between each other [1]. An actor has three different behaviors at its disposal: it can a) send messages to other actors, b) create new actors, and c) decide how to handle the next message – usually referred to as an actor's behavior. Actor systems manage the scaffolding around the messages and actor spawning, enabling actors to perform their actions. They are the entry point for the actor model.

Scala has had an implementation of actors in the standard library since 2006, Scala Actors [10]. Scala Actors founded the basis for adopting actors into mainstream software engineering; since 2006, a number of new actor libraries have appeared, many inspired by Scala Actors – directly

or indirectly. In 2013, with Scala 2.10, Scala Actors was deprecated in favor of Akka, an actor library directly inspired by it.

2.2.1 Akka

Akka is the de facto actor library for Scala [17]. It was originally developed and designed for Scala, but has inspired other actor libraries and spread to, e.g., C# and F# on the .NET platform [2].

This section will introduce core concepts in Akka: `Actor`, `ActorRef`, `ActorContext` and `ActorSystem`. Knowledge of those concepts is enough to do most things in Akka.

Actor An actor has state and behaviors. In Akka, the state can be mutable and will be protected from aliased access (i.e., two or more references to the same object) – as long as the state is limited to its actor. Akka's actors are single-threaded, and can therefore freely allow mutable state. If state needs to be shared between actors, it should not be done through aliasing its reference, but rather through message passing to the other actor. However, sending objects is also a way to alias them, – since they are passed by reference – and should be avoided as much as possible, and preferably not be used at all. Recall from the introduction that aliasing mutable state is a source of bugs, thus the need for isolation between actors. Avoiding aliasing of references between Akka actors is the primary goal of this thesis and is explored at depth in later sections.

Actors in Akka are classes, and are declared via inheritance. A class extends `Actor` and implements `def receive: Receive` where `Receive` is a type alias of `PartialFunction[Any, Unit]`. In other words, `receive` is a partial function which accepts anything and returns nothing (`Unit` is the no-value type in Scala, cf. `void` in Java). The convention is to declare `receive` using the match construct shorthand for partial functions. See Listing 2.7 for an example declaration.

In Akka, an actor's behavior is, in simplified terms, its `receive` method. The method decides what to do with incoming messages and handles most of what the actor does. Additionally, actors have the functionality to temporarily *become* a new behavior. In practice, `receive` can dynamically be overwritten by another partial function with different behavior.

Messages sent to an actor are enqueued in the actor's mailbox. When messages are sent from the same actor, their order is guaranteed to be

```
case object Increment
case object Decrement
case object Get
case class Set(n: Int)

class MyActor(var number: Int = 0) extends Actor {
  def receive: Receive = {
    case Increment =>
      number += 1
    case Decrement =>
      number -= 1
    case Set(n) =>
      number = n
    case Get =>
      sender ! number
  }
}
```

Listing 2.7: A minimal Akka actor with state. It is in effect a concurrently safe wrapper around an integer.

the same. However, there is no guarantee that messages sent from two different actors will arrive in their chronological order. Actors read and apply each message in the order it arrived in their mailbox, in a single-threaded manner. This is key to allowing the state inside an actor to be treated as if it was non-concurrent.

Akka actors follow the paradigm of failing fast and recovering early. Instead of manually trying to recover the state when something goes awry, the recommendation is to crash and let the actor system reboot the actor with a clean slate.

Actor reference The actor example in Listing 2.7 does nothing on its own; it needs to be sent messages. First, a reference to the actor needs to be acquired (see below for different ways to do that), and then the actor reference can be used to send messages to that actor. Normally, the operator `!`, also available as `tell`, is used to send messages. For most use cases, the tell operator suffices, but there are other methods, like `forward` which proxies the message for another actor to handle, or `ask` – usually in the form of the operator `?` – which sends a message and then asynchronously waits for a response.

Actor references, of type `ActorRef` in Akka, are immutable references to actors and do not carry state. They are designed to be safe to alias and pass around. The actor reference inside an actor can be acquired via the `self` field. To communicate with an actor, the reference is needed. The `sender` method on an actor retrieves the sender for the last received message, allowing the actor to send back messages to its sender without having a stored reference to it. This is most often used in conjunction with the aforementioned ask pattern. In fact, Listing 2.7 uses `sender` to return its state to an actor requesting it.

Actor context An actor can acquire its own actor reference through its context, which is of type `ActorContext`. Every actor has access to a context, from where they can get their actor reference as well as the underlying actor system. Additionally, in the cases where a sender of a message is recorded, that is also acquired via the context. The context is an actor's handle of the outside system, and is reached from within an actor via the `context` field.

Actor system An actor system, of type `ActorSystem` in Akka, is responsible for starting and connecting actors. The system assigns names, and addresses, to actors started in the system, allowing them to be reached either by address or reference. In Listing 2.9, the previously declared actor is created with an initial number of 5, and then both the tell and ask operators are used. Notice that the initialization of `MyActor` is done via the `Props` class. The `Props` instance stores a template to create and start an actor, and can be used multiple times to start different instances of an actor. That way, the actor system has full control of when and where actor instances are created, and can completely handle their lifecycles.

In this case, the state can only be the number 6. Normally, however, messages are sent from more than one place, and there are more than one actor running at a time; there would be nothing preventing another actor from sending, e.g., a `Decrement` before the `Get`.

Actor systems in Akka are heavyweight, maintaining all of the actors and the threads they operate on. There should be one actor system per logical application, i.e., usually one actor system per application. The systems are not limited to running on one single machine, and can be – when configured – distributed and run remotely from other machines. For instance, see Listing 2.9 where the reference to a remote actor system's

```
def main(args: Array[String]): Unit = {
  val system = ActorSystem("my-actor-system")
  val myActor: ActorRef =
    system.actorOf(Props(classOf[MyActor], 5))
  // The following messages will arrive in order,
  // despite being sent asynchronously.
  myActor ! Increment
  // The ask pattern is used with foreach to asynchronously
  // receive the response and assert its value.
  (myActor ? Get) foreach { number => assert(number, 6) }
}
```

Listing 2.8: Declaring and using an actor system in Akka.

actor is acquired, and then used as if it was a regular local actor reference. Remote actors, however, are outside the scope of this thesis. Naturally, the aforementioned issues of relying on mutable references via messages become even more apparent as the messages are sent remotely. With that kind of restriction, going from a local to remote setup is non-trivial.

```
val myActorRemote =
  context.actorSelection(
    "akka.tcp://my-actor-system@10.0.0.1:2552/user/MyActor")
// Assuming that this is the same ActorRef as before,
// this should give the same result.
(myActorRemote ? Get) foreach { number => assert(number, 6) }
```

Listing 2.9: Getting a remote `ActorRef` from a running actor system.

2.3 LaCasa

Haller and Loiko [12] introduced a formal model for LaCasa – a system in Scala based on object capabilities (more on those below) and lightweight affine types. An affine type can be used *at most* once. Once a variable of an affine type is used, it is *consumed*, and cannot be used again. With the help of affine types it is possible to enforce uniqueness.

LaCasa's core concept is the `Box` class, which encases messages with object capabilities. Once inside a box, a message cannot be retrieved unless the box is opened. Several restrictions are placed on opening

boxes, some of which are enforced by the use of spores, and others by a compiler plugin. This section details these components and how they form LaCasa.

2.3.1 Object capability safety

Object capabilities list the allowed operations on an object. An object is object capability safe if a) its fields are object capability safe, b) methods only use parameters and `this`, and c) methods only instantiate object capability safe objects [19, 13]. This excludes, e.g., classes that access a global singleton, since that enables leaking object fields. Object capability safety facilitates isolation of objects.

Scala's type system is expressive enough for a subset of LaCasa's features, but not all of them. Fortunately, the Scala compiler has a plugin system that allows hooking into the compiler's internals, even during type checking. LaCasa uses the plugin system to extend the type system to check for object capabilities at compile time. Only object capability safe types are allowed for use in boxes, in order to isolate their contents. That, together with the fact that boxes are forced to be unique allows LaCasa to provide means for software isolation of references.

Safe Deeply immutable data-only (without methods) types are object capability safe, and even safer than that; they do not need the box encapsulation because they have no mutable state to protect. In fact, all of Scala's built-in primitives are deeply immutable. As evidenced by [11], almost all case classes found in the Scala standard library and a selection of other applications are deeply immutable. Recall that case classes are often used when a plain data-carrying object is needed. Since case classes are (shallowly) immutable by default, they are considered deeply immutable if all their fields are immutable as well, e.g., built-in primitives like integers and strings.

Because deep immutability is not expressed by Scala's type system, relevant types need to manually be marked as such. Enter `Safe[T]`, a marker trait for deeply immutable types considered safe to pass around. A type, `T`, is marked as `Safe` by having an implicit value of `Safe[T]` in scope (see Section 2.1.2).

2.3.2 Spores

Spores are an extension on top of closures – anonymous functions that capture values from the environment – and can encode restrictions of what can be captured by the closure at type level. They are vital for LaCasa to work, to prevent the wrong types from being captured at compile time.[1] In some cases, a closure (or anonymous function) can be implicitly cast to a spore, thus requiring no additional input from the user. For the remaining cases, every captured variable needs to be explicitly declared, using the spore macro. See Listing 2.10 for an example invocation of a spore capturing two variables and accepting one argument.

```
spore {
    val capture1 = var1
    val capture2 = var2
    msg => ...
}
```

Listing 2.10: Declaration of a spore with two explicitly captured values.

Currently, any types that are `Safe` are considered safe to capture in LaCasa. In theory, `Safe` types are a subset of the types that should be safe to capture, so it might be possible to relax the requirements in the future.

2.3.3 Boxes

In LaCasa, encapsulation of messages are handled by the `Box[+T]` class. It has a single field that saves the encapsulated content, and a type member `C`. The type member is never explicitly instantiated, and therefore every instantiation of `Box` gets an anonymous unique type for `C`. See Listing 2.11 for an example method signature where this fact is leveraged to create an access permission system, via an empty class `CanAccess` with a single type member `C`. Then, any method that receives a box can also receive that box's specific access permission through the use of path-dependent type refinement. Note that access permissions are expected to be implicit, allowing increased convenience when calling methods that receives a box and its access permission.

[1] In the implementation used in this thesis, spores are omitted and replaced functions, since there were issues with the spore dependency.

```
def foo(box: Box[Any])(implicit acc: CanAccess { type C =
    box.C })
```

Listing 2.11: The required parameters for a method to receive a box and be able to use it – via its access permission.

Internally, boxes and their permissions are wrapped inside a custom class called `Packed`, with the purpose of simplifying passing around a box with its permission. It is also used in some user-facing methods when, e.g., constructing a box.

To ease the cognitive load of remembering the access permission syntax, the `Box` class provides a type alias. Using the type alias, the purpose of the method signature becomes more obvious, as seen in Listing 2.12.

```
def foo(box: Box[Any])(implicit acc: box.Access)
```

Listing 2.12: Using the box's type alias for the access permission to simplify the method signature.

Creating boxes Boxes can be created via a family of functions, all prefixed with `mkBox`. A function in the family accepts a spore and creates a new instance with the same type as the type parameter on the function. See Listing 2.13 where the usage is exemplified by `mkBoxFor`, one of the functions in the family. Within the spore, access is given to the box of the created class instance via a `Packed[T]` argument; without the access permission, the box would be useless as nothing would be able to access its contents. Accepting a function to continue the execution is called continuation-passing style (CPS).

```
case class Data(var counter: Int)

mkBoxFor(Data(3)) { packed =>
  implicit val access = packed.access
  val box: Box[Data] = packed.box

   ...
}
```

Listing 2.13: Creating a box by using one of the `mkBox` family of functions.

Accessing the contents In order to access a box's content, it needs to be opened. Like `mkBox`, the `open` method accepts a spore. Other than spore restrictions, there are additional limitations on what can be accessed from inside the spore; no top-level classes can be accessed from inside the scope, since that could enable leaking the mutable box contents. Inside the spore, however, the content can freely be accessed and mutated.

Consuming boxes Once a box has been created, any reads or writes to the captured values on stack would break the encapsulation, and remove the purpose of boxing the values. LaCasa solves this by preventing any further access to the stack after box creation by utilizing the Scala construct `ControlThrowable` together with CPS, as demonstrated in Listing 2.14. `ControlThrowable` is a built-in Scala concept for exception-handled control flows, and is effectively an uncaught exception used for controlling program flow, enabling LaCasa to provide runtime-enforced affine types; a box is consumed when the exception is thrown. Because of the exception-handled control flow, any time boxes are consumed it needs to be done in a controlled environment – in practice, the exception needs to be captured before it propagates upwards and kills the application.

The easiest way to be compliant to the demands on the environment is to use the `Box.unsafe` function and pass the body that needs the environment, i.e., because it is consuming a box. As the name implies, using the method is unsafe, and should rarely, if ever, be used outside writing libraries. It is an escape hatch that can be used when no other tools are usable.

Methods that consume a box need to stop execution after the fact. If execution needs to continue, e.g., due to a transformation, then CPS should be used; the method accepts a function and provides the transformation as an argument. Users can create their own methods by ensuring that they return `Nothing` (Scala's bottom type, a type that cannot be instantiated) and call the `consume` method on the box – or call another method that consumes it. The `consume` method ensures that the correct `ControlThrowable` instance is used to satisfy the compiler plugin.

Stack confinement Once an object is placed on the heap, it is no longer possible to prevent the aliasing of the object. For boxes and access permissions, they are required to be unique and maintain affinity, and therefore LaCasa provides a compiler plugin that confines them to the

```
case class Data(var counter: Int = 0)
mkBox[Data] { packed => // Packed[Data] is provided in CPS
  implicit val access = packed.access
  val box: Box[Data] = packed.box
  // opening a box does not consume it
  box open { x => x.counter += 1 }
  box open { x => x.counter += 1 }
  box open { x => println(s"Counter is $x") }
  // prints "Counter is 2"
}
// mkBox throws ControlThrowable and therefore
// any code here will not be executed
```

Listing 2.14: Constructing a box and modifying its contents.

stack. If an attempt is made to leak either a box or its access permission to the heap (typically a field), the compiler plugin emits an error.

Chapter 3

Related Work

There are two likes of work related to what is being investigated in this thesis. The first is languages which have built-in support for features similar to what LaCasa provides. The second is extensions on top of languages that do not normally include LaCasa-like functionalities. In both cases, it is important that they have access to actor libraries or equivalents. Finally, there is a section on empirical studies that evaluate the experience of writing applications with isolation-like features.

3.1 Languages with uniqueness and capabilities

Languages relevant to this thesis, with built-in support for uniqueness and capabilities in the type system, are listed below. Because these concepts are core functionalities of these languages, there is no previous work that compares the before and after their introduction, nor is there empirical studies on the effect of it. There are, however, subjective studies done on how the use of the increased safety is perceived by the developers. A summary of relevant reports can be found in Section 3.3.

3.1.1 Pony

Pony is a research language whose type system has capabilities integrated as a core component [7]. Every location of a type has an associated capability. In Pony's standard library, almost 90% of the types are not explicitly labeled with their capability, as the compiler can infer them on its own and from referring to the default capabilities for specific types.

In the remaining cases, however, users need to explicitly declare the capability.

LaCasa, in contrast, does not explicitly label any capabilities, since it is implicit that any types inside a box need to be object capability safe. In a way, the box is a capability label.

Deny capabilities Pony provides six kinds of deny capabilities. Recall that object capabilities list the allowed operations for an object. The opposite is true for deny capabilities; they list the denied operations. For Pony, the different operations that get denied are read and write aliases, on either a local or global level. Any given capability therefore has the option to deny between none and four operations.

The six kinds of deny capabilities in Pony are called `iso`, `trn`, `ref`, `val`, `box`, `tag`. The default for classes is `ref`, denying global read and write aliases. For primitives, the default capability is `val`, which denies local and global write aliases, but lets them be read.

Beyond local and global read and write aliasing, there is actually a fifth type: aliasing `tag`s. No matter the underlying capability, retrieving a type as a `tag` is never denied; it is only used as an identity and the tag itself denies all types of aliasing.

Built-in actors The default deny capability for active objects that are declared as `actor`s is `tag`. The `actor` keyword is built-in, and actors are supported directly in the type system. On an `actor`, behaviors can be declared, using the keyword `be`. With the `tag` capability, only behaviors are allowed to be invoked on a `tag` alias, as neither reads nor writes are allowed on the alias. In fact, its approximate parallel in the Akka world is an `ActorRef`, which essentially only allows message sending – c.f. behaviors – and does not allow any direct mutation of the actor itself, containing the state.

3.1.2 Joëlle

Clarke et al. [6] introduced a Java-like language, Joëlle, with ownership semantics. Its concurrency model is based on active objects, and not the actor model, but they are similar enough that it generalizes. Similarly to Pony, Joëlle provides annotations that dictate the type system. Unlike Pony, however, Joëlle is based on an effects system, and not on capabilities.

The authors highlight issues with sharing references in a concurrent environment, and mention different approaches to simplify reasoning about it: immutable data, deep cloning, ownership and uniqueness.

Annotations LaCasa uses a combination of ownership (a box can only be owned by one entity at a time) and uniqueness, and in Joëlle, the user can choose between immutability or uniqueness with annotations from its effects system. The effects relevant to LaCasa are `unique`, `immutable` and `safe`. On instantiation, an object reference can be declared as `unique`, and an `immutable` object can only be created from a unique reference.

Compare this to LaCasa, and Scala, where immutability is default for case classes, but a case class is not required be immutable. The immutability, in this case, is on class-level (c.f. `Safe`), not on an object-level. Unique references, however, are created on object-level, since an object itself is packed into a box, and LaCasa enforces the uniqueness of that box.

Methods can, in Joëlle, be labeled with effects, thereby restricting what is allowed inside the function; e.g., a method labeled `safe` can only access final fields with `immutable` or `safe` references. Additionally, there is a `read` effect, that allows observation of mutability, which `safe` does not.

Joëlle allows transferring of ownership of unique references, and values that are either safe or immutable – like LaCasa does for boxes and `Safe` values. Other types of objects need to be deeply copied.

The authors applied Joëlle's ownership annotations to a 20-line program, modifying 6 lines in the process. This is the closest to an empirical study on Joëlle's annotations that exists, and corresponds to a change of 30% of the source code. (The authors mention that, since the program is small, the ratio of modified lines is not indicative of a real program.) Furthermore, this is done on an example originally written in Joëlle, and is therefore not applicable to larger existing applications since they need to be written in Joëlle from the start.

3.1.3 Encore

Loosely based on Joëlle, Encore is a programming language that is being designed with parallelism first in mind [4]. Encore has three different groups of capabilities, defined on a class-level, i.e., not on an object-level

like some of the effects in Joëlle. The capability classes are `exclusive`, `shared` and `subordinate`. A class with an `exclusive` capability is restricted to a single thread, – similar to contents of an Akka actor – while a `shared` class is shared between threads and therefore needs some other way of protecting its mutability. Finally, a `subordinate` class relays its mutability handling to some active class (c.f. actors in Akka owning data) and is therefore free of data races, just like the `exclusive` capability.

The `shared` capability has a number of subcapabilities that have different characteristics, among them immutability, atomicity and lock-freeness. Those are considered safe, but there is also additionally a subcapability that is considered unsafe, that can optionally be enabled, e.g., when interfacing with C code.

A library writer can request that, e.g., a method parameter should conform to one of the capabilities, ensuring that they can use the value without risking data races. In order to ensure that a variable will not be used again, i.e., affinity, there is a `consume` keyword, which corresponds to a destructive read – like in LaCasa.

3.1.4 Rust

Mozilla Research started a project in 2008 to develop a web engine, Servo, with focus on parallelization and memory correctness. Rust the language was created for that purpose, and in contrast to the aforementioned languages, Rust is not garbage collected. Instead, Rust relies on its affine type system to statically ensure that memory is freed. With the help of uniqueness and affinity, Rust can ensure memory safety, effectively eliminating memory leaks and corruption.

Type system Rust has a type system with support for affine types, and uses a system called borrowing and lifetimes to enforce uniqueness, and by extension that memory is correctly freed. This type system is quite strict and Rust therefore presents an escape hatch in form of the unsafe keyword – similar to Joëlle. Anything within the unsafe scope is not checked for soundness by the compiler, and the user guarantees that it maintains the rules and invariants defined by Rust.

In addition, Rust uses marker traits to label which objects are safe to send between threads – similar to how `Safe` is a marker trait for LaCasa. Via the type system, it is then possible to statically enforce an absence of data races, as it is impossible to have aliased mutable pointers to the

same memory area. It is, however, possible to have aliased *immutable* pointers as they only provide a view into the memory – c.f. `Safe` types being aliasable in LaCasa.

There are actor libraries in Rust, one of the most popular being Actix, which includes a web framework of the same name that is implemented using the actor library [26]. No publications have been made on Actix or other actor libraries in Rust. Therefore, Rust is mostly relevant for its affine type system and the similarities to LaCasa it has.

3.2 Other applications of isolation types

In some cases, existing languages have been extended with isolation-like functionalities, like LaCasa is an extension on top of Scala. There are no empirical studies on their introduction, which is one of the contributions of this thesis.

3.2.1 Kilim

Kilim is a Java library that includes a post-processing tool called Weaver, introduced by Srinivasan and Mycroft [23]. Together, they provide an actor system with isolation, similar to the goals of this thesis. Weaver is an extension on top of Java (c.f. LaCasa on top of Scala), but is a separate tool to the Java compiler that runs after the program has been compiled to byte code. The analysis runs on the byte code output, and is essentially a second type-checking phase. This is dissimilar to LaCasa, that runs as a compiler plugin and therefore in conjunction with the normal type-checking phase. The authors' goal, also similar to LaCasa, is to provide a library that can be used in an existing language, instead of forcing users to migrate their entire ecosystems to a new language.

Kilim, the Java library, provides annotations that Weaver uses as indication of what kind of transformation should be done. The `@pausable` annotation, e.g., signifies that a method should be transformed to a lightweight thread in continuation-passing style (CPS). The annotation is propagated upwards, so any callers must also be annotated as `@pausable`, similar to an effects system like Joëlle has.

An actor message in Kilim is tree-shaped, protected from internal aliasing, and may only have one owner at once. In other words, messages are isolated and unique. Weaver statically enforces the isolation of messages. The user has three message annotations to their disposal, `@free`,

`@cuttable`, `@safe`, corresponding to capabilities on method parameters of messages. In decreasing order, they represent the amount of freedom the user has when acting on a message of that capability. Corollary, the restrictions on what kind of messages can be sent as arguments is ordered decreasingly, with `@free` imposing the most restrictions. This is different from how LaCasa works, since there is structurally only one type of message, namely boxes, and additionally `Safe` types that Kilim has no similar concept of.

3.2.2 SOTER

SOTER, *Safe Ownership Transfer enablER*, is a tool for JVM (the Java Virtual Machine, which both Java and Scala runs on) bytecode analysis (like Weaver), that can identify ownership transfers and warn if, in the Scala case, a reference is leaked on a send between actors [20]. Its purpose is similar to LaCasa, in that it wants to prevent aliasing of references between actors, and has ownership semantics. The authors show that SOTER is effective at finding messages that can be passed by reference (c.f. `Safe` messages in LaCasa) instead of by value (using deep-copy semantics), which is the default in their Java actor library of choice.

LaCasa supports a strictly larger set of messages for ownership transfers, since it has the box concept where messages that normally cannot (safely) be sent via reference can be wrapped in a box. SOTER supports the equivalent of `Safe` messages.

3.2.3 Singularity OS

In Singularity OS, a Microsoft research project, the only way to communicate between processes is with message passing [8]. A process acts as an actor and is capable of sending and receiving messages. There are contracts on the channels used for communication, enforcing the messages to look in a certain way at both compile time and runtime.

Channels are bidirectional and sending a message on a channel transfers ownership of the message to the receiver. Larger data structures that traditionally use shared memory between processes for memory efficiency can send their pointers over the channel, and the receiver will own the data, without having to send the contents of the data itself across the channel.

Singularity OS is written in an extension to C#, called Sing#. Simi-

larly to LaCasa, it provides additional functionality for type and memory safety. Additionally, Sing# supports message-based communication as part of the language.

3.3 Empirical studies

Below, a few empirical studies using Rust are highlighted, where its type system based on ownership and borrowing is both commended for preventing bugs, as well as criticized for its lack of convenience in certain scenarios.

Anderson et al. [3] details their experience using Rust to write Servo. After an informal inspection revealed that up to 50% of all bugs in Firefox were due to use-after-free, out-of-range access or integer overflow, the authors found that there had not been a single use-after-free bug after two years of developing Servo in Rust. To them, this justifies the additional work needed to get a program approved by Rust's stricter type system.

Moreover, Servo uses message passing to communicate between its components, and Anderson et al. found the biggest challenge to be to reason about the threads involved – not issues with memory management or data races.

Cimler, Doležal, and Pscheidl [5] created solutions to a problem in a low-level way in Rust and a high-level way in C#, and compared them. They found that it was easy to split the task into smaller pieces and parallelize them in Rust, knowing that the type system ensures thread safety. However, the effort required to solve it was generally lower in C# than it was in Rust, except in some specific parts of the problem.

Levy et al. [16] used Rust in creating an OS for use in embedded systems. They found that Rust was overly restrictive in its memory guarantees with regards to embedded systems. E.g., a resource in an embedded system might always be there, while in a regular OS, a resource may disappear at any given time.

Chapter 4

LaCasa Adapter for Akka

Chapter 2 showed that there are convenience quirks in LaCasa, e.g., implicits and path-dependent types. Since those features require changing the signature of methods in order to work, they cannot be put verbatim into Akka. The process of going from an initial implementation in Akka with similar characteristics to the LaCasa actor system, to more convenient versions is described below. First, the interfaces, i.e. the user-facing API, are introduced and then the next section focuses on how the interfaces are implemented.

4.1 First attempts at an Akka integration

For LaCasa to work properly, its implicit access permissions and exception-handled control flow need to be integrated into Akka. Akka expects a method `receive` of type `PartialFunction[Any, Unit]` to be overridden by subclasses of the `Actor` trait. LaCasa includes a minimal actor implementation, to demonstrate its applicability, and in comparison to the corresponding receive method in that implementation (see Listing 4.1), the header is different, most notably in the additional parameter list including the implicit access value.

Semantically, the biggest difference is that messages in LaCasa are encapsulated in boxes and are typed. In Akka, a message is of type `Any` and via the `PartialFunction[Any, Unit]`, a match is performed to conditionally branch depending on the value and type of the message content. Moreover, the fact that there is an implicit value being passed along in the `receive` method cannot be reflected in the Akka version without changing its method signature, and it is impossible to rewrite a

```
abstract class Actor[T] {

    def receive(box: Box[T])(implicit acc: box.Access): Unit

    ⋮

}
```

Listing 4.1: The receive method signature for the minimal LaCasa actor implementation.

method's signature when overriding it.

The first attempt was to use implicit conversions to convert Akka's `ActorRef` to a LaCasa `ActorRef` wrapper, allowing the use of LaCasa-enabled methods. It did not prove convenient enough, since users could accidentally use a regular Akka method. The attempt was therefore abandoned, in favor of what is presented below. Ideally, users should be forced into an environment where there is no risk of using regular Akka, i.e. non-isolated, versions of methods, without the intention of doing it.

4.2 Interfaces

The final design decision was to create alternative versions of `Actor`, `ActorRef`, `ActorSystem`, et al. (*Actor** from now on) with the same names, but in their own package hierarchy. That way, a change of package path is enough to get started, and the concepts from Akka carry over. To accomplish that goal, each alternative version internally wraps their Akka namesake, and reroutes methods to act on the underlying implementation, while providing a new, safer, API. This approach is borrowed from the design of the internals of Akka Typed [18]. The underlying implementations are described in Section 4.3.

In idea, the design is very similar to the adapter pattern; a new interface is created to match other expectations than what the original interface can provide [9]. Furthermore, it is a very specific application of the adapter pattern where the method signatures are typed more restrictively, adding more type system features, i.e., boxes and their access permissions (in the form of implicit parameters). Introducing the new concept of a *Typed Signature Adapter*, TSA. The purpose of a TSA is to improve the signatures of an interface, and thereby provide more

type safety. The adapter implementations that follow are all applications of TSA. TSA is instrumental in maintaining users in the new adapter ecosystem.

Since the interfaces superficially look the same as their Akka namesakes, the concepts are not foreign to a user familiar with Akka. Furthermore, putting them in a different package with no visible connection to regular Akka prevents users from accidentally using a regular Akka interface when the intention is to contain them in the new ecosystem. Still, it should be possible to provide escape hatches for users to interface with regular Akka actors, so that they can migrate their application piece by piece instead of all at once.

In the sections below, each alternative Akka version is described in detail, including their final user-facing APIs.

4.2.1 Actor system

Recall that in Akka, an actor system is used to bootstrap the Akka application and to construct and initialize the actors. On initialization of an actor via the system, an actor reference is returned. Normally, this is the Akka `ActorRef`, and if a user acquires that reference, they can easily use the wrong methods, even by mistake. The goal is to prevent users from making easily avoided mistakes, and at the same time allow them to fall back to the original implementation, should some functionality be missing.

The way that an actor is created in the system is via the `Props` interface. The props are created as a template for an actor instance, i.e., `Props(new MyActor(1))` can be used multiple times to create a new instance of `MyActor`, each separate from the previous. Once an actor has been spawned in the actor system via the props, an actor reference is returned.

Since the design was decided on creating an interface that mimics Akka's, the problem of users accidentally using regular Akka methods is fully avoided. In practice, for the actor system, a new abstract class was created under a similar package, and exports a subset of methods with the same name, as its namesake in Akka. See Listing 4.2 for the differences in the actor system's path between Akka and the adapter. A similar thing was done for `Props`, creating a new public trait with, on the surface, the same kind of functionality as the Akka props class. However, it accepts subclasses of `BaseActor` (introduced in Section 4.2.4), instead

of subclasses of the regular Akka actor interface.

```
// Akka import statement
import akka.actor.ActorSystem
// Adapter import statement
import akka.lacasa.actor.ActorSystem
```

Listing 4.2: Comparison of the `ActorSystem` import statements for Akka and the adapter. The only difference is the addition of `lacasa` in the package path.

Methods that normally return, e.g., `akka.actor.ActorRef` will return the adapter equivalent of `akka.lacasa.actor.ActorRef` – again with a subset of methods with a very similar profile. This way, it is not possible to accidentally get an Akka `ActorRef` when using the adapter `ActorSystem`, thanks to the TSA pattern.

In order to minimize work required to implement the adapter, features were implemented on a by-need basis. In the end, the following interface, in Listing 4.3, sufficed for the actor system.

```
abstract class ActorSystem extends ActorRef {
  def name: String
  def actorOf(props: Props, name: String): ActorRef
  def terminate(): Future[Terminated]
}
```

Listing 4.3: The final `ActorSystem` interface used for the adapter.

There is not a lot of functionality that is necessary for the basics of the actor system. Essentially, being able to name the actor system and have it spawn new actors is enough. As recently established, `ActorRef` in this case is an interface in the LaCasa package structure, not the original Akka `ActorRef`. Any *Actor** interface mentioned from now on will be in the new LaCasa family (in the `akka.lacasa.actor` path hierarchy), unless otherwise specified.

4.2.2 Actor reference

For actor references, there are some scenarios where sending a non-`Safe` message might ruin the encapsulation. One such example is the ask pattern; if a user sends a `Safe` message they should expect a `Safe` message

back, but if there are no restrictions on the interface, they might instead get a boxed message back. Recall that the way the ask pattern works, is that once a message is sent as an ask to another actor, the sending actor waits for a response. Since the type of the response is included in the signature of the ask method, supporting both types of messages (`Safe` and boxed) is non-trivial because of their different requirements on method signatures. Therefore, the decision was made to only allow `Safe` messages to be returned via the `sender` field.

For that purpose, the `ActorRef` interface was split in two, one for `Safe` messages, and the other an extension to the first with boxed messages. Thereby, those methods (e.g., `sender`) that want to enforce that only `Safe` messages can be sent, can use the `Safe` actor reference version, and the rest of the methods can default to the less restrictive version. The interface for the aforementioned `Safe` actor reference can be seen in Listing 4.4.

```
trait SafeActorRef {
  def tell[T: Safe](msg: T): Unit
  def path: ActorPath
}
```

Listing 4.4: A simplified version of the `SafeActorRef` implementation.

The non-`Safe` actor reference interface only adds one additional method, see Listing 4.5. Statically, there is obviously a difference between the two traits, but at runtime, they are represented by the same object that extends both traits and wraps the Akka actor reference in a hidden implementation (more on that in Section 4.3).

```
trait ActorRef extends SafeActorRef {
  def tell(msg: Box[Any])(implicit access: msg.Access):
    Nothing
}
```

Listing 4.5: The final implementation of `ActorRef`.

In addition to the actor reference traits, there are implicit definitions that add operators for `tell`, enabling users to write `actorRef ! message` instead for the `Safe` actor reference. Likewise for the non-`Safe` actor reference, an operator is added for sending boxed messages:

`actorRef !! boxedMessage`, as well as a method for continuing execution after the message is sent, continuation-passing style (CPS). The definitions for these are shown in Listing 4.6.

```
implicit final class SafeActorRefOps(val ref: SafeActorRef)
    extends AnyVal {
  // Allows continued execution, just like regular Akka
  def ![T: lacasa.Safe](msg: T): Unit = ref.tell(msg)
}

implicit final class ActorRefOps(val ref: ActorRef) extends
    AnyVal {
  // Consumes the box, so nothing will be run after this call
  def !!(msg: Box[Any])(implicit access: msg.Access): Nothing
   = ref.tell(msg)(access)

  // CPS version of tell, allowing continued execution
  def tellAndThen[S](msg: Box[Any])(cont:
   NullarySpore[S]{type Excluded = msg.C})(implicit access:
   msg.Access): Nothing = {
    // Box.unsafe provides an environment that continues
    // even after a box is consumed, which normally
    // is not safe, hence the name.
    Box.unsafe(ref.tell(msg)(access))
    cont()
    msg.consume
  }
}
```

Listing 4.6: Implicit conversions for `ActorRef` to add tell operators to both the `Safe` and non-`Safe` versions.

4.2.3 Actor context

The interface for `ActorContext` is very similar to its Akka namesake, and the implementation is therefore straightforward. Its methods expose the LaCasa versions of the *Actor** traits, as introduced above. The `sender` method, specifically, is more restrictive in that it returns a `SafeActorRef`, instead of the more permissive normal actor reference (that also allows sending boxes). This is to prevent the aforementioned

issue of returning boxes when the sender expects a `Safe` message. This way, it is statically guaranteed.

Following the same implementation pattern as `ActorSystem` and `ActorRef`, `ActorContext` provides a public interface as well as a hidden adapter implementation that wraps the Akka `ActorContext`. See Listing 4.7 for the interface. The most important components of the interface are `self` and `system`, since they are both accessed from inside an actor. Additionally, `stop` is a common pattern to use, stopping an actor.

```
trait ActorContext {
  def self: ActorRef
  def system: ActorSystem
  def sender(): SafeActorRef
  def stop(child: ActorRef): Unit
  implicit def executionContext: ExecutionContextExecutor
}
```

Listing 4.7: The final `ActorContext` interface used as basis for the adapter.

4.2.4 Actor

In the base actor interface, both `receive` methods for `Safe` and non-`Safe` messages need to be implemented, see Listing 4.8. In Listing 4.9, a few helper traits are provided, to simplify the cases when only either of the receives are needed. An abstract class, `Actor[T]`, is also provided, bridging the gap when migrating a common scenario of an actor receiving only `Safe` messages. By extending `Actor[T]` with a `Safe T`, users can still use the `Receive` type and to an end user, the `receive` method definition will look the same as regular Akka. In reality, it is a parameterized partial function over `T` instead of `Any`, hidden in the `Receive` type alias.

4.2.5 Actor logging

To show the applicability of mix-in actor traits, just like regular Akka has, a copy of the Akka `ActorLogging` trait was created. Its application is done exactly the same as in Akka, by mixing in the trait in an actor implementation – i.e., extending it in conjunction with one of the aforementioned actor traits. That way, users gain access to the `log` instance variable, and can use it to log messages as normal.

```
trait BaseActor {
  def receive[T: Safe](msg: T): Unit
  def receive(msg: Box[Any])(implicit acc: msg.Access): Unit
  implicit val context: ActorContext = ...
  implicit final val self: ActorRef = context.self
  implicit val executionContext: ExecutionContext =
    context.executionContext
  final def sender(): SafeActorRef = context.sender()
}
```

Listing 4.8: The final `BaseActor` interface that end users should implement (or a derivative thereof) when using the adapter.

```
trait OnlySafe { self: BaseActor =>
  // Implement box receive;
  // subclasses must only implement safe receive
  def receive(msg: Box[Any])(implicit acc: msg.Access): Unit
    = ...
}

trait NoSafe { self: BaseActor =>
  // Implement safe receive;
  // subclasses must only implement box receive
  def receive[T: Safe](msg: T): Unit = ...
}

trait TypedSafe[T] { self: BaseActor =>
  type Receive = PartialFunction[T, Unit]

  def receive: Receive

    ⋮
}

abstract class Actor[T]
  (implicit val tag: scala.reflect.ClassTag[T],
   implicit val safe: lacasa.Safe[T])
  extends BaseActor with TypedSafe[T]
```

Listing 4.9: Helper traits for the user to implement together with the `BaseActor` trait. The ellipses hide implementation details that are irrelevant to the interface.

In addition to the new trait, any classes that mix the `ActorLogging` trait need to have an implicit evidence for `ActorLogSource`. Therefore, those are provided for `BaseActor`, just like how Akka provides it for its `Actor` – see Listing 4.10 for how it was done in the adapter.

```
object BaseActor {
  implicit val actorLogSource: LogSource[BaseActor] = new
   LogSource[BaseActor] {
    def genString(a: BaseActor) = a.self.path.toString
  }
}
```

Listing 4.10: Implicit evidence for `LogSource` provided for `BaseActor`, to allow logging from inside an actor.

4.3 Implementations

In the previous section, the *Actor** interfaces were introduced and described, and their implementations were only briefly mentioned. This section will exemplify one of the interfaces, `ActorSystem`, but it generalizes to both `ActorRef` and `ActorContext`. There is a section dedicated entirely to the implementation of `Actor`, since it is quite different from the rest; there is an underlying regular Akka actor that receives messages and reroutes them to the relevant receive method in the user-facing API. First, however, a section on the internal wrappers used for indicating what kind of messages are being sent.

4.3.1 Wrapping values

Internally, there are two different types of wrappers used for indicating to the system what kind the sent message is. In the receive method of the underlying actor, the wrappers are destructured on type and handled accordingly. If the message is neither of the two wrappers, it is most likely a message that was accidentally sent from a regular Akka actor. In this case, it would be possible to reroute it to a user-defined receive method and have them handle it. For the purposes of this thesis, it was not necessary and was therefore omitted. Corollary, when sending messages via an actor reference, the adapter ensures that the messages are wrapped in the relevant wrapper type. The two types are:

`Packed[T]` When sending boxes, the LaCasa-provided `pack` method is used to acquire a `Packed[T]` from a box. However, since it is package-private in LaCasa, a work-around is used to be able to access it outside of the LaCasa package. See Listing 4.11 for how it is done.

```
package lacasa
object PackABoxHelper {
  final class NoReturnControl extends lacasa.NoReturnControl

  def pack[T](box: Box[T])(implicit access: box.Access):
    lacasa.Packed[T] =
     box.pack()
}
```

Listing 4.11: A helper object that allows access to LaCasa's package-private internals by claiming to be in the same package.

`SafeWrapper[T]` When sending `Safe` messages, a hidden class `SafeWrapper[T]` is used, which holds the value of the message, as well as its `Safe` evidence. The implementation is minimal, see Listing 4.12.

```
private case class SafeWrapper[T](value: T)(implicit val
    safe: lacasa.Safe[T])
```

Listing 4.12: A wrapper class for `Safe` messages, used internally to enforce that messages sent and received are `Safe`.

4.3.2 Interface implementations exemplified

Each of the interfaces have an underlying adapter implementation, hidden from the user. Since `Actor` is special, it gets its own section below. However, for `ActorRef`, `ActorContext`, `ActorSystem` and `Props`, the general idea is the same – they all follow the TSA pattern. Therefore, to simplify and avoid redundancy, only `ActorSystem` will be detailed.

There are three components that makes actor system work in the adapter: an Akka actor system instance, the adapter `ActorSystem` interface, and the underlying implementation. The underlying implementation wraps the Akka instance, and implements the interface. It is also

```
private class ActorSystemAdapter(val unsafe:
    akka.actor.ActorSystemImpl)
  extends ActorSystem with ActorRef with ActorRefImpl {

    ⋮

  override def toString: String = unsafe.toString

  override def name: String = unsafe.name

  override def terminate(): Future[Terminated] =
    unsafe.terminate()

  override def actorOf(props: Props, name: String): ActorRef =
     ActorRefAdapter(unsafe.actorOf(PropsAdapter(props), name))
}
```

Listing 4.13: The hidden implementation of the `ActorSystem` interface, with some details omitted.

private, i.e., hidden from the user, see Listing 4.13 for the most relevant parts of the implementation.

Methods are rerouted to the wrapped Akka instance, and when applicable, no additional conversions are needed. E.g., for `name` that returns a string, calling the same method on the underlying Akka instance is enough. However, for some methods, there is a need to convert both parameters and return types. Take `actorOf`, for instance, where the passed props are of the adapter type, and need to be converted to Akka's own props type before calling the Akka `actorOf` method. Additionally, the result is an Akka `ActorRef` and needs to be converted to an adapter `ActorRef`, which is done via its adapter class.

Since some methods in the Akka `ActorSystem` interface rely on hidden implementation details, to mimic that same interface, the adapter also needs to rely on the same implementation details. Therefore, the wrapped Akka instance in `ActorSystemAdapter` is of type `ActorSystemImpl`, which is an implementation hidden to packages in the Akka namespace. Luckily, the adapter is in the Akka namespace (it is enough to have the top-most package level be called `akka`), and can therefore access the – normally hidden – implementation.

The implementations for `ActorRef`, `ActorContext` and `Props` are

very similar in execution, and are therefore omitted for the sake of brevity. The final interface, `Actor`, is somewhat different in execution, and is therefore presented on its own below.

4.3.3 Actor

In principle, the implementation for `Actor` is similar to the other interfaces; wrap the Akka implementation and hide it behind an interface. For actors, however, the Akka version is the first to receive the messages, instead of a direct call on the interface, which is true for the rest of the interfaces. Instead of wrapping the Akka implementation in a private adapter, a backing Akka actor receives all messages, and sends it to the wrapped adapter actor interface. Consider it the duality of how the rest of the interfaces are designed.

As introduced in Section 4.3.1, `SafeWrapper` and `Packed` are used in the backing actor to indicate what kind the messages received are. From there, the backing actor can reroute the messages to its corresponding method on the adapter actor interface. See Listing 4.14 for the basis of what the backing actor looks like.

Note that the constructor argument to the adapter is a *call-by-name* type variable; the expression of type `BaseActor` is received as is, and is not instantiated before it is used. Had it not been call-by-name, the adapter would immediately instantiate the actor instance. The actor adapter needs to instantiate the user actor only once it has been instantiated itself by the underlying Akka runtime. Recall that, in Akka, the actor system handles instantiation and actor lifecycles. If an actor is instantiated outside of the actor system, it does not work. It is also important that the constructor parameter `_actor` is only used once (in the constructor body), since multiple uses would instantiate the actor multiple times.

Communicating with regular Akka When a message that is not wrapped inside `Packed` or `SafeWrapper` is received, the adapter will reject it. Another solution would be to route it to a fallback receive method where users can handle regular Akka messages, and via helper methods on an actor reference, send messages to a regular Akka actor. That way, users can migrate their application piece by piece, instead of attempting to convert everything at once. This is currently not implemented, but is straightforward to do, since the Akka versions of both actor references

```
private class ActorAdapter(_actor: => BaseActor) extends
    akka.actor.Actor {

  val ref = _actor

  def receive: Receive = {
    case packed: Packed[_] =>
      // Wrap box receive in Box environment.
      Box.unsafe({
        ref.receive(packed.box)(packed.access)
      })
    case x: SafeWrapper[_] =>
      ref.receive(x.value)(x.safe)
    case x =>
      // Call escape hatch receive for Akka actors.
      // (Not implemented.)
  }
}
```

Listing 4.14: The hidden implementation of an Akka actor, wrapping the `BaseActor` trait.

and actors are readily available from their interface implementations.

4.4 Safe marker

Providing implicit evidences for a type to indicate that it is `Safe` can require mundane copy-paste-driven typing, and is prone to errors. In a common scenario, when all messages being received by an actor are `Safe`, and they all have a common superclass that is used together with `Actor[T]`, it should be simple to declare all of the messages as `Safe`. Therefore, a new trait was created, tasked with the simplification of this process. Its name is also `Safe`, but the idea is that users will almost always use this new `Safe`, and rarely resort to the `Safe` in the LaCasa package.

Usage of this new `Safe` is straightforward: on the message superclass, users ensure that the message extends `Safe`. In Listing 4.15, the framework for this new `Safe` is shown. The trait acts as a marker, and then for every `T` implementing the trait, an implicit evidence of `Safe[T]` is automatically provided.

```
trait Safe

object Safe {
  implicit def safeIsLaCasaSafe[T <: Safe]: lacasa.Safe[T] =
    new lacasa.Safe[T] {}
}
```

Listing 4.15: Actor `Safe` marker trait, with the definition of its implicit evidence for LaCasa `Safe`, simplifying the marking of `Safe` classes.

To prevent accidentally marking a non-`Safe` message as `Safe`, users should make the superclass sealed (i.e., prevent new subclasses from being declared outside of the file), and ensure that all subclasses are `Safe` via manual inspection. Right now, LaCasa cannot confirm that a class really is `Safe`, so it has to rely on users to get it right. Warnings for mislabeling a class as `Safe` could be a potential addition to LaCasa in the future, however.

4.5 Feature summary

In the newly introduced interfaces, most of their methods have the same header as they do in Akka – superficially, by name and return type translation to their own versions of the *Actor** interfaces. The `receive` and `tell` methods have different headers to restrict messages to only `Safe` and boxed messages, thereby only allowing isolated messages to be sent and received, which is an application of the TSA pattern. Anything other than those two types of messages should not be allowed – but could be, to allow an easier transition from regular Akka to using the adapter. In effect, users can change the package path of Akka imports to the adapter's package path, and then, in theory, fix the compiler errors that arise.

In the case that all messages sent are `Safe` and only the supported subset of functionality is used, the translation from regular Akka to the adapter should be straightforward. The biggest distinction is that *either* the header for receive is changed, *or* `Actor` needs to be parameterized by a `Safe` superclass of all received messages. The latter is preferred, since it forces users to think about what messages are being received by the actor.

Some original Akka functionality is ignored in the adapter implemen-

tation; there is, e.g., no `become`. The ignored functionalities are merely conveniences and the same behaviors can be replicated by introducing additional actors. There is nothing inherently preventing them from being implemented in the given structure, but since their omission is not affecting the expressiveness of the LaCasa-Akka adapter, they are excluded for the purpose of this thesis.

Chapter 5

Evaluation

In this chapter, the applicability of the developed adapter is evaluated. An empirical study is performed by migrating a selection of programs from regular Akka to the LaCasa-Akka adapter. The program selection is listed and described in Section 5.1, and the metrics are introduced and motivated in Section 5.2, along with the actual results. Finally, the performance is compared before and after the migration. Patterns identified during the migration are described in Chapter 6.

5.1 Empirical study

To examine the feasibility of applying the adapter, a selection of programs is necessary. The Savina actor benchmark suite [15] – the de facto benchmark suite for actor programs[1] – has an Akka version for each benchmark. In order to minimize the differences between the Akka version and the adapter version, the programs are first stripped of parts related to their benchmark harness. That way, the programs are kept as simple as possible, and empirical results are not affected by the harness code.

5.1.1 Test cases

The selection of programs is done to capture different kinds of actor applications, and the idea is to find convenience pain points in the process and resolve them as they appear. Below, the different programs are

[1]The Savina paper by Imam and Sarkar is widely cited; 51 times according to Google Scholar.

listed, along with a short description of what they do. The `CHAMENEOS` and `BANKING` programs were both instrumental in the development of the adapter as they served a testing ground for new ideas.[2]

`CHAMENEOS` One master actor and multiple "chameneo" actors – the chameneos all meet up, and once they meet someone of a different color they change to their complement color.

`BNDBUFFER` Multiple producers and consumers with a manager actor that delegates work between them. The manager actor manages state in a mutable buffer, but data sent between the actors is immutable.

`PHILOSOPHER` Based on the classic dining philosophers problem, with one arbitrator actor that oversees the philosopher actors, using a shared list of atomic booleans stating whether a shared resource is busy or not.

`BANKING` One teller actor, and multiple account actors. Once an account actor receives a credit message, it will send a debit request to another account provided in the message, using the ask pattern (introduced in Section 2.2.1).

`NQUEENK` One master actor with multiple worker actors that solve the "n-queens" problem using divide and conquer. It sends a mutable list to each worker and they process it in memory.

`BITONICSORT` An implementation of the bitonic sort parallel algorithm, with multiple actors handling different stages of it. Actors are spawned dynamically in loops, cascading across the different actor types.

`SIEVE` The sieve of Eratosthenes, a well-known algorithm for finding prime numbers, implemented using actors – one actor for producing numbers and another one for filtering non-prime numbers, dynamically spawning new actors of itself.

5.2 Results

In this thesis, the convenience of using the adapter is the primary concern (performance is of secondary concern, and is presented in the next

[2]They were, together with `THREADRING`, presented in the initial results of this thesis.

section). Therefore, the two metrics chosen to evaluate the test cases are *lines of code* and *lines of code changed* – presented below in that order. Additionally, they are both summarized in a table following the same format as the initially presented results.

The metrics do not necessarily denote an exact translation of how convenient using the adapter is, but they give an estimate of required code changes when migrating an application, which gives an idea of the adapter's convenience. Some lines are excluded from both measurements, namely import statements (they are constant overhead), comments and blank lines.

5.2.1 Lines of code

In Figure 5.1, the number of lines of code (LoC) for each program is listed twice – once for the regular Akka version and once for the adapter version. In 6 out of 7 test cases – where `NQUEENK` and `NQUEENK[BOX]` are considered as one – there is only one code line of difference in favor of the Akka version. However, for the 7th test case, `NQUEENK`, the `Safe` program has one fewer lines of code, and for `NQUEENK[BOX]`, the difference is greater because of the usage of boxes. The test case `NQUEENK` is the only anomaly in requiring some changes during the migration to be `Safe`, since it originally uses mutable arrays to pass data. With the change of data structure, the API is changed as well, and one of the usages for arrays in the original `NQUEENK` is to create a new array and copy an old one; that specific pattern has direct support in the immutable replacement, thus reducing lines of code for the same action. A more in-depth study on these migration patterns is done in Chapter 6.

The reason that the rest of the test cases only has one extra line of code is for the added superclass of all messages; a line with `sealed trait Message extends Safe` is added to all the `Safe` test cases. Then, each message extends `Message`, which is used to parameterize the `Actor` class. More on this migration pattern is described in Chapter 6.

5.2.2 Lines of code changed

Counting LoC before and after the migration does not show the entire picture; if every single line is replaced, it would appear from the previous graph that nothing has changed. Together with number of lines of code changed, however, they should give a clear indication of how much needs

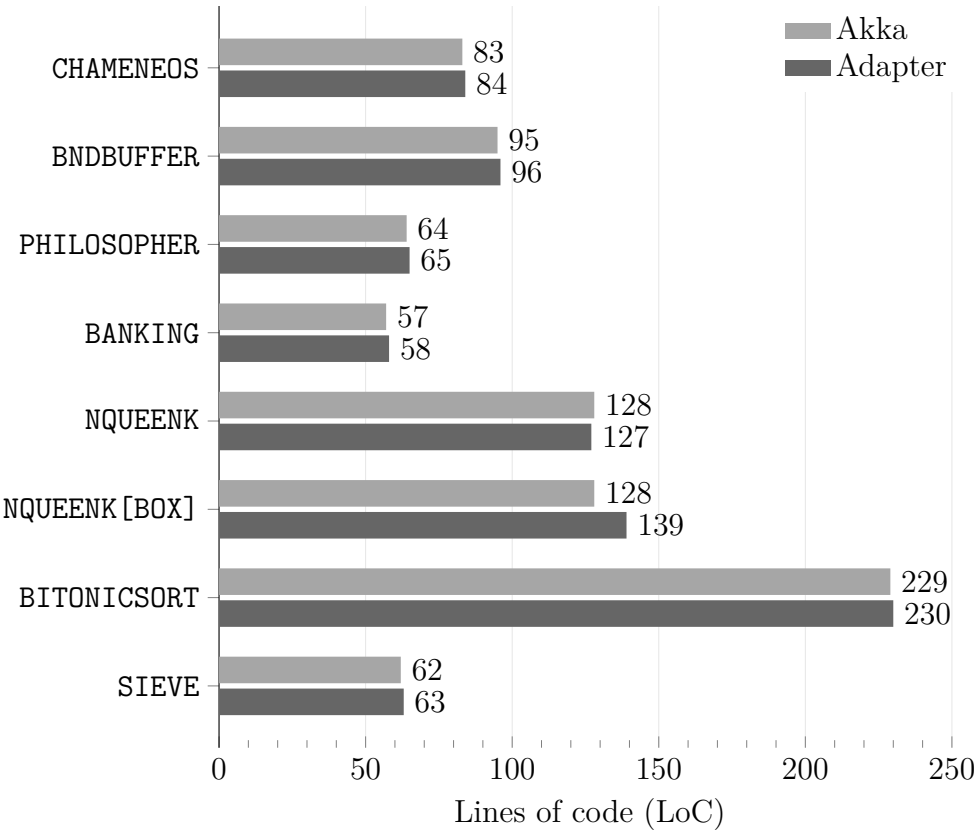

Figure 5.1: Number of lines of code (excluding imports) for each of the test cases, one for the Akka version and one for the migrated version using the adapter. The adapter version is for `Safe` messages, except for `NQUEENK[BOX]`. The fewer lines the adapter has compared to Akka, the better.

to be changed without having to manually inspect the code differences.

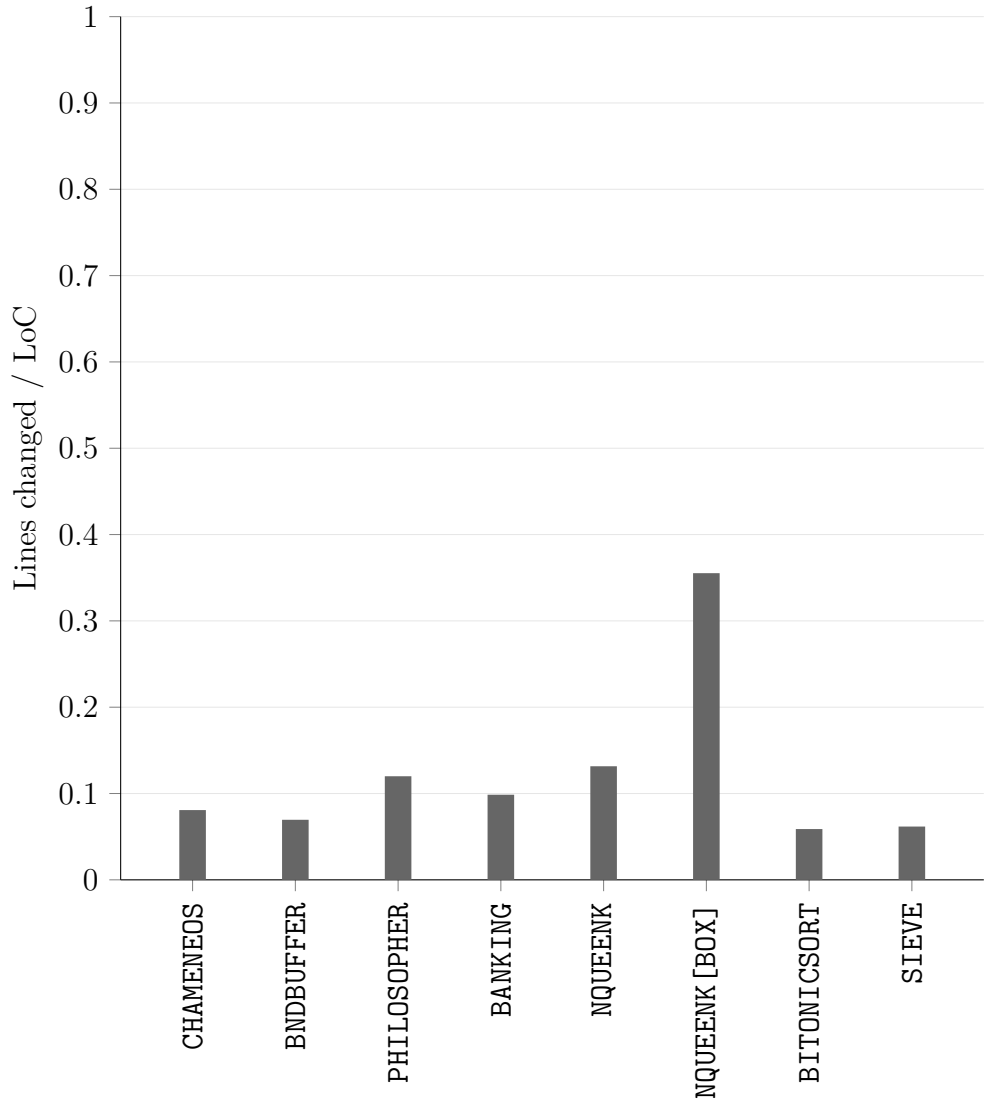

Figure 5.2: The ratio of number lines changed between the adapter and Akka versions and the LoC for the Akka version, for each of the test cases. The adapter version is for `Safe` messages, except for `NQUEENK[BOX]`. Smaller is better.

From Figure 5.2, it reads that on average 10% of the total LoC need to be modified in some way when only `Safe` messages are used. One of the larger programs, `BITONICSORT`, has the lowest ratio of lines

changed, whereas PHILOSOPHER with its low LoC has a bigger ratio – it also has comparatively more messages in relation to its size than the other programs. The reason that both NQUEENK and NQUEENK[BOX] have the biggest ratio out of all the test cases can be attributed to the previous explanation: that one of its messages is not Safe, and therefore had related changes propagated throughout the program. That is especially true for the box version (NQUEENK[BOX]), which is apparent from requiring almost 40% of the code to be modified in some way.

5.2.3 Detailed data

The above metrics are summarized in Table 5.1, which follows the same style as what was presented in the initial results [14] of this thesis. Those results presented data for THREADRING, CHAMENEOS and BANKING. The new test cases are less trivial than THREADRING, providing more challenges for migration. Therefore THREADRING is ignored in these new results. Do note that in the initial results every line was counted, including blank lines, and in these new results only lines that contain code are counted. That means that changes reported in percent shows favorably for the old results. Despite that, the new results are better even in change percentage, for both CHAMENEOS and BANKING, due to improvements in the adapter.

Program	Akka (LoC)	Adapter (LoC)	Changes	Changes (%)
CHAMENEOS	83	84	7+/8-	8.1%
BNDBUFFER	95	96	7+/8-	7.0%
PHILOSOPHER	64	65	8+/9-	12.0%
BANKING	57	58	6+/7-	9.9%
NQUEENK	128	127	20+/19-	13.2%
NQUEENK[BOX]	128	139	35+/54-	35.5%
BITONICSORT	229	230	19+/20-	5.9%
SIEVE	62	63	4+/5-	6.2%
Average	106	108		12.2%

Table 5.1: Detailed results of the empirical study.

5.3 Performance

While the previous section show that few changes are required in most cases to convert a program, a trivial implementation of actor isolation would be to use deep-copy semantics on every send. Using such semantics is not performant, especially for larger messages. Therefore, performance is clearly important to motivate the use of the more intricate design on actor isolation presented in this thesis.

To evaluate the performance of the test programs, their initial designs from the Savina benchmark suite are modified to use only Scala. Their configuration classes are written in Java, so they are manually converted to Scala. (The reason that they are written in Java in Savina, is that more than the Akka library are used in the suite, and it simplifies sharing code between the actor libraries since some of them are Java only.) Rewriting the configuration classes is not necessary, but it minimizes the difference between the code for the regular Akka benchmark and for the adapter benchmark, and therefore simplifies the migration. It has no effect on the benchmark results.

Each of the programs – including benchmark harness code – are copied from the Savina project to create a separate adapter version, exactly like what was done when evaluating the LoC count. There are now two copies for each program and version, one with harness and one without, resulting in a total of four copies per program.

5.3.1 Benchmark helpers

There is an associated actor wrapper that handles setup and teardown of every actor iteration, and waits for the actor system to shutdown. It consists of two elements, the actor wrapper class, `AkkaActor`, and a global state to count the number of active actors (and ensure that once they all exit, the run is done), `AkkaActorState`. These are copied to create adapter parallels, naming them `LAkkaActor` (where `LAkka` is short for LaCasa-Akka) and `LAkkaActorState` . There are few changes required to get them to work as helpers for the adapter programs; the actor wrapper needs to ensure that the same restrictions that apply to an adapter actor also applies to the wrapper, and changing the method signatures is enough. Since the execution is essentially the same as what was done for the *Actor** traits (see Chapter 4), its details are omitted for the sake of brevity.

5.3.2 Benchmarks

Using the new benchmark helpers is a matter of changing the imports and replacing every occurrence with the new helpers. To fully migrate the benchmarks to be adapter compatible, all relevant messages are marked as `Safe` (see Section 4.4 for how it is done). This is enough to migrate all programs, except `NQUEENK`, where the same translations that were done in the previous evaluation (i.e., one version changing the data structure, and one version using boxes) are reapplied.

With all the benchmarks in place, for both the adapter and Akka versions, the benchmarks are run with default parameters and 10 iterations on a 4-core 2.67 GHz machine with 8 GB RAM. To ensure that the results are representative, the benchmarks are run multiple times until the average for each run (with 10 iterations each) converges. Only the latest benchmark run is used, with the average runtime over 10 iterations. These results are shown in Table 5.2. Ignoring the outliers, `BANKING` and `NQUEENK`, they show that the difference between Akka and the adapter is negligible, and within the error windows of the reported numbers. In some cases, the adapter even appears to be faster (which should not be the case, and is not statistically significant). The fact that `NQUEENK[BOX]` shows no difference towards the Akka version of `NQUEENK` indicates that the performance impact of using LaCasa's boxes is minimal, and should not be a reason to avoid using them.

As for the outliers, `BANKING` uses the ask pattern, and it could be that the pattern is not optimized enough. In the adapter implementation, it casts the response from the underlying Akka actors to be of the correct class, to provide a typed response. In the case of `NQUEENK`, it uses an immutable data structure instead of a mutable one, which seems to be the cause of its difference in performance, as `NQUEENK[BOX]` shows no difference in performance compared to regular Akka.

Program	Akka (ms)	Adapter (ms)	Difference (ms)	Difference (%)
BANKING	1423	1666	-243	17.1%
BITONICSORT	261	282	-21	8.0%
BNDBUFFER	1991	1970	**21**	**1.1%**
CHAMENEOS	180	190	-10	5.6%
NQUEENK	433	805	-372	85.9%
NQUEENK[BOX]	433	428	**5**	**1.2%**
PHILOSOPHER	253	273	-20	8.0%
SIEVE	269	258	**11**	**4.1%**

Table 5.2: Benchmark results with average runtime in milliseconds over 10 iterations. Positive differences are bolded.

Chapter 6

Migration Patterns

During the migration, five distinct patterns were observed. The five patterns are presented in this chapter, where two of the patterns relate to `Safe` messages, and the other three relate to handling boxes and their control flow. The aim of this chapter is to serve as a reference that can be used when migrating other applications. Throughout the chapter, examples from the migration are used to identify and exemplify the issues that the patterns are meant to solve.

Each section below is structured into three parts. The first part introduces and identifies the problem as found during the migration, and the second part provides the solution with a real example. Finally, the third part puts the pattern into context and discusses its general usability.

Before that, however, there is a step that all applications need to do before being able to access the new adapter classes: change imports. The required changes are minor, but are exemplified in Listing 6.1 for completeness sake. Importing `Safe` is only necessary if messages need to be marked as `Safe`. If there are no such messages, users would most likely need classes in the `lacasa` package, e.g., `lacasa.Box` for boxes and their general functionality.

6.1 Messages are all `Safe`

As long as an application sends and receives only `Safe` messages, and has access to the declaration of all messages, it can fully be migrated by applying three rules:

1) Create a new trait `Message` and have it extend `Safe`.

```
import akka.lacasa.pattern.ask

import akka.lacasa.actor.{ActorLogging, ActorSystem,
    ActorRef, Props, Actor, Safe}
```

Listing 6.1: Differences in imports between regular Akka and using the adapter. Additions when using the adapter are highlighted with yellow background. Example taken from `BANKING` – the only test case that required two Akka import lines (because of the ask pattern) instead of one.

2) Ensure all messages extend the new `Message` trait, and thereby are also marked as `Safe`.

3) Parameterize `Actor` with the newly created `Message` superclass.

Example When it comes to 1), 2) and 3), the changes are minor and straightforward, just like for import statements. In Listing 6.2, one line is added for the new `Message` trait, and all the messages are modified to be marked as `Safe`, but not intrusively so; it is enough to append **`extends`** `Message` at the end of each message's declaration line.

Discussion In all test cases, except `NQUEENK`, every message sent could be marked as `Safe`. Clearly, this is the most common pattern, at least for the chosen programs. Given that case classes are the recommended way to send data in Akka, and that case classes generally are deeply immutable, it is reasonable to expect that in most cases, all messages in an application are `Safe`.

If an application does not have access to the declaration of a message `Msg`, and it is not marked as `Safe` by other means, the message can be marked as `Safe` through the creation of an implicit value of type `lacasa.Safe[Msg]` (see Listing 6.3) – as long as the message is manually verified to be `Safe`. In that case, parameterizing the actor on a common superclass does not work. A solution is to split the actor into multiple actors to, e.g., have an actor that only receives `Msg` and then forwards it in some other container – that also needs to be marked as `Safe`– to the main actor. Thereby, the three rules can still be used, although with additional work. This exact scenario was not seen during the migration, but is still reasonable to expect.

```
// 1)
sealed trait Message extends Safe

// 2)
case class MeetMsg(color: Color, sender: ActorRef)
    extends Message

case class ChangeMsg(color: Color, sender: ActorRef)
    extends Message

case class MeetingCountMsg(count: Int, sender: ActorRef)
    extends Message

case class ExitMsg(sender: ActorRef) extends Message

// 3)
class ChameneosMallActor(var meetingsLeft: Int, numChameneos:
    Int)
    extends Actor[Message]
```

Listing 6.2: Differences in message declarations between regular Akka and using the adapter. Additions when using the adapter are highlighted with a lighter background. Example taken from `CHAMENEOS`.

```
implicit def msgIsSafe: lacasa.Safe[Msg] = new
    lacasa.Safe[Msg] {}
```

Listing 6.3: Marking a message `Msg` as `Safe` without having access to its declaration.

6.2 Convert non-`Safe` messages to `Safe`

Sometimes, the approach in the previous section does not work. More specifically, if not all messages are `Safe`. There are two approaches given that premise: either change the data type of the non-`Safe` message to one that is `Safe`, or resort to using boxes for the offending messages; the adapter supports receiving both `Safe` messages and boxes from the same actor. This section describes the former approach, while the next sections describe the latter approach.

Changing a core data type could take significant effort, so it might not always be a feasible option. A case-by-case decision should be made by manually inspecting the message and where it is used, and from there decide if the changes are reasonable to perform.

More often than not, the reason that a message is not `Safe` is due to using a mutable data structure, e.g., an array. The `Array` class is Scala's wrapper around regular mutable Java arrays. For other container types in Scala, there are usually two versions, one that is mutable (in the `scala.collection.mutable` package), and one with similar functionality but immutable (in the `scala.collection.immutable` package).

Example The original `NQUEENK` has a message that cannot be declared `Safe` since it contains a mutable array (see Listing 6.4). In terms of collection APIs, arrays are quite special as they rely on Java's implementation, and as a result have limited features.

```
case class WorkMessage(priority: Int, data: Array[Int],
   depth: Int)
```

Listing 6.4: The non-`Safe` message declared in `NQUEENK`. The offending field is `data`, since it contains a mutable collection.

Through inspection of the immutable collections available in Scala's standard library, the vector class is deemed the best fit to replace the array due to its performance characteristics – because changing a core data type such as a collection will clearly have an effect on runtime performance. However, since the actual size of the arrays are small, the choice of collection is chiefly based on API functionality – one such pattern in `NQUEENK` being copying the collection and modifying one of the indices, which is available as a single method call in the vector API. The resulting message is shown in Listing 6.5.

```
case class WorkMessage(priority: Int, data: Vector[Int],
    depth: Int)
```

Listing 6.5: The `Safe` message declared in the `Safe` version of `NQUEENK`, replacing `Array` with `Vector`.

With the offending message changed to conform to the requirements of being `Safe`, the rest of the code can be changed in accordance to 1), 2) and 3) as was demonstrated in the previous section. However, more changes than applying the three rules need to be made, since modifying the data in the message now requires using a new API. For `NQUEENK`, most changes are to method signatures, changing `Array[Int]` to `Vector[Int]`. In one core loop, however, modifications to the message need to be changed, since they rely on the array API. The changes made, and what it looked like before that change, are both shown in Listing 6.6.

```
(0 until size)
  .map { i =>
    // Using Vector
    workMessage.data
      .padTo(newDepth, 0)
      .updated(depth, i)

    /* Using Array
    val newData = new Array[Int](newDepth)
    System.arraycopy(workMessage.data, 0, newData, 0, depth)
    newData(depth) = i
    newData
    */
  }
  .filter { boardValid(newDepth, _) }
  .foreach {
    master ! WorkMessage(newPriority, _, newDepth)
  }
```

Listing 6.6: Excerpt from usage of the `Vector` API, showing how the `Array` API (in a commented block) is used to solve the same problem.

Discussion The immutable collections usually use structural sharing (i.e., internal aliasing of data to avoid deep copies), and have similar APIs

to their mutable counterparts. Therefore, a transition from a mutable collection to an immutable one might be simpler than expected. However, performance can still be affected, and should be considered when deciding whether to change data types or use another approach like boxes.

For `NQUEENK`, the intention of the updated code using the vector API reads clearer than when it used the array API. Additionally, recall from Section 5.3 that there was a significant difference from the original implementation and the `Safe` implementation using vector instead of array. If performance is more important, boxes is to prefer over changing a data structure, at least for `NQUEENK`. Performance impact should be measured on a case-by-case basis, and a decision whether changing data types is worth it made thereafter.

In more trivial cases, a message could be declared with mutable fields, e.g., `var age: Int`, to allow for in-place modifications (or due to general oversight). The suggested approach for that is to do the modification outside the message, and then create a (new) copy of the message with the relevant fields modified. Using mutability for in-place modifications of primitive types should rarely, if ever, provide enough gain in performance to motivate the lack of guaranteed isolation between actors.

6.3 Create boxes on startup

In some cases, it is convenient to do things on actor startup, e.g., sending start messages to other actors. Normally, in regular Akka, those actions can be done in the constructor (i.e., body of the class) without any restrictions. The same goes for adapter programs where all messages are `Safe`. However, if an adapter actor would want to send boxes to other actors on startup, the same pattern cannot be used.

Recall that boxes require special environments on consumption and creation (see Section 2.3.3), and sending a message to another actor on startup is a clear example of when a new message needs to be created. At compile time, nothing prevents the box creation statements from existing in the constructor, but as soon as the actor is instantiated, it will crash. Therefore, a special method called `preStart` is provided, allowing users to create and send boxes on actor startup, without having to manually provide the environment.

Example The startup routine for the `Safe NQUEENK` version can be seen in Listing 6.7. There is no boilerplate, and it does exactly what it

seems like; calling the `sendWork` method with a `WorkMessage` containing some initial data.

```
sendWork(WorkMessage(priorities, Vector(0), 0))
```

Listing 6.7: Actions taken on startup for the `Safe NQUEENK`.

On the other hand, Listing 6.8, shows how the same idea needs to be executed differently for `NQUEENK[BOX]` since it is handling boxes.

```
override def preStart(): Unit = {
  Box.mkBoxFor(
    WorkMessage(priorities, new Array[Int](0), 0)
  ) { packed =>
    sendWork(packed.box)(packed.access)
  }
}
```

Listing 6.8: Actions taken on startup for `NQUEENK[BOX]`.

Discussion Generally, the two versions read the same, as long as the reader is familiar with LaCasa's box API, but the additional syntax makes it harder to see what the piece of code does. Another solution to providing the `preStart` method would be to force users to declare their own box environment via `Box.unsafe`. However, users should not need to know about the unsafe environment, and providing a startup method is therefore preferable.

6.4 Receive boxes

In both regular Akka, and when all messages are `Safe` in the adapter, there is only one receive method; one arrival point for all messages the actor receives, and one place to perform behaviors. As soon as boxes are introduced to an adapter actor, this is no longer the case. There is going to be two receive methods: one for `Safe` messages (since the actor can still receive those), and one for boxed messages.

Example Consider the receive in the `Safe` version of `NQUEEN`, shown in Listing 6.9. It receives and handles all the messages sent to the actor.

The message of interest is `WorkMessage`, which is sent to the `sendWork` method once such a message is received.

```
override def receive: Receive = {
  case msg: WorkMessage => sendWork(msg)
  case _: ResultMessage => ...
  case _: DoneMessage   => ...
  case _: StopMessage   => ...
}
```

Listing 6.9: Receive for the master actor in the `Safe NQUEENK`.

Now consider the `Safe` receive in `NQUEEN[BOX]`, shown in Listing 6.10, and note that it receives the same messages (and looks the same) as the `Safe NQUEENK`, *except* for `WorkMessage`, which is the non-`Safe` message, since that message is received in the dedicated box receive method.

```
override def receive: Receive = {
  case _: ResultMessage => ...
  case _: DoneMessage   => ...
  case _: StopMessage   => ...
}
```

Listing 6.10: `Safe` receive for the master actor in `NQUEENK[BOX]`.

The receive method for the master actor in `NQUEEN[BOX]` that handles `WorkMessage`s, looks similar to how `NQUEENK` handles the messages, with added LaCasa box boilerplate on top (see Listing 6.11). Since the received message is a box, and the `sendWork` method needs a box (because it modifies the state of the actor, which cannot be done inside the box open method), opening the box is not possible since that only mutates the content. Therefore, there are methods to both check the type of the box contents, as well as cast them to the wanted type, providing a continuation closure with the new typed box and its access permission. These methods (`isBoxOf[T]` and `asBoxOf[T]`) are used to convert the box parameter of type `Box[Any]` to a `Box[WorkMessage]` that the `sendWork` method accepts.

The probably more common case is that opening boxes and mutating the contents is exactly what is desired. Acting on the contents and sending more messages to other actors is perfectly legal to do inside the box open method. In the worker actor of `NQUEEN[BOX]`, this pattern is

```
override def receive(msg: Box[Any])(implicit acc:
    msg.Access): Unit =
  if (msg.isBoxOf[WorkMessage]) {
    msg.asBoxOf[WorkMessage] { packed =>
      sendWork(packed.box)(packed.access)
    }
  }
```

Listing 6.11: Box receive for the master actor in `NQUEENK[BOX]`.

used to recursively calculate a solution, sending messages to other actors as necessary, see Listing 6.12. (The contents of the two methods that are called are irrelevant other than the fact that they do not mutate any state.)

```
override def receive(msg: Box[Any])(implicit acc:
    msg.Access): Unit = msg open {
  case msg: WorkMessage
    if size == msg.depth || msg.depth >= threshold =>
    nQueensKernelSeq(msg.data, msg.depth)
    master ! DoneMessage()
  case msg: WorkMessage =>
    nQueensKernelPar(msg)
}
```

Listing 6.12: Box receive for the worker actor in `NQUEENK[BOX]`.

Discussion Overall, the most important distinction between the new box receives and the regular receives is that they need to be declared separately. This can be an issue in terms of readability since the behavior in an actor will be split between two methods instead of only one. A potential way to reduce the impact of that issue is to receive the boxed message and minimize the work done in that receive method, to instead forward `Safe` custom messages to itself and the `Safe` receive method. No such patterns were needed in this case, since the non-`Safe` message in `NQUEENK` is standalone and neither affects, nor is affected by other messages.

6.5 Adapt control flow to boxes

Recall from Section 2.3.3 that boxes require a continuation-passing style (CPS) to enforce affinity at runtime, together with an exception-handled control flow. LaCasa already provides some helper functions to do regular control-flow related activities, like looping. Additionally, the adapter provides its own set of methods that help continued execution via CPS after, e.g., sending a box (i.e., transferring its ownership to the receiving actor).

Example Consider a modified excerpt from `NQUEENK` that loops through an iterator and creates and sends a message out of each element to another actor, in Listing 6.13. It works exactly as regular method calls would work, and it is easily understood what it does.

```
validBoards.foreach {
  master ! WorkMessage(newPriority, _, newDepth)
}
master ! DoneMessage()
```

Listing 6.13: Looping over an iterator while sending messages, and finally sending a message once the looping is done. Example modified from `NQUEENK`

In `NQUEENK[BOX]`, the same location in the code needs to use the LaCasa-provided looping construct, and additionally needs to provide a continuation closure (i.e., CPS) to send a message once it is done looping (see Listing 6.14).

Similarly, consider an excerpt of another method from `NQUEENK` in Listing 6.15. It accesses the actor-local list of workers and sends the message to one of them. Then, it updates the counter to ensure that a new worker is selected the next time, for the next message. It also updates the actor-local counter for how much work has been sent.

The same method in `NQUEENK[BOX]`, Listing 6.16, has a completely different signature to be able to accept a box with the message instead. Additionally, it no longer uses the `!` operator to send the message, instead opting for the `tellAndThen` method which sends the message and then accepts a continuation closure (again, CPS) to continue execution after the message is sent. In this continuation closure, the same things that are done in `NQUEENK` are done as well.

```
loopAndThen(validBoards)({ newData =>
  Box.mkBoxFor(
    WorkMessage(newPriority, newData, newDepth)) { packed =>
    implicit val acc = packed.access
    master !! packed.box
  }
}) { () => // continuation closure
  master ! DoneMessage()
}
```

Listing 6.14: Looping over an iterator while creating and sending boxes, and finally sending a message once the looping is done. Example taken from `NQUEENK[BOX]`.

```
def sendWork(workMessage: WorkMessage) {
  workers(messageCounter) ! workMessage
  messageCounter = (messageCounter + 1) % numWorkers
  numWorkSent += 1
}
```

Listing 6.15: The method in `Safe NQUEENK` for sending work to worker actors.

```
def sendWork(msg: Box[WorkMessage])(implicit acc:
    msg.Access): Unit = {
  workers(messageCounter).tellAndThen(msg) { () =>
    messageCounter = (messageCounter + 1) % numWorkers
    numWorkSent += 1
  }
}
```

Listing 6.16: The method in `NQUEENK[BOX]` for sending work to worker actors.

Discussion The control flow is explicit in the form of continuation closures (and generally CPS), but it at least looks similar to the regular Scala control flow. Explicit control-flow-handling is an unfortunate effect of LaCasa's design (there is work to address that, see Section 7.2.2), but as long as it is limited to one level of CPS, it looks pretty similar to the regular control flow.

Chapter 7

Discussion

This section details what could have been done differently in the thesis and provides rationale for why some decisions were made.

7.1 Evaluation

If the selection of programs had been more diverse, other features of the adapter might have needed changing to accommodate for those patterns. In combination with the chosen metrics, the view of the adapter as presented in this thesis is biased towards the selected suite of programs. That is an inherent issue of empirical studies, and could only potentially be solved by a wider range of program sources. However, the other reasonable sources found – Apache Gearpump [27] and Signal/Collect [24] – were too big, complex, or otherwise had features that did not make it feasible to include in the study. Additionally, the Savina benchmark suite targets a wide selection of program types, and therefore did provide a somewhat diverse set of programs. It also allowed performance to be evaluated.

7.2 Future work

There is a variety of directions to take this project. Below, a few of the most interesting directions are discussed.

7.2.1 Support more patterns

The purpose of this thesis was not to support as many Akka patterns as possible, and there are therefore patterns that are not currently supported, but should be possible to implement. The more patterns that are supported, the less work necessary – on average – to migrate an application.

7.2.2 Expressions following `Nothing`

When using functions like `mkBox`, any following expressions will not be executed. This might be surprising at first, and some interactions with blocks and no-argument closures can cause confusion as well. By introducing a warning for expressions following these kinds of functions, the user can feel secure in knowing that there will be warnings for expressions that will not execute at all – and not just quietly fail.

Another more promising approach would be to introduce flow-sensitive LaCasa [22] to the adapter. That way, the user would both gain convenience in terms of limiting increasing indentation levels due to the CPS nature of LaCasa, as well as reduce the possibility to add code that will never be reached.

7.2.3 Automatically implement `Safe` for deeply immutable types

For types that can be identified as deeply immutable, they should be automatically marked as `Safe`. This change would significantly reduce the amount of manual labor required for marking conforming types as `Safe`. Remember that most case classes are deeply immutable. Additionally, they are the common way to declare a message in Akka. Thus, for a lot of messages, the need to manually mark them as `Safe` can possibly be removed entirely.

7.3 Conclusion

This thesis has presented an adapter, introducing LaCasa to the Akka ecosystem, and with that object capabilities and uniqueness to isolate actors at compile time. Thus, for the main research question of whether it is possible to create an adapter for that purpose, the answer is *yes*.

Furthermore, as demonstrated in Chapters 5 and 6, for a lot of scenarios the required effort definitely makes it *feasible to use the adapter for existing applications*. This exceeds the expectations as stated in the hypothesis and paves way for further work in improving the usability of the adapter in even more scenarios.

7.3.1 Availability

The source code for the migrated programs, LaCasa, the adapter and the benchmarks are available at `https://github.com/fsommar/lacasa`.

Bibliography

[1] Gul A. Agha. *Actors : a model of concurrent computation in distributed systems*. English (US). MIT Press, 1986. ISBN: 9780262010924.

[2] Roger Alsing et al. *Akka.NET*. [Accessed Apr 2017]. 2017. URL: http://getakka.net/.

[3] Brian Anderson et al. "Experience report: Developing the Servo web browser engine using Rust". In: *arXiv preprint arXiv:1505.07383* (2015).

[4] Stephan Brandauer et al. "Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore". In: *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. Ed. by Marco Bernardo and Einar Broch Johnsen. Cham: Springer International Publishing, 2015, pp. 1–56. ISBN: 978-3-319-18941-3. DOI: 10.1007/978-3-319-18941-3_1. URL: https://doi.org/10.1007/978-3-319-18941-3_1.

[5] Richard Cimler, Ondřej Doležal, and Pavel Pscheidl. "Comparison of RUST and C# as a Tool for Creation of a Large Agent-Based Simulation for Population Prediction of Patients with Alzheimer's Disease in EU". In: *International Conference on Computational Collective Intelligence*. Springer. 2016, pp. 252–261.

[6] Dave Clarke et al. "Minimal Ownership for Active Objects". In: *Programming Languages and Systems*. Ed. by G. Ramalingam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–154. ISBN: 978-3-540-89330-1.

[7] Sylvan Clebsch et al. "Deny capabilities for safe, fast actors". In: *Proceedings of the 5th International Workshop on Programming*

Based on Actors, Agents, and Decentralized Control. ACM. 2015, pp. 1–12.

[8] Manuel Fähndrich et al. "Language support for fast and reliable message-based communication in Singularity OS". In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM. 2006, pp. 177–190.

[9] Eric Freeman et al. *Head First Design Patterns: A Brain-Friendly Guide*. " O'Reilly Media, Inc.", 2004. ISBN: 9780596007126.

[10] Philipp Haller. "On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective". In: *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*. AGERE! 2012. Tucson, Arizona, USA: ACM, 2012, pp. 1–6. ISBN: 978-1-4503-1630-9. DOI: 10.1145/2414639.2414641. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/2414639.2414641.

[11] Philipp Haller and Ludvig Axelsson. "Quantifying and Explaining Immutability in Scala". In: *arXiv preprint arXiv:1704.03095* (2017).

[12] Philipp Haller and Alex Loiko. "LaCasa: Lightweight affinity and object capabilities in Scala". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM. 2016, pp. 272–291.

[13] Philipp Haller and Martin Odersky. "Capabilities for uniqueness and borrowing". In: *European Conference on Object-Oriented Programming*. Springer. 2010, pp. 354–378.

[14] Philipp Haller and Fredrik Sommar. "Towards an Empirical Study of Affine Types for Isolated Actors in Scala". In: *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*. 2017, pp. 3–9. DOI: 10.4204/EPTCS.246.3.

[15] Shams M Imam and Vivek Sarkar. "Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries". In: *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. ACM. 2014, pp. 67–80.

[16] Amit Levy et al. “Ownership is theft: experiences building an embedded OS in Rust”. In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. ACM. 2015, pp. 21–26.

[17] Lightbend Inc. *Akka*. [Accessed Apr 2017]. 2017. URL: `http://akka.io`.

[18] Lightbend Inc. *Akka Typed*. [Accessed Oct 2018]. 2018. URL: `https://doc.akka.io/docs/akka/2.5/typed/index.html`.

[19] Mark Samuel Miller. “Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control”. PhD thesis. Baltimore, Maryland, USA: Johns Hopkins University, May 2006.

[20] Stas Negara, Rajesh K. Karmani, and Gul A. Agha. “Inferring ownership transfer for efficient message passing”. In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*. ACM, 2011, pp. 81–90. DOI: `10.1145/1941553.1941566`.

[21] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, 3rd Edition*. Artima Press, 2016. ISBN: 9780981531687.

[22] Erik Reimers. *Lightweight Software Isolation via Flow-Sensitive Capabilities in Scala*. 2017.

[23] Sriram Srinivasan and Alan Mycroft. “Kilim: Isolation-Typed Actors for Java”. In: *ECOOP 2008 – Object-Oriented Programming*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 104–128. ISBN: 978-3-540-70592-5.

[24] Philip Stutz, Abraham Bernstein, and William Cohen. “Signal/-collect: graph algorithms for the (semantic) web”. In: *International Semantic Web Conference*. Springer. 2010, pp. 764–780.

[25] Herb Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb’s journal* 30.3 (2005), pp. 202–210.

[26] The Actix Team. *Actix – Actor System and Web Framework for Rust*. [Accessed Oct 2018]. 2018. URL: `https://actix.rs`.

[27] The Apache Software Foundation. *Apache Gearpump: a real-time big data streaming engine*. [Accessed Sep 2018]. 2017. URL: `https://gearpump.apache.org`.

TRITA -EECS-EX-2018:747