Towards an Empirical Study of Affine Types for Isolated Actors in Scala

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Fredrik Sommar
KTH Royal Institute of Technology
Stockholm, Sweden
fsommar@kth.se

LaCasa is a type system and programming model to enforce the object capability discipline in Scala, and to provide affine types. One important application of LaCasa's type system is software isolation of concurrent processes. Isolation is important for several reasons including security and data-race freedom. Moreover, LaCasa's affine references enable efficient, by-reference message passing while guaranteeing a "deep-copy" semantics. This deep-copy semantics enables programmers to seamlessly port concurrent programs running on a single machine to distributed programs running on large-scale clusters of machines.

This paper presents an integration of LaCasa with actors in Scala, specifically, the Akka actor-based middleware, one of the most widely-used actor systems in industry. The goal of this integration is to statically ensure the isolation of Akka actors. Importantly, we present the results of an empirical study investigating the effort required to use LaCasa's type system in existing open-source Akka-based systems and applications.

1 Introduction

The desire for languages to catch more errors at compile time seems to have increased in the last couple of years. Recent languages, like Rust [15], show that a language does not have to sacrifice a lot, if any, convenience to gain access to safer workable environments. Entire classes of memory-related bugs can be eliminated, statically, through the use of affine types. In the context of this paper it is important that affine types can also enforce isolation of concurrent processes.

LaCasa [5] shows that affine types do not necessarily need to be constrained to new languages: it introduces affine types for Scala, an existing, widely-used language. LaCasa is implemented as a compiler plugin for Scala 2.11.[1] However, so far it has been unclear how big the effort is to apply LaCasa in practice. This paper is a first step to investigate this question empirically on open-source Scala programs using the Akka actor framework [8].

Contributions This paper presents an integration of LaCasa and Akka. Thus, our integration enforces isolation for an existing actor library. Furthermore, we present the results of an empirical study evaluating the effort to use isolation types in real applications. To our knowledge it is the first empirical study evaluating isolation types for actors in a widely-used language.

Selected Related Work Active ownership [3] is a minimal variant of ownership types providing race freedom for active objects while enabling by-reference data transfer between active objects. The system is realized as an extended subset of Java. Kilim [14] combines static analysis and type checking to

[1] See https://github.com/phaller/lacasa

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 3–9, doi:10.4204/EPTCS.246.3

provide isolated actors in Java. For neither of the two above systems, active ownership and Kilim, have the authors reported any empirical results on the syntactic overhead of the respective systems, unlike the present paper. SOTER [12] is a static analysis tool which infers if the content of a message is compatible with an ownership transfer semantics. This approach is complementary to a type system which enables developers to require ownership transfer semantics. Pony [4] and Rust [15] are new language designs with type systems to ensure data-race freedom in the presence of zero-copy transfer between actors/concurrent processes. It is unclear how to obtain empirical results on the syntactic overhead of the type systems of Pony or Rust. In contrast, LACASA extends an existing, widely-used language, enabling empirical studies.

2 Background

In this paper we study affine types as provided by LACASA [5], an extension of the Scala programming language. LACASA is implemented as a combination of a compiler plugin for the Scala 2.11.x compiler and a small runtime library. LACASA provides affine references which may be consumed at most once. In LACASA an affine reference to a value of type `T` has type `Box[T]`. The name of type constructor `Box` indicates that access to an affine reference is restricted. Accessing the wrapped value of type `T` requires the use of a special `open` construct:

```
1  val box: Box[T] = ...
2  box open { x => /* use ‘x‘ */ }
```

`open` is implemented as a method which takes the closure `{ x => /* use ‘x‘ */ }` as an argument. The closure body then provides access to the object `x` wrapped by the `box` of type `Box[T]`. However, LACASA restricts the environment (*i.e.,* the captured variables) of the argument closure in order to ensure affinity: mutable variables may not be captured. Without this restriction it would be simple to duplicate the wrapped value, violating affinity:

```
1  val box: Box[T] = ...
2  var leaked: Option[T] = None
3  box open { x =>
4    leaked = Some(x) // illegal
5  }
6  val copy: T = leaked.get
```

LACASA also protects against leaking wrapped values to global state:

```
1  object Global { var cnt: LeakyCounter = null }
2  class LeakyCounter {
3    var state: Int = 0
4    def increment(): Unit = { state += 1 }
5    def leak(): Unit = { Global.cnt = this }
6    ...
7  }
8  val box: Box[LeakyCounter] = ... // illegal
9  box open { cnt =>
10   cnt.leak()
11 }
12 val copy: LeakyCounter = Global.cnt
```

The above `LeakyCounter` class is illegal to be wrapped in a box. The reason is that even without capturing a mutable variable within `open`, it is possible to create a copy of the counter, because the `leak` method leaks a reference to the counter to global mutable state (the `Global` singleton object). To prevent

```
1   def m[T](b: Box[T])(implicit p: CanAccess { type C = b.C }): Unit = {
2     b open { x => /* use `x` */ }
3   }
```

Figure 1: Boxes and permissions in LaCasa.

```
1  class Box[T] { self =>
2    type C
3    def open(fun: T => Unit)
4            (implicit p: CanAccess { type C = self.C }): Box[T] = {
5      ...
6    }
7  }
```

Figure 2: Type signature of the `open` method.

this kind of affinity violation, LaCasa restricts the creation of boxes of type `Box[A]` to types `A` which conform to the object capability discipline [11]. According to the object capability discipline, a method `m` may only use object references that have been passed explicitly to `m`, or `this`. Concretely, accessing `Global` on line 5 is illegal, since `Global` was not passed explicitly to method `leak`.

In previous work [5] we have formalized the object capability discipline as a type system and we have shown that in combination with LaCasa's type system, affinity of box-typed references is ensured.

Affine references, *i.e.,* references of type `Box[T]`, may be consumed, causing them to become unaccessible. Consumption is expressed using *permissions* which control access to box-typed references. Consuming an affine reference consumes its corresponding permission.

Ensuring at-most-once consumption of affine references thus requires each permission to be linked to a specific box, and this link must be checked statically. In LaCasa permissions are linked to boxes using path-dependent types [2]. For example, Figure 1 shows a method `m` which has two formal parameters: a box `b` and a permission `p` (its `implicit` modifier may be ignored for now). The type `CanAccess` of permissions has a *type member* `C` which is used to establish a static link to box `b` by requiring the equality `type C = b.C` to hold. The type `b.C` is a path-dependent type with the property that there is only a single runtime object, namely `b`, whose type member `C` is equal to type `b.C`. In order to prevent forging permissions, permissions are only created when creating boxes; it is impossible to create permissions for existing boxes.

Since permissions may be consumed (as shown below), it is important that opening a box requires its permission to be available. Figure 2 shows how this is ensured using an *implicit parameter* [13] of the `open` method (line 5). Note that the shown type signature is simplified; the actual signature uses a *spore* type [10] instead of a function type on line 4 to ensure that the types of captured variables are immutable.

Consuming Permissions Permissions in LaCasa are just Scala implicit values. This means their availability is flow-insensitive. Therefore, changing the set of available permissions requires changing scope. In LaCasa, calling a permission-consuming method requires passing an explicit continuation closure. The LaCasa type checker enforces that the consumed permission is then no longer available in the scope of this continuation closure. Figure 3 shows an example. LaCasa enforces that such continuation-passing methods do not return (see [5]), indicated by Scala's bottom type, `Nothing`.

```
1   def m[T](b: Box[T])(cont: () => Unit)(implicit p: CanAccess { type C = b.C }): Nothing = {
2     b open { x => /* use `x` */ }
3     consume(b) {
4       // explicit continuation closure
5       // permission `p` unavailable
6       ...
7       cont() // invoke outer continuation closure
8     }
9   }
```

Figure 3: Consuming permissions in LACASA.

```
1  class ExampleActor extends Actor {
2    def receive = {
3      case msgpat_1 =>
4      ...
5      case msgpat_n =>
6    }
7  }
```

Figure 4: Defining actor behavior in Akka.

2.1 Akka

Akka [8] is an implementation of the actor model [6, 1] for Scala. Actors are concurrent processes communicating via asynchronous messages. Each actor buffers received messages in a local "mailbox" – a queue of incoming messages. An Akka actor processes at most one incoming message at a time. Figure 4 shows the definition of an actor's behavior in Akka. The behavior of each actor is defined by a subclass of a predefined `Actor` trait. The `ExampleActor` subclass implements the `receive` method which is abstract in trait `Actor`. The `receive` method returns a message handler defined as a block of pattern-matching cases. This message handler is used to process each message in the actor's mailbox. The `Actor` subclass is then used to create a new actor as follows:

```
1   val ref: ActorRef = system.actorOf(Props[ExampleActor], "example-actor")
2   ref ! msg
```

The result of creating a new actor is a reference object (`ref`) of type `ActorRef`. An `ActorRef` is a handle that can be used to send asynchronous messages to the actor using the `!` operator (line 2).

3 Integrating LACASA and Akka

The Adapter The LaCasa-Akka adapter[2] is an extension on top of Akka. During its design, an important constraint was to keep it separate from Akka's internals – primarily to limit the effect of internal changes as Akka evolves.

The adapter consists of two parts: `SafeActor[T]` and `SafeActorRef[T]`, both with the same responsibilities as their counterparts in the Akka API. However, note that in contrast to the latter, they are generic over the message type. Akka instead relies on pattern matching to discern the types of received

[2] See `https://github.com/fsommar/lacasa/tree/akka`

```
1 trait SafeActor[T] extends Actor {
2   def receive(msg: Box[T])(implicit acc: CanAccess { type C = msg.C }): Unit
3 }
```

Figure 5: Usage of LaCasa's boxes and permissions in `SafeActor`.

Program	LOC (Scala/Akka)	LOC (LaCasa/Akka)	Changes	Changes (%)
ThreadRing	130	153	27 add./10 del.	28.5%
Chameneos	143	165	26 add./7 del.	23.1%
Banking	118	135	27 add./12 del.	33.1%
Average	130	151		28.2%

Table 1: Results of the empirical study.

messages (see Section 2.1). For the LaCasa-Akka adapter, however, it is necessary to know the types of messages at compile time, to prevent the exchange of unsafe message types.

SafeActor A subclass of Akka's `Actor`, `SafeActor` provides a different `receive` method signature, which is the primary difference between the two. Instead of receiving an untyped message, of type `Any`, `SafeActor[T]` receives a boxed message of type `T`, and an access permission for the contents of the box (see Figure 5).

SafeActorRef The API for `SafeActorRef` is a wrapper of Akka's `ActorRef`, and contains a subset of the latter's methods and functionality. It uses the same method names, but method signatures are different, to include necessary safety measures. For every method accepting a box, there is a dual method accepting a box and a continuation closure. Recall that it is the only way to enforce that boxes are consumed (see Section 2). The dual methods use the `AndThen` suffix to indicate that they accept a continuation closure.

For message types that are immutable, the API can be significantly simplified, resembling that of a regular Akka `ActorRef`. Meanwhile, internally, the message is still boxed up and forwarded for handling by the `SafeActor`. Importantly, though, the box does not have to be consumed, enabling the method to return and continue execution – removing the need for the `AndThen` family of methods.

4 Empirical Study

We converted several Scala/Akka programs to use the LaCasa-Akka adapter described in Section 3. The goal of this conversion is to evaluate the effort required to use LaCasa's type system in practice. The converted programs are part of the Savina actor benchmark suite [7]. Concretely, we converted the following programs: (1) In `ThreadRing`, an integer token message is passed around a ring of N connected actors. This benchmark is adopted from Theron [9]; (2) `Chameneos` is a micro-benchmark measuring the effects of contention on shared resources while processing messages; (3) `Banking` is a bank transaction micro-benchmark measuring synchronous request-response with interfering transactions.

Table 1 shows the results. On average 28.2% of the lines of code of each program needed to be changed (we exclude changes to imports). It is important to note that we expect this number to be significantly lower for larger applications where sequential, non-actor-based code dominates the code

base. The most important reasons for code changes are (a) the declaration of safe message classes and (b) the insertion and removal of messages into/from boxes. For example, in `ThreadRing` 33.3% of added lines are due to declaring message classes as safe.

5 Conclusion

LaCasa extends Scala's type system with affine types, with applications to race-free concurrent programming and safe off-heap memory management. This paper shows how LaCasa can ensure the isolation of actors in Akka, a widely-used actor framework for Scala, while providing safe and efficient ownership transfer of asynchronous messages. According to our empirical study, adjusting existing Akka-based Scala programs requires changing 28.2% of the lines of code on average. However, this initial result represents a worst-case scenario, since the study only considered micro-benchmarks where actor-related code dominates, unlike larger real-world applications. An empirical study extending our results to medium-to-large open-source code bases is ongoing.

References

[1] Gul A. Agha (1986): *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence, The MIT Press, Cambridge, Massachusetts.

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf & Sandro Stucki (2016): *The Essence of Dependent Object Types*. In: *A List of Successes That Can Change the World*, Springer, pp. 249–272, doi:10.1007/978-3-319-30936-1_14.

[3] Dave Clarke, Tobias Wrigstad, Johan Östlund & Einar Broch Johnsen (2008): *Minimal Ownership for Active Objects*. In: *APLAS*, Springer, pp. 139–154, doi:10.1007/978-3-540-89330-1_11.

[4] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing & Andy McNeil (2015): *Deny capabilities for safe, fast actors*. In: *AGERE!@SPLASH*, ACM, pp. 1–12, doi:10.1145/2824815.2824816.

[5] Philipp Haller & Alex Loiko (2016): *LaCasa: Lightweight affinity and object capabilities in Scala*. In: *OOPSLA*, ACM, pp. 272–291, doi:10.1145/2983990.2984042.

[6] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *IJCAI*, William Kaufmann, pp. 235–245.

[7] Shams Mahmood Imam & Vivek Sarkar (2014): *Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries*. In: *AGERE!@SPLASH*, ACM, pp. 67–80, doi:10.1145/2687357.2687368.

[8] Lightbend, Inc. (2009): *Akka*. http://akka.io/.

[9] Ashton Mason (2012): *The ThreadRing benchmark*. http://www.theron-library.com/index.php?t=page&p=threadring.

[10] Heather Miller, Philipp Haller & Martin Odersky (2014): *Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution*. In: *ECOOP*, Springer, pp. 308–333, doi:10.1007/978-3-662-44202-9_13.

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

[12] Stas Negara, Rajesh K. Karmani & Gul A. Agha (2011): *Inferring ownership transfer for efficient message passing*. In: *PPOPP*, ACM, pp. 81–90, doi:10.1145/1941553.1941566.

[13] Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee & Kwangkeun Yi (2012): *The implicit calculus: a new foundation for generic programming*. In: *PLDI*, ACM, pp. 35–44, doi:10.1145/2254064.2254070.

[14] Sriram Srinivasan & Alan Mycroft (2008): *Kilim: Isolation-Typed Actors for Java*. In: *ECOOP*, Springer, pp. 104–128, doi:10.1007/978-3-540-70592-5_6.

[15] Aaron Turon (2017): *Rust: from POPL to practice (keynote)*. In: *POPL*, ACM, p. 2, doi:10.1145/3009837.3011999.