Abstract

Aliasing is a known source of challenges in the context of imperative object-oriented languages, which have led to important advances in type systems for aliasing control. However, their large-scale adoption has turned out to be a surprisingly difficult challenge. While new language designs show promise, they do not address the need of aliasing control in existing languages.

This paper presents a new approach to isolation and uniqueness in an existing, widely-used language, Scala. The approach is unique in the way it addresses some of the most important obstacles to the adoption of type system extensions for aliasing control. First, adaptation of existing code requires only a minimal set of annotations. Only a single bit of information is required per class. Surprisingly, the paper shows that this information can be provided by the object-capability discipline, widely-used in program security. We formalize our approach as a type system and prove key soundness theorems. The type system is implemented for the full Scala language, providing, for the first time, a sound integration with Scala’s local type inference. Finally, we empirically evaluate the conformity of existing Scala open-source code on a corpus of over 75,000 LOC.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs

Keywords Aliasing, uniqueness, object capabilities, Scala

1. Introduction

Uncontrolled aliasing in imperative object-oriented languages introduces a variety of challenges in large-scale software development. Among others, aliasing can increase the difficulty of reasoning about program behavior and software architecture. It can also introduce data races in concurrent programs. These observations have informed the development of a number of type disciplines aimed at providing static aliasing properties, such as linear types, region inference, unique references, and ownership types.

While there have been important advances in the flexibility and expressiveness of type systems for aliasing control, large-scale adoption has been shown to be a much greater challenge than anticipated. Recent efforts in the context of new language designs like Rust are promising, but they do not address the increasing need for aliasing control in existing, widely-used languages.

One of the most important obstacles to the adoption of a type system extension in a widely-used language with a large ecosystem is the adaptation of existing code, including third-party libraries. Typically, adaptation consists of adding annotations required by the type system extension. With a large ecosystem of existing libraries, this may be prohibitively expensive even for simple annotations. A second, and almost equally critical obstacle is robust support for the entirety of an existing language’s type system in a way that satisfies requirements for backward compatibility.

This paper presents a new approach to integrating a flexible type system for isolation and uniqueness into an existing, full-featured language, Scala. Our approach minimizes the annotations necessary for reusing existing code in a context where isolation and uniqueness is required. In the presented system, a single bit of information is enough to decide whether an existing class supports isolation and uniqueness. A key insight of our approach is that this single bit of information is provided by the object-capability discipline. The object capability model is an established methodology in the context of program security, and has been proven in large-scale industrial use for secure sandboxing of JavaScript applications.

This paper makes the following contributions:

- We present a new approach to separation and uniqueness which aims to minimize the annotations necessary to reuse existing code (Section 2). In our system, reusability is based on the object capability model. Thus, when
annotating existing code bases, only a single bit of information is required per class.

- We formalize our approach in the context of two object-oriented core languages (Section 5). The first core language formalizes a type-based notion of object capabilities. The second core language additionally provides external uniqueness via flow-insensitive permissions.

- We provide complete soundness proofs, formally establishing heap separation and uniqueness invariants for our two core languages (Section 4). Moreover, a concurrent extension enables the formation of an isolation theorem for processes in the presence of a shared heap and efficient, by-reference message passing (Section 4.1). We have also mechanized the operational semantics and type system of the first core language in Coq (see the companion technical report [37]).

- We implement our approach for the full Scala language as a compiler plugin (Section 5). To our knowledge, our implementation of (external) uniqueness is the first to integrate soundly with local type inference in Scala. Moreover, the implementation leverages a unique combination of previous proposals for (a) implicit parameters [24, 25], and (b) closures with capture control [31, 45].

- We empirically evaluate the conformity of existing Scala classes to the object capability model on a corpus of over 75,000 LOC of popular open-source projects (Section 6). Results show that between 21% and 79% of the classes of a project adhere to a strict object capability discipline.

In the following we discuss the most closely related work, and defer a discussion of other related work to Section 7. In Section 8 we conclude.

Selected Related Work. Most closely related to our system are approaches based on permissions or capabilities. Of particular relevance is previous work on capabilities for uniqueness in Scala by Haller and Odersky [39] (“Cap4S”). While this prior work shares our high-level goal of lightweight unique references in Scala, the two approaches are significantly different, with important consequences concerning soundness, robustness, and compatibility. First, Cap4S is based on flow-sensitive capabilities which are modeled using Scala’s annotations, similar to the use of extended type annotations in Java 8 for pluggable type systems [29]. However, the interaction between Scala’s local type inference [53] and annotation propagation has been shown to be a source of unsoundness and implementation complexities for such pluggable type systems [58]; these challenges are exacerbated in flow-sensitive type systems. In contrast, LACASA models capabilities using Scala’s implicit parameters [24], an intrinsic part of type inference in Scala. In addition, foundations of implications have been studied [25], whereas Scala’s annotations remain poorly understood. Second, LACASA fundamentally simplifies type checking: as long as a class conforms to the object-capability model, LACASA’s constructs enable isolation and uniqueness for instances of the class. This has two important consequences: (a) a minimal set of additional annotations (a single bit of information per class) enables reusing existing code, and (b) type checking reusable class declarations is simple and well-understood, following the object-capability discipline, which we adapt for Scala.

2. Overview

We proceed with an informal overview of LACASA: its programming model in Scala, and its type system.

A First Example. Consider the case of asynchronous communication between two concurrent processes. This style of concurrency is well-supported by the actor model [2, 40] for which multiple implementations exist for Scala [36, 38, 43]. Figure 1 shows the definition of two actor classes. The behavior of each actor is implemented by overriding the receive method inherited from superclass Actor. The receive method is invoked by the actor runtime system whenever an actor is ready to process an incoming message. In the example, whenever ActorA has received an instance of class Start, it creates an instance of class Message, initializes the instance with an integer array, and sends the instance to the next actor.

![Figure 1. Two communicating actors in Scala.](https://i.imgur.com/12345.png)

1 LACASA is available under an open-source license at [https://github.com/phaller/lacasa/](https://github.com/phaller/lacasa/)

2 In favor of clarity of explanation, Figure 1 shows hypothetical Scala code which requires slight changes for compilation with the Akka library.
Note that field next of class Start has type ActorRef[Message] (line 27) instead of Actor[Message]. An ActorRef serves as an immutable and serializable handle to an actor. The public interface of ActorRef is minimal; its only purpose is to provide methods for asynchronously sending messages to the ActorRef's underlying actor (an instance of a subclass of Actor). The purpose of ActorRef as a type separate from Actor is to provide a fault handling model similar to Erlang \[31\]. In this model, a faulty actor may be restarted in a way where its underlying Actor instance is replaced with a new instance of the same class. Importantly, any ActorRef referring to the actor that is being restarted switches to using the new Actor instance in a way that is transparent to clients (which only depend on ActorRefs). This enables introducing fault-handling logic in a modular way (cf. Erlang’s OTP library \[32\]).

The shown program suffers from multiple safety hazards: first, within the leak method (line 19), the array of the current Message instance is stored in the global singleton object SomeObject (line 20); thus, subsequently, multiple actors could access the array through SomeObject concurrently. Second, after sending newMsg to ActorB (line 6), ActorA mutates the array contained in newMsg (line 7); this could lead to a data race, since ActorB may be accessing newMsg.arr at this point.

LACASA prevents the two safety hazards of the example using two complementary mechanisms: object capabilities and affine access permissions. Figure 2 shows the same example written in LACASA. LACASA introduces two main changes to the Actor and ActorRef library classes:

1. Actors send and receive boxes of type Box[T], rather than direct object references. As explained in the following, LACASA's type system enforces strong encapsulation properties for boxes.

2. The type of the receive method is changed to additionally include an implicit permission parameter. (We explain implicit permissions in detail below.)

Due to these changes, LACASA provides its own versions of the Actor and ActorRef library classes: 2

Boxes. A box of type Box[T] encapsulates a reference to an object of type T. However, this reference is only accessible using an open method: box.open({ x => ... }); here, x is an alias of the encapsulated reference. For example, on line 22 ActorB opens the received box in order to print the array of the Message instance. Note that the use of open on lines 4–16 relies on Scala’s syntax for partial functions: a block of case clauses.

\[3\] Scala’s original actor implementation \[33\] only provided an Actor type; the distinction between Actor and ActorRef was introduced with the adoption of Akka as Scala’s standard actor implementation.

\[4\] In ongoing work we are developing adapter classes to conveniently integrate LACASA and the Akka actor library.

---

```scala
class ActorA extends Actor[Message] {
  // ...
  override def receive(box: Box[Message]) {
    // ...
    case s: Start =>
      mbox[Message] { packed =>
        val access = packed.access
        packed.box.open({
          msg =>
            msg.arr = Array(1, 2, 3, 4)
        })(access)
        s.next.send(packed.box){
          // ...
        })(access)
        // ...
    }
    // ...
  }
  // ...
}

class ActorB extends Actor[Message] {
  // ...
  override def receive(box: Box[Message]) {
    // ...
  }
  // ...
}
```

---

**Figure 2.** Two communicating actors in LACASA.

```scala
case pat1 => e1
... case patn => en
```

creates a partial function with the same run-time semantics as the function

```scala
x => x match {
  case pat1 => e1
  ... case patn => en
}
```

In combination with LACASA’s type system, boxes enforce constraints that directly prevent the first safety hazard in the previous example. Boxes may only encapsulate instances whose classes follow the object-capability discipline. Roughly speaking, the object-capability discipline prevents an object obj from obtaining references that were not explicitly passed to obj via constructor or method calls; in particular, it is illegal for obj to access shared, global singleton objects like SomeObject. As a result, the problematic leak (line 20 in Figure 1) causes a compilation error.

**Capture Control.** In general, the requirement of boxes to encapsulate object-capability safe classes is not sufficient to ensure isolation, as the following example illustrates:

```scala
// box: Box[Message]
var a: Array[Int] = null
box.open({ msg =>
  a = msg.arr
}
```
In this case, by capturing variable \texttt{a} in the body of open, and by making \texttt{a} an alias of the array in \texttt{msg}, it would be possible to access the array even after sending it (inside \texttt{msg} inside box) to next. To prevent such problematic leaks, the body of open is not allowed to capture anything (i.e., it must not have free variables). Furthermore, the body of open is not allowed to access global singleton objects. Thus, both the access to \texttt{a} on line 4 and the access to SomeObject on line 5 cause compilation errors. Finally, the body of open may only create instances of object-capability safe classes to prevent indirect leaks such as on line 20 in Figure 1.

The second safety hazard illustrated in Figure 1, namely accessing a box that has been transferred, is prevented using a combination of boxes, capture control, and access permissions, which we discuss next.

**Access Permissions.** A box can only be accessed (e.g., using open) at points in the program where its corresponding access permission is in scope. Box operations take an extra argument which is the permission required for accessing the corresponding box. For example, the open invocation on lines 22–24 in Figure 2 takes the acc permission as an argument (highlighted) in addition to the closure. Note that acc is passed within a separate argument list. The main reason for using an additional argument list instead of just an additional argument is the use of implicits to reduce the syntactic overhead (see below).

The static types of access permissions are essential for alias tracking. Importantly, the static types ensure that an access permission is only compatible with a single Box[T] instance. For example, the acc parameter on line 3 has type CanAccess {type \texttt{C} = box.C} where box is a parameter of type Box[\texttt{Any}]. Thus, the type member \texttt{C} of the permission's type is equal to the type member \texttt{C} of box.

In Scala, box.C is a path-dependent type \[5, 6\]; box.C is equivalent to the type box.type\#C which selects type \texttt{C} from the singleton type box.type. The type box.type is only compatible with singleton types x.type where the type checker can prove that \texttt{x} and box are always aliases. (Thus, in a type box.type, box may not be re-assignable.) Access permissions in LACASA leverage this aliasing property of singleton types: since it is impossible to create a box \texttt{b} such that \texttt{b.C} is equal to the type member \texttt{C} of an existing box, it follows that an access permission is only compatible with at most one instance of Box[T].

The only way to create an access permission is by creating a box using an \texttt{mkBox} expression. For example, the \texttt{mkBox} expression on line 6 in Figure 2 creates a box of type Box[Message] as well as an access permission. Besides a type argument, \texttt{mkBox} also receives a closure of the form \{ packed => ... \}. The closure’s packed parameter encapsulates both the box and the access permission, since both need to be available in the scope of the closure.

Certain operations consume access permissions, causing associated boxes to become unavailable. For example, the message send on line 11 consumes the access permission of packed.box to prevent concurrent accesses from the sender and the receiver. As a result, packed.box is no longer accessible in the continuation of send.

Note that permissions in LACASA are flow-insensitive by design. Therefore, the only way to change the set of available permissions is by entering scopes that prevent access to consumed permissions. In LACASA, this is realized using continuation closures: each operation that changes the set of available permissions also takes a closure that is the continuation of the current computation; the changed set of permissions is only visible in the continuation closure. Furthermore, by discarding the call stack following the execution of a continuation closure, LACASA enforces that scopes where consumed permissions are visible (and therefore “accessible”) are never re-entered. The following LACASA operations discard the call stack: \texttt{mkBox}, send, swap (see below). In contrast, open does not discard the call stack, since it does not change the set of permissions.

In the example, the send operation takes a continuation closure (line 11–13) which prevents access to the permission of packed.box; furthermore, the call stack is discarded, which means that if there was any additional code in the case branch ending at line 14, it would be unreachable.

**Implicit Permissions.** To make sure access permissions do not have to be explicitly threaded through the program, they are modeled using implicits \[24, 25\]. For example, consider the receive method on line 2–3. In addition to its regular box parameter, the method has an acc parameter which is marked as implicit. This means at invocation sites of receive the argument passed to the implicit parameter is optionally inferred (or resolved) by the type checker. Importantly, implicit resolution fails if no type-compatible implicit value is in scope, or if multiple ambiguous type-compatible implicit values are in scope.\footnote{See the Scala language specification \[51\] for details of implicit resolution.} The benefit of marking acc as implicit is that within the body of receive, acc does not have to be passed explicitly to methods requiring access permissions, including LACASA expressions like open. Figure 2 makes all uses of implicits explicit (shaded). This explicit style also requires making parameter lists explicit; for example, consider lines 22–24: box.open({ x => ... })\(\texttt{(acc).} \) In contrast, passing the access permission acc implicitly enables the more lightweight Scala syntax box open \{ x => ... \}.

**Stack Locality.** It is important to note that the above safety measures with respect to object capabilities, capture control, and access permissions could be circumvented by creating heap aliases of boxes and permissions. Therefore, boxes and
permissions are confined to the stack by default. This means, without additional annotations they cannot be stored in fields of heap objects or passed as arguments to constructors.

**Unique Fields.** Strict stack confinement of boxes would be too restrictive in practice. For example, an actor might have to store a box in the heap to maintain access across several invocations of its message handler while enabling a subsequent ownership transfer. To support such patterns LACASA enables boxes to have *unique fields* which store boxes\(^6\) Access is restricted to maintain external uniqueness of unique fields (see Section 5 for a formalization of the uniqueness and aliasing guarantees of unique fields).

The actor in Figure 3 receives a box of type Box[Container] where class Container declares two unique fields, part1 and part2 (line 25–26). These fields are identified as unique fields through their box types; they have to be accessed using a swap expression. swap “removes” the box of a unique field and replaces it with another box. For example, on line 7, swap extracts the part1 field of the msg box and replaces it with packed.box, the box created by mkBox on line 5. The extracted box is accessible via the pack parameter of the subsequent “spore” (see Section 5).

---

6 As a pragmatic extension, LACASA also enables actors to have unique fields; this is safe, since actors are alias-free, in contrast to ActorRef's.

---

\(p ::= \overline{cd v/ t} \quad \overline{cd} ::= \text{class } C \text{ extends } D \{ \overline{vd m/} \} \quad \overline{vd} ::= \text{var } f : C \quad \text{variable} \quad \text{method type}\)

\(\sigma, \tau ::= \text{def } m(x : \sigma) : \tau = t \quad \text{type}\)

\(\quad \quad C, D \quad \text{class type} \quad \text{boxed type} \quad \text{null type}\)

Figure 4. CLC\(^1\) syntax. \(C, D\) range over class names, \(f, m, x\) range over term names.

### 3. Formalization

We formalize the main concepts of LACASA in the context of typed object-oriented core languages. Our approach, however, extends to the whole of Scala (see Section 5). Our technical development proceeds in two steps. In the first step, we formalize simple object capabilities. In combination with LACASA’s boxes and its open construct object capabilities, we enforce an essential heap separation invariant (Section 3.2). In the second step, we extend our first core language with lightweight affinity based on permissions. The extended core language combines permissions and continuation terms to enable expressing (external) uniqueness, ownership transfer, and unique fields. Soundness, isolation, and uniqueness invariants are established based on small-step operational semantics and syntax-directed type rules.

#### 3.1 Object Capabilities

This section introduces CORELACASA\(^1\) (CLC\(^1\)), a typed, object-oriented core language with object capabilities.

**Syntax.** Figure 4 and Figure 5 show the syntax of CLC\(^1\). A program consists of a sequence of class definitions, \(\overline{cd}\), a sequence of global variable declarations, \(\overline{vd}\), and a “main” term \(t\). Global variables model top-level, stateful singleton objects of our realization in Scala (see Section 5). A class \(C\) has exactly one superclass \(D\) and a (possibly empty) sequence of fields, \(\overline{vd}\), and methods, \(\overline{md}\). The superclass may be AnyRef, the superclass of all classes. To simplify the presentation, methods have exactly one parameter \(x\); their body is a term \(t\). There are three kinds of types: class types \(C\), box types \(\text{Box}[C]\), and the Null type. Null is a subtype of all class types; it is used to assign a type to null.

In order to simplify the presentation of the operational semantics, programs are written in A-normal form (ANF) which requires all subexpressions to be named. We enforce ANF by introducing two separate syntactic categories for terms and expressions, shown in Figure 5. Terms are either variables or let bindings. Let bindings introduce names for intermediate results. Most expressions are standard, except that the usual object-based expressions, namely field selections, field assignments, and method invocations, have only variables as trivial subexpressions. The instance creation ex-
expression (new) does not take arguments: all fields of newly-created objects are initialized to null.

Two kinds of expressions are unique to our core language: box[C] creates a box containing a new instance of class C. The expression box[C] has type Box[C]. The expression x.open \{ y ⇒ t \} provides temporary access to box x.

### 3.1.1 Dynamic Semantics

We formalize the dynamic semantics as a small-step operational semantics based on two reduction relations, H, F \xrightarrow{\cdot} H', F', and H, FS \xrightarrow{\cdot} H', FS'. The first relation reduces single (stack) frames F in heap H, whereas the second relation reduces entire frame stacks FS in heap H.

A heap H maps references o ∈ dom(H) to run-time objects \langle C, FM \rangle where C is a class type and FM is a field map that maps field names to values in dom(H) ∪ \{null\}. FS is a sequence of stack frames F; we use the notation FS = F ∘ FS' to indicate that in stack frame F, F is the top-most frame which is followed by frame stack FS'.

A single frame F = \langle L, t \rangle \triangleright consists of a variable environment L = env(F), a term t, and an annotation l. The variable environment L maps variable names x to values v ∈ dom(H) ∪ \{null\} ∪ \{b(o) | o ∈ dom(H)\}. A value b(o) is a box reference created using CLC1's box[C] expression. A box reference b(o) prevents accessing the members of o using regular selection, assignment, and invocation expressions; instead, accessing o’s members requires the use of an open expression to temporarily “borrow” the encapsulated reference. As is common, L' = L[x ↦ v] denotes the updated mapping where L'(y) = L(y) if y ≠ x and L'(y) = v if y = x. A frame annotation l is either empty (or non-existent), expressed as l = ε, or equal to a variable name x. In the latter case, x is the name of a variable in the next frame which is to be assigned the return value of the current frame.

As is common \[42\], fields(C) denotes the fields of class C, and mbody(C, f) = x → t denotes the body of a method def m(x : σ) : τ = t.

\[ H, \langle L, \text{let } x = \text{new } C \rangle t \triangleright H', \langle L[x ↦ o], t \rangle \quad \text{(E-NEW)} \]

\[ H, \langle L, \text{let } x = y \rangle t \triangleright H', \langle L[x ↦ L(y)], t \rangle \quad \text{(E-VAR)} \]

\[ H(L(y)) = \langle C, FM \rangle \quad f ∈ \text{dom}(FM) \]

\[ H, \langle L, \text{let } x = y.f \rangle t \triangleright H, \langle L[x ↦ FM(f)], t \rangle \quad \text{(E-SELECT)} \]

\[ L(y) = o \quad H(o) = \langle C, FM \rangle \]

\[ L' = H[ o ↦ (C, FM[l ↦ L(z)]) ] \]

\[ H, \langle L, \text{let } x = y.f \rangle t \triangleright H', \langle L[x ↦ z], t \rangle \quad \text{(E-ASSIGN)} \]

\[ o \notin \text{dom}(H) \quad \text{fields}(C) = \mathcal{F} \]

\[ H' = H[ o ↦ (C, \mathcal{F} ↦ \text{null}) ] \quad \text{(E-NEW)} \]

\[ H, \langle L, \text{let } x = \text{box}[C] \rangle t \triangleright H', \langle L[x ↦ b(o)], t \rangle \quad \text{(E-BOX)} \]

Figure 5. CLC1 terms and expressions.

Figure 6. CLC1 frame transition rules.

Reduction of a program \( p = \overline{cd \ vd \ t} \) begins in an initial environment \( H_0, F_0 \circ \epsilon \) such that \( H_0 = \{ o_g ↦ \langle C_g, FM_g \rangle \} \) (initial heap), \( F_0 = \langle L_0, t \rangle \) (initial frame), \( L_o = \{ \text{global} ↦ o_g \}, FM_o = \{ x ↦ \text{null} \mid \text{var } x : C \in \overline{vd} \} \), and \( o_g \) a fresh object identifier; \( C_g \) is a synthetic class defined as: \text{class } C_g \text{ extends AnyRef } \{ \overline{vd} \}. Thus, a global variable \text{var } x : C \text{ is accessed using global.x}; we treat global as a reserved variable name.

**Single Frame Reduction.** Figure \[\text{[5]}\] shows single frame transition rules. Thanks to the fact that terms are in ANF in our core language, the reduced term is a let binding in each case. This means reduction results can be stored immediately in the variable environment, avoiding the introduction of locations or references in the core language syntax. Rule E-BOX is analogous to rule E-NEW, except that variable x is bound to a box reference b(o). As a result, fields of the encapsulated object o are not accessible using regular field selection and assignment, since rules E-SELECT and E-ASSIGN would not be applicable. Apart from E-BOX the transition rules are similar to previous stack-based formalizations of class-based core languages with objects \[\text{[11][12][55]}\].

**Frame Stack Reduction.** Figure \[\text{[7]}\] shows the frame stack transition rules. Rule E-INVOCe creates a new frame, annotated with x, that evaluates the body of the called method. Rule E-RETURN1 uses the annotation y of the top-most frame to return the value of x to its caller’s frame. Rule
Well-formedness of Programs

A program is well-formed if all its class definitions are well-formed and its top-level term is well-typed in a type environment \( \Gamma_0 = \{ \text{global} : C_0 \} \) (WF-PROGRAM). Rule WF-CLASS defines well-formed class definitions. In a well-formed class definition (a) all methods are well-formed, (b) the superclass is either AnyRef or a well-formed class in the same program, and (c) method overriding (if any) is well-formed; fields may not be overridden. We use a standard function \( \text{fields}(D) \) \cite{42} to obtain the fields in class \( D \) and superclasses of \( D \). Rule WF-OVERRIDE defines well-formed method overriding: overriding of a method \( m \) in class \( C \) with superclass \( D \) is well-formed if \( D \) (transitively) does not define a method \( m \) or the type of \( m \) in \( D \) is the same as the type of \( m \) in \( C \). A method \( m \) is well-typed in class \( C \) if its body is well-typed with type \( \tau' \) under effect \( \epsilon \) in environment \( \Gamma_0 \), this : \( C : x : \sigma ; \epsilon \vdash t : \tau' \), where \( \sigma \) is the type of \( m \)'s parameter \( x \), such that \( \tau' \) is a subtype of \( m \)'s declared result type \( \tau \) (WF-METHOD).

Object Capabilities. For a class \( C \) to satisfy the constraints of the object-capability discipline, written \( \text{ocap}(C) \), it must be well-formed according to the rules shown in Figure \ref{fig:ocap}

Essentially, for a class \( C \) we have \( \text{ocap}(C) \) if its superclass is ocap, the types of its fields are ocap, and its methods are well-formed according to \( \text{ocap} \). Rule WF-METHOD looks a lot like rule WF-METHOD, but there are two essential differences: first, the method body must be well-typed in a type environment that does not contain the global environment \( \Gamma_0 \); thus, global variables are inaccessible. Second, the method body must be well-typed under effect ocap; this means that within the method body only ocap classes may be instantiated.

Subtyping and Subtypes. In CLC\(^1\), the subtyping relation \( <: \), defined by the class table, is identical to that of FJ \cite{42} except for two additional rules:

\[
\begin{align*}
C & \ <: D & \text{(}\llnull \ <: \sigma \text{ (}\llnull \text{-Null})} \\
\Box[C] & \ <: \Box[D] & \text{(}\llbox \text{-Box})
\end{align*}
\]

Term and Expression Typing. Figure \ref{fig:term-type} shows the inference rules for typing terms and expressions. The type rules are standard except for T-NEW, T-BOX, and T-OPEN. Under effect ocap T-NEW requires the instantiated class to be ocap. An expression \( \text{box}[C] \) has type \( \Box[C] \) provided \( \text{ocap}(C) \)
Theorem 2 (Well-typed Heap). A heap $H$ is well-typed, written $\vdash H : \tau$ iff

$$
\forall o \in \text{dom}(H). H(o) = (C, FM) \implies
(\text{dom}(FM) = \text{fields}(C) \land
\forall f \in \text{dom}(FM). FM(f) = \text{null} \lor
\text{typeof}(H, FM(f)) <: \text{ftype}(C, f))
$$

To formalize the heap structure enforced by CLC$^1$ we use the following definitions.

Definition 3 (Separation). Two object identifiers $o$ and $o'$ are separate in heap $H$, written $sep(H, o, o')$, iff

$$
\forall q, q' \in \text{dom}(H). \text{reach}(H, o, q) \land \text{reach}(H, o', q') \implies q \neq q'.
$$

Definition 4 (Box Separation). For heap $H$ and frame $F$, boxSep($H$, $F$) holds iff

$$
F = \langle L, t \rangle^l \land \forall x \mapsto b(o), y \mapsto b(o') \in L.
$$

Definition 5 (Box-Object Separation). For heap $H$ and frame $F$, boxObjSep($H$, $F$) holds iff

$$
F = \langle L, t \rangle^l \land \forall x \mapsto b(o), y \mapsto b(o') \in L. \text{sep}(H, o, o')
$$

Definition 6 (Box Ocap Invariant). For heap $H$ and frame $F$, boxOcap($H$, $F$) holds iff

$$
F = \langle L, t \rangle^l \land \forall x \mapsto b(o), y \mapsto b(o') \in L. \text{sep}(H, o, o')
$$

In a well-formed frame, (a) two box references that are not aliases are disjoint (Def. 4), (b) box references and non-box

\[ L(x) = \text{null} \lor \text{typeof}(H, L(x)) <: \Gamma(x) \]

\[ H \vdash \Gamma; L, x \]

\[ dom(\Gamma) \subseteq dom(L) \]

\[ \forall x \in dom(\Gamma). H \vdash \Gamma; L, x \]

\[ H \vdash \epsilon \] (T-EmpFS)

\[ H \vdash \langle L, t \rangle^l : \sigma \] (T-Frame2)

\[ H \vdash F^\tau : \tau \] (T-FS-NA1)

\[ H \vdash F^\tau F^\epsilon : \sigma \] (T-FS-NA2)

\[ H \vdash F^\tau F^\epsilon : \sigma \] (T-FS-Sign)

\[ H \vdash \langle L, t \rangle^l : \sigma \] (T-Frame1)

\[ \Gamma, x : \tau; a \vdash t : \sigma \]

\[ H \vdash \Gamma; L \]

\[ \Gamma \vdash x : C \]

\[ ftype(C, f) = D \]

\[ \Gamma \vdash x.f : D \]

\[ a = \text{ocap} \implies \text{ocap}(C) \]

\[ \Gamma \vdash \box{C} : \text{Box}[C] \]

\[ \Gamma \vdash x : \text{Box}[C] \]

\[ y : C; \text{ocap} \vdash t : \sigma \] (T-OPEN)

Figure 10. CLC$^1$ term and expression typing.
boxSep(H, F) \quad boxObjSep(H, F)
\begin{align*}
a = \text{ocap} & \implies \text{globalOcapSep}(H, F) \\
H; a & \vdash F \text{ ok} \\
H; b & \vdash F' \text{ ok} \\
H; a & \vdash F' \text{ o ok}
\end{align*}
(F-OK)
\begin{align*}
boxRoot(o, F) & \vdash \text{reach}(H, o, o') \\
boxRoot(o, F) & \vdash \text{ocap}(H, o, o') \\
boxRoot(o, F) & \vdash \text{boxRoot}(o, F)
\end{align*}
(FS-OK)
\begin{align*}
H; b & \vdash F' \text{ o o ok} \\
H; b; l & \vdash o \in \text{env}(F)
\end{align*}

Figure 13. Auxiliary predicates.

Definition 10 (Open Box Propagation). For heap H, frame F, and frame stack FS, \(\text{openBoxPropagation}(H, F, FS)\) holds iff
\[
x \mapsto b(o) \in L \\
boxRoot(o, \langle L, t \rangle) \\
boxRoot(o, F) \\
boxRoot(o, F, \tilde{c})
\]
Informally, \(\text{openBoxPropagation}(H, F, FS)\) holds iff frame \(F'\) preserves the open boxes in the top-most frame of frame stack \(FS\).

The judgement \(H; a \vdash F \text{ ok}\) combines these invariants as shown in Figure 12; the corresponding judgement for frame stacks uses the following additional invariants.

Definition 8 (Box Separation). For heap H, frame F, and frame stack FS, \(\text{boxSeparation}(H, F, FS)\) holds iff
\[
\forall o, o' \in \text{dom}(H), \quad \text{boxRoot}(o, F) \land \text{boxRoot}(o', F) \land o \neq o' \implies \text{sep}(H, o, o')
\]

Definition 9 (Unique Open Box). For heap H, frame F, and frame stack FS, \(\text{uniqueOpenBox}(H, F, FS)\) holds iff
\[
\forall o, o' \in \text{dom}(H), \quad \text{openBox}(H, o, F, FS) \land \text{openBox}(H, o', F, FS) \implies o = o'
\]

3.2 Soundness and Heap Separation

Type soundness of CLC\(^1\) follows from the following preservation and progress theorems. Instead of proving these theorems directly, we prove corresponding theorems for an extended core language (Section 4).

Theorem 1 (Preservation). If \(H; *\) then:
1. If \(H \vdash F : \sigma, H; a \vdash F \text{ ok} \) and \(H, F \rightarrow H', F'\) then \(H' : * \rightarrow H' : * \vdash F' \text{ ok} \).
2. If \(H : F, H; a \vdash F \text{ ok} \) and \(H, F \rightarrow H', F'\) then \(H' : \vdash H' : * \rightarrow \text{ok} \).

Theorem 2 (Progress). If \(H; *\) then:
If \(H \vdash F\) and \(H; a \vdash F \text{ ok}\) then either \(H, F \rightarrow H', F'\) or \(F = \langle L, t \rangle\) \(\text{or}\) \(t \in \{y.f, \tilde{c} = z, y.m\}, y.open\{z \rightarrow t'\}\) and \(L(y) = \text{null}\).

The following corollary expresses an essential heap separation invariant enforced by CLC\(^1\). Informally, the corollary states that objects “within a box” (reachable from a box reference) are never mutated unless their box is “open” (a reference to the box entry object is on the stack).

Corollary 1 (Heap Separation). If \(H; *\) then:
If \(H \vdash F\), \(H; a \vdash F \text{ ok}\), \(H, F \rightarrow H', F'\), \(F = \langle L, t \rangle\), and \(L(y) = \text{null}\).
\[ p ::= cd \text{ } vd \text{ } t \]  

\[ cd ::= \text{class } C \text{ extends } D \{ fd \text{ } md \} \]  

\[ vd ::= \text{var } f : C \]  

\[ ud ::= \text{var } f : \text{Box}[C] \]  

\[ fd ::= \text{vd } \cup \text{ud} \]  

\[ md ::= \text{def } m(x : \sigma) : C = t \]  

\[ \sigma, \tau ::= C, D \]  

\[ \| \text{Box}[C] \| \]  

\[ \| \text{Null} \| \]  

\[ \pi ::= \sigma, \tau \]  

\[ Q \triangleright \text{Box}[C] \]  

\[ \bot \]

Figure 14. CLC² syntax. C, D range over class names, f, m, x range over term names. Q ranges over abstract types.

\[ L(y) = o', \text{boxRoot}(o, FS), \text{and reach}(H, o, o'), \text{then } w \mapsto o \in \text{env}(G) \text{ where } G \in FS. \]

**Proof sketch.** First, by \( H ; a \vdash FS \text{ ok}, FS-\text{ ok}, \text{and } FS = F \circ GS \text{ we have } H ; a \vdash F \text{ ok}. \) By F-ok we have \( \text{boxObjSep}(H, F). \) By def. [5] this means \( u \mapsto b(o) \notin L \) and therefore \( \neg\text{boxRoot}(o, F). \) Given that \( L(y) = o' \), \( \text{reach}(H, o, o') \), and \( \text{boxRoot}(o, FS) \) it must be that \( \text{boxRoot}(o, GS). \) Given that \( \text{boxRoot}(o, GS) \) by def. \( \text{boxRoot} \) (Figure 13), there is a frame \( G' \in GS \text{ such that } u \mapsto b(o) \in \text{env}(G'). \) Well-formedness of FS implies well-formedness of all its frames (FS-ok). Therefore, \( G' \) is well-formed and by F-ok, \( o \) is disjoint from other boxes (def. [4] and other objects (def. [5]) reachable in \( G' \), including the global variable. By the transition rules, box reference \( u \mapsto b(o) \) prevents field selection; as a result, between frames \( F \) and \( G' \) there must be a frame created by opening \( b(o). \) By E-OPEN, this means there is a frame \( G \in FS \text{ such that } w \mapsto o \in \text{env}(G). \)

3.3 Lightweight Affinity

This section introduces the CORELACASA² language (CLC²) which extends CLC¹ with affinity, such that boxes may be consumed at most once. Access to boxes is controlled using permissions. Permissions themselves are neither flow-sensitive nor affine. Consequently, they can be maintained in the type environment \( \Gamma. \) Our notion of affinity is based on continuation terms: consumption of permissions, and, thus, boxes, is only possible in contexts where an explicit continuation is provided. The consumed permission is then no longer available in the continuation.

**Syntax.** Figure 14 and Figure 15 show the syntactic differences between CLC² and CLC¹: first, field types are either class types \( C \) or box types \( \text{Box}[C] \); second, we introduce a bottom type \( \bot \) and guarded types \( Q \triangleright \text{Box}[C] \) where \( Q \) ranges over a countably infinite supply of abstract types; third, we introduce continuation terms.

In CLC² types are divided into surface types which can occur in the surface syntax, and general types, including guarded types, which cannot occur in the surface syntax; guarded types are only introduced by type inference (see Section 3.3.2). The bottom type \( \bot \) is the type of continuation terms \( t' \); these terms come in three forms:

1. A term \( \text{box}[C] \{ x \mapsto t \} \) creates a box containing a new instance of class type \( C \), and makes that box accessible as \( x \) in the continuation \( t. \) Note that in CLC¹ boxes are created using expressions of the form \( \text{box}[C] \). In CLC² we require the continuation term \( t \), because creating a box in addition creates a permission only available in \( t. \)

2. A term \( \text{capture}(x, f, y) \{ z \mapsto t \} \) merges two boxes \( x \) and \( y \) by assigning the value of \( y \) to the field \( f \) of the value of \( x. \) In the continuation \( t \) \( y \)'s permission is no longer available, and \( b \) \( z \) refers to box \( x. \)

3. A term \( \text{swap}(x, f, y) \{ z \mapsto t \} \) extracts the value of the unique field \( x.f \) and makes it available as the value of a box \( z \) in the continuation \( t; \) in addition, the value of box \( y \) replaces the previous value of \( x.f. \) Finally, \( y \)'s permission is consumed.

Given that only continuation terms can create boxes in CLC², method invocations cannot return boxes unknown to the caller. As a result, any box returned by a method invocation must have been passed as the single argument in the invocation. However, a method that takes a box as an argument, and returns the same box can be expressed using a combination of open and a method that takes the contents of the box as an argument. Therefore, method return types are always class types in CLC², simplifying the meta-theory.

3.3.1 Dynamic Semantics

CLC² extends the dynamic semantics compared to CLC¹ with dynamically changing permissions. A dynamic access to a box requires its associated permission to be available. For this, we extend the reduction relations compared to CLC¹ with permission sets \( P \). Thus, a frame \( \langle L, t, P \rangle \) combines a variable environment \( L \) and term \( t \) with a set of permissions \( P. \) (As before, the label \( l \) is used for transferring return values from method invocations.)

The transition rules of CLC² for single frames are identical to the corresponding transition rules of CLC¹; the permission sets do not change. In contrast, the transition rules for frame stacks affect the permission sets of frames.

The extended transition rules of CLC² are shown in Figure 16. Rule E-INVOKΕ additionally requires permission \( p \) to be available in \( P \) in case the argument of the invocation is a box protected by \( p; \) in this case permission \( p \) is also transferred to the new frame (the “activation record”).

\[ Therefore, \text{the transition rules (trivially) extended with permission sets are only shown in Appendix A.} \]
t ::=  
| x let x = e in t  terms  
| t'  continuation term

e ::=  
| null expressions  
| x variable  
| x.f assignment  
| new C instance creation  
| x.open {y ↦ t} open box

t' ::=  
| box[C] {x ↦ t} continuation term  
| capture(x.f, y) {z ↦ t} capture  
| swap(x.f, y) {z ↦ t} swap

Figure 15. CLC² terms and expressions.

$H(L(y)) = \langle C, FM \rangle$  mbody(C, m) = x → t'  
$L' = L_0[\text{this} \mapsto L(y), x \mapsto L(z)]  
P' = \emptyset \lor (L(z) = b(o, p) \land p \in P \land P' = \{p\})$

$H, (L, \text{let } x = y, m(z) \text{ in } t, P)' \circ FS  
\quad \rightarrow H, (L', t', P') \circ FS$  

(E-INVOC)

$H, (L, x, P)' \circ (L', t', P') \circ FS  
\quad \rightarrow H, (L'[y \mapsto L(x)], t', P') \circ FS$  

(E-RETURN1)

$H, (L, x, P)' \circ (L', t', P') \circ FS  
\quad \rightarrow H, (L', t', P') \circ FS$  

(E-RETURN2)

$L(y) = b(o, p)  \quad p \in P  \quad L' = [z \mapsto o]$  

$H, (L, \text{let } x = y.open \{z \mapsto t'\} \text{ in } t, P)' \circ FS  
\quad \rightarrow H, (L', t', \emptyset) \circ (L[x \mapsto L(y)], t, P') \circ FS$  

(E-OPEN)

$\exists o \notin \text{dom}(H)  \quad \text{fields}(C) = \emptyset  
\quad H' = H[o \mapsto \langle C, \emptyset \mapsto \emptyset \rangle]  
\quad p \text{ fresh}$  

$H, (L, \text{box}[C] \{x \mapsto t\}, P)' \circ FS  
\quad \rightarrow H', (L[x \mapsto b(o, p)], t, P \cup \{p\}) \circ \epsilon$  

(E-BOX)

Figure 16. CLC² frame stack transition rules.

duction gets stuck if permission p is not available. Rules E-RETURN1 and E-RETURN2 do not affect permission sets and are otherwise identical to the corresponding rules of CLC¹. Rule E-OPEN requires that permission p of the box-to-open b(o, p) is one of the currently available permissions P. The permission set of the new frame is empty. Rule E-Box creates a box b(o, p) accessible in continuation t using fresh permission p. Note that rule E-Box discards frame stack FS in favor of the continuation t.

$\begin{align*}
L(x) &= b(o, p) & L(y) &= b(o', p') & \{p, p'\} \subseteq P \\
H(o) &= \langle C, FM \rangle & H' &= H[o \mapsto \langle C, FM[f \mapsto o']\rangle] \\
H, (L, \text{capture}(x.f, y) \{z \mapsto t\}, P)' \circ FS  
\quad \rightarrow H', (L[z \mapsto L(x)], t, P \setminus \{p\}') \circ \epsilon
\end{align*}$  

(E-CAPTURE)

$\begin{align*}
L(x) &= b(o, p) & L(y) &= b(o', p') & \{p, p'\} \subseteq P \\
H(o) &= \langle C, FM \rangle & H' &= H[o \mapsto \langle C, FM[f \mapsto o']\rangle]  \\
H, (L, \text{swap}(x.f, y) \{z \mapsto t\}, P)' \circ FS  
\quad \rightarrow H', (L[z \mapsto b(o'', p')], t, (P \setminus \{p'\}) \cup \{p''\}) \circ \epsilon
\end{align*}$  

(E-SWAP)

Figure 17. Transition rules for capture and swap.

$\Gamma_0, \text{this} : C, x : D ; \epsilon \vdash t : E'  
\quad E' < : E$  

(C : \text{def } m(x : D) : E = t)  

(WF-METHOD1)

$\Gamma = \Gamma_0, \text{this} : C, x : Q \Box D, \text{Perm}[Q]  
\quad Q \text{ fresh}$  

$\Gamma, \emptyset \vdash t : E'  
\quad E' < : E$  

(C : \text{def } m(x : D)[E] : E = t)  

(WF-METHOD2)

Figure 18. Well-formed CLC² methods.

Figure [17] shows CLC²’s two new transition rules. E-CAPTURE merges box b(o, p) and box b(o', p') by assigning o' to field f of object H(o). The semantics of capture is thus similar to that of a regular field assignment. However, capture additionally requires both permissions p and p' to be available; moreover, in continuation t permission p' is no longer available, effectively consuming box b(o', p'). Like E-Box, E-CAPTURE discards frame stack FS. Finally, E-SWAP provides access to a unique field f of an object in box b(o, p): in continuation t variable z refers to the previous object o'' in f; the object o' in box b(o', p') replaces o''. Like E-CAPTURE, E-SWAP requires both permissions p and p' to be available, and in continuation t permission p' is no longer available, consuming box b(o', p').

3.3.2 Static Semantics  
Well-Formed Programs. CLC² adapts method well-formedness for the case where static permissions are propagated to the callee context: the body of a method with a parameter of type Box[D] is type-checked in an environment $\Gamma$ which includes a static permission Perm[Q] where Q is a fresh abstract type; furthermore, the parameter has a guarded type Q > Box[D]. This environment $\Gamma$ ensures the method body has full access to the argument box.

The OCAP* rules for CLC² treat box-typed method parameters analogously, and are left to the appendix.

Subclassing and Subtypes. In CLC², the subtyping relation $< :$ is identical to that of CLC¹ except for one additional rule for the $\bot$ type:
Perm environment used to type-check the continuation in and provides access to it under alias unique field $x$ provides access to box is consumed in each case. In its continuation, expressions. Both rules require $t$ continuation term $x$ opened. Finally, T-B box expressions. (Box-typed fields are then accessible using.

Figure 20 shows the type rules for CLC^2’s capture and swap.

Well-Formedness. CLC^2 extends CLC^1 with unique fields of type Box[C] (see Figure[14]; the following refined definition of well-typed heaps in CLC^2 reflects this extension:

Definition 11 (Well-typed Heap). A heap $H$ is well-typed, written $\vdash H : \star \text{iff}$

$$\forall o \in \text{dom}(H). \; H(o) = (C, FM) \implies \begin{cases} \text{dom}(FM) = \text{fields}(C) \land \forall f \in \text{dom}(FM). \; FM(f) = \text{null} \lor \\
\text{typeof}(H, FM(f)) = \bot \land \\
ftype(C, f) \in \{D, \text{Box}[D]\} \end{cases}$$

The most interesting additions of CLC^2 with respect to well-formedness concern (a) separation invariants and (b) field uniqueness. In CLC^1, two boxes $x$ and $y$ are separate as long as $x$ is not an alias of $y$. In CLC^2, the separation invariant is more complex, because capture merges two boxes, and swap replaces the value of a unique field. The key idea is to make separation conditional on the availability of permissions.

Box separation for frames in CLC^2 is defined as follows:

Definition 12 (Box Separation). For heap $H$ and frame $F$, $\text{boxSep}(H, F)$ holds iff

$$F = \{L, t, P\} \land \forall x \rightarrow b(o, p), y \rightarrow b(o', p') \in L. \; p \neq p’ \land \{p, p’\} \subseteq P \implies \text{sep}(H, o, o’).$$

Two box references are disjoint if they are guarded by two different permissions which are both available. As soon as a box is consumed, e.g., via capture, box separation no longer holds, as expected. In other invariants like $\text{boxObjSep}$, box permissions are not required. Similarly, the differences in $\text{boxOcap}$ and $\text{globalOcapSep}$ are minor, and therefore left to the appendix.

CLC^2’s addition of unique fields requires a new field uniqueness invariant for well-formed frames:
Definition 13 (Field Uniqueness). For heap $H$ and frame $F = \langle L, t, P \rangle^I$, fieldUniqueness $(H, F)$ holds iff
\[ \forall x \Rightarrow b(o, p) \in L, o', o \in \text{dom}(H), p \in P \wedge \text{reach}(H, o, \hat{o}) \wedge H(\hat{o}) = \langle C, FM \rangle \wedge ftype(C, f) = \text{Box}[D] \wedge \text{reach}(H, FM(f), o') \implies \text{domedge}(H, \hat{o}, f, o, o') \]
This invariant expresses the fact that all reference paths from a box $b(o, p)$ to an object $o'$ reachable from a unique field $f$ of object $\hat{o}$ must “go through” that unique field. In other words, in all reference paths from $o$ to $o'$, the edge $(\hat{o}, f)$ is a dominating edge. (A precise definition of domedge appears in the appendix.)

Frame Stack Invariants. The frame stack invariants of CLC$^3$ are extended to take the availability of permissions into account. For example, box separation is now only preserved for boxes (a) that are not controlled by the same permission, and (b) whose permissions are available:

Definition 14 (Box Separation). Frame $F$ and frame stack $FS$ satisfy the box separation property in $H$, written $\text{boxSep}(H, F, FS)$ iff
\[ \forall o, o' \in \text{dom}(H). \text{boxRoot}(o, F, p) \wedge \text{boxRoot}(o', FS, F', p') \wedge p \neq p' \implies \text{sep}(H, o, o') \]
Note that the availability of permissions is required indirectly by the boxRoot predicate (its other details are uninteresting, and therefore omitted).

4. Soundness

Theorem 3 (Preservation). If $H \vdash H$ : $* \then:
1. If $H \vdash F : \sigma, H ; a \vdash F \ ok$, and $H, F \to H', F'$ then $H' : *, H' \vdash F' : \sigma, H' ; a \vdash F' \ ok$.
2. If $H \vdash FS, H ; a \vdash FS \ ok$, and $H, FS \to H', FS'$ then $H' : *, H' \vdash FS', H' ; b \vdash FS' \ ok$.

Proof. Part (1) is proved by induction on the derivation of $H, F \to H', F'$. Part (2) is proved by induction on the derivation of $H, FS \to H', FS'$ and part (1). (See the companion technical report [37] for the full proof.)

Theorem 4 (Progress). If $H \vdash FS$ and $H ; a \vdash FS \ ok$ then either $H, FS \to H', FS'$ or $FS = \langle L, x, P \rangle^I \circ \epsilon$ or $FS = F \circ GS$ where

- $F = \langle L, \text{let } x = t \tin t', P \rangle^I, t \in \{y.f, y.f = z, y.m(z), y.open \{z \Rightarrow t''\}\}$, and $L(x) = \text{null}$; or
- $F = \langle L, \text{capture}(x, y, \{z \Rightarrow t\}), P \rangle^I$ where $L(x) = \text{null} \land L(y) = \text{null}$; or
- $F = \langle L, \text{swap}(x, f, y, \{z \Rightarrow t\}), P \rangle^I$ where $L(x) = \text{null} \land L(y) = \text{null}$.

Proof. By induction on the derivation of $H \vdash FS$. (See the companion technical report [37] for the full proof.)

Importantly, for a well-formed frame configuration, CLC$^3$ ensures that all required permissions are dynamically available; thus, reduction is never stuck due to missing permissions.

4.1 Isolation

In order to state an essential isolation theorem, in the following we extend CLC$^2$ with a simple form of message-passing concurrency. We call the resulting language CLC$^3$. CLC$^3$ enables the statement of Theorem 5 which expresses the fact that the type system of CLC$^3$ enforces process isolation in the presence of a shared heap and efficient, by-reference message passing.

Figure 21 summarizes the syntax extensions. CLC$^3$ adds a generic type $\text{Proc}[C]$ for processes capable of receiving messages of type $\text{Box}[C]$. An expression of the form $\text{proc } \{ (x : \text{Box}[C]) \Rightarrow t \}$ creates a concurrent process which applies the function $\{ (x : \text{Box}[C]) \Rightarrow t \}$ to each received message. A continuation term of the form $\text{send}(x, y) \{ z \Rightarrow t \}$ asynchronously sends box $y$ to process $x$ and then applies the continuation closure $\{ z \Rightarrow t \} \to x$.

Dynamic Semantics. CLC$^3$ extends the dynamic semantics of CLC$^2$ such that the configuration of a program consists of a shared heap $H$ and a set of processes $P$. Each process $FS^o$ is a frame stack $FS$ labelled with an object identifier $o$. A heap $H$ maps the object identifier $o$ of a process $FS^o$ to a process record $\langle \text{Box}[C], M, x \to t \rangle$ where $\text{Box}[C]$ is the type of messages the process can receive, $M$ is a set of object identifiers representing (buffered) incoming messages, and $x \to t$ is the message handler function. CLC$^3$ introduces a third reduction relation $H, P \to H', P'$ which reduces a set of processes $P$ in heap $H$.

Figure 22 shows the process transition rules. Rule E-PROC creates a new process by allocating a process record with an empty received message set and the message type and handler function as specified in the proc expression. The new process $\epsilon$ starts out with an empty frame stack, since it is initially idle. Rule E-SEND sends the object identifier in box $y$ to process $x$. The required permission $p'$ of box $y$ is consumed in the resulting frame $F'$. The call stack

Figure 21. Syntax extensions for concurrency.
Figure 23. CLC³ well-formedness rules.

\[
F = \langle L, \text{let } x = \text{proc } \{ y : \Box[C] \Rightarrow t' \} \in t, P \rangle^l \\
F' = \langle L[x \mapsto o', t], P \rangle^l \ \text{o' fresh} \\
H' = H'[o' \mapsto \Box[C], \emptyset, y \mapsto t']
\]

\[
H, \{(F \circ FS)^o \cup P \} \rightarrow H', \{(F' \circ FS)^o, \epsilon^o \} \cup P
\]

(E-PROC)

\[
F = \langle L, \text{send}(x, y) \{ z \mapsto t \}, P \rangle^l \ \text{L(x) = o} \\
H(o) = \Box[C], M, f \ \text{L(y) = b(o', p') \ p' \in P} \\
F' = \langle L[z \mapsto o], t, P \setminus \{ p' \} \rangle^r \\
H' = H[o \mapsto \Box[C], M \cup \{ o' \}, f]
\]

\[
H, F \circ FS \rightarrow H', \{ F' \circ \epsilon^o \} \cup P
\]

(E-RECEIVE)

Figure 22. CLC³ process transition rules.

\[
H \vdash FS^o \ H \vdash \mathcal{P} \\
\vdash \{ FS^o \} \cup \mathcal{P}
\]

(WF-SOUP)

\[
H \vdash FS \ H : a \vdash FS \ \text{ok}
\]

(WF-PROC)

\[
\vdash \{ FS^o \}
\]

Figure 23. CLC³ well-formedness rules.

is discarded, since send is a continuation term. In rule E-RECEIVE process o is ready to process a message from its non-empty set of incoming messages M, since (the term in) frame F cannot be reduced further and there are no other frames on the frame stack. (The \( \cup \) operator denotes disjoint set union.) Frame F' starts message processing with the parameter bound to a box reference with a fresh permission.

Static Semantics. Figure 23 shows the well-formedness rules that CLC³ introduces for (sets of) processes. A set of processes is well-formed if each process is well-formed (WF-SOUP). A process is well-formed if its frame stack is well-formed (WF-PROC). CLC³ also extends the object capability rules: \( \text{ocap}(C) \Rightarrow \text{ocap}(\text{Proc}[C]) \).

Figure 24 shows the typing of process creation and message sending. Rule T-PROC requires the body of a new process to be well-typed in an environment that only contains the parameter of the message handler and a matching access permission which is fresh. Importantly, body term t is type-checked under effect ocap. This means that t may only instantiate ocap classes. As a result, it is impossible to access global variables from within the newly created process. Rule T-SEND requires the permission \( \text{Perm}[Q] \) of sent box y to be available in context \( \Gamma \). The body t of the continuation closure must be well-typed in a context where \( \text{Perm}[Q] \) is no longer available. As with all continuation terms, the type of a send term is \( \perp \).

\[
x : Q \triangleright \Box[C], \text{Perm}[Q] ; \text{ocap} \vdash t : \pi
\]

(T-PROC)

\[
\Gamma ; a \vdash \text{proc } \{ x : \Box[C] \Rightarrow t \} : \text{Proc}[C]
\]

\[
\Gamma ; a \vdash x : \text{Proc}[C] \quad \Gamma ; a \vdash y : Q \triangleright \Box[D] \\
\text{Perm}[Q] \in \Gamma \quad D < : C
\]

\[
\Gamma \{ \text{Perm}[Q] \}, z : \text{Proc}[C] ; a \vdash t : \pi
\]

(T-SEND)

Figure 24. CLC³ typing rules.

\[
\vdash x \mapsto o \in L \lor (x \mapsto b(o, p) \in L \land p \in P)
\]

\[
\text{accRoot}(o, (L, t, P)^l)
\]

(ACC-F)

\[
\text{accRoot}(o, F) \lor \text{accRoot}(o, FS)
\]

(ACC-FS)

\[
\forall o, o' \in \text{dom}(H), (\text{accRoot}(o, FS)) \land \text{accRoot}(o', FS') \Rightarrow \text{sep}(H, o, o')
\]

(ISO-FS)

\[
\text{isolated}(H, FS, FS')
\]

(ISO-PROC)

\[
\forall q \in M, q' \in M', \text{sep}(H, q, q')
\]

\[
\text{isolated}(H, FS, GS)
\]

\[
\text{isolated}(H, FS^o, GS^o)
\]

(ISO-PROC)

Figure 25. CLC³ process and frame stack isolation.

Figure 25 defines a predicate \( \text{isolated} \) to express isolation of frame stacks and processes. Isolation of frame stacks builds on an \( \text{accRoot} \) predicate: identifier o is an accessible root in frame F, written \( \text{accRoot}(o, F) \), iff \( \text{env}(F) \) contains a binding \( x \mapsto o \) or \( x \mapsto b(o, p) \) where permission p is available in F (ACC-F); o is an accessible root in frame stack FS iff \( \text{accRoot}(o, F) \) holds for any frame \( F \in FS \) (ACC-FS). Two frame stacks are then isolated in H iff all their accessible roots are disjoint in H (ISO-FS). Finally, two processes are isolated iff their message queues and frame stacks are disjoint (ISO-PROC).

Theorem 5 (Isolation). If \( \vdash H : * \) then:

If \( H \vdash P, \forall P', P' \in P, P \neq P' \Rightarrow \text{isolated}(H, P, P') \), and \( H, P \rightarrow H', P' \) then \( H' : *, H' \vdash P', \) and \( \forall Q, Q' \in P', Q \neq Q' \Rightarrow \text{isolated}(H', Q, Q') \).

Theorem 5 states that \( \rightarrow \) preserves isolation of well-typed processes. Isolation is preserved even when boxes are transferred by reference between concurrent processes. Informally, the validity of this statement rests on the preservation of well-formedness of frames, frame stacks, and processes; well-formedness guarantees the separation of boxes with available permissions (def. [12] and def. [14], and the separation of boxes with available permissions from objects “outside” of boxes (def. [15] and def. [17] in Appendix A).
Informally, Theorem 5 together with the soundness of CLC\(^2\) implies data-race freedom. According to the reduction rules of CLC\(^2\), an access to a box reference with permission \(p\) cannot be reduced if \(p\) is not available dynamically. By Theorem 4, a well-typed program never attempts such a reduction. However, Theorem 5 states that accessing boxes with available permissions preserves process isolation.

5. Implementation

LACASA is implemented as a combination of a compiler plugin for the current Scala 2 reference compiler and a runtime library. The plugin extends the compilation pipeline with an additional phase right after regular type checking. Its main tasks are (a) object-capability checking, (b) checking the stack locality of boxes and permissions, and (c) checking the constraints of LACASA expressions. In turn, (c) requires object-capability checking: type arguments of mkBox invocations must be object-capability safe, and open bodies may only instantiate object-capability safe classes. Certain important constraints are implemented using spores [45].

Object Capability Checking in Scala. Our empirical study revealed the importance of certain Scala-specific “tweaks” to conventional object-capability checking. We describe the most important one. The Scala compiler generates so-called “companion” singleton objects for case classes and custom value classes if the corresponding companions do not already exist. For a case class such a synthetic companion object provides, e.g., factory and extractor [30] methods. Synthetic companion objects are object-capability safe.

Leveraging Spores. We leverage constraints supported by spores in several places in LACASA. We provide two examples where spores are used in our implementation.

The first example is the body of an open expression. According to the rules of LACASA, it is not allowed to have free variables (see Section 2). Using spores, this constraint can be expressed in the type of the open method as follows:

```
def open(fun: Spore[T, Unit])(implicit
  acc: CanAccess (type C = self.C ),
  noCapture: OnlyNothing[fun.Captured]): Unit
```

Besides the implicit access permission, the method also takes an implicit parameter of type OnlyNothing[fun.Captured]. The generic OnlyNothing type is a trivial type class with only a single instance, namely Nothing, Scala’s bottom type. Consequently, for an invocation of open, the compiler is only able to resolve the implicit parameter noCapture in the case where type fun.Captured is equal to Nothing. In turn, spores ensure this is only the case when the fun spore does not capture anything.

The second example where LACASA leverages spores is permission consumption in swap:

```
def swap[S](select: T => Box[S])
  (assign: (T, Box[S]) => Unit, b: Box[S])
```

5.1 Discarding the Stack Using Exceptions

Certain LACASA operations require discarding the stack of callers in order to ensure consumed access permissions become unavailable. For example, recall the message send shown in Figure 2 (see Section 2):

```
s.next.send(packed.box)({
  // continuation closure
  3 }) (access)
  4 // unreachable
```

Here, the send invocation consumes the access permission: the permission is no longer available in the continuation. This semantics is enforced by ensuring (a) the access permission is unavailable within the explicit continuation closure (line 2), and (b) code following the send invocation (line 4) is unreachable. The former is enforced analogously to swap discussed above. The latter is enforced by discarding the stack of callers.

Discarding the call stack is a well-known technique in Scala, and has been widely used in the context of event-based actors [38] where the stack of callers is discarded when an actor suspends with just a continuation closure. The implementation consists of throwing an exception which unwinds the call stack up to the actor’s event-loop, or up to the boundary of a concurrent task.

Prior to throwing the stack-unwinding exception, operations like send invoke their continuation closure which is provided explicitly by the programmer:

```
def send(msg: Box[T])(cont: NullarySpore[Unit] {...})
  2 (implicit acc: CanAccess {...}): Nothing = {
    3 ... // enqueue message
    4 cont() // invoke continuation closure
    5 throw new NoReturnControl // discard stack
  6 }
```

The thrown NoReturnControl exception is caught either within the main thread where the main method is wrapped

---

8 See [https://github.com/twitter-archive/kestrel/blob/3e64b2a3ad4e71256213e2bdc6e8a997/kestrelHandler.scala](https://github.com/twitter-archive/kestrel/blob/3e64b2a3ad4e71256213e2bdc6e8a997/kestrelHandler.scala) for a usage example in a large-scale production system.
class MessageHandlerTask(
  receiver: Actor[T],
  packed: Packed[T]) extends Runnable {
  def run(): Unit = {
    // process message in 'packed' object
    try {
      // invoke 'receive' method of 'receiver' actor
      receiver.receive(packed.box)(packed.access)
    } catch {
      case nrc: NoReturnControl => /* do nothing */
    }
    // check for next message
    ... ...
  }
}

Figure 26. Handling NoReturnControl within actors.

By catching and not propagating the ControlThrowable, the access permission remains accessible from line 8, enabling sending the same object (packed.box) twice.

In order to prevent such soundness issues, the LACASA compiler plugin performs additional checks: a try-catch expression is valid if either (a) none of its catch clauses match ControlThrowables, or (b) all catch clauses matching ControlThrowables propagate caught exceptions. Furthermore, to support trusted LACASA code, a marker method permits unsafe catches.

6. Empirical Evaluation

The presented approach to object isolation and uniqueness is based on object capabilities. Isolation is enforced only for instances of ocap classes, i.e., classes adhering to the object-capability discipline. Likewise, ownership transfer is supported only for instances of ocap classes. Therefore, it is important to know whether the object-capability discipline imposes an undue burden on developers; or whether, on the contrary, developers tend to design classes and traits in a way that naturally follows the object-capability discipline.

Specifically, our empirical evaluation aims to answer the following question: How many classes/trait in medium to large open-source Scala projects already satisfy the object-capability constraints required by LACASA?

Methodology. For our empirical analysis we selected Scala’s standard library, a large and widely-used class library, as well as two medium to large open-source Scala applications. In total, our corpus comprises 78,617 source lines of code (obtained using [28]). Determining the prevalence of ocap classes and traits is especially important in the case of Scala’s standard library, since it tells us for which classes/trait LACASA supports isolation and ownership transfer “out of the box,” i.e., without code changes. (We will refer to both classes and traits as “classes” in the following.)

The two open-source applications are Signal/Collect (S/C) and GeoTrellis. S/C [61] is a distributed graph processing framework with applications in machine learning and the semantic web, among others. Concurrency and distribution are implemented using the Akka actor framework [43]. Consequently, S/C could also benefit from LACASA’s additional safety. GeoTrellis is a high performance data processing engine for geographic data, used by the City of Asheville [17] (NC, USA) and the U.S. Army, among others. Like S/C, GeoTrellis utilizes actor concurrency through Akka.

In each case we performed a clean build with the LACASA compiler plugin enabled. We configured the plugin to check ocap constraints for all compiled classes. In addition, we collected statistics on classes that directly violate ocap constraints through accesses to global singleton objects.

Results. Figure 27 shows the collected statistics.

For Scala’s standard library we found that 43% of all classes follow the object-capability discipline. While this
number might seem low, it is important to note that a strict form of ocap checking was used: accesses to top-level singleton objects were disallowed, even if these singletons were themselves immutable and object-capability safe. Thus, classes directly using helper singletons were marked as insecure. Interestingly, only 25% of the insecure classes directly access top-level singleton objects. This means, the majority of insecure classes is insecure due to dependencies on other insecure classes. These results can be explained as follows. First, helper singletons (in particular, “companion objects”) play an important role in the architecture of Scala’s collections package. In turn, with 22,958 SLOC the collections package is by far the library’s largest package, accounting for 69% of its total size. Second, due to the high degree of reuse enabled by techniques such as the type class pattern, even a relatively small number of classes that directly depend on singletons leads to an overall 57% of insecure classes.


directly using helper singletons were marked as insecure. Interestingly, only 25% of the insecure classes directly access top-level singleton objects. This means, the majority of insecure classes is insecure due to dependencies on other insecure classes. These results can be explained as follows. First, helper singletons (in particular, “companion objects”) play an important role in the architecture of Scala’s collections package. In turn, with 22,958 SLOC the collections package is by far the library’s largest package, accounting for 69% of its total size. Second, due to the high degree of reuse enabled by techniques such as the type class pattern, even a relatively small number of classes that directly depend on singletons leads to an overall 57% of insecure classes.

In S/C 67% of all classes satisfy strict ocap constraints, a significantly higher percentage than for the Scala library. At the same time, the percentage of classes that are not ocap due to direct accesses to top-level singletons is also much higher (78% compared to 25%). This means there is less reuse of insecure classes in S/C. All analyzed components of GeoTrellis have a similarly high percentage of “directly insecure” classes. Interestingly, even with its reliance on “companion objects” and its high degree of reuse, the proportion of ocap classes in the standard library is significantly higher compared to GeoTrellis where it ranges between 21% and 35%.

### Immutability and Object Capabilities

Many singleton objects in Scala’s standard library (a) are deeply immutable, (b) only create instances of ocap classes, and (c) never access global state. Such singletons are safe to access from within ocap classes. To measure the impact of such singletons on the proportion of ocap classes, we reanalyzed S/C with knowledge of safe singletons in the standard library. As a result, the percentage of ocap classes increased from 67% to 79%, while the proportion of directly insecure classes remained identical. Thus, knowledge of “safe singleton objects” is indeed important for object capabilities in Scala.

### 7. Other Related Work

A number of previous approaches leverages permissions or capabilities for uniqueness or related notions. Approaches limited to tree-shaped object structures for unique references include 

- PACL [15, 16, 51, 57, 64, 65]. In contrast, LACASA provides external uniqueness [18], which allows internally-aliased object graphs. Permissions in LACASA indicate objects (“boxes”) are accessible in the current scope.

In contrast, the deny capabilities of the Pony language [22] indicate which operations are denied on aliases to the same object. By distinguishing read/write as well as (actor-)local and global aliases, Pony derives a fine-grained matrix of reference capabilities, which are more expressive than the presented system. While Pony is a new language design, LACASA integrates affine references into an existing language, while minimizing the effort for reusing existing classes.

The notion of uniqueness provided by our system is similar to UTT [49], an extension of Universe types [28] with ownership transfer. Overall, UTT is more flexible, whereas LACASA requires fewer annotations for reusing existing code; it also integrates with Scala’s local type inference. Active ownership [19] shares our goal of providing a minimal type system extension, however it requires owner-polymorphic methods and existential owners whose integration with local type inference is not clear. A more general overview of ownership-based aliasing control is provided in [20]. There is a long line of work on unique object references [4, 10, 14, 41] which are more restrictive than external uniqueness; a recurring theme is the interaction between unique, immutable, and read-only references, which is also exploited in a variant of C# for systems programming [55]. Several systems combine ownership with concurrency control to prevent data races. RaceFree Java [11] associates fields with locks, and an effect system ensures correct lock acquisition. Boyapati et al. [13] and Zhao [67] extend type system guarantees to deadlock prevention.

Our system takes important inspiration from Loci [66], a type system for enforcing thread locality which requires very

---

**Figure 27.** Evaluating the object-capability discipline in real Scala projects. Each project is an active open-source project hosted on GitHub. ★ represents the number of “stars” (or interest) a repository has on GitHub, and ✈ represents the number of contributors of the project (March 2016).
few source annotations. However, LACASA supports ownership transfer, which is outside the domain of Loci. Kilim \cite{60} combines type qualifiers with an intra-procedural shape analysis to ensure isolation of Java-based actors. To simplify the alias analysis and annotation system, messages must be tree-shaped. Messages in LACASA are not restricted to trees; moreover, LACASA uses a type-based approach rather than static analysis. StreamFlex \cite{59} and FlexoTasks \cite{9} are implicit ownership systems for stream-based programming; like LACASA, they allow reusing classes which pass certain sanity checks, but the systems are more restrictive than external uniqueness.

8. Conclusion

This paper presents a new approach to integrating isolation and uniqueness into an existing full-featured language. A key novelty of the system is its minimization of annotations necessary for reusing existing code. Only a single bit of information per class is required to determine its reusability. Interestingly, this information is provided by the object capability model, a proven methodology for applications in security, such as secure sandboxing. We present a complete formal account of our system, including proofs of key soundness theorems. We implement the system for the full Scala language, and evaluate the object capability model on a corpus of over 75,000 LOC of popular open-source projects. Our results show that between 21% and 79% of the classes of a project adhere to a strict object capability discipline. In summary, we believe our approach has the potential to make a flexible form of uniqueness practical on a large scale and in existing languages with rich type systems.

Acknowledgments

We would like to thank the anonymous OOPSLA referees for their thorough reviews which greatly helped us improve the quality of the paper. Sophia Drossopoulou, Sylvan Clebsch, Tim Wood, George Steed, Luke Cheeseman, and Rakhyila Mekhtieva provided valuable feedback on an earlier draft, suggesting examples we now include in the LACASA distribution. Last but not least, we are grateful to members of the research groups of Martin Odersky and Victor Kuncak for feedback on an earlier design of LACASA.

References


Figure 28. Well-formed ocap classes.

\[
x \mapsto b(o, p) \in L \quad p \in P
\]
\[
\text{boxRoot}(o, \langle L, t, P \rangle^t)
\]
\[
x \mapsto b(o, p) \in L \quad p \in P
\]
\[
\text{boxRoot}(o, \langle L, t, P \rangle^t, p)
\]
\[
\text{boxRoot}(F, p)
\]
\[
\text{boxRoot}(o, F \circ \epsilon, p)
\]
\[
\text{boxRoot}(o, F \circ \epsilon, p) \lor \text{boxRoot}(o, FS, P)
\]
\[
\text{boxRoot}(o, F \circ FS, P)
\]

\[
\gamma : \text{permTypes}(\Gamma) \rightarrow P \text{ injective}
\]
\[
\forall x \in \text{dom}(\Gamma),
\]
\[
\Gamma(x) = Q \triangleright \text{Box}[C] \land L(x) = b(o, p) \land \text{Perm}[Q] \in \Gamma
\]
\[
\gamma(Q) = p
\]
\[
\vdash \Gamma; L; P
\]

(WF-PERM)

\[
L(x) = \text{null} \lor
\]
\[
L(x) = o \land \text{typeof}(H, o) <; \Gamma(x) \lor
\]
\[
L(x) = \text{globalOcapSep}(H, F)
\]
\[
\text{boxOcap}(H, F)
\]
\[
\text{boxObjSep}(H, F)
\]
\[
\text{fieldUniqueness}(H, F)
\]
\[
H; a \vdash \text{F ok}
\]

(F-OK)

Figure 29. Frame and frame stack typing.

A. Additional Rules and Definitions

Figure 28 shows CLC²’s OCAP.* rules. Figure 29 shows the updated rules for frame and frame stack typing in CLC². Figure 30 shows the boxRoot predicate.

**Definition 15 (Box-Object Separation).** Frame F satisfies the box-object separation invariant in H, written boxObjSep(H, F), iff
\[
F = \langle L, t, P \rangle^t \land \forall x \mapsto b(o, p), y \mapsto o' \in L. \text{sep}(H, o, o')
\]

**Definition 16 (Box Ocap Invariant).** Frame F satisfies the box ocap invariant in H, written boxOcap(H, F), iff
\[
F = \langle L, t, P \rangle^t \land \forall x \mapsto b(o, p), y \mapsto o' \in L. \text{ocap}(H, o, o')
\]

**Definition 17 (Global Ocap Separation).** Frame F satisfies the global ocap separation invariant in H, written globalOcapSep(H, F), iff
\[
F = \langle L, t, P \rangle^t \land \forall x \mapsto o \in L, y \mapsto o' \in L_0. \text{ocap}(H, o, o') \land \text{sep}(H, o, o')
\]

**Definition 18 (Dominating Edge).** Field f of o is a dominating edge for paths from o to o’ in H, written domedge(H, o, f, o’), iff
\[
\forall P \in \text{path}(H, o, o’). \quad P = o \ldots \dot{f}. \quad \text{FM}(f) \ldots o'
\]
where H(\dot{f}) = \langle C, FM \rangle and f \in \text{dom}(FM).