LaCasa: Lightweight Affinity and Object Capabilities in Scala

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

Alex Loiko
Google, Sweden *
aleloi@google.com

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 [3], and it can 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 [33, 51, 64], region inference [62, 63], unique references [14, 18, 41, 48], and ownership types [21, 50].

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 [7] 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 (type) 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* [27, 46]. 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 [23, 47, 56].

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

* Work done while at KTH Royal Institute of Technology, Sweden.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2984042

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 3). 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 statement 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).[1] 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 implicits [24], an intrinsic part of type inference in Scala. In addition, foundations of implicits 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.

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

```
1  class ActorA extends Actor[Any] {
2    override def receive(msg: Any): Unit = msg match {
3      case s: Start =>
4        val newMsg = new Message
5        newMsg.arr = Array(1, 2, 3, 4)
6        s.next.send(newMsg)
7        newMsg.arr(2) = 33
8        // ...
9      case other => // ...
10   }
11 }
12 class ActorB extends Actor[Message] {
13   override def receive(msg: Message): Unit = {
14     println(msg.arr.mkString(","))
15   }
16 }
17 class Message {
18   var arr: Array[Int] = _
19   def leak(): Unit = {
20     SomeObject.fld = arr
21   }
22 }
23 object SomeObject {
24   var fld: Array[Int] = _
25 }
26 class Start {
27   var next: ActorRef[Message] = _
28 }
```

Figure 1. Two communicating actors in 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.[2] 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.

[2] In favor of clarity of explanation, Figure 1 shows hypothetical Scala code which requires slight changes for compilation with the Akka [43] 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 [8].[3] 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.[4]

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

```
{
  case pat_1 => e_1
    ...
  case pat_n => e_n
}
```

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

```
x => x match {
  case pat_1 => e_1
    ...
  case pat_n => e_n
}
```

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:

```
1   // box: Box[Message]
2   var a: Array[Int] = null
3   box.open({ msg =>
4     a = msg.arr
```

```
1  class ActorA extends Actor[Any] {
2    override def receive(box: Box[Any])
3        (implicit acc: CanAccess { type C = box.C }) {
4      box.open({
5        case s: Start =>
6          mkBox[Message] { packed =>
7            val access = packed.access
8            packed.box.open({ msg =>
9              msg.arr = Array(1, 2, 3, 4)
10           }) (access)
11           s.next.send(packed.box)({
12             // ...
13           }) (access)
14         }
15       case other => // ...
16     }) (acc)
17   }
18 }
19 class ActorB extends Actor[Message] {
20   override def receive(box: Box[Message])
21       (implicit acc: CanAccess { type C = box.C }) {
22     box.open({ msg =>
23       println(msg.arr.mkString(","))
24     }) (acc)
25   }
26 }
```

Figure 2. Two communicating actors in LaCasa.

[3] Scala's original actor implementation [38] 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.

```
5       SomeObject.fld = msg.arr
6     })(acc)
7     next.send(box)({
8       a(2) = 33
9     })(acc)
```

In this case, by capturing variable `a` in the body of `open`, and by making `a` an alias of the array in `msg`, it would be possible to access the array even after sending it (inside `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 `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 C = box.C }` where `box` is a parameter of type `Box[Any]`. Thus, the *type member* `C` of the permission's type is equal to the type member `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 `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 `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 `b` such that `b.C` is equal to the type member `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 `mkBox` expression. For example, the `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, `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: `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.[5] 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 => ... })(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

[5] See the Scala language specification [54] for details of implicit resolution.

```
1  class ActorA(next: ActorRef[C])
2      extends Actor[Container] {
3    def receive(msg: Box[Container])
4        (implicit acc: CanAccess { type C = msg.C }) {
5      mkBox[C] { packed =>
6        // ...
7        msg.swap(_.part1)(_.part1 = _, packed.box)(
8          spore { pack =>
9            val acc = pack.access
10
11           pack.box.open({ part1Obj =>
12             println(part1Obj.arr.mkString(","))
13             part1Obj.arr(0) = 1000
14           })(acc)
15
16           next.send(pack.box)({
17             // ...
18           })(acc)
19         }
20       )(acc)
21     }
22   }
23 }
24 class Container {
25   var part1: Box[C] = _
26   var part2: Box[C] = _
27 }
28 class C {
29   var arr: Array[Int] = _
30 }
```

Figure 3. An actor accessing a unique field via `swap`.

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* [18] of unique fields (see Section 3 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.

$p ::= \overline{cd}\ \overline{vd}\ t$	program
$cd ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{vd}\ \overline{md}\}$	class
$vd ::= \texttt{var}\ f : C$	variable
$md ::= \texttt{def}\ m(x : \sigma) : \tau = t$	method
$\sigma, \tau ::=$	type
$\quad C, D$	class type
$\quad \mid \texttt{Box}[C]$	box type
$\quad \mid \texttt{Null}$	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 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 $\texttt{Box}[C]$, 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* [34] (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-

$t ::=$		terms
	x	variable
	$\mid \texttt{let}\ x = e\ \texttt{in}\ t$	let binding
$e ::=$		expressions
	$\texttt{null}$	null reference
	$\mid x$	variable
	$\mid x.f$	selection
	$\mid x.f = y$	assignment
	$\mid \texttt{new}\ C$	instance creation
	$\mid x.m(y)$	invocation
	$\mid \texttt{box}[C]$	box creation
	$\mid x.\texttt{open}\ \{y \Rightarrow t\}$	open box

Figure 5. CLC[1] terms and expressions.

pression (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 $\{y \Rightarrow t\}$ 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, $H, F \longrightarrow H', F'$, and $H, FS \twoheadrightarrow H', FS'$. 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 $o \in dom(H)$ 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 $dom(H) \cup \{\texttt{null}\}$. FS is a sequence of stack frames F; we use the notation $FS = F \circ FS'$ to indicate that in stack FS frame F is the top-most frame which is followed by frame stack FS'.

A single frame $F = \langle L, t\rangle^l$ 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 $v \in dom(H) \cup \{\texttt{null}\} \cup \{b(o) \mid o \in dom(H)\}$. A value $b(o)$ is a *box reference* created using CLC[1]'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, $L' = L[x \mapsto v]$ denotes the updated mapping where $L'(y) = L(y)$ if $y \neq x$ and $L'(y) = v$ if $y = x$. A frame annotation l is either empty (or non-existant), expressed as $l = \epsilon$, 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 $mbody(C, f) = x \to t$ denotes the body of a method $\texttt{def}\ m(x : \sigma) : \tau = t$.

$$\frac{}{H, \langle L, \texttt{let}\ x = \texttt{null}\ \texttt{in}\ t\rangle^l \longrightarrow H, \langle L[x \mapsto \texttt{null}], t\rangle^l} \text{(E-NULL)}$$

$$\frac{}{H, \langle L, \texttt{let}\ x = y\ \texttt{in}\ t\rangle^l \longrightarrow H, \langle L[x \mapsto L(y)], t\rangle^l} \text{(E-VAR)}$$

$$\frac{H(L(y)) = \langle C, FM\rangle \qquad f \in dom(FM)}{H, \langle L, \texttt{let}\ x = y.f\ \texttt{in}\ t\rangle^l \longrightarrow H, \langle L[x \mapsto FM(f)], t\rangle^l} \text{(E-SELECT)}$$

$$\frac{L(y) = o \qquad H(o) = \langle C, FM\rangle \qquad H' = H[o \mapsto \langle C, FM[f \mapsto L(z)]\rangle]}{H, \langle L, \texttt{let}\ x = y.f = z\ \texttt{in}\ t\rangle^l \longrightarrow H', \langle L, \texttt{let}\ x = z\ \texttt{in}\ t\rangle^l} \text{(E-ASSIGN)}$$

$$\frac{o \notin dom(H) \qquad fields(C) = \overline{f} \qquad H' = H[o \mapsto \langle C, \overline{f \mapsto \texttt{null}}\rangle]}{H, \langle L, \texttt{let}\ x = \texttt{new}\ C\ \texttt{in}\ t\rangle^l \longrightarrow H', \langle L[x \mapsto o], t\rangle^l} \text{(E-NEW)}$$

$$\frac{o \notin dom(H) \qquad fields(C) = \overline{f} \qquad H' = H[o \mapsto \langle C, \overline{f \mapsto \texttt{null}}\rangle]}{H, \langle L, \texttt{let}\ x = \texttt{box}[C]\ \texttt{in}\ t\rangle^l \longrightarrow H', \langle L[x \mapsto b(o)], t\rangle^l} \text{(E-BOX)}$$

Figure 6. CLC[1] frame transition rules.

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

Single Frame Reduction. Figure 6 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 [11, 12, 55].

Frame Stack Reduction. Figure 7 shows the frame stack transition rules. Rule E-INVOKE 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

$$\frac{\begin{array}{c}H(L(y)) = \langle C, FM\rangle \\ mbody(C,m) = x \to t' \\ L' = L_0[\texttt{this} \mapsto L(y), x \mapsto L(z)]\end{array}}{\begin{array}{c}H, \langle L, \texttt{let}\ x = y.m(z)\ \texttt{in}\ t\rangle^l \circ FS \\ \twoheadrightarrow\ H, \langle L', t'\rangle^x \circ \langle L, t\rangle^l \circ FS\end{array}}\ \text{(E-INVOKE)}$$

$$\frac{H, \langle L, x\rangle^y \circ \langle L', t'\rangle^l \circ FS}{H, \langle L, x\rangle^y \circ \langle L', t'\rangle^l \circ FS \twoheadrightarrow H, \langle L'[y \mapsto L(x)], t'\rangle^l \circ FS}\ \text{(E-RETURN1)}$$

$$\frac{H, \langle L, x\rangle^\epsilon \circ \langle L', t'\rangle^l \circ FS}{H, \langle L, x\rangle^\epsilon \circ \langle L', t'\rangle^l \circ FS \twoheadrightarrow H, \langle L', t'\rangle^l \circ FS}\ \text{(E-RETURN2)}$$

$$\frac{L(y) = b(o) \qquad L' = [z \mapsto o]}{\begin{array}{c}H, \langle L, \texttt{let}\ x = y.\texttt{open}\ \{z \Rightarrow t'\}\ \texttt{in}\ t\rangle^l \circ FS \\ \twoheadrightarrow\ H, \langle L', t'\rangle^\epsilon \circ \langle L[x \mapsto L(y)], t\rangle^l \circ FS\end{array}}\ \text{(E-OPEN)}$$

Figure 7. CLC^1 frame stack transition rules.

$$\frac{p \vdash \overline{cd} \qquad p \vdash \Gamma_0 \qquad \Gamma_0\ ;\ \epsilon \vdash t : \sigma}{p \vdash \overline{cd}\ \overline{vd}\ t}\ \text{(WF-PROGRAM)}$$

$$\frac{\begin{array}{c}C \vdash \overline{md} \qquad D = \texttt{AnyRef} \vee p \vdash \texttt{class}\ D\ \ldots \\ \forall\ (\texttt{def}\ m\ \ldots) \in \overline{md}.\ override(m, C, D) \\ \forall\ \texttt{var}\ f : \sigma \in \overline{fd}.\ f \notin fields(D)\end{array}}{p \vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\}}\ \text{(WF-CLASS)}$$

$$\frac{mtype(m, D) \text{ not defined} \vee mtype(m, D) = mtype(m, C)}{override(m, C, D)}\ \text{(WF-OVERRIDE)}$$

$$\frac{\begin{array}{c}\Gamma_0, \texttt{this} : C, x : \sigma\ ;\ \epsilon \vdash t : \tau' \\ \tau' <: \tau\end{array}}{C \vdash \texttt{def}\ m(x : \sigma) : \tau = t}\ \text{(WF-METHOD)}$$

Figure 8. Well-formed CLC^1 programs.

E-RETURN2 enables returning from an ϵ-annotated frame. Rule E-OPEN creates such an ϵ-annotated frame. In the new frame, the object encapsulated by box y is accessible under alias z. In contrast to E-INVOKE, the new frame does not include the global environment L_0; instead, z is the only variable in the (domain of the) new environment.

3.1.2 Static Semantics

Type Assignment. A judgement of the form $\Gamma\ ;\ a \vdash t : \sigma$ assigns type σ to term t in type environment Γ under effect a. When assigning a type to the top-level term of a program the effect a is ϵ which is the unrestricted effect. In contrast, the body of an open expression must be well-typed under effect ocap which requires instantiated classes to be *ocap*.

Well-Formed Programs. Figure 8 shows the rules for well-formed programs. (We write ... to omit unimportant parts of a program.) A program is well-formed if all its class definitions are well-formed and its top-level term is well-typed in

$$\frac{}{ocap(\texttt{AnyRef})}\ \text{(OCAP-ANYREF)}$$

$$\frac{\begin{array}{c}p \vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \\ C \vdash_{ocap} \overline{md} \qquad ocap(D) \\ \forall\ \texttt{var}\ f : E \in \overline{fd}.\ ocap(E)\end{array}}{ocap(C)}\ \text{(OCAP-CLASS)}$$

$$\frac{\begin{array}{c}\texttt{this} : C, x : \sigma\ ;\ \texttt{ocap} \vdash t : \tau' \\ \tau' <: \tau\end{array}}{C \vdash_{ocap} \texttt{def}\ m(x : \sigma) : \tau = t}\ \text{(OCAP-METHOD)}$$

Figure 9. Object capability rules.

type environment $\Gamma_0 = \{\texttt{global} : C_g\}$ (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 $fields(D)$ [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 τ' under effect ϵ in environment $\Gamma_0, \texttt{this} : C, x : \sigma$, where σ is the type of m's parameter x, such that τ' is a subtype of m's declared result type τ (WF-METHOD).

Object Capabilities. For a class C to satisfy the constraints of the object-capability discipline, written $ocap(C)$, it must be well-formed according to the rules shown in Figure 9. Essentially, for a class C we have $ocap(C)$ if its superclass is ocap, the types of its fields are ocap, and its methods are well-formed according to $\vdash_{ocap}$. Rule OCAP-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 Γ_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.

Subclassing and Subtypes. In CLC^1, the subtyping relation <:, defined by the class table, is identical to that of FJ [42] except for two additional rules:

$$\frac{C <: D}{\texttt{Box}[C] <: \texttt{Box}[D]}\ \text{(<:-BOX)} \qquad \texttt{Null} <: \sigma\ \text{(<:-NULL)}$$

Term and Expression Typing. Figure 10 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 box$[C]$ has type Box$[C]$ provided $ocap(C)$

$$\Gamma\ ;\ a \vdash \texttt{null} : \texttt{Null}\ \text{(T-NULL)} \qquad \frac{x \in dom(\Gamma)}{\Gamma\ ;\ a \vdash x : \Gamma(x)}\ \text{(T-VAR)}$$

$$\frac{\Gamma\ ;\ a \vdash e : \tau \qquad \Gamma, x : \tau\ ;\ a \vdash t : \sigma}{\Gamma\ ;\ a \vdash \texttt{let}\ x = e\ \texttt{in}\ t : \sigma}\ \text{(T-LET)}$$

$$\frac{\Gamma\ ;\ a \vdash x : C \qquad ftype(C, f) = D}{\Gamma\ ;\ a \vdash x.f : D}\ \text{(T-SELECT)}$$

$$\frac{\Gamma\ ;\ a \vdash x : C \qquad ftype(C, f) = D \qquad \Gamma\ ;\ a \vdash y : D' \qquad D' <: D}{\Gamma\ ;\ a \vdash x.f = y : D}\ \text{(T-ASSIGN)}$$

$$\frac{a = \texttt{ocap} \implies ocap(C)}{\Gamma\ ;\ a \vdash \texttt{new}\ C : C}\ \text{(T-NEW)}$$

$$\frac{\Gamma\ ;\ a \vdash x : C \qquad mtype(C, m) = \sigma \to \tau \qquad \Gamma\ ;\ a \vdash y : \sigma' \qquad \sigma' <: \sigma}{\Gamma\ ;\ a \vdash x.m(y) : \tau}\ \text{(T-INVOKE)}$$

$$\frac{ocap(C)}{\Gamma\ ;\ a \vdash \texttt{box}[C] : \texttt{Box}[C]}\ \text{(T-BOX)}$$

$$\frac{\Gamma\ ;\ a \vdash x : \texttt{Box}[C] \qquad y : C\ ;\ \texttt{ocap} \vdash t : \sigma}{\Gamma\ ;\ a \vdash x.\texttt{open}\ \{y \Rightarrow t\} : \texttt{Box}[C]}\ \text{(T-OPEN)}$$

Figure 10. CLC[1] term and expression typing.

holds (T-BOX). Finally, T-OPEN requires the body t of open to be well-typed under effect ocap and in a type environment consisting only of $y : C$. The type of the open expression itself is $\texttt{Box}[C]$ (it simply returns box x).

Well-Formedness. Frames, frame stacks, and heaps must be well-formed. Figure 11 shows the well-formedness rules for environments, frames, and frame stacks. Essentially, Γ, L are well-formed in heap H if for all variables $x \in dom(\Gamma)$ the type of $L(x)$ in H is a subtype of the static type of x in Γ (WF-VAR, WF-ENV). A frame $\langle L, t\rangle^l$ is well-typed in H if its term t is well-typed in some environment Γ such that Γ, L are well-formed in H (T-FRAME1, T-FRAME2). A frame stack is well-formed if all its frames are well-typed. Rules T-FS-NA and T-FS-NA2 are required for ϵ-annotated frames. Well-typed heaps are defined as follows. Note that we make use of a predicate $reach(H, o, o')$ which holds iff object o' is reachable from o in H. The definition is standard and therefore omitted.

Definition 1 (Object Type)**.** For an object identifier $o \in dom(H)$ where $H(o) = \langle C, FM\rangle$, $typeof(H, o) := C$

$$\frac{L(x) = \texttt{null} \vee typeof(H, L(x)) <: \Gamma(x)}{H \vdash \Gamma; L; x}\ \text{(WF-VAR)}$$

$$\frac{dom(\Gamma) \subseteq dom(L) \qquad \forall x \in dom(\Gamma).\ H \vdash \Gamma; L; x}{H \vdash \Gamma; L}\ \text{(WF-ENV)}$$

$$H \vdash \epsilon\ \text{(T-EMPFS)}$$

$$\frac{\Gamma\ ;\ a \vdash t : \sigma \qquad H \vdash \Gamma; L}{H \vdash \langle L, t\rangle^l : \sigma}\ \text{(T-FRAME1)}$$

$$\frac{\Gamma, x : \tau\ ;\ a \vdash t : \sigma \qquad H \vdash \Gamma; L}{H \vdash^{\tau}_{x} \langle L, t\rangle^l : \sigma}\ \text{(T-FRAME2)}$$

$$\frac{H \vdash F^{\epsilon} : \sigma \qquad H \vdash FS}{H \vdash F^{\epsilon} \circ FS}\ \text{(T-FS-NA)} \qquad \frac{H \vdash^{\tau}_{x} F^{\epsilon} : \sigma \qquad H \vdash FS}{H \vdash^{\tau}_{x} F^{\epsilon} \circ FS}\ \text{(T-FS-NA2)}$$

$$\frac{H \vdash F^{x} : \tau \qquad H \vdash^{\tau}_{x} FS}{H \vdash F^{x} \circ FS}\ \text{(T-FS-A)} \qquad \frac{H \vdash^{\sigma}_{y} F^{x} : \tau \qquad H \vdash^{\tau}_{x} FS}{H \vdash^{\sigma}_{y} F^{x} \circ FS}\ \text{(T-FS-A2)}$$

Figure 11. Well-formed environments, frames, and frame stacks.

Definition 2 (Well-typed Heap)**.** A heap H is well-typed, written $\vdash H : \star$ iff

$$\begin{aligned}\forall o \in dom(H).\ H(o) = \langle C, FM\rangle \implies\ & (dom(FM) = fields(C) \wedge \\ & \forall f \in dom(FM).\ FM(f) = \texttt{null} \vee \\ & typeof(H, FM(f)) <: ftype(C, f))\end{aligned}$$

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

$\forall q, q' \in dom(H).$
$reach(H, o, q) \wedge reach(H, o', q') \implies q \neq q'.$

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

$F = \langle L, t\rangle^l \wedge \forall x \mapsto b(o), y \mapsto b(o') \in L.$
$o \neq o' \implies sep(H, o, o')$

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

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

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

$F = \langle L, t\rangle^l \wedge \forall x \mapsto b(o) \in L, o' \in dom(H).$
$reach(H, o, o') \implies ocap(typeof(H, o'))$

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

$$\frac{boxSep(H,F) \qquad boxObjSep(H,F) \qquad boxOcap(H,F) \qquad a = \texttt{ocap} \implies globalOcapSep(H,F)}{H\ ;\ a \vdash F\ \textbf{ok}} \quad \text{(F-OK)}$$

$$\frac{H\ ;\ a \vdash F\ \textbf{ok}}{H\ ;\ a \vdash F \circ \epsilon\ \textbf{ok}} \quad \text{(SINGFS-OK)}$$

$$\frac{H\ ;\ b \vdash F^l\ \textbf{ok} \qquad H\ ;\ a \vdash FS\ \textbf{ok} \qquad b = \begin{cases} \texttt{ocap} & \text{if } a = \texttt{ocap} \vee l = \epsilon \\ \epsilon & \text{otherwise} \end{cases} \qquad boxSeparation(H,F,FS) \qquad uniqueOpenBox(H,F,FS) \qquad openBoxPropagation(H,F^l,FS)}{H\ ;\ b \vdash F^l \circ FS\ \textbf{ok}} \quad \text{(FS-OK)}$$

Figure 12. Separation invariants of frames and frame stacks.

references are disjoint (Def. 5), and (c) all types reachable from box references are ocap (Def. 6).

Definition 7 (Global Ocap Separation)**.** For heap H and frame F, $globalOcapSep(H,F)$ holds iff

$F = \langle L, t\rangle^l \wedge \forall x \mapsto o \in L, y \mapsto o' \in L_0.$ $ocap(typeof(H,o)) \wedge sep(H,o,o')$

In addition, in a well-formed frame that is well-typed under effect ocap, non-box references have ocap types, and they are disjoint from the global variables in L_0 (Def. 7).

The judgement $H\ ;\ a \vdash F$ **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, $boxSeparation(H,F,FS)$ holds iff

$\forall o, o' \in dom(H).\ boxRoot(o,F) \wedge boxRoot(o',FS) \wedge o \neq o' \implies sep(H,o,o')$

Def. 8 uses auxiliary predicate $boxRoot$ shown in Figure 13. $boxRoot(o,F)$ holds iff there is a box reference to o in frame F; $boxRoot(o,FS)$ holds iff there is a box reference to o in one of the frames FS. Informally, $boxSeparation(H,F,FS)$ holds iff non-aliased boxes are disjoint.

Definition 9 (Unique Open Box)**.** For heap H, frame F, and frame stack FS, $uniqueOpenBox(H,F,FS)$ holds iff

$\forall o, o' \in dom(H).\ openbox(H,o,F,FS) \wedge openbox(H,o',F,FS) \implies o = o'$

Def. 9 uses auxiliary predicate $openbox$ shown in Figure 13. $openbox(H,o,F,FS)$ holds iff $boxRoot(o,FS)$ and there is a local variable in frame F which points to an object reachable from o (box o is "open" in frame F). Informally, $uniqueOpenBox(H,F,FS)$ holds iff at most one box is open (*i.e.,* accessible via non-box references) in frame F.

$$\frac{x \mapsto b(o) \in L}{boxRoot(o, \langle L, t\rangle^l)}$$

$$\frac{boxRoot(o,F)}{boxRoot(o, F \circ \epsilon)}$$

$$\frac{boxRoot(o,F) \vee boxRoot(o,FS)}{boxRoot(o, F \circ FS)}$$

$$\frac{boxRoot(o,FS) \qquad x \mapsto o' \in env(F) \qquad reach(H,o,o')}{openbox(H,o,F,FS)}$$

Figure 13. Auxiliary predicates.

Definition 10 (Open Box Propagation)**.** For heap H, frame F^l, and frame stack FS, $openBoxPropagation(H,F^l,FS)$ holds iff

$l \neq \epsilon \wedge FS = G \circ GS \wedge openbox(H,o,F,FS) \implies openbox(H,o,G,GS)$

Informally, $openBoxPropagation(H,F^l,FS)$ holds iff frame F^l preserves the open boxes in the top-most frame of frame stack FS.

According to rule FS-OK shown in Figure 12, well-formed frame stacks ensure (a) non-aliased boxes are disjoint (Def. 8), (b) at most one box is open (*i.e.,* accessible via non-box references) per frame (Def. 9), and (c) method calls preserve open boxes (Def. 10).

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* $\vdash H : \star$ *then:*

1. *If* $H \vdash F : \sigma$, $H\ ;\ a \vdash F$ ***ok****, and* $H, F \longrightarrow H', F'$ *then* $\vdash H' : \star$, $H' \vdash F' : \sigma$, *and* $H'\ ;\ a \vdash F'$ ***ok****.*
2. *If* $H \vdash FS$, $H\ ;\ a \vdash FS$ ***ok****, and* $H, FS \twoheadrightarrow H', FS'$ *then* $\vdash H' : \star$, $H' \vdash FS'$, *and* $H'\ ;\ b \vdash FS'$ ***ok****.*

Theorem 2 (Progress)**.** *If* $\vdash H : \star$ *then:*

If $H \vdash FS$ *and* $H\ ;\ a \vdash FS$ ***ok*** *then either* $H, FS \twoheadrightarrow H', FS'$ *or* $FS = \langle L, x\rangle^l \circ \epsilon$ *or* $FS = F \circ GS$ *where* $F = \langle L, \texttt{let}\ x = t\ \texttt{in}\ t'\rangle^l$, $t \in \{y.f, y.f = z, y.m(z),$ $y.\texttt{open}\ \{z \Rightarrow t''\}\}$, *and* $L(y) = \texttt{null}$.

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* $\vdash H : \star$ *then:*

If $H \vdash FS$, $H\ ;\ a \vdash FS$ ***ok****,* $H, FS \twoheadrightarrow H', FS'$, $FS = F \circ GS$, $F = \langle L, \texttt{let}\ x = y.f = z\ \texttt{in}\ t\rangle^l$,

$p ::= \overline{cd}\ \overline{vd}\ t$	program
$cd ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{fd}\ \ \overline{md}\}$	class
$vd ::= \texttt{var}\ f : C$	variable
$ud ::= \texttt{var}\ f : \texttt{Box}[C]$	unique field
$fd ::= vd \mid ud$	field
$md ::= \texttt{def}\ m(x : \sigma) : C = t$	method
$\sigma, \tau ::=$	surface type
$\quad C, D$	class type
$\quad \mid \texttt{Box}[C]$	box type
$\quad \mid \texttt{Null}$	null type
$\pi ::=$	type
$\quad \sigma, \tau$	surface type
$\quad Q \triangleright \texttt{Box}[C]$	guarded type
$\quad \bot$	bottom type

Figure 14. CLC^2 syntax. C, D range over class names, f, m, x range over term names. Q ranges over abstract types.

$L(y) = o'$, *boxRoot*(o, FS), *and* $reach(H, o, o')$, *then* $w \mapsto o \in env(G)$ *where* $G \in FS$.

Proof sketch. First, by $H\ ;\ a \vdash FS$ **ok**, FS-OK, and $FS = F \circ GS$ we have $H\ ;\ a \vdash F$ **ok**. By F-OK we have $boxObjSep(H, F)$. By def. 5 this means $u \mapsto b(o) \notin L$ and therefore $\neg boxRoot(o, F)$. Given that $L(y) = o'$, $reach(H, o, o')$, and $boxRoot(o, FS)$ it must be that $boxRoot(o, GS)$. Given that $boxRoot(o, GS)$ by def. *boxRoot* (Figure 13), there is a frame $G' \in GS$ such that $u \mapsto b(o) \in env(G')$. 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 $G \in FS$ such that $w \mapsto o \in env(G)$. □

3.3 Lightweight Affinity

This section introduces the CORELACASA^2 language (CLC^2) which extends CLC^1 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 Γ. 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 CLC^2 and CLC^1: first, field types are either class types C or box types $\texttt{Box}[C]$; second, we introduce a bottom type $\bot$ and *guarded types* $Q \triangleright \texttt{Box}[C]$ where Q ranges over a countably infinite supply of abstract types; third, we introduce *continuation terms*.

In CLC^2 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^c; these terms come in three forms:

1. A term $\texttt{box}[C]\ \{x \Rightarrow t\}$ 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 CLC^1 boxes are created using expressions of the form $\texttt{box}[C]$. In CLC^2 we require the continuation term t, because creating a box in addition creates a *permission* only available in t.
2. A term $\texttt{capture}(x.f, y)\ \{z \Rightarrow t\}$ 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 (a) y's permission is no longer available, and (b) z refers to box x.
3. A term $\texttt{swap}(x.f, y)\ \{z \Rightarrow t\}$ 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 CLC^2, 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 CLC^2, simplifying the meta-theory.

3.3.1 Dynamic Semantics

CLC^2 extends the dynamic semantics compared to CLC^1 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 CLC^1 with *permission sets* P. Thus, a frame $\langle L, t, P\rangle^l$ 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 CLC^2 for single frames are identical to the corresponding transition rules of CLC^1; the permission sets do not change.[7] In contrast, the transition rules for *frame stacks* affect the permission sets of frames.

The extended transition rules of CLC^2 are shown in Figure 16. Rule E-INVOKE 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"). Re-

[7] Therefore, the transition rules (trivially) extended with permission sets are only shown in Appendix A.

$$
\begin{array}{lll}
t ::= & & \text{terms} \\
& x & \text{variable} \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ t & \text{let binding} \\
& \mid t^c & \text{continuation term} \\
\\
e ::= & & \text{expressions} \\
& \texttt{null} & \text{null reference} \\
& \mid x & \text{variable} \\
& \mid x.f & \text{selection} \\
& \mid x.f = y & \text{assignment} \\
& \mid \texttt{new}\ C & \text{instance creation} \\
& \mid x.m(y) & \text{invocation} \\
& \mid x.\texttt{open}\ \{y \Rightarrow t\} & \text{open box} \\
\\
t^c ::= & & \text{continuation term} \\
& \texttt{box}[C]\ \{x \Rightarrow t\} & \text{box creation} \\
& \mid \texttt{capture}(x.f, y)\ \{z \Rightarrow t\} & \text{capture} \\
& \mid \texttt{swap}(x.f, y)\ \{z \Rightarrow t\} & \text{swap}
\end{array}
$$

Figure 15. CLC^2 terms and expressions.

$$
\frac{H(L(y)) = \langle C, FM\rangle \qquad mbody(C, m) = x \to t' \qquad L' = L_0[\texttt{this} \mapsto L(y), x \mapsto L(z)] \qquad P' = \emptyset \vee (L(z) = b(o,p) \wedge p \in P \wedge P' = \{p\})}{H, \langle L, \texttt{let}\ x = y.m(z)\ \texttt{in}\ t, P\rangle^l \circ FS \twoheadrightarrow H, \langle L', t', P'\rangle^x \circ \langle L, t, P\rangle^l \circ FS} \text{ (E-INVOKE)}
$$

$$
H, \langle L, x, P\rangle^y \circ \langle L', t', P'\rangle^l \circ FS \twoheadrightarrow H, \langle L'[y \mapsto L(x)], t', P'\rangle^l \circ FS \text{ (E-RETURN1)}
$$

$$
H, \langle L, x, P\rangle^\epsilon \circ \langle L', t', P'\rangle^l \circ FS \twoheadrightarrow H, \langle L', t', P'\rangle^l \circ FS \text{ (E-RETURN2)}
$$

$$
\frac{L(y) = b(o,p) \qquad p \in P \qquad L' = [z \mapsto o]}{H, \langle L, \texttt{let}\ x = y.\texttt{open}\ \{z \Rightarrow t'\}\ \texttt{in}\ t, P\rangle^l \circ FS \twoheadrightarrow H, \langle L', t', \emptyset\rangle^\epsilon \circ \langle L[x \mapsto L(y)], t, P\rangle^l \circ FS} \text{ (E-OPEN)}
$$

$$
\frac{o \notin dom(H) \qquad fields(C) = \overline{f} \qquad H' = H[o \mapsto \langle C, \overline{f \mapsto \texttt{null}}\rangle] \qquad p \text{ fresh}}{H, \langle L, \texttt{box}[C]\ \{x \Rightarrow t\}, P\rangle^l \circ FS \twoheadrightarrow H', \langle L[x \mapsto b(o,p)], t, P \cup \{p\}\rangle^\epsilon \circ \epsilon} \text{ (E-BOX)}
$$

Figure 16. CLC^2 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 CLC^1. 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.

$$
\frac{L(x) = b(o,p) \qquad L(y) = b(o',p') \qquad \{p,p'\} \subseteq P \qquad H(o) = \langle C, FM\rangle \qquad H' = H[o \mapsto \langle C, FM[f \mapsto o']\rangle]}{H, \langle L, \texttt{capture}(x.f, y)\ \{z \Rightarrow t\}, P\rangle^l \circ FS \twoheadrightarrow H', \langle L[z \mapsto L(x)], t, P \setminus \{p'\}\rangle^\epsilon \circ \epsilon} \text{ (E-CAPTURE)}
$$

$$
\frac{L(x) = b(o,p) \qquad L(y) = b(o',p') \qquad \{p,p'\} \subseteq P \qquad H(o) = \langle C, FM\rangle \qquad FM(f) = o'' \qquad p'' \text{ fresh} \qquad H' = H[o \mapsto \langle C, FM[f \mapsto o']\rangle]}{H, \langle L, \texttt{swap}(x.f, y)\ \{z \Rightarrow t\}, P\rangle^l \circ FS \twoheadrightarrow H', \langle L[z \mapsto b(o'',p'')], t, (P \setminus \{p'\}) \cup \{p''\}\rangle^\epsilon \circ \epsilon} \text{ (E-SWAP)}
$$

Figure 17. Transition rules for `capture` and `swap`.

$$
\frac{\Gamma_0, \texttt{this} : C, x : D\ ;\ \epsilon \vdash t : E' \qquad E' <: E}{C \vdash \texttt{def}\ m(x : D) : E = t} \text{ (WF-METHOD1)}
$$

$$
\frac{\Gamma = \Gamma_0, \texttt{this} : C, x : Q \triangleright \texttt{Box}[D], \texttt{Perm}[Q] \qquad Q \text{ fresh} \qquad \Gamma\ ;\ \epsilon \vdash t : E' \qquad E' <: E}{C \vdash \texttt{def}\ m(x : \texttt{Box}[D]) : E = t} \text{ (WF-METHOD2)}
$$

Figure 18. Well-formed CLC^2 methods.

Figure 17 shows CLC^2'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. CLC^2 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 $\texttt{Box}[D]$ is type-checked in an environment Γ which includes a static permission $\texttt{Perm}[Q]$ where Q is a fresh abstract type; furthermore, the parameter has a *guarded type* $Q \triangleright \texttt{Box}[D]$. This environment Γ ensures the method body has full access to the argument box.

The OCAP-* rules for CLC^2 treat box-typed method parameters analogously, and are left to the appendix.

Subclassing and Subtypes. In CLC^2, the subtyping relation $<:$ is identical to that of CLC^{1}except for one additional rule for the $\perp$ type:

$$\frac{\Gamma\ ;\ a \vdash x : C \qquad mtype(C, m) = \sigma \to \tau \qquad \Gamma\ ;\ a \vdash y : \sigma' \qquad \sigma' <: \sigma \vee \\ (\sigma = \mathsf{Box}[D] \wedge \sigma' = Q \triangleright \mathsf{Box}[D] \wedge \mathsf{Perm}[Q] \in \Gamma)}{\Gamma\ ;\ a \vdash x.m(y) : \tau} \text{ (T-INVOKE)}$$

$$\frac{a = \mathsf{ocap} \implies ocap(C) \qquad \forall\, \mathsf{var}\ f : \sigma \in \overline{fd}.\ \exists\, D.\ \sigma = D}{\Gamma\ ;\ a \vdash \mathsf{new}\ C : C} \text{ (T-NEW)}$$

$$\frac{\Gamma\ ;\ a \vdash x : Q \triangleright \mathsf{Box}[C] \qquad \mathsf{Perm}[Q] \in \Gamma \qquad y : C\ ;\ \mathsf{ocap} \vdash t : \sigma}{\Gamma\ ;\ a \vdash x.\mathsf{open}\ \{y \Rightarrow t\} : Q \triangleright \mathsf{Box}[C]} \text{ (T-OPEN)}$$

$$\frac{ocap(C) \qquad Q \text{ fresh} \qquad \Gamma, x : Q \triangleright \mathsf{Box}[C], \mathsf{Perm}[Q]\ ;\ a \vdash t : \sigma}{\Gamma\ ;\ a \vdash \mathsf{box}[C]\ \{x \Rightarrow t\} : \bot} \text{ (T-BOX)}$$

Figure 19. CLC^2 term and expression typing.

$$\bot <: \pi \quad (<:\text{-BOT})$$

The $<:$-BOT rule says that $\bot$ is a subtype of any type π. (Note that type `Null` is a subtype of any *surface type*, whereas π ranges over *all* types including guarded types.)

Term and Expression Typing. Figure 19 shows the changes in the type rules. In T-INVOKE, if the method parameter has a type $\mathsf{Box}[D]$ then the argument y must have a type $Q \triangleright \mathsf{Box}[D]$ such that the static permission $\mathsf{Perm}[Q]$ is available in Γ. (Otherwise, box y has been consumed.) T-NEW checks that none of the field types are box types. This makes sure classes with box-typed fields are only created using box expressions. (Box-typed fields are then accessible using swap.) T-OPEN requires the static permission $\mathsf{Perm}[Q]$ corresponding to the guarded type $Q \triangleright \mathsf{Box}[C]$ of the opened box x to be available in Γ; this ensures consumed boxes are never opened. Finally, T-BOX assigns a guarded type $Q \triangleright \mathsf{Box}[C]$ to the newly created box x where Q is a fresh abstract type; the permission $\mathsf{Perm}[Q]$ is available in the type context of the continuation term t. The box expression itself has type $\bot$, since reduction never "returns"; t is the (only) continuation.

Figure 20 shows the type rules for CLC^2's two new expressions. Both rules require x and y to have guarded types such that the corresponding permissions are available in Γ. In both cases the permission of y is removed from the environment used to type-check the continuation t; thus, box y is consumed in each case. In its continuation, `capture` provides access to box x under alias z; thus, z's type is equal to x's type. In contrast, `swap` extracts the value of a *unique field* and provides access to it under alias z in its continuation. CLC^2 ensures the value extracted from the unique field is externally unique. Therefore, the type of z is a guarded type $R \triangleright \mathsf{Box}[D]$ where R is fresh; permission $\mathsf{Perm}[R]$ is created for use in continuation t.

$$\frac{\Gamma\ ;\ a \vdash x : Q \triangleright \mathsf{Box}[C] \qquad \Gamma\ ;\ a \vdash y : Q' \triangleright \mathsf{Box}[D] \qquad \{\mathsf{Perm}[Q], \mathsf{Perm}[Q']\} \subseteq \Gamma \qquad D <: ftype(C, f) \qquad \Gamma \setminus \{\mathsf{Perm}[Q']\}, z : Q \triangleright \mathsf{Box}[C]\ ;\ a \vdash t : \sigma}{\Gamma\ ;\ a \vdash \mathsf{capture}(x.f, y)\ \{z \Rightarrow t\} : \bot} \text{ (T-CAPTURE)}$$

$$\frac{\Gamma\ ;\ a \vdash x : Q \triangleright \mathsf{Box}[C] \qquad \Gamma\ ;\ a \vdash y : Q' \triangleright \mathsf{Box}[D'] \qquad \{\mathsf{Perm}[Q], \mathsf{Perm}[Q']\} \subseteq \Gamma \qquad ftype(C, f) = \mathsf{Box}[D] \qquad D' <: D \qquad R \text{ fresh} \qquad \Gamma \setminus \{\mathsf{Perm}[Q']\}, z : R \triangleright \mathsf{Box}[D], \mathsf{Perm}[R]\ ;\ a \vdash t : \sigma}{\Gamma\ ;\ a \vdash \mathsf{swap}(x.f, y)\ \{z \Rightarrow t\} : \bot} \text{ (T-SWAP)}$$

Figure 20. Typing CLC^2's `capture` and `swap`.

Well-Formedness. CLC^2 extends CLC^1 with unique fields of type $\mathsf{Box}[C]$ (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 $\vdash H : \star$ iff

$$\begin{aligned} \forall o \in dom(H).\ H(o) = \langle C, FM \rangle \implies\ & (dom(FM) = fields(C) \wedge \\ & \forall f \in dom(FM).\ FM(f) = \mathtt{null} \vee \\ & (typeof(H, FM(f)) <: D \wedge \\ & \quad ftype(C, f) \in \{D, \mathsf{Box}[D]\})) \end{aligned}$$

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, $boxSep(H, F)$ holds iff

$F = \langle L, t, P \rangle^l \wedge \forall x \mapsto b(o, p), y \mapsto b(o', p') \in L.\ p \neq p' \wedge \{p, p'\} \subseteq P \implies sep(H, o, o')$

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 $boxObjSep$, box permissions are not required. Similarly, the differences in $boxOcap$ and $globalOcapSep$ 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 $F = \langle L, t, P\rangle^l$, $fieldUniqueness(H, F)$ holds iff
$\forall x \mapsto b(o,p) \in L, o', \hat{o} \in dom(H).$
$p \in P \ \wedge \ reach(H, o, \hat{o}) \ \wedge \ H(\hat{o}) = \langle C, FM\rangle \ \wedge$
$ftype(C, f) \ = \ \texttt{Box}[D] \ \wedge \ reach(H, FM(f), o') \implies$
$domedge(H, \hat{o}, f, o, o')$

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^2 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 $boxSep(H, F, FS)$ iff
$\forall o, o' \in dom(H).\ boxRoot(o, F, p) \wedge$
$boxRoot(o', FS, p') \wedge p \neq p' \implies sep(H, o, o')$

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* $\vdash H : \star$ *then:*

1. *If* $H \vdash F : \sigma$, $H \ ; \ a \vdash F$ ***ok****, and* $H, F \longrightarrow H', F'$ *then* $\vdash H' : \star$, $H' \vdash F' : \sigma$, *and* $H' \ ; \ a \vdash F'$ ***ok****.*
2. *If* $H \vdash FS$, $H \ ; \ a \vdash FS$ ***ok****, and* $H, FS \twoheadrightarrow H', FS'$ *then* $\vdash H' : \star$, $H' \vdash FS'$, *and* $H' \ ; \ b \vdash FS'$ ***ok****.*

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

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

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

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

$\sigma, \tau ::=$	surface type
$\ldots$	
$\mid \texttt{Proc}[C]$	process type
$e ::=$	expression
$\ldots$	
$\mid \texttt{proc}\ \{(x : \texttt{Box}[C]) \Rightarrow t\}$	process creation
$t^c ::=$	continuation term
$\ldots$	
$\mid \texttt{send}(x, y)\ \{z \Rightarrow t\}$	message send

Figure 21. Syntax extensions for concurrency.

Importantly, for a well-formed frame configuration, CLC^2 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 $\texttt{Proc}[C]$ for processes capable of receiving messages of type $\texttt{Box}[C]$. An expression of the form $\texttt{proc}\ \{(x : \texttt{Box}[C]) \Rightarrow t\}$ creates a concurrent process which applies the function $\{(x : \texttt{Box}[C]) \Rightarrow t\}$ to each received message. A continuation term of the form $\texttt{send}(x, y)\ \{z \Rightarrow t\}$ 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 $\mathcal{P}$. 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* $\langle \texttt{Box}[C], M, x \rightarrow t\rangle$ where $\texttt{Box}[C]$ is the type of messages the process can receive, M is a set of object identifiers representing (buffered) incoming messages, and $x \rightarrow t$ is the message handler function. CLC^3 introduces a third reduction relation $H, \mathcal{P} \rightsquigarrow H', \mathcal{P}'$ which reduces a set of processes $\mathcal{P}$ 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 $\epsilon^{o'}$ 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

$$\frac{F = \langle L, \texttt{let}\ x = \texttt{proc}\ \{(y : \texttt{Box}[C]) \Rightarrow t'\}\ \texttt{in}\ t, P\rangle^l \quad F' = \langle L[x \mapsto o'], t, P\rangle^l \quad o' \text{ fresh} \quad H' = H[o' \mapsto \langle \texttt{Box}[C], \emptyset, y \to t'\rangle]}{H, \{(F \circ FS)^o\} \cup \mathcal{P} \rightsquigarrow H', \{(F' \circ FS)^o, \epsilon^{o'}\} \cup \mathcal{P}} \text{(E-PROC)}$$

$$\frac{F = \langle L, \texttt{send}(x, y)\ \{z \Rightarrow t\}, P\rangle^l \quad L(x) = o \quad H(o) = \langle \texttt{Box}[C], M, f\rangle \quad L(y) = b(o', p') \quad p' \in P \quad F' = \langle L[z \mapsto o], t, P \setminus \{p'\}\rangle^\epsilon \quad H' = H[o \mapsto \langle \texttt{Box}[C], M \cup \{o'\}, f\rangle]}{H, F \circ FS \twoheadrightarrow H', F' \circ \epsilon} \text{(E-SEND)}$$

$$\frac{H(o) = \langle \texttt{Box}[C], M, x \to t\rangle \quad M = M' \uplus \{o'\} \quad F = \langle L, y, P\rangle^l \quad F' = \langle \emptyset[x \mapsto b(o', p)], t, \{p\}\rangle^\epsilon \quad p \text{ fresh} \quad H' = H[o \mapsto \langle \texttt{Box}[C], M', x \to t\rangle]}{H, \{(F \circ \epsilon)^o\} \cup \mathcal{P} \rightsquigarrow H', \{(F' \circ \epsilon)^o\} \cup \mathcal{P}} \text{(E-RECEIVE)}$$

Figure 22. CLC^3 process transition rules.

$$\frac{H \vdash FS^o \quad H \vdash \mathcal{P}}{H \vdash \{FS^o\} \cup \mathcal{P}} \text{(WF-SOUP)}$$

$$\frac{H \vdash FS \quad H\ ;\ a \vdash FS\ \textbf{ok}}{H \vdash FS^o} \text{(WF-PROC)}$$

Figure 23. CLC^3 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 $\uplus$ 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 CLC^3 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). CLC^3 also extends the object capability rules: $ocap(C) \implies ocap(\texttt{Proc}[C])$

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 $\texttt{Perm}[Q]$ of sent box y to be available in context Γ. The body t of the continuation closure must be well-typed in a context where $\texttt{Perm}[Q]$ is no longer available. As with all continuation terms, the type of a send term is $\bot$.

$$\frac{x : Q \triangleright \texttt{Box}[C], \texttt{Perm}[Q]\ ;\ \texttt{ocap} \vdash t : \pi \quad Q \text{ fresh}}{\Gamma\ ;\ a \vdash \texttt{proc}\ \{(x : \texttt{Box}[C]) \Rightarrow t\} : \texttt{Proc}[C]} \text{(T-PROC)}$$

$$\frac{\Gamma\ ;\ a \vdash x : \texttt{Proc}[C] \quad \Gamma\ ;\ a \vdash y : Q \triangleright \texttt{Box}[D] \quad \texttt{Perm}[Q] \in \Gamma \quad D <: C \quad \Gamma \setminus \{\texttt{Perm}[Q]\}, z : \texttt{Proc}[C]\ ;\ a \vdash t : \pi}{\Gamma\ ;\ a \vdash \texttt{send}(x, y)\ \{z \Rightarrow t\} : \bot} \text{(T-SEND)}$$

Figure 24. CLC^3 typing rules.

$$\frac{x \mapsto o \in L \vee (x \mapsto b(o, p) \in L \wedge p \in P)}{accRoot(o, \langle L, t, P\rangle^l)} \text{(ACC-F)}$$

$$\frac{accRoot(o, F) \vee accRoot(o, FS)}{accRoot(o, F \circ FS)} \text{(ACC-FS)}$$

$$\frac{\forall o, o' \in dom(H).\ (accRoot(o, FS) \wedge accRoot(o', FS')) \Rightarrow sep(H, o, o')}{isolated(H, FS, FS')} \text{(ISO-FS)}$$

$$\frac{H(o) = \langle \texttt{Box}[C], M, f\rangle \quad H(o') = \langle \texttt{Box}[D], M', g\rangle \quad \forall q \in M, q' \in M'.\ sep(H, q, q') \quad isolated(H, FS, GS)}{isolated(H, FS^o, GS^{o'})} \text{(ISO-PROC)}$$

Figure 25. CLC^3 process and frame stack isolation.

Figure 25 defines a predicate $isolated$ to express isolation of frame stacks and processes. Isolation of frame stacks builds on an $accRoot$ predicate: identifier o is an accessible root in frame F, written $accRoot(o, F)$, iff $env(F)$ 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 $accRoot(o, F)$ 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* $\vdash H : \star$ *then:*
If $H \vdash \mathcal{P}, \forall P, P' \in \mathcal{P}.\ P \neq P' \implies isolated(H, P, P')$, *and* $H, \mathcal{P} \rightsquigarrow H', \mathcal{P}'$ *then* $\vdash H' : \star$, $H' \vdash \mathcal{P}'$, *and* $\forall Q, Q' \in \mathcal{P}'.\ Q \neq Q' \implies isolated(H', Q, Q')$.

Theorem 5 states that $\rightsquigarrow$ 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 for type `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`:

```
1  def swap[S](select: T => Box[S])
2             (assign: (T, Box[S]) => Unit, b: Box[S])
3       (fun: Spore[Packed[S], Unit] {
4               type Excluded = b.C
5             })
6       (implicit acc: CanAccess { type C = b.C }): Unit
```

`select` and `assign` are field accessor functions (see Section 2); the LaCasa plugin must enforce that they are trivial function literals. The box `b` is put into the unique field. The implicit `acc` parameter ensures the availability of `b`'s permission. Crucially, `b`'s permission is consumed by the assignment to the unique field. Therefore, `b` must not be accessed in the continuation spore `fun`, which is expressed using the spore's `Excluded` type member. As a result, `fun`'s body can no longer capture the permission. Note that it is impossible to capture a permission indirectly via another object or closure, since permissions are confined to the stack.

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):

```
1  s.next.send(packed.box)({
2    // 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.[8] 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:

```
1  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/3e64b28ad4e71256213e2bd6e8bd68a9978a2486/src/main/scala/net/lag/kestrel/KestrelHandler.scala for a usage example in a large-scale production system.

```
1  class MessageHandlerTask(
2          receiver: Actor[T],
3          packed: Packed[T]) extends Runnable {
4    def run(): Unit = {
5      // process message in ‘packed‘ object
6      try {
7        // invoke ‘receive‘ method of ‘receiver‘ actor
8        receiver.receive(packed.box)(packed.access)
9      } catch {
10       case nrc: NoReturnControl => /* do nothing */
11     }
12     // check for next message
13     ...
14   }
15   ...
16 }
```

Figure 26. Handling `NoReturnControl` within actors.

in a `try-catch` (see below), or within a worker thread of the actor system's thread pool. In the latter case, the task that executes actor code catches the `NoReturnControl` exception, as shown in Figure 26. Note that the exception handler is at the actor's "top level:" after processing the received message (in `packed`) the `receiver` actor is ready to process the next message (if any).

Scala's standard library provides a special `ControlThrowable` type for such cases where exceptions are used to manage control flow. The above `NoReturnControl` type extends `ControlThrowable`. The latter is defined as follows:

```
trait ControlThrowable extends Throwable
                       with NoStackTrace
```

Mixing in the `NoStackTrace` trait disables the generation of JVM stack traces, which is expensive and not needed. The `ControlThrowable` type enables exception handling without disturbing exception-based control-flow transfers:

```
try { ... } catch {
  case c: ControlThrowable => throw c // propagate
  case e: Exception => ...
}
```

Crucially, exceptions of a subtype of `ControlThrowable` are propagated in order not to influence the in-progress control flow transfer. Patterns such as the above are unchecked in Scala. However, in the case of LaCasa, failure to propagate `ControlThrowable`s could result in unsoundness. For example, consider the following addition of a `try-catch` to the previous example (shown at the beginning of Section 5.1):

```
1  try {
2    s.next.send(packed.box)({
3      // continuation closure
4    })(access)
5  } catch {
6    case c: ControlThrowable => // do nothing
7  }
8  other.send(packed.box)({
9    ...
10 })(access)
```

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 `ControlThrowable`s, or (b) all catch clauses matching `ControlThrowable`s 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/traits 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 [26]). 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/-traits 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

Project	Version	SLOC	GitHub stats	#classes/traits	#ocap (%)	#dir. insec. (%)
Scala Standard Library	2.11.7	33,107	★5,795 👥257	1,505	644 (43%)	212/861 (25%)
Signal/Collect	8.0.6	10,159	★123 👥11	236	159 (67%)	60/77 (78%)
GeoTrellis	0.10.0-RC2		★400 👥38			
-engine		3,868		190	40 (21%)	124/150 (83%)
-raster		22,291		670	233 (35%)	325/437 (74%)
-spark		9,192		326	101 (31%)	167/225 (74%)
Total		78,617		2,927	1,177 (40%)	888/1,750 (51%)

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).

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 [52]. 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 [24], 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.[9] 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.

[9] Analogous rules have been used for static fields in Joe-E [44], an object-capability secure subset of Java.

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 [15, 16, 51, 57, 64, 65]. In contrast, LaCasa provides external uniqueness [18], which allows internally-aliased object graphs. Permissions in LaCasa indicate which 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, 48] 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 [35]. Several systems combine ownership with concurrency control to prevent data races. RaceFree Java [1] 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

few source annotations. However, LaCasa supports ownership transfer, which is outside the domain of Loci. Kilim [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 [59] and FlexoTasks [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 Rakhylia 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 Viktor Kuncak for feedback on an earlier design of LaCasa.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst*, 28(2):207–255, 2006.

[2] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.

[4] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, pages 32–59, 1997.

[5] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In *OOPSLA*, pages 233–249, 2014.

[6] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, pages 249–272. Springer, 2016.

[7] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin. Experience report: Developing the Servo web browser engine using Rust. *CoRR*, abs/1505.07383, 2015.

[8] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[9] J. S. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for Java. In *LCTES*, pages 1–11, 2008.

[10] H. G. Baker. 'use-once' variables and linear objects - storage management, reflection and multi-threading. *SIGPLAN Notices*, 30(1):45–52, 1995.

[11] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, Apr. 2003.

[12] G. M. Bierman, C. V. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'n' play: Formalizing asynchronous C#. In *ECOOP*, pages 233–257, 2012.

[13] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[14] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw, Pract. Exper*, 31(6):533–553, 2001.

[15] J. Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.

[16] L. Caires and J. C. Seco. The type discipline of behavioral separation. In *POPL*, pages 275–286, 2013.

[17] City of Asheville, NC, USA. Priority Places project. `http://priorityplaces.ashevillenc.gov/`.

[18] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.

[19] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *APLAS*, pages 139–154, 2008.

[20] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 15–58. Springer, 2013.

[21] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[22] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *AGERE!@SPLASH*, pages 1–12. ACM, 2015.

[23] D. Crockford. ADsafe. `http://www.adsafe.org`, 2011.

[24] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360, 2010.

[25] B. C. d. S. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: a new foundation for generic programming. In *PLDI*, pages 35–44, 2012.

[26] A. Danial and contributors. cloc. http://cloc.sourceforge.net/, 2006. Accessed: 2016-03-20.

[27] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[28] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[29] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and using pluggable type-checkers. In *ICSE*, pages 681–690, 2011.

[30] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, pages 273–298, 2007.

[31] J. Epstein, A. P. Black, and S. L. P. Jones. Towards Haskell in the cloud. In *Haskell*, pages 118–129, 2011.

[32] Ericsson AB. Erlang/OTP. https://github.com/erlang/otp, 2010. Accessed: 2016-07-10.

[33] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

[34] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

[35] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, pages 21–40, 2012.

[36] P. Haller. On the integration of the actor model in mainstream technologies: The Scala perspective. In *AGERE!@SPLASH*, pages 1–6, 2012.

[37] P. Haller and A. Loiko. Object capabilities and lightweight affinity in Scala: Implementation, formalization, and soundness. *CoRR*, abs/1607.05609, 2016. URL http://arxiv.org/abs/1607.05609.

[38] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[39] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, pages 354–378, 2010.

[40] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell*, 8(3):323–364, 1977.

[41] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285, 1991.

[42] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst*, 23(3):396–450, 2001.

[43] Lightbend, Inc. Akka. http://akka.io/, 2009. Accessed: 2016-03-20.

[44] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *NDSS*, 2010.

[45] H. Miller, P. Haller, and M. Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP*, pages 308–333, 2014.

[46] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[47] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. *Google, Inc., Tech. Rep*, 2008.

[48] N. H. Minsky. Towards alias-free pointers. In *ECOOP*, pages 189–209, 1996.

[49] P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478, 2007.

[50] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.

[51] M. Odersky. Observers for linear types. In *ESOP*, pages 390–407, 1992.

[52] M. Odersky and A. Moors. Fighting bit rot with types. In *FSTTCS*, pages 427–451, 2009.

[53] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *POPL*, pages 41–53, 2001.

[54] M. Odersky, P. Altherr, V. Cremet, G. Dubochet, B. Emir, P. Haller, S. Micheloud, N. Mihaylov, A. Moors, L. Rytz, M. Schinz, E. Stenman, and M. Zenger. The Scala language specification version 2.11. http://www.scala-lang.org/files/archive/spec/2.11/, Apr. 2014.

[55] J. Östlund and T. Wrigstad. Welterweight Java. In *TOOLS*, pages 97–116, 2010.

[56] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. *CoRR*, abs/1506.07813, 2015.

[57] F. Pottier and J. Protzenko. Programming with permissions in Mezzo. In *ICFP*, pages 173–184, 2013.

[58] L. Rytz. *A Practical Effect System for Scala*. PhD thesis, EPFL, Lausanne, Switzerland, Sept. 2013.

[59] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in Java. In *OOPSLA*, pages 211–228, 2007.

[60] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP*, pages 104–128, 2008.

[61] P. Stutz, A. Bernstein, and W. W. Cohen. Signal/Collect: Graph algorithms for the (semantic) web. In *ISWC*, pages 764–780, 2010.

[62] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL*, pages 188–201, 1994.

[63] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput*, 132(2):109–176, 1997.

[64] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

[65] E. M. Westbrook, J. Zhao, Z. Budimlic, and V. Sarkar. Practical permissions for race-free parallelism. In *ECOOP*, pages 614–639, 2012.

[66] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, pages 445–469, 2009.

[67] Z. Y. *Concurrency Analysis Based On Fractional Permission System*. PhD thesis, University of Wisconsin–Milwaukee, 2007.

$$\frac{x \mapsto b(o,p) \in L \qquad p \in P}{boxRoot(o, \langle L, t, P\rangle^l)}$$

$$\frac{x \mapsto b(o,p) \in L \qquad p \in P}{boxRoot(o, \langle L, t, P\rangle^l, p)}$$

$$\frac{boxRoot(o, F, p)}{boxRoot(o, F \circ \epsilon, p)}$$

$$\frac{boxRoot(o, F, p) \vee boxRoot(o, FS, p)}{boxRoot(o, F \circ FS, p)}$$

Figure 30. Auxiliary predicates.

$$\frac{}{\begin{array}{c}H, \langle L, \texttt{let } x = \texttt{null in } t, P\rangle^l \\ \longrightarrow\ H, \langle L[x \mapsto \texttt{null}], t, P\rangle^l\end{array}}\ \text{(E-NULL)}$$

$$\frac{}{\begin{array}{c}H, \langle L, \texttt{let } x = y \texttt{ in } t, P\rangle^l \\ \longrightarrow\ H, \langle L[x \mapsto L(y)], t, P\rangle^l\end{array}}\ \text{(E-VAR)}$$

$$\frac{H(L(y)) = \langle C, FM\rangle \qquad f \in dom(FM)}{\begin{array}{c}H, \langle L, \texttt{let } x = y.f \texttt{ in } t, P\rangle^l \\ \longrightarrow\ H, \langle L[x \mapsto FM(f)], t, P\rangle^l\end{array}}\ \text{(E-SELECT)}$$

$$\frac{\begin{array}{c}L(y) = o \qquad H(o) = \langle C, FM\rangle \\ H' = H[o \mapsto \langle C, FM[f \mapsto L(z)]\rangle]\end{array}}{\begin{array}{c}H, \langle L, \texttt{let } x = y.f = z \texttt{ in } t, P\rangle^l \\ \longrightarrow\ H', \langle L, \texttt{let } x = z \texttt{ in } t, P\rangle^l\end{array}}\ \text{(E-ASSIGN)}$$

$$\frac{\begin{array}{c}o \notin dom(H) \qquad fields(C) = \overline{f} \\ H' = H[o \mapsto \langle C, \overline{f \mapsto \texttt{null}}\rangle]\end{array}}{\begin{array}{c}H, \langle L, \texttt{let } x = \texttt{new } C \texttt{ in } t, P\rangle^l \\ \longrightarrow\ H', \langle L[x \mapsto o], t, P\rangle^l\end{array}}\ \text{(E-NEW)}$$

Figure 31. CLC^2 frame transition rules.

$$\frac{\begin{array}{c}p \vdash \texttt{class } C \texttt{ extends } D\ \{\overline{fd}\ \overline{md}\} \\ C \vdash_{ocap} \overline{md} \qquad ocap(D) \\ \forall\ \texttt{var } f : \sigma \in \overline{fd}.\ ocap(\sigma) \vee \sigma = \texttt{Box}[E] \wedge ocap(E)\end{array}}{ocap(C)}$$

(OCAP-CLASS)

$$\frac{\texttt{this} : C, x : D\ ;\ \texttt{ocap} \vdash t : E' \qquad E' <: E}{C \vdash_{ocap} \texttt{def } m(x : D) : E = t}$$

(OCAP-METHOD1)

$$\frac{\begin{array}{c}\Gamma = \texttt{this} : C, x : Q \triangleright \texttt{Box}[D], \texttt{Perm}[Q] \qquad Q \text{ fresh} \\ \Gamma\ ;\ \texttt{ocap} \vdash t : E' \qquad E' <: E\end{array}}{C \vdash_{ocap} \texttt{def } m(x : \texttt{Box}[D]) : E = t}$$

(OCAP-METHOD2)

Figure 28. Well-formed ocap classes.

$$\frac{\begin{array}{c}\gamma : permTypes(\Gamma) \longrightarrow P \text{ injective} \\ \forall x \in dom(\Gamma), \\ \Gamma(x) = Q \triangleright \texttt{Box}[C] \wedge L(x) = b(o,p) \wedge \texttt{Perm}[Q] \in \Gamma \implies \\ \gamma(Q) = p\end{array}}{\vdash \Gamma; L; P}$$

(WF-PERM)

$$\frac{\begin{array}{c}L(x) = \texttt{null} \vee \\ L(x) = o \wedge typeof(H, o) <: \Gamma(x) \vee \\ L(x) = b(o,p) \wedge \Gamma(x) = Q \triangleright \texttt{Box}[C] \wedge \\ typeof(H, o) <: C\end{array}}{H \vdash \Gamma; L; x}$$

(WF-VAR)

$$\frac{\Gamma\ ;\ a \vdash t : \sigma \qquad l \neq \epsilon \implies \sigma <: C \qquad H \vdash \Gamma; L \qquad H \vdash \Gamma; L; P}{H \vdash \langle L, t, P\rangle^l : \sigma}$$

(T-FRAME1)

$$\frac{\Gamma, x : \tau\ ;\ a \vdash t : \sigma \qquad l \neq \epsilon \implies \sigma <: C \qquad H \vdash \Gamma; L \qquad H \vdash \Gamma; L; P}{H \vdash^{\tau}_{x} \langle L, t, P\rangle^l : \sigma}$$

(T-FRAME2)

$$\frac{\begin{array}{c}boxSep(H, F) \qquad boxObjSep(H, F) \\ boxOcap(H, F) \\ a = \texttt{ocap} \implies globalOcapSep(H, F) \\ fieldUniqueness(H, F)\end{array}}{H\ ;\ a \vdash F\ \textbf{ok}}$$

(F-OK)

Figure 29. Frame and frame stack typing.

A. Additional Rules and Definitions

Figure 28 shows CLC^2's OCAP-* rules. Figure 29 shows the updated rules for frame and frame stack typing in CLC^2. 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
$F = \langle L, t, P\rangle^l \wedge \forall x \mapsto b(o,p), y \mapsto o' \in L.\ sep(H, o, o')$

Definition 16 (Box Ocap Invariant)**.** Frame F satisfies the *box ocap* invariant in H, written $boxOcap(H, F)$, iff
$F = \langle L, t, P\rangle^l \wedge \forall x \mapsto b(o,p) \in L, o' \in dom(H).\ p \in P \wedge reach(H, o, o') \implies ocap(typeof(H, o'))$

Definition 17 (Global Ocap Separation)**.** Frame F satisfies the *global ocap separation* invariant in H, written $globalOcapSep(H, F)$, iff
$F = \langle L, t, P\rangle^l \wedge \forall x \mapsto o \in L, y \mapsto o' \in L_0.$
$ocap(typeof(H, o)) \wedge sep(H, o, o')$

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