This is the accepted version of a paper presented at *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Citation for the original published paper:

https://doi.org/10.1145/3022671.2984042

N.B. When citing this work, cite the original published paper.

© Author | ACM 2016. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, http://dx.doi.org/10.1145/3022671.2984042.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-197902
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, 45]. 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
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 [27]).

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

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

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

Selected Related Work. Most closely related to our system are approaches based on permissions or capabilities. Of particular relevance is previous work on capabilities for uniqueness in Scala by Haller and Odersky [39] (“Cap4S”). While this prior work shares our high-level goal of lightweight uniqueness for instances of the class. This has two important consequences: (a) a minimal set of additional annotations (a single bit of information per class) enables reusing existing code, and (b) type checking reusable class declarations is simple and well-understood, following the object-capability discipline, which we adapt for Scala.

2. Overview

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

A First Example. Consider the case of asynchronous communication between two concurrent processes. This style of concurrency is well-supported by the actor model [21] [40] for which multiple implementations exist for Scala [36] [38] [43]. Figure 1 shows the definition of two actor classes.

The behavior of each actor is implemented by overriding the receive method inherited from superclass Actor. The receive method is invoked by the actor runtime system whenever an actor is ready to process an incoming message. In the example, whenever ActorA has received an instance of class Start, it creates an instance of class Message, initializes the instance with an integer array, and sends the instance to the next actor.

1 LACASA is available under an open-source license at [https://github.com/phaller/lacasa/](https://github.com/phaller/lacasa/).
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 \[\text{[3]}\]. 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 \[\text{[32]}\]).

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.\[\text{[3]}\]

**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()`; 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

```
1  class ActorA extends Actor[message] {  2      override def receive(box: Box[message]) {  3          implicit acc: CanAccess (type C = box.C) {  4              box.open(  5                  case s: Start =>  6                      mbox[Box][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  }
20  class ActorB extends Actor[message] {  21      override def receive(box: Box[message]) {  22          implicit acc: CanAccess (type C = box.C) {  23              box.open(msg =>  24                  println(msg.arr.mkString(","))  25              ) (acc) 26      }
27  }
```

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

```
  x => x match {  2      case pat1 => e1  3                        ...  4      case patn => en  5                      }
```

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
```

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

\[\text{[3]}\] In ongoing work we are developing adapter classes to conveniently integrate LACASA and the Akka actor library.
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 \(C = \text{box.C}\) \} where box is a parameter of type Box\([\text{Any}]\). 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 such that box 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\([\text{Message}]\) as well as an access permission. Besides a type argument, mkBox also receives a closure of the form \{ packed => \ldots \}. 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.** 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(\(\{ x => \ldots \}\))(acc). In contrast, passing the access permission acc implicitly enables the more lightweight Scala syntax box.open(\(\{ x => \ldots \}\).

**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 \([51]\) for details of implicit resolution.
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 of a unique field and replaces it with another box.

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).

---

```scala
class ActorA(next: ActorRef[C])
  extends Actor[Container] {
    def receive(msg: Box[Container]) {
      // ...
      msg.swap(_.part1)(_.part1 = _, packed.box)(
        spore { pack =>
          val acc = pack.access
          // ...
          spore acc
        })(acc)
    }
    next.send(pack.box)((
      // ...
      })(acc)
  }
}
```

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

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 CORELACASA¹ (CLC¹), a typed, object-oriented core language with object capabilities.

**Syntax.** Figure 4 and Figure 5 show the syntax of CLC¹. A program consists of a sequence of class definitions, `cd`, a sequence of global variable declarations, `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, `vd`, and methods, `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 `Box[C]`, and the Null type `Null`. Null is a subtype of all class types; it is used to assign a type to null.

In order to simplify the presentation of the operational semantics, programs are written in A-normal form (ANF) which requires all subexpressions to be named. We enforce ANF by introducing two separate syntactic categories for *terms* and *expressions*, shown in Figure 5. Terms are either variables or let bindings. Let bindings introduce names for intermediate results. Most expressions are standard, except that the usual object-based expressions, namely field selections, field assignments, and method invocations, have only variables as trivial subexpressions. The instance creation ex-

---

Figure 4. CLC¹ syntax. `C`, `D` range over class names, `f`, `md`, `x` range over term names.
A single frame \( F \) in heap \( H \) comprises a stack of frames \( FS \). 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 \( o \in \text{dom}(H) \) 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 \( \text{dom}(H) \cup \{ \text{null} \} \). The first relation reduces \( FS \) to \( FS' \), and \( H, FS \rightarrow H', FS' \). The second relation reduces entire frame stacks \( FS \) in heap \( H \).

A single frame \( F = \langle L, t \rangle \) consists of a variable environment \( L = \text{env}(F) \), a term \( t \), and an annotation \( l \). The variable environment \( L \) maps variable names \( x \) to values \( v \in \text{dom}(H) \cup \{ \text{null} \} \cup \{ b(o) \mid o \in \text{dom}(H) \} \). A value \( b(o) \) is a box reference created using \( \text{Box}[C] \) 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-existent), 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, \( \text{fields}(C) \) denotes the fields of class \( C \), and \( \text{mbody}(C, f) = x \rightarrow t \) denotes the body of a method \( \text{def } m(x : \sigma) : \tau = t \).

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

**Single Frame Reduction.** Figure 5 shows single frame transition rules. Thanks to the fact that terms are in ANF in our core language, the reduced term is a let binding in each case. This means reduction results can be stored immediately in the variable environment, avoiding the introduction of locations or references in the core language syntax. Rule E-Box is analogous to rule E-New, except that variable \( x \) is bound to a box reference \( b(o) \). As a result, fields of the encapsulated object \( o \) are not accessible using regular field selection and assignment, since rules E-Select and E-Assign would not be applicable. Apart from E-Box the transition rules are similar to previous stack-based formalizations of class-based core languages with objects [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
Well-formed programs. Figure 9 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 type environment $\Gamma_0 = \{ \text{global} : C_0 \}$ (WF-PROGRAM). Rule WF-CLASS defines well-formed class definitions. In a well-formed class definition (a) all methods are well-formed, (b) the superclass is either AnyRef or a well-formed class in the same program, and (c) method overriding (if any) is well-formed; fields may not be overridden. We use a standard function $\text{fields}(D)$ [42] to obtain the fields in class $D$ and superclasses of $D$. Rule WF-OVERRIDE defines well-formed method overriding: overriding of a method $m$ in class $C$ with superclass $D$ is well-formed if $D$ (transitively) does not define a method $m$ or the type of $m$ in $D$ is the same as the type of $m$ in $C$. A method $m$ is well-typed in class $C$ if its body is well-typed with type $\tau'$ under effect $\epsilon$ in environment $\Gamma_0$, this : $C, x : \sigma$, where $\sigma$ is the type of $m$’s parameter $x$, such that $\tau'$ is a subtype of $m$’s declared result type $\tau$ (WF-METHOD).

Object Capabilities. For a class $C$ to satisfy the constraints of the object-capability discipline, written $ocap(C)$, 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 $\Gamma_ocap$. 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 $\Gamma_0$; thus, global variables are inaccessible. Second, the method body must be well-typed under effect ocap; this means that within the method body only ocap classes may be instantiated.

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

$$C : \ll D$$

$\text{Box} [C] : \ll \text{Box} [D]$ ($\ll$-Box)

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 $\text{Box} [C]$ provided $ocap(C)$. 

---

**Figure 7.** CLC frame stack transition rules.

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

$$\begin{align*}
C & \vdash \overline{md} \\
D & = \text{AnyRef} \lor p \vdash \text{class } D \ldots \\
\forall (\text{def } m \ldots) & \in \overline{md}, \text{override}(m, C, D) \\
\forall \text{var } f : \sigma & \in \overline{td}, f \notin \text{fields}(D) \\
p & \vdash \text{class } C \text{ extends } D \{ \overline{td} \overline{md} \} \quad (\text{WF-CLASS})
\end{align*}$$

$$mtype(m, D) \text{ not defined} \lor mtype(m, D) = mtype(m, C)$$

$$\text{override}(m, C, D) \quad (\text{WF-OVERRIDE})$$

$$\Gamma_0, \text{this : } C, x : \sigma ; \epsilon \vdash \overline{t} : \tau'$$

$$\tau' < : \tau \quad (\text{WF-METHOD})$$

**Figure 8.** Well-formed CLC programs.

E-RETURN2 enables returning from an $\epsilon$-annotated frame. Rule E-OPEN creates such an $\epsilon$-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 $\Gamma_0$; 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 $\sigma$ to term $t$ in type environment $\Gamma$ under effect $a$. When assigning a type to the top-level term of a program the effect $a$ is $\epsilon$ 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 9 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 type environment $\Gamma_0 = \{ \text{global} : C_0 \}$ (WF-PROGRAM). Rule WF-CLASS defines well-formed class definitions. In a well-formed class definition (a) all methods are well-formed, (b) the superclass is either AnyRef or a well-formed class in the same program, and (c) method overriding (if any) is well-formed; fields may not be overridden. We use a standard function $\text{fields}(D)$ [42] to obtain the fields in class $D$ and superclasses of $D$. Rule WF-OVERRIDE defines well-formed method overriding: overriding of a method $m$ in class $C$ with superclass $D$ is well-formed if $D$ (transitively) does not define a method $m$ or the type of $m$ in $D$ is the same as the type of $m$ in $C$. A method $m$ is well-typed in class $C$ if its body is well-typed with type $\tau'$ under effect $\epsilon$ in environment $\Gamma_0$, this : $C, x : \sigma$, where $\sigma$ is the type of $m$’s parameter $x$, such that $\tau'$ is a subtype of $m$’s declared result type $\tau$ (WF-METHOD).

Object Capabilities. For a class $C$ to satisfy the constraints of the object-capability discipline, written $ocap(C)$, 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 $\Gamma_ocap$. 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 $\Gamma_0$; thus, global variables are inaccessible. Second, the method body must be well-typed under effect ocap; this means that within the method body only ocap classes may be instantiated.

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

$$C : \ll D$$

$\text{Box} [C] : \ll \text{Box} [D]$ ($\ll$-Box)

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 $\text{Box} [C]$ provided $ocap(C)$.
\[ \Gamma; a \vdash \text{null : Null} \]
\[ \text{(T-NULL)} \quad \Gamma; x \in \text{dom}(\Gamma) \]
\[ \Gamma; a \vdash x : \Gamma(x) \quad \text{(T-VAR)} \]
\[ \Gamma; a \vdash e : \tau \quad \Gamma; x : \tau; a \vdash t : \sigma \]
\[ \Gamma; a \vdash \text{let } e = t \text{ in } t : \sigma \quad \text{(T-LET)} \]
\[ \Gamma; a \vdash x : C \quad \text{ftype}(C, f) = D \]
\[ \Gamma; a \vdash x.f : D \quad \text{(T-SELECT)} \]
\[ \Gamma; a \vdash x : C \quad \text{ftype}(C, f) = D \]
\[ \Gamma; a \vdash y : D' \quad D' < : D \quad \text{(T-ASSIGN)} \]
\[ \Gamma; a \vdash x.m(y) : \tau \quad \text{(T-INVOC)} \]
\[ \text{ocap}(C) \quad \Gamma; a \vdash \text{box}[C] : \text{Box}[C] \quad \text{(T-BOX)} \]
\[ \Gamma; a \vdash x : \text{Box}[C] \quad y : C \quad \text{ocap } t : \sigma \]
\[ \Gamma; a \vdash \text{open } \{ y \mapsto t \} : \text{Box}[C] \quad \text{(T-OPEN)} \]

**Figure 10.** CLC^1 term and expression typing.

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, \( \Gamma, L \) are well-formed in heap \( H \) if for all variables \( x \in \text{dom}(\Gamma) \) the type of \( L(x) \) in \( H \) is a subtype of the static type of \( x \) in \( \Gamma \) (WF-VAR, WF-ENV). A frame \( \langle L, t \rangle \) is well-formed in \( H \) if its term \( t \) is well-formed in some environment \( \Gamma \) such that \( \Gamma, L \) are well-formed in \( H \) (T-FRAME1, T-FRAME2). A frame stack is well-formed if all its frames are well-formed. Rules T-FS-NA and T-FS-NA2 are required for \( \epsilon \)-annotated frames.

Well-typed heaps are defined as follows. Note that we make use of a predicate \( \text{reach}(H, o, o') \) 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 \( o \in \text{dom}(H) \) where \( H(o) = \langle C, FM \rangle \), the type of \( H(o) \) is \( C \).

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

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

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

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

**Definition 3 (Separation).** Two object identifiers \( o \) and \( o' \) are separate in heap \( H \), written \( \text{sep}(H, o, o') \), iff

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

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

\[ F = \langle L, t \rangle \land \forall x \mapsto b(o), y \mapsto b(o') \in L. o \neq o' \implies \text{sep}(H, o, o') \]

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

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

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

\[ F = \langle L, t \rangle \land \forall x \mapsto b(o), \text{reach}(H, o, o') \implies \text{ocap}(\text{typeof}(H, o')) \]

In a well-formed frame, (a) two box references that are not aliases are disjoint (Def. 3), (b) box references and non-box
Definition 7 (Global Ocap Separation). For heap $H$ and frame $F$, $globalOcapSep(H, F)$ holds iff

$$F = \langle L, t \rangle \land \forall x \rightarrow o \in L, y \rightarrow o' \in L_0, ocap(typeof(H(o))) \land sep(H, o, o')$$

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

Definition 8 (Box Separation). For heap $H$, frame $F$, and frame stack $FS$, $boxSeparation(H, F, FS)$ holds iff

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

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

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

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 $F$.

Figure 12. Separation invariants of frames and frame stacks.

3.2 Soundness and Heap Separation

Type soundness of CLC 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 $\vdash H : *$ then:

1. If $H \vdash F : \sigma, H; a \vdash F \text{ ok}$ and $H, F \rightarrow H', F'$ then $H' : \star, H' \vdash F' : \sigma$, and $H': a \vdash F' \text{ ok}$.
2. If $H \vdash FS, H; a \vdash FS \text{ ok}$ and $H, FS \rightarrow H', FS'$ then $H' : \star, H' \vdash FS', H' : b \vdash FS' \text{ ok}$.

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

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

The following corollary expresses an essential heap separation invariant enforced by CLC. 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 $\vdash H : *$ then:

If $H \vdash FS, H; a \vdash FS \text{ ok}$, $H, FS \rightarrow H', FS'$, $FS = F \circ GS$, $F = \langle L, \text{let } x = y.f = z \text{ in } t \rangle$, $x \mapsto b(o) \in L$ $\Rightarrow$ $boxRoot(o, \langle L, t \rangle)$ $\Rightarrow$ $boxRoot(o, F)$ $\Rightarrow$ $boxRoot(o, F \circ \epsilon)$ $\Rightarrow$ $boxRoot(o, F \circ \epsilon)$ $\Rightarrow$ $boxRoot(o, F \circ \epsilon)$ $\Rightarrow$ $reach(H, o, o')$ $\Rightarrow$ $openbox(H, o, F, FS)$
that (def. 4) and other objects (def. 5) reachable in OK is well-formed and by F-well-formedness of all its frames (FS-OK). This section introduces the CLC\textsuperscript{2} language (CLC\textsuperscript{2}) which extends CLC\textsuperscript{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 \( \Gamma \). Our notion of affinity is based on continuation terms: consumption of permissions, and, thus, boxes, is only possible in contexts where an explicit continuation is provided. The consumed permission is then no longer available in the continuation.

\textbf{Syntax.} Figure 14 and Figure 15 show the syntactic differences between CLC\textsuperscript{2} and CLC\textsuperscript{1}; first, field types are either class types \( C \) or box types \( \Box[C] \); second, we introduce a bottom type \( \bot \) and guarded types \( Q \bowtie \Box[C] \) where \( Q \) ranges over a countably infinite supply of abstract types; third, we introduce continuation terms.

In CLC\textsuperscript{2} types are divided into surface types which can occur in the surface syntax, and general types, including guarding types, which cannot occur in the surface syntax; guarded types are only introduced by type inference (see Section 3.3.2). The bottom type \( \bot \) is the type of continuation terms \( t \); these terms come in three forms:

1. A term \( \Box[C] \{ x \to t \} \) 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\textsuperscript{1} boxes are created using expressions of the form \( \Box[C] \). In CLC\textsuperscript{2} we require the continuation term \( t \), because creating a box in addition creates a permission only available in \( t \).

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

3. A term \( \text{swap}(x,y) \{ z \to t \} \) 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\textsuperscript{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\textsuperscript{2}, simplifying the meta-theory.

\subsection{3.3 Lightweight Affinity} This section introduces the CORELACASA\textsuperscript{2} language (CLC\textsuperscript{2}) which extends CLC\textsuperscript{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 \( \Gamma \). Our notion of affinity is based on continuation terms: consumption of permissions, and, thus, boxes, is only possible in contexts where an explicit continuation is provided. The consumed permission is then no longer available in the continuation.

\textbf{Syntax.} Figure 14 and Figure 15 show the syntactic differences between CLC\textsuperscript{2} and CLC\textsuperscript{1}; first, field types are either class types \( C \) or box types \( \Box[C] \); second, we introduce a bottom type \( \bot \) and guarded types \( Q \bowtie \Box[C] \) where \( Q \) ranges over a countably infinite supply of abstract types; third, we introduce continuation terms.

In CLC\textsuperscript{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 \( t \); these terms come in three forms:

1. A term \( \Box[C] \{ x \to t \} \) 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\textsuperscript{1} boxes are created using expressions of the form \( \Box[C] \). In CLC\textsuperscript{2} we require the continuation term \( t \), because creating a box in addition creates a permission only available in \( t \).

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

3. A term \( \text{swap}(x,y) \{ z \to t \} \) 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\textsuperscript{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\textsuperscript{2}, simplifying the meta-theory.

\subsection{Dynamic Semantics} CLC\textsuperscript{2} extends the dynamic semantics compared to CLC\textsuperscript{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\textsuperscript{1} with permission sets \( P \). Thus, a frame \( \langle L, t, P \rangle \) combines a variable environment \( L \) and term \( t \) with a set of permissions \( P \). (As before, the label \( l \) is used for transferring return values from method invocations.)

The transition rules of CLC\textsuperscript{2} for single frames are identical to the corresponding transition rules of CLC\textsuperscript{1}; the permission sets do not change\footnote{Therefore, the transition rules (trivially) extended with permission sets are only shown in Appendix A.} In contrast, the transition rules for frame stacks affect the permission sets of frames.

The extended transition rules of CLC\textsuperscript{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-
$$t ::= \begin{array}{l}
x \\
| \text{let } x = e \text{ in } t \\
| \text{continue } t' \end{array}$$

$$e ::= \begin{array}{l}
\text{null} \\
| x \\
| x.f \\
| \text{new } C \\
| x.m(y) \\
| x.open \{ y \mapsto t \} \end{array}$$

$$t' ::= \begin{array}{l}
\boxed{C} \{ x \mapsto t \} \\
\text{capture}(x.f, y) \{ z \mapsto t \} \\
\text{swap}(x.f, y) \{ z \mapsto t \} \end{array}$$

**Figure 15.** CLC$^2$ terms and expressions.

$$H(L(y)) = \langle C, FM \rangle$$

$$mbody(C, m) = x \mapsto t'$$

$$L' = L_0[\text{this } \mapsto L(y), x \mapsto L(z)]$$

$$P' = \emptyset \lor (L(z) = b(o, p) \land p \in P \land P' = \{ p \})$$

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

(E-INVoke)

$$H, \langle L, x, P \rangle^y \circ \langle L', t', P'^l \circ FS \rangle \rightarrow H, \langle L'[y] \mapsto L(x) \rangle, t', P'^l \circ FS$$

(E-RETURN1)

$$H, \langle L', x, P \rangle \circ \langle L', t', P'^l \circ FS \rangle \rightarrow H, \langle L', t', P'^l \circ FS \rangle$$

(E-RETURN2)

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

$$H, \langle L, \text{let } x = y, \text{open } \{ z \mapsto t' \} \text{ in } t, P \rangle^l \circ FS \rightarrow H, \langle L', t, P \rangle^l \circ FS$$

(E-OPEN)

$$o \notin \text{dom}(H) \quad \text{fields}(C) = \emptyset$$

$$H' = H[o \mapsto (C, \overrightarrow{\text{null}})]$$

$$p \text{ fresh }$$

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

(E-Box)

**Figure 16.** CLC$^2$ frame stack transition rules.

**Section 3.3 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 Box[D] is type-checked in an environment $\Gamma$ which includes a static permission $\text{Perm}(Q)$ where $Q$ is a fresh abstract type; furthermore, the parameter has a guarded type $Q \gg \text{Box}(D)$. This environment $\Gamma$ ensures the method body has full access to the argument box.

The $\text{OCAP}^*$ 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 $\ll$ is identical to that of CLC$^1$ except for one additional rule for the $\bot$ type:
Figure 19. CLC² term and expression typing.

\[ \bot \ll : \pi \ (\ll : - \text{Bot}) \]

The \( \ll : - \text{Bot} \) rule says that \( \bot \) is a subtype of any type \( \pi \). (Note that type \( \text{Null} \) is a subtype of any surface type, whereas \( \pi \) 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 Box\([D]\) then the argument \( y \) must have a type Box\([D]\) such that the static permission Perm\([Q]\) is available in \( \Gamma \). (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 Perm\([Q]\) corresponding to the guarded type Box\([C]\) of the opened box \( x \) to be available in \( \Gamma \); this ensures consumed boxes are never opened. Finally, T-Box assigns a guarded type Box\([C]\) to the newly created box \( x \) where \( Q \) is a fresh abstract type; the permission 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 CLC²'s two new expressions. Both rules require \( x \) and \( y \) to have guarded types such that the corresponding permissions are available in \( \Gamma \). 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. CLC² ensures the value extracted from the unique field is externally unique. Therefore, the type of \( z \)

\[ \Gamma ; a \vdash x : C \quad \text{mtype}(C, m) = \sigma \rightarrow \tau \]
\[ \Gamma ; a \vdash y : \sigma' \quad \sigma' <: \sigma \lor \]
\[ (\sigma = \text{Box}[D] \land \sigma' = \text{Box}[D] \land \text{Perm}[Q] \in \Gamma) \]
\[ \Gamma ; a \vdash x.m(y) : \tau \quad \text{(T-Invoke)} \]

\[ a = \text{ocap} \implies \text{ocap}(C) \]
\[ \forall \, \var f : \sigma \in \text{fd}. \exists \, D, \sigma = D \]
\[ \Gamma ; a \vdash \text{new } C : C \quad \text{(T-New)} \]

\[ \Gamma ; a \vdash x : \text{Box}[C] \quad \text{Perm}[Q] \in \Gamma \]
\[ y : C ; \text{ocap} \vdash t : \sigma \]
\[ \Gamma ; a \vdash x.\text{open } \{ y \Rightarrow t \} : \text{Box}[C] \quad \text{(T-Open)} \]

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

**Well-Formedness.** CLC² extends CLC¹ with unique fields of type Box\([C]\) (see Figure 14); the following refined definition of well-typed heaps in CLC² reflects this extension:

**Definition 11 (Well-typed Heap).** A heap \( H \) is well-typed, written \( \vdash H : * \iff \)

\[ \forall o \in \text{dom}(H) \quad H(o) = (C, \text{FM}) \implies \]
\[ (\text{dom}(\text{FM}) = \text{fields}(C) \land \]
\[ \forall f \in \text{dom}(\text{FM}) \quad \text{FM}(f) = \text{nul} \lor \]
\[ (\text{typeof}(H, \text{FM}(f)) <: D \land \]
\[ \text{ftype}(C, f) \in \{ D, \text{Box}[D] \}) \]

The most interesting additions of CLC² with respect to well-formedness concern (a) separation invariants and (b) field uniqueness. In CLC¹, two boxes \( x \) and \( y \) are separate as long as \( x \) is not an alias of \( y \). In CLC², 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 CLC² is defined as follows:

**Definition 12 (Box Separation).** For heap \( H \) and frame \( F \), boxSep\((H, F)\) holds iff

\[ F = \{ L, t, P \} \land \forall x : b(o, p), y : b(o', p') \in L, p \neq p' \land \{ p, p' \} \subseteq F \implies \text{sep}(H, o, o') \]

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

CLC²’s addition of unique fields requires a new field uniqueness invariant for well-formed frames:
Definition 13 (Field Uniqueness). For heap \( H \) and frame \( F = \langle L, t, P \rangle \), fieldUniqueness\( (H, F) \) holds iff
\[
\forall x \mapsto b(o, p) \in L, o', \delta \in \text{dom}(H), p \in P \wedge \text{reach}(H, o, \delta) \wedge H(\delta) = \langle C, FM \rangle \wedge ftype(C, f) = \text{Box}[D] \wedge \text{reach}(H, FM(f), o') \implies \text{domedge}(H, \delta, f, o, o')
\]
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 \( \delta \) must “go through” that unique field. In other words, in all reference paths from \( o \) to \( o' \), the edge \((\delta, f)\) is a dominating edge. (A precise definition of \text{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 \( \text{boxSep}(H, F, FS) \) iff
\[
\forall o, o' \in \text{dom}(H). \text{boxRoot}(o, F, p) \wedge \text{boxRoot}(o', FS, p') \wedge p \neq p' \implies \text{sep}(H, o, o')
\]
Note that the availability of permissions is required indirectly by the \text{boxRoot} predicate (its other details are uninteresting, and therefore omitted).

4. Soundness

Theorem 3 (Preservation). If \( \vdash H : \ast \) then:
1. If \( H \vdash F : \sigma, H ; a \vdash F \text{ ok} \) and \( H, F \rightarrow H', F' \) then \( H' : \ast, H' \vdash F' : \sigma, H'; a \vdash F' \text{ ok} \).
2. If \( H \vdash FS, H ; a \vdash FS \text{ ok} \) and \( H, FS \rightarrow H', FS' \) then \( H' : \ast, H' \vdash FS', H'; b \vdash FS' \text{ ok} \).

Proof. Part (1) is proved by induction on the derivation of \( H, F \rightarrow H', F' \). Part (2) is proved by induction on the derivation of \( H, FS \rightarrow H', FS' \) and part (1). (See the companion technical report \[57\] for the full proof.)

Theorem 4 (Progress). If \( \vdash H : \ast \) then:
If \( H \vdash FS \) and \( a \vdash FS \text{ ok} \) then either \( H, FS \rightarrow H', FS' \) or \( FS = \langle L, x, P \rangle \circ e \) or \( FS = F \circ GS \) where
\[
\begin{align*}
F &= \langle L, \text{let } x = t \rightarrow t', P \rangle, t \in \{ y, f, y, f = z, y, m(z), y, \text{open} \{ z \Rightarrow t'' \} \}, \text{and } L(x) = \text{null} \text{ or }\\
F &= \langle L, \text{capture}(x, y) \{ z \Rightarrow t \}, P \rangle \text{ where } L(x) = \text{null} \land L(y) = \text{null} \text{ or }\\
F &= \langle L, \text{swap}(x, f, y) \{ z \Rightarrow t \}, P \rangle \text{ where } L(x) = \text{null} \land L(y) = \text{null}.
\end{align*}
\]

Proof. By induction on the derivation of \( H \vdash FS \). (See the companion technical report \[57\] for the full proof.)

4.1 Isolation

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

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

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

Figure 22 shows the process transition rules. Rule E-PRO\(C \) 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 \text{proc} expression. The new process \( e^o \) 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
is discarded, since send is a continuation term. In rule E-RECEIVE process $o$ is ready to process a message from its non-empty set of incoming messages $M$, since (the term in) frame $F$ cannot be reduced further and there are no other frames on the frame stack. (The $\cup$ operator denotes disjoint set union.) Frame $F'$ starts message processing with the parameter bound to a box reference with a fresh permission.

**Static Semantics.** Figure 23 shows the well-formedness rules that $\text{CLC}^3$ introduces for (sets of) processes. A set of processes is well-formed if each process is well-formed (WF-PROC). A process is well-formed if its frame stack is well-formed (WF-SOUP). $\text{CLC}^3$ also extends the object capability rules: $\text{ocap}(C) \implies \text{ocap}(\text{Proc}[C])$.

Figure 24 shows the typing of process creation and message sending. Rule T-PROC requires the body of a new process to be well-typed in an environment that only contains the parameter of the message handler and a matching access permission which is fresh. Importantly, body term $t$ is type-checked under effect $\text{Perm}[Q]$.

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

If $H \vdash \mathcal{P}, \forall Q, Q' \in \mathcal{P}, Q \neq Q' \implies \text{isolated}(H, P, P')$, and $H \vdash \mathcal{P} \rightarrow \mathcal{P}', \forall Q, Q' \in \mathcal{P}', Q \neq Q' \implies \text{isolated}(H', Q, Q')$.

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

5. Implementation

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

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 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:

```scala
```

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 scope does not capture anything.

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

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

5.1 Discarding the Stack Using Exceptions

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

```scala
s.next.send(packed.box)() {
  // continuation closure
  } // 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 where the stack of callers is discarded when an actor suspends with just a continuation closure. The implementation consists of throwing an exception which unwinds the call stack up to the actor’s event-loop, or up to the boundary of a concurrent task.

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

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

The thrown NoReturnControl exception is caught either within the main thread where the main method is wrapped
class MessageHandlerTask{
    receiver: Actor[T],
    packed: Packed[T]) extends Runnable {
    def run(): Unit = {
        // process message in ‘packed’ object
        try {
            // invoke ‘receive’ method of ‘receiver’ actor
            receiver.receive(packed.box)(packed.access)
        } catch {
            case nrc: NoReturnControl => /* do nothing */
        }
        // check for next message
        ...
    }
}

Figure 26. Handling NoReturnControl within actors.

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 {
    // process message in 'packed' object
    try {
        // invoke 'receive' method of 'receiver' actor
        receiver.receive(packed.box)(packed.access)
    } 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 ControlThrowables 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):

try {
    s.next.send(packed.box)({
        // continuation closure
    })(access)
} catch {
    case c: ControlThrowable => // do nothing
    other.send(packed.box)({
        ...
    })(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 ControlThrowables, or (b) all catch clauses matching ControlThrowables propagate caught exceptions. Furthermore, to support trusted LACASA code, a marker method permits unsafe catches.

6. Empirical Evaluation

The presented approach to object isolation and uniqueness is based on object capabilities. Isolation is enforced only for instances of ocap classes, i.e., classes adhering to the object-capability discipline. Likewise, ownership transfer is supported only for instances of ocap classes. Therefore, it is important to know whether the object-capability discipline imposes an undue burden on developers; or whether, on the contrary, developers tend to design classes and traits in a way that naturally follows the object-capability discipline. Specifically, our empirical evaluation aims to answer the following question: How many classes/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/trait LACASA supports isolation and ownership transfer “out of the box,” i.e., without code changes. (We will refer to both classes and traits as “classes” in the following.)

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

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

Results. Figure 27 shows the collected statistics.

For Scala’s standard library we found that 43% of all classes follow the object-capability discipline. While this
number might seem low, it is important to note that a strict form of ocap checking was used: accesses to top-level singleton objects were disallowed, even if these singletons were themselves immutable and object-capability safe. Thus, classes directly using helper singletons were marked as insecure. Interestingly, only 25% of the insecure classes directly access top-level singleton objects. This means, the majority of insecure classes is insecure due to dependencies on other insecure classes. These results can be explained as follows. First, helper singletons (in particular, “companion objects”) play an important role in the architecture of Scala’s collections package [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. To measure the impact of such singletons on the proportion of ocap classes, we reanalyzed S/C with knowledge of safe singletons in the standard library. As a result, the percentage of ocap classes increased from 67% to 79%, while the proportion of directly insecure classes remained identical. Thus, knowledge of “safe singleton objects” is indeed important for object capabilities in Scala.

7. Other Related Work

A number of previous approaches leverages permissions or capabilities for uniqueness or related notions. Approaches limited to tree-shaped object structures for unique references include [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 [23] 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 [55]. Several systems combine ownership with concurrency control to prevent data races. RaceFree Java [11] associates fields with locks, and an effect system ensures correct lock acquisition. Boyapati et al. [13] and Zhao [67] extend type system guarantees to deadlock prevention.

Our system takes important inspiration from Loci [66], a type system for enforcing thread locality which requires very
few source annotations. However, LACASA supports ownership transfer, which is outside the domain of Loci. Kilm [60] combines type qualifiers with an intra-procedural shape analysis to ensure isolation of Java-based actors. To simplify 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 Victor Kuncak for feedback on an earlier design of LACASA.

References

\[
x \mapsto b(o, p) \in L \quad \text{if} \quad p \in P
\]

\[
\text{boxRoot}(o, \langle L, t, P \rangle^t)
\]

\[
x \mapsto b(o, p) \in L \quad \text{if} \quad p \in P
\]

\[
\text{boxRoot}(o, \langle L, t, P \rangle^t, p)
\]

\[
\text{boxRoot}(o, F, p)
\]

\[
\text{boxRoot}(o, F \circ \epsilon, p)
\]

\[
\text{boxRoot}(o, F, p) \vee \text{boxRoot}(o, FS, p)
\]

\[
\text{boxRoot}(o, F \circ FS, p)
\]

\[\forall f : \text{permTypes}(\Gamma) \rightarrow P \text{ injective} \]

\[\forall x \in \text{dom}(\Gamma), \Gamma(x) = Q \triangleright \text{Box}[C] \land L(x) = b(o, p) \land \text{Perm}[Q] \in \Gamma \Rightarrow \gamma(Q) = p \]

\[\Gamma \vdash L; P \]

\[\text{boxSep}(H, F) \quad \text{boxObjSep}(H, F) \quad \text{boxOCap}(H, F) \quad a = \text{ocap} \implies \text{globalOCapSep}(H, F) \]

\[\text{fieldUniqueness}(H, F) \]

\[H \vdash a \vdash F \text{ ok} \]

\[\text{WF-VAR} \]

\[\text{T-FRAME2} \]

\[\text{T-FRAME1} \]

\[\text{E-NEW} \]

\[\text{E-ASSIGN} \]

\[\text{E-VAR} \]

\[\text{E-NUL} \]

\[\text{E-SELECT} \]

A. Additional Rules and Definitions

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

Definition 15 (Box-Object Separation). Frame F satisfies the box-object separation invariant in H, written boxObjSep(H, F), iff

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

Definition 16 (Box Ocap Invariant). Frame F satisfies the box ocap invariant in H, written boxOcap(H, F), iff

\[F = (\langle L, t, P \rangle^t \land \forall x \mapsto b(o, p) \in L, o' \in \text{dom}(H), p \in P \land \text{reach}(H, o, o') \implies \text{ocap}(\text{typeof}(H, o'))) \]

Definition 17 (Global Ocap Separation). Frame F satisfies the global ocap separation invariant in H, written globalOcapSep(H, F), iff

\[F = (\langle L, t, P \rangle^t \land \forall x \mapsto o \in L, y \mapsto o' \in L_0. \text{ocap}(\text{typeof}(H, o)) \land \text{sep}(H, o, o') \]

Definition 18 (Dominating Edge). Field f of o is a dominating edge for paths from o to o' in H, written domedge(H, o, f, o', o'), iff

\[\forall P \in \text{path}(H, o, o'), P = o \ldots \hat{o}. FM(f) \ldots o' \]

where H(\hat{o}) = \langle C, FM \rangle and f \in \text{dom}(FM).