Isolated Actors for Race-Free Concurrent Programming

THÈSE N^{O} 4874 (2010)

PRÉSENTÉE LE 26 NOVEMBRE 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Philipp HALLER

acceptée sur proposition du jury:

Prof. B. Faltings, président du jury
Prof. M. Odersky, directeur de thèse
Prof. D. Clarke, rapporteur
Prof. V. Kuncak, rapporteur
Prof. P. Müller, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

Abstract

Message-based concurrency using actors has the potential to scale from multi-core processors to distributed systems. However, several challenges remain until actor-based programming can be applied on a large scale.

First, actor implementations must be efficient and highly scalable to meet the demands of large-scale distributed applications. Existing implementations for mainstream platforms achieve high performance and scalability only at the cost of flexibility and ease of use: the control inversion introduced by event-driven designs and the absence of fine-grained message filtering complicate the logic of application programs.

Second, common requirements pertaining to performance and interoperability make programs prone to concurrency bugs: reusing code that relies on lower-level synchronization primitives may introduce livelocks; passing mutable messages by reference may lead to data races.

This thesis describes the design and implementation of Scala Actors. Our system offers the efficiency and scalability required by large-scale production systems, in some cases exceeding the performance of state-of-the-art JVM-based actor implementations. At the same time, the programming model (a) avoids the control inversion of event-driven designs, and (b) supports a flexible message reception operation. Thereby, we provide experimental evidence that Erlang-style actors can be implemented on mainstream platforms with only a modest overhead compared to simpler actor abstractions based on inversion of control. A novel integration of event-based and thread-based models of concurrency enables a safe reuse of lock-based code from inside actors.

Finally, we introduce a new type-based approach to actor isolation which avoids data races using unique object references. Simple, static capabilities are used to enforce a flexible notion of uniqueness and at-most-once consumption of unique references. Our main point of innovation is a novel way to support internal aliasing of unique references which leads to a surprisingly simple type system, for which we provide a complete soundness proof. Using an implementation as a plug-in for the EPFL Scala compiler, we show that the type system can be integrated into full-featured languages. Practical experience with collection classes

and actor-based concurrent programs suggests that the system allows type checking real-world Scala code with only few changes.

Keywords: Concurrent programming, actors, threads, events, join patterns, chords, aliasing, linear types, unique pointers, capabilities

Kurzfassung

Nachrichtenbasierte Nebenläufigkeit mit Aktoren hat das Potential von Mehrkern-Prozessoren hin zu verteilten System zu skalieren. Es gibt jedoch noch mehrere Herausforderungen zu meistern bis aktorenbasierte Programmierung im grossen Massstab angewandt werden kann.

Zum einen werden effiziente Implementierungen benötigt, die hochgradig skalierbar sind, um den Anforderungen moderner verteilter Anwendungen gerecht zu werden. Existierende Implementierungen für verbreitete Plattformen erreichen hohe Leistung und Skalierbarkeit nur auf Kosten von Flexibilität und Benutzbarkeit: Die Steuerfluss-Inversion, die ereignisbasierte Entwürfe mit sich bringen, und das Fehlen von feingranularer Nachrichtenfilterung führen oft dazu, dass die Anwendungslogik deutlich komplizierter wird.

Zum anderen bringen Leistungs- und Interoperabilitätsanforderungen oft eine erhöhte Anfälligkeit für Synchronisierungsfehler mit sich: Die Wiederverwendung von Quellcode, der auf Synchronisierungsmechanismen einer niedrigeren Abstraktionsebene basiert, kann Livelocks zur Folge haben; das Senden von Referenzen auf nichtkonstante Daten als Nachrichten kann zu Dataraces führen.

Diese Dissertation beschreibt den Entwurf und die Implementierung von Scala Actors. Unser System stellt die Effizienz und Skalierbarkeit zur Verfügung, die für grosse Systeme in Produktionsumgebungen erforderlich ist, wobei in manchen Fällen die Leistung anderer Javabasierter Aktorimplementierungen deutlich übertroffen wird. Gleichzeitig wird vom Programmiermodell (a) die Steuerfluss-Inversion ereignisbasierter Entwürfe vermieden, und (b) eine flexible Nachrichtenempfangsoperation unterstützt. Damit zeigen wir mit Hilfe experimenteller Ergebnisse, dass Erlang-Aktoren mit nur geringem Overhead im Vergleich zu einfacheren Programmiermodellen, die auf Steuerfluss-Inversion basieren, auf weitverbreiteten Plattformen implementiert werden können. Eine neuartige Integration von ereignisbasierten und threadbasierten Nebenläufigkeitsmodellen erlaubt eine sichere Wiederverwendung von lockbasiertem Quellcode innerhalb von Aktoren.

Im letzten Teil der Dissertation führen wir einen neuen typbasierten Ansatz zur Isolierung von Aktoren ein, bei dem Dataraces mit Hilfe von eindeutigen Objektreferenzen vermieden werden. Einfache, statische Capabilities werden genutzt

um sowohl eine flexible Form von Referenzeindeutigkeit als auch den höchstens einmaligen Verbrauch eindeutiger Referenzen sicherzustellen. Unsere wichtigste Innovation ist eine neuartige Methode, internes Aliasing eindeutiger Referenzen zu erlauben, was zu einem erstaunlich einfachen Typsystem führt; wir stellen einen vollständigen Beweis der Typsicherheit unseres Systems zur Verfügung. Mit Hilfe einer Implementierung als Plugin für den EPFL Scala-Compiler zeigen wir, dass das Typsystem in umfangreiche, produktionsreife Sprachen integriert werden kann. Praktische Experimente mit Collections und aktorbasierten, nebenläufigen Programmen zeigen, dass das System die Typprüfung praktisch benutzbaren Scala-Quellcodes erlaubt, wobei nur wenige zusätzliche Änderungen benötigt werden.

Stichwörter: Nebenläufige Programmierung, Aktoren, Threads, Ereignisse, Join-Kalkül, Chords, Alias-Analyse, Lineare Typen, Eindeutige Referenzen, Capabilities

Acknowledgements

I am deeply indebted to my advisor Martin Odersky for his support and insight, without which this dissertation would not have been possible. More than once he encouraged me to work out another less-than-half-baked idea which ended up getting published, and eventually formed the heart of this thesis.

I'd like to thank the past and present members of the Scala team at EPFL for providing an outstanding research environment.

I want to thank Tom Van Cutsem for sharing his passion for concurrent programming, and for his contributions to a joint paper which forms the basis of chapter 3 of this dissertation.

I'd also like to thank my other committee members Dave Clarke, Peter Müller, and Viktor Kuncak for their time and helpful feedback on drafts of the material presented in this dissertation.

Finally, my deepest thanks go to my family and friends for enjoying the highs of doctoral school together with me, and for supporting me during the unavoidable lows.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In today's computing landscape it is paramount to find viable solutions to pervasive concurrency. On the one hand, application programmers have to structure their programs in a way that leverages the resources of current and future multicore processors. On the other hand, concurrency is an intrinsic aspect of emerging computing paradigms, such as web applications and cloud computing.

The two main approaches to concurrency are shared memory and message passing. In the shared memory approach, the execution of concurrent threads of control is typically synchronized using locks or monitors. Locking has a simple semantics, and can be implemented efficiently [10]; however, it suffers from well-known problems pertaining to correctness, liveness, and scalability [72].

Several researchers have proposed software transactional memory to overcome the problems of locking in shared-memory concurrency [75, 112, 5]. However, it is not yet clear, whether the induced overhead can be made small enough to make software transactions practical [24].

The above concerns lead us to explore concurrent programming based on message passing in this thesis. In message-based concurrency, programs are structured as collections of processes (or agents, or actors) that share no common state. Messages are the only way of synchronization and communication. There are two categories of message-based systems: actor-based systems and channel-based systems. In actor-based systems [76, 3], messages are sent directly to processes. Channel-based systems introduce channels as an intermediary abstraction: messages are sent to channels, which can be read by one or more processes. In distributed systems, channels are usually restricted to be readable only by a single process. Although channel-based concurrency has been studied more extensively by the research community (*e.g.*, the π-calculus [92]), in practice actor-based systems are more wide-spread.

One of the earliest popular implementations of actor-based concurrency is the Erlang programming language [8], which was created by Ericsson. Erlang sup-

ports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes [7, 95]. The language was used at first in telecommunication systems, but is now also finding applications in internet commerce, such as Amazon's SimpleDB [113]. Erlang's strong separation between address spaces of processes ensures that its concurrent processes can only interact through message sends and receives. It thus excludes race conditions of shared-memory systems by design and in practice also reduces the risks of deadlock. These guarantees are paid for by the added overhead of communication: data has to be copied between actors when sent in a message. This would rule out the Erlang style in systems that pass large amounts of state between actors.

Despite the initial success of Erlang in certain domains, the language is still not being as widely adopted as other concurrent, object-oriented languages, such as Java.[1] In contrast, programming models based on actors or agents are becoming more and more popular, with implementations being developed as part of both new languages, such as Clojure [49], and libraries for mainstream languages, such as Microsoft's Asynchronous Agents Library [33] for C++. However, there are several remaining challenges that must be addressed to make actor-based programming systems a viable solution for concurrent programming on a large scale. In this thesis we focus on what we believe are two of the most important problems:

1. *Implementations of the actor model on mainstream platforms that are efficient and flexible.* The standard concurrency constructs of platforms such as the Java virtual machine (JVM), shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control [125, 35]. The challenge is to provide actor implementations with the efficiency of event-driven run-time systems while avoiding this control inversion.

   Moreover, in practice it is important that actors integrate with existing synchronization mechanisms. For instance, in a JVM-based setting it is necessary to provide a safe way to interact with existing thread-based code that uses locks and monitors for synchronization.

2. *Safe and efficient message passing between local and remote actors.* To enable seamless scalability of applications from multi-core processors to distributed systems, local and remote message send operations should behave

[1] Given its age, it is surprising that in the TIOBE Programming Community Index of July 2010, Scala is already more popular than Erlang. See `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

the same. A good candidate for a uniform semantics is that a sent message gets moved from the memory region of the sender to the (possibly disjoint) memory region of the receiver. This means that the sender loses access to a message after it has been sent. Using such a semantics even inside the same shared-memory (virtual) machine has the advantage that it avoids data races when accessing heap objects, provided concurrent processes communicate only by passing messages.

However, physically moving messages through marshaling (*i.e.*, copying) is expensive. In performance-critical code where messages can be large, such as network protocol stacks [48, 50] or image-processing pipelines, the overhead of copying the state of messages is not acceptable. Instead, the underlying implementation must pass messages between processes running inside the same address space (or virtual machine) by reference. As a result, enforcing race freedom becomes much more difficult, especially in the context of imperative, object-oriented languages, where aliasing is common.

This thesis describes a practical approach to race-free concurrent programming with actors that relies on integrating threads and events, and a lightweight type system that tracks uniqueness of object references. Our approach builds on features that have been adopted in widely-available languages, such as Scala and F#, namely first-class functions, pattern matching, and type system plug-ins.

Establishing this thesis required us to advance the state of the art in implementing concurrent programming models, and in static type systems. In the following we summarize the specific contributions we make in each of these areas.

1.1 Contributions

1.1.1 Design and implementation of programming models for concurrency

We present the design and implementation of an actor-based programming system that is efficient and flexible. The system is efficient thanks to a lightweight, event-driven execution model that can leverage work-stealing thread pools. Experimental results show that our system outperforms state-of-the-art actor implementations in important scenarios. The programming model is more flexible than previous designs by combining the following properties in the same system:

- Event-driven systems can be programmed without an inversion of control. In conventional event-driven designs, the program logic is fragmented across several event handlers; control flow is expressed through manipulation of shared state [26]. Our design avoids this control inversion.

- Incoming messages can be filtered in a fine-grained way using expressive primitives for message reception. This allows expressing common message-passing protocols in a direct and intuitive way [8].

- Event-driven code can interact safely with thread-based, blocking code. The behavior of a single actor can be expressed using both event-driven and thread-based code.

We provide a complete implementation of our programming model in the *Scala Actors* library, which is part of the Scala distribution [83]. It requires neither special syntax nor compiler support. The main advantage of a library-based design is that it is easy to extend and adapt. Apart from lowering the implementation effort, it also helps make the system future proof by enabling non-trivial extensions.

We show how to extend our programming system with a high-level synchronization construct inspired by the join-calculus [56, 57]. Our implementation technique is novel in the way it integrates with Scala's standard pattern matching; this allows programmers to avoid certain kinds of boilerplate code that are inevitable when using existing library-based approaches. We provide a complete prototype implementation that supports join patterns with multiple synchronous events and a restricted form of guards [63].

1.1.2 Static type systems

We introduce a type system that uses capabilities for enforcing both a flexible notion of uniqueness and at-most-once consumption of unique references, making the system uniform and simple. The type system supports methods that operate on unique objects without consuming them in the caller's context. This is akin to *lent* or *borrowed* parameters in ownership type systems [94, 31, 136], which allow temporary aliasing across method boundaries. Our approach identifies uniqueness and borrowing as much as possible. In fact, the only difference between a unique and a borrowed object is that the unique object comes with the capability to consume it (*e.g.*, through ownership transfer). While uniform treatments of uniqueness and borrowing exist [51, 19], our approach requires only simple, unstructured capabilities. This has several advantages: first, it provides simple foundations for uniqueness and borrowing. Second, it does not require complex features such as existential ownership or explicit regions in the type system. Third, it avoids the problematic interplay between borrowing and destructive reads, since unique references subsume borrowed references. The specific contributions of our approach are as follows.

1. We introduce a simple and flexible annotation system used to guide the type checker. The system is simple in the sense that only local variables, fields

and method parameters are annotated. This means that type declarations remain unchanged. This facilitates the integration of our annotation system into full-featured languages, such as Scala.

2. We formalize our type system in the context of an imperative object calculus and prove it sound. Our main point of innovation is a novel way to support internal aliasing of unique references, which is surprisingly simple. By protecting all aliases pointing into a unique object (graph) with the same capability, illegal aliases are avoided by consuming that capability. The formal model corresponds closely to our annotation system: all types in the formalization can be expressed using those annotations. We also extend our system with constructs for actor-based concurrency and prove an isolation theorem.

3. We extend our system to support closures and nested classes, features that have been almost completely ignored by existing work on unique object references. However, we found these features to be indispensable for type-checking real-world Scala code, such as collection classes.

4. We have implemented our type system as a pluggable annotation checker for the EPFL Scala compiler. We show that real-world actor-based concurrent programs can be type-checked with only a small increase in type annotations.

1.1.3 Publications

Parts of the above contributions have been published in the following papers. At the beginning of each chapter we clarify more precisely its relationship to the corresponding publication(s).

- Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010

- Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009

- Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08)*, pages 135–152. Springer, June 2008

- Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proceedings of the 7th Joint Modular Languages Conference (JMLC'06)*, pages 4–22. Springer, September 2006

1.2 Outline of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2 we introduce the Scala Actors library, which provides an embedded domain-specific language for programming with actors in Scala. This chapter explains our approach to integrating threads and events, and provides experimental evidence that our implementation is indeed practical. Chapter 3 presents a novel implementation of join patterns based on Scala's support for extensible pattern matching; we also show how to integrate joins into Scala Actors. Chapter 4 introduces a novel type-based approach to actor isolation. We present a formalization of our type system in the context of an imperative object calculus. The formal development is used to establish soundness of the type system (a complete proof appears in Appendix A.) This chapter also includes an isolation theorem that guarantees race freedom in concurrent programs (a proof of this theorem appears in the appendix.) Finally, we report on our implementation in Scala and practical experience with mutable collections and mid-sized concurrent programs. Chapter 5 concludes this dissertation.

Chapter 2

Integrating Threads and Events

In Chapter 1 we introduced the Erlang programming language [8] as a popular implementation of actor-style concurrency. An important factor of Erlang's success (at least in the domain of telecommunications software [95]) is its lightweight implementation of concurrent processes [7]. Mainstream platforms, such as the JVM [90], have been lacking an equally attractive implementation. Their standard concurrency constructs, shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control [125, 35].

In this chapter we introduce a programming model for Erlang-style actors that unifies thread-based and event-based models of concurrency. The two models are supported through two different operations for message reception. The first operation, `receive`, corresponds to thread-based programming: when the actor cannot receive a message, it suspends keeping the entire call stack of its underlying thread intact. Subsequently, the actor can be resumed just like a regular blocked thread. The second operation, `react`, corresponds to event-based programming: here, the actor suspends using only a *continuation closure*; the closure plays the same role as an event handler in event-driven designs. An actor suspended in this way is resumed by scheduling its continuation closure for execution on a thread pool. By allowing actors to use both `receive` and `react` for implementing their behavior, we combine the benefits of the respective concurrency models. Threads support blocking operations such as system I/O, and can be executed on multiple processor cores in parallel. Event-based computation, on the other hand, is more lightweight and scales to larger numbers of actors. We also present a set of combinators that allows a flexible composition of these actors.

The presented scheme has been implemented in the *Scala Actors* library.[1] It requires neither special syntax nor compiler support. A library-based implementation has the advantage that it can be flexibly extended and adapted to new needs. In fact, the presented implementation is the result of several previous iterations. However, to be easy to use, the library draws on several of Scala's advanced abstraction capabilities; notably partial functions and pattern matching [47].

The rest of this chapter is organized as follows. Section 2.1 introduces our actor-based programming model and explains how it can be implemented as a Scala library. In Section 2.2 we present an extension of our programming model that allows us to unify thread-based and event-based models of concurrency under a single abstraction of actors. We also provide an overview and important details of our implementation. Section 2.3 illustrates the core primitives of Scala Actors using larger examples. Section 2.4 introduces channels for type-safe and private communication. By means of a case study we show in Section 2.5 how our unified programming model can be applied to programming advanced web applications. Experimental results are presented in Section 2.6. Section 2.7 discusses related work on implementing concurrent processes, and actors in particular. Our main concerns are efficiency, the particular programming model, and the approach taken to integrate with the concurrency model of the underlying platform (if any).

This chapter is based on a paper published in Theor. Computer Science [68]. A preliminary version of the paper appears in the proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION 2007) [67]. The paper was written by the author of this thesis, except for parts of this introduction and parts of Sections 2.1 and 2.3, which were contributed by Martin Odersky. We also acknowledge the anonymous reviewers for their helpful feedback.

2.1 The Scala Actors Library

In the following, we introduce the fundamental concepts underlying our programming model and explain how various constructs are implemented in Scala. The implementation of message reception is explained in Section 2.1.1. Section 2.1.2 shows how first-class message handlers support the extension of actors with new behavior.

Actors The Scala Actors library provides a concurrent programming model based on *actors*. An actor [76, 3] is a concurrent process that communicates with other actors by exchanging messages. Communication is *asynchronous*; messages are

[1] Available as part of the Scala distribution [83].

buffered in an actor's *mailbox*. An actor may respond to an asynchronous message by creating new actors, sending messages to known actors (including itself), or changing its behavior. The behavior specifies how the actor responds to the next message that it receives.

Actors in Scala Our implementation of actors in Scala adopts the basic communication primitives virtually unchanged from Erlang [8]. The expression `a ! msg` sends message `msg` to actor `a` (asynchronously). The receive operation has the following form:

```
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```

The first message which matches any of the patterns $msgpat_i$ is removed from the mailbox, and the corresponding $action_i$ is executed (see Figure 2.1 for an example of a message pattern). If no pattern matches, the actor suspends.

New actors can be created in two ways. In the first alternative, we define a new class that extends the `Actor` trait.[2] The actor's behavior is defined by its `act` method. For example, an actor executing `body` can be created as follows:

```
class MyActor extends Actor {
  def act() { body }
}
```

Note that after creating an instance of the `MyActor` class the actor has to be started by calling its `start` method. The second alternative for creating an actor is as follows. The expression `actor {body}` creates a new actor which runs the code in `body`. Inside `body`, the expression `self` is used to refer to the currently executing actor. This "inline" definition of an actor is often more concise than defining a new class. Finally, we note that every Java thread is also an actor, so even the main thread can execute `receive`.[3]

The example in Figure 2.1 demonstrates the usage of all constructs introduced so far. First, we define an `orderMngr` actor that tries to receive messages inside an infinite loop. The `receive` operation waits for two kinds of messages. The `Order(s, item)` message handles an order for `item`. An object which represents the order is created and an acknowledgment containing a reference to the order

[2] A trait in Scala is an abstract class that can be mixin-composed with other traits. [99]

[3] Using `self` outside of an actor definition creates a dynamic proxy object which provides an actor identity to the current thread, thereby making it capable of receiving messages from other actors.

```
// base version
val orderMngr = actor {
  while (true) receive {
    case Order(s, item) =>
      val o =
        handleOrder(s, item)
      s ! Ack(o)
    case Cancel(s, o) =>
      if (o.pending) {
        cancelOrder(o)
        s ! Ack(o)
      } else s ! NoAck
    case x => junk += x
  }
}
val customer = actor {
  orderMngr ! Order(self, it)
  receive {
    case Ack(o) => ...
  }
}
```

```
// version with reply and !?
val orderMngr = actor {
  while (true) receive {
    case Order(item) =>
      val o =
        handleOrder(sender, item)
      reply(Ack(o))
    case Cancel(o) =>
      if (o.pending) {
        cancelOrder(o)
        reply(Ack(o))
      } else reply(NoAck)
    case x => junk += x
  }
}
val customer = actor {
  orderMngr !? Order(it) match {
    case Ack(o) => ...
  }
}
```

Figure 2.1: Example: orders and cancellations

object is sent back to the sender `s`. The `Cancel(s, o)` message cancels order `o` if it is still pending. In this case, an acknowledgment is sent back to the sender. Otherwise a `NoAck` message is sent, signaling the cancellation of a non-pending order.

The last pattern `x` in the `receive` of `orderMngr` is a variable pattern which matches any message. Variable patterns allow to remove messages from the mailbox that are normally not understood ("junk"). We also define a customer actor which places an order and waits for the acknowledgment of the order manager before proceeding. Since spawning an actor (using `actor`) is asynchronous, the defined actors are executed concurrently.

Note that in the above example we have to do some repetitive work to implement request/reply-style communication. In particular, the sender is explicitly included in every message. As this is a frequently recurring pattern, our library has special support for it. Messages always carry the identity of the sender with them. This enables the following additional operations:

- `a !? msg` sends `msg` to `a`, waits for a reply and returns it.

- `sender` refers to the actor that sent the message that was last received by `self`.
- `reply(msg)` replies with `msg` to `sender`.
- `a forward msg` sends `msg` to `a`, using the current `sender` instead of `self` as the sender identity.

With these additions, the example can be simplified as shown on the right-hand side of Figure 2.1. In addition to the operations above, an actor may explicitly designate another actor as the reply destination of a message send. The expression `a.send(msg, b)` sends `msg` to `a` where actor `b` is the reply destination. This means that when `a` receives `msg`, `sender` refers to `b`; therefore, any reply from `a` is sent directly to `b`. This allows certain forwarding patterns to be expressed without creating intermediate actors [140].

Looking at the examples shown above, it might seem that Scala is a language specialized for actor concurrency. In fact, this is not true. Scala only assumes the basic thread model of the underlying host. All higher-level operations shown in the examples are defined as classes and methods of the Scala library. In the following, we look "under the covers" to find out how each construct is defined and implemented. The implementation of concurrent processing is discussed in Section 2.2.3.

The send operation `!` is used to send a message to an actor. The syntax `a ! msg` is simply an abbreviation for the method call `a.!(msg)`, just like `x + y` in Scala is an abbreviation for `x.+(y)`. The `!` method is defined in the `Reactor` trait, which is a super trait of `Actor`:[4]

```
trait Reactor[Msg] {
  val mailbox = new Queue[Msg]
  def !(msg: Msg): Unit = ...
  ...
}
```

The method does two things. First, it enqueues the message argument in the receiving actor's mailbox which is represented as a field of type `Queue[Msg]`, where `Msg` is the type of messages that the actor can receive. Second, if the receiving actor is currently suspended in a `receive` that could handle the sent message, the execution of the actor is resumed. Note that the `Actor` trait extends `Reactor[Any]`. This means an actor created in one of the ways discussed above can receive any type of message. It is also possible to create and start instances of `Reactor` directly. However, `Reactor`s do not support the (thread-based) `receive` operation

[4]For simplicity we omit unimportant implementation details, such as super traits and modifiers.

that we discuss in the following; `Reactor`s can only receive messages using the (event-based) `react` primitive, which we introduce in Section 2.2.2.

The `actor` and `self` constructs are realized as methods defined by the `Actor` *object*. Objects have exactly one instance at runtime, and their methods are similar to static methods in Java.

```
object Actor {
  def self: Actor ...
  def actor(body: => Unit): Actor ...
  ...
}
```

Note that Scala has different namespaces for types and terms. For instance, the name `Actor` is used both for the object above (a term) and the trait which is the result type of `self` and `actor` (a type). In the definition of the `actor` method, the argument `body` defines the behavior of the newly created actor. It is a closure returning the unit value. The leading `=>` in its type indicates that it is passed by name.

2.1.1 The receive operation

The `receive { ... }` construct is particularly interesting. In Scala, the pattern matching expression inside braces is treated as a first-class object that is passed as an argument to the `receive` method. The argument's type is an instance of `PartialFunction`, which is a subclass of `Function1`, the class of unary functions. The two classes are defined as follows.

```
abstract class Function1[-A, +B] {
  def apply(x: A): B
}
abstract class PartialFunction[-A, +B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

Functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which tests whether the function is defined for a given argument. Both classes are parametrized; the first type parameter `A` indicates the function's argument type and the second type parameter `B` indicates its result type.[5]

[5] Parameters can carry `+` or `-` variance annotations which specify the relationship between instantiation and subtyping. The `-A, +B` annotations indicate that functions are contravariant in their argument and covariant in their result. In other words `Function1[X1, Y1]` is a subtype of `Function1[X2, Y2]` if `X2` is a subtype of `X1` and `Y1` is a subtype of `Y2`.

A pattern matching expression `{ case` p_1 `=>` e_1`; ...; case` p_n `=>` e_n `}` is then a partial function whose methods are defined as follows.

- The `isDefinedAt` method returns `true` if one of the patterns p_i matches the argument, `false` otherwise.
- The `apply` method returns the value e_i for the first pattern p_i that matches its argument. If none of the patterns match, a `MatchError` exception is thrown.

The `receive` construct is realized as a method (of the `Actor` trait) that takes a partial function as an argument.

```
def receive[R](f: PartialFunction[Any, R]): R
```

The implementation of `receive` proceeds roughly as follows. First, messages in the mailbox are scanned in the order they appear. If `receive`'s argument `f` is defined for a message, that message is removed from the mailbox and `f` is applied to it. On the other hand, if `f.isDefinedAt(m)` is `false` for every message `m` in the mailbox, the receiving actor is suspended.

There is also some other functionality in Scala's actor library which we have not covered. For instance, there is a method `receiveWithin` which can be used to specify a time span in which a message should be received allowing an actor to timeout while waiting for a message. Upon timeout the action associated with a special `TIMEOUT` pattern is fired. Timeouts can be used to suspend an actor, completely flush the mailbox, or to implement priority messages [8].

2.1.2 Extending actor behavior

The fact that message handlers are first-class partial function values can be used to make actors extensible with new behaviors. A general way to do this is to have classes provide actor behavior using methods, so that subclasses can override them.

Figure 2.2 shows an example. The `Buffer` class extends the `Actor` trait to define actors that implement bounded buffers containing at most `N` integers. We omit a discussion of the array-based implementation (using the `buf` array and a number of integer variables) since it is completely standard; instead, we focus on the actor-specific parts. First, consider the definition of the `act` method. Inside an infinite loop it invokes `receive` passing the result of the `reaction` method. This method returns a partial function that defines actions associated with the `Put(x)` and `Get` message patterns. As a result, instances of the `Buffer` class are actors that repeatedly wait for `Put` or `Get` messages.

Assume we want to extend the behavior of buffer actors, so that they also respond to `Get2` messages, thereby removing two elements at once from the buffer.

```
class Buffer(N: Int) extends Actor {
  val buf = new Array[Int](N)
  var in = 0; var out = 0; var n = 0
  def reaction: PartialFunction[Any, Unit] = {
    case Put(x) if n < N =>
      buf(in) = x; in = (in + 1) % N; n = n + 1; reply()
    case Get if n > 0 =>
      val r = buf(out); out = (out + 1) % N; n = n - 1; reply(r)
  }
  def act(): Unit = while (true) receive(reaction)
}
class Buffer2(N: Int) extends Buffer(N) {
  override def reaction: PartialFunction[Any, Unit] =
    super.reaction orElse {
      case Get2 if n > 1 =>
        out = (out + 2) % N; n = n - 2
        reply (buf(out-2), buf(out-1))
    }
}
```

Figure 2.2: Extending actors with new behavior

The `Buffer2` class below shows such an extension. It extends the `Buffer` class, thereby overriding its `reaction` method. The new method returns a partial function which combines the behavior of the superclass with a new action associated with the `Get2` message pattern. Using the `orElse` combinator we obtain a partial function that is defined as `super.reaction` except that it is additionally defined for `Get2`. The definition of the `act` method is inherited from the superclass which results in the desired overall behavior.

2.2 Unified Actor Model and Implementation

Traditionally, programming models for concurrent processes are either thread-based or event-based. We review their complementary strengths and weaknesses in Section 2.2.1. Scala Actors unify both programming models, allowing programmers to trade efficiency for flexibility in a fine-grained way. We present our unified, actor-based programming model in Section 2.2.2. Section 2.2.3 provides an overview as well as important details of the implementation of the Scala Actors library. Finally, Section 2.2.4 introduces a set of combinators that allows one to

compose actors in a modular way.

2.2.1 Threads vs. events

Concurrent processes such as actors can be implemented using one of two implementation strategies:

- Thread-based implementation: The behavior of a concurrent process is defined by implementing a thread-specific method. The execution state is maintained by an associated thread stack (see, *e.g.*, [85]).

- Event-based implementation: The behavior is defined by a number of (non-nested) event handlers which are called from inside an event loop. The execution state of a concurrent process is maintained by an associated record or object (see, *e.g.*, [134]).

Often, the two implementation strategies imply different programming models. Thread-based models are usually easier to use, but less efficient (context switches, memory consumption) [102], whereas event-based models are usually more efficient, but very difficult to use in large designs [125].

Most event-based models introduce an *inversion of control*. Instead of calling blocking operations (*e.g.*, for obtaining user input), a program merely registers its interest to be resumed on certain *events* (*e.g.*, signaling a pressed button). In the process, *event handlers* are installed in the execution environment. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is "inverted". Because of inversion of control, switching from a thread-based to an event-based model normally requires a global re-write of the program [26, 35].

2.2.2 Unified actor model

The main idea of our programming model is to allow an actor to wait for a message using two different operations, called `receive` and `react`, respectively. Both operations try to remove a message from the current actor's mailbox given a partial function that specifies a set of message patterns (see Section 2.1). However, the semantics of `receive` corresponds to thread-based programming, whereas the semantics of `react` corresponds to event-based programming. In the following we discuss the semantics of each operation in more detail.

The `receive` operation

The `receive` operation has the following type:

```
def receive[R](f: PartialFunction[Any, R]): R
```

If there is a message in the current actor's mailbox that matches one of the cases specified in the partial function `f`, the result of applying `f` to that message is returned. Otherwise, the current thread is suspended; this allows the receiving actor to resume execution normally when receiving a matching message. Note that `receive` retains the complete call stack of the receiving actor; the actor's behavior is therefore a sequential program which corresponds to thread-based programming.

The `react` operation

The `react` operation has the following type:

```
def react(f: PartialFunction[Any, Unit]): Nothing
```

Note that `react` has return type `Nothing`. In Scala's type system a method that never returns normally has return type `Nothing`. This means that the action specified in `f` that corresponds to the matching message is the last code that the current actor executes. The semantics of `react` closely resembles event-based programming: the current actor registers the partial function `f` which corresponds to a set of event handlers, and then releases the underlying thread. When receiving a matching message the actor's execution is resumed by invoking the registered partial function. In other words, when using `react`, the argument partial function has to contain the rest of the current actor's computation (its *continuation*) since calling `react` never returns. In Section 2.2.4 we introduce a set of combinators that hide these explicit continuations.

2.2.3 Implementation

Before discussing the implementation it is useful to clarify some terminology. In this section we refer to an actor that is unable to continue (*e.g.*, because it is waiting for a message) as being *suspended*. Note that this notion is independent of a specific concurrency model, such as threads. However, it is often necessary to indicate whether an actor is suspended in an event-based or in a thread-based way. We refer to an actor that is suspended in a `react` as being *detached* (since in this case the actor is detached from any other thread). In contrast, an actor that is suspended in a `receive` is called *blocked* (since in this case the underlying worker thread is blocked). More generally, we use the term blocking as a shortcut for *thread-blocking*.

Implementation Overview

In our framework, multiple actors are executed on multiple threads for two reasons:

1. Executing concurrent code in parallel may result in speed-ups on multi-processors and multi-core processors.

2. Executing two interacting actors on different threads allows actors to invoke blocking operations without affecting the progress of other actors.

Certain operations provided by our library introduce concurrency, namely spawning an actor using `actor`, and asynchronously sending a message using the `!` operator. We call these operations *asynchronous operations*. Depending on the current load of the system, asynchronous operations may be executed in parallel. Invoking an asynchronous operation creates a task that is submitted to a thread pool for execution. More specifically, a task is generated in the following three cases:

1. Spawning a new actor using `actor {body}` generates a task that executes `body`.

2. Sending a message to an actor suspended in a `react` that enables it to continue generates a task that processes the message.

3. Calling `react` where a message can be immediately removed from the mailbox generates a task that processes the message.

The basic idea of our implementation is to use a thread pool to execute actors, and to *resize* the thread pool whenever it is necessary to support blocking thread operations. If actors use only operations of the event-based model, the size of the thread pool can be fixed. This is different if some of the actors use blocking operations such as `receive` or system I/O. In the case where every worker thread is occupied by a blocked actor and there are pending tasks, the thread pool has to grow.

For example, consider a thread pool with a single worker thread, executing a single actor a. Assume a first spawns a new actor b, and then waits to receive a message from b using the thread-based receive operation. Spawning b creates a new task that is submitted to the thread pool for execution. Execution of the new task is delayed until a releases the worker thread. However, when a suspends, the worker thread is *blocked*, thereby leaving the task unprocessed indefinitely. Consequently, a is never resumed since the only task that could resume it (by sending it a message) is never executed. The system is deadlocked.

In our library, system-induced deadlocks are avoided by increasing the size of the thread pool whenever necessary. It is necessary to add another worker thread whenever there is a pending task and all worker threads are blocked. In this case, the pending task(s) are the only computations that could possibly unblock any of the worker threads (*e.g.*, by sending a message to a suspended actor). To do this, our system can use one of several alternative mechanisms. In the most flexible alternative, a scheduler thread (which is separate from the worker threads of the thread pool) periodically checks whether the number of worker threads that are *not* blocked is smaller than the number of available processors. In that case, a new worker thread is added to the thread pool that processes any remaining tasks.

Implementation Details

A detached actor (*i.e.*, suspended in a `react` call) is not represented by a blocked thread but by a closure that captures the actor's continuation. This closure is executed once a message is sent to the actor that matches one of the message patterns specified in the `react`. When an actor detaches, its continuation closure is stored in the `waitingFor` field of the `Actor` trait:[6]

```
trait Actor {
  val mailbox = new Queue[Any]
  var waitingFor: PartialFunction[Any, Unit]
  def !(msg: Any): Unit = ...
  def react(f: PartialFunction[Any, Unit]): Nothing = ...
  ...
}
```

An actor's continuation is represented as a partial function of type `PartialFunction[Any, Unit]`. When invoking an actor's continuation we pass the message that enables the actor to resume as an argument. The idea is that an actor only detaches when `react` fails to remove a matching message from the mailbox. This means that a detached actor is always resumed by sending it a message that it is waiting for. This message is passed when invoking the continuation. We represent the continuation as a *partial function* rather than a function to be able to test whether a message that is sent to an actor enables it to continue. This is explained in more detail below.

The `react` method saves the continuation closure whenever the receiving actor has to suspend (and therefore detaches):

[6] To keep the explanation of the basic concurrency mechanisms as simple as possible, we ignore the fact that in our actual implementation the `Actor` trait has several super traits.

```
def react(f: PartialFunction[Any, Unit]): Nothing =
  synchronized {
    mailbox.dequeueFirst(f.isDefinedAt) match {
      case Some(msg) =>
        schedule(new Task({ () => f(msg) }))
      case None =>
        waitingFor = f
        isDetached = true
    }
    throw new SuspendActorException
  }
```

Recall that a partial function, such as `f`, is usually represented as a block with a list of patterns and associated actions. If a message can be removed from the mailbox (tested using `dequeueFirst`) the action associated with the matching pattern is scheduled for execution by calling the `schedule` operation. It is passed a task which contains a delayed computation that applies `f` to the received message, thereby executing the associated action. Tasks and the `schedule` operation are discussed in more detail below.

If no message can be removed from the mailbox, we save `f` as the continuation of the receiving actor in the `waitingFor` field. Since `f` contains the complete execution state we can resume the execution at a later point when a matching message is sent to the actor. The instance variable `isDetached` is used to tell whether the actor is detached (as opposed to blocked in a `receive`). If it is, the value stored in the `waitingFor` field is a valid execution state.

Finally, by throwing a special exception, control is transferred to the point in the control flow where the current actor was started or resumed. Since actors are always executed as part of tasks, the `SuspendActorException` is only caught inside task bodies.

Tasks are represented as instances of the following class (simplified):

```
class Task(cont: () => Unit) {
  def run() {
    try { cont() } // invoke continuation
    catch { case _: SuspendActorException =>
      // do nothing }
  } }
```

The constructor of the `Task` class takes a continuation of type `() => Unit` as its single argument. The class has a single `run` method that wraps an invocation of the continuation in an exception handler. The exception handler catches exceptions of type `SuspendActorException` which are thrown whenever an actor detaches. The body of the exception handler is empty since the necessary bookkeeping, such

as saving the actor's continuation, has already been done at the point where the exception was thrown.

Sending a message to an actor involves checking whether the actor is waiting for the message, and, if so, resuming the actor according to the way in which it suspended (*i.e.*, using `receive` or `react`):

```
def !(msg: Any): Unit = synchronized {
  if (waitingFor(msg)) {
    val savedWaitingFor = waitingFor
    waitingFor = Actor.waitingForNone
    if (isDetached) {
      isDetached = false
      schedule(new Task({ () => savedWaitingFor(msg) }))
    } else
      resume() // thread-based resume
  } else mailbox += msg
}
```

When sending a message to an actor that it does not wait for (*i.e.*, the actor is not suspended or its continuation is not defined for the message), the message is simply enqueued in the actor's mailbox. Otherwise, the internal state of the actor is changed to reflect the fact that it is no longer waiting for a message (`Actor.waitingForNone` is a partial function that is not defined for any argument). Then, we test whether the actor is detached; in this case we schedule a new task that applies the actor's continuation to the newly received message. The continuation was saved when the actor detached the last time. If the actor is not detached (which means it is blocked in a `receive`), it is resumed by notifying its underlying blocked thread.

Spawning an actor using `actor {body}` generates a task that executes `body` as part of a new actor:

```
def actor(body: => Unit): Actor = {
  val a = new Actor {
    def act() = body
  }
  schedule(new Task({ () => a.act() }))
  a
}
```

The `actor` function takes a delayed expression (indicated by the leading `=>`) of type `Unit` as its single argument. After instantiating a new `Actor` with the given `body`, we create a new task that is passed a continuation that simply executes the actor's body. Note that the actor may detach later on (*e.g.*, by waiting in a `react`),

in which case execution of the task is finished early, and the rest of the actor's body is run as part of a new continuation which is created when the actor is resumed subsequently.

The `schedule` operation submits tasks to a thread pool for execution. A simple implementation strategy would be to put new tasks into a global queue that all worker threads in the pool access. However, we found that a global task queue becomes a serious bottle neck when a program creates short tasks with high frequency (especially if such a program is executed on multiple hardware threads). To remove this bottle neck, each worker thread has its own local task queue. When a worker thread generates a new task, *e.g.*, when a message send enables the receiver to continue, the (sending) worker puts it into its local queue. This means that a receiving actor is *often* executed on the same thread as the sender. This is not always the case, because *work stealing* balances the work load on multiple worker threads (which ultimately leads to parallel execution of tasks) [16]. This means that idle worker threads with empty task queues look into the queues of other workers for tasks to execute. However, accessing the local task queue is much faster than accessing the global task queue thanks to sophisticated non-blocking algorithms [86]. In our framework the global task queue is used to allow non-worker threads (any JVM thread) to invoke asynchronous operations.

As discussed before, our thread pool has to grow whenever there is a pending task and all worker threads are blocked. Our implementation provides two different mechanisms for avoiding pool lock ups in the presence of blocking operations. The mechanisms mainly differ in the kinds of blocking operations they support.

The first mechanism uses an auxiliary *scheduler thread* that periodically determines the number of blocked worker threads. If the number of workers that are *not* blocked is smaller than the number of available processors, additional workers are started to process any remaining tasks. Aside from avoiding pool lock ups, this mechanism can also improve CPU utilization on multi-core processors.

The second mechanism is based on the so-called *managed blocking* feature provided by Doug Lea's fork/join pool implementation for Java ([86] discusses a predecessor of that framework). Managed blocking enables the thread pool to control the invocation of blocking operations. Actors are no longer allowed to invoke arbitrary blocking operations directly. Instead, they are only permitted to directly invoke blocking operations defined for actors, such as `receive`, `receiveWithin`, or `!?`. Any other blocking operation must be invoked indirectly through a method of the thread pool that expects an instance of the following `ManagedBlocker` trait. The blocking operations provided by the actors library are implemented in terms of that trait (see below).

```
trait ManagedBlocker {
  def block(): Boolean
  def isReleasable: Boolean
}
```

An instance of `ManagedBlocker` allows invoking the blocking operation via its `block` method. Furthermore, using the `isReleasable` method one can query whether the blocking operation has already returned. This enables the thread pool to delay the invocation of a blocking operation until it is safe (for instance, after a spare worker thread has been created). In addition, the result of invoking `isReleasable` indicates to the thread pool if it is safe to terminate the temporary spare worker that might have been created to support the corresponding blocking operation.

More specifically, the two methods are implemented in the following way. The `block` method invokes a method that (possibly) blocks the current thread. The underlying thread pool makes sure to invoke `block` only in a context where blocking is safe; for instance, if there are no idle worker threads left, it first creates an additional thread that can process submitted tasks in the case all other workers are blocked. The Boolean result indicates whether the current thread might still have to block even after the invocation of `block` has returned. None of the blocking operations defined for actors require blocking after `block` returns; therefore, it is sufficient to just return `true`, which indicates that no additional blocking is necessary. The `isReleasable` method, like `block`, indicates whether additional blocking is necessary. Unlike `block`, it should not invoke possibly blocking operations itself. Moreover, it can (and should) return `true` even if a previous invocation of `block` returned `false`, but blocking is no longer necessary.

Figure 2.3 shows the `Blocker` class (simplified) which is used by the blocking `receive` operation to safely block the current worker thread (we omit unimportant parts of the code, including visibility modifiers of methods).

2.2.4 Composing actor behavior

Without extending the unified actor model, defining an actor that executes several given functions in sequence is not possible in a modular way.

For example, consider the two methods below:

```
def awaitPing = react { case Ping => }
def sendPong = sender ! Pong
```

It is not possible to sequentially compose `awaitPing` and `sendPong` as follows:

```
actor { awaitPing; sendPong }
```

```
trait Actor extends ... {
  // ...

  class Blocker extends ManagedBlocker {
    def block() = {
      Actor.this.suspendActor()
      true
    }
    def isReleasable =
      !Actor.this.isSuspended
  }

  def suspendActor() = synchronized {
    while (isSuspended) {
      try {
        wait()
      } catch {
        case _: InterruptedException =>
      }
    }
  }
}
```

Figure 2.3: Extending the `ManagedBlocker` trait for implementing blocking actor operations

Since `awaitPing` ends in a call to `react` which never returns, `sendPong` would never get executed. One way to work around this restriction is to place the continuation into the body of `awaitPing`:

```
def awaitPing = react { case Ping => sendPong }
```

However, this violates modularity. Instead, our library provides an `andThen` combinator that allows actor behavior to be composed sequentially. Using `andThen`, the body of the above actor can be expressed as follows:

```
awaitPing andThen sendPong
```

`andThen` is implemented by installing a hook function in the actor. This function is called whenever the actor terminates its execution. Instead of exiting, the code of the second body is executed. Saving and restoring the previous hook function permits chained applications of `andThen`.

```
class InOrder(n: IntTree) extends Producer[Int] {
  def produceValues() {
    traverse(n)
  }
  def traverse(n: IntTree) {
    if (n != null) {
      traverse(n.left)
      produce(n.elem)
      traverse(n.right)
    }
  }
}
```

Figure 2.4: Producer that generates all values in a tree in in-order

The `Actor` object also provides a `loop` combinator. It is implemented in terms of `andThen`:

```
def loop(body: => Unit) = body andThen loop(body)
```

Hence, the body of `loop` can end in an invocation of `react`. Similarly, we can define a `loopWhile` combinator that terminates the actor when a provided guard evaluates to `false`.

2.3 Examples

In this section we discuss two larger examples. These examples serve two purposes. First, they show how our unified programming model can be used to make parts of a threaded program event-based with minimal changes to an initial actor-based program. Second, they demonstrate the use of the combinators introduced in Section 2.2.4 to turn a complex program using non-blocking I/O into a purely event-driven program while maintaining a clear threaded code structure.

2.3.1 Producers and iteration

In the first example, we are going to write an abstraction of *producers* that provide a standard iterator interface to retrieve a sequence of produced values. Producers are defined by implementing an abstract `produceValues` method that calls a `produce` method to generate individual values. Both methods are inherited from a `Producer` class. For example, Figure 2.4 shows the definition of a producer that generates the values contained in a tree in in-order.

```
class Producer[T] {
  def produce(x: T) {
    coordinator ! Some(x)
  }
  val producer = actor {
    produceValues
    coordinator ! None
  }
  ...
}
```

```
val coordinator = actor {
  while (true) receive {
    case Next => receive {
      case x: Option[_] =>
        reply(x)
    }
  }
}
```

Figure 2.5: Implementation of the producer and coordinator actors

```
val coordinator = actor {
  loop { react {
    // ... as in Figure 2.5
  }}}
```

Figure 2.6: Implementation of the coordinator actor using `react`

Figure 2.5 shows an implementation of producers in terms of two actors, a *producer* actor, and a *coordinator* actor. The producer runs the `produceValues` method, thereby sending a sequence of values, wrapped in `Some` messages, to the coordinator. The sequence is terminated by a `None` message. The coordinator synchronizes requests from clients and values coming from the producer.

It is possible to economize one thread in the producer implementation. As shown in Figure 2.6, this can be achieved by changing the call to `receive` in the coordinator actor into a call to `react` and using the `loop` combinator instead of the `while` loop. By calling `react` in its outer loop, the coordinator actor allows the scheduler to detach it from its worker thread when waiting for a `Next` message. This is desirable since the time between client requests might be arbitrarily long. By detaching the coordinator, the scheduler can re-use the worker thread and avoid creating a new one.

2.3.2 Pipes and asynchronous I/O

In this example, a pair of processes exchanges data over a FIFO pipe. Such a pipe consists of a sink and a source channel that are used for writing to the pipe and reading from the pipe, respectively. The two processes communicate over the pipe as follows. One process starts out writing some data to the sink while the process

at the other end reads it from the source. Once all of the data has been transmitted, the processes exchange roles and repeat this conversation.

To make this example more realistic and interesting at the same time, we use non-blocking I/O operations. A process that wants to write data has to register its interest in writing together with an event handler; when the I/O subsystem can guarantee that the next write operation will not block (*e.g.*, because of enough buffer space), it invokes this event handler.

The data should be processed concurrently; it is therefore not sufficient to put all the program logic into the event handlers that are registered with the I/O subsystem. Moreover, we assume that a process may issue blocking calls while processing the received data; processing the data inside an event handler could therefore block the entire I/O subsystem, which has to be avoided. Instead, the event handlers have to either notify a thread or an actor, or submit a task to a thread pool for execution.

In the following, we first discuss a solution that uses threads to represent the end points of a pipe. After that, we present an event-based implementation and compare it to the threaded version. Finally, we discuss a solution that uses Scala Actors. The solutions are compared with respect to synchronization and code structure.

We use a number of objects and methods whose definitions are omitted because they are not interesting for our discussion. First, processes have a reference `sink` to an I/O channel. The channel provides a `write` method that writes the contents of a buffer to the channel. The non-blocking I/O API is used as follows. The user implements an event handler which is a class with a single method that executes the I/O operation (and possibly other code). This event handler is registered with an I/O event dispatcher `disp` together with a channel; the dispatcher invokes an event handler when the corresponding (read or write) event occurs on the channel that the handler registered with. Each event handler is only registered until it has been invoked. Therefore, an event handler has to be registered with the dispatcher for each event that it should handle.

Thread-based pipes

In the first solution that we discuss, each end point of a pipe is implemented as a thread. Figure 2.7 shows the essential parts of the implementation. The `run` method of the `Proc` class on the left-hand side shows the body of a process thread. First, we test whether the process should start off writing or reading. The `writeData` and `readData` operations are executed in the according order. After the writing process has written all its data, it has to synchronize with the reading process, so that the processes can safely exchange roles. This is necessary to avoid the situation where both processes have registered a handler for the same kind of

```
class Proc(write: Boolean,
           exch: Barrier)
  extends Thread {
  ...
  override def run() {
    if (write) writeData
    else readData
    exch.await
    if (write) readData
    else writeData
  } }
```

```
def writeData {
  fill(buf)
  disp.register(sink,
                writeHnd)
  var finished = false
  while (!finished) {
    dataReady.await
    dataReady.reset
    if (bytesWritten==32*1024)
      finished = true
    else {
      if (!buf.hasRemaining)
        fill(buf)
      disp.register(sink,
                    writeHnd)
    } } }
val writeHnd = new WriteHandler {
  def handleWrite() {
    bytesWritten +=
      sink.write(buf)
    dataReady.await
  } }
```

Figure 2.7: Thread-based pipes

I/O event. In this case, a process might wait indefinitely for an event because it was dispatched to the other process. We use a simple barrier of size 2 for synchronization: a thread invoking `await` on the `exch` barrier is blocked until a second thread invokes `exch.await`. The `writeData` method is shown on the right-hand side of Figure 2.7 (the `readData` method is analogous). First, it fills a buffer with data using the `fill` method. After that, it registers the `writeHnd` handler for write events on the `sink` with the I/O event dispatcher (`writeHnd` is discussed below). After that, the process enters a loop. First, it waits on the `dataReady` barrier until the write event handler has completed the next write operation. When the thread resumes, it first resets the `dataReady` barrier to the state where it has not been invoked, yet. The thread exits the loop when it has written 32 KB of data. Otherwise, it refills the buffer if it has been completed, and re-registers the event handler for the next write operation. The `writeHnd` event handler implements a single method `handleWrite` that writes data stored in `buf` to the `sink`, thereby counting the number of bytes written. After that, it notifies the concurrently run-

```
class Proc(write: Boolean,
           pool: Executor)
{
  ...
  var last = false
  if (write) writeData
  else readData
  ...
  def writeData {
    fill(buf)
    disp.register(...)
  }
}
```

```
val task = new Runnable {
  def run() {
    if (bytesWritten==32*1024) {
      if (!last) {
        last = true; readData
      }
    } else {
      if (!buf.hasRemaining)
        fill(buf)
      disp.register(sink,
                    writeHnd)
    }
  }
}
val writeHnd = new WriteHandler {
  def handleWrite() {
    bytesWritten +=
      sink.write(buf)
    pool.execute(task)
  }
}
```

Figure 2.8: Event-driven pipes

ning writer thread by invoking `await` on the `dataReady` barrier.

Event-driven pipes

Figure 2.8 shows an event-driven version that is functionally equivalent to the previous threaded program. The process constructor which is the body of the `Proc` class shown on the left-hand side, again, tests whether the process starts out writing or reading. However, based on this test only *one* of the two I/O operations is called. The reason is that each I/O operation, such as `writeData`, registers an event handler with the I/O subsystem, and then returns immediately. The event handler for the second operation may only be installed when the last handler of the previous operation has run. Therefore, we have to decide inside the event handler of the write operation whether we want to read subsequently or not. The `last` field keeps track of this decision across all event handler invocations. If `last` is `false`, we invoke `readData` after `writeData` has finished (and *vice versa*); otherwise, the sequence of I/O operations is finished. The definition of an event handler

for write events is shown on the right-hand side of Figure 2.8 (read events are handled in an analogous manner). As before, the `writeHnd` handler implements the `handleWrite` method that writes data from `buf` to the `sink`, thereby counting the number of bytes written. To do the concurrent processing the handler submits a task to a thread pool for execution. The definition of this task is shown above. Inside the task we first test whether all data has been written; if so, the next I/O operation (in this case, `readData`) is invoked depending on the field `last` that we discussed previously. If the complete contents of `buf` has been written, it is refilled. Finally, the task re-registers the `writeHnd` handler to process the next event.

Compared to thread-based programming, the event-driven style obscures the control flow. For example, consider the `writeData` method. It does some work, and then registers an event handler. However, it is not clear what the operational effect of `writeData` is. Moreover, what happens after `writeData` has finished its actual work? To find out, we have to look inside the code of the registered event handler. This is still not sufficient, since also the submitted task influences the control flow. In summary, the program logic is implicit, and has to be recovered in a tedious way. Moreover, state has to be maintained across event handlers and tasks. In languages that do not support closures this often results in manual stack management [1].

Actor-based pipes

Figure 2.9 shows the same program using Scala Actors. The `Proc` class extends the `Actor` trait; its `act` method specifies the behavior of an end point. The body of the `act` method is similar to the process body of the thread-based version. There are two important differences. First, control flow is specified using the `andThen` combinator. This is necessary since `writeData` (and `readData`) may suspend using `react`. Without using `andThen`, parts of the actor's continuation not included in the argument closure of the suspending `react` would be "lost". Basically, `andThen` appends the closure on its right-hand side to whatever continuation is saved during the execution of the closure on its left-hand side. Second, end point actors exchange messages to synchronize when switching roles from writing to reading (and *vice versa*). The `writeData` method is similar to its thread-based counterpart. The `while` loop is replaced by the `loopWhile` combinator since inside the loop the actor may suspend using `react`. At the beginning of each loop iteration the actor waits for a `Written` message signaling the completion of a write event handler. The number of bytes written is carried inside the message which allows us to make `bytesWritten` a local variable; in the thread-based version it is shared among the event handler and the process. The remainder of `writeData` is the same as in the threaded version. The `writeHnd` handler used in the actor-based

```
class Proc(write: Boolean,
           other: Actor)
  extends Actor {
  ...
  def act() {
    { if (write)
        writeData
      else
        readData
    } andThen {
      other ! Exchange
      react {
        case Exchange =>
          if (write)
            readData
          else
            writeData }
    } }
}
```

```
def writeData {
  fill(buf)
  disp.register(sink,
                writeHnd)
  var bytesWritten = 0
  loopWhile(bytesWritten<32*1024)
    react { case Written(num) =>
      bytesWritten += num
      if (bytesWritten==32*1024)
        exit()
      else {
        if (!buf.hasRemaining)
          fill(buf)
        disp.register(sink,
                      writeHnd)
      } }
}
val writeHnd =
  new WriteHandler {
    def handleWrite() {
      val num =
        sink.write(buf)
      proc!Written(num)
    } }
```

Figure 2.9: Actor-based pipes

program is similar to the thread-based version, except that it notifies its process using an asynchronous message send. Note that, in general, the event handler is run on a thread which is different from the worker threads used by our library to execute actors (the I/O subsystem might use its own thread pool, for example). To make the presented scheme work, it is therefore crucial that arbitrary threads may send messages to actors.

Conclusion Compared to the event-driven program, the actor-based version improves on the code structure in the same way as the thread-based version. Passing result values as part of messages makes synchronization slightly clearer and reduces the number of global variables compared to the thread-based program. However, in Section 2.6 we show that an event-based implementation of a benchmark version of the pipes example is much more efficient and scalable than a

purely thread-based implementation. Our unified actor model allows us to implement the pipes example in a purely event-driven way while maintaining the clear code structure of an equivalent thread-based program.

2.4 Channels and Selective Communication

In the programming model that we have described so far, actors are the only entities that can send and receive messages. Moreover, the receive operation ensures *locality*, *i.e.*, only the owner of the mailbox can receive messages from it. Therefore, race conditions when accessing the mailbox are avoided by design. Types of messages are flexible: they are usually recovered through pattern matching. Ill-typed messages are ignored instead of raising compile-time or run-time errors. In this respect, our library implements a dynamically-typed embedded domain-specific language.

However, to take advantage of Scala's rich static type system, we need a way to permit strongly-typed communication among actors. For this, we use channels which are parameterized with the types of messages that can be sent to and received from it, respectively. Moreover, the visibility of channels can be restricted according to Scala's scoping rules. That way, communication between sub-components of a system can be hidden. We distinguish input channels from output channels. Actors are then treated as a special case of output channels:

```
trait Actor extends OutputChannel[Any] { ... }
```

The possibility for an actor to have multiple input channels raises the need to selectively communicate over these channels. Up until now, we have shown how to use `receive` to remove messages from an actor's mailbox. We have not yet shown how messages can be received from multiple input channels. Instead of adding a new construct, we generalize `receive` to work over multiple channels.

For example, a model of a component of an integrated circuit can receive values from both a control and a data channel using the following syntax:

```
receive {
  case DataCh ! data => ...
  case CtrlCh ! cmd => ...
}
```

2.5 Case Study

In this section we show how our unified actor model addresses some of the challenges of programming web applications. In the process, we review event- and

thread-based solutions to common problems, such as blocking I/O operations. Our goal is then to discuss potential benefits of our unified approach. Advanced web applications typically pose at least the following challenges to the programmer:

- *Blocking operations.* There is almost always some functionality that is implemented using blocking operations. Possible reasons are lack of suitable libraries (*e.g.*, for non-blocking socket I/O), or simply the fact that the application is built on top of a large code base that uses potentially blocking operations in some places. Typically, rewriting infrastructure code to use non-blocking operations is not an option.

- *Non-blocking operations.* On platforms such as the JVM, web application servers often provide some parts (if not all) of their functionality in the form of non-blocking APIs for efficiency. Examples are request handling, and asynchronous HTTP requests.

- *Race-free data structures.* Advanced web applications typically maintain user profiles for personalization. These profiles can be quite complex (some electronic shopping sites apparently track every item that a user visits). Moreover, a single user may be logged in on multiple machines, and issue many requests in parallel. This is common on web sites, such as those of electronic publishers, where single users represent whole organizations. It is therefore mandatory to ensure race-free accesses to a user's profile.

2.5.1 Thread-based approaches

VMs overlap computation and I/O by transparently switching among threads. Therefore, even if loading a user profile from disk blocks, only the current request is delayed. Non-blocking operations can be converted to blocking operations to support a threaded style of programming: after firing off a non-blocking operation, the current thread blocks until it is notified by a completion event. However, threads do not come for free. On most mainstream VMs, the overhead of a large number of threads–including context switching and lock contention–can lead to serious performance degradation [134, 46]. Overuse of threads can be avoided by using bounded thread pools [85]. Shared resources such as user profiles have to be protected using synchronization operations. This is known to be particularly hard using shared-memory locks [87]. We also note that alternatives such as transactional memory [71, 72], even though a clear improvement over locks, do not provide seamless support for I/O operations as of yet. Instead, most approaches require the use of compensation actions to revert the effects of I/O operations, which further complicate the code.

2.5.2 Event-based approaches

In an event-based model, the web application server generates events (network and I/O readiness, completion notifications etc.) that are processed by event handlers. A small number of threads (typically one per CPU) loop continuously removing events from a queue and dispatching them to registered handlers. Event handlers are required not to block since otherwise the event-dispatch loop could be blocked, which would freeze the whole application. Therefore, all operations that could potentially block, such as the user profile look-up, have to be transformed into non-blocking versions. Usually, this means executing them on a newly spawned thread, or on a thread pool, and installing an event handler that gets called when the operation completed [103]. Usually, this style of programming entails an inversion of control that causes the code to loose its structure and maintainability [26, 35].

2.5.3 Scala Actors

In our unified model, event-driven code can easily be wrapped to provide a more convenient interface that avoids inversion of control without spending an extra thread [66]. The basic idea is to decouple the thread that signals an event from the thread that handles it by sending a message that is buffered in an actor's mailbox. Messages sent to the same actor are processed atomically with respect to each other. Moreover, the programmer may explicitly specify in which order messages should be removed from its mailbox. Like threads, actors support blocking operations using implicit thread pooling as discussed in Section 2.2.3. Compared to a purely event-based approach, users are relieved from writing their own *ad hoc* thread pooling code. Since the internal thread pool can be global to the web application server, the thread pool controller can leverage more information for its decisions [134]. Finally, accesses to an actor's mailbox are race-free. Therefore, resources such as user profiles can be protected by modeling them as (thread-less) actors.

2.6 Experimental Results

Optimizing performance across threads and events involves a number of non-trivial trade-offs. Therefore, we do not want to argue that our framework is better than event-based systems or thread-based systems or both. Instead, the following basic experiments show that the performance of our framework is comparable to those of both thread-based and event-based systems.

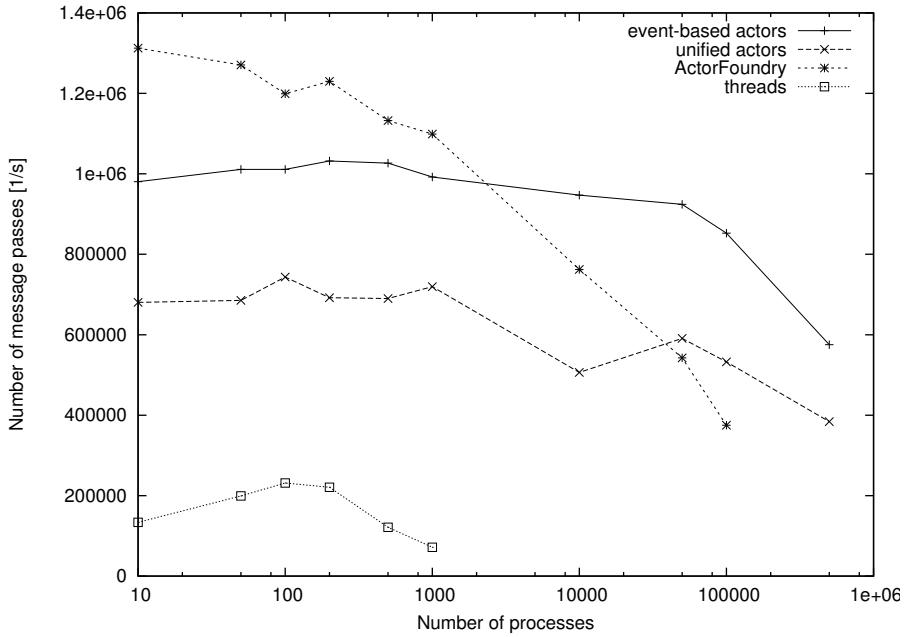

Figure 2.10: Throughput (number of message passes per second) when passing a single message around a ring of processes

2.6.1 Message passing

In the first benchmark we measure throughput of blocking operations in a queue-based application. The application is structured as a ring of n producers/consumers (in the following called *processes*) with a shared queue between each of them. Initially, only one of the queues contains a message and the others are empty. Each process loops taking a message from the queue on its right and putting it into the queue on its left.

The following tests were run on a 3.33 GHz Intel Core2 Duo processor with 3072 MB memory and 6 MB cache; the processor supports 4 hardware threads via two hyper-threaded cores. We used Sun's Java HotSpot Server VM 1.6.0 under Linux 2.6.32 (SMP configuration). We configured the JVM to use a maximum heap size of 256 MB, which provides for sufficient physical memory to avoid any disk activity. In each case we took the median of 5 runs. The execution times of equivalent implementations written using (1) event-based actors, (2) unified actors, (3) ActorFoundry [81] (version 1.0), an actor framework for Java based on Kilim [117], and (4) pure Java threads, respectively, are compared.

Before discussing our results, we want to point out that, unlike implementations (1) and (2), ActorFoundry does not implement full Erlang-style actors: first, message reception is based on inversion of control; instead of providing a receive

operation that can be used anywhere in the actor's body, methods are annotated to allow invoking them asynchronously. Second, upon reception, messages are only filtered according to the static method dispatching rules of Java; while this enables an efficient implementation, it is less expressive than Erlang-style message filtering using `receive` or `react`, which allows encoding priority messages among others. Given the above two restrictions, comparing the performance of ActorFoundry with our actors is not entirely fair. However, it allows us to quantify how much we have to pay in terms of performance to get the flexibility of full Erlang-style actors compared to ActorFoundry's simpler programming model.

Figure 2.10 shows the number of message passes per second (throughput) depending on the ring size. Note that the horizontal scale is logarithmic. For 200 processes or less, actors are 3.6 times faster than Java threads. This factor increases to 5.1 for purely event-based actors. Event-based actors are more efficient because (1) they do not need to maintain thread-local state (for interoperability with Java threads), (2) they do not transmit implicit sender references, and (3) the overhead of send/receive is lower, since only a single mode of suspension is supported (`react`). At ring sizes of 500 to 1000, the throughput of threads breaks in (only 71736 messages per second for 1000 threads), while the throughput of actors (both event-based and unified) stays basically constant (at around 1,000,000 and 700,000 messages per second, respectively). The process ring cannot be operated with 5000 or more threads, since the JVM runs out of heap memory. In contrast, using actors (both event-based and unified) the ring can be operated with as many as 500,000 processes. For 200 processes or less, the throughput of event-based actors is around 24% lower compared to ActorFoundry. Given that ActorFoundry uses Kilim's CPS transformation for implementing lightweight actors, this slowdown is likely due to the high frequency of exceptions that are used to implement the `react` message receive operation both in event-based and in unified actors. Interestingly, event-based actors scale much better with the number of actors compared to ActorFoundry. At 50,000 processes, both event-based and unified actors are faster than ActorFoundry. At 500,000 processes the ActorFoundry benchmark times out. The improvement in scalability is likely due to the fact that in Scala, actors are implemented using a lightweight fork/join execution environment that is highly scalable. However, most importantly, the high frequency of control-flow exceptions does not negatively impact scalability. This means that control-flow exceptions are indeed a practical and scalable way to implement our nested `react` message receive operation.

2.6.2 I/O performance

The following benchmark scenario is similar to those used in the evaluation of high-performance thread implementations [126, 89]. We aim to simulate the ef-

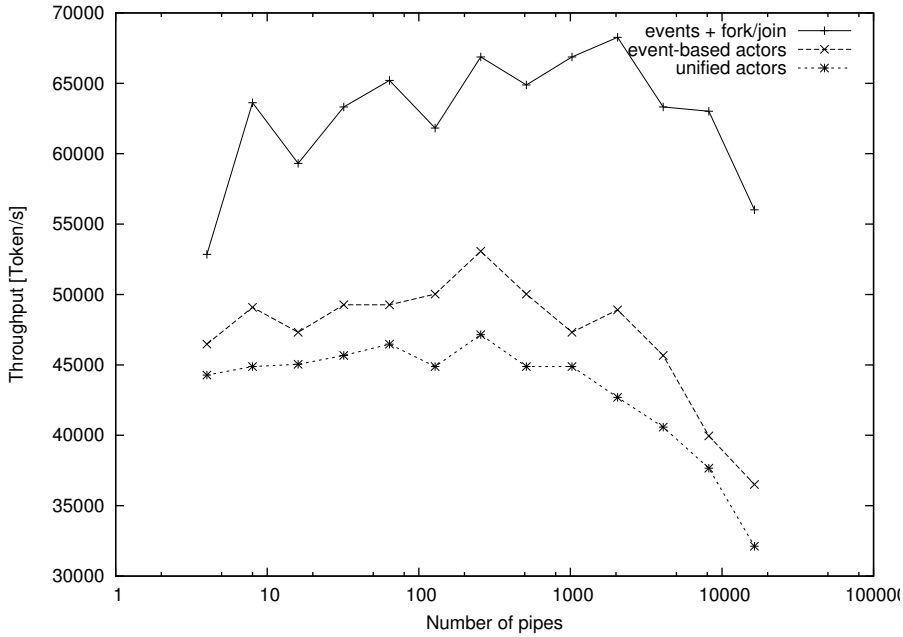

Figure 2.11: Network scalability benchmark, single-threaded

fects of a large number of mostly-idle client connections. For this purpose, we create a large number of FIFO pipes and measure the throughput of concurrently passing a number of tokens through them. If the number of pipes is less than 128, the number of tokens is one quarter of the number of pipes; otherwise, exactly 128 tokens are passed concurrently. The idle end points are used to model slow client links. After a token has been passed from one process to another, the processes at the two end points of the pipe exchange roles, and repeat this conversation.

This scenario is interesting, because it allows us to determine the overhead of actors compared to purely event-driven code in a worst-case scenario. That is, event-driven code always performs best in this case. The main reasons are: (1) directly invoking an event handler is more efficient than creating, sending, and dispatching a message to an actor; (2) the protocol logic is programmed explicitly using inversion of control, as opposed to using high-level combinators.

Figure 2.11 shows the performance of implementations based on events, event-based actors, and unified actors under load. The programs used to obtain these results are slightly extended versions of those discussed in Section 2.3.2. We used the same system configuration as in Section 2.6.1. In each case, we took the average of 5 runs.

The first version uses a purely event-driven implementation; concurrent tasks are run on a lightweight fork/join execution environment [86]. The second version uses event-based actors. The third program is basically the same as the second

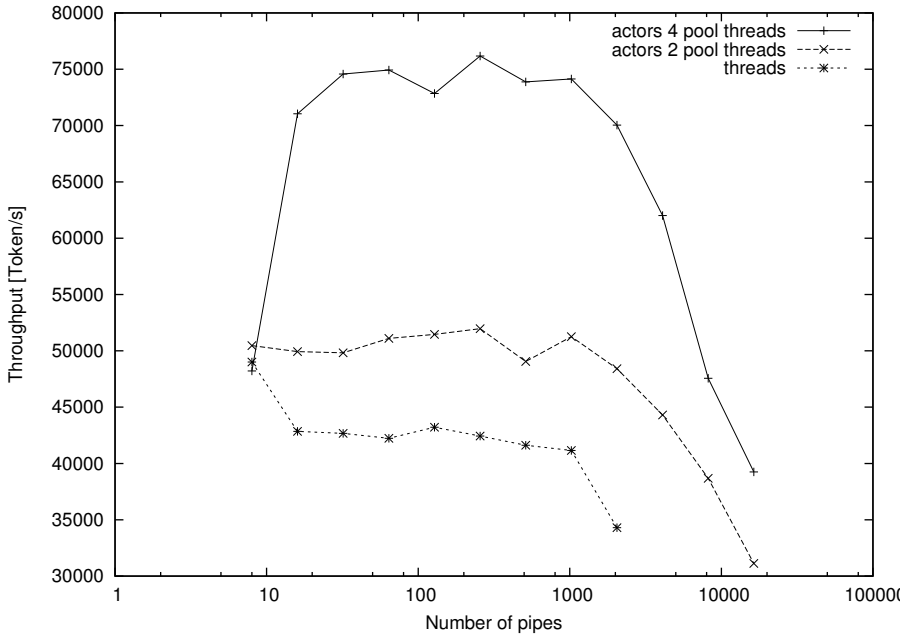

Figure 2.12: Network scalability benchmark, multi-threaded

one, except that actors run in "unified mode". For each implementation we configure the run-time system to utilize only a single worker thread. This allows us to measure overheads (compared to the purely event-driven implementation) that are independent of effects pertaining to scalability. The overhead of unified actors compared to purely event-based actors ranges between 5% (4 pipes) and 15% (2048 pipes). The event-driven version is on average 33% faster than event-based actors. The difference in throughput is at most 58% (at 8196 pipes).

Figure 2.12 shows how throughput changes when the number of utilized worker threads is increased. (Recall that our system configuration supports 4 hardware threads, see Section 2.6.1). We compare the performance of a naive thread-based implementation with an implementation based on unified actors (the third version of our previous experiment). We run the actor-based program using two different configurations, utilizing 4 worker threads or 2 worker threads, respectively. The thread-based version uses two threads per pipe (one reader thread and one writer thread), independent of the number of available hardware threads. Therefore, we expect this implementation to fully utilize the processor cores of our system. For a number of pipes between 16 and 1024, the throughput achieved by actors using 2 worker threads is on average 20% higher than with threads. The overhead of threads is likely due to context switches, which are expensive on the HotSpot JVM, since threads are mapped to heavyweight OS processes. Actors using 4 worker threads provide a throughput that is on average 46% higher than with only

2 worker threads. Note that the gain in throughput is only achieved with at least 16 pipes. The reason is that at 8 pipes, only 2 tokens are concurrently passed through the pipes, which is not sufficient to fully utilize all worker threads. The thread-based version can only be operated up to a number of 2048 pipes; at 4096 pipes the JVM runs out of memory. We ran the actor-based version up to a number of 16384 pipes, at which point throughput has decreased by 39% and 47% for 2 worker threads and 4 worker threads, respectively, compared to the throughput at 1024 pipes.

2.7 Discussion and Related Work

There are two main approaches to model the interleaving of concurrent computations: threads (or processes) and events. In Section 2.7.1 we review previous work on implementing concurrency using threads or events. Section 2.7.2 discusses the application of continuations to lightweight concurrency. In Section 2.7.3 we relate our work to existing actor-based programming systems.

2.7.1 Threads and events

Lauer and Needham [84] note in their seminal work that threads and events are dual to each other. They suggest that any choice of either one of them should therefore be based on the underlying platform. Almost two decades later, Ousterhout [102] argues that threads are a bad idea not only because they often perform poorly, but also because they are hard to use. More recently, von Behren and others [125] point out that even though event-driven programs often outperform equivalent threaded programs, they are too difficult to write. The two main reasons are: first, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern [60]). Second, control flow among handlers is expressed implicitly through manipulation of shared state [26]. In the Capriccio system [126], static analysis and compiler techniques are employed to transform a threaded program into a cooperatively-scheduled event-driven program with the same behavior. Responders [26] provide an event-loop abstraction as a Java language extension. Since their implementation manages one VM thread per event-loop, scalability is limited on standard JVMs.

The approach used to implement thread management in the Mach 3.0 kernel [45] is conceptually similar to ours. When a thread blocks in the kernel, either it preserves its register state and stack and resumes by restoring this state, or it preserves a pointer to a continuation function that is called when the thread is resumed. The latter form of suspension is more efficient and consumes much less memory; it allows threads to be very lightweight. Instead of function pointers we

use closures to represent the continuation of a suspended actor. Moreover, our library provides a set of higher-order functions that allows composing continuation closures in a flexible way (see Section 2.2.4).

2.7.2 Concurrency via continuations

The idea to implement lightweight concurrent processes using continuations has been explored many times [133, 73, 32]. However, existing techniques impose major restrictions when applied to VMs such as the JVM because (1) the security model restricts accessing the run-time stack directly, and (2) heap-based stacks break interoperability with existing code. Delimited continuations based on a type-directed CPS transform [108] can be used to implement lightweight concurrent processes in Scala, the host language of our system. However, this requires CPS-transforming all code that could potentially invoke process-suspending operations. This means that processes are not allowed to run code that cannot be CPS transformed, such as libraries that cannot be transformed without breaking existing clients.

In languages like Haskell and Scala, the continuation monad can also be used to implement lightweight concurrency [28]. In fact, it is possible to define a monadic interface for the actors that we presented in this chapter; however, a thorough discussion is beyond the scope of this thesis. Li and Zdancewic [89] use the continuation monad to combine events and threads in a Haskell-based system for writing high-performance network services. However, they require blocking system calls to be wrapped in non-blocking operations. In our library actors subsume threads, which makes this wrapping unnecessary; essentially, the programmer is relieved from writing custom thread-pooling code.

2.7.3 Actors and reactive objects

The actor model has been integrated into various Smalltalk systems. Actalk [20] is an actor library for Smalltalk-80 that does not support multiple processor cores. Actra [121] extends the Smalltalk/V VM to provide lightweight processes. In contrast, we implement lightweight actors on unmodified VMs.

Our library was inspired to a large extent by Erlang's elegant programming model. Erlang [8] is a dynamically-typed functional programming language designed for programming real-time control systems. The combination of lightweight isolated processes, asynchronous message passing with pattern matching, and controlled error propagation has been proven to be very effective in telecommunication systems [7, 95]. One of our main contributions lies in the integration of Erlang's programming model into a full-fledged object-oriented and functional language. Moreover, by lifting compiler magic into library code we achieve compat-

ibility with standard, unmodified JVMs. To Erlang's programming model we add new forms of composition as well as *channels* which permit strongly-typed and secure inter-actor communication. Termite Scheme [62] integrates Erlang's programming model into Scheme. Scheme's first-class continuations are exploited to express process migration. However, their system apparently does not support multiple processor cores; all published benchmarks were run in a single-core setting.

SALSA [124] is a JVM-based actor language that supports features, such as universal names and migration, which make it particularly suited for distributed and mobile computing. However, its implementation is not optimized for local applications running on a single JVM: first, each actor is mapped to its own VM thread; this limits scalability on standard JVMs [68]. Second, message passing performance suffers from the overhead of reflective method calls. Kilim [117] integrates a lightweight task abstraction into Java using a bytecode postprocessor that is guided by source-level annotations (this postprocessor is also used by ActorFoundry [81] which we discuss and evaluate experimentally in Section 2.6.1.) Building on these tasks, Kilim provides an actor-oriented programming model with first-class message queues (or mailboxes). The model does not support full Erlang-style actors: message queues are not filtered when receiving a message (*i.e.*, messages are always removed in FIFO order from their mailboxes); choice must be encoded using multiple mailboxes and a `select` primitive.

Timber [15] is an object-oriented and functional programming language designed for real-time embedded systems. It offers message passing primitives for both synchronous and asynchronous communication between concurrent *reactive objects*. In contrast to our programming model, reactive objects are not allowed to call operations that might block indefinitely. Frugal objects [61] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model. Similar to reactive objects in Timber, FROBs may not call blocking operations. Other concurrent programming languages and systems also use actors or actor-like abstractions. AmbientTalk [39] provides actors based on communicating event loops [91]. AmbientTalk implements a protocol mapping [37] that allows native (Java) threads to interact with actors while preserving non-blocking communication among event loops. However, the mapping relies on the fact that each actor is always associated with its own VM thread, whereas Scala's actors can be thread-less.

Chapter 3

Join Patterns and Actor-Based Joins

Recently, the pattern matching facilities of languages such as Scala and F# have been generalized to allow representation independence for objects used in pattern matching [47, 120]. Extensible patterns open up new possibilities for implementing abstractions in libraries which were previously only accessible as language features. More specifically, we claim that extensible pattern matching eases the construction of declarative approaches to synchronization in libraries rather than languages. To support this claim, in this chapter we show how a concrete declarative synchronization construct, join patterns, can be implemented in Scala, using extensible pattern matching.

Join patterns [56, 57] offer a declarative way of synchronizing both threads and asynchronous distributed computations that is simple and powerful at the same time. They form part of functional languages such as JoCaml [55] and Funnel [98]. Join patterns have also been implemented as extensions to existing languages [13, 127]. Recently, Russo [109] and Singh [114] have shown that advanced programming language features make it feasible to provide join patterns as libraries rather than language extensions. For example, based on Haskell's software transactional memory [72] it is possible to define a set of higher-order combinators that can encode expressive join patterns.

We argue that an implementation using extensible pattern matching can significantly improve the integration of a joins library into the host language. In existing library-based implementations, pattern variables are represented implicitly as parameters of join continuations. Mixing up parameters of the same type inside the join body may lead to obscure errors that are hard to detect. Our design avoids these errors by using the underlying pattern matcher to bind variables that are explicit in join patterns. Moreover, the programmer may use a rich pattern syntax to express constraints using nested patterns and guards.

The rest of this chapter is organized as follows. In Section 3.1 we briefly highlight join patterns as a declarative synchronization abstraction, how they have

been integrated into other languages before, and how combining them with pattern matching can improve this integration. Section 3.2 shows how to use our library to synchronize threads and actors using join patterns. In Section 3.3 we present a complete implementation of our design as a Scala library [63]. Moreover, we integrate our library into Scala Actors (see Chapter 2); this enables expressive join patterns to be used in the context of advanced synchronization modes, such as future-type message sending. Section 3.4 reviews related work and discusses specific properties of our design in the context of previous systems. Section 3.5 concludes.

This chapter is based on a paper published in the proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION 2008) [65]. The paper is joint work with Tom Van Cutsem. We also acknowledge the anonymous reviewers of the 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP 2008) for their helpful feedback.

3.1 Motivation

Background: Join Patterns A join pattern consists of a body guarded by a linear set of events. The body is executed only when *all* of the events in the set have been signaled to an object. Threads may signal synchronous or asynchronous events to objects. By signaling a synchronous event to an object, threads may implicitly suspend. The simplest illustrative example of a join pattern is that of an unbounded FIFO buffer. In Cω [13], it is expressed as follows:

```
public class Buffer {
  public async Put(int x);
  public int Get() & Put(int x) { return x; }
}
```

Let `b` be an instance of class `Buffer`. Threads may put values into `b` by invoking `b.Put(v)`; invoking `Put` never blocks, since the method is marked `async`. They may also read values from the buffer by invoking `b.Get()`. The join pattern `Get() & Put(int x)` (called a *chord* in Cω) specifies that a call to `Get` may only proceed if a `Put` event has previously been signaled. Hence, if there are no pending `Put` events, a thread invoking `Get` is automatically suspended until such an event is signaled.

The advantage of join patterns is that they allow a *declarative* specification of the synchronization between different threads. Often, the join patterns correspond closely to a finite state machine that specifies the valid states of the object [13]. In the following, we explain the benefits of our new implementation by means of an example.

Example Consider the traditional problem of synchronizing multiple concurrent readers with one or more writers who need exclusive access to a resource. In Cω, join patterns are supported as a language extension through a dedicated compiler. With the introduction of generics in C# 2.0, Russo has made join patterns available in a C# library called Joins [109]. In that library, a multiple reader/one writer lock can be implemented as follows:

```
public class ReaderWriter {
  public Synchronous.Channel Exclusive, ReleaseExclusive;
  public Synchronous.Channel Shared, ReleaseShared;
  private Asynchronous.Channel Idle;
  private Asynchronous.Channel<int> Sharing;
  public ReaderWriter() {
    Join j = Join.Create(); ... // Boilerplate omitted
    j.When(Exclusive).And(Idle).Do(delegate {});
    j.When(ReleaseExclusive).Do(delegate{ Idle(); });
    j.When(Shared).And(Idle).Do(delegate{ Sharing(1); });
    j.When(Shared).And(Sharing).Do(delegate(int n) {
      Sharing(n + 1); });
    j.When(ReleaseShared).And(Sharing).Do(delegate(int n) {
      if (n == 1) Idle(); else Sharing(n - 1); });
    Idle();
  }
}
```

In C# Joins, join patterns consist of linear combinations of channels and a delegate (a function object) which encapsulates the join body. Join patterns are triggered by invoking channels which are special delegates.

In the example, channels are declared as fields of the `ReaderWriter` class. Channel types are either synchronous or asynchronous. Asynchronous channels correspond to asynchronous methods in Cω (*e.g.*, `Put` in the previous example). Channels may take arguments which are specified using type parameters. For example, the `Sharing` channel is asynchronous and takes a single `int` argument. Channels are often used to model (parts of) the internal state of an object. For example, the `Idle` and `Sharing` channels keep track of concurrent readers (if any), and are therefore declared as `private`. To declare a set of join patterns, one first has to create an instance of the `Join` class. Individual join patterns are then created by chaining a number of method calls invoked on that `Join` instance. For example, the first join pattern is created by combining the `Exclusive` and `Idle` channels with an empty delegate; this means that invoking the synchronous `Exclusive` channel (a request to acquire the lock in exclusive mode) will not block the caller if the `Idle` channel has been invoked (the lock has not been acquired).

Even though the verbosity of programs written using C# Joins is slightly higher compared to Cω, basically all the advantages of join patterns are preserved. However, this code still has a number of drawbacks: first, the encoding of the internal state is *redundant*. Logically, a lock in idle state can be represented either by the non-empty `Idle` channel or the `Sharing` channel invoked with `0`.[1]

Note that it is impossible in C# (and in Cω) to use only `Sharing`. Consider the first join pattern. Implementing it using `Sharing` instead of `Idle` requires a delegate that takes an integer argument (the number of concurrent readers):

```
j.When(Exclusive).And(Sharing).Do(delegate(int n) {...}
```

Inside the body we have to test whether $n > 0$ in which case the thread invoking `Exclusive` has to block. Blocking without reverting to lower-level mechanisms such as locks is only possible by invoking a synchronous channel; however, that channel has to be different from `Exclusive` (since invoking `Exclusive` does not block when `Sharing` has been invoked) which re-introduces the redundancy.

Another drawback of the above code is the fact that arguments are passed *implicitly* between channels and join bodies: in the third case, the argument `n` passed to the delegate is the argument of the `Sharing` channel. Contrast this with the Cω buffer example in which the `Put` event explicitly binds its argument `x`. Not only are arguments passed implicitly, the order in which they are passed is merely *conventional* and not checked by the compiler. For example, the delegate of a (hypothetical) join pattern with two channels of type `Asynchronous.Channel<int>` would have two `int` arguments. Accidentally swapping the arguments in the body delegate would go unnoticed and result in errors.

In our implementation in Scala the above example is expressed as follows:

```
join {
  case Exclusive() & Sharing(0) => Exclusive.reply()
  case ReleaseExclusive() => Sharing(0); ReleaseExclusive.reply()
  case Shared() & Sharing(n) => Sharing(n+1); Shared.reply()
  case ReleaseShared() & Sharing(n) if n > 0 =>
    Sharing(n-1); ReleaseShared.reply()
}
```

The internal state of the lock is now represented uniformly using only `Sharing`. Moreover, two formerly separate patterns are unified (patterns 3 and 4 in the C# example) and the `if-else` statement is gone. (Inside join bodies, synchronous events are replied to via their `reply` method; this is necessary since, contrary to

[1] The above implementation actually ensures that an idle lock is always represented as `Idle` and never as `Sharing(0)`. However, this close relationship between `Idle` and `Sharing` is not explicit and has to be inferred from all the join patterns.

C# and Cω, Scala Joins supports multiple synchronous events per pattern, cf. section 3.2.) The gain in expressivity is due to *nested pattern matching*. In the first pattern, pattern matching constrains the argument of `Sharing` to `0`, ensuring that this pattern only triggers when no other thread is sharing the lock. Therefore, an additional `Idle` event is no longer necessary, which decreases the number of patterns. In the last pattern, a *guard* (`if n > 0`) prevents invalid states (*i.e.*, invoking `Sharing(n)` where $n < 0$).

3.2 A Scala Joins Library

We now discuss a Scala library, called Scala Joins, that implements join patterns using extensible pattern matching. In the following Section 3.2.1 we explain how Scala Joins enables the declarative synchronization of threads; Section 3.2.2 describes joins for actors.

3.2.1 Joining threads

Scala Joins draws on Scala's extensible pattern matching facility [47]. This has several advantages: first of all, the programmer may use Scala's rich pattern syntax to express constraints using nested patterns and guards. Moreover, reusing the existing variable binding mechanism avoids typical problems of other library-based approaches where the order in which arguments are passed to the function implementing the join body is merely conventional, as explained in Section 3.1. Similar to C# Joins's channels, joins in Scala Joins are composed of synchronous and asynchronous *events*. Events are strongly typed and can be invoked using standard method invocation syntax. The FIFO buffer example is written in Scala Joins as follows:

```
class Buffer extends Joins {
  val Put = new AsyncEvent[Int]
  val Get = new NullarySyncEvent[Int]
  join {
    case Get() & Put(x) =>
      Get reply x
  }
}
```

To enable join patterns, a class inherits from the `Joins` class. Events are declared as regular fields. They are distinguished based on their (a)synchrony and the number of arguments they take. For example, `Put` is an asynchronous event that takes a single argument of type `Int`. Since it is asynchronous, no return type

is specified (it immediately returns the `Unit` value when invoked). In the case of a synchronous event such as `Get`, the first type parameter specifies the return type. Therefore, `Get` is a synchronous event that takes no arguments and returns values of type `Int`.

Joins are declared using the `join { ... }` construct. This construct enables pattern matching via a list of `case` declarations that each consist of a left-hand side and a right-hand side, separated by `=>`. The left-hand side defines a join pattern through the juxtaposition of a linear combination of asynchronous and synchronous events. As is common in the joins literature, we use `&` as the juxtaposition operator. Arguments of events are usually specified as variable patterns. For example, the variable pattern `x` in the `Put` event can bind to any value (of type `Int`). This means that on the right-hand side, `x` is bound to the argument of the `Put` event when the join pattern matches. Standard pattern matching can be used to constrain the match even further (an example of this is given below).

The right-hand side of a join pattern defines the join body (an ordinary block of code) that is executed when the join pattern matches. Like JoCaml, but unlike Cω and C# Joins, Scala Joins allows any number of synchronous events to appear in a join pattern. Because of this, it is impossible to use the return value of the body to implicitly reply to the single synchronous event in the join pattern. Instead, the body of a join pattern explicitly replies to all of the synchronous events that are part of the join pattern on the left-hand side. Synchronous events are replied to by invoking their `reply` method. This wakes up the thread that originally signalled that event.

3.2.2 Joining actors

We now describe an integration of our joins library with Scala Actors (see Chapter 2). The following example shows how to re-implement the unbounded buffer example using joins:

```
object Put extends Join1[Int]
object Get extends Join
class Buffer extends JoinActor {
  def act() {
    loop {
      receive {
        case Get() & Put(x) => Get reply x
      }
    }
  }
}
```

It differs from the thread-based bounded buffer using joins in the following ways:

- The `Buffer` class inherits the `JoinActor` class to declare itself to be an actor capable of processing join patterns.
- Rather than defining `Put` and `Get` as synchronous or asynchronous *events*, they are all defined as *join messages*, which may support both kinds of synchrony (this is explained in more detail below).
- The `Buffer` actor defines `act` and awaits incoming messages by means of `receive`. Note that it is still possible for the actor to serve regular messages within the `receive` block. Logically, regular messages can be regarded as unary join patterns. However, they don't have to be declared as joinable messages; in fact, our joins extension is fully source compatible with the existing actor library.

We illustrate below how the buffer actor can be used as a coordinator between a consumer and a producer actor. The producer sends an asynchronous `Put` message while the consumer awaits the reply to a `Get` message by invoking it synchronously (using `!?`).[2]

```
val buffer = new Buffer; buffer.start()
actor {
  buffer ! Put(42)
}
actor {
  (buffer !? Get()) match {
    case x: Int => /* process x */
  }
}
```

By applying joins to actors, the synchronization dependencies between `Get` and `Put` can be specified declaratively by the buffer actor. The actor will receive `Get` and `Put` messages by queuing them in its mailbox. Only when all of the messages specified in the join pattern have been received is the body executed by the actor. Before processing the body, the actor atomically removes all of the participating messages from its mailbox. Replies may be sent to any or all of the messages participating in the join pattern. This is similar to the way replies are sent to events in the thread-based joins library described previously.

Contrary to the way events are defined in the thread-based joins library, an actor does not explicitly define a join message to be synchronous or asynchronous.

[2]Note that the `Get` message has return type `Any`. The type of the argument values is recovered by pattern matching on the result, as shown in the example.

We say that join messages are "synchronization-agnostic" because they can be used in different synchronization modes between the sender and receiver actors. However, when they are used in a particular join pattern, the sender and receiver actors have to agree upon a valid synchronization mode. In the previous example, the `Put` join message was sent asynchronously, while the `Get` join message was sent synchronously. In the body of a join pattern, the receiver actor replied to `Get`, but not to `Put`.

The advantage of making join messages synchronization agnostic is that they can be used in arbitrary synchronization modes, including advanced synchronization modes such as ABCL's future-type message sending [140] or Salsa's token-passing continuations [124]. Every join message instance has an associated *reply destination*, which is an output channel on which processes may listen for possible replies to the message. How the reply to a message is processed is determined by the way the message was sent. For example, if the message was sent purely asynchronously, the reply is discarded; if it was sent synchronously, the reply awakes the sender. If it was sent using a future-type message send, the reply resolves the future.

3.3 Joins and Extensible Pattern Matching

Our implementation technique for joins is unique in the way events interact with an extensible pattern matching mechanism. We explain the technique using a concrete implementation in Scala. However, we expect that implementations based on, *e.g.*, the active patterns of F# [120] would not be much different. In the following we first talk about pattern matching in Scala. After that we dive into the implementation of events which crucially depends on properties of Scala's extensible pattern matching. Finally, we highlight how joins have been integrated into Scala's actor framework.

3.3.1 Join patterns as partial functions

In the previous section we used the `join { ... }` construct to declare a set of join patterns. It has the following form:

```
join {
  case pat_1 => body_1
  ...
  case pat_n => body_n
}
```

The patterns pat_i consist of a linear combination of events evt_1 `&` `...` `&` evt_m. Threads synchronize over a join pattern by invoking one or several of the events listed in a pattern pat_i. When all events occurring in pat_i have been invoked, the join pattern matches, and its corresponding join $body_i$ is executed. Just like in the implementation of `receive` (see Section 2.1.1), the pattern matching expression inside braces is a value of type `PartialFunction` that is passed as an argument to the `join` method.

Whenever a thread invokes an event `e`, each join pattern in which `e` occurs has to be checked for a potential match. Therefore, events have to be associated with the set of join patterns in which they participate. As shown before, this set of join patterns is represented as a partial function. Invoking `join(pats)` associates each event occurring in the set of join patterns with the partial function `pats`.

When a thread invokes an event, the `isDefinedAt` method of `pats` is used to check whether any of the associated join patterns match. If yes, the corresponding join body is executed by invoking the `apply` method of `pats`. A question remains: what argument is passed to `isDefinedAt` and `apply`, respectively? To answer this question, consider the simple buffer example from the previous section. It declares the following join pattern:

```
join { case Get() & Put(x) => Get reply x }
```

Assume that no events have been invoked before, and a thread t invokes the `Get` event to remove an element from the buffer. Clearly, the join pattern does not match, which causes t to block since `Get` is a synchronous event (more on synchronous events later). Assume that after thread t has gone to sleep, another thread s adds an element to the buffer by invoking the `Put` event. Now, we want the join pattern to match since both events have been invoked. However, the result of the matching does not only depend on the event that was last invoked but also on the fact that *other events* have been invoked previously. Therefore, it is *not* sufficient to simply pass a `Put` message to the `isDefinedAt` method of the partial function the represents the join patterns. Instead, when the `Put` event is invoked, the `Get` event has to somehow "pretend" to also match, even though it has nothing to do with the current event. While previous invocations can simply be buffered inside the events, it is non-trivial to make the pattern matcher actually consult this information during the matching, and "customize" the matching results based on this information. To achieve this customization we use extensible pattern matching.

3.3.2 Extensible pattern matching

Emir et al. [47] recently introduced *extractors* for Scala that provide representation independence for objects used in patterns. Extractors play a role similar to *views* in functional programming languages [128, 101] in that they allow con-

versions from one data type to another to be applied implicitly during pattern matching. As a simple example, consider the following object that can be used to match even numbers:

```
object Twice {
  def apply(x: Int) = x*2
  def unapply(z: Int) = if (z%2 == 0) Some(z/2) else None
}
```

Objects with `apply` methods are uniformly treated as functions in Scala. When the function invocation syntax `Twice(x)` is used, Scala implicitly calls `Twice.apply(x)`. The `unapply` method in `Twice` reverses the construction in a pattern match. It tests its integer argument `z`. If `z` is even, it returns `Some(z/2)`. If it is odd, it returns `None`. The `Twice` object can be used in a pattern match as follows:

```
val x = Twice(21)
x match {
  case Twice(y) => println(x+" is two times "+y)
  case _ => println("x is odd")
}
```

To see where the `unapply` method comes into play, consider the match against `Twice(y)`. First, the value to be matched (`x` in the above example) is passed as argument to the `unapply` method of `Twice`. This results in an optional value which is matched subsequently.[3] The preceding example is expanded as follows:

```
val x = Twice.apply(21)
Twice.unapply(x) match {
  case Some(y) => println(x+" is two times "+y)
  case None => println("x is odd")
}
```

Extractor patterns with more than one argument correspond to `unapply` methods returning an optional tuple. Nullary extractor patterns correspond to `unapply` methods returning a Boolean.

In the following we show how extractors can be used to implement the matching semantics of join patterns. In essence, we define appropriate `unapply` methods for events which get implicitly called during the matching.

3.3.3 Matching join patterns

As shown previously, a set of join patterns is represented as a partial function. Its `isDefinedAt` method is used to find out whether one of the join patterns matches.

[3] The optional value is of parameterized type `Option[T]` that has the two subclasses `Some[T](x: T)` and `None`.

In the following we are going to explain the code that the Scala compiler produces for the body of this method. Let us revisit the join pattern that we have seen in the previous section:

```
Get() & Put(x)
```

In our library, the & operator is an extractor that defines an `unapply` method; therefore, the Scala compiler produces the following matching code:

```
&.unapply(m) match {
  case Some((u, v)) =>
    u match {
      case Get() => v match {
        case Put(x) => true
        case _ => false }
      case _ => false }
  case None => false }
```

We defer a discussion of the argument m that is passed to the & operator. For now, it is important to understand the general scheme of the matching process. Basically, calling the `unapply` method of the & operator produces a pair of intermediate results wrapped in `Some`. Standard pattern matching decomposes this pair into the variables `u` and `v`. These variables, in turn, are matched against the events `Get` and `Put`. Only if both of them match, the overall pattern matches.

Since the & operator is left-associative, matching more than two events proceeds by first calling the unapply methods of all the & operators from right to left, and then matching the intermediate results with the corresponding events from left to right. Events are objects that have an `unapply` method; therefore, we can expand the code further:

```
&.unapply(m) match {
  case Some((u, v)) =>
    Get.unapply(u) match {
      case true => Put.unapply(v) match {
        case Some(x) => true
        case None => false }
      case false => false }
  case None => false }
```

As we can see, the intermediate results produced by the `unapply` method of the & operator are passed as arguments to the `unapply` methods of the corresponding events. Since the `Get` event is parameterless, its `unapply` method returns a Boolean, telling whether it matches or not. The `Put` event, on the other hand, takes a parameter; when the pattern matches, this parameter gets bound to a concrete value that is produced by the `unapply` method.

The `unapply` method of a parameterless event such as `Get` essentially checks whether it has been invoked previously. The `unapply` method of an event that takes parameters such as `Put` returns the argument of a previous invocation (wrapped in `Some`), or signals failure if there is no previous invocation. In both cases, previous invocations have to be buffered inside the event.

Firing join patterns As mentioned before, executing the right-hand side of a pattern that is part of a partial function amounts to invoking the `apply` method of that partial function. Basically, this repeats the matching process, thereby binding any pattern variables to concrete values in the pattern body. When firing a join pattern, the events' `unapply` methods have to dequeue the corresponding invocations from their buffers. In contrast, invoking `isDefinedAt` does not have any effect on the state of the invocation buffers. To signal to the events in which context their `unapply` methods are invoked, we therefore need some way to propagate out-of-band information through the matching. For this, we use the argument m that is passed to the `isDefinedAt` and `apply` methods of the partial function. The `&` operator propagates this information verbatim to its two children (its `unapply` method receives m as argument and produces a pair with two copies of m wrapped in `Some`). Eventually, this information is passed to the events' `unapply` methods.

3.3.4 Implementation details

Events are represented as classes that contain queues to buffer invocations. Figure 3.1 shows the abstract `Event` class, which is the super class of all synchronous and asynchronous events.[4] The `Event` class takes two type arguments `R` and `Arg` that indicate the result type and parameter type of event invocations, respectively. Events have a unique `owner` which is an instance of the `Joins` class. This class provides the `join` method that we used in the buffer example to declare a set of join patterns. An event can appear in several join patterns declared by its `owner`. The `tag` field holds an identifier which is unique with respect to a given `owner` instance. Whenever an event is invoked via its `apply` method, we first acquire an owner-global lock. The reason is that invoking an event may require accessing the buffers of several events participating in the same join pattern. For thread-safety, all accesses must occur as part of a single atomic action. The lock is released at the point where no owner-global state has to be accessed any more. Before checking for a matching join pattern, we append the provided argument to the `buf` list, which queues logical invocations. The abstract `invoke` method is used to run synchronization-specific code; synchronous and asynchronous events dif-

[4] In our actual implementation the fact whether an event is parameterless is factored out for efficiency. For clarity of exposition, we show a simplified class hierarchy.

```
abstract class Event[R, Arg] {
  val owner: Joins
  val tag = owner.freshTag
  var buf: List[Arg] = Nil
  def apply(arg: Arg): R = {
    owner.lock.acquire
    buf = buf ::: List(arg)
    invoke()
  }
  def invoke(): R
  def unapply(isDryRun: Boolean): Option[Arg] = {
    if (isDryRun && !buf.isEmpty)
      Some(buf.head)
    else if (!isDryRun && owner.matches(tag)) {
      val arg = buf.head
      buf = buf.tail
      if (owner.isLastEvent)
        owner.lock.release
      Some(arg)
    } else None
  }
}
```

Figure 3.1: The abstract super class of synchronous and asynchronous events

fer mainly in their implementation of the `invoke` method (we show a concrete implementation for synchronous events below).

In the `unapply` method we first test whether matching occurs during a "dry run", indicated by the `isDryRun` parameter. `isDryRun` is `true` when we only check for a matching join pattern; in this case the buffer state is not modified. The argument of a queued invocation is returned wrapped in `Some`. If there is no previous invocation, we return `None` to indicate that the event, and therefore the current pattern, does not match. When firing a join pattern, `isDryRun` is `false`; in this case the invocations that form part of the corresponding match are removed from their buffers. However, it is still possible that the current event does not match, since the pattern matcher will also invoke the `unapply` methods of events that occur in cases preceding the matching pattern. Therefore, we also have to check that the current event (represented by its unique `tag`) belongs to the actual match (`owner.matches(tag)`). In this case the argument value corresponding to its oldest invocation is removed from the buffer. We also release the owner's lock

```
class SyncEvent[R, Arg] extends Event[R, Arg] {
  val waitQ = new Queue[SyncVar[R]]
  def invoke(): R = {
    val res = new SyncVar[R]
    waitQ += res
    owner.matchAndRun()
    res.get
  }
  def reply(res: R) {
    owner.lock.acquire
    waitQ.dequeue().set(res)
    owner.lock.release
  }
}
```

Figure 3.2: A class implementing synchronous events

if the current event occurs last in the matching pattern.

The `SyncEvent` class shown in Figure 3.2 implements synchronous events. Synchronous events contain a logical queue of waiting threads, `waitQ`, which is implemented using the implicit wait set of synchronous variables.[5] The `invoke` method is run whenever the event is invoked (see above). It creates a new `SyncVar` and appends it to the `waitQ`. Then, the `owner`'s `matchAndRun` method is invoked to check whether the event invocation triggers a complete join pattern. After that, the current thread waits for the `SyncVar` to become initialized by calling its `get` method. If the owner detects (during `owner.matchAndRun()`) that a join pattern triggers, it will apply the join, thereby re-executing the pattern match (binding variables etc.) and running the join body. Inside the body, synchronous events are replied to by invoking their `reply` method. Replying means dequeuing a `SyncVar` and setting its value to the supplied argument. If none of the join patterns matches, the thread that invoked the synchronous event is blocked (upon calling `res.get`) until another thread triggers a join pattern that contains the same synchronous event.

Thread safety Our implementation avoids races when multiple threads try to match a join pattern at the same time; checking whether a join pattern matches is an atomic operation. Notably, the `isDefinedAt/apply` methods of the join set

[5] A `SyncVar` is an atomically updatable reference cell; it blocks threads trying to `get` the value of an uninitialized cell.

are only called from within the `matchAndRun` method of the `Joins` class. This method, in turn, is only called after the owner's lock has been acquired. The `unapply` methods of events, in turn, are only called from within the matching code inside the partial function, and are thus guarded by the same lock. The internal state of individual events is updated consistently: the `apply` method acquires the owner's lock, which is released after matching is finished; the dequeueing of a waiting thread inside the `reply` method is guarded by the owner's lock. We don't assume any concurrency properties of the queues used to buffer invocations or waiting threads.

3.3.5 Implementation of actor-based joins

Actor-based joins integrate with Scala's pattern matching in essentially the same way as the thread-based joins, making both implementations very similar. We highlight how joins are integrated into the actor library, and how reply destinations are supported.

As explained in Section 2.1.1, `receive` is a method that takes a `PartialFunction` as a sole argument, similar to the `join` method defined previously. To make `receive` aware of join patterns, the abstract `JoinActor` class overrides this method by wrapping the partial function into a specialized partial function that understands join messages. `JoinActor` also overrides `send` to set the reply destination of a join message. Message sends such as `a ! msg` are interpreted as calls to `a`'s `send` method.

```
abstract class JoinActor extends Actor {
  override def receive[R](f: PartialFunction[Any, R]): R =
    super.receive(new JoinPatterns(f))
  override def send(msg: Any, replyTo: OutputChannel[Any]) {
    setReplyDest(msg, replyTo)
    super.send(msg, replyTo) }
  def setReplyDest(msg: Any, replyTo: OutputChannel[Any]) { ... }
}
```

`JoinPatterns` (see below) is a special partial function that detects whether its argument message is a join message. If it is, then the argument message is transformed to include out-of-band information that will be passed to the pattern matcher, as is the case for events in the thread-based joins library. The Boolean argument passed to the `asJoinMessage` method indicates to the pattern matcher whether or not join message arguments should be dequeued upon successful pattern matching. If the `msg` argument is not a join message, `asJoinMessage` passes the original message to the pattern matcher unchanged, enabling regular actor messages to be processed as normal.

```
class JoinPatterns[R](f: PartialFunction[Any, R])
  extends PartialFunction[Any, R] {
  override def isDefinedAt(msg: Any) =
    f.isDefinedAt(asJoinMessage(msg, true))
  override def apply(msg: Any) =
    f(asJoinMessage(msg, false))
  def asJoinMessage(msg: Any, isDryRun: Boolean): Any =
    ...
}
```

Recall from the implementation of synchronous events that thread-based joins used constructs such as `SyncVar`s to synchronize the sender of an event with the receiver. Actor-based joins do not use such constructs. In order to synchronize sender and receiver, every join message has a reply destination (which is an `OutputChannel`, set when the message is sent in the actor's `send` method) on which a sender may listen for replies. The `reply` method of a `JoinMessage` simply forwards its argument value to this encapsulated reply destination. This wakes up an actor that performed a synchronous send (`a !? msg`) or that was waiting on a future (`a !! msg`).

3.4 Discussion and Related Work

In Section 3.1 we already introduced Cω [13], a language extension of C# supporting *chords*, linear combinations of methods. In contrast to Scala Joins, Cω allows at most one synchronous method in a chord. The thread invoking this method is the thread that eventually executes the chord's body. The benefits of Cω as a language extension over Scala Joins are that chords can be enforced to be well-formed and that their matching code can be optimized ahead of time. In Scala Joins, the joins are only analyzed at pattern-matching time. The benefit of Scala Joins as a library extension is that it provides more flexibility, such as multiple synchronous events. Benton et al. [13] note that supporting general guards in join patterns is difficult to implement efficiently as it requires testing all possible combinations of queued messages to find a match. Side effects pose another problem. The authors suggest a restricted language for guards to overcome these issues. However, to the best of our knowledge, there is currently no joins framework that supports a sufficiently restrictive yet expressive guard language to implement efficient guarded joins. Our current implementation handles (side-effect free) guards that only depend on arguments of events that queue at most one invocation at a time.

Russo's C# Joins library [109] exploits the expressiveness of C# 2.0's generics to implement Cω's synchronization constructs. Piggy-backing on an existing

variable binding mechanism allows us to avoid problems with C# Joins' delegates where the order in which arguments are passed is merely conventional. Scala Joins extends both Cω and C# Joins with *nested patterns* that can avoid certain redundancies by generalizing events and patterns. CCR [27] is a C# library for asynchronous concurrency that supports join patterns without synchronous components. Join bodies are scheduled for execution in a thread pool. Our library integrates with JVM threads using synchronous variables, and supports event-based programming through its integration with Scala Actors. CML [107] allows threads to synchronize on first-class composable events; because all events have a single commit point, certain protocols may not be specified in a modular way (for example when an event occurs in several join patterns). By combining CML's events with all-or-nothing transactions, transactional events [44] overcome this restriction but may have a higher overhead than join patterns.

Synchronization in actor-based languages is a well-studied domain. Activation based on message sets [58] is more general than joins since events/channels have a fixed owner, which enables important optimizations. Other actor-based languages allow for a synchronization style similar to that supported by join patterns. For example, *behavior sets* in Act++ [80] or *enabled sets* in Rosette [123] allow an actor to restrict the set of messages which it may process. They do so by partitioning messages into different sets representing different actor states. Joins do not make these states explicit, but rather allow state transitions to be encoded in terms of sending messages. The novelty of Scala Joins for actors is that such synchronization is integrated with the actor's standard message reception operation using extensible pattern matching. In SALSA [124] actors can synchronize upon the arrival of multiple replies to previously sent messages. In contrast, Scala Joins allow actors to synchronize on incoming messages that do not originate from previous requests. Work by Sulzmann et al. [119] extends Erlang-style actors with receive patterns consisting of multiple messages, which is very similar to our join-based actors. The two approaches are complementary: their work focuses on providing a formal matching semantics in form of Constraint Handling Rules [59] whereas the emphasis of our work lies on the integration of joins with extensible pattern matching; Scala Joins additionally permits joins for standard (non-actor) threads that do not have a mailbox. JErlang [106] integrates join-style message patterns into Erlang's receive construct. In contrast to our approach which does not need special compiler support, their system relies on an experimental syntax transformation module that is run as part of compilation. JErlang's patterns may be non-linear (a single type of message occurs several times in the same pattern) and guards may be side-effect-free Boolean expressions (without calls to user-defined functions). The pattern language supported by our system is less expressive, although it could be extended to handle more general guards. Our system contributes synchronization-agnostic messages: each message is as-

sociated with its sending actor (which is transmitted implicitly); synchronous and future-type message send operations are supported by replying to the messages of the corresponding request.

3.5 Conclusion

We presented a novel implementation of join patterns based on Scala's extensible pattern matching. Unlike previous library-based implementations, the embedding into pattern matching enables us to reuse an existing variable binding mechanism, thereby avoiding certain usage errors. Our technique also opens up new possibilities for supporting features such as nested patterns and guards in joins. Programs written using our library are often as concise as if written in dedicated language extensions. We implemented our approach as a Scala library and furthermore integrated it with the Scala Actors concurrency framework without changing the syntax and semantics of programs without joins.

Chapter 4

Type-Based Actor Isolation

In this chapter we introduce a new type-based approach to actor isolation. The main idea of our approach is to use a type system with static capabilities to enforce uniqueness of object references. Transferring a mutable object from one actor to another requires a unique reference to that object. Moreover, after the (unique) object has been sent, it is no longer accessible to the sender; the capability required to access the object has been consumed. Thereby, we ensure that at most one actor accesses a mutable object at any point in time; this means that actors are isolated even in the presence of efficient by-reference message passing.

The rest of this chapter is organized as follows. Section 4.2 provides the necessary background on statically checking separation and uniqueness by reviewing existing proposals from the literature. In Section 4.3 we provide an informal overview of our type system and the user-provided annotations. Section 4.4 presents a formal account in the context of a small core language with objects. We establish soundness of the type system (see Section 4.5) using a small-step operational semantics and the traditional method of preservation and progress theorems (a complete proof appears in Appendix A.) Section 4.6 introduces immutable types, which permit more flexible aliasing patterns; they integrate seamlessly with uniqueness types. In Section 4.7 we extend our formal development with actors. This allows us to prove an isolation theorem, which says that actors only access immutable objects concurrently. Section 4.8 presents several extensions of our system informally, notably closures and nested classes. In Section 4.9 we outline our implementation for Scala; we also provide evidence that our system is practical by using it to type-check mutable collection classes and real-world, concurrent programs.

This chapter is based on a paper published in the proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010) [69]. The material on immutable types (Section 4.6) as well as the extension to actor-based concurrency (Section 4.7 including the isolation theorem of Section 4.7.6) is new

and has not been published, yet. Section 4.8.1 adds a discussion of an application to automatic resource management; the ray tracer example in Section 4.9.1 is also new. The conference paper (without the mentioned extensions) was written by the author of this thesis, except for parts of the introduction, which were contributed by Martin Odersky; he also helped shape the final version of our formal semantics and type system. We are grateful for the detailed and helpful feedback of the anonymous reviewers.

4.1 Introduction

A promise of message-based concurrency are robust programming models that scale from multi-core processors to distributed systems, such as web applications. However, this requires a uniform semantics for local and remote message sends (see Chapter 1). To support distributed systems where actors do not share state, we consider a semantics where sent messages are moved from the memory region of the sender to the (disjoint) memory region of the receiver. Thus, a message is no longer accessible to its sender after it has been sent. This semantics also avoids data races if concurrent processes running on the same computer communicate only by passing messages.

However, moving messages physically requires expensive marshaling/copying. This would prohibit the use of message passing altogether in performance-critical code that deals with large messages, such as network protocol stacks [48, 50]. To achieve the necessary performance in these applications, the underlying implementation must pass messages between processes running on the same shared-memory computer by reference. But reference passing makes it challenging to enforce race freedom, especially in the context of imperative, object-oriented languages, where aliasing of object references is common.

The two main approaches to address this problem are:

- Immutable messages. Only allow passing objects of immutable type. Examples are Java-style primitive types (*e.g.*, `int`, `boolean`), immutable strings, and tree-shaped data, such as XML.

- Alias-free messages. Only a single, unique reference may point to each message; upon transfer, the unique reference becomes unusable [50, 116, 117].

Immutable messages are used, for instance, in Erlang (see Section 2.7.3). The second approach usually imposes constraints on the shape of messages (*e.g.*, trees [117]). Even though messages are passed by reference, message shape constraints may lead indirectly to copying overheads: data stored in an object graph that does not

Proposal	Type System	Unique Objects
Islands	(~ linear types)	alias-free
Balloons	(abstr. interpr.)	alias-free
PacLang	quasi-linear types	alias-free, flds. prim.
PRFJ	expl. ownership	alias-free
StreamFlex	impl. ownership	alias-free, flds. prim.
Kilim	impl. ownership	alias-free
External U.	expl. ownership	intern. aliases
UTT	impl. ownership	intern. aliases
BR	capabilities	intern. aliases
MOAO	expl. ownership	intern. aliases
Sing#	capabilities	intern. aliases
This thesis	capabilities	intern. aliases

Table 4.1: Proposals for uniqueness: types and unique objects

satisfy the shape constraints must first be serialized into a permitted form before it can be sent within a message.

In our actors library described in Chapter 2, messages can be any kind of data, mutable as well as immutable. When sending messages between actors operating on the same computer, the message state is not copied; instead, messages are transferred by reference only. This makes the system flexible and guarantees high performance. However, without additional static or dynamic checks, passing mutable messages by reference can lead to data races.

This chapter introduces a new type-based approach to statically enforce race safety in Scala's actors. Our main goal is to ensure race safety with a type system that's simple and expressive enough to be deployed in production systems by normal users. Our system removes important limitations of existing approaches concerning permitted message shapes. At the same time it allows interesting programming idioms to be expressed with fewer annotations than previous work, while providing equally strong safety guarantees.

4.2 Statically Checking Separation and Uniqueness

Our approach to isolating actors is based on a static type system to check separation and uniqueness properties of object references. Section 4.2.1 reviews related work on uniqueness and full encapsulation. In Section 4.2.2 we relate our approach to linear types, region-based memory management, and separation logic. We discuss other approaches to isolating concurrent processes in Section 4.2.3.

Proposal	Encapsulation	Program Annotations
Islands	full	type qualifiers, purity
Balloons	full	type qualifiers
PacLang	full	type qualifiers
PRFJ	deep/full	owners, regions, effects
StreamFlex	full	type qualifiers
Kilim	full	type qualifiers
External U.	deep	owners, borrowing
UTT	deep	type qualifiers, regions
BR	deep	type qual., regions, effects
MOAO	full	simple owners, borrowing
Sing#	full	type qualifiers, borrowing
This thesis	full	type qualifiers

Table 4.2: Proposals for uniqueness: encapsulation and annotations

4.2.1 Type systems for uniqueness and full encapsulation

There exists a large number of proposals for unique object references. A comprehensive survey is beyond the scope of this thesis; Clarke and Wrigstad [29] provide a good overview of earlier work where unique references are not allowed to point to internally-aliased objects, such as doubly-linked lists. Aliases that are strictly internal to a unique object are not observable by external clients and are therefore harmless [136]. Importantly, "external" uniqueness enables many interesting programming patterns, such as merging of data structures and abstraction of object creation (through factory methods [60]). In the following we consider two kinds of alias encapsulation policies:

- *Deep encapsulation:* [94] the only access (transitively) to the internal state of an object is through a single entry point. References to external state are allowed.

- *Full encapsulation:* same as deep encapsulation, except that no references to objects outside the encapsulated object from within the encapsulation boundary are permitted.

Our motivation to study full encapsulation is concurrent programming, where deep encapsulation is generally not sufficient to avoid data races. In the following we compare proposals from the literature that provide either uniqueness with internal aliasing, full alias encapsulation, or both. (Section 4.2.2 discusses other related work on linear types, regions, and program logics.)

Table 4.1 classifies existing approaches according to (a) the kind of type system they use, and (b) the notion of unique/linear objects they support. Table 4.2

classifies the same approaches according to (c) the alias encapsulation they provide, and (d) the program annotations they require for static (type) checking. We distinguish three main kinds of type systems: explicit (parametrized) ownership types [31], implicit ownership types, and systems based on capabilities/permissions. The third column of Table 4.1 specifies whether unique objects are allowed to have internal aliases; in general, alias-free unique references may only point to tree-shaped object graphs. The second column of Table 4.2 indicates the encapsulation policy. We are going to explain the program annotations in the third column of Table 4.2 in the context of each proposal.

Islands [78] provide fully-encapsulated objects protected by "bridge" classes. However, extending an Island requires unique objects, which must be alias-free. Almeida's Balloon Types [4] provide unique objects with full encapsulation; however, the unique object itself may not be (internally) aliased. Ennals et al. [48] have used quasi-linear types [82] for efficient network packet processing in PacLang; in their system, packets may not contain nested pointers. The PRFJ language of Boyapati et al. [18] associates owners with shared-memory locks to verify correct lock acquisition. PRFJ does not support unique references with internal aliasing; it requires adding explicit owner parameters to classes and read/write effect annotations. StreamFlex [116] (like its successor Flexotasks [9]) supports stream-based programming in Java. It allows zero-copy message passing of "capsule" objects along linear filter pipelines. Capsule classes must satisfy stringent constraints: their fields may only store primitive types or arrays of primitive types. Kilim [117] 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. StreamFlex, Flexotasks, and Kilim are systems where object ownership is enforced implicitly, *i.e.*, types in their languages do not have explicit owners or owner parameters. This keeps their annotation systems pleasingly simple, but significantly reduces expressivity: unique objects may not be internally-aliased.

Universe Types [42, 41] is a more general implicit ownership type system that restricts only object mutations, while permitting arbitrary aliasing. Universe Types are particularly attractive for us, because its type qualifiers are very lightweight. In fact, some of the annotations proposed in this paper are very similar, suggesting a close connection. Generally, however, the systems are very different, since restricting only modifications of objects does not prevent data races in a concurrent setting. UTT [93] extends Universe Types with ownership transfer; it increases the flexibility of external uniqueness by introducing explicit regions ("clusters"); an additional static analysis helps avoiding common problems of destructive reads. In Vault [51] Fähndrich and DeLine introduce adoption and focus for embedding linear values into aliased containers (adoption), providing a way to recover linear access to such values (focus). Their system builds on Alias

Types [131] that allow a precise description of the shape of recursive data structures in a type system. Boyland and Retert [19] (BR in Table 4.1 and Table 4.2) generalize adoption to model both effects and uniqueness. While their type language is very expressive, it is also clearly more complex than Vault. Their realized source-level annotations include region ("data group") and effect declarations.

MOAO [30] combines a minimal notion of ownership, external uniqueness, and immutability into a system that provides race freedom for active objects [140, 22]. To reduce the annotation burden messages have a flat ownership structure: all objects in a message graph have the same owner. It requires only simple owner annotations; however, borrowing requires existential owners [137] and owner-polymorphic methods. Sing# [50] uses capabilities [51] to track the linear transfer of message records that are explicitly allocated in a special exchange heap reserved for inter-process communication. Their tracked pointers may have internal aliases; however, storing a tracked pointer in the heap requires dynamic checks that may lead to deadlocks. Their annotation system consists of type qualifiers as well as borrowing ("expose") blocks for accessing fields of unique objects.

Summary In previous proposals, borrowing has largely been treated as a second-class citizen. Several researchers [19, 93] have pointed out the problems of ad-hoc type rules for borrowing (particularly in the context of destructive reads). Concurrency is likely to exacerbate these problems. However, principled treatments of borrowing currently demand a high toll: they require either existential ownership types with owner-polymorphic methods, or type systems with explicit regions, such as Universe Types with Transfer or Boyland and Retert's generalized adoption. Both alternatives significantly increase the syntactic overhead and are extremely challenging to integrate into practical object-oriented programming languages.

4.2.2 Linear types, regions, and separation logic

In functional languages, linear types [129] have been used to implement operations like array updating without the cost of a full copy. An object of linear type must be used exactly once; as a result, linear objects must be threaded through the computation. Wadler's `let!` or observers [97] can be used to temporarily access a linear object under a non-linear type. Linear types have also been combined with regions, where `let!` is only applicable to regions [132]. Bierhoff and Aldrich [14] build on an expressive linear program logic for modular typestate checking in an object-oriented setting. Their system provides unique references; however, uniqueness does not imply an encapsulation property like external uniqueness [29], which is crucial for our application to actor isolation. In

the system of Bierhoff and Aldrich, the encapsulation policy would have to be expressed using explicit invariants provided by the programmer; it is not clear whether the encapsulation policy we use can be expressed in their system. Beckman et al. [12] use a similar system for verifying the correct use of software transactions. JAVA(X) [40] tracks linear and affine resources using type refinement and capabilities, which are structured, unlike ours. The authors did not consider applications to concurrency. Shoal [6] combines static and dynamic checks to enforce sharing properties in concurrent C programs; in contrast, our approach is purely static. Like in region-based memory management [122, 130, 77, 141], in our system objects inside a region may not refer to objects inside another region that may be separately consumed. The main differences are: first, regions in our system do not have to be consumed/deleted, since they are garbage-collected; second, regions in our system can be merged. Separation logic [100] is a program logic designed to reason about separation of portions of the heap; the logic is not decidable, unlike our approach. Bornat et al. [17] study permission accounting in separation logic; unlike our system, their approach is not automated. Parkinson and Bierman [104] extend the logic to an object-oriented setting; however, applications [43] still require a theorem prover and involve extensive program annotation. To avoid aliasing, swapping [70] has been proposed previously as an alternative to copying pointers; in contrast to earlier work, our approach integrates swapping with internally-aliased unique references and local aliasing.

4.2.3 Isolating concurrent processes

ProActive [21] is a middleware for programming distributed Grid applications. Its main abstractions are deterministic active objects [23] that communicate via asynchronous method calls and futures. Transfering data between different active objects requires cloning; this also applies to communication among components in ToolBus [38]. Coboxes [111] generalize active objects by supporting cooperative scheduling of multiple tasks inside a single active object. Moreover, coboxes partition the heap hierarchically into isolated groups of objects. Access to objects local to a cobox is guaranteed to be race-free; only immutable objects can be shared by multiple coboxes. Transferring mutable objects in a way that makes them locally accessible inside the receiving cobox is only possible via deep copying. This is unlike the approach presented in this chapter which allows transferring mutable objects by reference between concurrent actors, while guaranteeing race-free access to these objects.

Guava [11] is a variant of Java that categorizes data into objects, monitors, and values. Objects always remain local to a single thread. Monitors may be freely shared between threads, since their methods are always synchronized. Values behave like primitives in Java: they don't have identity and they are deeply

```
def runTests(kind: String, tests: List[Files]) {
  var succ, fail = 0
  val logs: LogList @unique = new LogList
  for (test <- tests) {
    val log: LogFile @unique = createLogFile(test)
    // run test...
    logs.add(log)
  }
  report(succ, fail, logs)
}
def report(succ: Int, fail: Int, logs: LogList @unique) {
  master ! new Results(succ, fail, logs)
}
```

Figure 4.1: Running tests and reporting results

copied (except for embedded references to monitors) on assignment. A destructive "move" operator can be used for assigning values without the overhead of a copy operation. However, only immutable values may be passed by reference from one thread to another, unlike our approach which also permits mutable values. Guava allows objects to be temporarily passed between monitors; in contrast, our system allows objects to be transfered *permanently* to another actor protecting its state. Similar to our system, regions and ownership are used to enforce that monitors do not hold on to objects owned by a different monitor or value. The authors present the type system as a collection of informal rules. It is suggested that a variant of the race-free type system of Flanagan and Abadi [53] could be used to model Guava's region types. X10 [110] is a new object-oriented programming language with a type system supporting constraints [96] to check properties of concurrent programs. Using *place types* the location of data can be tracked at compile time; the compiler can exploit this information to optimize performance and scalability on parallel platforms. X10 does not support unique references to transfer mutable data; instead, computations can move to the (mutable) data which remains local.

4.3 Overview

As a running example we use the simplified core of partest, the parallel testing framework used to test the Scala compiler and standard libraries. The framework uses actors for running multiple tests in parallel, thereby achieving significant speed-ups on multi-core processors.

In this application, a master actor creates multiple worker actors, each of which receives a list of tests to be run. A worker executes the `runTests` method shown in Figure 4.1, which prompts the test execution. Each test is associated with a log file that records the output produced by compiling and, in some cases, running the test. These log files are collected in the `logs` list that the worker sends back to the master upon completing the test execution. Note that log files are neither immutable nor cloneable.[1] Therefore, it is impossible to create a copy of the log files upon sending them to the master. To ensure that passing `logs` by reference is safe, we annotate its type as `@unique`. Inside the for-comprehension, we also annotate the `log` variable, which refers to a single log file, as `@unique`; this enables adding `log` to `logs` without losing the uniqueness of `logs`. (Below we explain how to check that the invocation of `add` is safe.)

The worker reports the test results to its master using `report`. The `@unique` annotation requires the `logs` parameter to be unique. Moreover, it indicates that the caller loses the permission to access the passed argument subsequently. In fact, any object reachable from `logs` becomes inaccessible to the caller. Conversely, `report` has the full permission to access `logs`. This allows sending it as part of a unique `Results` message to the master. Sending a unique object (using the `!` method) makes it unusable, as well as all objects reachable from it, including the `logs`.

In the above example we have shown how to use the `@unique` annotation to ensure the safety of passing message objects by reference. In the following we introduce aliasing invariants of our type system that guarantee the soundness of this approach.

4.3.1 Alias invariant

The alias invariant that our system guarantees is based on a separation predicate on stack variables. (Below, we extend this invariant to fields.) We characterize two variables x, y as being *separate*, written $separate(x, y)$, if and only if they do not share a common reachable object.[2] In other words, two variables are separate if they point to disjoint object graphs in the heap. Based on this predicate we define what it means for a variable to be *separately-unique*.

Definition 1 (Separate Uniqueness) *A variable x is separately-unique* iff $\forall y \neq x.\ y$ *live* $\Rightarrow separate(x, y)$.

[1] `LogFile` inherits from `java.io.File`, which is not cloneable.

[2] For simplicity we leave the heap implicit in the following discussion; we formalize it precisely in Section 4.4.

Figure 4.2: Comparing (a) external uniqueness and (b) separate uniqueness ($\Rightarrow$ unique reference, $\rightarrow$ legal reference, $\dashrightarrow$ illegal reference)

This definition of uniqueness implies that if x is a separately-unique variable, there is no other live variable on the stack that shares a common reachable object with x.

In contrast, this does not hold for external uniqueness [29], which is the notion of uniqueness most closely related to ours. Figure 4.2 compares the two notions of uniqueness. We assume that object A owns object B. This means that references r and i are internal to the ownership context of A. Ownership makes reference f' illegal. u is a unique reference to A; uniqueness makes reference f illegal. Importantly, external uniqueness permits the s reference, which points to an object that is reachable without using u. Therefore, even if u is unusable, the target of s is still reachable. In contrast, our system enforces full encapsulation by forbidding the s reference. This means that making u unusable results in all objects reachable using u being unusable. Therefore, separate uniqueness avoids races when unique references are passed among concurrent processes (we prove this in Appendix A). With external uniqueness, one has to enforce additional constraints to ensure safety [30].

We are now ready to state the alias invariant that our type system provides.

Definition 2 (Alias Invariant) *Unique parameters are separately-unique.*

Note that this invariant does not require unique *variables* to be separately-unique. In particular, unique variables may be aliased by local variables on the stack. However, it is valid to pass a unique variable to a method expecting a unique argument. This means that it must always be possible to make unique variables separately-unique. In the following we explain how we can enforce this using a system of capabilities.

4.3.2 Capabilities

A unique variable has a type guarded by some capability ρ, written $\rho \triangleright \mathsf{T}$ (typically, T is the underlying class type). Capabilities have two roles: first, they serve

as static names for (disjoint) regions of the heap. Second, they embody access permissions [130, 19, 25] to those regions. The typing rules of our system consume and produce sets of capabilities. A variable with a type guarded by ρ can only be accessed if ρ is available, *i.e.*, if it is contained in the input set of capabilities in the typing rule. Therefore, consuming ρ makes all variables of types guarded by ρ unusable. The following invariant expresses the fact that accessible variables guarded by different capabilities point to disjoint object graphs.

Definition 3 (Capability Type Invariant) *Let x be a unique variable with guarded type $\rho \triangleright T$. If y is an accessible variable such that $\neg separate(x, y)$, then y has guarded type $\rho \triangleright S$.*

Note that the above definition permits variables z of guarded type $\delta \triangleright U$ ($\delta \neq \rho$) such that $\neg separate(x, z)$. This is safe as long as δ is not available, which makes z inaccessible.

In summary, the above invariant implies that if x's type is guarded by some capability ρ, consuming ρ makes all variables y such that $\neg separate(x, y)$ inaccessible. Therefore, the separate uniqueness of unique arguments can be enforced as follows: first, unique arguments must have guarded type $\rho \triangleright T$. Second, capability ρ is consumed (and, therefore, must be available) in the caller's context. Third, capabilities guarding other arguments (if any) must be different from ρ. In Section 4.4 we formalize the mapping between annotations in the surface syntax, such as `@unique`, and method types with capabilities.

We have introduced two invariants that are fundamental to the soundness of unique variables and parameters in our system. In the following we motivate and discuss extensions of our annotation system.

4.3.3 Transient and peer parameters

Our discussion of the example shown in Figure 4.1 did not address the problem of mutating the unique `logs` list after running a single test. Crucially, `logs` must remain unique (and accessible) after adding `log` to it. This means we cannot use `@unique` to annotate the receiver of the `add` method, since it would make `logs` inaccessible. Furthermore, `add`'s parameter must point into the same region as the receiver, since `add` makes `log` reachable from `logs`. To express those requirements, we introduce two additional annotations: `@transient` and `@peer`. They are used to annotate the `add` method as follows.

```
class LogList {
  var elem: LogFile = null
  var next: LogList = this
  @transient def add(file: LogFile @peer(this)) =
```

```
    if (isEmpty) { elem = file; next = new LogList }
    else next.add(file)
}
```

Note that the `@transient` annotation applies to the receiver, *i.e.*, `this`. `@transient` is equivalent to `@unique`, except that it does not consume the capability to access the annotated parameter (including the receiver). Consequently, it is illegal to pass a transient parameter, or any object reachable from it, to a method expecting a unique parameter, which would consume its capability.

The `@peer(this)` annotation on the parameter type indicates that `file` points into the same region as `this`. The effect on available capabilities is determined by the argument of `@peer`: since `this` is transient, invoking `add` does not consume the capability of `file`.

Note that our system does not restrict references between objects inside the same region; this means that `this` and `file` can refer to each other in arbitrary ways. In the type system this is expressed by having field selections propagate guards: if `this` has type $\rho \triangleright$ `LogList`, then `this.elem` has type $\rho \triangleright$ `LogFile`. Since `file` is a peer of `this`, its type is $\rho \triangleright$ `LogFile`; therefore, assigning `file` to `elem` in the then-branch of the conditional expression is safe.

To verify the safety of calling `add` in the else-branch, we have to check that `next` and `file` have types guarded by the same capability. Moreover, this capability must be available. Since both conditions are true (the receiver of `isEmpty` is transient), the invocation type-checks.

We have introduced the `@transient` annotation to express the fact that a method maintains the uniqueness and accessibility of a receiver or parameter. The `@peer` annotation indicates that certain parameters are in the same (logical) region of the heap, which allows creating reference paths between them. Together, these annotations enable methods to mutate unique objects without destroying their uniqueness. In the following section we show how the disjoint regions of two unique objects can be merged.

4.3.4 Merging regions

Recall that the parameter of the `add` method shown above is marked as `@peer(this)`, which means that it must be in the same region as the receiver. However, when using `add` in the example of Figure 4.1, the `log` variable is separately-unique; this means it is contained in a region that is disjoint from the region of `logs`, the receiver of the method call. This is reflected in the types: `log` and `logs` have types $\rho \triangleright$ `LogFile` and $\delta \triangleright$ `LogList`, respectively, for some capabilities $\rho \neq \delta$. Therefore, the invocation `logs.add(log)` is not type-correct. What we need is a way

to merge the regions of `log` and `logs` prior to invoking `add`.[3]

In our system, regions are merged using a capture expression of the form t_1 `capturedBy` t_2. The arguments of `capturedBy` must have guarded types $\rho_1 \triangleright T_1$ and $\rho_2 \triangleright T_2$, respectively, such that ρ_1 is available. Our goal is to merge the regions ρ_1 and ρ_2 in a way that (still) permits separately-unique references into region ρ_2, while giving up the disjointness from region ρ_1. For this, `capturedBy` returns an alias of t_1, but with a type guarded by ρ_2 instead of ρ_1. This allows t_1 and t_2 to refer to each other subsequently. To satisfy the Capability Type Invariant (Definition 3), `capturedBy` consumes ρ_1. This ensures that t_1 can no longer be accessed under a type guarded by ρ_1. Therefore, it is safe to break the separation of t_1 and t_2 subsequently. Since ρ_2 is still available, it is possible for separately-unique variables to point into region ρ_2.

In the example, we use `capturedBy` to merge the regions of `log` and `logs` before invoking `add`:

```
logs.add(log capturedBy logs)
```

Note that `capturedBy` consumes the capability of `log`, while the capability of `logs` remains available. The result of `capturedBy` is an alias of `log` in the same region as `logs`. Therefore, the precondition of `add` (see above) is satisfied.

4.3.5 Unique fields

In the example of Figure 4.1, we made the simplifying assumption that the list of log files is stored in a local variable. This is not the case in the original program, where the log files are stored in a field of the class containing the `runTests` method. The main reason is that the lexical scope of `runTests` is too restrictive. It is simpler to create the log file in a method transitively called by `runTests`, at a point where more information about the test is available, and close to the point where the log file is actually used. Consequently, updating the list of log files inside `runTests` would be cumbersome, since it would require returning the log file back into the context of `runTests`. Keeping the `logs` in a field avoids passing it around using (extra) method parameters.

In our system, unique fields must be accessed using an expression of the form `swap`$(t_1.l, t_2)$; it returns the current value of the field $t_1.l$ and updates it with the new value t_2. The first argument must select a unique field. The second argument must be a unique object to be stored as the new value in the field. The object that `swap` returns is always unique, guarded by a fresh capability. The capability of the second argument is consumed, which makes it separately-unique.

[3] This is similar to changing the owner in systems based on ownership; here, ownership of an object is transferred from one (usually a special "unique") owner to another [29].

In our example, the list of log files can be maintained in a unique field `logFiles` as follows.

```
val logs: LogList @unique = swap(this.logFiles, null)
logs.add(log capturedBy logs)
swap(this.logFiles, logs)
```

First, we obtain the current value of the unique `logFiles` field, providing `null` as its new (dummy) value.[4] Then, we add the `log` file to `logs`, maintaining the uniqueness of `logs` as we discussed above. Finally, we use a second `swap` to update `logFiles` with the modified `logs`.

We now extend the alias invariant introduced above to unique fields. The only way to obtain a reference to an object stored in a unique field is to use the `swap` expression that we just introduced. Therefore, a property that holds for all (references to) objects returned by `swap` is an invariant of unique fields in our system. This allows us to formulate a unique fields invariant that is pleasingly simple.

Definition 4 (Unique Fields Invariant) *References returned by `swap` are separately-unique.*

4.4 Formalization

This section presents a formalization of our type system. To simplify the presentation of key ideas, we present our type system in the context of a core subset of Java. We add the `capturedBy` and `swap` expressions introduced in the previous section, and augment the type system with capabilities to enforce uniqueness and aliasing constraints. Section 4.8 discusses extensions that are important when integrating our system into full-featured languages. In Section 4.9 we report on an implementation for Scala.

Syntax Figure 4.3 shows the core language syntax. The syntax of programs, classes, terms, and expressions is standard, except for the `capturedBy` and `swap` expressions, which are new. A program consists of a sequence of class definitions followed by a single top-level term. (We use the common over-bar notation [79] for sequences.) Class definitions consist of declaring a single super-class followed by a body containing field and method definitions. Field definitions carry an additional modifier α, which indicates whether the field points to a unique object ($\alpha =$ `unique`), or not ($\alpha =$ `var`). Method definitions are extended with two additional capability annotations that we explain below. The term language is mostly

[4] Note that it is always safe to treat literals as unique values.

$P ::= \overline{cdef}\ t$	program
$cdef ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fld}\ \overline{meth}\}$	class
$fld ::= \alpha\ l : C$	field
$meth ::= \texttt{def}\ m[\delta](\overline{x : T}) : (T, \Delta) = e$	method
$t ::=$	terms
$\quad \texttt{let}\ x = e\ \texttt{in}\ t$	let binding
$\quad y.l := z$	field assignment
$\quad y$	variable
$e ::=$	expressions
$\quad \texttt{new}\ C(\overline{y})$	instance creation
$\quad y.l$	field selection
$\quad y.m(\overline{z})$	method invocation
$\quad y\ \texttt{capturedBy}\ z$	region capture
$\quad \texttt{swap}(y.l, z)$	unique field swap
$\quad t$	term

$C, D \in Classes$	$x, \overline{y}, \overline{z} \in Vars$	$T ::= \rho \triangleright C$
$l \in Fields$	$\alpha \in \{\texttt{var}, \texttt{unique}\}$	$\Delta ::= \cdot \mid \Delta \oplus \rho$
$m \in Methods$	$\rho \in Caps$	

Figure 4.3: Core language syntax

standard. However, note that terms are written in A-normal form [54]: all sub-expressions are variables and the result of each expression is immediately stored into a field or bound in a let. x, y, z are local variables and $x \neq \texttt{this}$.

Types and capabilities In our system, there are only guarded types and method types. Guarded types T are composed of an atomic capability ρ and the name of a class. ρ can be seen as the static representation of a region of the heap that contains all objects of a type guarded by ρ. A compound capability Δ is a set of atomic capabilities.

Method types are extended with capabilities δ and Δ. Roughly, Δ indicates which arguments become inaccessible at the call site when the method is invoked; δ is the capability of the result type if it is fresh. The annotations introduced in Section 4.3 correspond to method types in our core language as follows. A parameter x of type `C` marked as `@unique` or `@transient` is mapped to a guarded type $\rho \triangleright$ `C`, where ρ is distinct from the capabilities guarding other parameter types. If x is `@transient`, the method returns ρ, *i.e.*, $\rho \in \Delta$. If x is `@unique`, the method consumes ρ, *i.e.*, $\rho \notin \Delta$. A parameter y of type `D` marked as `@peer(x)` is mapped to a guarded type $\rho' \triangleright$ `D` if x's type is guarded by ρ'. `@peer` has no influence on Δ. The receiver (`this`) is treated like a parameter. The capability δ is distinct from

the capabilities of parameters. If the result type is marked `@unique`, its type is guarded by δ. We have $\delta \in \Delta$ only if the result type is guarded by δ, otherwise δ is unused. An unannotated method in the setting of Section 4.3 has the following type in our core language: the parameters (including `this`) and the result are guarded by the same capability ρ that the method does not consume ($\rho \in \Delta$).

Note that the mapping we just described establishes a precise correspondence: all types expressible in the core language can be expressed using the annotation system of Section 4.3. This ensures that the formal model is not more powerful than our implemented system.

4.4.1 Operational semantics

We formalize the dynamic semantics in the form of small-step reduction rules. Reduction rules are written in the form $H, V, R, t \longrightarrow H', V', R', t'$. Terms t are reduced in a context consisting of a heap H, a variable environment V, and a set of (dynamic) capabilities R. Figure 4.4 shows their syntax. A heap maps reference locations to class instances. An instance $C(\overline{r})$ stores location r_i in its i-th field. An environment maps variables to guarded reference locations $\beta \triangleright r$. Note that we do not model explicit stack frames. Instead, method invocations are "flattened" by renaming the method parameters before binding them to their argument values in the environment (as in LJ [118]).

We use the following notational conventions. $R \oplus \beta$ is a short hand for the disjoint union $R \uplus \{\beta\}$. We define $R \oplus \overline{\beta} := R \oplus \beta_1 \oplus \ldots \oplus \beta_n$ where $\overline{\beta} = \beta_1, \ldots, \beta_n$.

According to the grammar in Figure 4.3, expressions are always reduced in the context of a let-binding, except for field assignments. Each operand of an expression is a variable y that the environment maps to a guarded reference location $\beta \triangleright r$. Reducing an expression containing y requires β to be present in the set of capabilities. Since the environment is a flat list of variable bindings, let-bound variables must be alpha-renamable: $\texttt{let}\ x = e\ \texttt{in}\ t \equiv \texttt{let}\ x' = e\ \texttt{in}\ [x'/x]t$ where $x' \notin FV(t)$. (We omit the definition of the FV function to obtain the free variables of a term, since it is completely standard [105].)

The top-level term of a program is reduced in the initial configuration $(r \mapsto \texttt{Object}(\epsilon)), (\texttt{this} \mapsto \rho \triangleright r), \{\rho\}$ for some $r \in RefLocs, \rho \in Caps$. In the following we explain the reduction rules.

$$\frac{H, V, R, t_1 \longrightarrow H', V', R', t_1'}{\begin{array}{c} H, V, R, \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \\ \longrightarrow H', V', R', \texttt{let}\ x = t_1'\ \texttt{in}\ t_2 \end{array}} \quad \text{(R-LET)}$$

The congruence rule for `let` is standard. The term $\texttt{let}\ x = y\ \texttt{in}\ t$ is reduced in the obvious way.

$$
\begin{array}{lll}
H ::= & \emptyset \mid (H, r \mapsto C(\overline{r})) & \text{heap } (r \notin dom(H)) \\
V ::= & \emptyset \mid (V, y \mapsto \beta \triangleright r) & \text{envir. } (y \notin dom(V)) \\
R ::= & \emptyset \mid R \oplus \beta & \text{dynamic capability} \\
r \in & RefLocs & \text{reference location} \\
\beta \in & DynCaps & \text{atomic dyn. capability}
\end{array}
$$

Figure 4.4: Syntax for heaps, environments, and dynamic capabilities

$$
\frac{\begin{array}{c} V(y) = \delta \triangleright r \qquad \delta \in R \\ H(r) = C(\overline{r}) \end{array}}{\begin{array}{l} H, V, R, \texttt{let}\ x = y.l_i\ \texttt{in}\ t \\ \longrightarrow\ H, (V, x \mapsto \delta \triangleright r_i), R, t \end{array}}\ \text{(R-SELECT)}
\qquad
\frac{\begin{array}{c} V(y) = \delta \triangleright r \qquad V(z) = \delta \triangleright r' \\ H(r) = C(\overline{r}) \qquad \delta \in R \\ H' = H[r \mapsto C([r'/r_i]\overline{r})] \end{array}}{H, V, R, y.l_i := z\ \longrightarrow\ H', V, R, y}\ \text{(R-ASSIGN)}
$$

The result of selecting a field of a variable y is guarded by the same capability as y. Intuitively, this means that objects transitively reachable from y can only be accessed using variables guarded by the same capability as y. We make this intuition more precise in Section 4.4.2 where we formalize the separation invariant of Section 4.3. Assigning to a field requires the variable whose field is updated and the right-hand side to be guarded by the same capability. The heap changes in the standard way.

$$
\frac{V(\overline{y}) = \overline{\beta \triangleright r} \qquad H' = (H, r \mapsto C(\overline{r})) \qquad r \notin dom(H) \qquad \gamma \text{ fresh}}{H, V, R \oplus \overline{\beta}, \texttt{let}\ x = \texttt{new}\ C(\overline{y})\ \texttt{in}\ t\ \longrightarrow\ H', (V, x \mapsto \gamma \triangleright r), R \oplus \gamma, t}\ \text{(R-NEW)}
$$

Creating a new instance consumes the capabilities of the constructor arguments. This ensures that the arguments are effectively separately-unique. Consequently, it is safe to assign (some of) the arguments to unique fields of the new instance. In our core language, creating a new instance always yields a unique object. Therefore, the new let-bound variable that refers to it is guarded by a fresh capability.

$$
\frac{\begin{array}{c} V(y) = \beta_1 \triangleright r_1 \qquad H(r_1) = C_1(\_) \\ V(\overline{z}) = \beta_2 \triangleright r_2 \ldots \beta_n \triangleright r_n \qquad \overline{\beta} \subseteq R \\ mbody(m, C_1) = (\overline{x}, e) \end{array}}{\begin{array}{l} H, V, R, \texttt{let}\ x = y.m(\overline{z})\ \texttt{in}\ t \\ \longrightarrow\ H, (V, \overline{x \mapsto \beta \triangleright r}), R, \texttt{let}\ x = e\ \texttt{in}\ t \end{array}}\ \text{(R-INVOKE)}
$$

The rule for method invocation uses a standard auxiliary function $mbody$ to obtain the body of a method. It is defined as follows. Let $\texttt{def}\ m[\delta](\overline{x : T}) : (R, \Delta) = e$ be a method defined in the most direct super-class of C that defines m. Then $mbody(m, C) = (\overline{x}, e)$.

$$\frac{V(y) = \beta \triangleright r \qquad V(z) = \gamma \triangleright \_}{\begin{array}{c} H, V, R \oplus \beta \oplus \gamma, \texttt{let}\ x = y\ \texttt{capturedBy}\ z\ \texttt{in}\ t \\ \longrightarrow\ H, (V, x \mapsto \gamma \triangleright r), R \oplus \gamma, t \end{array}} \quad \text{(R-CAPTURE)}$$

Reducing a `capturedBy` term merges the regions of its two arguments y and z. It returns an alias of y guarded by the capability of z. This allows storing a reference to y in a field of z and vice versa (see rule R-ASSIGN above). By consuming y's capability, we make sure that objects that used to be in region β remain accessible only through variables guarded by γ, which is the capability of z. This enforces that all objects are accessible as part of at most one region at a time. (Recall that variables whose capabilities are not available cannot be accessed.)

$$\frac{\begin{array}{ccc} V(y) = \beta \triangleright r & H(r) = C(\overline{r}) & \gamma \text{ fresh} \\ V(z) = \beta' \triangleright r' & H' = H[r \mapsto C([r'/r_i]\overline{r})] & \end{array}}{\begin{array}{c} H, V, R \oplus \beta \oplus \beta', \texttt{let}\ x = \texttt{swap}(y.l_i, z)\ \texttt{in}\ t \\ \longrightarrow\ H', (V, x \mapsto \gamma \triangleright r_i), R \oplus \beta \oplus \gamma, t \end{array}} \quad \text{(R-SWAP)}$$

The only way to access a unique field is using `swap`. It mutates a unique field to point to a new object, and returns the field's previous value. The first argument must select a unique field such that the capability of the containing object is available. The second argument must be guarded by a different capability, which is consumed. This ensures that the new value and the object containing the unique field are separate prior to evaluating `swap`. `swap` returns the field's old value; the new let-bound variable that refers to it is guarded by a fresh capability, which allows treating the variable as separately-unique.

4.4.2 Type system

Well-formed programs

A program is well-formed if all its class definitions are well-formed. Classes and methods are well-formed according to the following rules. (We write ... to omit unimportant parts of code in a program P.)

$$\frac{\begin{array}{c} C \vdash \overline{meth} \\ D = \texttt{Object} \vee P(D) = \texttt{class}\ D\ \ldots \\ \forall\ (\texttt{def}\ m\ \ldots) \in \overline{meth}.\ override(m, C, D) \\ \forall \alpha\ l : E \in \overline{fld}.\ l \notin fields(D) \end{array}}{\vdash\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fld}\ \overline{meth}\}} \quad \text{(WF-CLASS)}$$

All well-formed class hierarchies are rooted in `Object`. All methods in a well-formed class definition are well-formed. We explain well-formed method overriding below. Fields may not be overridden; their names must be different from

the names of fields in super-classes. We use a standard function $fields(D)$ [79] to obtain all fields in D and super-classes of D.

$$\frac{\begin{array}{c}\overline{T} = \overline{\rho \triangleright C} \qquad \overline{x : T}\ ;\ \{\rho \mid \rho \in \overline{\rho}\} \vdash e : R\ ;\ \Delta \\ x_1 = \texttt{this} \qquad R = \delta' \triangleright D \qquad \delta' \in \Delta \\ \delta = \begin{cases} \delta' & \text{if } \delta' \notin \overline{\rho} \\ \text{fresh} & \text{otherwise} \end{cases}\end{array}}{C_1 \vdash \texttt{def}\ m[\delta](\overline{x : T}) : (R, \Delta) = e} \quad \text{(WF-METHOD)}$$

In a well-formed method definition that appears in class C_1, the first parameter is always `this` and its class type is C_1. The method body must be type-checkable in an environment that binds the parameters to their declared types, and that provides all capabilities of the parameter types. After type-checking the body, the capabilities in Δ must still be available. The result type of a method must be guarded by a capability in Δ. If the capability of the result type does not guard one of the parameter types, it is unknown in the caller's context. In this case we treat it as existentially quantified; the square brackets are used as its binder. If the capability of the result type guards one of the parameter types, the quantified capability is unused.

$$\frac{\begin{array}{c}mtype(m, D) \text{ not defined} \vee \\ (mtype(m, D) = \exists\delta.\ (\rho \triangleright D, \overline{T}) \to (R, \Delta) \wedge \\ mtype(m, C) = \exists\delta.\ (\rho \triangleright C, \overline{T}) \to (R, \Delta))\end{array}}{override(m, C, D)} \quad \text{(WF-OVERRIDE)}$$

A method defined in class C satisfies the rule for well-formed overriding if the super-class D does not define a method of the same name, or the method types differ only in the first `this` parameter.

Subclassing and subtypes Each program defines a class table, which defines the subtyping relation $<:$. In our system, $<:$ is identical to that of FJ [79], except for the following rule for guarded types, which is new. It expresses the fact that guarded types can only be sub-types if their capabilities are equal.

$$\frac{C <: D}{\rho \triangleright C <: \rho \triangleright D} \quad \text{(<:-CAP)}$$

Type assignment

Terms are type-checked using the judgement $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$. Γ maps variables to their types. The facts that Γ implies can be used arbitrarily often

in typing derivations. Δ and Δ' are capabilities, which may not be duplicated. As part of the typing derivation, capabilities may be consumed or generated. Δ' denotes the capabilities that are available after deriving the type of the term t. In a typing derivation where $\Delta' = \Delta$ we omit Δ' for brevity.

$$\frac{\Gamma(y) = \rho \triangleright C \qquad \rho \in \Delta}{\Gamma \,;\, \Delta \vdash y : \rho \triangleright C \,;\, \Delta} \text{(T-VAR)} \qquad \frac{\Gamma \,;\, \Delta \vdash y : \rho \triangleright C \quad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i \neq \texttt{unique}}{\Gamma \,;\, \Delta \vdash y.l_i : \rho \triangleright D_i \,;\, \Delta} \text{(T-SELECT)}$$

A variable is well-typed in Γ, Δ if Γ contains a binding for it, and Δ contains the capability of its (guarded) type. This ensures that the capabilities of variables occurring in a typing derivation are statically available. Selecting a field from a variable y of guarded type yields a type guarded by the same capability. The selected field must not be unique. Because of rule T-VAR, the capability of y must be available.

$$\frac{\Gamma \,;\, \Delta \vdash y : \rho \triangleright C \qquad \Gamma \,;\, \Delta \vdash z : \rho \triangleright D_i \quad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i \neq \texttt{unique}}{\Gamma \,;\, \Delta \vdash y.l_i := z : \rho \triangleright C \,;\, \Delta} \text{(T-ASSIGN)}$$

Assigning to a non-unique field of a variable y with guarded type $\rho \triangleright C$ requires also the right-hand side to be guarded by ρ. The term has the same type as y, which is the result of reducing the assignment (see Section 4.4.1).

$$\frac{\Gamma \,;\, \Delta \vdash \overline{y : \rho \triangleright D} \qquad \Delta = \Delta' \oplus \overline{\rho} \quad fields(C) = \overline{\alpha\ l : D} \qquad \rho' \text{ fresh}}{\Gamma \,;\, \Delta \vdash \texttt{new}\, C(\overline{y}) : \rho' \triangleright C \,;\, \Delta' \oplus \rho'} \text{(T-NEW)}$$

The rule for instance creation requires all constructor arguments to be guarded by distinct capabilities, which must be available. Intuitively, this means that the arguments are in mutually disjoint regions. Therefore, it is safe to assign them to unique fields of the new instance. By consuming the capabilities of the arguments, we ensure that there is no usable reference left that could point into the object graph rooted at the new instance; thus, we can assign a type guarded by a fresh capability ρ' to the new instance and make ρ' available to the context. Note that we can relax this rule for initializing non-unique fields: multiple non-unique fields may be guarded by the same capability. (See Section 4.8 for a discussion in the context of nested classes.)

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash\ y : \rho_1 \triangleright D_1 \\ \Gamma\ ;\ \Delta \vdash\ z_{i-1} : \rho_i \triangleright D_i,\ i = 2..n \\ mtype(m, D_1) = \exists\delta.\ \overline{\delta \triangleright D} \to (R, \Delta_m) \\ \sigma = \overline{\delta \mapsto \rho} \circ \delta \mapsto \rho \text{ injective} \qquad \rho \text{ fresh} \\ \Delta = \Delta' \uplus \{\rho \mid \rho \in \overline{\rho}\}\end{array}}{\Gamma\ ;\ \Delta \vdash\ y.m(\overline{z}) : \sigma R\ ;\ \sigma\Delta_m \oplus \Delta'} \quad \text{(T-INVOKE)}$$

In the rule for method invocations, the capabilities of all arguments must be available in Δ. We look up the method type based on the static type of the receiver. The capabilities in method types are abstract, and have to be instantiated with concrete ones. To satisfy the pre-condition of the method, there must be a substitution that maps the capabilities of the formal parameters to the capabilities of the arguments. Importantly, the substitution must be *injective* to prevent mapping different formal capabilities to the same argument capability; this would mean that the requirement to have two different formal capabilities could be met using only a single argument capability, which would amount to duplicating that capability. In our system, capabilities may never be duplicated. The resulting set of capabilities is composed of the capabilities provided by the method after applying the substitution ($\sigma\Delta_m$) and those capabilities Δ' that were provided by the context, but that were not required by the method.

$$\frac{\Gamma\ ;\ \Delta \vdash\ y : \rho \triangleright C \qquad \Gamma\ ;\ \Delta \vdash\ z : \rho' \triangleright C' \qquad \Delta = \Delta' \oplus \rho}{\Gamma\ ;\ \Delta \vdash\ y\ \texttt{capturedBy}\ z : \rho' \triangleright C\ ;\ \Delta'} \quad \text{(T-CAPTURE)}$$

Typing the expression y `capturedBy` z requires the capabilities of y and z to be present (this follows from rule T-VAR, see above). The capability of the first argument is consumed, thereby making all variables pointing into its region inaccessible. The result has the same class type as y, but guarded by the capability of z. Essentially, `capturedBy` casts its first argument from its current region to the region of the second argument; in Section 4.5 we prove that the cast can never fail at run-time.

$$\frac{\begin{array}{cc}\Gamma\ ;\ \Delta \vdash\ y : \rho \triangleright C & \Gamma\ ;\ \Delta \vdash\ z : \rho' \triangleright D_i \\ fields(C) = \overline{\alpha\ l : D} & \alpha_i = \texttt{unique} \\ \Delta = \Delta' \oplus \rho' & \rho'' \text{ fresh}\end{array}}{\Gamma\ ;\ \Delta \vdash\ \texttt{swap}(y.l_i, z) : \rho'' \triangleright D_i\ ;\ \Delta' \oplus \rho''} \quad \text{(T-SWAP)}$$

The first argument of `swap` must select a unique field. Recalling the dynamic semantics, `swap` returns the current value of this field, and assigns the value of z to it. Therefore, the field must have the same class type as z (possibly using subsumption, see below). The arguments must be guarded by two different capabilities, which must be present in Δ. (Again, ρ is present because of rule T-VAR.)

This means that the arguments point to disjoint regions in the heap. By consuming the capability of z, we ensure that it is separately-unique. Since the reference returned by `swap` is unique, the result is guarded by a fresh capability.

$$\frac{\Gamma\ ;\ \Delta \vdash e : T\ ;\ \Delta' \qquad \Gamma, x : T\ ;\ \Delta' \vdash t : T'\ ;\ \Delta''}{\Gamma\ ;\ \Delta \vdash \texttt{let}\ x = e\ \texttt{in}\ t : T'\ ;\ \Delta''}\ \text{(T-LET)} \qquad \frac{\Gamma\ ;\ \Delta \vdash e : T'\ ;\ \Delta' \qquad T' <: T}{\Gamma\ ;\ \Delta \vdash e : T\ ;\ \Delta'}\ \text{(T-SUB)}$$

The rule for let is standard, except for the fact that type derivations may change the set of capabilities. The subsumption rule can be applied wherever the type of an expression is derived. In particular, deriving the type of variables is subject to subsumption.

Well-formedness

We require terms to be reduced in well-formed configurations. A well-formed configuration must satisfy at least the following two invariants, which are central to the soundness of our system. The first invariant expresses the fact that two accessible variables guarded by different capabilities do not share a common reachable object. It is based on a predicate $separate$, which is defined as follows.

Definition 1 (Separation) *Two reference locations r and r' are separate in heap H, written $separate(H, r, r')$,* iff
$\forall q, q' \in dom(H).\ reachable(H, r, q) \wedge reachable(H, r', q') \Rightarrow q \neq q'$

The $separate$ predicate expresses the fact that two references point to two disjoint object graphs. Based on $separate$ we define a Separation Invariant on local variables.

Definition 2 (Separation Invariant) *A configuration V, H, R satisfies the Separation Invariant, written $separation(V, H, R)$,* iff
$\forall\ (x \mapsto \delta \triangleright r), (x' \mapsto \delta' \triangleright r') \in V.$
$(\delta \neq \delta' \wedge \{\delta, \delta'\} \subseteq R \Rightarrow separate(H, r, r'))$

Note that we can only conclude that the two variables are separate if both capabilities are present. In particular, capturing a variable y does not violate the invariant even though it creates an alias of y guarded by a different capability. The reason is that `capturedBy` consumes y's capability, thereby making it inaccessible. Therefore, the invariant continues to hold for accessible variables, that is, variables whose capabilities are present.

Definition 3 (Unique Fields Invariant) *A configuration V, H, R satisfies the Unique Fields Invariant, written $uniqFlds(V, H, R)$,* iff

$\forall\ (x \mapsto \delta \triangleright r) \in V.\ H(q) = C(\overline{p}) \Rightarrow \forall i \in uniqInd(C).$
$\delta \in R \wedge reachable(H, p_i, r') \Rightarrow domedge(H, q, i, r, r')$

The unique fields invariant says that all reference paths from a variable x to some object r' reachable from a unique field must "go through" that unique field. The $reachable$ and $domedge$ predicates are based on the following definition of reference paths.

$$\frac{r \in dom(H)}{[r] \in path(H, r, r)} \qquad \frac{H(r) = C(\overline{p}) \quad \exists i.\ P \in path(H, p_i, r')}{r :: P \in path(H, r, r')} \qquad \frac{path(H, r, r') \neq \emptyset}{reachable(H, r, r')}$$

Basically, a reference path is a sequence of reference locations, where each reference (except the first) is stored in a field of the preceding location. The definition of $domedge$ is as follows.

$domedge(H, q, i, r, r') \Leftrightarrow$
$\forall P \in path(H, r, r').\ P = r \ldots q, p_i, \ldots r'$ where $H(q) = C(\overline{p})$

This predicate expresses the fact that all paths from r to r' must contain the sequence q, p_i, which corresponds to selecting the i-th (unique) field of object q.

$$\frac{\begin{array}{c}\Sigma \vdash H \\ \Gamma\ ;\ \Delta\ ;\ \Sigma \vdash V\ ;\ R \\ separation(V, H, R) \\ uniqFlds(V, H, R)\end{array}}{\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R} \quad \text{(WF-CONFIG)}$$

Aside from the separation and unique fields invariants, well-formed configurations must have well-formed environments and heaps.

$$\frac{\Sigma \vdash H \qquad \Sigma(r) = C \qquad fields(C) = \overline{\alpha\ l : D} \qquad \Sigma \vdash \overline{p : D}}{\Sigma \vdash (H, r \mapsto C(\overline{p}))}\ \text{(WF-HEAP)} \qquad \frac{\Sigma(r) = D \qquad D <: C}{\Sigma \vdash r : C}\ \text{(HEAP-TYPE)}$$

The rule for well-formed heaps is completely standard: the heap typing Σ must agree with the heap H on the type of each class instance. Moreover, the types of instances referred to from its fields must be compatible with their declared types using the HEAP-TYPE rule.

$$\frac{\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash V\ ;\ R \qquad \Sigma \vdash r : C \qquad \rho \in \Delta \text{ iff } \beta \in R}{(\Gamma, y : \rho \triangleright C)\ ;\ \Delta\ ;\ \Sigma \vdash (V, y \mapsto \beta \triangleright r)\ ;\ R} \quad \text{(WF-ENV)}$$

In the rule for well-formed environments we require the type environment Γ to agree with the heap typing Σ on the class type of instances referred to from variables. This rule also contains the key to relating static and dynamic capabilities: the static capability of a variable is contained in the set of static capabilities if and only if its dynamic capability in the environment is contained in the set of dynamic capabilities. This precise correspondence allows us to prove that the reduction of a well-typed term will never get stuck because of missing capabilities (see Section 4.5).

4.5 Soundness

In this section we present the main soundness result for the type system introduced in Section 4.4. We prove type soundness using the standard syntactic approach of preservation plus progress [135]. A complete proof of soundness appears in Appendix A.

In a first step, we prove a preservation theorem: it states that the reduction of a well-typed term in a well-formed context preserves the term's type. Moreover, the resulting context (heap, environment, and capabilities) is well-formed with respect to a new type environment, static capabilities, and heap typing.

Theorem 1 (Preservation) *If*

- $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$
- $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$
- $H, V, R, t \longrightarrow H', V', R', t'$

then there are $\Gamma' \supseteq \Gamma$, Δ'', *and* $\Sigma' \supseteq \Sigma$ *such that*

- $\Gamma'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
- $\Gamma'\ ;\ \Delta''\ ;\ \Sigma' \vdash H'\ ;\ V'\ ;\ R'$

This theorem guarantees that reduction preserves the separation and unique fields invariants that we introduced in Section 4.4.2. These invariants are implied by the well-formedness of the result context H', V', R'. Also note that the new type environment Γ' and the new heap typing Σ' are super sets of their counterparts Γ

and Σ, respectively. This means that merging two regions does not require strong type updates in our system.

In a second step, we prove a progress theorem, which guarantees that a well-typed term can be reduced in a well-formed context, unless it is a value. Variables are the only values in our language.

Theorem 2 (Progress) *If* $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$ *and* $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$*, then either* $t = y$*, or there is a reduction* $H, V, R, t \longrightarrow H', V', R', t'$*.*

The progress theorem makes sure that the reduction of a term does not get stuck, because of missing capabilities; that is, if a term type-checks, all required capabilities will be available during its reduction. Soundness of the type system follows from Theorem 1 and Theorem 2.

We can formulate a uniqueness theorem as a corollary of preservation and progress. Formally, separate uniqueness is defined as follows:

Definition 4 (Separate Uniqueness) *A variable* $(y \mapsto \beta \triangleright r) \in V$ *such that* $\beta \in R$ *is separately-unique in configuration* $H, V, R,$ `let` $x = t$ `in` t' iff
$\forall\ (y' \mapsto \beta' \triangleright r') \in V.$
$(\neg separate(H, r, r') \wedge H, V, R, t \longrightarrow^* H', V', R', e') \Rightarrow \beta' \notin R'$

Intuitively, the definition says that the capabilities of aliases of a variable y are unavailable when reducing t' if y is separately-unique in t. By Theorem 2 and the reduction rules, none of y's aliases are accessed after the reduction of t, since $\beta' \notin R'$.

The following corollary guarantees that a variable passed as an argument to a method expecting a unique parameter is separately-unique; this means that all variables that are still accessible after the invocation are separate from the argument.

Corollary 1 (Uniqueness) *If*

- $\Gamma\ ;\ \Delta \vdash$ `let` $x = t$ `in` $t' : T\ ;\ \Delta'$ *where* $t = y.m(\overline{z}) \wedge \Gamma(y) = \_ \triangleright C$
- $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$
- $mtype(m, C) = \exists \delta.\ \overline{\delta \triangleright D} \rightarrow (T_R, \Delta_m)$ *where* $\delta_i \notin \Delta_m$

then z_i *is separately-unique in* $H, V, R,$ `let` $x = t$ `in` t'*.*

4.6 Immutable Types

In this section we introduce a family of immutable types. We use a very simple notion of immutability: objects of immutable type cannot be mutated after they have been constructed. This means that their fields can only be assigned once as part of object creation. Furthermore, we require the types of the fields of an immutable type to be immutable, too. This means that the graph of all objects reachable from an immutable instance (*i.e.*, an instance of an immutable type) is not changeable after it has been constructed (*i.e.*, after reducing an instance creation expression).

Immutable objects are useful in a number of ways. We point out two use cases that are particularly important for our purposes. First, in the context of unique references, they enable more flexible aliasing while maintaining strong invariants with respect to mutation through unique references. For instance, several different unique objects may point to the same immutable object. The fact that the immutable object is shared is not visible to the clients using the unique references. Second, immutable objects may be shared among concurrent processes without restrictions, since data races cannot occur.

4.6.1 Immutable classes

In the following we introduce a refinement of the class types of Section 4.4.2. We split the vocabulary of classes into immutable classes $\mathcal{I}$ and regular classes $\mathcal{C}$. By adapting the reduction, typing, and well-formedness rules, we ensure instances of classes in $\mathcal{I}$ are (deeply) immutable.

4.6.2 Reduction

In a first step, we adapt the operational semantics of Section 4.4.1 to enforce that (the field values of) instances of classes in $\mathcal{I}$ cannot be modified during reduction.

$$\frac{\begin{array}{c} V(y) = \delta \triangleright r \qquad V(z) = \delta \triangleright r' \\ H(r) = C(\overline{r}) \qquad C \notin \mathcal{I} \qquad \delta \in R \\ H' = H[r \mapsto C([r'/r_i]\overline{r})] \end{array}}{H, V, R, y.l_i := z \;\longrightarrow\; H', V, R, y} \qquad \text{(R-ASSIGN)}$$

The next step is to adapt the reduction rules to allow immutable objects to be shared by multiple regions. We do this by never consuming the capabilities that guard immutable objects. This means, an immutable object may be captured by multiple regions, and, therefore, be referred to from multiple regions.

Note that it is not sufficient to introduce a rule that allows duplicating capabilities guarding immutable objects. The reason is that an immutable object may be pointed to from a mutable object in the same region. Therefore, duplicating the capability of an immutable object could allow consuming a regular unique object more than once, which is unsound.

$$\frac{\begin{array}{c} V(\overline{y}) = \overline{\beta \triangleright r} \qquad H' = (H, r \mapsto C(\overline{r})) \qquad r \notin dom(H) \qquad \gamma \text{ fresh} \\ R' = \{\beta_i \in \overline{\beta} \mid immutable(r_i, H)\} \end{array}}{H, V, R \oplus \overline{\beta}, \texttt{let}\ x = \texttt{new}\ C(\overline{y})\ \texttt{in}\ t \ \longrightarrow\ H', (V, x \mapsto \gamma \triangleright r), R \oplus R' \oplus \gamma, t} \quad \text{(R-NEW)}$$

In the rule for instance creation we do not consume capabilities of arguments that have an immutable (run-time) type. This allows an immutable object to become part of the region of the newly created instance, while allowing it to be captured by other regions subsequently. The $immutable$ predicate is defined as follows.

Definition 5 (Immutable Location) *A reference location $r \in dom(H)$ is immutable in heap H, written $immutable(r, H)$,* iff
$\exists C \in \mathcal{I}, \{r_i \mid r_i \in \bar{r}\} \subseteq dom(H).\ H(r) = C(\bar{r})$

$$\frac{\begin{array}{c} V(y) = \beta \triangleright r \qquad V(z) = \gamma \triangleright \_ \\ R' = \{\beta \mid immutable(r, H)\} \end{array}}{\begin{array}{c} H, V, R \oplus \beta \oplus \gamma, \texttt{let}\ x = y\ \texttt{capturedBy}\ z\ \texttt{in}\ t \\ \longrightarrow\ H, (V, x \mapsto \gamma \triangleright r), R \oplus R' \oplus \gamma, t \end{array}} \quad \text{(R-CAPTURE)}$$

In the rule for `capturedBy`, we do not consume the capability of the captured object if it is immutable. Therefore, multiple regions can capture the same immutable object.

$$\frac{\begin{array}{c} V(y) = \beta \triangleright r \qquad H(r) = C(\overline{r}) \qquad \gamma \text{ fresh} \\ V(z) = \beta' \triangleright r' \qquad H' = H[r \mapsto C([r'/r_i]\overline{r})] \\ R' = \{\beta' \mid immutable(r', H)\} \end{array}}{\begin{array}{c} H, V, R \oplus \beta \oplus \beta', \texttt{let}\ x = \texttt{swap}(y.l_i, z)\ \texttt{in}\ t \\ \longrightarrow\ H', (V, x \mapsto \gamma \triangleright r_i), R \oplus R' \oplus \beta \oplus \gamma, t \end{array}} \quad \text{(R-SWAP)}$$

The rule for `swap` is adapted in a way analogous to rule R-CAPTURE; the capability of z is not consumed if it refers to an immutable object.

4.6.3 Typing rules

With the modified reduction rules, it is impossible to prove a progress theorem using the existing typing rules. We have to update the typing rules accordingly to reflect the changes in the operational semantics.

$$\frac{\Gamma\ ;\ \Delta \vdash\ y : \rho \triangleright C \qquad C \notin \mathcal{I} \qquad \Gamma\ ;\ \Delta \vdash\ z : \rho \triangleright D_i \qquad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i \neq\ \texttt{unique}}{\Gamma\ ;\ \Delta \vdash\ y.l_i := z : \rho \triangleright C\ ;\ \Delta} \quad \text{(T-ASSIGN)}$$

The typing rules T-NEW, T-CAPTURE, and T-SWAP have to be adapted not to consume the capabilities of immutable objects.

$$\frac{\Gamma\ ;\ \Delta \vdash\ \overline{y : \rho \triangleright D} \qquad \Delta = \Delta' \oplus \overline{\rho} \qquad fields(C) = \overline{\alpha\ l : D} \qquad \rho' \text{ fresh} \qquad \Delta'' = \{\rho_i \in \overline{\rho} \mid D_i \in I\}}{\Gamma\ ;\ \Delta \vdash\ \texttt{new}\ C(\overline{y}) : \rho' \triangleright C\ ;\ \Delta' \oplus \Delta'' \oplus \rho'} \quad \text{(T-NEW)}$$

$$\frac{\Gamma\ ;\ \Delta \vdash\ y : \rho \triangleright C \qquad \Gamma\ ;\ \Delta \vdash\ z : \rho' \triangleright C' \qquad \Delta = \begin{cases} \Delta' & \text{if } C \in I \\ \Delta' \oplus \rho & \text{otherwise} \end{cases}}{\Gamma\ ;\ \Delta \vdash\ y\ \texttt{capturedBy}\ z : \rho' \triangleright C\ ;\ \Delta'} \quad \text{(T-CAPTURE)}$$

$$\frac{\Gamma\ ;\ \Delta \vdash\ y : \rho \triangleright C \qquad \Gamma\ ;\ \Delta \vdash\ z : \rho' \triangleright D_i \qquad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i = \texttt{unique} \qquad \rho'' \text{ fresh} \qquad \Delta = \begin{cases} \Delta' & \text{if } D_i \in I \\ \Delta' \oplus \rho' & \text{otherwise} \end{cases}}{\Gamma\ ;\ \Delta \vdash\ \texttt{swap}(y.l_i, z) : \rho'' \triangleright D_i\ ;\ \Delta' \oplus \rho''} \quad \text{(T-SWAP)}$$

4.6.4 Well-formedness

To ensure that immutable objects are deeply immutable, and that this is preserved by sub-classing, we impose a number of well-formedness conditions on immutable classes; these conditions are embodied in an extended rule for class well-formedness:

$$\frac{C \vdash\ \overline{meth} \qquad D = \texttt{Object} \vee P(D) = \texttt{class}\ D\ \ldots \qquad \forall\ (\texttt{def}\ m\ \ldots) \in \overline{meth}.\ override(m, C, D) \qquad \forall \alpha\ l : E \in \overline{fld}.\ l \notin fields(D) \wedge (C \in I \Rightarrow E \in I) \qquad D \in I \Rightarrow C \in I}{\vdash\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fld}\ \overline{meth}\}} \quad \text{(WF-CLASS)}$$

First, fields of an immutable class must have an immutable class type; this ensures the objects in the graph (transitively) reachable from a field of an immutable

object are immutable. To ensure sub-typing preserves immutability, we require subclasses of immutable classes to be immutable. Conversely, a regular class may have fields of immutable class type.

Separation and uniqueness

Using the reduction and typing rules of the previous sections, well-typed programs no longer preserve the separation invariant of Section 4.4.2 during reduction. The reason is, of course, that the same immutable object may become reachable from two different regions. Therefore, we have to modify the separation invariant to take immutable objects into account. To do this, we first define a predicate $sep - imm$ that refines the predicate $separate$.

Definition 6 (Separate Immutable) $sep - imm(H, r, r') \Leftrightarrow$
$\forall q, q' \in dom(H).\ reachable(H, r, q) \wedge reachable(H, r', q') \Rightarrow$
$q \neq q' \vee immutable(q, H)$

Based on $sep - imm$ we can now define a refined separation invariant:

Definition 7 (Separation Immutability Invariant)
A configuration V, H, R satisfies the Separation Immutability Invariant, written $separation - imm(V, H, R)$, iff
$\forall\ (x \mapsto \delta \triangleright r), (x' \mapsto \delta' \triangleright r') \in V.$
$(\delta \neq \delta' \wedge \{\delta, \delta'\} \subseteq R \Rightarrow sep - imm(H, r, r'))$

Compared to Definition 2 we only replaced the use of $separate$ with $sep - imm$.

Analogously, we have to update the invariant for unique fields to allow sharing of immutable objects.

Definition 8 (Unique Fields Immutability Invariant)
A configuration V, H, R satisfies the Unique Fields Immutability Invariant, written $uniqFlds - imm(V, H, R)$, iff
$\forall\ (x \mapsto \delta \triangleright r) \in V.\ H(q) = C(\overline{p}) \Rightarrow \forall i \in uniqInd(C).$
$\delta \in R \wedge reachable(H, p_i, r') \Rightarrow (domedge(H, q, i, r, r') \vee immutable(r', H))$

Basically, the modified invariant says that the edge (q, p_i) (following the pointer stored in the i-th field of the object at location q) does not have to be dominating on all paths from r to r' if the object at location r' is immutable.

To ensure immutable objects remain deeply immutable during reduction, we add the following invariant.

Definition 9 (Deep Immutability Invariant)
A heap H satisfies the Deep Immutability Invariant, written $deep - imm(H)$, iff
$\forall\, r \in dom(H).\ immutable(r, H) \Rightarrow$
$\forall\, r' \in dom(H).\ reachable(H, r, r') \Rightarrow immutable(r', H)$

The $separation - imm$, $uniqFlds - imm$, and $deep - imm$ invariants must be preserved when reducing a well-typed term. Therefore, we include them in a modified WF-CONFIG rule. The $separation - imm$ and $uniqFlds - imm$ invariants replace their immutability-oblivious counterparts.

$$\frac{\begin{array}{c}\Sigma \vdash H \qquad \Gamma\ ;\ \Delta\ ;\ \Sigma \vdash V\ ;\ R \\ separation - imm(V, H, R) \\ uniqFlds - imm(V, H, R) \\ deep - imm(H)\end{array}}{\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R} \qquad \text{(WF-CONFIG)}$$

4.6.5 Soundness

In this section we sketch a proof explaining why the addition of immutable types as presented above is sound. We restrict ourselves to the preservation proof. Our preservation proof in Section A.2 uses structural induction on the shape of terms with a case analysis of the reduction rule that is applied. In the following we provide sketches for selected proof cases. In each case we consider a single reduction step $H, V, R, t \longrightarrow H', V', R', t'$.

- *Case R-Capture.* This case demonstrates why the $deep - imm$ invariant is necessary to establish that the successor configuration satisfies the extended separation invariant $separation - imm$. Consider a configuration H, V, R where a local variable y points to an immutable object r guarded by δ (*i.e.*, $V(y) = \delta \triangleright r$). Moreover, let r point to some mutable object r'. Reducing the expression $\texttt{let}\ x = y\ \texttt{capturedBy}\ z\ \texttt{in}\ t$ would create a binding for x in the successor environment V' such that $V'(x) = \delta' \triangleright r$, assuming $V(z) = \delta' \triangleright \_$ ($\delta \neq \delta'$). In this case, $separation - imm$ does not hold for V' and H, since the mutable object r' can be reached from both x and y, but $\delta \neq \delta'$ (we can easily satisfy that $\{\delta, \delta'\} \subseteq R$). Requiring $deep - imm$ to hold in H avoids this problem by enforcing that r' is immutable.

- *Case R-New.* This case demonstrates why the modified WF-CLASS rule is necessary to establish $deep - imm$ in the successor heap H'. Without the requirement of rule WF-CLASS that fields of immutable classes have immutable class types, the expression $\texttt{let}\ x = \texttt{new}\ C(y)$ would be well-typed

$t ::=$		terms
	$\ldots$	
	y `!` z	message send
$e ::=$		expressions
	$\ldots$	
	`receive`$[C]$	message receive
	`actor` C	actor creation

Figure 4.5: Language syntax extension for concurrent programming with actors

in an environment Γ, Δ such that $\Gamma\ ;\ \Delta \vdash\ y : \rho \triangleright D$, $fields(C) = \texttt{val}\ l : D$, and $D \notin \mathcal{I}$, but $C \in \mathcal{I}$. Therefore, we could reduce the expression in an environment H, V, R where $V(y) = \delta \triangleright r$ and $\neg immutable(r, H)$. As a result, $deep - imm(H')$ would not be satisfied, since $H'(r') = C(r)$ and $immutable(r', H')$.

4.7 Concurrency

In this section we extend our language to include constructs for concurrent programming with actors. The goal of this section is to show that the type system introduced in Section 4.4 can be used to enforce actor isolation in the presence of a shared heap and efficient, by-reference message passing.

4.7.1 Syntax

The syntax extensions of our sequential core language are summarized in Figure 4.5. We add primitives for sending and receiving messages, as well as an expression for creating actors. The term y `!` z asynchronously sends (the location of) z to the actor y. The expression `receive`$[C]$ tries to remove a message of type C from the current actor's mailbox. This simplifies the more general `receive` expressions of Erlang and Scala, which allow arbitrary message patterns and choice. If there is no message of type C in the actor's mailbox, a `receive`$[C]$ expression cannot be reduced. Actors are created using the expression `actor` C. Here, C is a subclass of `Actor`; the `Actor` class defines a single `act` method whose body the new actor evaluates.

Figure 4.6 shows an example that uses the new terms and expressions. The program defines two classes `Adder` and `Client`, whose `act` methods define the behavior of actors created using those classes. An actor of type `Adder` receives two objects of type `Int` (we assume the existence of a standard `Int` class type), adds them, and sends the result to a `sender` actor. A `Client` actor creates a

```
class Adder extends Actor {
  def act(self: Adder) =
    let x = receive[Int] in
    let y = receive[Int] in
    let z = x.plus(y) in
    let sender = receive[Actor] in
    sender ! z
}
class Client extends Actor {
  def act(self: Client) =
    let adder = actor Adder in
    let x = 40 in
    let y = 2 in
    adder ! x; adder ! y; adder ! self;
    let res = receive[Int]
}
let c = actor Client in c
```

Figure 4.6: Concurrent program showing the use of `actor`, `receive`, and the send operator (`!`).

new `Adder` actor using the expression `actor Adder`. Using three asynchronous messages it sends two integers and a reference to itself, respectively, to the `adder` actor (following common practice we write t `;` t' for `let` $x = t$ `in` t' where x does not occur free in t'). Finally, the `Client` actor receives the integer result from `adder`.

4.7.2 Sharing and immutability

The type system for sequential programs introduced in Section 4.4 enforces that variables guarded by different capabilities are separate if their capabilities are available. In a concurrent setting this separation invariant must be extended to multiple actors. Intuitively, we would like to enforce that objects accessible by one actor are separate from objects accessible by any other actor that is active at the same time. However, such a naive rule would be too restrictive for objects of type `Actor`. When creating a new actor, both the creating actor and the created actor obtain an accessible reference to the new actor; both actors must be able to send messages to the new actor to enable dynamic changes in the communication topology, which is fundamental to the actor model of concurrency [2].

We enable actor locations to be shared by making the `Actor` class immutable

(`Actor` $\in \mathcal{I}$). Therefore, by rule WF-CLASS of Section 4.6.4, all subclasses of `Actor` are immutable. In our system, immutable objects may be shared freely among actors.

4.7.3 Operational semantics

The configuration of an actor-based program consists of a shared heap H, and an unordered "soup" of actors $\mathcal{A}$. An actor is represented as a pair consisting of an execution state and a mailbox containing messages that have been sent to the actor, but not yet processed. Since messages can be arbitrary objects (including actors), we represent the mailbox as a set of reference locations. An actor's execution state $S = \langle V, R, t\rangle$ consists of an environment V, a set of capabilities R, and a continuation term t. Formally, an actor is written $A = (S, M)_r$, where M is its mailbox and r is the reference location of an object of type `Actor`.

$$\frac{\begin{array}{c} r \notin dom(H) \qquad H' = (H, r \mapsto C(\epsilon)) \qquad \rho, \rho' \text{ fresh} \qquad C \in \mathcal{I} \\ V' = (V, x \mapsto \rho \triangleright r) \qquad mbody(\texttt{act}, C) = (\texttt{this}, e) \\ A = (\langle \texttt{this} \mapsto \rho' \triangleright r, \{\rho'\}, \texttt{let } y = e \texttt{ in } y\rangle, \emptyset)_r \end{array}}{\begin{array}{c} H \,;\, \{(\langle V, R, \texttt{let } x = \texttt{actor } C \texttt{ in } t\rangle, \_)\} \cup \mathcal{A} \\ \longrightarrow\; H' \,;\, \{(\langle V', R \oplus \rho, t\rangle, \_)\} \cup \mathcal{A} \cup \{A\} \end{array}} \text{(R-ACTOR)}$$

Reducing an `actor` expression creates a new actor based on a class C that contains a method `act`; the body of this method defines the continuation term of the newly created actor. Since actors are objects, `actor` returns the reference location of the new actor. Note that both the creating and the created actor need access to the new actor object; this is allowed since C is immutable. The fresh capability ρ' is available when reducing the continuation of the new actor.

$$\frac{\begin{array}{c} V(y) = \beta \triangleright r \qquad V(z) = \beta' \triangleright r' \qquad H(r') = C(\_) \\ R = R' \oplus \beta' \qquad R'' = R' \cup \{\beta' \mid C \in \mathcal{I}\} \end{array}}{\begin{array}{c} H \,;\, \{(\langle V, R, y \texttt{ ! } z\rangle, \_), (\_, M)_r\} \cup \mathcal{A} \\ \longrightarrow\; H \,;\, \{(\langle V, R'', y\rangle, \_), (\_, \{r'\} \cup M)_r\} \cup \mathcal{A} \end{array}} \text{(R-SEND)}$$

Sending a message to y requires that the location of y corresponds to an existing actor in the actor soup. The capability of the object z to be sent must be available. It is not consumed, if z points to an immutable object. The location of z (without its guard) is added to the mailbox of the target actor. The send expression reduces to the (variable of the) target actor.

$$\frac{V' = (V, x \mapsto \beta \triangleright r) \qquad H(r) = C(\_) \qquad \beta \text{ fresh}}{\begin{array}{c} H \,;\, \{(\langle V, R, \texttt{let } x = \texttt{receive}[C] \texttt{ in } t\rangle, \{r\} \cup M)\} \cup \mathcal{A} \\ \longrightarrow\; H \,;\, \{(\langle V', R \oplus \beta, t\rangle, M)\} \cup \mathcal{A} \end{array}} \text{(R-RECV)}$$

Reducing a `receive`$[C]$ expression removes an object of class type C from the current actor's mailbox. The objects in an actor's mailbox are guaranteed to be immutable or separate from all other messages and accessible variables (see below). Therefore, it is sound to guard the removed location by a fresh capability in the receiver's environment.

The sequential reduction rules of Section 4.4.1 are integrated using the following "step" rule, which reduces a single actor; its mailbox remains unchanged.

$$\frac{H, V, R, t \longrightarrow H', V', R', t'}{\begin{array}{c} H \;;\; \{(\langle V, R, t\rangle, M)\} \cup \mathcal{A} \\ \longrightarrow\; H' \;;\; \{(\langle V', R', t'\rangle, M)\} \cup \mathcal{A} \end{array}} \quad \text{(R-STEP)}$$

4.7.4 Typing

We extend the sequential typing rules to include the new `actor`, send (`!`), and `receive` expressions.

$$\frac{C <: \texttt{Actor} \qquad \rho \text{ fresh}}{\Gamma \;;\; \Delta \vdash \texttt{actor}\, C : \rho \triangleright \texttt{Actor} \;;\; \Delta \oplus \rho} \quad \text{(T-ACTOR)}$$

Creating a new actor using `actor` requires C to be a subclass of `Actor`; C is immutable, because `Actor` is (see Section 4.6.3). Therefore, it is sound to guard the new `Actor` instance, which `actor` returns, by a fresh capability.

$$\frac{\begin{array}{c} \Gamma \;;\; \Delta \vdash y : \rho \triangleright \texttt{Actor} \qquad \Gamma \;;\; \Delta \vdash z : \rho' \triangleright C \\ \Delta = \Delta' \oplus \{\rho' \mid C \notin \mathcal{I}\} \end{array}}{\Gamma \;;\; \Delta \vdash y \;\texttt{!}\; z : \rho \triangleright \texttt{Actor} \;;\; \Delta'} \quad \text{(T-SEND)}$$

Sending a message requires the receiver y to have class type `Actor`. The message object z to be sent must be guarded by a capability that is available; it is not consumed if the class type of z is immutable. The result has the type of the target actor (recall the corresponding reduction rule (R-SEND), which reduces the send to y).

$$\frac{\rho \text{ fresh}}{\Gamma \;;\; \Delta \vdash \texttt{receive}[C] : \rho \triangleright C \;;\; \Delta \oplus \rho} \quad \text{(T-RECV)}$$

The result of `receive` is guaranteed to be immutable or a unique object; therefore, it is guarded by a fresh capability, which is made available to the receiving actor. The resulting class type is always C, since `receive`$[C]$ removes only objects of class type C from the actor's mailbox.

4.7.5 Well-formedness

This section introduces well-formedness rules that extend the rules for our sequential language that we discussed in Section 4.4.2 and Section 4.6.4. Note that none of the existing well-formedness rules, such as the WF-CONFIG rule, have to be changed; in particular, we reuse the Separation and Unique Fields Immutability Invariants.

$$\frac{\begin{array}{c}\Sigma \vdash H\ ;\ \mathcal{A} \\ \exists \Gamma, \Delta.\ \Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ A \\ \forall A' \in \mathcal{A}.\ isolated(H, A, A')\end{array}}{\Sigma \vdash H\ ;\ \{A\} \cup \mathcal{A}} \quad \text{(WF-SOUP)}$$

The WF-SOUP rule defines well-formed configurations consisting of a heap H and a set of actors $\{A\} \cup \mathcal{A}$. The definition is by induction on the set of actors. Each actor must be well-formed in the heap according to some per-actor type environment Γ and static capabilities Δ. Finally, using the new $isolated$ predicate it expresses the fact that in a well-formed configuration all actors are mutually isolated from each other. The $isolated$ predicate is defined as follows:

$$\begin{array}{l} isolated(H, (S, M), (S', M')) \Leftrightarrow \\ isolated(H, S, S') \wedge isolated(H, S, M') \wedge \\ isolated(H, S', M) \wedge isolated(H, M, M') \end{array}$$

Isolation of two actor execution states is defined as follows:

$$\begin{array}{l} isolated(H, \langle V, R, t\rangle, \langle V', R', t'\rangle) \Leftrightarrow \\ (V(x) = \beta \triangleright r \wedge \beta \in R \wedge V'(x') = \beta' \triangleright r' \wedge \beta' \in R') \Rightarrow sep - imm(H, r, r') \end{array}$$

This definition says that two *accessible* variables in the environments of two different actors are separate up to immutable objects. Note that a variable in environment V is accessible iff its capability is contained in the capability set R that corresponds to V.

$$\begin{array}{l} isolated(H, \langle V, R, t\rangle, M') \Leftrightarrow \\ (V(x) = \beta \triangleright r \wedge \beta \in R \wedge r' \in M') \Rightarrow sep - imm(H, r, r') \end{array}$$

An accessible variable in the environment of some actor is separate (in the sense of $sep - imm$) from all messages in the mailbox of some other actor.

$$\begin{array}{l} isolated(H, M, M') \Leftrightarrow \\ \forall r \in M, r' \in M'.\ sep - imm(H, r, r') \end{array}$$

$$sepEnvMbox(H, V, R, M) \Leftrightarrow$$
$$\forall\, (x \mapsto \beta \triangleright r) \in V, r' \in M.\ \beta \in R \Rightarrow sep - imm(H, r, r')$$

$$sepMbox(H, M) \Leftrightarrow$$
$$\forall\, r, r' \in M.\ sep - imm(H, r, r')$$

$$uniqFldsMbox(H, M) \Leftrightarrow$$
$$\forall\, r \in M.\ H(q) = C(\overline{p}) \Rightarrow$$
$$(\forall i \in uniqInd(C).reachable(H, p_i, r') \Rightarrow$$
$$(domedge(H, q, i, r, r') \vee immutable(r', H)))$$

Figure 4.7: Definitions of auxiliary predicates for well-formed actors

Analogous to the previous overloaded definitions of $isolated$, two messages in two different mailboxes are separate up to immutable objects.

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta' \\ \Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R \\ sepEnvMbox(H, V, R, M) \\ sepMbox(H, M) \\ uniqFldsMbox(H, M)\end{array}}{\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ (\langle V, R, t\rangle, M)} \qquad \text{(WF-ACTOR)}$$

The execution state of an actor must be well-formed: the actor's continuation t must be well-typed, and H, V, R must be well-formed according to rule WF-CONFIG of the sequential system. The $sepEnvMbox$ predicate ensures that variables accessible in the actor's environment are separate from messages in its mailbox (see Figure 4.7). Furthermore, two messages in an actor's mailbox are separate up to immutable objects ($sepMbox$). Finally, the Unique Fields Immutability Invariant of Section 4.6.4 also holds for messages in an actor's mailbox ($uniqFldsMbox$).

4.7.6 Isolation

In this section we present the main isolation result for our concurrent language. Basically, we prove a preservation theorem, which guarantees that the isolation invariants introduced above are preserved by reduction. The full proof appears in Section A.5.

Theorem 3 (Isolation) *If* $\Sigma \vdash H \,;\, \mathcal{A}$ *and* $H \,;\, \mathcal{A} \longrightarrow H' \,;\, \mathcal{A}'$*, then there is a* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \vdash H' \,;\, \mathcal{A}'$.

Given the above well-formedness rules (in particular, the definition of $isolated$), this theorem states that accessible variables (*i.e.*, variables whose capabilities are available) in the environments of two different actors are separate up to immutable objects. Together with Theorem 2, which guarantees that only variables whose capabilities are available are accessed during reduction, it follows that two different actors will only access immutable objects concurrently.

4.8 Extensions

In this section we address some of the issues when integrating our type system into full-featured languages like Scala or Java that we omitted from the formalization for simplicity.

4.8.1 Closures

A number of object-oriented languages, such as Scala, have special support for closures. In this section we discuss how our type system handles closures that capture unique variables in their environment.

Consider the following example: a unique list of books should be inserted into a hash map in a way that enables fast access to the books in a certain category. The following code achieves this in an efficient way.

```
val list: List[Book] @unique = ...
val map = new HashMap[String, List[Book]]
list.foreach { b =>
  val sameCat = map(b.cat)
  map.put(b.cat, b :: sameCat)
}
```

For safety, we require the produced hash map to be unique. This means that the capability of `map` must be available after the `foreach`. Intuitively, this is the case if `list` and `map` are in the same region; the body of `foreach` only adds references from the hash map to the books.

Technically, the closure is type-checked as follows. First, we collect the capabilities of references that the closure captures. We require all captured references to be guarded by the same capability, say, ρ. The reason is that all of these references are stored in the same (closure) object, which must, therefore, be guarded

by ρ.[5] In a second step, the body is type-checked assuming that the closure's parameters are also guarded by ρ. In addition, we require that ρ is not consumed in the body. This check allows us to associate a single capability, ρ, with the closure. It indicates that ρ must be available when invoking the closure; moreover, arguments must be guarded by ρ. The type of a closure guarded by ρ, written $\rho \triangleright (A \Rightarrow B)$, effectively corresponds to the method type $\rho \triangleright A \rightarrow (\rho \triangleright B, \{\rho\})$.

Revisiting our example, the `foreach` method in class `List` can then be annotated and type-checked as follows.

```
@transient def foreach(f: (A => Unit) @peer(this)) {
  if (!this.isEmpty) { f(this.head); this.tail.foreach(f) } }
```

Here, `f`'s argument, `this.head`, must be guarded by the same capability as `f`; this is the case, since `f` is a peer of `this`. It is important to note that this does not break any existing code: the annotations merely express that the receiver and the variables captured by `f` must be in the same region. Unannotated objects of existing clients are (all) contained in the global "shared" region, and are therefore compatible with the annotated `foreach`.

What if a closure does not capture a variable in the environment? In this case, we assume that the closure's parameters are guarded by some fresh capability, say, δ, when checking the body. When a closure of type $\delta \triangleright (A \Rightarrow B)$ is passed to a method invoked on an object guarded by ρ, we first capture the closure; this yields a reference to the closure with type $\rho \triangleright (A \Rightarrow B)$, consuming δ. Note that this capturing can occur implicitly, without additions to the program.

Automatic Resource Management

Transient closures that do not capture variables in their environment prevent their arguments to escape to the closure's environment. An important application of closures with non-escaping arguments is automatic resource management (ARM). The goal of ARM is to automate aspects of resource management that are state-dependent and prone to run-time errors. For example, consider operating on a file object using a closure that expects a file argument. Furthermore, suppose that before operating on the file, it must be opened; at the end the file must be closed to free up operating system resources, such as its descriptor. The opening/closing of a file could be done manually inside each closure that is used to operate on the file. However, this would not guard against errors of the programmer where she forgets to open or close the file.

[5] Captured references guarded by different capabilities would have to be stored in unique fields of the closure object; accessing them would require `swap`. Currently, we do not see a practical way to support that.

```
var leaked: BufferedReader
doWith("example.txt", { reader: BufferedReader =>
  val line = reader.readLine()
  println(line)
  leaked = reader
})
val line2 = leaked.readLine()
```

Figure 4.8: Leaking a managed resource

A more robust way to ensure that the argument file is accessed correctly is to use ARM. We explain the ARM pattern using a library function in Scala; however, some languages, such as C# [74], have built-in support for it. The goal of ARM is to provide a library function `doWith` that enables operating on a file while ensuring the file is opened before its use and closed afterwards. For example, `doWith` could be used to read a line of text from a file as follows.

```
doWith("example.txt", { reader: BufferedReader =>
  val line = reader.readLine()
  println(line)
})
```

The closure's `reader` argument provides a high-level interface (Java's `BufferedReader` class) to read lines of text from the file specified as the first argument of `doWith`. Note that the closure neither opens nor closes the `reader` that is used to access the file. Instead, `doWith` opens the file with the name provided as first argument, obtains a reader interface to it, applies the closure to the reader, and then closes the reader again. Importantly, `doWith` ensures that the reader is closed after the closure has been applied to it, even in the presence of exceptions.

Even though the above `doWith` automates some important tasks when managing file resources, it can be misused resulting in run-time errors. An important class of errors stems from leaking the managed resource out of the closure passed to `doWith`; the example in Figure 4.8 illustrates this. Here, we assign the closure's `reader` argument to the variable `leaked` in the environment. The definition of `line2` shows why leaking the reader resource out of the closure's scope is problematic: since `doWith` ensures that `reader` is closed after the closure has been applied to it, reading a line using the `leaked` reader alias results in an exception.

The following implementation of `doWith` shows how to prevent leaking the managed resource using capabilities.

```
def doWith(fileName: String,
           fun: (BufferedReader => Unit)@transient) {
```

```
  val reader: BufferedReader @unique =
    new BufferedReader(new FileReader(fileName))
  try {
    fun(reader)
  } finally {
    reader.close()
  }
}
```

The idea is to require `doWith`'s argument closure to be transient. According to the typing rules explained above this means that the closure's argument must be transient. Therefore, the `fun` closure can be applied to the unique `reader` instance. Since `fun` does not consume `reader`'s capability, we can close the `reader` in the finally clause. Importantly, since the closure's argument is transient, we ensure that it is not leaked to a variable in the closure's environment. In particular, the assignment of `reader` to the `leaked` variable in Figure 4.8 is rejected by the type checker: the capability of `reader` is different from the capability of `leaked`, which makes their types incompatible. In conclusion, transient closures can be used to ensure important safety properties of abstractions for automatic resource management.

4.8.2 Nested classes

Nested classes can be seen as a generalization of closures; a nested class may define multiple methods, and it may be instantiated several times. An important use case are anonymous iterator definitions in collection classes.

For instance, the `SingleLinkedList` class in Scala's standard library provides the following method for obtaining an iterator (the `A` type parameter is the collection's element type):

```
def elements: Iterator[A] = new Iterator[A] {
  var elems = SingleLinkedList.this
  def hasNext = (elems ne null)
  def next = { val res = elems.elem; elems = elems.next; res }
}
```

The nested `Iterator` subclass stores a captured reference to the receiver in its `elems` field. Therefore, the iterator instance cannot be unique, since it is not separate from the receiver. However, it is safe to create the iterator in the region of the receiver.

In general, new nested class instances can be created in the region of captured references if all those references are in the same region, say, ρ (similar to closures). If there are constructor arguments, they must also be guarded by ρ. Note

that creating the instance does not consume ρ. This means, we are relaxing the rule for instance creation introduced in Section 4.4, which requires the capabilities of constructor arguments to be distinct and consumed; it applies equally to non-nested classes. Note that nested classes may have unique fields; initializing unique fields through constructor parameters must follow the same rule as normal instance creation, that is, the arguments must be guarded by distinct capabilities, which are consumed.

Revisiting the iterator example, we can use the `@peer` annotation to express the fact that the iterator is created in the same region as the receiver:

```
@transient def elements: Iterator[A] @peer(this) = ...
```

This enables arbitrary uses of an iterator while the capability of its underlying unique collection is available.

4.8.3 Transient classes

We say that a class is *transient* if none of its fields are unique and all of its methods can be annotated such that the receiver is marked as `@transient` and all parameters are marked as `@peer(this)`. This means that the receiver and the parameters of a method must be guarded by the same capability. It ensures that neither the receiver nor objects reachable from it are leaked to (potentially) shared objects, since (1) shared objects are guarded by the special "shared" capability, and (2) capabilities of method parameters are universally quantified, making them incompatible with the shared capability. We have found that most classes used in messages are transient (see Section 4.9). This means that most objects only interact with objects from its enclosing aggregate, which is consistent with the results for thread-locality in Loci [139]. To abbreviate the canonical annotation, we allow classes to be annotated as `@transient`.

4.9 Implementation

We have implemented our type system as a plug-in [64] for the Scala compiler developed at EPFL. The plug-in inserts an additional compiler phase that runs right after the normal type checker. The extended compiler first does standard Scala type checking on the erased terms and types of our system. Then, types and capabilities are checked using (an extension of) the type rules presented in Section 4.4 and Section 4.6. For subsequent code generation, all capabilities and `capturedBy` expressions are erased, since they have no observable effects.

4.9.1 Practical experience

In a first step, we annotated the (mutable) `DoubleLinkedList`, `ListBuffer`, and `HashMap` classes from the collections of Scala 2.7 including all classes/traits that these classes transitively extend, comprising 2046 lines of code. Making all classes transient (see Section 4.8.3) required changing 60 source lines.

To evaluate our system in a concurrent setting, we use two applications: a simple ray tracer, and a parallel testing framework. We first report on our experience with the ray tracer application; after that we discuss the testing framework.

Ray Tracer

The ray tracer application is parallelized by dividing the rendering of a single image into multiple equal-sized chunks, such that each chunk is rendered by an actor. To avoid copying completed chunks, the computed data is accumulated in mutable pixel buffers supporting a constant-time append operation. Once all chunks have been rendered, the pixel buffers are sent to an actor that creates an image object that is displayed using classes of the Java Swing GUI library.

To ensure that all actors are isolated from each other, the (mutable) pixel buffers must be sent using unique references. For this, the pixel buffer classes are declared as transient. Moreover, code manipulating pixel buffers must do this using unique references. The ray tracer application comprises 414 lines of code (including whitespace). Enabling the compiler plug-in to check for uniqueness required changing or adding 18 lines of code (adding 3 `@transient` and 8 `@unique` annotations). We found that programming with unique pixel buffers is straightforward, since our system permits local aliasing within method bodies. Interestingly, the fact that the Java image class for displaying the computed images is unannotated does not pose a problem, since instances of that class can be created locally on the event dispatch thread of the GUI library; only the pixel buffers used to create such image objects are transferred among multiple actors.

The annotation checker does, however, include an escape hatch for transferring objects that cannot be verified to be unique: an expression that has a type annotated with `@uncheckedUnique` is checked as if its type were guarded by a fresh capability, effectively treating it as unique. We did not make use of this escape hatch in our case study.

Parallel Testing Framework

In Section 4.3, we have already introduced the partest testing framework, which is used to run the check-in and nightly tests for the Scala compiler and standard library. Although the majority of code deals with compiler/test set-up and reporting, the unique objects that are transferred among actors are used pervasively

throughout large parts of the code. An example for such a class is `LogFile`, which receives output from various sources (compiler, test runner etc.). For creating unique `LogFile` instances it is sufficient that the class is transient; However, `LogFile` inherits from the standard `java.io.File` class, which is unchecked. Fortunately, according to the Java version 6 API [34], "instances of the `File` class are immutable." We configured our type checker to skip checking immutable classes. In general, however, this is unsound if such classes could mutate or leak method parameters. Overall, the most important changes were:

1. Annotating the message classes. We found that all message classes could be annotated as `@transient`.

2. Handling of unique fields. We had to annotate a field holding a list of created log files as `@unique`. Three `swap` expressions were sufficient to cover all accesses to the field.

In summary, out of the 4182 lines of code (including whitespace), we had to change 32 lines and add 29 additional lines. The following observation helped interoperability: passing a unique object to an unannotated method is often unproblematic if the method expects an immutable type. However, this is unsound in the general case, since instances of such types could be downcast to mutable types. In our study we allowed passing a `LogFile` instance to methods of unannotated Java classes expecting a `java.io.File`.

Chapter 5

Conclusion and Future Work

In this thesis we have explored concurrent programming based on message passing, specifically in the context of actors. We identified two challenges that we believe are important to address before actors can be a viable solution for large-scale concurrency.

The first challenge concerns actor implementations. Production systems, such as Twitter, demand efficient actor implementations that scale to very large numbers of actors. At the same time, it is necessary that actors interoperate with the threading model of the underlying platform to enable re-use of existing thread-based code.

The second challenge concerns safety and efficiency of message passing. When scaling an actor-based application from a single multi-core processor to a distributed cluster-based system, it is important that local and remote message sends have the same semantics. At the same time, local message sends should be implemented efficiently without introducing potential data races.

This thesis describes a practical approach that addresses both challenges. One of the main insights is that a single actor-based abstraction can be used to program both in a thread-based and in an event-driven style. The two programming styles are embodied in two different operations for message reception. Event-based actors only use event-based operations. They provide excellent scalability; in a standard benchmark their throughput is about two times higher than that of state-of-the-art JVM-based actor implementations. Moreover, we provide experimental evidence that Erlang-style actors can be implemented with only a modest overhead compared to simpler actor abstractions based on inversion of control. Thread-based actors support blocking operations, but they are more heavyweight because of the additional resources consumed by their associated VM threads. Importantly, whether an actor is thread-based or not is not fixed at the time when the actor is created; the same actor can use both thread-based and event-based operations. This means that actors may become thread-based temporarily, even

if they mostly use event-based operations. As a result, programmers can trade the efficiency of event-based actors for the flexibility of thread-based actors in a fine-grained way.

To address the second challenge we have introduced a new type-based approach to uniqueness in object-oriented programming languages. Simple capabilities enforce both aliasing constraints for uniqueness and at-most-once consumption of unique references. By identifying unique and borrowed references as much as possible our approach provides a number of benefits: first, a simple formal model, where unique references "subsume" borrowed references. Second, the type system does not require complex features, such as existential ownership or explicit region declarations. The type system has been proven sound and can be integrated into full-featured languages, such as Scala. Practical experience with collection classes and actor-based concurrent programs suggests that the system allows type checking real-world Scala code with only few changes.

5.1 Future Work

We envision several avenues for further research. In this section we outline future work in the areas of fault tolerance and type systems.

5.1.1 Fault tolerance

The Scala Actors library includes a runtime system that provides basic support for remote (*i.e.*, inter-VM) actor communication. To provide support for fault tolerancy (for instance, in mobile ad-hoc networks), it would be interesting to extend the framework with remote actor references that support volatile connections, similar to ambient references [36]. Integrating transactional abstractions for fault-tolerant distributed programming (*e.g.*, [52, 142]) into Scala Actors is another interesting area for future work.

5.1.2 Type systems

Rich types for safe concurrency In this thesis we have introduced a type-and-capability system to enforce race safety in actor-based programs. There are a number of ways in which our initial results could be extended or applied to different concurrency paradigms.

First, in recent work by Leino et al. [88] static permissions (similar to our capabilities) are used to verify the absence of deadlocks in concurrent programs that use channels or locks for synchronization. This result suggests that our type-and-

capability system could be extended to verify deadlock freedom in actor-based programs.

Second, type systems based on capabilities/permissions could be used to check interesting safety properties of high-level concurrency models different from actors such as data-parallel programming and software transactional memory. For instance, it seems that capabilities could be used to verify safe publication and privatization idioms in programs using software transactional memory [115]. Similarly, uniqueness types could allow the identification of parallelizable code portions, thereby supporting a form of implicit parallelism. Finally, it would be interesting to explore hybrid concurrency models where, *e.g.*, actors can internally be deterministically concurrent (using parallel collections, say).

Unified framework for types with stack wrappers A number of type systems that have recently been published, such as Loci [139], Like Types [138], and our Capabilities for Uniqueness [69], are formalized using wrapped pointers on the stack. For instance, in our system shape and uniqueness properties can be enforced (transitively) by constraining stack values using appropriate wrappers, called guarded types; these types depend on linear/affine capabilities. This suggests that systems using this or a similar technique could be modeled as instances of the same typing framework.

Appendix A

Full Proofs

A.1 Lemmas

Lemma 1 (Weakening) *If* $\Gamma\ ;\ \Delta \vdash\ t : T\ ;\ \Delta'$*,* $x \notin dom(\Gamma)$*, then* $\Gamma, x : S\ ;\ \Delta \vdash\ t : T\ ;\ \Delta'$.

Proof: By induction on the typing derivation.

- Case (T-Var).
  1. By the assumptions
     (a) $\Gamma(y) = \rho \triangleright C$
     (b) $\rho \in \Delta$
  2. Define $\Gamma' := (\Gamma, x : S)$. Since $x \notin dom(\Gamma)$, $\Gamma'(y) = \rho \triangleright C$.
  3. By 1.b) and 2., $\Gamma'\ ;\ \Delta \vdash\ y : \rho \triangleright C\ ;\ \Delta$.
- Cases (T-Select), (T-Assign), (T-New), (T-Let), (T-Invoke), (T-Capture), (T-Swap), (T-Sub) follow directly from the induction hypothesis.

□

Lemma 2 *If* $\Gamma\ ;\ \Delta \vdash\ t : \rho \triangleright C\ ;\ \Delta'$*, then* $\rho \in \Delta'$.

Proof: By induction on the typing derivation.

- Cases (T-Var), (T-New), (T-Swap) are immediate.
- Case (T-Sub).
  1. By the assumptions

   (a) $\Gamma\ ;\ \Delta \vdash\ e : T'\ ;\ \Delta'$
   (b) $T' <: \rho \triangleright C$
2. By 1.b) and (<:-Cap), $T' = \rho \triangleright D$.
3. By 1.a), 2., and the IH, $\rho \in \Delta'$.

- Case (T-Select). By the assumptions, $\Gamma\ ;\ \Delta \vdash\ y\ :\ \rho \triangleright C\ ;\ \Delta$. The IH provides $\rho \in \Delta$.

- Case (T-Invoke).

1. By the assumptions
   (a) $mtype(m, D_1) = \exists\delta.\ \overline{\delta \triangleright D} \to (R, \Delta_m)$
   (b) $\sigma = \overline{\delta \mapsto \rho} \circ \delta \mapsto \rho'$ injective
   (c) ρ' fresh
2. By 1.a) and (WF-Method)
   (a) $R = \delta' \triangleright D$
   (b) $\delta' \in \Delta_m$
   (c) $\delta = \begin{cases} \delta' & \text{if } \delta' \notin \overline{\delta} \\ \text{fresh} & \text{otherwise} \end{cases}$
3. If $\delta' \notin \overline{\delta}$, then by 2.c), $\sigma\delta' = \sigma\delta = \rho'$. By 1.b) and 2.b), $\sigma\delta' \in \sigma\Delta_m$.
4. If $\delta' = \delta_i \in \overline{\delta}$, then by 1.b) and 2.b), $\sigma\delta' = \sigma\delta_i = \rho_i \in \sigma\Delta_m$.

- Case (T-Capture).

1. By the assumptions
   (a) $\Gamma\ ;\ \Delta \vdash\ z : \rho' \triangleright C'\ ;\ \Delta$
   (b) $\Delta = \Delta' \oplus \rho$
2. By 1.a) and the IH, $\rho' \in \Delta$, and since $\rho \neq \rho'$, we have by 1.b), $\rho' \in \Delta'$.

- Case (T-Let).

1. By the assumptions
   (a) $\Gamma\ ;\ \Delta \vdash\ e : T'\ ;\ \Delta''$
   (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash\ t : \rho \triangleright C\ ;\ \Delta'$
2. By 1.b) and the IH, $\rho \in \Delta'$.

- Case (T-Assign). By the assumptions, $\Gamma\ ;\ \Delta \vdash\ y\ :\ \rho \triangleright C\ ;\ \Delta$. The IH provides $\rho \in \Delta$.

□

Lemma 3 (Capability Weakening) *If* $\Gamma\ ;\ \Delta\ \vdash\ t : T\ ;\ \Delta'$ *and* $\forall\ (x : \rho \triangleright C) \in \Gamma.\ \rho \neq \delta$*, then* $\Gamma\ ;\ \Delta \oplus \delta\ \vdash\ t : T\ ;\ \Delta' \oplus \delta$*.*

Proof: By induction on the typing derivation.

- Case (T-Let).
  1. By the assumptions
     (a) $\Gamma\ ;\ \Delta\ \vdash\ e : \rho' \triangleright C'\ ;\ \Delta''$
     (b) $\Gamma, x : \rho' \triangleright C'\ ;\ \Delta''\ \vdash\ t : T\ ;\ \Delta'$
  2. By 1.a) and the IH, $\Gamma\ ;\ \Delta \oplus \delta\ \vdash\ e : \rho' \triangleright C'\ ;\ \Delta'' \oplus \delta$.
  3. By 1.a) and Lemma 2, $\rho' \in \Delta''$.
  4. By 2. and 3., $\forall\ (y \mapsto \rho \triangleright D) \in (\Gamma, x : \rho' \triangleright C').\ \rho \neq \delta$.
  5. By 1.b), 4., and the IH, $\Gamma, x : \rho' \triangleright C'\ ;\ \Delta'' \oplus \delta\ \vdash\ t : T\ ;\ \Delta' \oplus \delta$
  6. By 2., 5., (T-Let), $\Gamma\ ;\ \Delta \oplus \delta\ \vdash\ \texttt{let}\ x = e\ \texttt{in}\ t : T\ ;\ \Delta' \oplus \delta$
- All other cases follow directly from the induction hypothesis.

□

Lemma 4 (Capability Renaming) *If* $\Gamma\ ;\ \Delta\ \vdash\ t\ :\ T\ ;\ \Delta'$ *and* $\sigma\ \in\ Caps \rightarrow Caps$ *is injective, then* $\sigma\Gamma\ ;\ \sigma\Delta\ \vdash\ t : \sigma T\ ;\ \sigma\Delta'$*.* σ *extends to type environments, capabilities, and types in the natural way:*

- $\sigma(\Gamma, x : T) = (\sigma\Gamma, x : \sigma T)$
- $\sigma(\Delta \oplus \rho) = \sigma\Delta \oplus \sigma\rho$
- $\sigma(\rho \triangleright C) = \sigma(\rho) \triangleright C$

Proof: By induction on the typing derivation with case analysis of the last type rule used.

- Case (T-Var): $\Gamma\ ;\ \Delta\ \vdash\ y : \rho \triangleright C\ ;\ \Delta$.
  1. By the assumptions
     (a) $\Gamma(y) = \rho \triangleright C$
     (b) $\rho \in \Delta$
  2. By 1.a,b), def. of σ

   (a) $(\sigma\Gamma)(y) = \sigma(\rho) \triangleright C$
   (b) $\sigma(\rho) \in \sigma\Delta$
3. By 2.a,b), (T-Var), $\sigma\Gamma \;;\; \sigma\Delta \vdash y : \sigma(\rho) \triangleright C \;;\; \sigma\Delta$.
4. By 3., and the def. of σ, $\sigma\Gamma \;;\; \sigma\Delta \vdash y : \sigma(\rho \triangleright C) \;;\; \sigma\Delta$.

- Case (T-Select): $\Gamma \;;\; \Delta \vdash y.l_i : \rho \triangleright D_i \;;\; \Delta$.

1. By the assumptions
   (a) $\Gamma \;;\; \Delta \vdash y : \rho \triangleright C$
   (b) $fields(C) = \overline{\alpha\; l : D}$
   (c) $\alpha_i \neq \texttt{unique}$
2. By 1.a) and the IH, $\sigma\Gamma \;;\; \sigma\Delta \vdash y : \sigma(\rho \triangleright C)$.
3. By 2., and the def. of σ, $\sigma\Gamma \;;\; \sigma\Delta \vdash y : \sigma(\rho) \triangleright C$.
4. By 1.b,c), 3., and (T-Select), $\sigma\Gamma \;;\; \sigma\Delta \vdash y.l_i : \sigma(\rho) \triangleright D_i \;;\; \sigma\Delta$.
5. By 4. and the def. of σ, $\sigma\Gamma \;;\; \sigma\Delta \vdash y.l_i : \sigma(\rho \triangleright D_i) \;;\; \sigma\Delta$.

- Case (T-Assign). Similar to case (T-Select) and therefore omitted.

- Cases (T-Let) and (T-Sub) follow directly from the IH.

- Case (T-New): $\Gamma \;;\; \Delta \vdash \texttt{new}\, C(\overline{y}) : T \;;\; \Delta'$.

1. By the assumptions
   (a) $\Gamma \;;\; \Delta \vdash \overline{y : \rho \triangleright D} \;;\; \Delta$
   (b) Has been removed.
   (c) $fields(C) = \overline{\alpha\; l : D}$
   (d) ρ' fresh
   (e) $\Delta = \hat{\Delta} \oplus \overline{\rho}$
   (f) $\Delta' = \hat{\Delta} \oplus \rho'$
   (g) $T = \rho' \triangleright C$
2. By 1.a) and the IH, $\sigma\Gamma \;;\; \sigma\Delta \vdash \overline{y : \sigma(\rho \triangleright D)} \;;\; \sigma\Delta$.
3. By 1.e) and since σ is injective, we have $|\{\sigma(\overline{\rho})\}| = |\overline{y}|$.
4. By 1.d,e,f) and the def. of σ
   (a) $\sigma(\rho')$ fresh
   (b) $\sigma\Delta = \sigma\hat{\Delta} \oplus \sigma(\overline{\rho})$
   (c) $\sigma\Delta' = \sigma\hat{\Delta} \oplus \sigma(\rho')$

5. By 1.c), 2., 3., 4.a,b,c), (T-New), $\sigma\Gamma\ ;\ \sigma\Delta\ \vdash\ \mathsf{new}\ C(\overline{y})\ :\ \sigma(\rho')\triangleright C\ ;\ \sigma\Delta'$.
6. By 1.g), 5. and the def. of σ, $\sigma\Gamma\ ;\ \sigma\Delta\ \vdash\ \mathsf{new}\ C(\overline{y}) : \sigma T\ ;\ \sigma\Delta'$.

- Case (T-Invoke): $\Gamma\ ;\ \Delta\ \vdash\ y.m(\overline{z}) : T\ ;\ \Delta'$.

1. By the assumptions
   (a) $\Gamma\ ;\ \Delta\ \vdash\ y : \rho_1\triangleright D_1\ ;\ \Delta$
   (b) $\forall i \in 2..n.\ \Gamma\ ;\ \Delta\ \vdash\ z_{i-1} : \rho_i\triangleright D_i\ ;\ \Delta$
   (c) $mtype(m, D_1) = \exists\delta.\ \overline{\delta\triangleright D}\to(R, \Delta_m)$
   (d) $\sigma' = \overline{\delta\mapsto\rho}\circ\delta\mapsto\rho$ injective
   (e) $\Delta = \hat{\Delta}\uplus\{\rho\mid\rho\in\overline{\rho}\}$
   (f) $T = \sigma' R$
   (g) $\Delta' = \sigma'\Delta_m\oplus\hat{\Delta}$
   (h) ρ fresh
2. By 1.a,b), the IH, and the def. of σ
   (a) $\sigma\Gamma\ ;\ \sigma\Delta\ \vdash\ y : \sigma(\rho_1)\triangleright D_1\ ;\ \sigma\Delta$
   (b) $\forall i \in 2..n.\ \sigma\Gamma\ ;\ \sigma\Delta\ \vdash\ z_{i-1} : \sigma(\rho_i)\triangleright D_i\ ;\ \sigma\Delta$
3. By 1.d), the def. of σ, and the fact that σ is injective, $\sigma\circ\sigma' = \overline{\delta\mapsto\sigma(\rho)}\circ\delta\mapsto\sigma(\rho)$ injective.
4. By 1.e,f,g), and the def. of σ
   (a) $\sigma\Delta = \sigma\hat{\Delta}\uplus\{\rho\mid\rho\in\sigma(\overline{\rho})\}$
   (b) $\sigma T = (\sigma\circ\sigma')R$
   (c) $\sigma\Delta' = (\sigma\circ\sigma')\Delta_m\oplus\sigma\hat{\Delta}$
5. By 1.c), 2.a,b), 3., 4.a,b,c), (T-Invoke), $\sigma\Gamma\ ;\ \sigma\Delta\ \vdash\ y.m(\overline{z}) : \sigma T\ ;\ \sigma\Delta'$.

- The cases (T-Capture) and (T-Swap) are similar to case (T-Select) and are therefore omitted.

□

Lemma 5 *If* $\Gamma\ ;\ \Delta\ \vdash\ y : \rho\triangleright C\ ;\ \Delta$, $\Gamma\ ;\ \Delta\ ;\ \Sigma\vdash_\varphi V\ ;\ R$, *and* $V(y) = \beta\triangleright r$, *then* $\Sigma\ \vdash\ r : C$ *and* $\rho\in\Delta$ *iff* $\beta\in R$.

Proof: By induction on the typing derivation with case analysis of the last rule used in the derivation.

- Case (T-Var).

1. By the assumptions and (T-Var), $\Gamma(y) = \rho \triangleright C$.
2. By 1., (WF-Env)
   (a) $\Sigma \vdash r : C$
   (b) $\rho \in \Delta$ iff $\beta \in R$

- Case (T-Sub).

1. By the assumptions and (T-Sub)
   (a) $\Gamma\ ;\ \Delta \vdash y : T\ ;\ \Delta$
   (b) $T <: \rho \triangleright C$
2. By 1.b), (<:-Cap)
   (a) $T = \rho \triangleright D$
   (b) $D <: C$
3. By 1.a), 2.a), and the induction hypothesis
   (a) $\Sigma \vdash r : D$
   (b) $\rho \in \Delta$ iff $\beta \in R$
4. By 3.a), (Heap-Type)
   (a) $\Sigma(r) = E$
   (b) $E <: D$
5. By 2.b), 4.b), (<:-Trans), $E <: C$.
6. By 4.a), 5., (Heap-Type), $\Sigma \vdash r : C$.

□

Lemma 6 *If* $D <: C$, $fields(C) = \overline{fld^C}$, $fields(D) = \overline{fld^D}$, *and* $i \leq |fields(C)|$, *then* $\forall j \leq i.\ fld^D_j = fld^C_j$.

Proof: By induction on the derivation of $D <: C$ with case analysis of the last rule used.

- Case (<:-P). Then, $P(D) = \texttt{class}\ D\ \texttt{extends}\ C\ \{\overline{fld}\ \overline{meth}\}$

1. By (WF-Class), $\forall\ \beta\ l : G \in \overline{fld}.\ l \notin fields(C)$
2. By def. of $fields$, $fields(D) = \overline{fld^C}, \overline{fld}$. Therefore, $fld^D_j = fld^C_j,\ j \leq i$.

- Case (<:-Refl). We have $\overline{fld^C} = \overline{fld^D}$.

- Case (<:-Trans).

1. There is a type E such that
   (a) $D <: E$
   (b) $E <: C$
2. Let $fields(E) = \overline{fld^E}$. By 1.b) and the IH, $fld^E_j = fld^C_j,\ j \leq i$.
3. By 1.a), the IH, and 2., $fld^D_j = fld^C_j,\ j \leq i$.

□

Lemma 7 *If* $\Gamma\ ;\ \Delta \vdash\ t : T\ ;\ \Delta'$, $x \in dom(\Gamma)$, $x' \notin dom(\Gamma)$, *then* $[x'/x]\Gamma\ ;\ \Delta \vdash [x'/x]t : T\ ;\ \Delta'$.

Proof: Straightforward induction on the typing derivation. □

A.2 Proof of Theorem 1

Theorem 1 (Preservation) *If*

- $\Gamma\ ;\ \Delta \vdash\ t : T\ ;\ \Delta'$
- $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash\ H\ ;\ V\ ;\ R$
- $H, V, R, t \longrightarrow H', V', R', t'$

then there are $\Gamma' \supseteq \Gamma$, Δ'', *and* $\Sigma' \supseteq \Sigma$ *such that*

- $\Gamma'\ ;\ \Delta'' \vdash\ t' : T\ ;\ \Delta'$
- $\Gamma'\ ;\ \Delta''\ ;\ \Sigma' \vdash\ H'\ ;\ V'\ ;\ R'$

We use the following extended rules for well-formed configurations and environments, which maintain an injective mapping between static and dynamic capabilities.

$$\frac{\begin{array}{c}\Sigma \vdash\ H \\ \Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R \\ separation(V, H, R) \\ uniqFlds(V, H, R) \\ \varphi \text{ injective} \qquad \varphi(\varsigma) = \varsigma\end{array}}{\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash\ H\ ;\ V\ ;\ R} \qquad \text{(WF-CONFIG)}$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R \\ \Sigma \vdash\ r : C \qquad \rho \in \Delta \text{ iff } \beta \in R \qquad \varphi(\rho) = \beta\end{array}}{(\Gamma, y : \rho \triangleright C)\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi (V, y \mapsto \beta \triangleright r)\ ;\ R} \qquad \text{(WF-ENV)}$$

Proof: By induction on the reduction derivation with case analysis of the last rule used in the derivation.

- Case (R-Assign). Then, $t = y.l_i := z$.

  1. By (WF-Config)
     (a) $\Sigma \vdash H$
     (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R$
     (c) $separation(V, H, R)$
     (d) $uniqFlds(V, H, R)$
     (e) φ injective
  2. By the assumptions and (T-Assign)
     (a) $\Delta' = \Delta$
     (b) $\Gamma\ ;\ \Delta \vdash y : \rho \triangleright C$
     (c) $\Gamma\ ;\ \Delta \vdash z : \rho \triangleright D_i$
     (d) $fields(C) = \overline{\alpha\ l : D}$
     (e) $\alpha_i \neq$ `unique`
  3. By 1.b), 2.b), Lemma 5
     (a) $V(y) = \delta \triangleright r$
     (b) $\Sigma \vdash r : C$
     (c) $\rho \in \Delta$ iff $\delta \in R$
  4. By (R-Assign)
     (a) $V(z) = \delta \triangleright r'$
     (b) $H(r) = D(\overline{p})$
     (c) $fields(D) = \overline{\beta\ l : G}$
     (d) $\beta_i \neq$ `unique`
     (e) $H' = H[r \mapsto D([r'/p_i]\overline{p})]$
  5. By 1.a), 4.b,c), (WF-Heap)
     (a) $\Sigma \vdash \overline{p} : \overline{G}$
     (b) $\Sigma(r) = D$
  6. By 3.b), 5.b), (Heap-Type), $D <: C$.
  7. By 2.d,e), 4.c), 6., Lemma 6, $D_i = G_i$.
  8. By 1.b), 2.c), 4.a), Lemma 5, $\Sigma \vdash r' : D_i$.
  9. By 1.a), 4.c), 5.a,b), 7., 8., (WF-Heap), $\Sigma \vdash H'$.

10. Let $(x : \gamma \triangleright u), (x' : \gamma' \triangleright u') \in V$ such that $\{\gamma, \gamma'\} \subseteq R$. Let r_c be an object reachable in H' from both u and u'.
    (a) Case $\gamma \neq \delta \wedge \gamma' \neq \delta$. By 1.c), 4.a), $\delta \in R$
        i. $\neg reach(H, r', r_c)$, therefore
        ii. r_c reachable in H from both u and u'. By 1.c),
        iii. $\gamma = \gamma'$
    (b) Case $\gamma = \delta$. We have to consider the following subcases:
        i. r_c is a common reachable object of u and u' in H. By the IH, $\gamma = \gamma'$.
        ii. r_c is a common reachable object of r' and u' in H and $reach(H, u, r)$. By 1.c), $\gamma' = \delta = \gamma$.
        iii. r_c is a common reachable object of r' and u in H and $reach(H, u', r)$. By 1.c), 3.a), $\gamma' = \delta = \gamma$.

    Therefore, we have $separation(V, H', R)$.
11. Let $(x \mapsto \delta \triangleright o) \in V$, $H'(q) = C(\overline{q})$, $j \in uniqInd(C)$, $reach(H', q_j, o')$.
    (a) Case $q \neq r$. Then $H'(q) = H(q)$. Assume $\neg reach(H, q_j, o')$. Then
        i. $reach(H, r', o')$
        ii. $reach(H, q_j, r)$

        By (a).ii, and the IH, $domedge_H(q, j, o, r)$. This contradicts with 3.a). Therefore, $reach(H, q_j, o')$. By IH, $domedge_H(q, j, o, o')$. Assume $\neg domedge_{H'}(q, j, o, o')$. Then
        i. $reach(H, o, r)$
        ii. $reach(H, r', o')$

        where none of the paths goes through edge (q, j). By the second case and the IH, $domedge_H(q, j, r', o')$. Contradiction. Therefore, we have $domedge_{H'}(q, j, o, o')$ as required.
    (b) Case $q = r$. By (R-Assign), $H'(q) = D([r'/p_i]\overline{p})$. Since $i \notin uniqInd(D)$, the rest of the proof is identical to the previous case.
12. By 2.b), (T-Var), $\Gamma\ ;\ \Delta \vdash y : \rho \triangleright C\ ;\ \Delta$.
13. By 1.b,e), 9., 10., 11., (WF-Config), $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H'\ ;\ V\ ;\ R$.

- Case (R-Invoke). Then, $t = \texttt{let}\ x = y.m(\overline{z})\ \texttt{in}\ t'$.

1. By (WF-Config)
    (a) $\Sigma \vdash H$

   (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R$
   (c) $separation(V, H, R)$
   (d) $uniqFlds(V, H, R)$
   (e) φ injective

2. By the assumptions and (T-Let)
   (a) $\Gamma\ ;\ \Delta \vdash y.m(\overline{z}) : T'\ ;\ \Delta''$
   (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$

3. By 2.a) and (T-Invoke)
   (a) $\Gamma\ ;\ \Delta \vdash y : \rho_1 \triangleright C_1$
   (b) $\forall i \in 2..n.\ \Gamma\ ;\ \Delta \vdash z_{i-1} : \rho_i \triangleright D_i\ ;\ \Delta$
   (c) $mtype(m, C_1) = \exists \delta.\ \overline{\delta \triangleright D} \to (R, \Delta_m)$
   (d) $\sigma = \overline{\delta \mapsto \rho} \circ \delta \mapsto \rho$ injective
   (e) $\Delta = \Delta_r \uplus \{\rho \mid \rho \in \overline{\rho}\}$
   (f) $T' = \sigma R$
   (g) $\Delta'' = \sigma\Delta_m \oplus \Delta_r$
   (h) ρ fresh

4. By 1.b), 3.a), Lemma 5
   (a) $V(y) = \beta_1 \triangleright r_1$
   (b) $\Sigma \vdash r_1 : C_1$

5. By 1.b), 3.b,d,e), Lemma 5
   (a) $\forall i \in 2..n.\ V(z_{i-1}) = \beta_i \triangleright r_i$
   (b) $\Sigma \vdash \overline{r : D}$
   (c) $\overline{\beta} \subseteq R$, and by 1.e)
   (d) $\overline{\rho \mapsto \beta}$ injective

6. By 4.b), (Heap-Type)
   (a) $\Sigma(r_1) = D_1$
   (b) $D_1 <: C_1$

7. By 1.a), 6.a), (WF-Heap), $H(r_1) = D_1(\_)$.

8. By 4.a), 5.a,c), 7., (R-Invoke)
   (a) $mbody(m, D_1) = (\overline{x}, e)$
   (b) $V' = (V, \overline{x \mapsto \beta \triangleright r})$

9. By 3.c), 8.a), (WF-Method)
   (a) $R = \delta' \triangleright D$

(b) $\overline{x : \delta \triangleright D}\ ;\ \{\delta \mid \delta \in \overline{\delta}\} \vdash e : \delta' \triangleright D\ ;\ \Delta_m$

10. By 3.d,h), 9.b), Lemma 4
    (a) $\overline{x : \rho \triangleright D}\ ;\ \{\rho \mid \rho \in \overline{\rho}\} \vdash e : \rho \triangleright D\ ;\ \sigma\Delta_m$
    (b) $\rho = \begin{cases} \rho_i & \text{if } \delta' = \delta_i \\ \text{fresh} & \text{otherwise} \end{cases}$
11. Has been removed.
12. By 3.d,e), 10.a), Lemma 3, $\overline{x : \rho \triangleright D}\ ;\ \{\rho \mid \rho \in \overline{\rho}\} \oplus \Delta_r \vdash e : \rho \triangleright D\ ;\ \sigma\Delta_m \oplus \Delta_r$.
13. WLOG, $\overline{x} \cap dom(\Gamma) = \emptyset$. By 12., Lemma 1, $\Gamma, \overline{x : \rho \triangleright D}\ ;\ \{\rho \mid \rho \in \overline{\rho}\} \oplus \Delta_r \vdash e : \rho \triangleright D\ ;\ \sigma\Delta_m \oplus \Delta_r$.
14. By 3.d,f,h), 9.a), 10.b), $\rho \triangleright D = \sigma R = T'$.
15. By 2.b), Lemma 1, $\Gamma, x : T', \overline{x : \rho \triangleright D}\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$.
16. By 2.b), 3.e,f,g), 13., 14., 15., (T-Let), $\Gamma, \overline{x : \rho \triangleright D}\ ;\ \Delta \vdash \texttt{let}\ x = e\ \texttt{in}\ t' : T\ ;\ \Delta'$.
17. By 1.b), 3.e), 5.b,c), 8.b)
    (a) $\Gamma, \overline{x : \rho \triangleright D}\ ;\ \Delta\ ;\ \Sigma \vdash_{\varphi'} V'\ ;\ R$ and by 5.d)
    (b) $\varphi' = \varphi \cup \overline{\rho \mapsto \beta}$ injective
18. By 1.d), 4.a), 5.a), 8.b), $separation(V', H, R)$.
19. By 1.e), 4.a), 5.a), 8.b), $uniqFlds(V', H, R)$.
20. By 1.a), 17.a,b), 18., 19., (WF-Config), $\Gamma, \overline{x : \rho \triangleright D}\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V'\ ;\ R$.

- Case (R-Swap). Then, $t = \texttt{let}\ x = \texttt{swap}(y.l_i, z)\ \texttt{in}\ t'$.

1. By (WF-Config)
   (a) $\Sigma \vdash H$
   (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R$
   (c) $separation(V, H, R)$
   (d) $uniqFlds(V, H, R)$
   (e) φ injective
2. By the assumptions and (T-Let)
   (a) $\Gamma\ ;\ \Delta \vdash \texttt{swap}(y.l_i, z) : T'\ ;\ \Delta''$
   (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
3. By 2.a) and (T-Swap)
   (a) $\Gamma\ ;\ \Delta \vdash y : \rho \triangleright C\ ;\ \Delta$

   (b) $\Gamma\ ;\ \Delta \vdash\ z : \rho' \triangleright D_i\ ;\ \Delta$
   (c) $fields(C) = \overline{\alpha\ l : D}$
   (d) $\alpha_i = \texttt{unique}$
   (e) $\Delta = \hat{\Delta} \oplus \rho'$
   (f) $T' = \rho'' \triangleright D_i$ where
   (g) ρ'' fresh
   (h) $\Delta'' = \hat{\Delta} \oplus \rho''$

4. By 1.b), 3.a), Lemma 5
   (a) $V(y) = \delta \triangleright r$
   (b) $\Sigma \vdash\ r : C$
   (c) $\rho \in \Delta$ iff $\delta \in R$

5. By 4.b), (Heap-Type)
   (a) $\Sigma(r) = D$
   (b) $D <: C$

6. By 1.a), 5.a), (WF-Heap), (R-Swap)
   (a) $H(r) = D(\overline{p})$
   (b) $V(z) = \gamma \triangleright r'$
   (c) $R = R' \oplus \delta \oplus \gamma$
   (d) γ' fresh
   (e) $H' = H[r \mapsto D([r'/p_i]\overline{p})]$
   (f) $V' = (V, x \mapsto \gamma' \triangleright p_i)$
   (g) $R'' = R' \oplus \delta \oplus \gamma'$

7. By 1.a), 5.a), 6.a), (WF-Heap)
   (a) $fields(D) = \overline{\beta\ k : E}$
   (b) $\Sigma \vdash\ \overline{p} : \overline{E}$

8. By 3.c), 5.b), 7.a), Lemma 6, $\forall j \in \{1..i\}$.
   (a) $\beta_j = \alpha_j$
   (b) $k_j = l_j$
   (c) $E_j = D_j$

9. By 1.b), 3.b), 6.b), Lemma 5
   (a) $\Sigma \vdash\ r' : D_i$
   (b) $\rho' \in \Delta$ iff $\gamma \in R$

10. By 1.a), 5.a), 6.e), 7.a,b), 8.c), 9.a), (WF-Heap), $\Sigma \vdash\ H'$.

11. By 1.b), 3.b), 6.b), (WF-Env), $\Gamma\ ;\ \Delta \setminus \rho'\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R \setminus \gamma$.
12. By 3.a,e), Lemma 2, $\rho \in \hat{\Delta}$.
13. By 3.g), 6.d,e,h), 11., 12., (WF-Env), $\Gamma\ ;\ \Delta''\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R' \oplus \delta \oplus \gamma'$.
14. By 7.b), 8.c), $\Sigma \vdash p_i : D_i$.
15. By 3.f,h), 6.f), 13., 14., (WF-Env)
    (a) $\Gamma, x : T'\ ;\ \Delta''\ ;\ \Sigma \vdash_{\varphi'} V'\ ;\ R' \oplus \delta \oplus \gamma'$ and by 1.e), 3.g), 6.d)
    (b) $\varphi' = \varphi \cup \rho'' \mapsto \gamma'$ injective
16. By 1.d), 4.a), 6.b,c), $sep(H, r, r')$.
17. Let $(w \mapsto \kappa \triangleright q), (w' \mapsto \kappa' \triangleright q') \in V$, $\kappa \in R''$.
    (a) Case $w \neq x \wedge w' \neq x$ ($\Rightarrow \kappa \neq \gamma' \wedge \kappa' \neq \gamma' \wedge \kappa \in R$). Let r_c be a common reachable object of q and q' in H'. We show that either $\kappa' = \kappa$ or else $\kappa' \notin R''$. Consider the following two cases:
        i. Case $reach(H, q, r_c) \wedge reach(H, q', r_c)$. Then, by 1.d), $\kappa' = \kappa$.
        ii. Case $\neg reach(H, q, r_c)$ (case $\neg reach(H, q', r_c)$ is symmetric). By def. H' and 13., we must have $reach(H, r', r_c) \wedge reach(H, q, r)$. Since $reach(H', q', r_c)$, we have to consider the following subcases:
            A. $reach(H, q', r)$. If $\kappa' \in R$, then by 1.d) and $reach(H, q, r)$, $\kappa' = \kappa$.
            B. $\neg reach(H, q', r)$. Then it must be the case that $reach(H, q', r_c)$. Assume that $\kappa' \in R''$. Then $\kappa' \in R$ (since $\kappa' \neq \gamma'$). By 1.d) and $reach(H, r', r_c)$, $\kappa' = \gamma$. Contradiction. Therefore, $\kappa' \notin R''$.
    (b) Case $w = x$ ($\Rightarrow \kappa = \gamma' \wedge q = p_i$).
        i. Assume $\neg sep(H', q, q')$. Then there is $\hat{r} \in dom(H')$ such that $reach(H', q, \hat{r}) \wedge reach(H', q', \hat{r})$.
        ii. By def. H', $reach(H, q, \hat{r})$.
        iii. There are two possible cases for $reach(H', q', \hat{r})$ (i):
            A. the path from q' to $\hat{r}$ uses edge (r, i). Then, by ii. and def. H', $reach(H, q', \hat{r})$.
            B. the path from q' to $\hat{r}$ does not use edge (r, i). Then, by def. H', $reach(H, q', \hat{r})$.

            In both cases we have $reach(H, q', \hat{r})$.
        iv. By ii. and iii., $\neg sep(H, q, q')$.
        v. By iv. and 1.c), $\kappa' = \gamma' \vee \kappa' \notin R$.

vi. By v. and 6.c,g), $\kappa' = \gamma' = \kappa \vee \kappa' \notin R''$.

According to cases (a) and (b), we have $separation(V', H', R'')$.

18. We show $uniqFlds(V', H', R'')$. Let $(w \mapsto \kappa \triangleright q) \in V$ such that $\neg sep(H', q, r)$ (the case where $sep(H', q, r)$ is obvious) and $\kappa \in R''$. Since $sep(H, r, r')$ (16.) we only have to consider the case where $reach(H', r', \hat{r}) \wedge reach(H', q, \hat{r})$.
    (a) By def. H' and 16., $reach(H, r', \hat{r})$.
    (b) By 17. and $\kappa \in R''$, $\kappa = \delta$.
    (c) By 1.d) and (b), $sep(H, q, r')$, and therefore, $\neg reach(H, q, \hat{r})$.
    (d) By def. H' and (c), we have $domedge_{H'}(r, i, q, \hat{r})$.
19. By 10., 15.a,b), 17., 18., and (WF-Config), $\Gamma, x : T' \,;\, \Delta'' \,;\, \Sigma \vdash H' \,;\, V' \,;\, R''$.

- Case (R-Capture). Then, $t = \texttt{let}\ x = \texttt{capture}(y, z)\ \texttt{in}\ t'$.

1. By (WF-Config)
    (a) $\Sigma \vdash H$
    (b) $\Gamma \,;\, \Delta \,;\, \Sigma \vdash_{\varphi} V \,;\, R$
    (c) $separation(V, H, R)$
    (d) $uniqFlds(V, H, R)$
    (e) φ injective
2. By the assumptions and (T-Let)
    (a) $\Gamma \,;\, \Delta \vdash \texttt{capture}(y, z) : T' \,;\, \Delta''$
    (b) $\Gamma, x : T' \,;\, \Delta'' \vdash t' : T \,;\, \Delta'$
3. By 2.a) and (T-Capture)
    (a) $\Gamma \,;\, \Delta \vdash y : \rho \triangleright C$
    (b) $\Gamma \,;\, \Delta \vdash z : \rho' \triangleright C'$
    (c) $\Delta = \Delta'' \oplus \rho$
    (d) $T' = \rho' \triangleright C$
4. By 1.b), 3.a,c), Lemma 5
    (a) $V(y) = \beta \triangleright r$
    (b) $\Sigma \vdash r : C$
    (c) $\beta \in R$ iff $\rho \in \Delta$, and therefore by 3.c)
    (d) $R = R' \oplus \beta$
5. By 1.b), 3.b,c), 4.d), Lemma 5

   (a) $V(z) = \beta' \triangleright r'$
   (b) $\Sigma \vdash r' : C'$
   (c) $\rho' \in \Delta''$ iff $\beta' \in R'$
6. By 4.a,d), 5.a), (R-Capture), $V' = (V, x \mapsto \beta' \triangleright r)$.
7. By 1.c,d), 4.a,d), 6.
   (a) $separation(V', H, R')$
   (b) $uniqFlds(V', H, R')$
8. By 1.b), 3.c), 4.c,d), (WF-Env), $\Gamma\ ;\ \Delta''\ ;\ \Sigma \vdash_\varphi V\ ;\ R'$.
9. By 3.d), 4.b), 5.c), 6., 8., (WF-Env), $\Gamma, x : T'\ ;\ \Delta''\ ;\ \Sigma \vdash_\varphi V'\ ;\ R'$.
10. By 1.a), 7.a,b), 9., and (WF-Config), $\Gamma, x : T'\ ;\ \Delta''\ ;\ \Sigma \vdash H\ ;\ V'\ ;\ R'$.

- Case (R-New). Then, $t = \texttt{let}\ x = \texttt{new}\ C(\overline{y})\ \texttt{in}\ t'$.

  1. By (WF-Config)
     (a) $\Sigma \vdash H$
     (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R$
     (c) $separation(V, H, R)$
     (d) $uniqFlds(V, H, R)$
     (e) φ injective
  2. By the assumptions and (T-Let)
     (a) $\Gamma\ ;\ \Delta \vdash \texttt{new}\ C(\overline{y}) : T'\ ;\ \Delta''$
     (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
  3. By 2.a), and (T-New)
     (a) $\Gamma\ ;\ \Delta \vdash \overline{y : \rho \triangleright D}\ ;\ \Delta$
     (b) Has been removed
     (c) $fields(C) = \overline{\alpha\ l : D}$
     (d) ρ' fresh
     (e) $\Delta = \hat{\Delta} \oplus \overline{\rho}$
     (f) $\Delta'' = \hat{\Delta} \oplus \rho'$
     (g) $T' = \rho' \triangleright C$
  4. By 1.b), 3.a), Lemma 5
     (a) $V(\overline{y}) = \overline{\beta \triangleright r}$
     (b) $\Sigma \vdash \overline{r} : \overline{D}$
     (c) $\overline{\rho} \subseteq \Delta$ iff $\overline{\beta} \subseteq R$, and therefore by 1.e) and 3.e)

   (d) $R = R'' \oplus \overline{\beta}$

5. By 4.a,d), (R-New)
   (a) $H' = (H, r \mapsto C(\overline{r}))$
   (b) $V' = (V, x \mapsto \gamma \triangleright r)$
   (c) $R' = R'' \oplus \gamma$
   (d) $r \notin dom(H)$
   (e) γ fresh
6. Define $\Sigma' := (\Sigma, r : C)$. Then, $\Sigma' \vdash r' : C' \; \forall \, r' \in dom(\Sigma)$ such that $\Sigma \vdash r' : C'$.
7. By 1.b), 3.d,e,f), 4.d), 5.c,e), $(\Gamma(y) = \rho \triangleright C \wedge V(y) = \beta \triangleright r) \Rightarrow \rho \in \Delta''$ iff $\beta \in R'$.
8. By 3.a), 4.a,b), 6., 7., (WF-Env)
   (a) $\Gamma \; ; \; \Delta'' \; ; \; \Sigma' \vdash_{\varphi'} V \; ; \; R'$, and by 1.e), 3.d), 5.e)
   (b) $\varphi' = \varphi \cup \rho' \mapsto \gamma$ injective
9. By 3.f), 5.b,c), 6., 8.a,b), (WF-Env), $\Gamma, x : \rho' \triangleright C \; ; \; \Delta'' \; ; \; \Sigma' \vdash_{\varphi'} V' \; ; \; R'$.
10. By 1.a), 5.a), 6., (WF-Heap), $\Sigma' \vdash H'$.
11. By 5.a,c), we have $separation(V, H', R')$. Let $(x' \mapsto \beta' \triangleright r') \in V$, and let r'' be a common reachable object of r and r' in H'. Then, by 5.a), $reach(H, r_i, r'')$. By 1.c), $\beta' = \beta_i \vee \beta' \notin R$. By 5.c), $\beta' \notin R'$. Therefore, we have $separation(V', H', R')$.
12. By 5.a), the unique fields of objects other than r remain unique in V', H', R'. Therefore, we only have to check whether the unique fields of r are actually unique. Let $(x' \mapsto \beta' \triangleright r') \in V$ such that $reach(H', r', r'') \wedge reach(H', r_i, r'')$. Then, by 5.a), $reach(H, r', r'') \wedge reach(H, r_i, r'')$, and therefore, $\neg sep(H, r', r_i)$. By 1.c), $\beta' = \beta_i \vee \beta' \notin R$. By 4.d) and 5.c), $\beta' \notin R'$. Therefore, we have $uniqFlds(V', H', R')$.
13. By 8.b), 9., 10., 11., 12., (WF-Config), $\Gamma, x : \rho' \triangleright C \; ; \; \Delta'' \; ; \; \Sigma' \vdash H' \; ; \; V' \; ; \; R'$.

- Case (R-Let). Then, $t = \texttt{let}\ x = t_1\ \texttt{in}\ t_2$.

1. By (WF-Config)
   (a) $\Sigma \vdash H$
   (b) $\Gamma \; ; \; \Delta \; ; \; \Sigma \vdash_{\varphi} V \; ; \; R$
   (c) $separation(V, H, R)$

(d) $uniqFlds(V, H, R)$

2. By the assumptions and (T-Let)
   (a) $\Gamma\ ;\ \Delta \vdash t_1 : T'\ ;\ \Delta''$
   (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t_2 : T\ ;\ \Delta'$
3. By (R-Let), $H, V, R, t_1 \longrightarrow H', V', R', t_1'$.
4. By the assumptions, 3., and the IH, there are Γ', $\hat{\Delta}$, and Σ' such that
   (a) $\Gamma'\ ;\ \hat{\Delta} \vdash t_1' : T'\ ;\ \Delta''$
   (b) $\Gamma'\ ;\ \hat{\Delta}\ ;\ \Sigma' \vdash H'\ ;\ V'\ ;\ R'$
   (c) $\Gamma' \supseteq \Gamma$
   (d) $\Sigma' \supseteq \Sigma$
5. By 2.b), 4.c), and Lemma 1, $\Gamma', x : T'\ ;\ \Delta'' \vdash t_2 : T\ ;\ \Delta'$.
6. By 4.a), 5., (T-Let), $\Gamma'\ ;\ \hat{\Delta} \vdash \texttt{let}\ x = t_1'\ \texttt{in}\ t_2 : T\ ;\ \Delta'$.

□

A.3 Proof of Theorem 2

Theorem 2 (Progress) *If* $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$ *and* $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$*, then either* $t = y$*, or there is a reduction* $H, V, R, t \longrightarrow H', V', R', t'$*.*

Proof: By induction on the shape of t.

- Case $t = y.l_i := z$.

  1. By (WF-Config)
     (a) $\Sigma \vdash H$
     (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R$
     (c) $separation(V, H, R)$
     (d) $uniqFlds(V, H, R)$
  2. By (T-Assign)
     (a) $\Gamma\ ;\ \Delta \vdash y : \rho \triangleright C\ ;\ \Delta$
     (b) $\Gamma\ ;\ \Delta \vdash z : \rho \triangleright D_i\ ;\ \Delta$
     (c) $fields(C) = \overline{\alpha\ l : D}$
     (d) $\alpha_i \neq \texttt{unique}$
  3. By 1.b), 2.a), Lemma 5
     (a) $\Sigma \vdash r : C$
     (b) $V(y) = \delta \triangleright r$
     (c) $\rho \in \Delta$ iff $\delta \in R$
  4. By 1.b), 2.b), Lemma 5
     (a) $\Sigma \vdash r' : D_i$
     (b) $V(z) = \delta' \triangleright r'$
     (c) $\rho \in \Delta$ iff $\delta' \in R$
  5. By 3.c) and 4.c), $\delta' = \delta$.
  6. By 3.a), (Heap-Type)
     (a) $\Sigma(r) = D$
     (b) $D <: C$
  7. By 2.a), 3.c), Lemma 2, $\delta \in R$.
  8. By 1.a), 6.a), (WF-Heap), $H(v) = D(\overline{p})$.
  9. By 1.c), 6.b), (<:-Trans), (<:-P), (WF-Class), $|\overline{p}| \geq i$.
  10. By 3.b), 4.b), 5., 7., 8., 9., rule (R-Assign) applies.

- Case $t = \texttt{let}\ x = y.m(\overline{z})\ \texttt{in}\ t'$.

  1. By (WF-Config)
     (a) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R$
     (b) $\Sigma \vdash H$
  2. By (T-Let)
     (a) $\Gamma\ ;\ \Delta \vdash y.m(\overline{z}) : T'\ ;\ \Delta''$
     (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
  3. By 2.a) and (T-Invoke)
     (a) $\Gamma\ ;\ \Delta \vdash y : \rho_1 \triangleright C_1\ ;\ \Delta$
     (b) $\Gamma\ ;\ \Delta \vdash z_{i-1} : \rho_i \triangleright D_i\ ;\ \Delta,\ i = 2 \ldots n$
     (c) $mtype(m, C_1) = \exists\delta.\ \overline{\delta \triangleright D} \to (T_R, \Delta_m)$
     (d) $\sigma = \overline{\delta \mapsto \rho} \circ \delta \mapsto \rho$ injective
     (e) $\Delta = \Delta_r \uplus \{\rho \mid \rho \in \overline{\rho}\}$
     (f) $T' = \sigma T_R$
     (g) ρ fresh
  4. By 1.a), 3.a), Lemma 5
     (a) $V(y) = \beta_1 \triangleright r_1$
     (b) $\Sigma \vdash r_1 : C_1$
     (c) $\rho_1 \in \Delta$ iff $\beta_1 \in R$
  5. By 4.b), (Heap-Type)
     (a) $\Sigma(r_1) = E_1$
     (b) $E_1 <: C_1$
  6. By 1.a), 3.b), Lemma 5
     (a) $V(\overline{z}) = \beta_2 \triangleright r_2 \ldots \beta_n \triangleright r_n$
     (b) $\Sigma \vdash r_i : D_i, i = 2 \ldots n$
     (c) $\rho_i \in \Delta$ iff $\beta_i \in R, i = 2 \ldots n$
  7. By 3.d,e)
     (a) $\overline{\rho} \subseteq \Delta$, and therefore by 4.c) and 6.c)
     (b) $\overline{\beta} \subseteq R$
  8. By 3.c), 5.a, b), (<:-Trans), (<:-P), (WF-Class)
     (a) $mbody(m, E_1) = (\overline{x}, e)$, where
     (b) $|\overline{x}| = |\overline{\delta \triangleright D}|$

9. By 1.b), 5.a), (WF-Heap), $H(r_1) = E_1(\_)$.
10. By 8.a,b), 7.b), 6.a), 9., rule (R-Invoke) applies.

- Case $t = \texttt{let}\ x = y.l_i\ \texttt{in}\ t'$: similar to case $y.l_i = z$ and therefore omitted.
- Case $t = \texttt{let}\ x = \texttt{capture}(y, z)\ \texttt{in}\ t'$.

  1. By (WF-Config), $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R$.
  2. By (T-Let)
     (a) $\Gamma\ ;\ \Delta \vdash \texttt{capture}(y, z) : T'\ ;\ \Delta''$
     (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
  3. By 2.a) and (T-Capture)
     (a) $\Gamma\ ;\ \Delta \vdash y : \rho \triangleright C\ ;\ \Delta$
     (b) $\Gamma\ ;\ \Delta \vdash z : \rho' \triangleright C'\ ;\ \Delta$
     (c) $\Delta = \hat{\Delta} \oplus \rho$
  4. By 1., 3.a), and Lemma 5
     (a) $V(y) = \beta \triangleright r$
     (b) $\Sigma \vdash r : C$
     (c) $\rho \in \Delta$ iff $\beta \in R$
  5. By 1., 3.b), and Lemma 5
     (a) $V(z) = \beta' \triangleright r'$
     (b) $\Sigma \vdash r' : C'$
     (c) $\rho' \in \Delta$ iff $\beta' \in R$
  6. By 3.c), 4.c), $R = R' \oplus \beta$.
  7. By 4.a), 5.a), 6., rule (R-Capture) applies.

- Case $t = \texttt{let}\ x = \texttt{new}\ C(\overline{y})\ \texttt{in}\ t'$.

  1. By (WF-Config)
     (a) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R$
     (b) φ injective
  2. By (T-Let)
     (a) $\Gamma\ ;\ \Delta \vdash \texttt{new}\ C(\overline{y}) : T'\ ;\ \Delta''$
     (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
  3. By 2.a) and (T-New)

   (a) $\Gamma\ ;\ \Delta \vdash \overline{y : \rho \triangleright D}$
   (b) $\Delta = \hat{\Delta} \oplus \overline{\rho}$
4. By 1.a), 3.a,b), Lemma 5
   (a) $V(\overline{y}) = \overline{\beta \triangleright r}$ and by 1.b)
   (b) $R = \hat{R} \oplus \overline{\beta}$
5. By 4.a,b), rule (R-New) applies.

- Case $t = \texttt{let}\ x = \texttt{swap}(y.l_i, z)\ \texttt{in}\ t'$: similar to case $y.l_i = z$ and therefore omitted.
- Case $t = \texttt{let}\ x = t_1\ \texttt{in}\ t_2$, where $t_1 \neq y$.
  1. By (T-Let), $\Gamma\ ;\ \Delta \vdash t_1 : T'\ ;\ \Delta''$.
  2. By the assumptions and 1., $H, V, R, t_1 \longrightarrow H', V', R', t_1'$.
  3. By 2., rule (R-Let) applies.

□

A.4 Proof of Corollary 1

Corollary 1 (Uniqueness) *If*

- $\Gamma\ ;\ \Delta \vdash \texttt{let}\ x = t\ \texttt{in}\ t' : T\ ;\ \Delta'$ *where* $t = y.m(\overline{z}) \wedge \Gamma(y) = \_ \triangleright C$
- $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$
- $mtype(m, C) = \exists \delta.\ \overline{\delta \triangleright D} \rightarrow (T_R, \Delta_m)$ *where* $\delta_i \notin \Delta_m$

then z_i *is separately-unique in* $H, V, R, \texttt{let}\ x = t\ \texttt{in}\ t'$.

Proof: Let y' be a variable in V that is not separate from z_i in H. We show that y' will not be accessed during reduction.

1. By the assumptions and (T-Let)
   (a) $\Gamma\ ;\ \Delta \vdash y.m(\overline{z}) : T'\ ;\ \Delta''$
   (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t : T\ ;\ \Delta'$
2. By 1.a), (T-Invoke)
   (a) $\Gamma(z_i) = \rho_i \triangleright C_i$

(b) $\rho_i \notin \Delta''$

3. By the assumptions and (WF-Config), $separation(V, H, R)$.

4. By 3. and the definition of $separation$

   (a) $V(y') = \beta' \triangleright \_$

   (b) $\beta' = \beta_i \vee \beta' \notin R$

   (c) $V(z_i) = \beta_i \triangleright \_$

5. By the assumptions, 1.a), Theorem 1, and Theorem 2

   (a) $H, V, R, y.m(\overline{z}) \longrightarrow^* H', V', R', e'$

   (b) $\Gamma' \,;\, \Delta'' \vdash e' : T' \,;\, \Delta''$

   (c) $\Gamma' \,;\, \Delta'' \,;\, \Sigma' \vdash H' \,;\, V' \,;\, R'$

   (d) $\Gamma' \supseteq \Gamma$

   (e) $\Sigma' \supseteq \Sigma$

   (f) $V' \supseteq V$

6. By 2.a,b), 4.c), 5.c,d,f), $\beta_i \notin R'$.

7. By 4.b), 6., $\beta' \notin R'$.

8. By Theorem 2 and the reduction rules, y' will not be accessed after the method call returns, since $\beta' \notin R'$. Therefore, z_i is separately-unique in H, V.

□

Lemma 8 *If* $\Gamma, x : T \ ; \ \Delta \vdash t : T' \ ; \ \Delta'$ *and* $S <: T$*, then* $\Gamma, x : S \ ; \ \Delta \vdash t : T' \ ; \ \Delta'$.

Proof: By induction on the typing derivation.

- Case (T-Var). Then $t = y$.
  1. By the assumptions
     (a) $\Gamma(y) = \rho \triangleright C$
     (b) $T' = \rho \triangleright C$
     (c) $\rho \in \Delta$
     (d) $\Delta' = \Delta$
  2. Consider the following cases:
     (a) Case $y = x$. Then $T = \rho \triangleright C$, $S = \rho \triangleright D$ for some $D <: C$, and $\Gamma, x : S \ ; \ \Delta \vdash y : \rho \triangleright D \ ; \ \Delta'$. By (T-Sub), $\Gamma, x : S \ ; \ \Delta \vdash y : \rho \triangleright C \ ; \ \Delta'$.
     (b) Case $y \neq x$. Then by 1.a), $(\Gamma, x : S)(y) = \rho \triangleright C$, and by 1.c), $\Gamma, x : S \ ; \ \Delta \vdash y : \rho \triangleright C \ ; \ \Delta'$.
- Cases (T-Select), (T-Assign), (T-New), (T-Invoke), (T-Capture), (T-Swap), (T-Let), and (T-Sub) follow directly from the IH.

□

A.5 Proof of Theorem 3

Theorem 3 (Isolation) *If* $\Sigma \vdash H \ ; \ \mathcal{A}$ *and* $H \ ; \ \mathcal{A} \longrightarrow H' \ ; \ \mathcal{A}'$*, then there is a* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \vdash H' \ ; \ \mathcal{A}'$.

Proof: By induction on the typing derivation with case analysis of the last rule used in the derivation.

- Case (R-Step).
  1. By (WF-Soup)
     (a) $\Sigma \vdash H \ ; \ \mathcal{A}'$
     (b) $\exists \Gamma, \Delta. \ \Gamma \ ; \ \Delta \ ; \ \Sigma \vdash H \ ; \ A$
     (c) $\forall A'' \in \mathcal{A}'. \ isolated(H, A, A'')$
  2. By (R-Step)

   (a) $A = (\langle V, R, t\rangle, M)$
   (b) $H, V, R, t \longrightarrow H', V', R', t'$
   (c) $A' = (\langle V', R', t'\rangle, M)$
3. By 1.b) and (WF-Actor)
   (a) $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$
   (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$
   (c) $sepEnvMbox(H, V, R, M)$
   (d) $sepMbox(H, M)$
   (e) $uniqFldsMbox(H, M)$
4. By 2.b), 3.a, b), and Theorem 1, there are $\Gamma', \Delta'', \Sigma'$ such that
   (a) $\Gamma' \supseteq \Gamma$
   (b) $\Sigma' \supseteq \Sigma$
   (c) $\Gamma'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
   (d) $\Gamma'\ ;\ \Delta''\ ;\ \Sigma' \vdash H'\ ;\ V'\ ;\ R'$
5. We prove $\Sigma' \vdash H'\ ;\ \mathcal{A}'$ by induction on $\mathcal{A}'$. Let $\mathcal{A}' = \{B\} \cup \mathcal{B}$. If $\mathcal{B} \neq \emptyset$ we can assume $\Sigma' \vdash H'\ ;\ \mathcal{B}$.
6. Let $B = (S_B, M_B)$ where $S_B = \langle V_B, R_B, t_B\rangle$. By 1.c)
   (a) $isolated(H, \langle V, R, t\rangle, S_B)$
   (b) $isolated(H, \langle V, R, t\rangle, M_B)$
   (c) $isolated(H, S_B, M)$
   (d) $isolated(H, M, M_B)$
7. By 1.a), 5., and (WF-Soup)
   (a) $\forall B' \in \mathcal{B}.\ isolated(H, B, B')$
   (b) $\Gamma_B\ ;\ \Delta_B\ ;\ \Sigma \vdash H\ ;\ B$ for some Γ_B, Δ_B
8. By 7.b) and (WF-Actor)
   (a) $\Gamma_B\ ;\ \Delta_B \vdash t_B : T_B\ ;\ \Delta'_B$
   (b) $\Gamma_B\ ;\ \Delta_B\ ;\ \Sigma \vdash H\ ;\ V_B\ ;\ R_B$
   (c) $sepEnvMbox(H, V_B, R_B, M_B)$
   (d) $sepMbox(H, M_B)$
   (e) $uniqFldsMbox(H, M_B)$
9. By 8.b) and (WF-Config)
   (a) $\Gamma_B\ ;\ \Delta_B\ ;\ \Sigma \vdash_\varphi V_B\ ;\ R_B$ where φ injective
   (b) $separation - imm(V_B, H, R_B)$

(c) $uniqFlds - imm(V_B, H, R_B)$

(d) $deep - imm(H)$

10. Let $(x \mapsto \beta \triangleright r) \in V'$ such that $\beta \in R'$ and $V_B(z) = \delta \triangleright q$ such that $\delta \in R_B$ and $m \in M_B, m' \in M$.

(a) Case $r \in dom(H)$. Let $r' \in dom(H)$ such that $sep{-}imm(H, r, r') \wedge \neg sep - imm(H', r, r')$.

i. By (R-Assign) and (R-Swap), there are $y, y' \in dom(V)$ such that

A. $V(y) = \beta \triangleright p \wedge \beta \in R \wedge reach(H, r, p)$

B. $V(y') = \beta' \triangleright p' \wedge \beta' \in R \wedge \neg sep - imm(H, p', r')$

ii. By i.A,B), 6.a), and def. isolated

A. $sep - imm(H, r, q)$

B. $sep - imm(H, p', q)$, and therefore

C. $sep - imm(H', r, q)$, and therefore

D. $isolated(H', \langle V', R', t' \rangle, \langle V_B, R_B, t_B \rangle$

iii. By i.A,B), 6.b), and def. isolated

A. $sep - imm(H, r, m)$

B. $sep - imm(H, p', m)$, and therefore

C. $sep - imm(H', r, m)$, and therefore

D. $isolated(H', \langle V', R', t' \rangle, M_B)$

iv. Analogously, we have by i.A,B) and 6.c), $isolated(H', S_B, M)$

v. Analogously, we have by i.A,B) and 6.d), $isolated(H', M, M_B)$

(b) Case $r \notin dom(H)$. This case is proved analogously to case (a) using (R-New).

11. Let $\hat{q}, s \in dom(H)$ such that $reach(H, q, \hat{q}) \wedge H'(s) \neq H(s)$. By (R-Assign) and (R-Swap), there is $y \in dom(V)$ such that $V(y) = \beta \triangleright s \wedge \beta \in R$.

12. By 6.a) and def. isolated, $sep - imm(H, q, s)$, and therefore

(a) $H'(\hat{q}) = H(\hat{q})$

(b) $reach(H', q, \hat{q})$

(c) $\forall q' \in dom(H').\, reach(H', q, q') \Rightarrow q' \in dom(H) \wedge reach(H, q, q')$

13. By 9.b,c,d), 12.a,b,c), def. $separation - imm$, def. $uniqFlds - imm$, def. $sepEnvMbox$, and def. $deep - imm$

(a) $separation - imm(V_B, H', R_B)$

(b) $uniqFlds - imm(V_B, H', R_B)$

(c) $sepEnvMbox(H', V_B, R_B, M_B)$

(d) $deep - imm(H')$

14. By 6.b), 8.d), 11., $\forall r, r' \in M_B$.

    (a) $sep - imm(H, r, s)$

    (b) $sep - imm(H, r', s)$, and therefore

    (c) $sep - imm(H', r, r')$

15. By 8.e) and 9.c), $\forall r \in M_B.\ H'(q) = C(\overline{p}) \Rightarrow$ $(\forall i \in uniqInd(C).\ reachable(H', p_i, r') \Rightarrow (domedge(H', q, i, r, r') \vee immutable(r', H')))$

16. By 4.b), 9.a), and (WF-Env), $\Gamma_B\ ;\ \Delta_B\ ;\ \Sigma' \vdash_\varphi V_B\ ;\ R_B$ where φ injective

17. By 4.d) and (WF-Config), $\Sigma' \vdash H'$.

18. By 13.a,b,d), 16., 17., and (WF-Config), $\Gamma_B\ ;\ \Delta_B\ ;\ \Sigma' \vdash H'\ ;\ V_B\ ;\ R_B$.

19. By 8.a), 13.c), 14.c), 15., 18., and (WF-Actor), $\Gamma_B\ ;\ \Delta_B\ ;\ \Sigma' \vdash H'\ ;\ B$.

20. Let $B' = (\langle V_{B'}, R_{B'}, t_{B'}\rangle, M_{B'})$. Let $(y \mapsto \delta \triangleright p) \in V_B, (y' \mapsto \delta' \triangleright p') \in V_{B'}$ such that $\delta \in R_B \wedge \delta' \in R_{B'}$. By $isolated(H, B, B')$, $sep - imm(H, p, p')$.

21. Let $r, r' \in dom(H)$ roots of V such that $sep - imm(H, r, r') \wedge \neg sep - imm(H', r, r')$.

    (a) By (R-Assign) and (R-Swap), there are $z, z' \in dom(V)$ such that

        i. $V(z) = \beta \triangleright s \wedge \beta \in R \wedge reach(H, r, s)$

        ii. $V(z') = \beta' \triangleright s' \wedge \beta' \in R \wedge \neg sep - imm(H, s', r')$

    (b) By a.i,ii), def. $isolated$, and def. $separation - imm$, there are $w, w' \in dom(V)$ such that $V(w) = \beta \triangleright r \wedge V(w') = \beta' \triangleright r'$.

    (c) By a.i,ii), 6.a), and def. $isolated$

        i. $sep - imm(H, r, p)$

        ii. $sep - imm(H, s', p)$

        iii. $sep - imm(H, r', p)$, and therefore

        iv. $sep - imm(H', r, p) \wedge sep - imm(H', r', p) \wedge sep - imm(H', s', p)$

    (d) Analogously, we have $sep - imm(H, q, p')$ for $q \in \{r, r', s'\}$.

    (e) Therefore, $sep - imm(H', p, p')$.

22. By 20., 21., and def. $isolated$, $isolated(H', \langle V_B, R_B, t_B\rangle, \langle V_{B'}, R_{B'}, t_{B'}\rangle)$.

23. Analogously to 22., we have
    (a) $isolated(H', \langle V_{B'}, R_{B'}, t_{B'}\rangle, M_B)$
    (b) $isolated(H', \langle V_B, R_B, t_B\rangle, M_{B'})$
    (c) $isolated(H', M_B, M_{B'})$
24. By 22., 23.a,b,c), and def. $isolated$, $isolated(H', B, B')$.
25. By 5., 19., 24., and (WF-Soup), $\Sigma' \vdash H' ; \mathcal{A}'$.
26. Let $(x \mapsto \beta \triangleright r) \in V'$ such that $\beta \in R'$ and $m, m' \in M$.
    (a) Case $r \in dom(H)$. Let $r' \in dom(H)$ such that $sep{-}imm(H, r, r') \wedge \neg sep - imm(H', r, r')$.
        i. By (R-Assign) and (R-Swap), there are $y, y' \in dom(V)$ such that
            A. $V(y) = \beta \triangleright p \wedge \beta \in R \wedge reach(H, r, p)$
            B. $V(y') = \beta' \triangleright p' \wedge \beta' \in R \wedge \neg sep - imm(H, p', r')$
        ii. By i.A,B) and 3.c)
            A. $sep - imm(H, p, m)$
            B. $sep - imm(H, p', m)$
            C. $sep - imm(H, p, m')$
            D. $sep - imm(H, p', m')$, and therefore
            E. $sep - imm(H', r, m)$
            F. $sep - imm(H', r, m')$
            G. $sep - imm(H', m, m')$
        iii. By ii.A,B) and 3.e), $H'(q) = C(\overline{p}) \Rightarrow (\forall i \in uniqInd(C).$ $reach(H', p_i, m'') \Rightarrow$ $(domedge(H', q, i, m, m'') \vee immutable(m'', H')))$.
    (b) Case $r \notin dom(H)$.
        i. By (R-New)
            A. $t = \texttt{let}\ x = \texttt{new}\ C(\overline{y})\ \texttt{in}\ t'$
            B. $V(\overline{y}) = \overline{\beta \triangleright r}$
            C. $H' = (H, r \mapsto C(\overline{r}))$
            D. β fresh
            E. $R = R'' \oplus \overline{\beta}$
            F. $R' = R'' \oplus \beta$
        ii. By i.B,E), 3.c), $\forall r_i \in \overline{r}$.
            A. $sep - imm(H, r_i, m)$
            B. $sep - imm(H, r_i, m')$

iii. By i.C), ii.
   A. $sep - imm(H', r, m)$
   B. $sep - imm(H', r, m')$
   C. $sep - imm(H', m, m')$
iv. By iii.A) and 3.e), $H'(q) = C(\overline{p}) \Rightarrow (\forall i \in uniqInd(C).$
$reach(H', p_i, m'') \Rightarrow$
$(domedge(H', q, i, m, m'') \vee immutable(m'', H')))$.

27. By 4.c,d), 26., and (WF-Actor), $\Gamma' \ ; \ \Delta'' \ ; \ \Sigma' \vdash H' \ ; \ A'$.
28. By 10., 25., 27., and (WF-Soup), $\Sigma' \vdash H' \ ; \ \{A'\} \cup \mathcal{A}'$.

- Case (R-Send). $\mathcal{A} = \{A\} \cup \mathcal{A}''$

1. By (WF-Soup)
   (a) $\Sigma \vdash H \ ; \ \mathcal{A}''$
   (b) $\exists \Gamma, \Delta. \ \Gamma \ ; \ \Delta \ ; \ \Sigma \vdash H \ ; \ A$
   (c) $\forall A'' \in \mathcal{A}''. \ isolated(H, A, A'')$
2. Let $\mathcal{A}'' = \{B\} \cup \mathcal{A}'''$ where $B = (\langle V_B, R_B, t_B \rangle, M_B)_r$. By (R-Send)
   (a) $A = (\langle V, R, y \ \texttt{!} \ z \rangle, M_A)$
   (b) $V(y) = \beta \triangleright r$
   (c) $V(z) = \beta' \triangleright r'$
   (d) $R = R' \oplus \beta'$
   (e) $A' = (\langle V, R'', y \rangle, M_A)$
   (f) $B' = (\_, \{r'\} \cup M_B)$
   (g) $H(r') = C(\_)$
   (h) $R'' = R' \cup \{\beta' \mid C \in I\}$
3. By 1.b), 2.a), and (WF-Actor)
   (a) $\Gamma \ ; \ \Delta \vdash y \ \texttt{!} \ z : T \ ; \ \Delta'$
   (b) $\Gamma \ ; \ \Delta \ ; \ \Sigma \vdash H \ ; \ V \ ; \ R$
   (c) $sepEnvMbox(H, V, R, M_A)$
   (d) $sepMbox(H, M_A)$
   (e) $uniqFldsMbox(H, M_A)$
4. By 3.a) and (T-Send)
   (a) $\Gamma \ ; \ \Delta \vdash y : \rho \triangleright \texttt{Actor} \ ; \ \Delta$
   (b) $\Gamma \ ; \ \Delta \vdash z : \rho' \triangleright C \ ; \ \Delta$
   (c) $\Delta = \Delta' \oplus \{\rho' \mid C \notin I\}$

5. By 1.a) and (WF-Soup)
   (a) $\Sigma \vdash H \; ; \; \mathcal{A}'''$
   (b) $\exists\, \Gamma_B, \Delta_B.\, \Gamma_B \; ; \; \Delta_B \; ; \; \Sigma \vdash H \; ; \; B$
   (c) $\forall D \in \mathcal{A}'''.\, isolated(H, B, D)$
6. By 5.b) and (WF-Actor)
   (a) $\Gamma_B \; ; \; \Delta_B \vdash t_B : T_B \; ; \; \Delta'_B$
   (b) $\Gamma_B \; ; \; \Delta_B \; ; \; \Sigma \vdash H \; ; \; V_B \; ; \; R_B$
   (c) $sepEnvMbox(H, V_B, R_B, M_B)$
   (d) $sepMbox(H, M_B)$
   (e) $uniqFldsMbox(H, M_B)$
7. By 1.c) and def. $isolated$
   (a) $isolated(H, \langle V, R, \_\rangle, \langle V_B, R_B, \_\rangle)$
   (b) $isolated(H, \langle V, R, \_\rangle, M_B)$
   (c) $isolated(H, \langle V_B, R_B, \_\rangle, M_A)$
   (d) $isolated(H, M_A, M_B)$
8. By 2.c,d) and 7.a), $\forall (x \mapsto \delta \triangleright p) \in V_B.\, \delta \in R_B \Rightarrow sep{-}imm(H, p, r')$.
9. By 2.c,d) and 7.b), $\forall p \in M_B.\, sep - imm(H, p, r')$.
10. Let $D \in \mathcal{A}'''$ such that $D = (\langle V_D, R_D, t_D\rangle, M_D)$. By 5.c) and def. $isolated$
    (a) $isolated(H, \langle V_B, R_B, t_B\rangle, \langle V_D, R_D, t_D\rangle)$
    (b) $isolated(H, \langle V_B, R_B, t_B\rangle, M_D)$
    (c) $isolated(H, \langle V_D, R_D, t_D\rangle, M_B)$
    (d) $isolated(H, M_B, M_D)$
11. By 1.c) and def. $isolated$, $isolated(H, \langle V, R, \_\rangle, \langle V_D, R_D, t_D\rangle)$.
12. By 10.c) and 11., $isolated(H, \langle V_D, R_D, t_D\rangle, \{r'\} \cup M_B)$.
13. By 1.c) and def. $isolated$
    (a) $isolated(H, \langle V, R, \_\rangle, M_D)$, and by 10.d)
    (b) $isolated(H, \{r'\} \cup M_B, M_D)$
14. Has been removed.
15. By 2.d,h), 7.a), and def. $isolated$, $isolated(H, \langle V, R'', y\rangle, \langle V_B, R_B, \_\rangle)$.
16. By 2.d,h), 7.b), and def. $isolated$, $isolated(H, \langle V, R'', y\rangle, \{r'\} \cup M_B)$.
17. By 2.c) and 3.c), $\forall p \in M_A.\, sep - imm(H, p, r')$.
18. By 7.d) and 17., $isolated(H, M_A, \{r'\} \cup M_B)$.

19. By 3.b) and (WF-Config)
    (a) $\Sigma \vdash H$
    (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_{\varphi} V\ ;\ R$ where φ injective
    (c) $separation - imm(V, H, R)$
    (d) $uniqFlds - imm(V, H, R)$
    (e) $deep - imm(H)$
20. By 2.c,d), 19.d), and def. $uniqFlds - imm$, $H(q) = C(\overline{p}) \Rightarrow (\forall i \in uniqInd(C).\, reach(H, p_i, r'') \Rightarrow domedge(H, q, i, r', r'') \vee immutable(r'', H))$.
21. By 2.c,d), 4.b,c), 19.b), and (WF-Env), $\varphi(\rho') = \beta'$.
22. By 19.b), $\varphi|_{\Delta'}$ injective.
23. We show $\Gamma\ ;\ \Delta'\ ;\ \Sigma \vdash_{\varphi|_{\Delta'}} V\ ;\ R''$.
    (a) Case $C \in I$. Then $R'' = R \wedge \Delta = \Delta'$. By 19.b), $\Gamma\ ;\ \Delta'\ ;\ \Sigma \vdash_{\varphi|_{\Delta'}} V\ ;\ R''$.
    (b) Case $C \notin I$. Then $R'' = R' \wedge \Delta = \Delta' \oplus \rho'$. By 2.d), 21., 22., and (WF-Env), $\Gamma\ ;\ \Delta'\ ;\ \Sigma \vdash_{\varphi|_{\Delta'}} V\ ;\ R''$.
24. By 4.a,c) and (T-Var), $\Gamma\ ;\ \Delta' \vdash y : \rho \triangleright \texttt{Actor}\ ;\ \Delta'$.
25. By 2.d,h), 19.c,d)
    (a) $separation - imm(V, H, R'')$
    (b) $uniqFlds - imm(V, H, R'')$
26. By 19.a,e), 22., 23., 25.a,b), and (WF-Config), $\Gamma\ ;\ \Delta'\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R''$.
27. By 2.d), 3.c,d,e), 24., 26., and (WF-Actor), $\Gamma\ ;\ \Delta'\ ;\ \Sigma \vdash H\ ;\ (\langle V, R'', y\rangle, M_A)$.
28. By 6.e) and 20., $\forall p \in \{r'\} \cup M_B.\, H(q) = C(\overline{p}) \Rightarrow (\forall i \in uniqInd(C).\, reach(H, p_i, p') \Rightarrow domedge(H, q, i, p, p') \vee immutable(p', H))$ and therefore, $uniqFldsMbox(H, \{r'\} \cup M_B)$.
29. By 7.c), 15., 16., 18., and def. $isolated$, $isolated(H, A', B')$ where $B' = (\langle V_B, R_B, t_B\rangle, \{r'\} \cup M_B)$.
30. By 1.c) and 29., $\forall C \in \hat{\mathcal{A}}.\, isolated(H, A', C)$ where $\hat{\mathcal{A}} = \{B'\} \cup \mathcal{A}'''$.
31. By 10.a,b), 12., 13.b), and def. $isolated$, $\forall D \in \mathcal{A}'''.\, isolated(H, B', D)$.
32. By 6.d) and 9., $sepMbox(H, \{r'\} \cup M_B)$.
33. By 6.c) and 8., $sepEnvMbox(H, V_B, R_B, \{r'\} \cup M_B)$.
34. By 6.a,b), 28., 32., 33., and (WF-Actor), $\exists\, \Gamma_B, \Delta_B.\ \Gamma_B\ ;\ \Delta_B\ ;\ \Sigma \vdash H\ ;\ B'$.

35. By 1.a) and (WF-Soup), $\Sigma \vdash H \; ; \; \mathcal{A}'''$.
36. By 31., 34., 35., and (WF-Soup), $\Sigma \vdash H \; ; \; \hat{\mathcal{A}}$ where $\hat{\mathcal{A}} = \{B'\} \cup \mathcal{A}'''$.
37. By 27., 30., 36., and (WF-Soup), $\Sigma \vdash H \; ; \; \{A', B'\} \cup \mathcal{A}'''$.

- Case (R-Receive). $\mathcal{A} = \{A\} \cup \mathcal{A}''$

  1. By (WF-Soup)
     (a) $\Sigma \vdash H \; ; \; \mathcal{A}''$
     (b) $\exists\, \Gamma, \Delta.\, \Gamma \; ; \; \Delta \; ; \; \Sigma \vdash H \; ; \; A$
     (c) $\forall A'' \in \mathcal{A}''.\, isolated(H, A, A'')$
  2. By (R-Receive)
     (a) $A = (\langle V, R, \texttt{let}\ x = \texttt{receive}[C]\ \texttt{in}\ t' \rangle, \{r\} \cup M)$
     (b) $H(r) = C(\_)$
     (c) β fresh
     (d) $V' = (V, x \mapsto \beta \triangleright r)$
     (e) $A' = (\langle V', R \cup \{\beta\}, t' \rangle, M)$
  3. By 1.b), 2.a), (WF-Actor)
     (a) $\Gamma \; ; \; \Delta \vdash \texttt{let}\ x = \texttt{receive}[C]\ \texttt{in}\ t' : T \; ; \; \Delta'$
     (b) $\Gamma \; ; \; \Delta \; ; \; \Sigma \vdash H \; ; \; V \; ; \; R$
     (c) $sepEnvMbox(H, V, R, \{r\} \cup M)$
     (d) $sepMbox(H, \{r\} \cup M)$
     (e) $uniqFldsMbox(H, \{r\} \cup M)$
  4. By 3.a), (T-Let)
     (a) $\Gamma \; ; \; \Delta \vdash \texttt{receive}[C] : T' \; ; \; \Delta''$
     (b) $\Gamma, x : T' \; ; \; \Delta'' \vdash t' : T \; ; \; \Delta'$
  5. By 4.a), (T-Recv)
     (a) $T' = \rho \triangleright C$
     (b) ρ fresh
     (c) $\Delta'' = \Delta \oplus \rho$
  6. By 3.b), (WF-Config)
     (a) $\Sigma \vdash H$
     (b) $\Gamma \; ; \; \Delta \; ; \; \Sigma \vdash_{\varphi} V \; ; \; R$ where φ injective
     (c) $separation - imm(V, H, R)$
     (d) $uniqFlds - imm(V, H, R)$

   (e) $deep - imm(H)$

7. By 2.c) and 3.c), $\forall y \in dom(V), \delta \in Guards, p \in dom(H).\ V(y) = \delta \triangleright p \wedge \delta \in R \Rightarrow \delta \neq \beta$.
8. By 5.b), 6.b), and 7.
   (a) $\varphi' = (\varphi, \rho \mapsto \beta)$ injective, and therefore
   (b) $\Gamma\ ;\ \Delta \oplus \rho\ ;\ \Sigma \vdash_{\varphi'} V\ ;\ R \cup \{\beta\}$
9. By 2.b), 6.a), and (WF-Heap), $\Sigma(r) = C$.
10. By 9., $C <: C$, and (Heap-Type), $\Sigma \vdash r : C$.
11. By 3.c), $\forall (y \mapsto \delta \triangleright p) \in V.\ \delta \in R \Rightarrow sep - imm(H, p, r)$.
12. By 3.d), $\forall p \in M.\ sep - imm(H, p, r)$.
13. Has been removed.
14. Let $B \in \mathcal{A}''$ ($\mathcal{A} = \{A\} \cup \mathcal{A}''$) such that $B = (\langle V_B, R_B, t_B\rangle, M_B)$. By 1.c) and def. $isolated$
   (a) $isolated(H, \langle V, R, \_\rangle, \langle V_B, R_B, \_\rangle)$
   (b) $isolated(H, \langle V, R, \_\rangle, M_B)$
   (c) $isolated(H, \langle V_B, R_B, \_\rangle, \{r\} \cup M)$
   (d) $isolated(H, \{r\} \cup M, M_B)$
15. By 14.a,c), and def. $isolated$,
    $isolated(H, \langle V', R \cup \{\beta\}, t'\rangle, \langle V_B, R_B, t_B\rangle)$.
16. By 14.b,d), and def. $isolated$, $isolated(H, \langle V', R \cup \{\beta\}, t'\rangle, M_B)$.
17. By 14.c,d), 15., 16., and def. $isolated$,
    $\forall A'' \in \mathcal{A}''.\ isolated(H, A', A'')$ where $A' = (\langle V', R \cup \{\beta\}, t'\rangle, M)$.
18. By 3.c), 7., and 12., $\forall (x \mapsto \delta \triangleright p) \in V', p' \in M.\ \delta \in R \cup \{\beta\} \Rightarrow sep{-}imm(H, p, p')$, and therefore $sepEnvMbox(H, V', R{\cup}\{\beta\}, M)$.
19. By 5.c), 8.a,b), 10., and (WF-Env), $\Gamma'\ ;\ \Delta''\ ;\ \Sigma \vdash_{\varphi'} V'\ ;\ R \cup \{\beta\}$ where φ' injective and $\Gamma' = (\Gamma, x : \rho \triangleright C)$.
20. By 6.c), 7., 11., and def. $separation - imm$
   (a) $\forall (y \mapsto \delta \triangleright p) \in V.\ \delta \in R \Rightarrow sep - imm(H, p, r)$, and therefore
   (b) $\forall (y \mapsto \delta \triangleright p), (y' \mapsto \delta' \triangleright p') \in (V, x \mapsto \beta \triangleright r).\ (\delta \neq \delta' \wedge \{\delta, \delta'\} \subseteq R \cup \{\beta\}) \Rightarrow sep - imm(H, p, p')$, and therefore by def. $separation - imm$
   (c) $separation - imm(V', H, R \cup \{\beta\})$
21. By 3.e), 6.d), and 7.

(a) $\forall(y \mapsto \delta \triangleright p) \in (V, x \mapsto \beta \triangleright r).\ H(q) = C(\overline{p}) \Rightarrow (\forall i \in uniqInd(C).\ \delta \in R \cup \{\beta\} \wedge reach(H, p_i, p') \Rightarrow (domedge(H, q, i, p, p') \vee immutable(H, p')))$, and therefore by def. $uniqFlds - imm$

(b) $uniqFlds - imm(V', H, R \cup \{\beta\})$

22. By 3.b), 6.e), 19., 20.c), 21.b), and (WF-Config), $\Gamma'\ ;\ \Delta''\ ;\ \Sigma \vdash H\ ;\ V'\ ;\ R \cup \{\beta\}$.
23. By 3.e), $\forall p \in M.\ H(q) = C(\overline{p}) \Rightarrow (\forall i \in uniqInd(C).\ reach(H, p_i, p') \Rightarrow (domedge(H, q, i, p, p') \vee immutable(H, p')))$ and therefore, $uniqFldsMbox(H, M)$.
24. By 3.d), 4.b), 18., 22., 23., and (WF-Actor), $\Gamma'\ ;\ \Delta''\ ;\ \Sigma \vdash H\ ;\ A'$.
25. By 1.a), 17., 24., and (WF-Soup), $\Sigma \vdash H\ ;\ \{A'\} \cup \mathcal{A}''$.

- Case (R-Actor). $\mathcal{A} = \{A\} \cup \mathcal{A}''$

1. By (WF-Soup)
   (a) $\Sigma \vdash H\ ;\ \mathcal{A}''$
   (b) $\exists\, \Gamma, \Delta.\ \Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ A$
   (c) $\forall B \in \mathcal{A}''.\ isolated(H, A, B)$
2. By (R-Actor)
   (a) $A = (\langle V, R, \texttt{let}\ x = \texttt{actor}\ C\ \texttt{in}\ t' \rangle, M)$
   (b) $r \notin dom(H)$
   (c) $H' = (H, r \mapsto C(\epsilon))$
   (d) $V' = (V, x \mapsto \beta \triangleright r)$
   (e) $mbody(\texttt{act}, C) = (\texttt{this}, e)$
   (f) $A' = (\langle V', R \cup \{\beta\}, t' \rangle, M)$
   (g) $B = (\langle \texttt{this} \mapsto \beta' \triangleright r, \{\beta'\}, \texttt{let}\ y = e\ \texttt{in}\ y \rangle, \emptyset)$
   (h) β, β' fresh
   (i) $C \in I$
3. By 1.b) and (WF-Actor)
   (a) $\Gamma\ ;\ \Delta \vdash \texttt{let}\ x = \texttt{actor}\ C\ \texttt{in}\ t' : T\ ;\ \Delta'$
   (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash H\ ;\ V\ ;\ R$
   (c) $sepEnvMbox(H, V, R, M)$
   (d) $sepMbox(H, M)$
   (e) $uniqFldsMbox(H, M)$

4. By 3.a) and (T-Let)
   (a) $\Gamma\ ;\ \Delta \vdash \texttt{actor}\, C : T'\ ;\ \Delta''$
   (b) $\Gamma, x : T'\ ;\ \Delta'' \vdash t' : T\ ;\ \Delta'$
5. By 4.a) and (T-Actor)
   (a) $C <: \texttt{Actor}$
   (b) ρ fresh
   (c) $T' = \rho \triangleright \texttt{Actor}$
   (d) $\Delta'' = \Delta \oplus \rho$
6. Has been removed.
7. By 3.b) and (WF-Config)
   (a) $\Sigma \vdash H$
   (b) $\Gamma\ ;\ \Delta\ ;\ \Sigma \vdash_\varphi V\ ;\ R$ where φ injective
   (c) $separation - imm(V, H, R)$
   (d) $uniqFlds - imm(V, H, R)$
   (e) $deep - imm(H)$
8. Define $\Sigma' := (\Sigma, r \mapsto C)$.
9. By 2.c), 7.a), 8., and (WF-Heap), $\Sigma' \vdash H'$.
10. By 5.a), 8., and (Heap-Type), $\Sigma' \vdash r : \texttt{Actor}$.
11. By 2.b), 7.b), 8., and (WF-Env), $\Gamma\ ;\ \Delta\ ;\ \Sigma' \vdash_\varphi V\ ;\ R$.
12. By 2.d), 5.c), 7.b), 10., 11., and (WF-Env), $\Gamma, x : T'\ ;\ \Delta''\ ;\ \Sigma' \vdash_{\varphi'} V'\ ;\ R \cup \{\beta\}$ where $\varphi' = (\varphi, \rho \mapsto \beta)$ injective.
13. By 2.b,c,d), 7.c,d,e), def. $separation - imm$, def. $uniqFlds - imm$, and def. $deep - imm$
   (a) $separation - imm(H', V', R \cup \{\beta\})$
   (b) $uniqFlds - imm(H', V', R \cup \{\beta\})$
   (c) $deep - imm(H')$
14. By 9., 12., 13.a,b,c), and (WF-Config), $\Gamma, x : T'\ ;\ \Delta''\ ;\ \Sigma' \vdash H'\ ;\ V'\ ;\ R \cup \{\beta\}$.
15. By 1.c), 2.b,c,d,f), and def. $isolated$, $\forall B' \in \mathcal{A}''.\, isolated(H', A', B')$.
16. By 1.a), 2.b,c), 8., and (WF-Soup), $\Sigma' \vdash H'\ ;\ \mathcal{A}''$.
17. By 2.b,c,d), 3.c), and def. $sepEnvMbox$, $sepEnvMbox(H', V', R \cup \{\beta\}, M)$.
18. By 2.b,c), 3.d), and def. $sepMbox$, $sepMbox(H', M)$.

19. By 2.b,c), 3.e), and def. $uniqFldsMbox$, $uniqFldsMbox(H', M)$.
20. By 4.b), 5.d), 14., 17., 18., 19., and (WF-Actor), $\Gamma, x : T' \; ; \; \Delta'' \; ; \; \Sigma' \vdash H' \; ; \; A'$.
21. Let D be the most direct super class of C that defines act. Then, by (WF-Class)
    (a) $D \vdash \texttt{def}\ act[\hat{\delta}](\texttt{this} : \delta \triangleright D) : (\delta \triangleright \texttt{Actor}, \{\delta\}) = e$, and by (WF-Method)
    (b) $(\texttt{this} \mapsto \delta \triangleright D) \; ; \; \{\delta\} \vdash e : \delta \triangleright \texttt{Actor} \; ; \; \{\delta\}$
22. By 21.b), Lemma 4, and Lemma 8, $(\texttt{this} \mapsto \rho' \triangleright C) \; ; \; \{\rho'\} \vdash e : \rho' \triangleright \texttt{Actor} \; ; \; \{\rho'\}$.
23. By (T-Var), $(\texttt{this} \mapsto \rho' \triangleright C, y \mapsto \rho' \triangleright \texttt{Actor}) \; ; \; \{\rho'\} \vdash y : \rho' \triangleright \texttt{Actor} \; ; \; \{\rho'\}$.
24. By 5.a), 22., 23., and (T-Let), $(\texttt{this} \mapsto \rho' \triangleright C) \; ; \; \{\rho'\} \vdash \texttt{let}\ y = e\ \texttt{in}\ y : \rho' \triangleright \texttt{Actor} \; ; \; \{\rho'\}$.
25. By 8., $C <: C$, and (Heap-Type), $\Sigma' \vdash r : C$.
26. By 25. and (WF-Env), $(\texttt{this} \mapsto \rho' \triangleright C) \; ; \; \{\rho'\} \; ; \; \Sigma' \vdash_{\varphi_B} (\texttt{this} \mapsto \beta' \triangleright r) \; ; \; \{\beta'\}$ where $\varphi_B = (\rho' \mapsto \beta')$ injective.
27. By 2.b,c), 7.c,d,e), def. $separation - imm$, def. $uniqFlds - imm$, and def. $deep - imm$
    (a) $separation - imm(\texttt{this} \mapsto \beta' \triangleright r, H', \{\beta'\})$
    (b) $uniqFlds - imm(\texttt{this} \mapsto \beta' \triangleright r, H', \{\beta'\})$
    (c) $deep - imm(H')$
28. By 9., 26., 27.a,b,c), and (WF-Config), $(\texttt{this} \mapsto \rho' \triangleright C) \; ; \; \{\rho'\} \; ; \; \Sigma' \vdash H' \; ; \; (\texttt{this} \mapsto \beta' \triangleright r) \; ; \; \{\beta'\}$.
29. By def. $sepEnvMbox$, def. $sepMbox$, and def. $uniqFldsMbox$
    (a) $sepEnvMbox(H', (\texttt{this} \mapsto \beta' \triangleright r), \{\beta'\}, \emptyset)$
    (b) $sepMbox(H', \emptyset)$
    (c) $uniqFldsMbox(H', \emptyset)$
30. By 24., 28., 29.a,b,c), and (WF-Actor), $(\texttt{this} \mapsto \rho' \triangleright C) \; ; \; \{\rho'\} \; ; \; \Sigma' \vdash H' \; ; \; B$.
31. By 2.b,c) and def. $isolated$, $\forall C \in \mathcal{A}''.\ isolated(H', B, C)$.
32. By 16., 30., 31., and (WF-Soup), $\Sigma' \vdash H' \; ; \; \{B\} \cup \mathcal{A}''$.
33. By 15. and $isolated(H', A', B)$,
    $\forall B' \in \{B\} \cup \mathcal{A}''.\ isolated(H', A', B')$.
34. By 20., 32., 33., and (WF-Soup), $\Sigma' \vdash H' \; ; \; \{A', B\} \cup \mathcal{A}''$.

□

Bibliography

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 289–302. USENIX, June 2002.

[2] Gul Agha. Concurrent object-oriented programming. *Comm. ACM*, 33(9):125–141, 1990.

[3] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[4] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, June 1997.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, January/February 2006.

[6] Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 98–109. ACM, June 2009.

[7] Joe Armstrong. Erlang — a survey of the language and its industrial applications. In *Proceedings of the 9th Symposium on Industrial Applications of Prolog (INAP'96)*, pages 16–18, October 1996.

[8] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[9] Joshua S. Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. Flexible task graphs: a unified restricted thread

programming model for Java. In *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 1–11. ACM, June 2008.

[10] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 258–268. ACM, June 1998.

[11] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 382–400. ACM, October 2000.

[12] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*, pages 227–244. ACM, October 2008.

[13] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst*, 26(5):769–804, 2004.

[14] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 301–320. ACM, October 2007.

[15] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.

[16] R. Blumofe et al. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[17] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 259–270. ACM, January 2005.

[18] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230. ACM, November 2002.

[19] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 283–295. ACM, January 2005.

[20] Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, July 1989.

[21] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[22] Denis Caromel. Towards a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.

[23] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 123–134. ACM, January 2004.

[24] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6(5):46–58, 2008.

[25] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 213–224. ACM, September 2008.

[26] Brian Chin and Todd D. Millstein. Responders: Language support for interactive applications. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 255–278. Springer, July 2006.

[27] Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C#. In *Proceedings of the 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL'05), OOPSLA*, 2005.

[28] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program*, 9(3):313–323, 1999.

[29] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, pages 176–200. Springer, July 2003.

[30] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*, pages 139–154. Springer, December 2008.

[31] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM, October 1998.

[32] Eric Cooper and Gregory Morrisett. Adding threads to Standard ML. Report CMU-CS-90-186, Carnegie-Mellon University, December 1990.

[33] Microsoft Corporation. Asynchronous agents library: `http://msdn.microsoft.com/en-us/library/dd492627.aspx`.

[34] Oracle Corporation. Java Platform, Standard Edition 6 API specification: `http://download.oracle.com/javase/6/docs/api/`.

[35] Ryan Cunningham and Eddie Kohler. Making events less slippery with EEL. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*. USENIX, June 2005.

[36] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Ambient references: addressing objects in mobile networks. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 986–997. ACM, October 2006.

[37] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80–98, 2009.

[38] Hayco Alexander de Jong. *Flexible Heterogeneous Software Systems*. PhD thesis, Univ. of Amsterdam, 2007.

[39] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254, July 2006.

[40] Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with Java(X). In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 550–574. Springer, 2007.

[41] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 28–53. Springer, 2007.

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

[43] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*, pages 213–226. ACM, October 2008.

[44] Kevin Donnelly and Matthew Fluet. Transactional events. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 124–135. ACM, September 2006.

[45] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. *Operating Systems Review*, 25(5):122–136, October 1991.

[46] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE Conference on Local Computer Networks (LCN'04)*, pages 455–462. IEEE Computer Society, November 2004.

[47] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 273–298. Springer, 2007.

[48] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Proceedings of the 13th European Symposium on Programming (ESOP'04)*, pages 204–218. Springer, 2004.

[49] Rich Hickey et al. Clojure: `http://clojure.org/`.

[50] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and

reliable message-based communication in Singularity OS. In *Proceedings of the 2006 EuroSys Conference*, pages 177–190. ACM, April 2006.

[51] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 13–24, June 2002.

[52] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 195–208. ACM, January 2005.

[53] Cormac Flanagan and Martín Abadi. Object types against races. In Jos C. M. Baeten and Sjouke Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer, August 1999.

[54] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 237–247. ACM, June 1993.

[55] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002.

[56] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM, January 1996.

[57] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer, August 1996.

[58] Svend Frølund and Gul Agha. Abstracting interactions based on message sets. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 1994.

[59] Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[61] Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper Honig Spring. Pervasive computing with frugal objects. In *Workshops Proceedings of the 21st International Conference on Advanced Information Networking and Applications (AINA'07)*, pages 13–18. IEEE Computer Society, May 2007.

[62] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency oriented programming in Termite Scheme. In *Proceedings of the 2006 Workshop on Scheme and Functional Programming*, September 2006.

[63] Philipp Haller. Scala Joins library: `http://lamp.epfl.ch/~phaller/joins/`.

[64] Philipp Haller. Uniqueness types for Scala, compiler plug-in: `http://lamp.epfl.ch/~phaller/capabilities.html`.

[65] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08)*, pages 135–152. Springer, June 2008.

[66] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proceedings of the 7th Joint Modular Languages Conference (JMLC'06)*, pages 4–22. Springer, September 2006.

[67] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION'07)*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190. Springer, June 2007.

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

[69] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010.

[70] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Eng*, 17(5):424–435, May 1991.

[71] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 388–402. ACM, October 2003.

[72] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'05)*, pages 48–60. ACM, June 2005.

[73] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *Symp. Lisp and Functional Programming*, pages 18–24. ACM, August 1984.

[74] Anders Hejlsberg, Mads Togersen, Scott Wiltamuth, and Peter Golde, editors. *The C# Programming Language*. Addison-Wesley, third edition, 2008.

[75] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300, May 1993.

[76] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 235–245, 1973.

[77] Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management (ISMM'04)*, pages 73–84, October 2004.

[78] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the 6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*. ACM, October 1991.

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

[80] D. Kafura, M. Mukherji, and G. Lavender. ACT++: A class library for concurrent programming in C++ using actors. *J. of Object-Oriented Programming*, 6(6), 1993.

[81] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ'09)*, pages 11–20. ACM, August 2009.

[82] Naoki Kobayashi. Quasi-linear types. In *Conference Record of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 29–42. ACM, January 1999.

[83] LAMP/EPFL and contributors. The Scala programming language, official distribution: `http://www.scala-lang.org/`.

[84] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.

[85] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.

[86] Doug Lea. A Java fork/join framework. In *Proceedings of the 2000 ACM Java Grande Conference*, pages 36–43. ACM, June 2000.

[87] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, University of California, Berkeley, January 2006.

[88] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer, March 2010.

[89] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services: Implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 189–199. ACM, June 2007.

[90] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[91] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers - programming in E as plan coordination. In *Proceedings of the 2005 International Symposium on Trustworthy Global Computing (TGC'05)*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer, April 2005.

[92] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[93] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. ACM, October 2007.

[94] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, pages 158–185. Springer, July 1998.

[95] J. H. Nyström, Philip W. Trinder, and David J. King. Evaluating distributed functional languages for telecommunications software. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, pages 1–7. ACM, August 2003.

[96] Nathaniel Nystrom, Vijay A. Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*, pages 457–474. ACM, October 2008.

[97] Martin Odersky. Observers for linear types. In *Proceedings of the 4th European Symposium on Programming (ESOP'92)*, pages 390–407. Springer, February 1992.

[98] Martin Odersky. Functional Nets. In *Proceedings of the 9th European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[99] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57. ACM, October 2005.

[100] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th*

International Workshop on Computer Science Logic (CSL'01), pages 1–19. Springer, September 2001.

[101] C. Okasaki. Views for Standard ML. In *Proceedings of the 1998 SIGPLAN Workshop on ML*, pages 14–23, September 1998.

[102] John Ousterhout. Why threads are a bad idea (for most purposes), January 1996. Invited talk at the 1996 USENIX Annual Technical Conference (USENIX'96).

[103] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX'99)*, pages 199–212. USENIX, June 1999.

[104] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 75–86. ACM, January 2008.

[105] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[106] Hubert Plociniczak and Susan Eisenbach. JErlang: Erlang with joins. In *Proceedings of the 12th International Conference on Coordination Models and Languages (COORDINATION'10)*, volume 6116 of *Lecture Notes in Computer Science*, pages 61–75. Springer, June 2010.

[107] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 294–305. ACM, June 1991.

[108] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 317–328. ACM, 2009.

[109] Claudio V. Russo. The Joins concurrency library. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL'07)*, pages 260–274, January 2007.

[110] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'07)*, page 271. ACM, March 2007.

[111] Jan Schäfer and Arnd Poetzsch-Heffter. JCobox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, June 2010.

[112] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[113] Amazon SimpleDB. `http://aws.amazon.com/simpledb/`.

[114] Satnam Singh. Higher-order combinators for join patterns using STM. In *Proceedings of the 2006 TRANSACT Workshop, OOPSLA*, 2006.

[115] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Brief announcement: Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 338–339. ACM, August 2007.

[116] Jesper Honig Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 211–228. ACM, October 2007.

[117] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, pages 104–128. Springer, July 2008.

[118] Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. The Java module system: core design and semantic definition. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 499–514. ACM, October 2007.

[119] Martin Sulzmann, Edmund S. L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08)*, pages 315–330. Springer, June 2008.

[120] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 29–40. ACM, October 2007.

[121] D. A. Thomas, W. R. Lalonde, J. Duimovich, M. Wilson, J. McAffer, and B. Berry. Actra: A multitasking/multiprocessing Smalltalk. *ACM SIGPLAN Notices*, 24(4):87–90, April 1989.

[122] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput*, 132(2):109–176, 1997.

[123] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'89)*, pages 103–112. ACM, October 1989.

[124] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. In *Proceedings of the 2001 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 20–34. ACM, October 2001.

[125] J. Robert von Behren, Jeremy Condit, and Eric A. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 19–24. USENIX, May 2003.

[126] J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric A. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 268–281. ACM, October 2003.

[127] G.S. von Itzstein and David Kearney. Join Java: An alternative concurrency semantic for Java. Technical Report ACRC-01-001, University of South Australia, 2001.

[128] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 307–313, January 1987.

[129] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, Israel, 1990. IFIP TC 2 Working Conference.

[130] David Walker, Karl Crary, and J. Gregory Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst*, 22(4):701–771, 2000.

[131] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Types in Compilation (TIC'00)*, pages 177–206. Springer, September 2000.

[132] David Walker and Kevin Watkins. On regions and linear types. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 181–192, September 2001.

[133] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. ACM, August 1980.

[134] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 230–243. ACM, October 2001.

[135] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput*, 115(1):38–94, November 1994.

[136] Tobias Wrigstad. *Ownership-Based Alias Managemant*. PhD thesis, KTH, Sweden, 2006.

[137] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4), 2007.

[138] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 377–388. ACM, January 2010.

[139] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, pages 445–469. Springer, July 2009.

[140] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, pages 258–268. ACM, November 1986.

[141] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251. IEEE Computer Society, 2004.

[142] Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 136–147. ACM, September 2006.

CURRICULUM VITÆ

PERSONAL INFORMATION

Name: Philipp Klaus Haller
Date of birth: October 1^{st}, 1980
Place of birth: Stuttgart, Germany

EDUCATION

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland 2006 – 2010
School of Computer and Communication Sciences
Research assistant and Ph.D. student in the doctoral school

Karlsruhe Institute of Technology (KIT), Germany 2001 – 2006
Department of Computer Science
Dipl.-Inform. (with distinction), Computer Science, May 2006

Karls-Gymnasium Stuttgart, Germany 1991 – 2000
Abitur, June 2000

AWARDS AND SCHOLARSHIPS

- Prize of excellence for an exceptional teaching contribution, EPFL 2008
- Best Student Paper Award, COORDINATION 2007
- Full Scholarship, German National Merit Foundation 2001 – 2006

PUBLICATIONS

Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009

Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08)*, pages 135–152. Springer, June 2008

Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proceedings of the 7th Joint Modular Languages Conference (JMLC'06)*, pages 4–22. Springer, September 2006