Capabilities for Uniqueness and Borrowing

Philipp Haller and Martin Odersky

EPFL, Switzerland
{philipp.haller, martin.odersky}@epfl.ch

Abstract. An important application of unique object references is safe and efficient message passing in concurrent object-oriented programming. However, to prevent the ill effects of aliasing, practical systems often severely restrict the shape of messages passed by reference. Moreover, the problematic interplay between destructive reads–often used to implement unique references–and temporary aliasing through "borrowed" references is exacerbated in a concurrent setting, increasing the potential for unpredictable run-time errors.

This paper introduces a new approach to uniqueness. The idea is to use capabilities for enforcing both at-most-once consumption of unique references, and a flexible notion of uniqueness. The main novelty of our approach is a model of uniqueness and borrowing based on simple, unstructured capabilities. The advantages are: first, it provides simple foundations for uniqueness and borrowing. Second, it can be formalized using a relatively simple type system, for which we provide a complete soundness proof. Third, it avoids common problems involving borrowing and destructive reads, since unique references subsume borrowed references.

We have implemented our type system as an extension to Scala. Practical experience suggests that our system allows type checking real-world actor-based concurrent programs with only a small number of additional type annotations.

1 Introduction

Message-based concurrency provides robust programming models that scale from multicore processors to distributed systems, web applications, and cloud computing. Seamless scalability requires that local and remote message send operations should behave the same. A good candidate for such 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. 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 (i.e., copying). This would prohibit the use of message-passing altogether in performance-critical code that deals with large messages, such as image processing pipelines or network protocol stacks [19, 20]. 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 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 [20, 39, 40].

Immutable messages are used, for instance, in Erlang [3], a programming language created by Ericsson that was used at first in telecommunication systems, but is now also finding applications in Internet commerce (e.g., Amazon's SimpleDB[1]).

The second approach usually imposes constraints on the shape of messages (e.g., trees [40]). 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 satisfy the shape constraints must first be serialized into a permitted form before it can be sent within a message.

Scala [35] provides Erlang-style concurrent processes as part of its standard library in the actors package [25]. Scala's actors run on the standard Java platform [31]; they are gaining rapidly support in industry, with applications in the Kestrel message queue system[2] powering the popular Twitter micro-blogging service, and others.

In Scala actors, 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, race safety has previously neither been enforced by the language, nor by the run-time library.

This paper proposes 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.

1.1 Background

There exists a large number of proposals for unique object references. A comprehensive survey is beyond the scope of this paper; Clarke and Wrigstad [12] 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 [48]. Importantly, "external" uniqueness enables many interesting programming patterns, such as merging of data structures and abstraction of object creation (through factory methods [23]). In the following we consider two kinds of alias encapsulation policies:

- *Deep encapsulation:* [33] the only access (transitively) to the internal state of an object is through a single entry point. References to external state are allowed.

[1] See `http://aws.amazon.com/simpledb/`.

[2] See `http://github.com/robey/kestrel/`.

Proposal	Type System	Unique Objects	Encapsulation	Program Annotations
Islands	(˜ linear types)	alias-free	full	type qualifiers, purity
Balloons	(abstr. interpr.)	alias-free	full	type qualifiers
PacLang	quasi-linear types	alias-free, flds. prim.	full	type qualifiers
PRFJ	expl. ownership	alias-free	deep/full	owners, regions, effects
StreamFlex	impl. ownership	alias-free, flds. prim.	full	type qualifiers
Kilim	impl. ownership	alias-free	full	type qualifiers
External U.	expl. ownership	intern. aliases	deep	owners, borrowing
UTT	impl. ownership	intern. aliases	deep	type qualifiers, regions
BR	capabilities	intern. aliases	deep	type qual., regions, effects
MOAO	expl. ownership	intern. aliases	full	simple owners, borrowing
Sing#	capabilities	intern. aliases	full	type qualifiers, borrowing
This paper	capabilities	intern. aliases	full	type qualifiers

Fig. 1. Proposals for uniqueness with (a) full encapsulation, (b) internal aliases, or (c) both

- *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. Figure 1 compares proposals from the literature that provide either uniqueness with internal aliasing, full alias encapsulation, or both. (Section 7 discusses other related work on linear types, regions, and program logics.) We classify existing approaches according to (a) the kind of type system they use, (b) the notion of unique/linear objects they support, (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 [14], implicit ownership types, and systems based on capabilities/permissions. The third column 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 fourth column indicates the encapsulation policy. We explain the annotations in the fifth column in the context of each proposal.

Islands [28] provide fully-encapsulated objects protected by "bridge" classes. However, extending an Island requires unique objects, which must be alias-free. Almeida's Balloon Types [1] provide unique objects with full encapsulation; however, the unique object itself may not be (internally) aliased. Ennals et al. [19] have used quasi-linear types [30] for efficient network packet processing in PacLang; in their system, packets may not contain nested pointers. The PRFJ language of Boyapati et al. [8] 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 [39] (like its successor Flexotasks [4]) 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 [40] 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 [17, 16] 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 [32] 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 [21] 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 [45] that allow a precise description of the shape of recursive data structures in a type system. Boyland and Retert [9] (BR in Figure 1) 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 [13] combines a minimal notion of ownership, external uniqueness, and immutability into a system that provides race freedom for active objects [51, 10]. 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 [49] and owner-polymorphic methods. Sing# [20] uses capabilities [21] 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 [9, 32] 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.

Contribution. 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. 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 [21, 9], our approach requires only simple, unstructured capabilities. This has several advantages: first, it provides simple foundations for uniqueness and borrowing. Second, it requires neither existential ownership nor explicit region declarations in the type system. Third, it avoids the problematic interplay between borrowing and destructive reads, since unique references subsume borrowed references. This paper also contributes:

- *A simple and flexible annotation system.* We introduce a small number of source-level annotations used to guide the type checker (Section 2). 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.
- *A simple formal model with soundness proof.* We formalize our type system in the context of an imperative object calculus (Section 3) and prove it sound (Section 4). (A complete proof of soundness appears in the companion technical report [24].) 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 the annotation system described in Section 2: all types in the formalization can be expressed using those annotations.
- *A practical design.* We extend our system to support closures and nested classes (Section 5), features that, so far, 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.

  We have implemented our type system as a pluggable annotation checker for the EPFL Scala compiler (Section 6). We show that real-world actor-based concurrent programs can be type-checked with only a small increase in type annotations.

2 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–lightweight concurrent processes–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 2, 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.[3] Therefore, it is impossible to create a copy of the log files upon sending them to the master. To ensure that

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

```
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)
}
```

Fig. 2. Running tests and reporting results.

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.

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.[4] 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 $\forall y \neq x.\ y$ *accessible* $\Rightarrow separate(x, y)$.

(A variable is accessible if it may be accessed in the current program execution.) This definition of uniqueness implies that if x is a separately-unique variable, there is no other accessible variable on the stack that shares a common reachable object with x.

[4] For simplicity we leave the heap implicit in the following discussion; we formalize it precisely in Section 3.

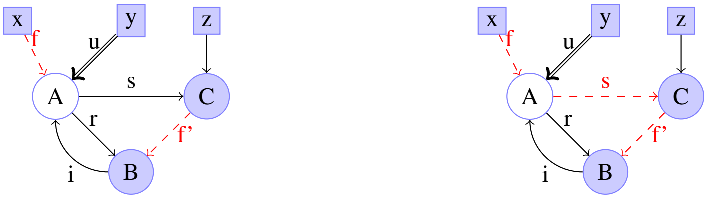

Fig. 3. Comparing (a) external uniqueness and (b) separate uniqueness. (⇒ unique reference, → legal reference, ⇢ illegal reference)

In contrast, this does not hold for external uniqueness [12], which is the notion of uniqueness most closely related to ours. Figure 3 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 the companion technical report [24]). With external uniqueness, one has to enforce additional constraints to ensure safety [13].

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.

Capabilities. A unique variable has a type guarded by some capability ρ, written $\rho \triangleright \texttt{T}$ (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 [44, 9, 11] 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 3 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 continue the discussion of our running example, thereby motivating extensions of our annotation system.

Transient and peer parameters. Our discussion of the example shown in Figure 2 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.

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 2, 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 $\rho \triangleright \texttt{LogFile}$ and $\delta \triangleright \texttt{LogList}$, 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.[5]

In our system, regions are merged using a capture expression of the form $\texttt{capture}(t_1, t_2)$. The arguments of capture 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, capture 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), capture 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 capture to merge the regions of log and logs before invoking add:

```
logs.add(capture(log, logs))
```

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

Unique fields. In the example of Figure 2, 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

[5] 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 [12].

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 $\texttt{swap}(t_1.l, t_2)$; 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.

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(capture(log, logs))
swap(this.logFiles, logs)
```

First, we obtain the current value of the unique `logFiles` field, providing `null` as its new (dummy) value.[6] 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.*

3 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 `capture` and `swap` expressions introduced in the previous section, and augment the type system with capabilities to enforce uniqueness and aliasing constraints. Our approach, however, extends to the whole of Java and other languages like Scala. We discuss important extensions in Section 5.

Syntax. Figure 4 shows the core language syntax. The syntax of programs, classes, terms, and expressions is standard, except for the `capture` 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 [29] for sequences.) Class definitions consist of declaring a single super-class followed by a body containing field

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

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

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

Fig. 4. Core language syntax

and method definitions. Field definitions carry an additional modifier α, which indicates whether the field points to a unique object ($\alpha = \texttt{unique}$), or not ($\alpha = \texttt{var}$). Method definitions are extended with two additional capability annotations that we explain below. The term language is mostly standard. However, note that terms are written in A-normal form [22]: 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 2 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 $\rho \triangleright \texttt{C}$, 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 $\rho' \triangleright \texttt{D}$ 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 2 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 ($\rho \in \Delta$).

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

Fig. 5. Syntax for heaps, environments, and dynamic capabilities

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 2. This ensures that the formal model is not more powerful than our implemented system.

3.1 Dynamic 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 5 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 [41]).

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, 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 [38].)

The top-level term of a program is reduced in the initial configuration $(r \mapsto \texttt{Object}(\epsilon)), (\texttt{this} \mapsto \rho \triangleright r), \{\rho\}$. In the following we explain the reduction rules. The congruence rule for `let` is omitted, since it is standard; $\texttt{let}\ x = y\ \texttt{in}\ t$ is reduced in the obvious way.

$$\frac{\begin{array}{c} V(y) = \delta \triangleright r \qquad \delta \in R \\ H(r) = C(\overline{r}) \end{array}}{\begin{array}{c} H, V, R, \texttt{let}\ x = y.l_i\ \texttt{in}\ t \\ \longrightarrow\ H, (V, x \mapsto \delta \triangleright r_i), R, t \end{array}} \quad \text{(R-SELECT)}$$

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

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 3.2 where we formalize the separation invariant of Section 2. 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.

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

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.

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

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 $\mathit{mbody}(m, C) = (\overline{x}, e)$.

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

Capture 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.)

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

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.

3.2 Static semantics

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.)

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

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 $fields(D)$ [29] to obtain all fields in D and super-classes of D.

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

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 $\varDelta$ must still be available. The result type of a method must be guarded by a capability in $\varDelta$. 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.

$$\frac{\begin{array}{c} mtype(m, D) \text{ not defined} \vee \\ (mtype(m, D) = \exists \delta.\ (\rho \triangleright D, \overline{T}) \rightarrow (R, \varDelta) \wedge \\ mtype(m, C) = \exists \delta.\ (\rho \triangleright C, \overline{T}) \rightarrow (R, \varDelta)) \end{array}}{override(m, C, D)} \qquad (\text{WF-O{\scriptsize VERRIDE}})$$

A method defined in class C satisfies the rule for well-formed overriding if the superclass 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 [29], 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.

$$\frac{C <: D}{\rho \triangleright C <: \rho \triangleright D} \qquad \text{(<:-Cap)}$$

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.

$$\frac{\Gamma(y) = \rho \triangleright C \qquad \rho \in \Delta}{\Gamma \ ; \ \Delta \vdash y : \rho \triangleright C \ ; \ \Delta} \ \text{(T-Var)} \qquad \frac{\begin{array}{c}\Gamma \ ; \ \Delta \vdash y : \rho \triangleright C \\ fields(C) = \overline{\alpha\, l : D} \qquad \alpha_i \neq \texttt{unique}\end{array}}{\Gamma \ ; \ \Delta \vdash y.l_i : \rho \triangleright D_i \ ; \ \Delta} \ \text{(T-Select)}$$

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.

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

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 3.1).

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

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 5 for a discussion in the context of nested classes.)

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

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.

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

The type rule for capture requires the capabilities of the arguments to be present (this follows from (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, capture casts its first argument from its current region to the region of the second argument; in Section 4 we prove that the cast can never fail at run-time.

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

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 (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.

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

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.

Definition 1 (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 `capture` consumes y's capability, thereby making it inaccessible. Therefore, the invariant continues to hold for accessible variables, that is, variables whose capabilities are present. (The predicate $separate$ is formally defined in the companion technical report [24].)

Definition 2 (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.

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

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.

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

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.

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

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).

4 Soundness

In this section we present the main soundness result for the type system introduced in Section 3. We prove type soundness using the standard syntactic approach of preservation plus progress [47]. A complete proof of soundness appears in the companion technical report [24].

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 3.2. 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 3 (Separate Uniqueness) *A variable* $(y \mapsto \beta \triangleright r) \in V$ *such that* $\beta \in R$ *is separately-unique in configuration* $H, V, R, \texttt{let}\ x = t\ \texttt{in}\ t'$ 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*

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

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

5 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.

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.[7] 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`

[7] 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.

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.

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 3, 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.

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 6). This means that most objects only interact with objects from its enclosing aggregate, which is consistent with the results for thread-locality in Loci [50]. To abbreviate the canonical annotation, we allow classes to be annotated as `@transient`.

6 Implementation

We have implemented our type system as a plug-in for the Scala compiler developed at EPFL.[8] 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 3. For subsequent code generation, all capabilities, `capture` and `swap` expressions are erased.

Practical Experience. As 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 5) required changing 60 source lines.

In Section 2, 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, “instances of the `File` class are immutable.”[9] 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`.

[8] See `http://lamp.epfl.ch/~phaller/capabilities.html`.

[9] See `http://java.sun.com/javase/6/docs/api/`.

7 Other Related Work

In functional languages, linear types [43] 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 [34] 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 [46]. Bierhoff and Aldrich [6] build on an expressive linear program logic for modular type-state checking in an object-oriented setting. It is not clear how their system could be applied to message-based concurrency, since unique references do not prevent read-only references to the same object. Beckman et al. [5] use a similar system for verifying the correct use of software transactions. JAVA(X) [15] tracks linear and affine resources using type refinement and capabilities, which are structured, unlike ours. The authors did not consider applications to concurrency. Shoal [2] 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 [42, 44, 27, 52], 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 [36] is a program logic designed to reason about separation of portions of the heap; the logic is not decidable and does not deal with stack variables, unlike our approach. Bornat et al. [7] study permission accounting in separation logic; unlike our system, their approach is not automated. Parkinson and Bierman [37] extend the logic to an object-oriented setting; however, applications [18] still require heavy-weight theorem proving and involve extensive program annotation. To avoid aliasing, swapping [26] 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.

8 Conclusion

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.

Acknowledgments: Thanks to Rémi Bonnet for discussions on earlier versions of the type system. Thanks to the anonymous reviewers for detailed and helpful comments.

References

1. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, pages 32–59, 1997.
2. Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *PLDI*, pages 98–109. ACM, 2009.
3. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
4. 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 *LCTES*, pages 1–11. ACM, 2008.
5. Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In Gail E. Harris, editor, *OOPSLA*, pages 227–244. ACM, 2008.
6. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320. ACM, 2007.
7. Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
8. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
9. John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL*, pages 283–295. ACM, 2005.
10. Denis Caromel. Towards a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.
11. Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ICFP*, pages 213–224. ACM, 2008.
12. Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200. Springer, 2003.
13. Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS*, pages 139–154. Springer, 2008.
14. David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
15. Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with java(X). In *ECOOP*, pages 550–574. Springer, 2007.
16. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP*, pages 28–53. Springer, 2007.
17. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
18. Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226. ACM, 2008.
19. Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *ESOP*, pages 204–218. Springer, 2004.
20. 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 *EuroSys*, pages 177–190. ACM, 2006.
21. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
22. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.
23. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

24. Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. Technical Report LAMP-REPORT-2009-004, `http://infoscience.epfl.ch/record/142817`, EPFL, Lausanne, Switzerland, December 2009.
25. Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.
26. 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.
27. Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, pages 73–84, 2004.
28. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
29. 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.
30. Naoki Kobayashi. Quasi-linear types. In *POPL*, pages 29–42, 1999.
31. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
32. Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *OOPSLA*, 2007.
33. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, pages 158–185. Springer, 1998.
34. Martin Odersky. Observers for linear types. In *ESOP*, pages 390–407. Springer, 1992.
35. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2008.
36. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19. Springer, 2001.
37. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM, 2008.
38. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
39. Jesper Honig Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228, 2007.
40. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP*, pages 104–128. Springer, 2008.
41. Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. The java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.
42. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput*, 132(2):109–176, 1997.
43. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, Israel, 1990. IFIP TC 2 Working Conference.
44. David Walker, Karl Crary, and J. Gregory Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst*, 22(4):701–771, 2000.
45. David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *TIC*, pages 177–206. Springer, 2000.
46. David Walker and Kevin Watkins. On regions and linear types. In *ICFP*, 2001.
47. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput*, 115(1):38–94, November 1994.
48. Tobias Wrigstad. *Ownership-Based Alias Managemant*. PhD thesis, KTH, Sweden, 2006.
49. Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4), 2007.
50. Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for java. In *ECOOP*, pages 445–469. Springer, 2009.
51. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, pages 258–268, 1986.
52. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *RTSS*, pages 241–251. IEEE Computer Society, 2004.