Capabilities for External Uniqueness

Philipp Haller and Martin Odersky

EPFL, Switzerland

LAMP-REPORT-2009-001

Abstract. Unique object references have many important applications in object-oriented programming. For instance, with sufficient encapsulation properties they enable safe and efficient transfer of message objects between concurrent processes. However, it is a long-standing challenge to integrate unique references into practical object-oriented programming languages.
This paper introduces a new approach to external uniqueness. The idea is to use capabilities for enforcing both aliasing constraints that guarantee external uniqueness, and linear consumption of unique references. We formalize our approach as a type system, and prove a type preservation theorem. Type safety rests on an alias invariant that builds on a novel formalization of external uniqueness.
We show how a capability-based type system can be used to integrate external uniqueness into widely available object-oriented programming languages. Practical experience suggests that our system allows adding uniqueness information to common collection classes in a simple and concise way.

1 Introduction

Message-based concurrency provides robust programming models that scale from multi-core 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 marshalling (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 [17, 18]. 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 [18, 35, 36].

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 [34]).

The second approach usually imposes constraints on the shape of messages (e.g., trees [36]). 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 [33] provides Erlang-style concurrent processes as part of its standard library in the actors package [21]. Scala's actors run on the standard Java platform [27]; they are gaining rapidly support in industry, with applications in the lift web application framework [20], the Kestrel message queue system [24] 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. Our approach combines capabilities [19, 7] and external uniqueness [10, 42]. Capabilities express access permissions for objects. In the simplest case, a capability is binary, that is, either the capability is available, in which case the associated object may be accessed, or the object is not accessible. External uniqueness is used to control aliasing of transferred objects.

The contributions of this paper are as follows:

– As our main technical contribution we introduce *Object Capability Types* (OCT), a type system that uses capabilities for enforcing both external uniqueness, and linear consumption of unique references, making the system uniform and simple. We present a formalization of OCT and provide a complete proof of a type preservation theorem.

– We formally state an alias invariant of OCT that implies safe ownership transfer. The invariant builds on a novel formalization of external uniqueness that is not based on ownership.
– We show how OCT can be applied to message-based concurrency by providing *tracked references* in the style of Singularity [18]. Our approach improves on the state of the art in message-based concurrency by (1) integrating external uniqueness and adoption [19], which enables aliased objects to point to unique objects in a statically safe way, while (2) avoiding the problems of destructive reads [7].
– We have implemented OCT as an extension of the EPFL Scala compiler. In Section 8.1 we show that virtually all methods of common collection classes in Scala's standard library can be annotated with uniqueness information in a simple way *without any changes to their implementation.*

The rest of the paper is organized as follows. In Section 2 we motivate why unique object references are useful in the context of message-based concurrency. Section 3 introduces a system of annotations that allows expressing and enforcing uniqueness properties in object-oriented programs. In Section 4 we formalize our approach in the form of type rules for an imperative object calculus. In Section 5 we introduce an operational semantics for our core language, and use it to prove a type preservation theorem. Section 6 formalizes aliasing guarantees of our type system. In Section 7 we extend our system to advanced constructs such as nested classes. We report on an implementation for Scala and practical experience in Section 8. Section 9 reviews related work, and Section 10 concludes.

2 Example

Consider the following example of an actor that receives a reference to a linked list.

```
actor {
  receive {
    case rlist: LinkedList => ...
  }
}
```

At first, `rlist` is the only active reference to the received list; therefore, it would be safe to pass it on to another actor, as long as the reference is not accessed after it has been sent. Assume, however, that another list should be appended to the received list before it is passed on.

```
  case rlist: LinkedList =>
    val other: LinkedList = ...
    rlist.append(other)
    next.send(rlist)
```

The `other` list may have been built locally or may have been received from elsewhere. Using the `append` method of the `LinkedList` class, we append `other`

```
class Node {
  var el: Object
  var prev, next: Node
}
class LinkedList {
  var head: Node
  def append(other: LinkedList) {
    if (head == null)
      head = other.head
    else if (other.head != null) {
      var h = head
      while (h.next != null) h = h.next
      h.next = other.head
      h.next.prev = h
    }
  }
}
```

Fig. 1. Example: Doubly-linked list with `append` method.

to `rlist`. Then, the extended `rlist` is sent to the `next` actor. Ideally, it should be safe to use the `append` method as it is shown in Figure 1, which involves no copying. For this, we have to check that the method does not change the (external) aliasing of its receiver. In other words, a unique reference to a list, such as `rlist`, should remain unique after invoking `append` on it. The challenge in verifying the safety of the `append` method is that it creates aliases of existing objects.

Such a scenario is typical in message-based concurrency, where it is important to avoid copying when constructing a new message from a received message. An actor should be able to mutate a received message, and then pass along a reference to the mutated message as in the above example. However, this requires checking that the mutation did not introduce any aliasing that could violate race safety.

In the next section we introduce a type system that can verify that the `append` method maintains the uniqueness of its receiver. This is done by showing that the aliases that the method creates are strictly internal to the data structure.

3 Object Capability Types

In this section we introduce a set of annotations and expressions that ensure external uniqueness in the presence of mutation and aliasing. The type safety of our system is based on an alias invariant that we discuss subsequently. Section 4 presents a formalization of the type rules that guarantee this alias invariant.

Figure 2 shows the example that we introduced in Section 2 with additional annotations that allow our type system to verify the safety of the `append` method.

```
class Node {
  var el: Object
  var prev, next: Node
}
class LinkedList {
  var head: Node
  @transient
  def append(other: LinkedList @unique) {
    expose (this) { xl =>
      val ol = localize(other, xl)
      if (xl.head == null)
        xl.head = ol.head
      else if (ol.head != null) {
        var h = xl.head
        while (h.next != null) h = h.next
        h.next = ol.head
        h.next.prev = h
      }
    }
  }
}
```

Fig. 2. Example: Safe append using `expose` and `localize`.

In Section 3.4, we show how some of the additional annotations can be inferred by our system. We now explain each of the annotations and operations in turn.

3.1 Annotations

Our system uses three annotations, `@unique`, `@transient`, and `@exposed`. Only local variables, method parameters (including the receiver), and method results can carry these annotations. This means that class declarations remain unchanged. In the following we explain the `@unique` and `@transient` annotations. We defer a discussion of `@exposed` until Section 3.5.

The `@transient` annotation at the definition of the `append` method in Figure 2 actually applies to the *receiver*, i.e., `this`. It requires `this` to be unique; moreover, calling the method will not consume this reference, that is, after the call the reference is still available (and unique). This corresponds to our requirement that ownership transfer should be safe after calling `append`. The `@unique` annotation at the argument type requires the argument to be unique. Moreover, it grants the method the right to consume the argument, which means that it is invalid to access the argument after the call returns.

3.2 Expose and Localize

To prevent aliases that could destroy its uniqueness, a unique object must be exposed before its fields can be accessed. In the example, the expression

```
expose (this) { xl => ... }
```

exposes the receiver and provides access to its fields via `xl`. Inside the new scope, `xl` has a type of the form $\rho \triangleright$ `LinkedList` (*guarded type*) where ρ is a capability (or guard) that is freshly generated.

Guarded types serve as a protection from unwanted aliases. A guarded type $\sigma \triangleright$ `C` is only compatible with guarded type $\sigma' \triangleright$ `C` if σ and σ' are known to be equal. In the example, this means that it is impossible to store `xl` in the field of some object that is reachable from outside the scope of `expose`, since the capability ρ in its type was freshly generated by the `expose`.

Note that it is safe for an object reachable from `xl` to refer to a unique object. That way, no external alias to `xl`, or an object reachable from `xl` is created. Thus, the external uniqueness of `xl` is preserved. For this reason, our system provides a `localize` operation that makes a unique object available to be referred to from an object with the same guard as its second argument. Note that `localize` consumes the unique object, otherwise objects with other guards could refer to it as well. In the example, the expression `localize(other, xl)` has type $\rho \triangleright$ `LinkedList`, since `xl` is guarded by ρ; this enables fields of `xl` to refer to the localized object.

Field selections propagate guards: `xl.head` has the same guard as `xl`; since `ol` is localized to `xl`, `ol.head` has the same guard. Thus, the assignments that follow are permitted.

Intuitively, the external uniqueness of `xl` is preserved, because we create only references between objects reachable from either `this` or `other`; since `other` was unique, no external aliases are created. In the following section, we introduce an alias invariant that precisely expresses what property our type system provides.

3.3 Alias Invariant

One of the goals of our system is to allow transferring a unique object to the context of another unique object without copying. Typically, the objects that we want to transfer contain sub-objects that represent the containing object. For instance, the representation of the doubly-linked list shown in Figure 1 consists of a group of linked `Node` objects. This example shows that the (internal) sub-objects are often aliased in possibly complex ways. In our system, the fact that it is safe to transfer an object depends on the objects reachable from it. An object o_1 is reachable from another object o_2 if and only if a field of o_2 refers to o_1 or an object reachable from o_2 refers to o_1. Our system guarantees safe transfer for object graphs characterized as *clusters*.

Definition 1 (Cluster) *Let o be an object and R be the set of objects reachable from o. Define $Cl(o) = R \cup \{o\}$. Then, $Cl(o)$ is a cluster if and only if every object in R is only reachable from objects in $Cl(o)$.*

For each object o that we want to transfer, $Cl(o)$ must be a cluster. This means that the objects reachable from o must be self-contained in the sense that they are not reachable from outside the cluster $Cl(o)$. There are *no restrictions wrt. references between objects in* $Cl(o)$ in our system.

Note that clusters are not declared explicitly in programs. Instead, our type system checks the cluster property according to the way an object is constructed and operated on subsequently.

Definition 2 (External Uniqueness) *A reference* r *to an object* o *is* externally-unique *if* $Cl(o)$ *is a cluster and* r *is the only accessible external reference into* $Cl(o)$*.*

A reference is external to a cluster $Cl(o)$ if it is not stored in a field of an object in $Cl(o)$. A reference is accessible if it may be accessed in the current program execution. We often call a reference just *unique* if it is externally-unique.

Instantiating a new object creates a unique reference, provided that all constructor arguments are unique (or immutable, such as primitive values), and the constructor satisfies certain sanity constraints, such as not leaking `this` or any of its arguments to static class members. We use a backwards flow analysis to enforce uniqueness of constructed objects like in other systems [35]. In existing (Java) programs, constructors return unique objects in most cases [9, 28].

For example, assume we want to create an instance of the doubly-linked list shown in Figure 1. The default constructors of the `LinkedList` and `Node` classes (not shown) initialize all fields to `null` and return unique references to the newly created instances.

Like method parameters, also local variables and method returns can carry the `@unique` annotation to indicate that the corresponding reference should be unique.

Definition 3 (Tracked Variable) Tracked variables *are local variables and method parameters that carry the* `@unique` *annotation.*

Note that a tracked variable does not always hold a unique reference. For instance, when a tracked variable is exposed, there may be multiple external references into the exposed cluster. However, transferring a unique object to another context is only safe when the tracked variable that holds a reference to it is not exposed. This is expressed by the following alias invariant.

Definition 4 (Alias Invariant) *A tracked variable is externally-unique when it is consumed.*

Our type system guarantees statically that a cluster of objects is externally-unique when it is transferred, which implies type safety.

3.4 Generalizing Uniqueness and Borrowing

The `SingleLinkedList` class in Scala's standard collections library contains the following recursive `append` method:

```
def append(that: SingleLinkedList) {
  if (next eq null)
    next = that
  else
    next.append(that)
}
```

Again, we would like to express that the method maintains the uniqueness of the receiver, provided the parameter `that` is unique. Therefore, we annotate the method as `@transient`, which actually means that the *receiver* is required to be unique; moreover, calling the method will not destroy its uniqueness. We mark the `that` parameter as `@unique`, because it is required to be unique and the method consumes it. Since `append` accesses the receiver's `next` field, we must expose the receiver to gain temporary access to its fields:

```
@transient
def append(that: SingleLinkedList @unique) {
  expose (this) { xt =>
    if (xt.next eq null)
      xt.next = that // consume ‘that‘
    else
      // ‘xt.next‘ is exposed
      xt.next.append(that)
  }
}
```

As before, `xt` has type $\rho \triangleright$`SingleLinkedList` for some guard ρ that was generated fresh for type checking the scope that `expose` opens. Note that in the first branch of the if-expression, `that` is localized implicitly to the cluster of `xt`; in many cases, such as the above, `localize` expressions can be inferred.

Since field selections propagate guards, the receiver `xt.next` of the method invocation in the else-branch of the if-expression has type $\rho \triangleright$`SingleLinkedList`. However, `append` expects the receiver to be transient. Informally, it is clear that the recursive call is harmless; it does not destroy the uniqueness of the original receiver. Indeed, our type system accepts the method invocation. In the following we explain why it is safe to do so.

A method that takes a transient parameter, say, `x`, is type-checked under the assumption that `x` is a unique reference into some cluster. The type system makes sure that at the end of the method body, the target of `x` is still a cluster, and *no additional external reference into the same cluster was created.* Exposed objects are generally not unique, since at any program point, there may exist arbitrary aliases from stack locations (local variables, method parameters, etc.). However, an exposed reference always points into a cluster. This property is maintained when passing an exposed reference as a transient parameter in a method call. Furthermore, the method is prevented from creating additional external references into the cluster. Therefore, the corresponding `expose` expression (still) guarantees the external uniqueness of the exposed object.

In summary, an exposed object is compatible with a transient formal method parameter as long as the underlying class types conform. This means that the transient qualifier is strictly more general than both the exposed and unique qualifiers, in the sense that both exposed and unique objects can be passed as transient parameters. This relationship between exposed and transient objects means that *it is safe to implicitly expose all transient parameters (including the receiver) for the entire method body.*

For example, in the body of the append method in Figure 2, it is not necessary to write the `expose` expression explicitly. Since the receiver is transient, the type checker infers an `expose` expression for `this` that spans the entire method body.

3.5 Exposed Parameters

In addition to the `append` method we discussed above, the `SingleLinkedList` class also contains the following `insert` method:

```
def insert(that: SingleLinkedList) {
  if (that ne null) {
    that.append(next)
    next = that
  }
}
```

It uses `append` to insert another list `that` at the current position (the receiver is the current node in the linked list). For this, it passes the receiver's `next` reference to `append`.

Assume that we want to make the receiver of `insert` transient or unique. Then, we have to expose it to make its `next` field accessible under some guarded type $\rho \triangleright$ `SingleLinkedList`. In the previous section we have seen that it is safe to pass a reference of guarded type to a method expecting a transient parameter. However, the parameter of `append` is marked as `@unique`, which is incompatible with references of guarded type, since the parameter is potentially consumed. This restriction is artificial, though. When considering only the (transitive) aliasing effect of `insert`, it should be valid to annotate the receiver with `@transient` and the parameter with `@unique`.

To allow reusing the `append` method, we generalize it by annotating its receiver and parameter as `@exposed`:

```
@exposed
def append(that: SingleLinkedList @exposed) {
  if (next eq null)
    next = that
  else
    next.append(that)
}
```

Note that we can now use the unmodified body of the original, non-annotated `append` method. The `@exposed` annotations indicate that both the receiver and the `that` parameter have guarded type $\rho \triangleright$ `SingleLinkedList` for some *common guard* ρ.

In general, the guard in the type of an exposed parameter (including the receiver) is shared by all other exposed parameters. This means that exposed parameters are required to be part of the same cluster. As a result, references between those objects are unrestricted, enabling flexible imperative implementations.

We can use the modified version of `append` in the implementation of `insert` as follows:

```
@transient
def insert(that: SingleLinkedList @unique) {
  val locThat = localize(that, this)
  if (locThat ne null) {
    locThat.append(next)
    next = locThat
  }
}
```

Note that the receiver is exposed implicitly; as discussed in the previous section, transient references can safely be treated as exposed. The parameter is localized to `this`. Therefore, the invocation of `append` and the following assignment type checks.

In summary, exposed parameters allow a flexible reuse of methods. Transient references are compatible with exposed parameters, requiring no changes at the call site. Unique references must be exposed or localized by the caller. Methods with exposed parameters enable a great deal of flexibility in their implementation. In Section 8.1 we show that virtually all methods of common collection classes in Scala's standard library can be annotated using `@exposed` without any changes to their implementation.

4 The Type System

This section presents a formal description 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 [4] inspired by FJ [23]. We add the `expose` and `localize` expressions, 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 7.

The core language syntax is shown in Figure 3. The syntax of programs, classes and expressions is standard; method definitions are extended with additional typing constraints on the receiver and the arguments that we discuss below. In the examples of the previous section, types could be annotated with `@unique`, `@transient`, and `@exposed`. In our formalization, these annotated types are represented as class types with associated capabilities; however, *these capabilities are implicit and never appear in actual programs.* In the following we

$P ::= \overline{cdef}\ t$
$cdef ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{l : C};\ \overline{meth}\}$
$meth ::= \texttt{def}\ m[\rho \star \Delta](\overline{x : T}) : (\Delta', T) = t$
$t ::=$
$\quad x \mid r \mid t.l \mid t_1.l = t_2 \mid t_0.m(\bar{t})$
$\quad \texttt{new}\ C(\bar{t}) \mid \texttt{localize}(t_1, t_2)$
$\quad \texttt{expose}\ x = t_1\ \texttt{in}\ t_2$
$T ::= \rho \blacktriangleright C \mid \rho \triangleright C$
$C, D \in Classes$
$x \in Vars$
$l \in Fields$
$r \in RefLocs$
$\rho \in Guards$

Fig. 3. Core language syntax

discuss the representation of types annotated with `@unique`. The `@transient` and `@exposed` annotations are only used in method definitions. We discuss them in Section 4.1.

The type `@unique` C translates to a *tracked type* of the form $\rho \blacktriangleright C$ for some guard ρ. Guards are used together with a system of capabilities to make sure that objects of tracked type are consumed at most once. The type system ensures that whenever an object of type $\rho \blacktriangleright C$ is accessed, the capabilities available at the current program point include the singleton capability $\{\rho\}$. This is done by tracking the set of available capabilities during type checking. When an object of type $\rho \blacktriangleright C$ is consumed, the capability $\{\rho\}$ is removed from the available capabilities. Since capabilities may not be duplicated, a unique reference becomes unusable once its capability is consumed.

Capabilities are formed by joining together singleton capabilities using the $\otimes$ operator. The empty capability is written $\varnothing$. The capability that includes only the singleton capabilities $\{\rho\}$ and $\{\sigma\}$ is expressed as $\rho \otimes \sigma$. Note that we omit the braces from singleton capabilities when joining them with other capabilities. We use the terms capability and guard interchangeably when talking about singleton capabilities. We use $\rho \in \Delta$ as an alternative notation for expressing $\Delta = \rho \otimes \Delta'$ for some Δ'. The capability $\Delta - \Delta'$ contains all guards ρ with $\rho \in \Delta \wedge \rho \notin \Delta'$.

Note that capabilities are a purely static concept; after type checking, capabilities and the `expose` and `localize` expressions, which only operate on capabilities, are erased from the program. Consequently, capability checking incurs no runtime overhead.

A tracked object that has been exposed is given a guarded type of the form $\rho \triangleright C$. In contrast to tracked types, objects of guarded type may have an unknown number of aliases. Roughly speaking, objects of guarded type can be mutated as long as their fields point to objects that have types guarded by the same capability. Standard class types are expressed as guarded types $\epsilon \triangleright C$ where the guard ϵ corresponds to the empty capability $\varnothing$.

4.1 Method Definitions

Method definitions are extended with two capabilities Δ and Δ' that specify required and provided capabilities, respectively. Guards in Δ may occur in the argument types $\overline{T}$. The provided capabilities Δ' contain the required capabilities that are (still) available after an invocation of the method. If the method returns a unique object, the result type is tracked, and Δ' contains an additional capability corresponding to the guard of the result type. Optionally, the type of the receiver can be qualified using $\rho\star$ where $\rho \in \Delta$ and $\star \in \{\triangleright, \blacktriangleright\}$. It allows expressing whether the receiver is required to be tracked or guarded, and whether it is potentially consumed ($\rho \notin \Delta'$) or not ($\rho \in \Delta'$).

The required and provided capabilities Δ and Δ', respectively, correspond to source-level annotations on parameter types (including the receiver) and result types. For example, the method definition

```
@transient
def append(other: LinkedList @unique)
```

corresponds to the following method type:

$$[\rho \blacktriangleright](\rho \otimes \sigma, \sigma \blacktriangleright \texttt{LinkedList}) \to (\{\rho\}, \texttt{Unit}) \tag{1}$$

Both the `@transient` and `@unique` annotations translate to tracked types, indicated by the $\rho \blacktriangleright$ and $\sigma \blacktriangleright$ prefixes. Moreover, the required capabilities include both ρ and σ, meaning both the receiver and the parameter must be accessible. The provided capabilities contain only ρ, since σ may be consumed. Parameter types annotated with `@exposed` correspond to guarded types. For example, the method definition

```
@exposed
def append(other: LinkedList @exposed)
```

corresponds to the following method type:

$$[\rho \triangleright](\{\rho\}, \rho \triangleright \texttt{LinkedList}) \to (\{\rho\}, \texttt{Unit}) \tag{2}$$

Note that the receiver and parameter types are guarded by the same capability ρ, as motivated in Section 3.5.

4.2 Well-Formedness

For type checking programs we assume a fixed, well-formed class table that defines the sub-typing relation <:. Furthermore, the class table provides a standard function $fields(C) = \overline{l : D}$ where $\overline{l : D}$ are all fields in C and super-classes of C. We also use the standard auxiliary functions $mtype(C, m) = [\rho\ \star](\Delta, \overline{T}) \to (\Delta', T)$ and $mbody(C, m) = (\overline{x}, t)$ where $\texttt{def}\ m[\rho \star \Delta](\overline{x : T}) : (\Delta', T) = t$ is defined in the most direct superclass of C that defines m.

A method m is well-formed in a class C if its body is well-typed in an environment Γ that maps m's formal parameters to their declared types, and `this`

$$\frac{\begin{array}{c}\Gamma = \overline{x : T}, \texttt{this} : \rho \star C \qquad \mathit{override}(C, m) \\ \Gamma \;;\; \Delta \vdash t : T_0 \;;\; \Delta' \qquad T_0 <: T \\ \Delta' \subseteq \Delta \vee (\Delta' = \hat{\Delta} \otimes \rho \wedge \hat{\Delta} \subseteq \Delta \wedge T_0 = \rho \blacktriangleright C_0)\end{array}}{C \vdash \texttt{def}\ m[\rho \star \Delta](\overline{x : T}) : (\Delta', T) = t} \quad \text{(WF-Method)}$$

$$\frac{\begin{array}{c}(\forall D.\ C <: D \Rightarrow \\ \mathit{mtype}(D, m) \text{ is undefined} \vee \\ \mathit{mtype}(C, m) = \sigma(\mathit{mtype}(D, m)))\end{array}}{\mathit{override}(C, m)} \quad \text{(WF-Override)}$$

$$\frac{C \vdash \overline{\mathit{meth}}}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{l : C};\ \overline{\mathit{meth}}\}} \quad \text{(WF-Class)}$$

Fig. 4. Well-Formedness

to $\rho \star C$ where $\rho\star$ is the guard that the method requires for the receiver (WF-Method). Moreover, the capability Δ', which is available after type-checking the method body, must contain a subset of the guards in the initial capability Δ, except if the return type T_0 is tracked, in which case Δ' must also contain the capability of T_0. A method m respects the overriding rule if it does not override or if the signatures of all overridden methods can be unified using appropriate substitutions (WF-Override). A class C is well-formed if all its method definitions are well-formed (WF-Class).

4.3 Type Rules

Figure 5 shows the typing rules. The typing judgement has the form $\Gamma \;;\; \Delta \vdash t : C \;;\; \Delta'$. Γ maps (free) variables to types. Γ is an immutable map and the facts that it 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.

The rule for typing variables (T-Var) is standard; the capability set is unchanged.

Selecting a field from a term t of guarded type yields a type guarded by the same capability (T-Sel). While deriving the type for t the set of available capabilities may change resulting in a new set Δ'; this set describes the capabilities available after typing the field selection.

Assigning to a field of some guarded type requires the right-hand side to be guarded by the same capability ρ (T-Assign). After deriving types for t_1 and t_2, ρ must be available. Furthermore, ρ is not consumed by the assignment.

The rule for instance creation (T-New) requires all constructor arguments to be tracked, and their corresponding capabilities to be available. In this case, there is no reference that could point into the object graph rooted at the new instance; thus, we can assign a tracked type with a fresh capability to the new instance. The capabilities ρ_i of the arguments are consumed.

Type Assignment $\boxed{\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'}$

$$\frac{x : T \in \Gamma}{\Gamma\ ;\ \Delta \vdash x : T\ ;\ \Delta} \qquad (\text{T-Var})$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t : \rho \triangleright C\ ;\ \Delta' \\ \mathit{fields}(C) = \overline{l : D}\end{array}}{\Gamma\ ;\ \Delta \vdash t.l_i : \rho \triangleright D_i\ ;\ \Delta'} \qquad (\text{T-Sel})$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t_1 : \rho \triangleright C\ ;\ \Delta' \\ l : D \in \mathit{fields}(C) \qquad E <: D \\ \Gamma\ ;\ \Delta' \vdash t_2 : \rho \triangleright E\ ;\ \Delta'' \otimes \rho\end{array}}{\Gamma\ ;\ \Delta \vdash t_1.l = t_2 : \rho \triangleright C\ ;\ \Delta'' \otimes \rho} \qquad (\text{T-Assign})$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}\ \Gamma\ ;\ \Delta_i \vdash t_i : \rho_i \blacktriangleright C_i\ ;\ \Delta_{i+1} \otimes \rho_i \\ \mathit{fields}(C) = \overline{l : D} \qquad \overline{C} <: \overline{D} \qquad \rho\ \mathit{fresh}\end{array}}{\Gamma\ ;\ \Delta_1 \vdash \texttt{new}\ C(\overline{t}) : \rho \blacktriangleright C\ ;\ \Delta_{n+1} \otimes \rho} \qquad (\text{T-New})$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t_1 : \rho_1 \blacktriangleright C\ ;\ \Delta' \otimes \rho_1 \\ \Gamma\ ;\ \Delta' \vdash t_2 : \rho_2 \triangleright D\ ;\ \Delta''\end{array}}{\Gamma\ ;\ \Delta \vdash \texttt{localize}(t_1, t_2) : \rho_2 \triangleright C\ ;\ \Delta''} \qquad (\text{T-Loc})$$

$$\frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash t_1 : \rho \blacktriangleright C\ ;\ \Delta' \otimes \rho \\ \rho'\mathit{fresh} \qquad \rho' \notin T \\ \Gamma, x : \rho' \triangleright C\ ;\ \Delta' \otimes \rho' \vdash t_2 : T\ ;\ \Delta'' \otimes \rho'\end{array}}{\Gamma\ ;\ \Delta \vdash \texttt{expose}\ x = t_1\ \texttt{in}\ t_2 : T\ ;\ \Delta'' \otimes \rho} \qquad (\text{T-Exp})$$

$$\frac{\begin{array}{c}\forall i \in \{0..n\}\ \Gamma\ ;\ \Delta_i \vdash t_i : U_i\ ;\ \Delta'_{i+1} \\ \Delta'_{i+1} := \begin{cases} \Delta_{i+1} \otimes \rho_i \text{ if } U_i = \rho_i \blacktriangleright C_i \\ \Delta_{i+1} \text{ otherwise}\end{cases} \\ \mathit{mtype}(m, C_0) = [\rho\ \star](\Delta', \overline{F}) \to (\Delta'', C) \\ \sigma = \mathit{unify}(\overline{U}, \overline{F}) \qquad \overline{U} <: \sigma\overline{F} \qquad \sigma(\rho) = \rho_0 \\ \Delta_t = \{\rho_i | U_i = \rho_i \blacktriangleright C_i\} \qquad U_0 = \rho_0 \star C_0 \\ \Delta_{n+1} \otimes \Delta_t = \sigma\Delta' \otimes \Delta_r \qquad \Delta''' = \sigma\Delta'' \otimes \Delta_r\end{array}}{\Gamma\ ;\ \Delta_0 \vdash t_0.m(\overline{t}) : \sigma C\ ;\ \Delta''' - (\Delta_t - \Delta_0)} \qquad (\text{T-Invk})$$

Fig. 5. Type Assignment

The rule for localization (T-Loc) allows a term t_1 of tracked type to be treated as a term guarded by the same capability as a term t_2. For example, this allows t_1 to refer to t_2 (and vice versa). Since from that point on, aliases of t_1 are no longer tracked, its capability ρ_1 is consumed.

The `expose` operation allows a term t_1 of tracked type to be bound to a variable of guarded type in a given term t_2 (T-Exp). Note that the capability ρ to access t_1 is temporarily revoked during the evaluation of t_2. This is done to prevent consuming t_1 in the scope of `expose`. The guard ρ' in the type of x may not occur in the type of t_2.

The rule for method invocation (T-Invk) is the most complex rule because it deals with *guard polymorphism*. Since guards are never written explicitly in a program, methods that operate on tracked or guarded types have to be polymorphic in the respective guards. For example, consider a method m defined in class C of the following type:

$$[\sigma_1 \triangleright](\sigma_1 \otimes \sigma_2, \sigma_2 \blacktriangleright D) \rightarrow (\{\sigma_1\}, \sigma_1 \triangleright E) \tag{3}$$

Clearly, an invocation `o.m(x)` of that method should be valid in an environment $\Gamma\ ;\ \Delta \otimes \rho_1 \otimes \rho_2$ where $\Gamma = \Gamma', o : \rho_1 \triangleright C, x : \rho 2 \blacktriangleright D$. We should be able to replace σ_i with ρ_i $(i = 1, 2)$ in the method type, so that `o.m(x)` can be assigned a type.

We make this intuition precise by universally quantifying over the capabilities required by the method. Type checking then proceeds as follows. First, the argument types are unified yielding a substitution that maps guard variables to concrete guards. The substitution is then applied to the argument types $\overline{F}$ that occur in the method type for checking the usual sub-typing constraints. The set of capabilities $\Delta_{n+1} \otimes \Delta_t$ that is available after assigning types to the receiver and all arguments must be composed of the capabilities required by the method after applying the substitution and additional capabilities Δ_r. The resulting set of capabilities is composed of the capabilities provided by the method after applying the substitution and Δ_r. Capabilities in $\Delta_t - \Delta_0$ are removed from the resulting capabilities, since they correspond to newly created objects that are no longer accessible in the context of the method invocation.

4.4 Typing Instance Creation

Using the type rules presented so far, creating a new instance that should become part of some exposed cluster involves several steps. Firstly, we have to create a unique object, which, according to rule (T-New), requires all constructor parameters to be unique. This means that we cannot pass any references to objects of the target cluster, since they are exposed with guarded types. Therefore, all fields that are supposed to refer to such objects must be initialized to unique dummy objects. In the second step, the new instance must be localized to the target cluster. Finally, all fields pointing to dummy objects have to be reassigned. This pattern can be simplified by using the following additional type rule for creating new instances.

$$\frac{\begin{array}{c}\forall i \in \{1..n\}\ \Gamma\ ;\ \Delta_i \vdash t_i : \rho \triangleright C_i\ ;\ \Delta_{i+1} \otimes \rho \\ \mathit{fields}(C) = \overline{l : D} \qquad \overline{C} <: \overline{D}\end{array}}{\Gamma\ ;\ \Delta_1 \vdash \texttt{new}\ C(\overline{t}) : \rho \triangleright C\ ;\ \Delta_{n+1} \otimes \rho} \quad (\text{T-N}{\small\text{EW}}2)$$

In this rule, we require that all constructor arguments have types guarded by the same capability ρ. Intuitively, all of those objects are part of the same cluster, which is currently exposed under ρ. Assigning a type guarded by ρ to the new instance allows that instance to become part of the same object graph (by subsequent field assignment). Note that the common capability ρ is not consumed in this case.

It is worth noting that any program using this additional type rule can be rewritten to a program that can be type-checked without this rule. For this, we need a way to obtain unique (or immutable) default objects for a given class type. In practical languages like Java, we could use the `null` literal for this purpose. In our simplified core language we assume there is a function $\mathit{default}(D)$ that returns a unique instance of class D by initializing its fields with default instances for their respective types (for simplicity we ignore recursive class types). Then, we can rewrite an instance creation expression ($\texttt{new}\ C(\overline{t}) : \rho \triangleright C$) type-checked with rule (T-NEW2) as follows (let $\mathit{fields}(C) = \overline{l : D}$).

$(\ldots$
$\quad(\texttt{localize}(\texttt{new}\ C(\mathit{default}(\overline{D})), t_1).l_1 = t_1).$
$\ldots).l_n = t_n$

4.5 Unique Fields

If unique references can only be stored on the stack, they have to be explicitly threaded through computations to the point where they are used. Explicit threading of unique references can be avoided by storing them in fields. This way, a single object can provide access to multiple unique references that may be consumed separately. In the annotation system of Section 3, fields that may hold unique references are marked as `@unique`. Any object in a cluster may have fields holding unique references. Since objects in a cluster may be arbitrary aliased, this means that we cannot assume that the unique fields are contained in unique objects. Moreover, when consuming (the target of) a unique field, we have to make sure that the field is re-assigned before it may be accessed again to avoid illegal aliases.

Rather than destructively reading the field, we record the field accessibility in the capability of the containing object. For this, we re-organize the tracked types and their capabilities as follows. Firstly, we omit the class type from tracked types, which are now simply written $tr(\rho)$. Secondly, we extend the capabilities, so that they include the class type plus information about unique fields. A singleton capability is now written $\{\rho \mapsto C[\overline{k : tr(\tau)}]\}$, instead of just $\{\rho\}$. Following the class type C, we provide a sequence of field names together with their (tracked) types. Each element $k_j : tr(\tau_j)$ of the sequence corresponds to a unique field of class C. The capability of a class without unique fields is written

$\{\rho \mapsto C\}$. Note that a simple tracked type $\rho \blacktriangleright C$ with capability $\{\rho\}$ is equivalent to a type $tr(\rho)$ with capability $\{\rho \mapsto C\}$. Each unique field k_j has a tracked type $tr(\tau_j)$. This allows us to control access to it using a capability $\{\tau_j \mapsto D_j\}$, where D_j is the class type of the field. Access to unique fields is provided using an additional rule for `expose`:

$$\frac{\begin{array}{c} \Gamma \;;\; \Delta \vdash t_1 : \rho \triangleright C \;;\; \Delta' \otimes \rho \\ uniqueFields(C) = \overline{k : E} \qquad \rho' fresh \\ \Delta_F := \{\rho' \mapsto C[\overline{k : tr(\tau)}]\}\overline{\otimes\{\tau \mapsto E\}} \\ \Gamma, x : tr(\rho') \;;\; \Delta' \otimes \Delta_F \vdash t_2 : D \;;\; \Delta''' \otimes \sigma\Delta_F \end{array}}{\Gamma \;;\; \Delta \vdash \texttt{expose}\ x = t_1\ \texttt{in}\ t_2 : D \;;\; \Delta''' \otimes \rho} \quad \text{(T-EXP2)}$$

The first difference to the previous rule (T-Exp) is that the object to be exposed, denoted by t_1, must have a guarded type $\rho \triangleright C$; rule (T-Exp) only exposes objects of tracked type. Using the *uniqueFields* predicate, we obtain a list of fields that are marked as unique in class C, together with their types. `expose` introduces x as an alias for (the result of) t_1, which provides access to it under a tracked type $tr(\rho')$. This allows us to record the types of its unique fields in the capability $\{\rho' \mapsto C[\overline{k : tr(\tau)}]\}$. The capability to access C's unique fields, $\overline{\otimes\{\tau \mapsto E\}}$, is available for type-checking t_2, the body of `expose`. This means that the unique fields are accessible through field selection under tracked types in the scope of t_2. Note that capability $\{\rho\}$ is not available in t_2 to avoid exposing t_1 multiple times, which could produce aliases of unique fields. At the end of type-checking t_2, the capability to access the containing object, as well as all unique fields, must be available. Together with the assignment rule below, this makes sure that the unique fields either have not been consumed, or have been re-assigned in the scope of t_2. The substitution σ indicates that some of the guards in $\overline{\tau}$ may be renamed due to re-assignment, which is checked using the following rule.

$$\frac{\begin{array}{c} \Gamma \;;\; \Delta \vdash t_1 : tr(\rho) \;;\; \Delta' \\ l : D \in fields(C) \qquad E <: D \qquad \tau' fresh \\ \Delta_\rho := \{\rho \mapsto C[\overline{k : tr(\tau)}]\} \qquad l = k_j \\ \Gamma \;;\; \Delta' \vdash t_2 : tr(\rho') \;;\; \Delta'' \otimes \{\rho' \mapsto E\} \otimes \Delta_\rho \end{array}}{\Gamma \;;\; \Delta \vdash t_1.l = t_2 : tr(\rho) \;;\; \Delta'' \otimes [\tau'/\tau_j]\Delta_\rho \otimes \{\tau' \mapsto D\}} \quad \text{(T-ASSIGN2)}$$

Term t_1 must have tracked type $tr(\rho)$, and its class must have a unique field l, that is, l must correspond to some k_j in the list of unique fields. The right-hand side t_2 must also have tracked type; its capability is consumed by the assignment. When assigning to a unique field, the capability to access it is restored. To make sure we do not accidentally create duplicate capabilities, we create a fresh guard τ' for the capability of the unique field. Then, we update the type of k_j to $tr(\tau')$ and add the new capability $\{\tau' \mapsto D\}$ to the environment.

Evaluation $\boxed{t, s, d \Downarrow t', s', d'}$

$$\frac{\begin{array}{c} t, s, d \Downarrow r, s', d' \qquad s'(r) = \sigma \triangleright C(\overline{r}) \\ s'(r_i) = \sigma_i \triangleright D(\overline{s}) \qquad s'' = s'[r_i \mapsto \sigma \triangleright D(\overline{s})] \end{array}}{t.l_i, s, d \Downarrow r_i, s'', d'} \qquad \text{(B-}\textsc{Sel}\text{)}$$

$$\frac{\begin{array}{c} t, s, d \Downarrow r, s', d' \\ t', s', d' \Downarrow r', s'', d'' \qquad s''(r) = \sigma \triangleright C(\overline{r}) \\ s''' = s''[r \mapsto \sigma \triangleright C([r'/r_i]\overline{r})] \end{array}}{t.l_i = t', s, d \Downarrow r, s''', d''} \qquad \text{(B-}\textsc{Ass}\text{)}$$

$$\frac{\begin{array}{c} t_1, s, d \Downarrow r_1, s_2, d_2 \\ \forall i \in \{2..n\}.\ t_i, s_i, (d_i - \{\sigma_{i-1}\}) \Downarrow r_i, s_{i+1}, d_{i+1} \\ \forall i \in \{1..n\}.\ s_{i+1}(r_i) = \sigma_i \blacktriangleright D_i(\overline{q_i}) \\ d' = (d_{n+1} - \{\sigma_n\}) \cup \{\sigma\} \qquad \sigma \notin d_{n+1} \\ s' = s_{n+1}[r \mapsto \sigma \blacktriangleright C(\overline{r})] \qquad r \notin dom(s_{n+1}) \end{array}}{\texttt{new}\ C(\overline{t}), s, d \Downarrow r, s', d'} \qquad \text{(B-}\textsc{New}\text{)}$$

$$\frac{\begin{array}{c} t_0, s, d \Downarrow r_0, s_1, d_1 \\ \forall i \in \{1..n\}.\ t_i, s_i, d_i \Downarrow r_i, s_{i+1}, d_{i+1} \\ s_{n+1}(r_0) = \sigma \star C(\overline{r'}) \qquad mbody(m, C) = (\overline{x}, t') \\ [\overline{r}/\overline{x}, r_0/\texttt{this}]t', s_{n+1}, d_{n+1} \Downarrow r', s', d' \end{array}}{t_0.m(\overline{t}), s, d \Downarrow r', s', d'} \qquad \text{(B-}\textsc{Invk}\text{)}$$

$$\frac{\begin{array}{c} t, s, d \Downarrow r, s', d' \qquad s'(r) = \sigma \blacktriangleright C(\overline{r}) \\ d'' = d' - \{\sigma\} \qquad t', s', d'' \Downarrow r', s'', d''' \\ s''(r') = \sigma' \triangleright D(\overline{s}) \qquad s''' = s''[r \mapsto \sigma' \triangleright C(\overline{r})] \end{array}}{\texttt{localize}(t, t'), s, d \Downarrow r, s''', d'''} \qquad \text{(B-}\textsc{Loc}\text{)}$$

$$\frac{\begin{array}{c} t_1, s, d \Downarrow r_1, s', d' \qquad s'(r_1) = \sigma \blacktriangleright D(\overline{s}) \\ \sigma \in d \qquad \sigma' \notin d' \qquad d'' = (d' - \{\sigma\}) \cup \{\sigma'\} \\ s'' = s'[r_1 \mapsto \sigma' \triangleright D(\overline{s})] \qquad [r_1/x]t_2, s'', d'' \Downarrow r_2, s''', d''' \\ \hat{s} = s'''[r_1 \mapsto \sigma \blacktriangleright D(\overline{s})] \qquad \hat{d} = (d''' - \{\sigma'\}) \cup \{\sigma\} \end{array}}{\texttt{expose}\ x = t_1\ \texttt{in}\ t_2, s, d \Downarrow r_2, \hat{s}, \hat{d}} \qquad \text{(B-}\textsc{Exp}\text{)}$$

Fig. 6. Operational Semantics

$$\frac{\forall r \in dom(\Gamma).\ \Gamma \vdash \Gamma(r)\ ok}{\Gamma\ ok} \qquad (\textsc{Store-Typing-WF})$$

$$\frac{\begin{array}{c}\Gamma\ ok \qquad dom(s) = dom(\Gamma) \\ s(r) = \rho \triangleright C(\overline{r}) \Leftrightarrow \Gamma(r) = \rho \triangleright C \\ s(r) = \rho \triangleright C(\overline{r}) \wedge fields(C) = \overline{l : D} \Rightarrow \\ (\forall r_i \in \overline{r}.\ r_i \in dom(\Gamma) \Rightarrow \Gamma(r_i) = \rho \triangleright D_i)\end{array}}{\Gamma \vdash s} \qquad (\textsc{Store-WF})$$

Fig. 7. Store Typing and Well-Formedness

5 Operational Semantics

In this section we discuss the dynamic aspects of our object calculus. The store typing rules shown in Figure 7 are standard. Γ is a store typing that maps reference locations to their types. The STORE-TYPING-WF rule ensures that every type occurring in the store typing is well-formed. A store s maps reference locations to object instantiations $\rho \star C(\overline{r})$ where $\star \in \{\triangleright, \blacktriangleright\}$. The STORE-WF rule ensures that the store typing Γ agrees with the store s on the type of each allocated object; furthermore, the types of objects stored in each field must correspond to the declared field type and have the same guard as the containing object.

Figure 6 shows a big-step operational semantics that defines the evaluation of a term t in the context of a store s and a set d of available capabilities. To simplify the presentation we do not include evaluation rules corresponding to the (T-EXP2) and (T-ASSIGN2) type rules of Section 4.5. The rules for field selection, assignment, instance creation and method invocation are mostly standard, except that they provide additional information about type guards and available capabilities. We use this information subsequently to show a number of invariances that our type system guarantees by relating the operational semantics to the type rules.

Two of the standard expression forms operate on guards and capabilities:

1. In the rule for field selection (B-SEL) the guard of the object that the field points to is updated to the guard of the selector object. This behavior reflects the corresponding type rule (T-SEL).
2. In the rule for instance creation (B-NEW) a fresh capability σ is created that is used to track the allocated object; σ is added to the capabilities that are available after the evaluation.

Reducing a `localize` expression consumes the capability that is used to track the object r that t reduces to; the type of r is changed such that it is guarded by the same capability that guards the object denoted by t'.

In the rule for `expose` we first determine the capability σ that tracks the object r_1 that is the result of reducing t_1. While reducing t_2 σ is not available; instead, r_1 is guarded by a fresh capability σ'. After t_2 has been reduced, the temporary capability σ' is replaced with σ; we also change the type of r_1 such that it is again tracked by σ.

5.1 Type Preservation

In this section we prove a standard type preservation theorem that relates the above operational semantics to the type system that we presented in Section 4.

Lemma 1 (Substitution) *If $\Gamma, x : S \; ; \; \Delta \vdash t : T \; ; \; \Delta'$ and $\Gamma(r) = S$, then $\Gamma \; ; \; \Delta \vdash [r/x]t : T \; ; \; \Delta'$.*

Proof: By induction on the typing derivation. $\square$

Lemma 2 *If $\Gamma \; ; \; \Delta \vdash t : T \; ; \; \Delta'$, then either $\Delta' \subseteq \Delta$, or $(T = \rho \blacktriangleright D \wedge \Delta' = \rho \otimes \Delta'' \wedge \Delta'' \subseteq \Delta)$.*

Proof: By induction on the typing derivation. The cases (T-VAR), (T-SEL), (T-ASS), (T-NEW), (T-LOC), and (T-EXP) are simple.
Case (T-INVK):
By IH, $\forall i \in \{0..n\}\ \Delta_{i+1} \subseteq \Delta_i$. By (WF-METHOD), $\sigma\Delta'' \subseteq \sigma\Delta'$ or $\sigma\Delta'' = \hat{\Delta} \otimes \rho \wedge \hat{\Delta} \subseteq \sigma\Delta' \wedge \sigma C = \rho \blacktriangleright D$. Moreover, $\Delta_{n+1} \otimes \Delta_t \subseteq \Delta_0 \otimes (\Delta_t - \Delta_0) \Rightarrow \sigma\Delta' \otimes \Delta_r \subseteq \Delta_0 \otimes (\Delta_t - \Delta_0)$. Therefore, $\Delta''' - (\Delta_t - \Delta_0) \subseteq \Delta_0$ or $\Delta''' - (\Delta_t - \Delta_0) = \Delta'_0 \otimes \rho$ and $\sigma C = \rho \blacktriangleright D$ where $\Delta'_0 \subseteq \Delta_0$.
$\square$

Lemma 3 (Weakening) *If $\Gamma \; ; \; \Delta \vdash t : T \; ; \; \Delta_f$, $\Delta' \supseteq \Delta$, and Γ' is a store-typing such that $\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \vee (\Gamma(q) = \rho \blacktriangleright C \wedge \rho \notin \Delta')$. Then: $\Gamma' \; ; \; \Delta' \vdash t : T \; ; \; \Delta_f \otimes (\Delta' - \Delta)$.*

Proof: By induction on the typing derivation with case analysis on the last type rule used in the derivation.
Case (T-SEL): Immediate from the IH.
Case (T-ASS):
We have $\Gamma \; ; \; \Delta \vdash t_1.l_i = t_2 : \rho \triangleright C \; ; \; \Delta''$. By IH, $\Gamma' \; ; \; \Delta' \vdash t_1 : \rho \triangleright C \; ; \; \Delta'' \otimes (\Delta' - \Delta)$. By Lemma 2, $\Delta'' \subseteq \Delta \subseteq \Delta'$, therefore $\Delta'' \otimes (\Delta' - \Delta) \subseteq \Delta'$. By IH, $\Gamma' \; ; \; \Delta'' \otimes (\Delta' - \Delta) \vdash t_2 : \rho \triangleright E \; ; \; \Delta_f \otimes (\Delta' - \Delta)$. Applying (T-ASS) closes this case.
Cases (T-LOC) and (T-EXP): analogous to case (T-ASS).
Case (T-NEW):
We have $\forall i \in \{1..n\}\ \Gamma \; ; \; \Delta_i \vdash t_i : \rho_i \blacktriangleright C_i \; ; \; \Delta_{i+1} \otimes \rho_i$. By Lemma 2, $\forall i \in \{1..n\}\ \Delta_i \otimes (\Delta' - \Delta_1) \subseteq \Delta'$. Therefore by IH, $\forall i \in \{1..n\}\ \Gamma' \; ; \; \Delta_i \otimes (\Delta' - \Delta_1) \vdash t_i : \rho_i \blacktriangleright C_i \; ; \; \Delta_{i+1} \otimes \rho_i \otimes (\Delta' - \Delta_1)$. Applying (T-NEW) closes this case. Case (T-INVK):
By Lemma 2, $\forall i \in \{1..n\}\ \Delta_{i+1} \subseteq \Delta_i \Rightarrow \forall i \in \{1..n\}\ \Delta_i \subseteq \Delta_0 \Rightarrow \forall i \in \{1..n\}\ \Delta_i \otimes (\Delta' - \Delta_0) \subseteq \Delta'$. By IH, $\forall i \in \{1..n\}\ \Gamma' \; ; \; \Delta_i \otimes (\Delta' - \Delta_0) \vdash t_i : U_i \; ; \; \Delta'_{i+1} \otimes (\Delta' - \Delta_0)$. Since $\Delta_{n+1} \otimes \Delta_t \otimes (\Delta' - \Delta_0) = \sigma\Delta' \otimes \Delta_r \otimes (\Delta' - \Delta_0)$, we can apply (T-INVK), which closes this case. $\square$

We are now ready to prove a type preservation theorem. We use the notation $\Delta \vdash d$ for indicating that the static capabilities Δ correctly approximate the dynamic capabilities d, that is, the dynamic capabilities contain at least the capabilities available statically. Formally, $\Delta \vdash d$ if and only if $\forall \rho \in \Delta.\ \rho \in d$.

Note that in the following type preservation theorem, the store typing Γ' that approximates the store after evaluation is not a superset of the initial store typing Γ. The reason is that references may be localized during evaluation; localizing a reference changes its tracked type $\rho \blacktriangleright C$ to a guarded type $\rho' \triangleright C$ while consuming ρ. Note that the underlying class type does not change.

Theorem 1 (Type Preservation) *If $\Gamma \ ; \ \Delta \vdash t : T \ ; \ \Delta'$, $\Gamma \vdash s$, $\Delta \vdash d$, and $t, s, d \Downarrow r, s', d'$, then $\Gamma'(r) = T$ and $\Delta' \vdash d'$ for some $\Gamma' \vdash s'$ such that*

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta') \tag{4}$$

Proof: By induction on the typing derivation with case analysis on the last type rule used in the derivation.
Case (T-Sel):
We have $\Gamma \ ; \ \Delta \vdash t'.l_i : \rho \triangleright D_i \ ; \ \Delta'$ for some ρ. By (T-Sel), $\Gamma \ ; \ \Delta \vdash t' : \rho \triangleright C \ ; \ \Delta'$ with $fields(C) = \overline{l : D}$. For evaluation only (B-Sel) applies. Therefore, $t', s, d \Downarrow r', s'', d'$. By IH, $\Gamma_1(r') = \rho \triangleright C$ and $\Delta' \vdash d'$ for some $\Gamma_1 \vdash s''$ such that

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma_1 \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta') \tag{5}$$

By (B-Sel), $s''(r') = \sigma \triangleright C(\overline{r})$ for some σ. Since $\Gamma_1 \vdash s''$, by (Store-WF): $\sigma = \rho$. Define $\Gamma' := \Gamma_1[r_i \mapsto \rho \triangleright D_i]$. By (B-Sel), $s' = s''[r_i \mapsto \rho \triangleright D_i(\overline{s})]$. We have $\Gamma' \supseteq \Gamma_1 \vdash s''$. Since $\Gamma'(r_i) = \rho \triangleright D_i$, we have by (Store-WF) that $\Gamma' \vdash s'$. It remains to show:

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta') \tag{6}$$

We show that $\Gamma' \supseteq \Gamma$, which implies (6).

1. Case $r_i \notin dom(\Gamma_1)$. Then, $\Gamma' \supset \Gamma_1 \supseteq \Gamma$.
2. Case $r_i \in dom(\Gamma_1)$. Since $s''(r') = \rho \triangleright C(\overline{r})$ and $\Gamma_1 \vdash s''$, we have by (Store-WF) that $\Gamma_1(r_i) = \rho \triangleright D_i$. Therefore, $\Gamma' = \Gamma_1 \supseteq \Gamma$.

Case (T-Ass):
We have $\Gamma \ ; \ \Delta \vdash t.l_i = t' : \rho \triangleright C \ ; \ \Delta'$ for some ρ. By (T-Ass), $\Gamma \ ; \ \Delta \vdash t : \rho \triangleright C \ ; \ \Delta''$ and $fields(C) = \overline{l : D}$. For evaluation only (B-Ass) applies. Therefore, $t, s, d \Downarrow r, s'', d''$. By IH, $\Gamma_1(r) = \rho \triangleright C$ and $\Delta'' \vdash d''$ for some $\Gamma_1 \vdash s''$ such that

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma_1 \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta'') \tag{7}$$

Weakening provides $\Gamma_1 \ ; \ \Delta'' \vdash t' : \rho \triangleright E \ ; \ \Delta'$. By (B-Ass), $t', s'', d'' \Downarrow r', s''', d'$. Therefore, by IH, $\Gamma'(r') = \rho \triangleright E$ and $\Delta' \vdash d'$ for some $\Gamma' \vdash s'''$ such that

$$\forall q \in dom(\Gamma_1).\ q : \Gamma_1(q) \in \Gamma' \vee (\Gamma_1(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta') \tag{8}$$

By (Store-WF), $s'''(r) = \rho \triangleright C(\overline{r})$. By (B-Ass), $s' = s'''[r \mapsto \rho \triangleright C([r'/r_i]\overline{r})]$. Since $\Gamma' \vdash s'''$ we have $\Gamma'(r) = \rho \triangleright C$. Thus, $\Gamma' \vdash s'$. It remains to show

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta') \tag{9}$$

Let $u \in dom(\Gamma)$.

1. Case $u : \Gamma(u) \in \Gamma_1$. Then either $u : \Gamma(u) \in \Gamma'$, or $\Gamma(u) = \sigma \blacktriangleright E$ and $\sigma \notin \Delta''$, in which case by Lemma 2, $\sigma \notin \Delta'$.
2. Case $\Gamma(u) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta''$. By Lemma 2, $\sigma \notin \Delta'$.

Case (T-New):
We have $\Gamma\ ;\ \Delta_1 \vdash \texttt{new}\ C(\overline{t}) : \rho \blacktriangleright C\ ;\ \Delta'$. By (T-New), $\forall i \in \{1..n\}.\ \Gamma\ ;\ \Delta_i \vdash t_i : \rho_i \blacktriangleright C_i\ ;\ \Delta_{i+1} \otimes \rho_i$, $fields(C) = \overline{l : D}$, and $\overline{C} <: \overline{D}$. For evaluation only (B-New) applies. Therefore, $t_1, s, d \Downarrow r_1, s_2, d_2$. By IH, $\Gamma_1(r_1) = \rho_1 \blacktriangleright C_1$ for some $\Gamma_1 \vdash s_2$ and $\Delta_2 \otimes \rho_1 \vdash d_2$ such that

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma_1 \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta_2 \otimes \rho_1) \tag{10}$$

Weakening provides $\Gamma_1\ ;\ \Delta_2 \vdash t_2 : \rho_2 \blacktriangleright C_2\ ;\ \Delta_3 \otimes \rho_2$. Since $\Delta_2 \otimes \rho_1 \vdash d_2$, we have $\Delta_2 \vdash (d_2 - \{\rho_1\})$. We apply the IH iteratively for all $t_i \in \overline{t}$, and we obtain $\forall i \in \{1..n\}.\ \Gamma_i(r_i) = \rho_i \blacktriangleright C_i$ for some $\Gamma_i \vdash s_{i+1}$ and $\Delta_{i+1} \otimes \rho_i \vdash d_{i+1}$ such that

$$\forall q \in dom(\Gamma_{i-1}).\ q : \Gamma_{i-1}(q) \in \Gamma_i \vee (\Gamma_{i-1}(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta_{i+1} \otimes \rho_i) \tag{11}$$

Define $\Gamma' := \Gamma_n[r \mapsto \rho \blacktriangleright C]$ and $\Delta' := \Delta_{n+1} \otimes \rho$ where $r \notin dom(s_{n+1})$ and $\rho \notin d_{n+1}$. By (B-New), $s' = s_{n+1}[r \mapsto \rho \blacktriangleright C(\overline{r})]$. Since $\Gamma_n \vdash s_{n+1}$, we have $\Gamma' \vdash s'$. We have $\Delta_{n+1} \otimes \rho_n \vdash d_{n+1}$. Therefore, $\Delta_{n+1} \vdash d_{n+1} - \{\rho_n\}$, and $\Delta' \vdash (d_{n+1} - \{\rho_n\}) \cup \{\rho\} = d'$. It remains to show:

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta') \tag{12}$$

Let $u \in dom(\Gamma) \Rightarrow u \neq r$ (since $r \notin dom(s)$ and $\Gamma \vdash s$).

1. Case $u : \Gamma(u) \in \Gamma_n \Rightarrow u : \Gamma(u) \in \Gamma'$.
2. Case $\Gamma_{i-1}(u) = \sigma \blacktriangleright E$ and $\sigma \notin \Delta_{i+1} \otimes \rho_i$. Then, $\Gamma(u) = \sigma \blacktriangleright E$ and by Lemma 2, $\sigma \notin \Delta_{n+1}$. Since $u \neq r$, $\sigma \notin \Delta'$.

Case (T-Loc):
We have $\Gamma\ ;\ \Delta \vdash \texttt{localize}(t, t') : \rho' \triangleright C\ ;\ \Delta'$ where $\Gamma\ ;\ \Delta \vdash t : \rho \blacktriangleright C\ ;\ \Delta'' \otimes \rho$ and $\Gamma\ ;\ \Delta'' \vdash t' : \rho' \triangleright D\ ;\ \Delta'$. For evaluation only (B-Loc) applies. Therefore, $t, s, d \Downarrow r, s'', d''$. By IH, $\Gamma_1(r) = \rho \blacktriangleright C$ for some $\Gamma_1 \vdash s''$ and $\Delta'' \otimes \rho \vdash d''$ such that

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma_1 \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta'' \otimes \rho) \tag{13}$$

By Weakening Lemma, $\Gamma_1 \ ; \ \Delta'' \ \vdash \ t' : \rho' \triangleright D \ ; \ \Delta'$. By (Store-WF), $s''(r) = \rho \blacktriangleright C(\overline{r})$. Let $d''' = d'' - \{\rho\}$, then $\Delta'' \ \vdash \ d'''$. By (B-Loc), $t', s'', d''' \ \Downarrow \ r', s''', d'$. By IH, $\Gamma_2(r') = \rho' \triangleright D$ for some $\Gamma_2 \ \vdash \ s'''$ and $\Delta' \ \vdash \ d'$ such that

$$\forall q \in dom(\Gamma_1).\ q : \Gamma_1(q) \in \Gamma_2 \lor (\Gamma_1(q) = \sigma \blacktriangleright E \land \sigma \notin \Delta') \tag{14}$$

By (Store-WF), $s'''(r') = \rho' \triangleright D(\overline{s})$. By (B-Loc), $s' = s'''[r \mapsto \rho' \triangleright C(\overline{r})]$. Define $\Gamma' := \Gamma_2[r \mapsto \rho' \triangleright C]$. Then, $\Gamma' \ \vdash \ s'$. It remains to show:

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \lor (\Gamma(q) = \sigma \blacktriangleright E \land \sigma \notin \Delta') \tag{15}$$

Let $u \in dom(\Gamma)$.

1. Case $u \neq r$.
   (a) Subcase $u : \Gamma(u) \in \Gamma_1$. Then, either $u : \Gamma_1(u) \in \Gamma_2$, and thus, $u : \Gamma_1(u) \in \Gamma'$. Or, $\Gamma_1(u) = \sigma \blacktriangleright E \land \sigma \notin \Delta'$.
   (b) Subcase $\Gamma(u) = \sigma \blacktriangleright E \land \sigma \notin \Delta'' \otimes \rho$. Thus, by Lemma 2, $\sigma \notin \Delta'$.
2. Case $u = r$. Then, $\Gamma_1(r) = \rho \blacktriangleright C \land \rho \notin \Delta'' \Rightarrow \rho \notin \Delta'$.
   (a) Subcase $r : \Gamma(r) \in \Gamma_1$. Then, by (13), $\Gamma(r) = \rho \blacktriangleright C \land \rho \notin \Delta'$.
   (b) Subcase $\Gamma(r) = \sigma \blacktriangleright E \land \sigma \notin \Delta'' \otimes \rho$. Thus, by Lemma 2, $\sigma \notin \Delta'$.

Case (T-Invk):
We have $\forall i \in \{0..n\} \ \Gamma \ ; \ \Delta_i \ \vdash \ t_i : U_i \ ; \ \Delta'_{i+1}$. By (B-Invk), $t_0, s, d \ \Downarrow \ r_0, s_1, d_1$. Therefore, by IH, $\Gamma_0(r_0) = U_0$ for some $\Gamma_0 \ \vdash \ s_1$ and $\Delta'_1 \ \vdash \ d_1$ such that

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma_0 \lor (\Gamma(q) = \tau \blacktriangleright E \land \tau \notin \Delta'_1) \tag{16}$$

By Weakening Lemma, $\Gamma_0 \ ; \ \Delta_1 \ \vdash \ t_1 : U_1 \ ; \ \Delta'_2$. We apply the IH iteratively for all $t_i \in \overline{t}$ and obtain $\forall i \in \{1..n\}$. $\Gamma_i(r_i) = U_i$ where $\Gamma_i \ \vdash \ s_{i+1}$ and $\Delta'_{i+1} \ \vdash \ d_{i+1}$ such that

$$\forall q \in dom(\Gamma_{i-1}).\ q : \Gamma_{i-1}(q) \in \Gamma_i \lor (\Gamma_{i-1}(q) = \tau \blacktriangleright E \land \tau \notin \Delta'_{i+1}) \tag{17}$$

By (T-Invk), $mtype(m, C_0) = [\rho \star](\Delta', \overline{F}) \to (\Delta'', C)$ and by (WF-Method), $\Gamma_P \ ; \ \Delta' \ \vdash \ t' : T_0 \ ; \ \Delta''$ and $T_0 <: C$ where $mbody(m, C_0) = (\overline{x}, t')$ and $\Gamma_P = \overline{x : F}, \texttt{this} : \rho \star C_0$. Let $\sigma = unify(\overline{U}, \overline{F})$. Then $\sigma\Gamma_P \ ; \ \sigma\Delta' \ \vdash \ t' : \sigma T_0 \ ; \ \sigma\Delta''$ since σ is injective.

Weakening provides $\Gamma_n, \sigma\Gamma_P \ ; \ \sigma\Delta' \otimes \Delta_r \ \vdash \ t' : \sigma T_0 \ ; \ \sigma\Delta'' \otimes \Delta_r$. By the Substitution Lemma, $\Gamma_n \ ; \ \sigma\Delta' \otimes \Delta_r \ \vdash \ [\overline{r}/\overline{x}, r_0/\texttt{this}]t' : \sigma T_0 \ ; \ \sigma\Delta'' \otimes \Delta_r$. Furthermore, we have $\Delta_{n+1} \otimes \Delta_t \ \vdash \ d_{n+1}$ and $\Gamma_n \ \vdash \ s_{n+1}$. By IH, $\Gamma'(r') = \sigma T_0$ for some $\Gamma' \ \vdash \ s'$ and $\Delta' = \sigma\Delta'' \otimes \Delta_r \ \vdash \ d'$ such that

$$\forall q \in dom(\Gamma_n).\ q : \Gamma_n(q) \in \Gamma' \lor (\Gamma_n(q) = \tau \blacktriangleright E \land \tau \notin \sigma\Delta'' \otimes \Delta_r) \tag{18}$$

It remains to show:

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \lor (\Gamma(q) = \tau \blacktriangleright E \land \tau \notin \Delta') \tag{19}$$

Let $u \in dom(\Gamma)$.

1. Case $u : \Gamma(u) \in \Gamma_n$. Then either $u : \Gamma(u) \in \Gamma'$, or $\Gamma(u) = \Gamma_n(u) = \tau \blacktriangleright E \wedge \tau \notin \Delta'$.
2. Case $\Gamma(u) = \Gamma_{i-1}(u) = \tau \blacktriangleright E \wedge \tau \notin \Delta'_{i+1} \Rightarrow \tau \notin \Delta_t$. By Lemma 2, $\tau \notin \Delta_{n+1}$ and also $\tau \notin \Delta'$.

Case (T-Exp):
We have $\Gamma \; ; \; \Delta \vdash \texttt{expose}\ x = t_1\ \texttt{in}\ t_2 : T \; ; \; \Delta'' \otimes \tau$ where $\Gamma \; ; \; \Delta \vdash t_1 : \tau \blacktriangleright C \; ; \; \Delta' \otimes \tau$ and $\Gamma, x : \sigma' \triangleright C \; ; \; \Delta' \otimes \sigma' \vdash t_2 : T \; ; \; \Delta'' \otimes \sigma'$ $(\sigma' \notin \Gamma \cup \Delta \cup \Delta' \cup \Delta'')$. By (B-Exp), $t_1, s, d \Downarrow r_1, s', d'$. By IH, $\Gamma_1(r_1) = \tau \blacktriangleright C$ for some $\Gamma_1 \vdash s'$ and $\Delta' \otimes \tau \vdash d'$ such that

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma_1 \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta' \otimes \tau) \qquad (20)$$

We have $\sigma' \notin d'$, since $\Delta' \otimes \tau \nvdash \sigma'$. By (Store-WF), $s'(r_1) = \tau \blacktriangleright C(\overline{s})$. By (B-Exp), $d'' = (d' - \{\tau\}) \cup \{\sigma'\}$ and $s'' = s'[r_1 \mapsto \sigma' \triangleright C(\overline{s})]$. Therefore, $\Delta' \otimes \sigma' \vdash d''$. Since $\Delta' \nvdash \tau$, we have by Weakening Lemma, $\Gamma_1, x : \sigma' \triangleright C \; ; \; \Delta' \otimes \sigma' \vdash t_2 : T \; ; \; \Delta'' \otimes \sigma'$.

Define $\Gamma'' := \Gamma_1[r_1 \mapsto \sigma' \triangleright C]$. Then, $\Gamma'' \vdash s''$. Weakening provides $\Gamma'', x : \sigma' \triangleright C \; ; \; \Delta' \otimes \sigma' \vdash t_2 : T \; ; \; \Delta'' \otimes \sigma'$. By Substitution Lemma, $\Gamma'' \; ; \; \Delta' \otimes \sigma' \vdash [r_1/x]t_2 : T \; ; \; \Delta'' \otimes \sigma'$. By (B-Exp), $[r_1/x]t_2, s'', d'' \Downarrow r_2, s''', d'''$. By IH, $\Gamma_2(r_2) = T$ for some $\Gamma_2 \vdash s'''$ and $\Delta'' \otimes \sigma' \vdash d'''$ such that

$$\forall q \in dom(\Gamma'').\ q : \Gamma''(q) \in \Gamma_2 \vee (\Gamma''(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta'' \otimes \sigma') \qquad (21)$$

By (B-Exp), $\hat{s} = s'''[r_1 \mapsto \tau \blacktriangleright C(\overline{s})]$ and $\hat{d} = (d''' - \{\sigma'\}) \cup \{\tau\}$. Thus, $\Delta'' \otimes \tau \vdash \hat{d}$. Define $\Gamma' := \Gamma_2[r_1 \mapsto \tau \blacktriangleright C]$. Then, $\Gamma' \vdash \hat{s}$. It remains to show:

$$\forall q \in dom(\Gamma).\ q : \Gamma(q) \in \Gamma' \vee (\Gamma(q) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta'' \otimes \tau) \qquad (22)$$

Let $u \in dom(\Gamma)$.

1. Case $u : \Gamma(u) \in \Gamma_1$.
   (a) Subcase $u \neq r_1 \Rightarrow u : \Gamma(u) \in \Gamma'' \Rightarrow u : \Gamma(u) \in \Gamma_2 \vee (\Gamma(u) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta'' \otimes \sigma') \Rightarrow u : \Gamma(u) \in \Gamma' \vee (\Gamma(u) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta'' \otimes \tau)$.
   (b) Subcase $u = r_1 \Rightarrow \Gamma_1(u) = \tau \blacktriangleright C$. Then, by (20), $\Gamma(u) = \tau \blacktriangleright C = \Gamma'(u)$, i.e., $u : \Gamma(u) \in \Gamma'$.
2. Case $u : \Gamma(u) \notin \Gamma_1 \Rightarrow u \neq r_1$. By (20), $\Gamma(u) = \sigma \blacktriangleright E \wedge \sigma \notin \Delta' \otimes \tau$. Thus, $\sigma \notin \Delta'$. By Lemma (2) and (21), $\sigma \notin \Delta'' \otimes \tau$.

□

We have not done a proof of a progress theorem, which would be needed for a complete soundness proof for the type system. Instead, we focus on establishing the alias invariant that we introduced in Section 3.3. The following section introduces a theorem that states that references of tracked type are externally-unique when they are accessible.

6 Aliasing Guarantees

In this section we formalize the aliasing properties that our system guarantees. We start by defining a reachability property of object locations.

Definition 1 (Stack-Confinement) *A reference location $r \in dom(s)$ is stack-confined in store s if and only if*
$(\forall r', r'' \in dom(s).\ (reachable_s(r, r')$
$\wedge\ reachable_s(r'', r')) \Rightarrow reachable_s(r, r''))$ *and*
$(\forall r'.\ reachable_s(r', r) \Rightarrow reachable_s(r, r'))$

Lemma 4 *Let s be a store and $r, r' \in dom(s)$ reference locations such that $reachable_s(r, r')$, $guard(r, s) = \rho$, and r is stack-confined.*
If $\Gamma\ ;\ \Delta \vdash t : T\ ;\ \Delta'$, and $t, s, d\ \Downarrow\ r', s', d'$ with $\Gamma \vdash s$ and $\Delta \vdash d$, then $guard(r', s') = \rho$.

Proof Sketch: By induction on the typing derivation. □

Lemma 5 (Expose Preserves Stack-Confinement) *Let $\Gamma\ ;\ \Delta \vdash t_1 : \sigma \blacktriangleright T\ ;\ \Delta' \otimes \sigma$ and $t_1, s, d\ \Downarrow\ r, s_1, d_1$ with $\Gamma \vdash s$, $\Delta \vdash d$, and r is stack-confined in s_1. If $t = \texttt{expose}\ x = t_1\ \texttt{in}\ t_2$ is well-typed and $t, s, d\ \Downarrow\ r_f, s_f, d_f$, then r is stack-confined in s_f.*

Proof: According to definition 1 there are two ways to break stack-confinement of r. We show that establishing $reachable_{s_f}(\hat{r}, r')$ for some $\hat{r}$ such that $reachable_{s_f}(r, r')$ implies $reachable_{s_f}(r, \hat{r})$ or $\hat{r}$ is garbage in s_f. (The proof for the other case is analogous.) We use the simplifying assumption that $reachable_{s_i}(r, r')$ throughout all stores $s_i \in s \dots s_f$.

Part 1. Let s' be the first store in the evaluation of t with $reachable_{s'}(\hat{r}, r')$. In the following we show: $guard(\hat{r}, s') = guard(r, s')$

To establish $reachable_{s_f}(\hat{r}, r')$, the evaluation of t must contain an instance of the rule (B-Ass) such that $\hat{t}.l_i = t', s_a, d_a\ \Downarrow\ \hat{r_r}, s', d'$ with

$$\hat{t}, s_a, d_a\ \Downarrow\ \hat{r_r}, \hat{s}, \hat{d} \tag{23}$$

$$t', \hat{s}, \hat{d}\ \Downarrow\ r', s'', d'' \tag{24}$$

where either $reachable_{s'}(\hat{r}, \hat{r_r})$ or $\hat{r} = \hat{r_r}$. For simplicity we assume $\hat{r} = \hat{r_r}$. By (B-Ass), $s''(\hat{r_r}) = \sigma \triangleright C(\overline{r})$ and $s' = s''[\hat{r_r} \mapsto \sigma \triangleright C([r'/r_i]\overline{r})]$. Since the assignment is well-typed, we have by type preservation that $guard(\hat{r}, s') = guard(r', s')$. Since $reachable_{\hat{s}}(r, r')$ and r is stack-confined in $\hat{s}$, we have by Lemma 4 that $guard(r', s') = guard(r', s'') = guard(r, \hat{s})$. It remains to show that $guard(r, s') = guard(r, \hat{s})$. Assume this is not the case. The only rule that could change the guard of r is (B-Sel); the rules (B-Loc) and (B-Exp) are not applicable since r is exposed. Therefore, to change r's guard in the evaluation of t', there must be an instance of rule (B-Sel) such that $t_b.l_i, s_b, d_b\ \Downarrow\ r, s_c, d_c$ where $t_b, s_b, d_b\ \Downarrow\ r_b, s'_b, d'_b$, $reachable_{s_b}(r_b, r)$, and $guard(r_b, s'_b) \neq guard(r, s'_b)$.

Then, by stack-confinement, $reachable_{s_b}(r, r_b)$. By Lemma 4, $guard(r_b, s'_b) = guard(r, s_b)$. We assume that the guard of r does not change during the evaluation from s_b to s'_b. Therefore, $guard(r_b, s'_b) = guard(r, s'_b) = guard(r, s_b)$. Contradiction. Therefore, we have shown $guard(\hat{r}, s') = guard(r, s')$.

Part 2. We show that either $reachable_{s'}(r, \hat{r})$ or $\hat{r}$ is garbage in s_f. Consider the sequence of stores $s, \ldots, \hat{s}, \ldots s' = S$. If there is no $s_i \in S$ such that $reachable_{s_i}(r, \hat{r})$, then $guard(\hat{r}, s') \neq guard(r, s')$ since `expose` creates a fresh guard (B-Exp) that is only propagated through field selection (B-Sel). Contradiction. Therefore, $reachable_{s_i}(r, \hat{r})$ for some $s_i \in S$. If not $reachable_{s_f}(r, \hat{r})$, then $\hat{r}$ is garbage in s_f since, by subject reduction, only objects reachable from r may refer to $\hat{r}$ according to the guards. □

Theorem 2 (External Uniqueness) *Let* $\Gamma \; ; \; \Delta \vdash t_1 : \sigma \blacktriangleright T \; ; \; \Delta' \otimes \sigma$ *and* $t_1, s, d \Downarrow r, s_1, d_1$ *with* $\Gamma \vdash s$, $\Delta \vdash d$, *and* r *is externally-unique in* s_1.

If t *is a well-typed term,* $t, s, d \Downarrow r_f, s_f, d_f$, *and* $\rho \in d_f$, *then* r *is externally-unique in* s_f.

Proof Sketch: First, r remains stack-confined, since the fields of the object o that r points to are only accessible when r is exposed, which preserves stack-confinement according to Lemma 5. Second, according to the evaluation rules (B-New) and (B-Loc) the capability ρ is consumed before r can be aliased. Taken together, r remains the only accessible external reference to o in s_f. □

7 Extensions

In this section we address some of the issues when integrating OCT into full languages like Scala or Java that we ommitted from the formalization for simplicity.

7.1 Closures

A number of object-oriented languages, such as Scala, have special support for closures. In this section we discuss how OCT can be extended to handle closures that capture unique references in their environment. We show that in certain cases, such a closure can be given a tracked type, thus providing external uniqueness.

In the case where a unique reference is captured by a closure, we have to be careful not to introduce accidental aliasing, so that the unique reference could be accessed and consumed multiple times. Consider the following example:

```
val r: C @unique = new C(...)
val myClosure = (x: Int) => {
  expose (r) { ... }
}
consume(r)
myClosure(5) // error!
```

The method creates a new instance of class `C` with a unique reference `r` (of type $\rho \blacktriangleright$ `C` for some ρ) pointing to it. This unique reference is captured by the closure that is assigned to `myClosure`. Note that type checking the `expose` expression is only possible in an environment where the capability of `r` is available. The invocation of `consume` consumes `r`. Finally, the closure is applied to some argument. This application should be rejected by the type checker since the unique object captured by the closure has already been consumed.

The following approach enables type-safe ownership transfer for closures. The idea is to record the information about required capabilities in the type of the closure. For simplicity we consider only the special case where a closure captures a single unique reference. Type checking proceeds according to the following inference rule:

$$\frac{\begin{array}{c}\Gamma, x : T_1 \ ; \ \Delta \vdash t : T_2 \ ; \ \Delta' \\ \Delta - (\Delta'|_{dom(\Delta)}) = \{\rho\} \qquad \rho \blacktriangleright T \in \Gamma\end{array}}{\Gamma \ ; \ \Delta \vdash \lambda x : T_1 .\ t : \rho \blacktriangleright (T_1 \to T_2) \ ; \ \Delta' \otimes \rho} \quad \text{(T-CLOS)}$$

Type-checking the body of the closure yields Δ' which lets us compute the set of capabilities that are consumed as $\Delta - (\Delta'|_{dom(\Delta)})$. If this set contains only a single capability ρ that is the component of a tracked type in the environment, then the type of the closure must also be tracked with ρ. Otherwise, an application of the closure could violate the uniqueness of the captured reference. This means that the evaluation context can choose to either consume the closure or the variable that was captured by the closure, but not both, since no additional capability is created. Note that the capability ρ, which is available in the initial capabilities Δ, is not consumed.

7.2 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):

```
override 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
    }
  }
```

In this example, it would be erroneous to annotate the receiver as `@transient`. The nested class instance captures a reference to the receiver and stores it into a field. Therefore, the iterator instance would have to be tracked with the same capability as the receiver, which is possible as we discussed above for the special case of closures. The `elements` method would then have to return a unique object that is tracked *with the same capability as the receiver*. While the type system that we introduced in Section 4 (which our implementation uses internally) can express this, our user-visible annotations cannot.

Fortunately, in our type system it is easy to check for illegal aliasing: nested class definitions are type-checked in an environment where the set of available capabilities is empty. This way, all capturing occurrences of unique references are detected as illegal.

However, a more flexible approach is to return an exposed iterator that operates on an exposed collection:

```
@exposed override def elements:
  Iterator[A] @exposed = { ... }
```

For convenience, we can implicitly localize new instances of the nested class to the cluster of the captured exposed reference. It is illegal to capture a unique reference in addition to an exposed reference. The above annotation allows arbitrary uses of an iterator instance while its underlying unique (or transient) collection is exposed. Closures that capture exposed references are supported in an analogous way.

7.3 Actor-Based Concurrency

Figure 8 shows a concurrent variant of an example adapted from [30]. The purpose of this example is to demonstrate how our system supports the repeated ownership transfer of single objects. This is common in work flow or communication systems that process packets or tasks. Our example models a concurrent work flow system for processing orders (class `Order`) using a pipeline of concurrently running processors (abstract class `Processor`).

A `Dispatcher` handles orders by iterating over its pipeline, sending the current order to each processor in sequence. The expression `(pipeline(i) !? o)` sends `o` as a message to the processor at index i of the pipeline and waits for a reply. The result of the expression is the received reply; its type is recovered by matching on it (using Scala's `match`). Each processor is implemented as an actor that waits to receive an order, processes it, and sends it back to the dispatcher. A processor receives each order as an externally unique object with the capability to consume it. To gain access to its fields, the order is exposed using `expose`. The processor's `doWork` method is then free to operate on the exposed order; the type system ensures that its external uniqueness is preserved. For example, the `Pricer` processor updates the order's `total` field.

```
class Order {
  var clientId: Int = 0
  var items: List[Int] = List()
  // fields to be filled by work flow stages
  var total: Int = 0
  // other fields and methods omitted
}

abstract class Processor extends Actor {
  def act() {
    react {
      case ord: Order =>
        expose (ord) { xord =>
          doWork(xord)
        }
        sender ! ord
        act()
    }
  }
  def doWork(current: Order @exposed)
}

class Pricer extends Processor {
  def doWork(current: Order @exposed) {
    val price: Int = /* determine price */
    current.total = price
  }
  // other fields and methods omitted
}

abstract class Dispatcher {
  val pipeline = new Array[Processor](5)
  def handleOrder(o: Order @unique) {
    var tmp: Order @unique = o
    for (i <- 0 until pipeline.length)
      (pipeline(i) !? tmp) match {
        case ord: Order @unique => tmp = ord
      }
  }
  // other fields and methods omitted
}
```

Fig. 8. Example: Concurrent work flow

8 Implementation

We have implemented a type checker for Object Capability Types as a compiler plug-in for the Scala compiler developed at EPFL.[1] 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. Secondly, the additional phase infers tracked and guarded types, and in some cases, `localize` and `expose` expressions (see Section 3), for each method separately. Then, types and capabilities are checked using (an extension of) the type rules presented in Section 4. For subsequent code generation, all guards, `localize` and `expose` expressions are erased.

8.1 Practical Experience

Our type system is designed to allow the use of mutable objects and containers as part of messages in concurrent actor-based programs. Consequently, we annotated some of the most important mutable container classes in Scala's standard collection library to evaluate the type system. The annotated classes are `DoubleLinkedList`, `HashMap`, `ListBuffer`, and `ArrayBuffer`, including all classes (or traits) that these classes directly or indirectly inherit (an exception is the `Seq` trait that virtually all collection classes share). Together, these classes comprise around 2640 lines of source code, including comments and whitespace.

We were interested to see how many of the methods of each class could be annotated such that the receiver and all parameters are `@exposed` (in this case we say the method is marked as exposed). As we discussed in Section 3, such an annotation is very flexible, since it allows passing unique or transient references that have been localized or exposed, respectively, as actual parameters in a call. While this annotation allows the receiver and the parameters to refer to each other (and to objects in their cluster), it prevents the creation of references between the exposed cluster and other objects in the system, notably globally visible objects that correspond to static classes and class members in Java.

In Section 3 we already reported on the most interesting issues with annotating the `DoubleLinkedList` class and its superclass `SingleLinkedList`. All its methods could be marked as exposed. The `HashMap` class inherits from 6 additional classes (or traits), comprising 70 non-abstract methods. All of these methods could be marked as exposed. The `ListBuffer` class defines or inherits 41 non-abstract methods. Of those, 40 could be marked as exposed. The following method could not be annotated:

```
override def ++(that: Iterable[A]): Seq[A] = {
  val buf = new ArrayBuffer[A]
  this.copyToBuffer(buf)
  that.copyToBuffer(buf)
  return buf
}
```

[1] See `http://lamp.epfl.ch/~phaller/uniquerefs/`.

The problem here is the two invocations of the `copyToBuffer` method; it is defined in the `Iterable` trait that we did not annotate. Therefore, the checker must conservatively assume that the (exposed) receiver and parameters could be consumed, which is illegal.

The `ArrayBuffer` class defines or inherits 48 methods. Of those, 3 could not be marked as exposed, because of missing annotations on external classes. Additionally, the checker reported a couple of spurious warnings, because `ArrayBuffer` uses a fast array-copy method, which did not carry annotations.

In summary, our experiments show that

1. in many classes that are used in Scala's standard collection libraries, all methods can be marked as exposed, suggesting the introduction of a class-level `@exposed` annotation, and that
2. our type system enables common mutable collection classes that involve complex internal aliasing to be used as targets of tracked references, providing external uniqueness.

9 Related Work

Ownership and Uniqueness. Our approach is closely related to previous work on ownership types and uniqueness types. Clarke and Wrigstad's proposal for external uniqueness [10, 42] is based on ownership types [31, 13]; our system can be seen as an alternative way of providing external uniqueness based on capabilities instead of ownership. Our guard parameters are similar to owner parameters, but, in contrast to owner parameters, they are never explicitly written in a program; instead, all guard parameters are inferred by the type checker. A complete discussion of previous proposals for unique pointers in object-oriented languages is beyond the scope of this paper. An overview of type systems that do not provide external uniqueness can be found in [10].

UTT [30] extends Universe Types [16, 15] to provide external uniqueness for object clusters. Clusters in UTT are declared explicitly, whereas in our type system, clusters are implicit. While explicit clusters require additional declarations, they enable UTT to infer many of the `capture` and `release` statements that correspond to our `localize` and `expose` expressions. UTT does not have a qualifier that is analogous to our transient. Finally, the alias invariant of UTT does not guarantee race freedom, since read-only references into clusters remain accessible after ownership transfer.

In a concurrent setting, temporary aliasing of unique references must be done in a way that avoids race conditions. In Eiffel∗ [29], Balloon Types [2], Capabilities for Sharing [6], Pivot Uniqueness [26], and AliasJava [1], a unique reference is still usable despite the existence of temporary (borrowed) aliases. Transferring a unique reference to another concurrent process while retaining a temporary alias may lead to data races. Our approach avoids this problem by revoking the capability to use a unique reference while temporary aliases exist. Several type systems based on ownership have been proposed to verify the safety of concurrent programs. The PRFJ language of Boyapati et al. [5]

associates owners with shared-memory locks to verify correct lock acquisition before accessing objects. Unlike our approach, PRFJ does not provide external uniqueness and uses destructive reads for ownership transfer. Universe Types have also been used to verify race safety of shared-memory threads [14].

9.1 Relationship with Minimal Ownership for Active Objects

Among the work that is most closely related to ours is Minimal Ownership for Active Objects [11] (MOAO in the following); it combines a minimal notion of ownership, external uniqueness, and immutability into a system that provides race freedom for active objects [43, 8]. MOAO's immutable and safe (*arg* references in the terminology of [31]) references are orthogonal and complementary to our approach. In MOAO, abstracting from the fact whether a method consumes a unique parameter or not requires explicit owner parameters. For that purpose our system provides the `@transient` and `@exposed` qualifiers, instead. Although these qualifiers are less general than explicit owner parameters, we found that they are sufficient to guarantee external uniqueness for common collection classes. Our `expose` expression is similar to MOAO's `borrow` in that it allows accessing the representation of a unique object while ensuring external uniqueness. Unlike `borrow` in MOAO, `expose` is inferred for parameters carrying the `@transient` qualifier.

Our system enforces the invariant that a unique reference is not consumed while it is borrowed. In contrast, in MOAO this invariant is not enforced, leading to a null-pointer exception in the following example.

```
class C {
  var l: Object;
  def mutate() { l = new Object; }
  def badBorrow(p: unique::C) {
    var x: unique::C;
    borrow p as a::C pb {
      x = p--;
      pb.mutate();
    }
  }
}
```

Method `badBorrow` contains a local variable `x` of type `unique::C`; `unique` is an owner that restricts `x` to only refer to externally unique objects. The `borrow` construct introduces an alias `pb` with owner `a` for `p`, so that it can be mutated using the `mutate` method (`mutate` is owner-polymorph wrt. the owner of the receiver; for brevity, the owner parameter is omitted). However, inside the body of `borrow`, `p` is read destructively (`p--`), thereby causing a null-pointer exception at the invocation of `mutate`. To see why MOAO does not reject the above program, it is instructive to look at the typing rule for `borrow` as given in [12].

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{lval} : \texttt{unique} :: C \\ \Gamma, a : *, x : a :: C \vdash s\ \mathit{ok}\end{array}}{\Gamma \vdash \texttt{borrow}\ \mathit{lval}\ \texttt{as}\ a :: C\ x\ \{s\}\ \mathit{ok}} \quad (\text{STAT-BORROW})$$

(Exact typing, and read/write effects have been omitted since they are irrelevant for the discussion.) The environment Γ binds variables to their type and records the owners that are currently in scope (with dummy type $*$). Since Γ is immutable, it is impossible to revoke the permission to access *lval* inside the body s, and to restore the permission afterwards. In contrast, our type system uses an additional capability context that restricts access to unique references. This context may change in the course of deriving the type of an expression. In the above example, the capability to access `p` is temporarily revoked while type checking the body of `borrow`, thereby rejecting the (destructive) read of `p`.

Linear Types In functional languages, linear types [38] 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 [32] 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 [41]. Ennals et al. [17] have used quasi-linear types [25] for efficient network packet processing; however, unlike our clusters, their packets may not contain nested pointers. Our system of capabilities is inspired to a large extent by Fähndrich and DeLine's adoption and focus [19]. The second type rule for our `expose` expression corresponds to their `let-focus`. Our system extends adoption and focus with external uniqueness in an object-oriented setting. Adoption and focus builds on Alias Types [40] that allow a precise description of the shape of recursive data structures in a type system. Boyland and Retert [7] generalize adoption to model both effects and uniqueness. Their type language is more expressive, but also significantly more complex than our type language. Their realized source-level annotations include data group and effect declarations, whereas our annotation system does not require those declarations. Sing# [18] allows the linear transfer of message records that are explicitly allocated in a special exchange heap reserved for inter-process communication. Our tracked references are similar to Sing#'s tracked pointers. However, our notion of uniqueness is more flexible since it allows arbitrary internal aliases. It should be possible to apply the principles of our type system to formalize the message exchange heap of Sing#. StreamFlex [35] is a programming model for stream-based programming in Java. It allows zero-copy message passing of so-called capsules along linear filter pipelines. Capsules are instances of classes that satisfy stringent constraints: fields of capsule classes may only have primitive types or primitive array types. In contrast, our system allows linear transfer of internally-aliased objects. Kilim [36] combines a system of type qualifiers with an intra-procedural shape analysis to verify the isolation of Java-based actors. In Kilim, the shape of messages is restricted to trees, whereas our system allows messages with internal aliasing thanks to external uniqueness.

Region-Based Memory Management. Object clusters in our system are related to regions in region-based memory management [37, 39, 22, 44]. Objects inside a cluster may not refer to objects inside another cluster that may be separately consumed; consumption of clusters corresponds to deletion of regions. The main difference between regions and our object clusters are that (1) clusters are first-class values that can be stored in aliased objects, and (2) clusters do *not* have to be explicitly consumed, since they are garbage-collected.

10 Conclusion and Future Work

We have presented a type system that integrates capability-checking and external uniqueness, striving for simplicity and flexibility in both the formalization and the practical realization. We implemented our approach as a lightweight extension to Scala, without changing the language syntax. Instead of introducing uniqueness-polymorphic methods through explicit parameterization, we provide a small set of type qualifiers. Practical experience shows that these qualifiers are sufficient to add uniqueness information to common collection classes of Scala's standard library without requiring changes to their implementation.

In future work, we intend to investigate ways to automatically infer uniqueness annotations from method implementations. Furthermore, we believe that it is worthwhile to explore the application of ownership transfer to shared-memory concurrency, where objects may be transferred between shared and unshared regions of memory.

References

1. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.
2. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, pages 32–59, 1997.
3. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice-Hall, 1996.
4. Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, 1996.
5. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
6. John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27. Springer, 2001.
7. John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL*, pages 283–295. ACM, 2005.
8. Denis Caromel. Towards a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.
9. Sigmund Cherem and Radu Rugina. Uniqueness inference for compile-time object deallocation. In *ISMM*, pages 117–128. ACM, 2007.
10. Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200. Springer, 2003.

11. Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS*, pages 139–154. Springer, 2008.
12. Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. Technical Report SEN-R0803, CWI, 2008.
13. David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
14. David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, August 2007.
15. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP*, pages 28–53. Springer, 2007.
16. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
17. Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *ESOP*, pages 204–218. Springer, 2004.
18. 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.
19. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
20. Lift Web Application Framework. `http://liftweb.net/`.
21. Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.
22. Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, pages 73–84. ACM, 2004.
23. 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.
24. Twitter Kestrel. `http://github.com/robey/kestrel/`.
25. Naoki Kobayashi. Quasi-linear types. In *POPL*, pages 29–42, 1999.
26. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, 2002.
27. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
28. Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for java. In *OOPSLA*, pages 423–440. ACM, 2007.
29. Naftaly H. Minsky. Towards alias-free pointers. In *ECOOP*, pages 189–209. Springer, 1996.
30. Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478. ACM, 2007.
31. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, pages 158–185. Springer, 1998.
32. Martin Odersky. Observers for linear types. In *ESOP*, pages 390–407. Springer, 1992.
33. Martin Odersky. The Scala experiment: can we provide better language support for component systems? In *POPL*, pages 166–167. ACM, 2006.
34. Amazon SimpleDB. `http://aws.amazon.com/simpledb/`.
35. Jesper Honig Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228. ACM, 2007.

36. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP*, pages 104–128. Springer, 2008.
37. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput*, 132(2):109–176, 1997.
38. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.
39. David Walker, Karl Crary, and J. Gregory Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst*, 22(4):701–771, 2000.
40. David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *TIC*, pages 177–206. Springer, 2000.
41. David Walker and Kevin Watkins. On regions and linear types. In *ICFP*, pages 181–192, 2001.
42. Tobias Wrigstad. *Ownership-Based Alias Managemant.* PhD thesis, KTH, Sweden, May 11 2006.
43. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, pages 258–268, 1986.
44. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *RTSS*, pages 241–251. IEEE Computer Society, 2004.