Latte: Lightweight Aliasing Tracking for Java

Conrad Zimmerman[*]
Brown University
Providence, RI, USA
conrad_zimmerman@brown.edu

Catarina Gamboa[*]
Carnegie Mellon University, and
Faculdade de Ciências da Universidade de Lisboa
Pittsburgh, PA, USA
cgamboa@andrew.cmu.edu

Alcides Fonseca
Faculdade de Ciências da Universidade de Lisboa
Lisbon, Portugal
amfonseca@fc.ul.pt

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

Abstract

Many existing systems track aliasing and uniqueness, each with their own trade-off between expressiveness and developer effort.

We propose Latte, a new approach that aims to minimize both the amount of annotations and the complexity of invariants necessary for reasoning about aliasing in an object-oriented language with mutation. Our approach only requires annotations for parameters and fields, while annotations for local variables are inferred. Furthermore, it relaxes uniqueness to allow aliasing among local variables, as long as this aliasing can be precisely determined. This enables support for destructive reads without changes to the language or its run-time semantics.

Despite this simplicity, we show how this design can still be used for tracking uniqueness and aliasing in a local sequential setting, with practical applications, such as modeling a stack.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Object oriented languages**.

Keywords: aliasing, uniqueness, ownership, java

1 Introduction

From low-level languages like C to high-level programming languages like Python, the combination of mutability with aliasing has been the source of many bugs, warranting its own term (Aliasing Bug).

Reasoning about aliasing is difficult, as it usually requires a global analysis of the program and its possible traces of execution. To overcome this challenge, the community has proposed type systems that track and restrict aliasing [7, 8]. In this very large design space, there are lines of work more focused on uniqueness, ownership and permissions. However, these proposals add complexity relative to ordinary type systems, and in some cases require developers to understand and reason about quite complex concepts. For example, ownership has been frequently mentioned as one of

the hardest concepts to learn in the Rust programming language [19, 23].

Thus, we propose a type system for uniqueness and aliasing that aims to be more usable and impose low overhead on developers. Moreover, we intend to keep our approach as simple as possible so that it can support the development of more complex type systems (like Liquid Types [24]) that require reasoning about uniqueness and aliasing.

In particular, we propose a type system for a subset of the Java language that tracks uniqueness and heap aliasing with low annotation effort and no necessary runtime changes. To handle uniqueness, we provide the developer with the simple invariant that *no two references in the heap point to the same unique object.*

We require few annotations, specifically two (unique and shared) for object fields and return types, and three (adding owned) for method parameters, and we infer the remaining information for local variables.

Moreover, we aim to maintain Java's runtime semantics unaltered, and therefore do not build in destructive reads; the programmer can, however, get a similar effect with an explicit assignment to `null`.

In the remainder of the paper, we present an overview of previous work that is closely related to ours (Section 2), followed by our approach with the presentation of an example and the system's grammar and typing rules (Section 3). At the end, we discuss some of the system's limitations and future directions for this work (Section 4).

2 Related Work

There are different approaches for managing aliasing in programming languages. One popular line of work focuses on ownership types [10], a type system that restricts access to objects according to their owners. There have been multiple proposals for ownership types with different flavors [9]. Unfortunately, classic ownership types alone do not track aliasing within ownership boundaries, making it difficult for verification tools to precisely track the effect of assignments.

---

[*]Both authors contributed equally to this research.

Tracking uniqueness can provide strong guarantees about aliasing, which is useful for verification as well as safe manual memory management, e.g. as implemented in Rust [21]. However, more powerful systems such as Rust's (which is called "ownership," though it provides uniqueness in the sense described in the research literature) are known to be complex and difficult for developers to understand [11].

Uniqueness types [12] allow uniqueness properties to be specified as part of a data type. However, this approach is tailored to functional programming languages and requires a significantly different type system, thus its applicability to Java is limited.

Other type systems for uniqueness focus on the use cases of concurrency and message passing. For example, Haller and Odersky [16] use capabilities [4] to add uniqueness and borrowing to a Java-like languages with a focus on message passing in concurrent object-oriented programming. They use a concept of *separate uniqueness*, where distinct variables do not share a common reachable object. Thus uniqueness is used to enforce separation, which is desirable for concurrent message-passing systems.

More recently, Milano et al. [22] presented a new language and type system for safe concurrency by statically ensuring that different threads cannot access the same heap regions. Their proposal focused on reducing the annotation burden and eliminating the need for unnatural rewrites required by more restrictive programming models. Also aiming for a minimal set of annotations, LaCasa [15] adds uniqueness to the Scala language using object capabilities. However, this approach requires classes to adhere to the object-capability discipline, and their empirical evaluation showed that most classes from the standard library do not follow these rules.

There are other approaches that focus on modeling different aspects of aliasing. Reachability types [2] uses *reachability sets* to reason about ownership, and tracks reachable values using type qualifiers. This work layers uniqueness, nested mutable state and other concepts over the tracking of reachability sets. Unique accesses are enabled by killing all other access paths to a reference.

Castegren and Wrigstad [6] combined many of the previously-mentioned concepts in their κ language. This language uses reference capabilities to ensure separation, and combines techniques from ownership types, linear types [25], and regions [14] in a concurrent and parallel object-oriented setting.

The systems described above were not implemented for Java, however, and it is unclear how to do so, as they rely on language features that Java does not have, such as capabilities in Scala or a primitive swap operation.

AliasJava [1] does extend Java with type annotations in Java that specify data sharing relationships. The type system includes four annotations: *unique, owned, lent* and *shared*, and reduces the annotation burden by inferring annotations.

Our approach is similar in spirit, but achieves greater simplicity by doing without ownership and ownership parameters, while allowing more local alising within a method.

Many early systems for uniqueness used destructive reads, but these often negatively impact program complexity by reducing the ability to query information contained in objects [5]. Therefore, alias burying [3] aims to define a uniqueness system for Java-like languages without using destructive reads by relying on the idea that aliases that will not be used again can be buried. However, as noted by Boyland and Retert [5], the analysis described in the initial work [3] exposes implementation details, such as the fields read by a method, which breaks encapsulation and modularity.

In summary, the prior work has one of three limitations: reliance on language or type system features not present in Java, modularity or coding pattern issues, or a larger and more complex set of abstractions for programmers to understand compared to our goals. All of these design choices raise the adoption cost for developers. In our approach, Latte, we try to address these difficulties by creating a lightweight uniqueness system with few annotations. Latte, which we present in the following section, requires no changes to the language semantics, and allows many common code patterns while precisely tracking aliasing.

3 Approach

As we described previously, our design aims to impose minimal restrictions while enforcing unique references (in the heap) and tracking aliasing (in the stack). While our system is not as expressive as others in previous work, its main advantages are an easily-understood programming model and low annotation complexity.

In particular, our design only requires annotations on fields and parameters (with only two and three possible choices, respectively). This burden can be further reduced by choosing sensible defaults. Local variables do not need annotations, as the aliasing between local variables and field values is inferred, which reduces the barrier to adopting this system.

In this section, we first give a high-level description of our approach, and then use an example to build intuition about our model. We then formally define the typing rules on top of a Featherweight Java [18]-inspired core language and explain how these rules result in the intended behavior. Finally, we demonstrate the expressive power of our system with a more complex example.

3.1 Description

First, we restrict our definition of uniqueness to only consider reachable values on the heap, thus unique values may be stored in at most one reachable heap location, and aliased in the local environment. However, these *dynamic aliases*

[17] may only be used as long as such aliasing can be precisely inferred. Our treatment of dynamic aliases and unreachable heap locations is similar to that of alias burying [5].

In Latte, the annotation unique is used to identify unique values (as defined above). owned identifies borrowed values, since it is only used on method parameters whose value will be owned by some other context when entering the method body. shared identifies values that may or may not be unique.

We aim to use our analysis in an automated verifier such as LiquidJava [13] to reason about mutation of unique (and borrowed) values. This requires precisely identifying all values that may be affected by a particular mutation. Our approach does this while permitting dynamic aliases, by inferring annotations of the form alias(p) or $\perp$ during type checking. These special annotations are only inferred; they are never written by the developer.

For each local variable x, our typing environment Δ contains a class C and an annotation α which describes the uniqueness of x at that point. The formal definition of Δ is given later in Figure 3.

3.2 Example

We illustrate the main features of our approach by implementing push and pop operations for a stack storing unique values. References to objects pushed onto the stack may not be stored on the heap anywhere else. This invariant could be used by an automated verifier to show that values pushed onto the stack will not be mutated until they are returned by pop. The code is shown in Figure 1.

First, we demonstrate how the push method is validated. At each step, we have listed (in Figure 1) the current typing context Δ to illustrate the verification process.

The typing environment at the beginning of the method body contains the parameters and their types (line 17). Because the two variables r and n are declared, but not yet initialized at line 18, they are annotated with $\perp$ to mark them inaccessible. r is aliased with this.root at line 19, and this aliasing information is added to the environment by annotating r with alias(this.root). Next, this.root is assigned, which *isolates* it in the typing environment (a process that we describe in §3.4.5). This invalidates all aliases to it, which allows the previously-aliased variable r to become unique and thus claim ownership.

Note that lines 19-20 are equivalent to a destructive read. However, we do not need to change the language semantics or introduce new language constructs. Also, the need for this destructive read is easily understood: we want to store this.root in a different place in the heap, thus we need to remove its current value from the heap. Otherwise, our uniqueness invariant would be violated since the value at this.root (which is declared unique) would be stored in multiple reachable places in the heap.

```
1  class Node {
2    unique Object value;
3    unique Node next;
4
5    Node(unique Object value, unique Node next)
6    { this.value = value;
7      this.next = next; }
8  }
9
10 class Stack {
11   unique Node root;
12
13   Stack(unique Node root)
14   { this.root = root; }
15
16   void push(owned Stack this,
17            unique Object value) {
        Δ = this : owned Stack, value : unique Object
18     Node r; Node n;
        Δ = ···, r : ⊥ Node, n : ⊥ Node
19     r = this.root;
        Δ = ···, r : alias(this.root) Node
20     this.root = null;
        Δ = ···, r : unique Node
21     n = new Node(value, r);
        Δ = this : owned Stack, value : ⊥ Object,
            r : ⊥ Node, n : unique Node
22     this.root = n;
        Δ = ···, n : alias(this.root) Node
23   }
24
25   unique Object pop(owned Stack this) {
26     Object value;
27     if (this.root == null) {
28       value = null;
29     } else {
30       value = this.root.value;
31       Node next;
32       next = this.root.next;
33       this.root = next;
34     }
35     return value;
36   }
37 }
```

Figure 1. Example: a stack for unique references

Continuing with our example, we initialize a new Node object at line 21. The constructor of Node has the signature (unique Object, unique Node), which states that it *consumes* both arguments. This marks value and r as inaccessible ($\perp$) in the calling context. We encapsulate constructor bodies, thus we do not know what value or r may be aliased with after they are passed to the constructor. Since

$$P ::= \overline{CL}$$

$$CL ::= \texttt{class } C \texttt{ extends } C \{ \overline{F} \; K \; \overline{M} \}$$

$$\alpha_f ::= \texttt{unique} \mid \texttt{shared}$$

$$\alpha_p ::= \texttt{unique} \mid \texttt{shared} \mid \texttt{owned}$$

$$F ::= \alpha_f \; C \; f;$$

$$K ::= C(\overline{\alpha_f \; C \; g}, \overline{\alpha_f \; C \; f})\{ \texttt{super}(\overline{g}); \overline{\texttt{this}.f = f;} \}$$

$$\tau ::= \alpha_f \; C \mid \texttt{void}$$

$$M ::= \texttt{void } m(\alpha_p \; C \texttt{ this}, \; \overline{\alpha_p \; C \; x}) \{ \overline{s} \}$$

$$\quad \mid \alpha_f \; C \; m(\alpha_p \; C \texttt{ this}, \; \overline{\alpha_p \; C \; x}) \{ \overline{s} \texttt{ return } e; \}$$

$$p ::= x \mid p.f$$

$$e ::= \texttt{null} \mid p$$

$$s ::= C \; x; \mid x = e; \mid x.f = e;$$

$$\quad \mid x = \texttt{new } C(\overline{e}); \mid x = x.m(\overline{e}); \mid x.m(\overline{e});$$

$$\quad \mid \texttt{if } (e == e) \; s \texttt{ else } s \mid \{ \overline{s} \}$$

Figure 2. Grammar extended from Featherweight Java

we cannot track this aliasing, any usage of these values is disallowed after they are passed to the constructor.

Finally, we assign n to this.root at line 22. Since our typing context only stores annotations for local variables, and not for fields, we update the annotation for the local variable n, which is on the RHS of the assignment. The annotation alias(this.root) simply denotes that its target this.root contains the same value as the annotated variable n, thus it does not matter which side of an alias is annotated. After this line, Δ indicates that n is aliased with this.root, thus we have precisely determined all local aliases to the unique value this.root, and ensured that this.root is stored at only one location on the heap.

This first method gives an overview of our approach; the second method (pop, at line 32) will be presented in Section 3.4.7 after the grammar and typing rules are introduced.

3.3 Grammar

For our grammar, presented in Figure 2, we extended Featherweight Java [18] with statements, including field and variable assignments as in Java, to better approximate the Java language in terms of mutability (a key concern in this paper). To model our particular system, we added the unique, shared, and owned annotations. All fields (F) must be annotated with either unique or shared (α_f), while method parameters (in M) must be annotated with one of the three annotations (α_p). Note that variable declarations are not annotated (first production of s).

Our owned annotation is often called *lent* or *borrowed* in other systems. Our choice reflects the state of the value

3.4 Typing rules

Our typing rules use a local type environment Δ. This environment maps variables to an annotated class ($\alpha \; C$), where the annotation specifies the current aliasing or uniqueness information. The form of Δ is given in Figure 3.

A shared annotation denotes that the variable can be accessed by outside objects – untracked aliases may exist. owned denotes that the value of the variable is borrowed; specifically, its value is unique in the current context and no new aliases may be added to the heap. unique denotes ownership – the value is only stored at this location (modulo precisely-tracked dynamic aliases). A local variable annotated unique may be converted to a shared, based on its usage. $\perp$ denotes that the value is inaccessible.

$$\alpha_e ::= \texttt{owned} \mid \texttt{shared} \mid \texttt{unique} \mid \texttt{unique}(p.f)$$

$$\alpha ::= \texttt{owned} \mid \texttt{shared} \mid \texttt{unique} \mid \texttt{alias}(p) \mid \perp$$

$$\Delta ::= \cdot \mid x : \alpha \; C, \; \Delta$$

Figure 3. Typing environment and annotations used in typing rules

3.4.1 Aliasing. Aliasing between variables and fields is tracked by entries of the form $x : \texttt{alias}(p)$ in Δ. This denotes that the x stores the same value as the *path* (a variable or some field access) p. Two paths are aliased iff they reference the same object, thus aliasing is an equivalence relation. This is encoded by the judgment $\Delta \vdash p_1 \equiv p_2$, which denotes that Δ indicates that the path p_1 is aliased with p_2. Formal rules are given in Appendix A.4.

Given an environment Δ, we define its *alias graph* to be a (undirected) graph whose nodes are syntactic paths (p as defined in Figure 2), and distinct paths p_1 and p_2 are connected iff $\Delta \vdash p_1 \equiv p_2$. Each component of this graph may contain at most one path annotated with owned, unique, or shared.

Intuitively, the alias graphs for each program point (which identify allowable aliasing), along with validation of uniqueness invariants (which ensures that no other aliases of unique values exists), is the primary product of our analysis. This output can then be used to automatically verify the effects of mutation or, more generally, separation invariants.

3.4.2 Side note: concurrency. Since we are tracking aliases across multiple statements, and our alias annotations may point to mutable heap locations, it may seem challenging to handle concurrency. However, we only claim to precisely track aliases of unique or borrowed values, i.e. expressions for which $\Delta \vdash e : \texttt{owned } C \dashv \Delta'$ holds for some C and Δ'.

Intuitively, if this holds for a variable x, and x is aliased to $y.f$, then $y.f$ is also unique, which requires y to be unique. In other words, either the current context or some calling context is the sole owner of y, and thus of the heap location $y.f$. (This reasoning may be extended for $y.f.g$, etc.) Therefore we can determine all mutations that would affect this alias relation. In other words, assuming soundness of our approach for sequential programs, it should remain sound for concurrent programs, as long as unique values accessible to the spawned thread are consumed after a fork operation.

3.4.3 Reachable aliasing. $\Delta \vdash p_1 \cong p_2$ denotes that a value reachable from p_1 (for example, $p_1.f$) may be aliased with a value reachable from p_2 (for example, $p_2.f.g$). Formal rules are given in appendix A.5. If $\Delta \nvdash p_1 \cong p_2$ then p_1 and p_2 are *separately unique*, as defined in [16].

3.4.4 Expression typing. $\Delta \vdash e : \alpha_e\ C \dashv \Delta'$ denotes that e may be used as a value with class C and ownership annotation α_e, provided that all future typing uses the Δ' typing environment. Formal rules are given in Figure 4.

$\Delta \vdash e : \text{unique}(p.f) \dashv \Delta'$ denotes that e refers to a unique value (as defined in §3.1), and e is aliased with $p.f$ in Δ'. This is used to validate assignments to unique fields, since the assignee is a field whose annotation is not stored in Δ. Thus aliasing information is tracked by annotating the assignment value, instead of annotating the assignee.

For a variable x annotated with $\text{alias}(p)$, $\Delta \vdash x : \alpha_e\ C \dashv \Delta'$ holds if and only if $\Delta \vdash p : \alpha_e\ C \dashv \Delta'$ – in other words, aliased variables may be used exactly how the path they alias may be used.

When borrowing a unique field value (i.e. passing the value to a parameter annotated as owned), the object reference must also be unique. For example, if we have variables $x : \text{shared } C$ and $y : \text{shared } C$, and C contains a unique field f, we cannot borrow $x.f$ because we do not know whether the same heap location is already borrowed through $y.f$. Thus one can introduce aliases to a unique field of a shared value, but those values can only be used after a destructive read or some equivalent operation.

Finally, a value of a subtype C may be used as a value of the supertype type D with the same annotation.

3.4.5 Isolation. $\Delta * p \dashv \Delta'$ denotes that p is *isolated* from Δ' – all references to p contained in Δ are removed in Δ'. Δ' represents a state where p is assigned a new value, thus all aliases to p in Δ should be removed in Δ'. Moreover, if p represented a unique value and a variable x was aliased to p, x contains a unique value after p is overwritten. Thus destructive reads are accomplished by first introducing an alias to p, and then overwriting p with a different value, such as `null`.

Given an environment Δ, we define its *reference graph* to be a (directed) graph whose nodes are syntactic paths.

$$\frac{\text{E-Owned}}{\Delta(x) = \text{owned } C}{\Delta \vdash x : \text{owned } C \dashv \Delta}$$

$$\frac{\text{E-Shared}}{\Delta(x) = \text{shared } C}{\Delta \vdash x : \text{shared } C \dashv \Delta}$$

$$\frac{\text{E-UniqueOwned}}{\Delta(x) = \text{unique } C}{\Delta \vdash x : \text{owned } C \dashv \Delta}$$

$$\frac{\text{E-UniqueShared}}{\Delta(x) = \text{unique } C}{\Delta \vdash x : \text{shared } C \dashv \Delta[x \mapsto \text{shared } C]}$$

$$\frac{\text{E-UniqueUnique}}{\Delta(x) = \text{unique } C}{\Delta \vdash x : \text{unique } C \dashv \Delta[x \mapsto \bot\ C]}$$

$$\frac{\text{E-UniqueAlias}}{\Delta(x) = \text{unique } C}{\Delta \vdash x : \text{unique}(p.f)\ C \dashv \Delta[x \mapsto \text{alias}(p.f)\ C]}$$

$$\frac{\text{E-Alias}}{\Delta(x) = \text{alias}(p)\ C \quad \Delta \vdash p : \alpha_e\ C \dashv \Delta'}{\Delta \vdash x : \alpha_e\ C \dashv \Delta'}$$

$$\frac{\text{E-FieldOwned}}{\Delta \vdash p : \text{owned } C \dashv \Delta \quad \text{ftype}(C)(f) = \text{unique } C'}{\Delta \vdash p.f : \text{owned } C' \dashv \Delta}$$

$$\frac{\text{E-FieldShared}}{\Delta \vdash p : C \quad \text{ftype}(C)(f) = \text{shared } C'}{\Delta \vdash p.f : \text{shared } C' \dashv \Delta}$$

$$\frac{\text{E-Sub}}{\Delta \vdash e : \alpha_e\ C \dashv \Delta' \quad C <: D}{\Delta \vdash e : \alpha_e\ D \dashv \Delta'}$$

$$\frac{\text{E-Null}}{\Delta \vdash \texttt{null} : \alpha_e\ C \dashv \Delta}$$

Figure 4. Expression usage typing rules

An edge $x \rightarrow p$ exists iff Δ contains an annotation $x : \text{alias}(p.\cdots)$.

Intuitively, the origin of an edge in the alias graph identifies a variable whose annotation requires updating when its target is mutated. Unlike the alias graph defined in §3.4.1, this graph is not symmetric or transitive.

For example, in the `pop` method, after line 30 we have the annotation $\texttt{value} : \text{alias}(\texttt{this.root.value})$. Thus the reference graph contains the edge $\texttt{value} \rightarrow \texttt{this.root}$. If the value of `this.root` is changed, we must determine a new annotation for `value`.

Formal rules are given in appendix A.7. The rules deal with three main cases:

1. No node in either the reference graph or the alias graph p is connected to p. Δ is unchanged, except to remove p from $\text{dom}(\Delta)$. (See the I-Remove* rules.) This case is applied during validation of line 30 in Figure 1 to remove the initial annotation $\texttt{value} : \bot$.

2. For some variable x (distinct from p), x is aliased with p. In this case, all paths rooted in p may be replaced by paths rooted in x. (See the I-Replace* rules.)

3. p is disconnected in the alias graph, but p is the target of an edge in the reference graph. In this case, we can

isolate the subfield that induces this edge (such as $p.f$) before isolating p. (See the I-Elim* rules.)

This case is applied during validation of line 33 in Figure 1 when isolating `this.root`. The annotation of `value` is updated from alias(`this.root.value`) to unique.

3.4.6 Framing. When a reference is passed to a method, any field of the referenced object can be modified. The *frame* of a method call contains all such fields. Any aliases to fields in the method's frame must be invalidated after the method call.

$\Delta \star \overline{e} \dashv \Delta'$ denotes that all variables that are connected (in the reference graph of Δ) to any path in the list $\overline{e}$ are marked in Δ' as either shared, in cases where alias tracking is unnecessary, or $\bot$, in cases where aliases must be tracked. Formal rules are given in appendix A.8.

3.4.7 Statement typing. Statements are typed by the judgment $\Delta \vdash s \dashv \Delta'$. Selected rules are shown in Figure 5. See appendix A.10 for a complete listing.

An assignment such as $x = p$ adds an edge $x \rightarrow p$ to the alias graph. Note that as a special case, $x =$ `null` adds the annotation $x :$ unique (see definition of alias in Appendix A.3).

When f is a unique field and e is a path, $x.f = e$ adds an edge $e \rightarrow x.f$ to the alias graph by using the annotation unique$(x.f)$ when validating e. Note that $x.f$ is isolated *before* typing e, which allows patterns such as lines 32-33 in Figure 1, where `this.root` is updated to point to `this.root.next` without a destructive read. Since the field `this.root` is unique, the variable aliased to `this.root.next` may overwrite `this.root` without violating uniqueness.

Method calls are validated by validating each argument according to the corresponding parameter definition. Note that after typing an argument as unique, all paths rooted in the argument are inaccessible. Therefore reachable aliasing cannot exist between parameters annotated with unique. However, aliasing is not prohibited simply by typing arguments passed to parameters annotated owned, since borrowed values are not consumed. Therefore we explicitly prohibit reachable aliasing between owned parameters. We also invalidate aliases to fields in the frame of the method call, as discussed in §3.4.6.

3.4.8 Unification. When an `if` statement is validated, each branch must be validated, which produces two separate typing environments, Δ_1 and Δ_2. These environments are *unified* by finding an annotation that can be used in both environments for each variable in the outer environment Δ. Formal rules are given in appendix A.9.

If some variable x has the same annotation in both branches, then this common annotation may be used. If x refers to a shared value in both Δ_1 and Δ_2, then it may be annotated

$$\frac{\text{S-AssignVar} \quad \Delta(x) = \alpha\ C}{\Delta \vdash e : C \qquad \Delta \nvdash x \equiv e \qquad \Delta * x \dashv \Delta'}{\Delta \vdash x = e\texttt{;} \dashv x : \text{alias}(e)\ C, \Delta'}$$

$$\text{S-Call}$$
$$\alpha_0\ C_0, \cdots, \alpha_n\ C_n \rightarrow \alpha\ C = \text{mtype}(C_0)(m)$$
$$\Delta \vdash e_0 : \alpha_0\ C_0, \cdots, e_n : \alpha_n\ C_n \dashv \Delta' \qquad \Delta'(x) = \alpha'\ B$$
$$B <: C \qquad \Delta' \star e_0, \cdots, e_n \dashv \Delta'' \qquad \Delta'' * x \dashv \Delta'''$$
$$\forall 0 \le i < j \le n : \left[\alpha_i = \text{owned} \implies \Delta' \nvdash e_i \cong e_j\right]$$
$$\overline{\Delta \vdash x = e_0.m(e_1, \cdots, e_n)\texttt{;} \dashv x : \alpha\ B, \Delta'''}$$

$$\text{S-AssignUnique}$$
$$\Delta \vdash x : C$$
$$\Delta * x.f \dashv \Delta' \qquad \text{ftype}(C)(f) = \text{unique}\ C'$$
$$\Delta \nvdash x.f \cong e \qquad \Delta' \vdash e : \text{unique}(x.f)\ C' \dashv \Delta''$$
$$\overline{\Delta \vdash x.f = e\texttt{;} \dashv \Delta''}$$

$$\text{S-Conditional}$$
$$\Delta \vdash e_1 : \texttt{Object} \qquad \Delta \vdash e_2 : \texttt{Object}$$
$$\Delta \vdash s_1 \dashv \Delta_1 \qquad \Delta \vdash s_2 \dashv \Delta_2$$
$$\overline{\Delta \vdash \texttt{if}\ (e_1 \texttt{==} e_2)\ s_1\ \texttt{else}\ s_2 \dashv \text{unify}(\Delta; \Delta_1; \Delta_2)}$$

Figure 5. Statement typing rules

with shared. If a variable is not referenced in starting Δ, the variable may be isolated from Δ_1 and Δ_2. This removes all aliasing to variables defined within a particular branch.

If x is annotated alias(p_1) in Δ_1 and alias(p_2) in Δ_2, and there is a path p where $\Delta_1 \vdash p_1 \equiv p$ and $\Delta_2 \vdash p_2 \equiv p$, then x may be annotated with alias(p) after unification. Otherwise, if none of these cases apply, the variable is annotated with $\bot$, thus making its value inaccessible.

Unification occurs in the `pop` method in Figure 1 prior to the `return` statement. In the first branch, `value` is unique since it is `null`. In the second branch, `value` is first aliased with `this.root.value`, but then becomes unique after line 33. Thus `value` is unique in both branches, and we can use `value` in the `return` statement, which requires a unique value.

Note that more specific unification procedures could be developed. For example, we could implement conditional aliasing annotations, or validate every possible execution path independently (and thus eliminate unification entirely). However, we feel that this conservative unification algorithm is more usable since it allows clear error messages when attempting to access variables that are inaccessible due to unification.

3.4.9 Program typing. A valid program is defined in appendix A.11, along with rules for typing method and class declarations.

```
1  unique Object dequeue(owned Stack this) {
2    Node r = this.root;
3    Object value;
4    if (r == null || r.next == null) {
5      value = this.pop();
6    } else {
7      value = dequeueHelper(r);
8    }
9    return value;
10 }
11
12 unique Object dequeueHelper(owned Stack this,
13                            owned Node n) {
14   Object value;
15   if (n.next.next == null) {
16     value = n.next.value;
17     n.next = null;
18   } else {
19     value = dequeueHelper(n.next);
20   }
21   return value;
22 }
```

Figure 6. Example: a `dequeue` method for the `Stack` class from Figure 1

3.5 Extended example

The `dequeue` method in Figure 6 allows the stack to be used as a FIFO queue. The recursive traversal of the linked list is handled by the `dequeueHelper` method. Note that the entire list is traversed, and the tail modified, using only a single destructive read.

This is enabled by borrowing the unique value `this.root`, and in turn each `next` node. The value of an owned parameter is guaranteed to be unique, but its value may not be consumed or placed on the heap. Thus `dequeueHelper` guarantees that no additional aliases to `n` will be introduced. However, the contents of an owned value may be modified, which allows the tail `Node` to be removed.

Also note that the unique value `n.next.value` is read without an explicit destructive read at line 16. Instead, it is known to be unique since its container (`n.next`) is isolated at line 17.

In a survey of related work, Milano et al. [22] found that ownership systems often require explicit destructive reads at each step when traversing and modifying a linked list as in this example. However, our isolation technique, combined with local aliased values and borrowing, eliminates this requirement while allowing common code patterns and requiring few annotations.

4 Future Work

While the design decisions were guided by making this system usable by developers, we would need to implement the system and evaluate it in larger examples with users. A comparison of the effort in different examples with the alternatives in the related work would also be interesting, to confirm whether our invariant helps developers to annotate their code. Additionally, our current approach includes only a core set of Java features that we would like to extend to include `while` loops.

One of the motivations of this work was to introduce enough information to reason about mutability to support Liquid Types in a mutable context. Flux [20] took the first steps in this area, by using Rust's ownership type in combination with a Liquid Type System. Our proposal targets the Java language instead, and serves as the basis to extend LiquidJava [13] to better model aliasing and uniqueness combined with refinements. Because Liquid Types supports a logic-based version of symbolic execution, the information from refinements could be used in the unification to have a more precise alias tracking, instead of the conservative invalidation we took instead.

5 Conclusion

We have described Latte, a simple type system for uniqueness and aliasing for Java, which prioritizes usability and low development overhead. Our vision is that more complex type systems may utilize the uniqueness and aliasing information determined by Latte. Latte enforces (and requires consideration of) simple invariants of values on the heap, imposes a low annotation burden, and requires no changes to existing Java semantics.

Our simple uniqueness invariant indicates that a unique object is stored at most once on the heap. In addition, all usable references to a unique object from the local environment are precisely inferred. The developer only needs to annotate field declarations and the parameters and return types of method declarations, using one of unique, owned or shared.

While it may lack the expressive power of related approaches, we hope that Latte provides a lower barrier to entry for existing Java developers, thus enhancing the appeal of automated verification tools built on Latte. Further evaluation of its usability, along with the development of such verification tools, is required to validate this goal.

6 Acknowledgments

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias annotations for program understanding. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 311–330. https://doi.org/10.1145/582419.582448

[2] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. https://doi.org/10.1145/3485516

[3] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exp.* 31, 6 (2001), 533–553. https://doi.org/10.1002/spe.370

[4] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 2–27. https://doi.org/10.1007/3-540-45337-7_2

[5] John Tang Boyland and William Retert. 2005. Connecting effects and uniqueness with adoption. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 283–295. https://doi.org/10.1145/1040305.1040329

[6] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[7] Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). 2013. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification.* Lecture Notes in Computer Science, Vol. 7850. Springer. https://doi.org/10.1007/978-3-642-36946-9

[8] Dave Clarke, James Noble, and Tobias Wrigstad. 2013. Beyond the Geneva Convention on the Treatment of Object Aliasing. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 1–6. https://doi.org/10.1007/978-3-642-36946-9_1

[9] Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2743)*, Luca Cardelli (Ed.). Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

[10] David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. https://doi.org/10.1145/286936.286947

[11] Will Crichton. 2020. The Usability of Ownership. *HATRA 2020* abs/2011.06171 (2020). arXiv:2011.06171 https://arxiv.org/abs/2011.06171

[12] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2007. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5083)*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer, 201–218. https://doi.org/10.1007/978-3-540-85373-2_12

[13] Catarina Gamboa, Paulo Canelas, Christopher Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1520–1532. https://doi.org/10.1109/ICSE48619.2023.00132

[14] Aaron Greenhouse and John Boyland. 1999. An Object-Oriented Effects System. In *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1628)*, Rachid Guerraoui (Ed.). Springer, 205–229. https://doi.org/10.1007/3-540-48743-3_10

[15] Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 272–291. https://doi.org/10.1145/2983990.2984042

[16] Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

[17] John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1991, Phoenix, Arizona, USA, October 6-11, 1991*, Andreas Paepcke (Ed.). ACM, 271–285. https://doi.org/10.1145/117954.117975

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[19] Steve Klabnik and Carol Nichols. 2023. What Is Ownership? https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[20] Nicolás Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2022. Flux: Liquid Types for Rust. *Proceedings of the ACM on Programming Languages* 7 (2022), 1533 – 1557.

[21] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

[22] Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 458–473. https://doi.org/10.1145/3519939.3523443

[23] Nicholas Rempel. 2023. Understanding rust's ownership model. https://www.learnrust.blog/p/understanding-rusts-ownership-model#:~:text=In%20R

[24] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[25] Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, 561.

A Typing rules

A.1 Reference typing

$\Delta \vdash p : C$ denotes that the path p is accessible and contains either `null` or an reference to an instance of class C.

T-Var
$$\frac{\alpha \neq \bot}{\Delta, x : \alpha\, C \vdash x : C}$$

T-Field
$$\frac{\Delta \vdash p : C \qquad \text{ftype}(C)(f) = \alpha\, C'}{\Delta \vdash p.f : C'}$$

A.2 Subtyping

$C <: D$ denotes that C extends a parent class D. It is a reflexive and transitive relation.

C-Id
$$\frac{}{C <: C}$$

C-Transitive
$$\frac{C <: D \qquad D <: E}{C <: E}$$

C-Def
$$\frac{\texttt{class } C \texttt{ extends } D\ \{\ \overline{F}\ K\ \overline{M}\ \}}{C <: D}$$

A.3 Auxiliary functions

fields(C) denotes the list of field declarations in C. ftype(C)(f) denotes the annotation and class specified in the declaration of field f in class C. mtype(C)(m) denotes the signature of method m in class C. alias(e) is a helper function which denotes the annotation alias(e) when e is a path, and unique when e is `null`.

F-Obj
$$\frac{}{\text{fields}(\texttt{Object}) = \cdot}$$

F-Decl
$$\frac{\texttt{class } C \texttt{ extends } D\ \{\ \overline{F}\ K\ \overline{M}\ \} \qquad \overline{F} = \overline{\alpha_f\, C\, f;} \qquad \text{fields}(D) = \overline{\alpha_{f'}\, D\, g}}{\text{fields}(C) = \overline{\alpha_{f'}\, D\, g}\ \overline{\alpha_f\, C\, f}}$$

F-Type
$$\frac{\alpha_f\, B\, f \in \text{fields}(C)}{\text{ftype}(C)(f) = \alpha_f\, B}$$

M-Class
$$\frac{\texttt{class } C \texttt{ extends } D\ \{\ \overline{F}\ K\ \overline{M}\ \} \qquad \tau\, m(\alpha_p\, \texttt{this}, \overline{\alpha_p\, B\, x})\ \{\ \overline{s}\ \} \in \overline{M}}{\text{mtype}(C)(m) = \overline{\alpha_p\, B} \to \tau}$$

M-Super
$$\frac{\texttt{class } C \texttt{ extends } D\ \{\ \overline{F}\ K\ \overline{M}\ \} \qquad \text{mtype}(D)(m) = \overline{\alpha_p\, B} \to \tau}{\text{mtype}(C)(m) = \overline{\alpha_p\, B} \to \tau}$$

$$\text{alias}(e) := \begin{cases} \text{unique} & \text{if } e = \texttt{null} \\ \text{alias}(e) & \text{otherwise} \end{cases}$$

A.4 Aliasing

$\Delta \vdash p \equiv p'$ denotes that a path p refers to the same value as p'.

A-Refl
$$\frac{}{\Delta \vdash p \equiv p}$$

A-Var
$$\frac{\Delta(x) = \text{alias}(p)\, C}{\Delta \vdash x \equiv p}$$

A-Cong
$$\frac{\Delta \vdash p_1 \equiv p_2}{\Delta \vdash p_1.f \equiv p_2.f}$$

A-Symm
$$\frac{\Delta \vdash p_1 \equiv p_2}{\Delta \vdash p_2 \equiv p_1}$$

A-Trans
$$\frac{\Delta \vdash p_1 \equiv p_2 \qquad \Delta \vdash p_2 \equiv p_3}{\Delta \vdash p_1 \equiv p_3}$$

A.5 Reachable aliasing

$\Delta \vdash p \cong p'$ denotes that a value reachable from p may be aliased with a value reachable from p', given the aliasing specified in Δ.

RA-Refl
$$\frac{}{\Delta \vdash p \cong p}$$

RA-Symm
$$\frac{\Delta \vdash p_1 \cong p_2}{\Delta \vdash p_2 \cong p_1}$$

RA-Trans
$$\frac{\Delta \vdash p_1 \cong p_2 \qquad \Delta \vdash p_2 \cong p_3}{\Delta \vdash p_1 \cong p_3}$$

RA-Field
$$\frac{\Delta \vdash p_1 \cong p_2}{\Delta \vdash p_1.f \cong p_2}$$

RA-Var
$$\frac{\Delta(x) = \text{alias}(p)\, C}{\Delta \vdash x \cong p}$$

A.6 Replacement

$p[p''/p']$ denotes the path p with the path p' replaced by p''. For example, $x.f[y.g/x] = y.g.f$.

$$x[p''/p'] := \begin{cases} p'' & \text{if } x = p' \\ x & \text{otherwise} \end{cases}$$

$$p.f[p''/p'] := \begin{cases} p'' & \text{if } p.f = p' \\ p[p''/p'].f & \text{otherwise} \end{cases}$$

$\Delta[p''/p']$ denotes the environment Δ with all aliases that reference the path p' updated to instead reference the path p''.

$$\cdot[p''/p'] := \cdot$$

$$(x : \alpha\, C, \Delta)[p''/p'] := \begin{cases} x : \text{alias}(p[p''/p'])\, C, \Delta[p''/p'] \\ \quad \text{if } \alpha = \text{alias}(p) \text{ for some } p \\ x : \alpha\, C, \Delta'[p''/p'] \\ \quad \text{otherwise} \end{cases}$$

A.7 Isolation

$\Delta \vdash x \rightsquigarrow p$ denotes that a variable x is aliased with a path rooted in p, i.e. p, $p.f$, $p.f.g$, etc. In other words, an aliased edge exists between x and p or a reference edge points to p.

VA-Intro
$$\frac{\Delta(x) = \text{alias}(p') \qquad \Delta \vdash p \equiv p'}{\Delta \vdash x \rightsquigarrow p}$$

VA-FieldElim
$$\frac{\Delta \vdash x \rightsquigarrow p.f}{\Delta \vdash x \rightsquigarrow p}$$

$\Delta * p \dashv \Delta'$ denotes that a path p is isolated from Δ', as described in Section 3.4.5.

$$\frac{\nexists x \in \text{dom}(\Delta) : \Delta \vdash x \rightsquigarrow p.f}{\Delta * p.f \dashv \Delta}$$
I-RemoveField

$$\frac{\begin{array}{c}\Delta = x : \text{alias}(p'.f)\, C, \Delta' \\ \Delta \vdash p \equiv p' \qquad \Delta \vdash p.f : \text{owned}\, C' \dashv \Delta\end{array}}{\Delta * p.f \dashv x : \text{unique}\, C, \Delta'[x/p.f]}$$
I-ReplaceUnique

$$\frac{\begin{array}{c}\Delta = x : \text{alias}(p'.f)\, C, \Delta' \\ \Delta \vdash p \equiv p' \qquad \Delta \vdash p.f : \text{shared}\, C' \dashv \Delta\end{array}}{\Delta * p.f \dashv x : \text{shared}\, C, \Delta'[x/p.f]}$$
I-ReplaceShared

$$\frac{\Delta = x : \text{alias}(p'.f)\, C, \Delta' \qquad \Delta \vdash p \equiv p'}{\Delta * p.f \dashv x : \bot\, C, \Delta'[x/p.f]}$$
I-ReplaceInaccessible

$$\frac{\begin{array}{c}\nexists x \in \text{dom}(\Delta) : \Delta(x) \vdash x \equiv p.f \\ \Delta \vdash x \rightsquigarrow p.f.g \qquad \Delta * p.f.g \dashv \Delta' \qquad \Delta' * p.f \dashv \Delta''\end{array}}{\Delta * p.f \dashv \Delta''}$$
I-ElimField

$$\frac{\Delta = x : \alpha\, C, \Delta' \qquad \nexists y \in \text{dom}(\Delta') : \Delta \vdash y \rightsquigarrow x}{\Delta * x \dashv \Delta'}$$
I-RemoveVar

$$\frac{\Delta = x : \text{alias}(p)\, C, \Delta'}{\Delta * x \dashv \Delta'[p/x]}$$
I-ReplaceAlias

$$\frac{\begin{array}{c}\Delta = x : \alpha\, C, y : \alpha'\, C', \Delta' \\ \Delta \vdash y \equiv x \qquad \alpha \in \{\text{unique, owned, shared}, \bot\}\end{array}}{\Delta * x \dashv y : \alpha\, C, \Delta'[y/x]}$$
I-ReplaceAliased

$$\frac{\begin{array}{c}\Delta = x : \alpha\, C, \Delta' \qquad \nexists y \in \text{dom}(\Delta') : \Delta \vdash y \equiv x \\ \Delta \vdash y \rightsquigarrow x.f \qquad \Delta * x.f \dashv \Delta' \qquad \Delta' * x \dashv \Delta''\end{array}}{\Delta * x \dashv \Delta''}$$
I-ElimVar

A.8 Framing

$\Delta \vdash \Delta \star p \dashv \Delta'$ denotes that all edges p in the reference graph of Δ are inaccessible or shared in Δ'. This is used to remove alias references to fields that are contained in a method call's frame since those aliases may no longer be correct if the method changes that field.

$\Delta \star \overline{p} \dashv \Delta'$ repeats this process for all paths in the list $\overline{p}$.

$$\frac{\alpha \in \{\text{shared, owned, unique}, \bot\} \qquad \Delta \vdash \Delta' \star p \dashv \Delta''}{\Delta \vdash x : \alpha\, C, \Delta' \star p \dashv x : \alpha\, C, \Delta''}$$
R-Shared

$$\frac{\Delta \vdash p' : \text{shared}\, C \dashv \Delta \qquad \Delta \vdash \Delta' \star p \dashv \Delta''}{\Delta \vdash x : \text{alias}(p')\, C, \Delta' \star p \dashv x : \text{shared}\, C, \Delta''}$$
R-SharedAlias

$$\frac{\nexists f : \Delta \vdash x \rightsquigarrow p.f \qquad \Delta \vdash \Delta' \star p \dashv \Delta''}{\Delta \vdash x : \alpha\, C, \Delta' \star p \dashv x : \alpha\, C, \Delta''}$$
R-Separate

$$\frac{\Delta \vdash \Delta' \star p \dashv \Delta''}{\Delta \vdash x : \alpha\, C, \Delta' \star p \dashv x : \bot\, C, \Delta''}$$
R-Inaccessible

$$\frac{}{\Delta \vdash \cdot \star p \dashv \cdot}$$
R-Empty

$$\Delta \star p_1, \cdots, p_n \dashv \Delta' \iff \Delta = \Delta_0 \vdash \Delta_0 \star p_1 \dashv \Delta_1$$
$$\vdots$$
$$\Delta_{n-1} \vdash \Delta_{n-1} \star p_n \dashv \Delta_n = \Delta'$$

A.9 Unification

$\alpha \leq \alpha'$ denotes that an annotation α may be used in place of α' – this means that α represents fewer permissions than α'.

$$\frac{}{\Delta \vdash \alpha \leq \alpha}$$
A-Id

$$\frac{}{\Delta \vdash \text{shared} \leq \text{unique}}$$
A-Shared

$$\frac{}{\Delta \vdash \bot \leq \alpha}$$
A-Inacc

$$\frac{\Delta \vdash p_1 \equiv p_2}{\Delta \vdash \text{alias}(p_1) \leq \text{alias}(p_2)}$$
A-Alias

$$\frac{\Delta \vdash p.f : \text{shared}\, C \dashv \Delta}{\Delta \vdash \text{shared} \leq \text{alias}(p.f)}$$
A-SharedAlias

unify$(\Delta; \Delta_1; \Delta_2)$ denotes the unification of branches Δ_1 and Δ_2, given a parent environment Δ.

$$\frac{}{\text{unify}(\cdot; \Delta_1; \Delta_2) = \cdot}$$
U-Empty

$$\frac{\text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta'}{\text{unify}(\Delta; \Delta_2; \Delta_1) = \Delta'}$$
U-Comm

$$\frac{\text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta' \quad \Delta_1(x) = \alpha_1\, C \quad \Delta_2(x) = \alpha_2\, C \quad \Delta' \vdash \alpha_1 \leq \alpha_2}{\text{unify}(x : \alpha\, C, \Delta_1; \Delta_2) = x : \alpha_1\, C, \Delta'}$$
U-Join

$$\frac{y * \Delta_1 \dashv \Delta_1' \qquad \text{unify}(\Delta; \Delta_1'; \Delta_2) = \Delta'}{\text{unify}(\Delta; y : \alpha\, C, \Delta_1; \Delta_2) = \Delta'}$$
U-Isolate

A.10 Statement typing

A statement s is valid if $\Delta \vdash s \dashv \Delta'$. Δ' represents the state after s is executed.

For ease of notation, we define $\Delta \vdash e_1 : \alpha_1\ C_1, \cdots, e_n : \alpha_n\ C_n \dashv \Delta'$ to hold iff, for some $\Delta_0, \cdots, \Delta_n$,

$$\Delta = \Delta_0 \vdash e_1 : \alpha_1\ C_1 \dashv \Delta_1$$

$$\vdots$$

$$\Delta_{n-1} \vdash e_n : \alpha_n\ C_n \dashv \Delta_n = \Delta'$$

Similarly, $\Delta \vdash s_1, \cdots, s_n \dashv \Delta'$ holds iff, for some $\Delta_0, \cdots, \Delta_n$,

$$\Delta = \Delta_0 \vdash s_1 \dashv \Delta_1, \cdots, \text{ and } \Delta_{n-1} \vdash s_n \dashv \Delta_n = \Delta'.$$

S-Block
$$\frac{\Delta \vdash \overline{s} \dashv \Delta'}{\Delta \vdash \{\ \overline{s}\ \} \dashv \Delta'}$$

S-Decl
$$\frac{}{\Delta \vdash C\ x; \dashv x : \bot\ C, \Delta}$$

S-AssignVar
$$\frac{\Delta(x) = \alpha\ C \qquad \Delta \vdash e : C \qquad \Delta \nvdash x \equiv e \qquad \Delta * x \dashv \Delta'}{\Delta \vdash x = e; \dashv x : \text{alias}(e)\ C, \Delta'}$$

S-AssignShared
$$\frac{\Delta \vdash x : C \qquad \Delta * x.f \dashv \Delta'}{\text{ftype}(C)(f) = \text{shared}\ C' \qquad \Delta' \vdash e : \text{shared}\ C' \dashv \Delta''}{\Delta \vdash x.f = e; \dashv \Delta''}$$

S-AssignUnique
$$\frac{\Delta \vdash x : C \qquad \Delta * x.f \dashv \Delta' \qquad \text{ftype}(C)(f) = \text{unique}\ C'}{\Delta \nvdash x.f \cong e \qquad \Delta' \vdash e : \text{unique}(x.f)\ C' \dashv \Delta''}{\Delta \vdash x.f = e; \dashv \Delta''}$$

S-New
$$\frac{\begin{array}{c}\alpha_1\ C_1\ f_1, \cdots, \alpha_n\ C_n\ f_n = \text{fields}(C) \\ \Delta \vdash e_1 : \alpha_1\ C_1, \cdots, e_n : \alpha_n\ C_n \dashv \Delta' \\ \Delta'(x) = \alpha\ B \qquad B <: C \qquad \Delta' * x \dashv \Delta'' \\ \forall 0 \leq i < j \leq n : \left[\alpha_i = \text{owned} \implies \Delta' \nvdash e_i \cong e_j\right]\end{array}}{\Delta \vdash x = \text{new}\ C(\overline{e}); \dashv x : \text{unique}\ B, \Delta''}$$

S-Call
$$\frac{\begin{array}{c}\alpha_0\ C_0, \cdots, \alpha_n\ C_n \rightarrow \alpha\ C = \text{mtype}(C_0)(m) \\ \Delta \vdash e_0 : \alpha_0\ C_0, \cdots, e_n : \alpha_n\ C_n \dashv \Delta' \qquad \Delta'(x) = \alpha'\ B \\ B <: C \qquad \Delta' \star e_0, \cdots, e_n \dashv \Delta'' \qquad \Delta'' * x \dashv \Delta''' \\ \forall 0 \leq i < j \leq n : \left[\alpha_i = \text{owned} \implies \Delta' \nvdash e_i \cong e_j\right]\end{array}}{\Delta \vdash x = e_0.m(e_1, \cdots, e_n); \dashv x : \alpha\ B, \Delta'''}$$

S-CallVoid
$$\frac{\begin{array}{c}\alpha_0\ C_0, \cdots, \alpha_n\ C_n \rightarrow \tau = \text{mtype}(C_0)(m) \\ \Delta \vdash e_0 : \alpha_0\ C_0, \cdots, e_n : \alpha_n\ C_n \dashv \Delta' \\ \Delta' \star e_0, \cdots, e_n \dashv \Delta'' \\ \forall 0 \leq i < j \leq n : \left[\alpha_i = \text{owned} \implies \Delta' \nvdash e_i \cong e_j\right]\end{array}}{\Delta \vdash e_0.m(e_1, \cdots, e_n); \dashv \Delta''}$$

S-Conditional
$$\frac{\begin{array}{c}\Delta \vdash e_1 : \texttt{Object} \qquad \Delta \vdash e_2 : \texttt{Object} \\ \Delta \vdash s_1 \dashv \Delta_1 \qquad \Delta \vdash s_2 \dashv \Delta_2 \qquad \text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta'\end{array}}{\Delta \vdash \texttt{if}\ (e_1 == e_2)\ s_1\ \texttt{else}\ s_2 \dashv \Delta'}$$

A.11 Program typing

A program $P = \overline{CL}$ is valid if CL $\texttt{ok}$ for all class declarations CL.

T-Method
$$\frac{\begin{array}{c}\texttt{this} : \alpha_{p0}\ C, \overline{x : \alpha_p\ C} \vdash s \dashv \Delta' \\ \Delta' \vdash e : \alpha\ C' \dashv \Delta'' \\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\ ...\ \} \\ \text{if mtype}(D)(m) = \overline{\alpha'_p\ D} \rightarrow \tau_d, \text{ then} \\ \overline{\alpha_p\ C} = \overline{\alpha'_p\ D} \text{ and } \alpha\ C' = \tau_d\end{array}}{\alpha\ C'\ m(\alpha_{p0}\ \texttt{this}, \overline{\alpha_p\ C\ x})\ \{\ \overline{s}\ \texttt{return}\ e;\ \} \\ \texttt{ok in}\ C}$$

T-VoidMethod
$$\frac{\begin{array}{c}\texttt{this} : \alpha_{p0}\ C, \overline{x : \alpha_p\ C} \vdash s \dashv \Delta' \\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \cdots\ \} \\ \text{if mtype}(D)(m) = \overline{\alpha'_p\ D} \rightarrow \tau_d,, \text{ then} \\ \overline{\alpha_p\ C} = \overline{\alpha'_p\ D} \text{ and } \texttt{void} = \tau_d\end{array}}{\texttt{void}\ m(\alpha_{p0}\ \texttt{this}, \overline{\alpha_p\ C\ x})\ \{\ \overline{s}\ \}\ \texttt{ok in}\ C}$$

T-Class
$$\frac{\begin{array}{c}K = C(\overline{\alpha_{f'}\ D\ g}, \overline{\alpha_f\ C\ f})\{\ \texttt{super}(\overline{g});\ \textit{this}.\overline{f} = \overline{f};\} \\ \overline{F} = \overline{\alpha_f\ C\ f}; \qquad \text{fields}(D) = \overline{\alpha_{f'}\ D\ g} \qquad \overline{M}\ \texttt{ok in}\ C\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{F}\ K\ \overline{M}\ \}\ \texttt{ok}}$$