# Ownership, Uniqueness and Immutability

Johan Östlund, Tobias Wrigstad
Royal Institute of Technology, Sweden
{johano, tobias}@dsv.su.se

Dave Clarke
CWI, Amsterdam, The Netherlands
dave@cwi.nl

Beatrice Åkerblom
Royal Institute of Technology, Sweden
beatrice@dsv.su.se

## Abstract

Programming in an object-oriented language demands a fine balance between high degrees of expressiveness and control. At one level, we need to permit objects to interact freely to achieve our implementation goals. At a higher level, we need to enforce architectural constraints so that the system can be understood by new developers and can evolve as requirements change. To resolve this tension, numerous explorers have ventured out into the vast landscape of type systems expressing ownership and behavioural restrictions such as immutability. (Many have never returned.) This work in progress reports on our consolidation of the resulting discoveries into a single programming language. Our language, Joe$_3$, imposes little additional syntactic overhead, yet can encode powerful patterns such as fractional permissions, and the reference modes of Flexible Alias Protection.

## 1. Introduction

Recent years have seen a number of proposals put forward to add more structure to object-oriented programming languages, for example, via ownership types [12], or to increase the amount of control over objects by limiting how they can be accessed by other objects, via notions such as read-only or immutability. Immutability spans the following spectrum: *Class immutability* ensures that all instances of a class are immutable, for example, Java's String class; *object immutability* ensures that some instances of a class are immutable, though other instances may remain mutable; and *read-only*—or *reference immutability*—prevents modifications of an object via certain references, without precluding the co-existence of normal and read-only references to the same object.

Immutable objects help avoid aliasing problems and data races in multi-threaded programs [4, 18], and also enhance program understanding, as read-only or immutable annotations are verified to hold at compile-time [31]. According to Zibin et al. [35], immutability (including read-only references) can be used for modelling, verification, compile- and run-time optimisations, refactoring, test input generation, regression oracle creation, invariant detection, specification mining and program comprehension. Read-only references have been used in proposals to strengthen object encapsulation and manage aliasing. Kniesel and Theisen [21] use read-only references to allow and to manage side-effects due to aliasing. Noble, Vitek and Potter [27] introduce an *arg* reference mode to allow aggregates to rely only on immutable parts of external objects. Hogg's Islands [19] and Müller and Poetzsch-Heffter's Universes [24] use read-references to allow temporary representation exposure in a safe fashion.

### 1.1 Our Contributions

The programming language, Joe$_3$, we propose in this paper offers ownership and uniqueness to control the alias structure of object graphs, and lightweight effects and a mode system to encode various notions of immutability. It is a relatively straightforward extension of Clarke and Wrigstad's *external uniqueness* proposal (Joline) [14, 32] (without inheritance), and the syntactic overhead due to additional annotations is surprisingly small given the expressiveness of the language. Not only can we encode the three forms of immutability mentioned above, but we can encode something akin to the *arg* mode from Flexible Alias Protection [27], Fractional Permissions [7], and the context-based immutability of Universes [24], all the while preserving the owners-as-dominators encapsulation invariant. Furthermore, as our system is based on ownership types, we can distinguish between outgoing aliases to external, non-rep objects and aliases to internal objects and allow modification of the former (but not the latter) through a read-only reference.

Our system is closest in spirit to SafeJava [4], but we allow access modes on all owner parameters of a class, read-only references and an interplay between borrowing and immutable objects that can encode fractional permissions.

### 1.2 Why We Could Add Read-Only To Java (Almost)

In his paper "Why We Shouldn't Add Read-Only To Java (Yet)" [8], John Boyland criticises existing proposals for handling read-only references on the following points:

1. Read-only arguments can be silently captured when passed to methods;
2. A read-only annotation cannot express whether
   (a) the referenced *object* is immutable, and hence the reference can be safely stored;
   (b) a read-only reference is unique and thus immutable, as no aliases exist which could be used to mutate the object;
   (c) mutable aliases of a read-only reference can exist, implying that the referenced object should be cloned before used, to prevent it being modified underfoot resulting in *observational exposure*.[1]

Joe$_3$ addresses all of these problems. First, Joe$_3$ supports owner-polymorphic methods, which can express that a method does not capture one or all of its arguments. Second, we decorate owners with modes that govern how the objects owned by that owner will be treated in a context. Together with auxiliary constructs inherited from Joline, the modes can express immutability both in terms of 2.a) and 2.b), and read-only which permits the existence of mutable aliases (2.c). Moreover, Joe$_3$ supports fractional permissions—converting a mutable unique reference into several immutable references for a certain context. This allows safe representation exposure without the risk for observational exposure (2.c).

Joe$_3$ allows class, object and reference immutability. Unique references, borrowing and owner-polymorphic methods allow us to

[1] Observational exposure occurs when changes to state are observed through a read-only reference.

simulate fractional permissions and staged, external initialisation of immutable objects through auxiliary methods. As we base modification rights on owners (in the spirit of Joe$_1$'s effects system), we achieve what we call *context-based* immutability, which is essentially the same kind of read-only found in Müller and Poetzsch-Heffter's Universes [24].

Joe$_3$ allows both read-only references and true immutables in the same language. This provides the safety desired by Boyland, but also allows coding patterns which do rely on observing changes in an object. Apart from the fact that we do not yet consider inheritance, which we believe to be a straightforward extension, we conclude that we could indeed add read-only to Java, now[2].

***Outline*** Section 2 introduces the Joe$_3$ language through a set of motivating examples—different nestings of mutable and immutable objects, context-based immutability, immutable objects, and staged construction of immutables. Section 3 gives a brief formal account of Joe$_3$. Section 4 outlines a few simple but important extensions—immutable classes and Greenhouse and Boyland style regions [17]—describes how they further enhance the system and discusses how to encode the modes of Flexible Alias Protection [27]. Section 5 surveys related work not covered above. Section 6 contains an outlook for the future, and Section 7 concludes.

## 2. Meet Joe$_3$

In this section we describe Joe$_3$ with the help of a couple of motivating examples. Joe$_3$ is a class-based, object-oriented programming language with deep ownership, owner-polymorphic methods, ownership transfer through external uniqueness, an effects (revocation) system and a simple mode system which decorates owners with permissions to indicate how references with the annotated owners can be used. Beyond the carefully designed combination of features, the annotation of owners with modes is the main novelty in Joe$_3$. The modes indicate that a reference may be read or written (+) or only read (-), or that the reference is immutable (*). Read and immutable annotations on an owner in the class header represent a promise that the code in the class body will not change objects owned by that owner. The key to preserving and respecting immutability and read-only in Joe$_3$ is a simple effects system, rooted in ownership types, and inspired by Clarke and Drossopoulou's Joe$_1$ [11]. Classes, and hence objects, have rights to read or modify objects belonging to certain owners; only a minor extension to the type system of Clarke and Wrigstad's Joline [14, 32] is required to ensure that these rights are not violated.

The syntax of Joe$_3$ (shown in Figure 5) should be understandable to a reader with insight into ownership types and Java-like languages. Classes are parameterised with owners related to each other by an inside/outside nesting relation. An owner is a permission to reference the representation of another object. Class headers have this form:

```
class List<data outside owner> { ... }
```

Each class has at least two owner parameters, `this` and `owner`, which represent the representation of the current object and the representation of the owner of the current object, respectively. In the example above, the `List` class has an additional permission to reference objects owned by `data`, which is nested outside `owner`. Types are formed by instantating the owner parameters, `this:List<owner>`. An object with this type belongs to the representation of the current object and has the right to reference objects owned by `owner`. There are two nesting relations between owners, inside and outside. They exist in two forms each, one reflexive (`inside`/`outside`) and one non-reflexive (`strictly-inside`/`strictly-outside`). Thus, going back to our list example, a type `this:List<this>` denotes a list object belonging to the current representation, storing objects in the current representation.

A more detailed introduction is given in Section 3. Apart from ownership types, the key ingredients in Joe$_3$ are the following:

- (externally) unique types (written `unique[p]:Object`), a special *borrowing* construct for temporarily treating a unique type non-uniquely, and *owner casts* for converting unique references permanently into normal references.
- modes on owners—mutable '+', read-only '-', and immutable '*'. These appear on every owner parameter of a class and owner polymorphic methods, though not on types.
- an effects revocation clause on methods which states which owners will not be modified in a method. An object's default set of rights is derived from the modes on the owner parameters in the class declaration. An additional example of a use of `revoke` is found at the end of Section 4.2.

Annotating owners at the level of classes (that is, for all instances) rather than types (for each reference) is a trade-off. Rather than permitting distinctions to be made using modes on a per reference basis, we admit only per class granularity. Some potential expressiveness is lost, though the syntax of types does not need to be extended. Nonetheless, the effects revocation clauses regain some expressiveness that per reference modes would give. Another virtue of using per class rather than per reference modes is that we avoid some covariance problems found in other proposals (see related work) as what you can do with a reference depends on the context and is not a property of the reference. Furthermore, our proposal is statically checkable in a modular fashion. We also need no run-time representation of the modes.

### 2.1 Motivating Examples

The following examples illustrate the range of constraints that can be expressed in Joe$_3$.

#### 2.1.1 A Mutable List With Immutable Contents

The code in Figure 1 shows parts of an implementation of a list class. The owner parameter `data` is decorated with the mode read-only (denoted '-'), indicating that the list will never cause write effects to objects owned by `data`.

The owner of the list is called `owner` and is implicitly declared. The method `getFirst()` is annotated with `revoke owner`, which means that the method will not modify the object or its transitive state. This means the same as if `owner-` and `this-` would have appeared in the class head. This allows the method to be called in objects where the list owner is read-only.

This list class can be instantiated in four different ways, depending on the access rights to the owners in the type held by the current context:

- both the list and its data objects are immutable, which only allows `getFirst()` to be invoked, and its resulting object is immutable;
- both are mutable, which imposes no additional restrictions;

[2] While the syntactic price of our proposal is no doubt steep when added to Java, adding it to a language with ownership types and uniqueness, such as Joline, is virtually for free. On a side-note, the authors believe that if ownership types is ever to make it into mainstream languages, simple but powerful extensions, such as external uniqueness, read-only references and immutability will be crucial in convincing programmers of the virtues of the added complexity.

```
class Link<data- strictly-outside owner> {
  data:Object obj = null;
  owner:Link<data> next = null;
}

class List<data- strictly-outside owner> {
  this:Link<data> first = null;

  void addFirst(data:Object obj) {
    this:Link<data> tmp = new this:Link<data>();
    tmp.obj = obj;
    tmp.next = this.first;
    this.first = tmp;
  }

  void filter(data:Object obj) {
    this:Link<data> tmp = this.first;
    if (tmp == null) return;
    while (tmp.next != null)
      if (tmp.next.obj == obj)
        tmp.next = tmp.next.next;
      else
        tmp = tmp.next;
    if (this.first != null && this.first.obj == obj)
      this.first = this.first.next;
  }

  data:Object getFirst() revoke owner {
    return this.first.obj;
  }
}
```

**Figure 1.** Fragment of a list class. As the `data` owner parameter is declared read-only (via '`-`') in the class header, no method in `List` may modify an object owned by `data`. Observe that the syntactic overhead is minimal for an ownership types system.

- the list is mutable but the data objects are not, which imposes no additional restrictions, though `getFirst()` returns a read-only reference; and
- the data objects are mutable, but the list not, which only allows `getFirst()` to be invoked, though the resulting object is mutable.

The last form is interesting and relies on the fact that we can specify, thanks to ownership types, that the data objects are not part of the representation of the list. Most existing proposals for read-only references (*e.g.*, Islands [19], JAC [20, 21], ModeJava [28, 29], Javari [31], and IGJ [35]) cannot express this constraint in a satisfactory way, as these proposals cannot distinguish between an object's outside and inside.

### 2.1.2 Context-Based Read-Only

As shown in Figure 2, different clients of the list can have different views of the same list at the same time. The class `Reader` does not have permission to mutate the list, but has no restrictions on mutating the list elements. Dually, the `Writer` class can mutate the list but not its elements.

As owner modes only reflect what a class is allowed to do to objects with a certain owner, `Writer` can add data objects to the list that are read-only to itself and the list, but writable by `Example` and `Reader`. This is a powerful and flexible idea. For example, `Example` can pass the list to `Writer` to filter out certain objects in the list. `Writer` can then consume or change the list, or copy its contents to another list, *but not modify them*. `Writer` can then return the list to `Example`, without `Example` losing its right to modify the objects obtained from the returned list. This is similar to the context-based read-only in Universes-based systems [24, 26]. In contrast, however, we do not allow representation exposure via read-only references.

```
class Writer<o+ outside owner, data- strictly-outside o> {
  void mutateList(o:List<data> list) {
    list.addFirst(new data:Object());
  }
}

class Reader<o- outside owner, data+ strictly-outside o> {
  void mutateElements(o:List<data> list) {
    list.elementAt(0).mutate();
  }
}

class Example {
  void example() {
    this:List<world> list = new this:List<world>();
    this:Writer<this, world> w =
                                 new this:Writer<this, world>();
    this:Reader<this, world> r =
                                 new this:Reader<this, world>();
    w.mutateList(list);
    r.mutateElements(list);
  }
}
```

**Figure 2.** Different objects can have different views of the same list at the same time. `r` can modify the elements of `list` but not the `list` itself, `w` can modify the `list` object, but not the list's contents, and instances of `Example` can modify both the list and its contents.

### 2.1.3 Borrowing Blocks and Owner-polymorphic Methods

Before moving on to the last two examples, we need to introduce borrowing blocks and owner-polymorphic methods [13, 32, 10], which make it easier to program using unique references and ownership. (The interaction between unique references, borrowing, and owner-polymorphic methods has been studied thoroughly by Clarke and Wrigstad [14, 32].) A borrowing block has the following syntax:

`borrow` *lval* `as` $\alpha\ x$ `in` { $s$ }

The borrowing operation destructively reads a unique reference from an l-value (*lval*) to a non-unique, stack-local variable ($x$) for the scope of the borrowing block ($s$). The block also introduces a fresh block-local owner that becomes the new owner of the borrowed value. Every type of every variable or field that stores an alias to the borrowed value must have this owner in its type. Clearly, this is not the case for any pre-existing field or variable. Owner-polymorphic methods (see below) allow granting permissions to reference the borrowed value for the duration of a method call. This is the only way in which references to borrowed values can be exported to outside a borrowing block. As all method calls in the borrowing block must have returned when the block exits, clearly no residual aliasing can exist. Thus, when the borrowing block exits, the borrowed value can be reinstated and is once again unique.

Due to the strong encapsulation of external uniqueness, borrowing borrows an entire unique aggregate in one single hit and makes it stack-local.

An owner-polymorphic method is simply a method which takes owners as parameters. The methods `m1` and `m2` in `Client` in Figure 3 are examples of such. Owner-polymorphic methods can be seen as accepting stack-local permissions to reference (and possibly mutate) objects that it otherwise may not be allowed to reference. Owner parameters ($p$`*` and $p$`-` in the methods in Figure 3) of

```
class Client {
  <p* inside world> void m1(p:Object obj) {
    obj.mutate(); // Error
    obj.toString(); // Ok
    // assign to field is not possible
  }

  <p- inside world> void m2(p:Object obj) {
    obj.mutate(); // Error
    obj.toString(); // Ok
  }
}

class Fractional<o+ outside owner> {
  unique[this]:Object obj = new this:Object();

  void example(o:Client c)  {
    borrow obj as p*:tmp in {   // **
      c.m1(tmp);                // ***
      c.m2(tmp);                // ****
    }
  }
}
```

**Figure 3.** An implementation of fractional permissions using borrowing and unique references.

owner-polymorphic methods are not in the scope at the class level. Thus, method arguments with such a parameter in its type cannot be captured within the method body (—it is *borrowed* [6]).

### 2.1.4 Immutability

The example in Figure 2 shows that a read-only reference to an object does not preclude the existence of mutable references to the same object elsewhere in the system. This allows observational exposure—for good and evil.

The immutability annotation '*' imposes all the restrictions a read-only type has, but it also guarantees that no aliases with write permission exist in the system. Our simple way of creating an immutable object is to move a *mutable* unique reference into a variable with immutable type, just as in SafeJava [4].

This allows us to encode fractional permissions and to do staged construction of immutables, both discussed below.

### 2.1.5 Fractional Permissions

The example in Figure 3 shows an implementation of Fractional Permissions. We can use Joline's borrowing construct to *temporarily* move a mutable unique reference into an immutable variable (line **), freely alias the reference (while preserving read-only) (lines *** and ****), and then implicitly move the reference back into the unique variable again and make it mutable. This is essentially Boyland's Fractional Permissions [7]. As stated above, both the owner-polymorphic methods and the borrowing block guarantee not to capture the reference. A borrowed reference can be aliased any number of times in any context to which it has been exported, without the need to keep track of "split permissions" [7] as we know for sure that all permissions to alias the pointer are invalidated when the borrowing block exits. The price of this convenience is that the conversion from mutable to immutable and back again must be done in the same place.

Interestingly, `m1` and `m2` are equally safe to call from `example`. Both methods have revoked their right to cause write effects to objects owned by `p`, indicated by the * and - annotations on `p`, respectively. The difference between the two methods is that the first method knows that `obj` will not change under foot (making it safe to, for example, use `obj` as a key in a hash table), whereas the second method cannot make such an assumption.

```
class Client<p* outside owner, data+ strictly-outside p> {
  void method() {
    this:Factory<p, data> f = new this:Factory<p, data>();
    p:List<data> immutable = f.createList();
  }
}

class Factory<p* inside world, data+ strictly-outside p> {
  p:List<data> createList() {
    unique[p]:List<data> list = new p:List<data>();
    borrow list as temp+ l in { // 2nd stage of construct.
      l.add(new data:Object());
    }
    return list--;   // unique reference returned
  }
}
```

**Figure 4.** Staged construction of an immutable list

### 2.1.6 Initialisation of Immutable Objects

An issue with immutable objects is that even such objects need to mutate in their construction phase. Unless caution is taken the constructor might leak a reference to `this` (by passing `this` to a method) or mutate other immutable objects of the same class. The standard solution to this problem in related proposals is to limit the construction phase to the constructor [31, 35, 18]. Continuing initialisation by calling auxiliary methods *after* the constructor returns is simply not possible. Joe$_3$, on the other hand, permits *staged construction*, as we demonstrate in Figure 4. In this example a client uses a factory to create an immutable list. The factory creates a unique list and populates it. The list is then destructively read and returned to the caller as an immutable.

## 3. A Formal Definition of Joe$_3$

In this section, we formally present the static semantics of Joe$_3$, and argue how it guarantees immutability and read-only.

### 3.1 Joe$_3$'s Static Semantics

We now describe Joe$_3$'s type system, which can be seen as a simplification of Joline's [14, 32] extended with effects annotations and modes on owners. To simplify the formal account, we omit inheritance and constructors. Furthermore, following Joline, we rely on destructive reads to preserve uniqueness and require that movement is performed using an explicit operation.

The abstract syntax of Joe$_3$ is shown in Figure 5. For simplicity, we assume that names of fields, method and classes are unique. $c, m, f, x$ are metavariables ranging over names of classes, methods, fields and local variables, respectively. $q$ and $p$ are names of owners.

Types have the syntax $p\,c\langle\overline{p}\rangle$. We sometimes write $p\,c\langle\sigma\rangle$ for some type where $\sigma$ is a map from the names of the owner parameters in the declaration of a class $c$ to the actual owners used in the type. In code, a type's owner is connected to the class name with a ':' to make the type one syntactic unit.

Unique types have the syntax $\texttt{unique}_p\; c\langle\overline{p}\rangle$. The keyword `unique` specifies that the owner of an object is really the field or variable that contains the only (external) reference to it in the system. The owner annotation on the unique type is called the *movement bound*. Movement bounds govern the maximal outwards movement of a unique, so as to preserve the owners-as-dominators property. In code, movement bounds are denoted `unique[p]`. For details, see Wrigstad [32].

In systems with ownership types, an owner is a permission to reference objects with that owner. Classes, such as the canonical list example, can be parameterised with owners to enable them to

$$\begin{array}{lll}
P & ::= \overline{C} & \text{(program)}\\
C & ::= \mathtt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ \{\ \overline{\mathit{fd}}\ \overline{\mathit{md}}\ \} & \text{(class)}\\
\mathit{fd} & ::= t\ f := e; & \text{(field)}\\
\mathit{md} & ::= \langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ t\ m(\overline{t\ x})\ \mathtt{revoke}E\ \{\ s; \mathtt{return}\ e\ \} & \text{(method)}\\
e & ::= \mathit{lval} \mid \mathit{lval}\mathtt{--} \mid e.m(\overline{e}) \mid \mathtt{new}\ p\ c\langle\sigma\rangle \mid \mathtt{null} & \text{(expr.)}\\
s & ::= \mathit{lval} := e \mid t\ x := e \mid s;s \mid e & \text{(statement)}\\
 & \quad \mid \mathtt{borrow}\ \mathit{lval}\ \mathtt{as}\ \alpha\ x\ \mathtt{in}\ \{\ s\ \} & \\
\mathit{lval} & ::= x \mid e.f & \text{(l-value)}\\
\mathsf{R} & ::= \prec^* \mid \succ^* \mid \prec^+ \mid \succ^+ & \text{(nesting relation)}\\
t & ::= p\ c\langle\overline{p}\rangle \mid \mathtt{unique}_p\ c\langle\overline{p}\rangle & \text{(type)}\\
E & ::= \epsilon \mid E, p & \text{(write right revocation clause)}\\
\Gamma & ::= \epsilon \mid \Gamma, x : t \mid \Gamma, \alpha\,\mathsf{R}\,p & \text{(environment)}\\
\sigma & ::= \overline{\alpha \mapsto p} & \text{(owner substitution)}\\
\alpha & ::= p\mathtt{-} \mid p\mathtt{+} \mid p\mathtt{*} & \text{(owner param.)}
\end{array}$$

**Figure 5.** Abstract syntax of Joe$_3$. In the code examples, owner nesting relations (R) are written as `inside` ($\prec^*$), or `strictly-inside` ($\prec^+$), etc. for clarity.

be given permission to access external objects. For example, the list class has an owner parameter for the (external) data objects of the list. In Joe$_3$ the owner parameters of a class or owner-polymorphic method also carry information about what effects *the current context may cause on* the objects having the owner in question. For example, if $p$`-` ($p$ is read-only) appears in some context $c$, this means that $c$ may reference objects owned by $p$, but not modify them directly. We refer to the part of an owner that controls its modification rights as the *mode*.

In contrast with related effect systems (*e.g.*, [17, 11]), we use effect annotations on methods to show what is *not* affected by the method—essentially *temporarily revoking* rights to change. For example, `getFirst()` in the list in Figure 1 does not modify the list object and is thus declared using a `revoke` clause thus:

```
data:Object getFirst() revoke owner { ... }
```

This will force the method body to type-check in an environment where `owner` (and `this`) are read-only.

***Notation*** Given $\sigma$, a map from (annotated) owner parameters to actual owners, let $\sigma^p$ mean $\sigma \uplus \{\mathtt{owner+} \mapsto p\}$. For the type `this:List<owner>`, $\sigma = \{\mathtt{owner+} \mapsto \mathtt{this}, \mathtt{data-} \mapsto \mathtt{owner}\}$. We write $\sigma(p\ c\langle\overline{p}\rangle)$ to mean $\sigma(p)\ c\langle\sigma(\overline{p})\rangle$. For simplicity, we sometimes completely disregard modes and allow $\sigma(p)$. On the other hand, $\sigma^\circ$ denotes a mode preserving variant of $\sigma$ *s.t.* if $q\mathtt{+} \mapsto p \in \sigma$, then $q\mathtt{+} \mapsto p\mathtt{+}$ in $\sigma^\circ$.

Let $\mathsf{md}(\alpha)$ and $\mathsf{nm}(\alpha)$ return the mode and owner name of $\alpha$, respectively. For example, if $\alpha = p\mathtt{+}$, then $\mathsf{md}(\alpha) = \mathtt{+}$ and $\mathsf{nm}(\alpha) = p$.

$\mathsf{CT}$ is a class table computed from a program $P$. It maps class names to type information for fields and methods in the class body. $\mathsf{CT}(c)(f) = t$ means that field $f$ in class $c$ has type $t$. $\mathsf{CT}(c)(m) = \forall\,\overline{\alpha\,\mathsf{R}\,q}.\,\overline{t} \rightarrow t; E$ means that method $m$ in class $c$ have formal owner-parameters declared $\overline{\alpha\,\mathsf{R}\,q}$, formal parameter types $\overline{t}$, return type $t$ and revoked rights $E$.

Predicate $\mathsf{isunique}(t)$ is true iff $t$ is a unique type. $\mathtt{owner}(t)$ returns the owner of a type, and $\mathsf{owners}(t)$ returns the owner names used in a type or a method type. Thus, $\mathtt{owner}(p\ c\langle\overline{p}\rangle) = p$ and $\mathsf{owners}(p\ c\langle\overline{p}\rangle) = \{p\} \cup \overline{p}$.

$E_c$ denotes the set of owners to which class $c$ has *write* permission. For example, the list class in Figure 1 has $E_{\mathtt{List}} = \{\mathtt{owner}\}$, whereas the writer class in Figure 2 has $E_{\mathtt{Writer}} = \{\mathtt{owner}, \mathtt{o}\}$. $E_c$ is defined thus:

$$E_c = \begin{cases} \{p \mid p\mathtt{+} \in \overline{\alpha}\} \cup \{\mathtt{owner}\} & \text{if } \mathtt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,\_}\rangle\ \{\ \_\ \} \in P \\ \bot & \text{otherwise} \end{cases}$$

| | |
|---|---|
| $\Gamma \vdash C$ | Good class |
| $\Gamma \vdash \mathit{fd}$ | Good field |
| $\Gamma \vdash \mathit{md}$ | Good method |
| $\Gamma \vdash s; \Gamma'$ | Statement $s$ is wf under $\Gamma$ and produces $\Gamma'$ |
| $\Gamma \vdash e : t$ | Expression $e$ has type $t$ under $\Gamma$ |
| $\Gamma \vdash t$ | Good type |
| $\Gamma \vdash E$ | Good write right revocation clause |
| $\Gamma \vdash \alpha\,\mathsf{R}\,p$ | Owner parameter $\alpha$ is R-related to $p$ in $\Gamma$ |
| $\Gamma \vdash \alpha\ \mathit{perm}$ | Good owner parameter $\alpha$ |
| $\Gamma \vdash p$ | Good owner |
| $\Gamma \vdash \diamond$ | The environment $\Gamma$ is well-formed |

**Table 1.** Judgments in the Joe$_3$ formalisation.

$E \setminus E'$ denotes set difference. The judgments in the type system are summarised in Table 1.

### *Good Class*

$$\frac{\begin{array}{c}\Gamma = \mathtt{owner+} \prec^* \mathtt{world}, \mathtt{this+} \prec^+ \mathtt{owner}, \overline{\alpha\,\mathsf{R}\,p}, \mathtt{this} : t \\ t = \mathtt{owner}\ c\langle\overline{\mathsf{nm}(\alpha)}\rangle \quad \Gamma \vdash \mathtt{owner+} \prec^* \overline{\mathsf{nm}(\alpha)} \quad \Gamma \vdash \overline{\mathit{fd}} \quad \Gamma \vdash \overline{\mathit{md}}\end{array}}{\vdash \mathtt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ \{\ \overline{\mathit{fd}}\ \overline{\mathit{md}}\ \}}\ \textsc{(class)}$$

A class is well-formed if all its owner parameters are outside `owner`. This makes sure that a class can only be given permission to reference external objects and is key to preserving the owners-as-dominators property of deep ownership systems [10]. The environment $\Gamma$ is constructed from the owners in the class header, their nesting relations and modes, plus `owner+` and `this+` giving an object the right to modify itself. Thus, class-wide read/write permissions are encoded in $\Gamma$, and must be respected by field declarations and methods.

***Good Field, Good Method*** The function $\Gamma\ \mathsf{revoke}\ E$ is a key player in our system—it revokes the write rights mentioned in $E$, by converting them to read rights in $\Gamma$. It also makes sure that `this` is not writable whenever `owner` is not. For example, given $E = \{p\}$, we have $p\mathtt{+} \notin \mathrm{dom}(\Gamma\ \mathsf{revoke}\ E)$, so if $\Gamma\ \mathsf{revoke}\ E \vdash s; \Gamma'$, $s$ does not write to objects owned by $p$.

$$\begin{array}{rcl}
\epsilon\ \mathsf{revoke}\ E & = & \epsilon \\
(\Gamma, x : t)\ \mathsf{revoke}\ E & = & (\Gamma\ \mathsf{revoke}\ E), x : t \\
(\Gamma, \alpha\,\mathsf{R}\,p)\ \mathsf{revoke}\ E & = & (\Gamma\ \mathsf{revoke}\ E), (\alpha\,\mathsf{R}\,p\ \mathsf{revoke}\ E) \\
(\alpha\,\mathsf{R}\,p)\ \mathsf{revoke}\ E & = & (\alpha\ \mathsf{revoke}\ E)\ \mathsf{R}\,p \\
p\mathtt{-}\ \mathsf{revoke}\ E & = & p\mathtt{-} \\
p\mathtt{+}\ \mathsf{revoke}\ E & = & p\mathtt{-},\ \text{if } p \in E \text{ else } p\mathtt{+} \\
\mathtt{this+}\ \mathsf{revoke}\ E & = & \mathtt{this-},\ \text{if } \mathtt{owner} \in E \text{ else } \mathtt{this+} \\
p\mathtt{*}\ \mathsf{revoke}\ E & = & p\mathtt{*}
\end{array}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash t\ f := e}\ \textsc{(field)} \qquad \frac{\begin{array}{c}\Gamma' = \Gamma, \overline{\alpha\,\mathsf{R}\,p} \quad \Gamma' \vdash E \\ (\Gamma'\ \mathsf{revoke}\ E), \overline{x : t} \vdash s; \Gamma'' \quad \Gamma'' \vdash e : t\end{array}}{\Gamma \vdash \langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ t\ m(\overline{t\ x})\ \mathtt{revoke}\ E\ \{\ s; \mathtt{return}\ e\ \}}\ \textsc{(method)}$$

A field declaration is well-formed if its initialising expression has the appropriate type. The rules for good method is a little more complex: any additional owner parameters in the method header are added to $\Gamma$, with modes and nesting. Furthermore, the effect clause must be valid: *i.e.*, you can only revoke rights that you own.

***Expressions*** The expression rules pretty much follow those of Joline extended to cater for effects.

$$\overset{\text{(EXPR-LVAL)}}{\dfrac{\Gamma \vdash_{\mathsf{lv}} \mathit{lval} : t \\ \neg\mathsf{isunique}(t)}{\Gamma \vdash \mathit{lval} : t}} \qquad \overset{\text{(EXPR-LVAL-DREAD)}}{\dfrac{\Gamma \vdash_{\mathsf{lv}} \mathit{lval} : t \quad \mathsf{isunique}(t) \\ \mathit{lval} \equiv e.f \Rightarrow \Gamma \vdash e : p\,c\langle\sigma\rangle \wedge \Gamma \vdash p\texttt{+}\ \mathit{perm}}{\Gamma \vdash \mathit{lval}\texttt{--} : t}}$$

Destructively reading a field in an object owned by some owner $p$ requires that $p$+ is in the environment.

$$\overset{\text{(EXPR-VAR)}}{\dfrac{x : t \in \Gamma}{\Gamma \vdash_{\mathsf{lv}} x : t}} \qquad \overset{\text{(EXPR-FIELD)}}{\dfrac{\Gamma \vdash e : p\,c\langle\sigma\rangle \quad \mathsf{CT}(c)(f) = t \\ \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}}{\Gamma \vdash_{\mathsf{lv}} e.f : \sigma^p(t)}}$$

Judgements of the form $\Gamma \vdash_{\mathsf{lv}} \mathit{lval} : t$ deal with l-values.

In Joline, owner arguments to owner-polymorphic methods must be passed in explicitly. Here, we assume the existence of an inference algorithm to bind the names of the owner parameters to the actual arguments at the call site. This is $\sigma_a$ in the rule.

$$\overset{\text{(EXPR-INVOKE)}}{\dfrac{\Gamma \vdash e : \underline{p\,c\langle\sigma\rangle} \ \ \mathsf{CT}(c)(m) = \forall\,\overline{\alpha\,\mathsf{R}\,p}.\,\overline{t} \rightarrow t; E \ \ \sigma' = \sigma^p \uplus \sigma_a \\ \Gamma \vdash \sigma'(\overline{\alpha\,\mathsf{R}\,p}) \quad \Gamma \vdash \sigma'^{\circ}(\overline{\alpha})\ \mathit{perm} \quad \Gamma \vdash \overline{e} : \sigma'(\overline{t}) \quad \Gamma \vdash \sigma'(t) \\ \Gamma \vdash \sigma'(E_c \backslash E) \quad \texttt{this} \in \mathsf{owners}(\mathsf{CT}(c)(m)) \Rightarrow e \equiv \texttt{this}}{\Gamma \vdash e.m(\overline{e}) : t}}$$

By the first clause of (EXPR-INVOKE), method invocations are not allowed on unique types. The third clause creates a substitution from the type of the receiver $(\sigma^p)$ and the implicit mapping from owner parameter to actual owner $(\sigma_a)$. $\Gamma \vdash \sigma'^{\circ}(\overline{\alpha})\ \mathit{perm}$ makes sure that owner parameters that are writable and immutable are instantiated with writable or immutable owners respectively. Clauses six and seven ensure that the argument expressions have the correct types and that the return type is valid. Clause eight checks that the method's effects are valid in in the current context, and clause nine makes sure that any method with `this` in its type (return types, argument types or owners in the owner parameters declaration) can only be invoked with `this` as receiver—this is the standard static visibility constraint of ownership types [12].

$$\overset{\text{(EXPR-NULL)}}{\dfrac{\Gamma \vdash t}{\Gamma \vdash \texttt{null} : t}} \qquad \overset{\text{(EXPR-NEW)}}{\dfrac{\Gamma \vdash p\,c\langle\overline{p}\rangle}{\Gamma \vdash \texttt{new}\ p\,c\langle\overline{p}\rangle : \texttt{unique}_p\ c\langle\overline{p}\rangle}}$$

By (EXPR-NULL), `null` can have any well-formed type. By (EXPR-NEW), object creation results in unique objects. (Without constructors, it is obviously the case that the returned reference is unique—see Wrigstad's dissertation [32] for an explanation why adding constructors is not a problem.)

***Good Statements***

$$\overset{\text{(STAT-LOCAL-ASGN)}}{\dfrac{x \neq \texttt{this} \\ x : t \in \Gamma \\ \Gamma \vdash e : t}{\Gamma \vdash x := e; \Gamma}} \qquad \overset{\text{(STAT-FIELD-ASGN)}}{\dfrac{\Gamma \vdash e : p\,c\langle\sigma\rangle \quad \mathsf{CT}(c)(f) = t \\ \Gamma \vdash e' : \sigma^p(t) \quad \Gamma \vdash p\texttt{+}\ \mathit{perm} \\ \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}}{\Gamma \vdash e.f := e'; \Gamma}}$$

In contrast to local variable update, assigning to a field requires write permission to the object containing the field.

$$\overset{\text{(STAT-BORROW)}}{\dfrac{\mathit{lval} \equiv e.f \Rightarrow \Gamma \vdash e : q\,c'\langle\_\rangle \wedge \Gamma \vdash q\texttt{+}\ \mathit{perm} \\ \Gamma \vdash \mathit{lval} : \texttt{unique}_p\ c\langle\sigma\rangle \quad \Gamma, \alpha \prec^{+} p, x : \mathsf{nm}(\alpha)\,c\langle\sigma\rangle \vdash s; \Gamma}{\Gamma \vdash \texttt{borrow}\ \mathit{lval}\ \ \texttt{as}\ \ \alpha\ x\ \texttt{in}\ \texttt{\{}\ s\ \texttt{\}}; \Gamma}}$$

In our system, unique references must be borrowed before they can be used as receivers of method calls or field accesses. The borrowing operation moves the unique object from the source l-value to a stack-local variable temporarily and introduces a fresh owner ordered strictly inside the unique object's movement bound. The new owner is annotated with a read/write permission which must be respected by the body of the borrowing block. As the owner of the borrowed unique goes out of scope when the borrowing block exits, all fields or variables with types that can refer to the borrowed object become inaccessible. Thus, the borrowed value can be reinstated and is once again unique. As borrowing temporarily nullifies the borrowed l-value, the same requirements as (EXPR-DREAD) applies with respect to modifying the owner of the l-value.

$$\overset{\text{(STAT-SEQUENCE)}}{\dfrac{\Gamma \vdash s; \Gamma' \quad \Gamma' \vdash s'; \Gamma''}{\Gamma \vdash s; s'; \Gamma''}} \qquad \overset{\text{(STAT-DECL)}}{\dfrac{\Gamma \vdash e : t \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash t\ x := e; \Gamma, x : t}}$$

Statements can be chained together in the obvious fashion. Local variable declaration and initialisation is straightforward.

***Good Effects Clause***

$$\overset{\text{(GOOD-EFFECT)}}{\dfrac{\forall p \in E.\, \Gamma \vdash p\texttt{+}\ \mathit{perm}}{\Gamma \vdash E}}$$

An effects clause is well-formed if it only revokes write permissions in the current environment.

***Good Environment***

$$\overset{\text{(GOOD-EMPTY)}}{\dfrac{}{\epsilon \vdash \diamond}} \qquad \overset{\text{(GOOD-R)}}{\dfrac{\Gamma \vdash q \\ p \notin \mathrm{dom}(\Gamma) \ \ \dagger \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\Gamma, p\dagger\,\mathsf{R}\,q \vdash \diamond}} \qquad \overset{\text{(GOOD-VARTYPE)}}{\dfrac{\Gamma \vdash t \\ x \notin \mathit{dom}(\Gamma)}{\Gamma, x : t \vdash \diamond}}$$

The rules for good environment require that owner variables are related to some owner already present in the environment or `world`, and that added variable bindings have types that are well-formed under the preceding environment.

***Good Permissions and Good Owner*** By (WORLD), `world` is a good owner and is always writable. By (GOOD-$\alpha$), a permission is good if it is in the environment. By (GOOD-$p$-), a read mode of objects owned by some owner $p$ is good if $p$ with any permission is a good permission—write or immutable implies read.

$$\overset{\text{(WORLD)}}{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{world+}\ \mathit{perm}}} \qquad \overset{\text{(GOOD-}\alpha\text{)}}{\dfrac{\Gamma \vdash \diamond \quad \alpha \in \mathit{dom}(\Gamma)}{\Gamma \vdash \alpha\ \mathit{perm}}}$$

$$\overset{\text{(GOOD-}p\text{-)}}{\dfrac{\Gamma \vdash p\dagger\ \mathit{perm} \quad \dagger \in \{\texttt{+}, \texttt{*}\}}{\Gamma \vdash p\texttt{-}\ \mathit{perm}}} \qquad \overset{\text{(GOOD-OWNER)}}{\dfrac{\Gamma \vdash \alpha\ \mathit{perm}}{\Gamma \vdash \mathsf{nm}(\alpha)}}$$

***Good Nesting*** We can easily define judgements $\Gamma \vdash p \prec^{*} q$ and $\Gamma \vdash p \prec^{+} q$ as the reflexive transitive closure and the transitive closure, respectively, of the relation generated from each $\alpha\,\mathsf{R}\,p \in \Gamma$, where $\mathsf{R} \in \{\prec^{*}, \prec^{+}\}$ or $\mathsf{R}^{-1} \in \{\prec^{*}, \prec^{+}\}$, combined with $p \prec^{*} \texttt{world}$ for all $p$.

***Good Type***

$$\overset{\text{(TYPE)}}{\dfrac{q\texttt{*} \in \overline{\alpha} \Rightarrow \sigma^{\circ}(q\texttt{*}) = p\texttt{*},\ \text{for some } p \\ \texttt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ \texttt{\{}\ \ldots\ \texttt{\}} \in P \\ \Gamma \vdash \sigma^p(\overline{\alpha\,\mathsf{R}\,p}) \quad \Gamma \vdash \sigma^{p\circ}(\overline{\alpha})\ \mathit{perm}}{\Gamma \vdash p\,c\langle\sigma\rangle}}$$

This rule checks that the owner parameters of the type satisfy the ordering declared in the class header, as well as checking that all the permissions are valid in the present context. In addition—and this is a subtle point—if an owner parameter in the class header was declared with the mode immutable, then the owner that instantiates the parameter in the type must also be immutable. Without this requirement, one could pass non-immutable objects where immutable objects are expected.

On the other hand, we allow parameters with read to be instantiated with write and *vice versa*. In the latter case, only methods that do not have write effects on the owners in question may be invoked on a receiver with the type in point.

$$\frac{\textsc{(unique-type)}\quad \Gamma \vdash p\, c\langle\sigma\rangle}{\Gamma \vdash \mathtt{unique}_p\, c\langle\sigma\rangle}$$

By (UNIQUE-TYPE), a unique type is well-formed if a non-unique type with the movement bound as owner is well-formed.

To simplify the formal account, we chose to make loss of uniqueness explicit using a movement operation rather than making it implicit via subtyping and subsumption, as such a rule would require a destructive read to be inserted. Instead, we require conversion to be explicit, as in the following rule:

$$\frac{\textsc{(expr-lose-uniqueness)}\quad \Gamma \vdash e : \mathtt{unique}_q\, c\langle\sigma\rangle \quad \Gamma \vdash p \prec^* q}{\Gamma \vdash (p)\ e : p\, c\langle\sigma\rangle}$$

This "owner-cast" expression moves the contents of a unique into a subheap of some object or block (whatever the $p$ owner corresponds to). This is well-formed if the expression has a unique type and if the movement bound of the type is outside the owner of the type cast to.

## 3.2 Brief Explanation of the System

In this section, we take a hands-on approach to showing how the system works by applying it to the example in Figure 1. For simplicity, we ignore everything that is not related to preserving read-only.

The key rules of the system are (METHOD), (EXPR-LVAL-DREAD), (STAT-FIELD-ASGN), (EXPR-INVOKE), and (STAT-BORROW).

In (METHOD), any write permissions revoked in the revocation clause $E$ are removed from $\Gamma$. Thus, the method body must be well-typed under a restricted $\Gamma$.

Destructively reading, borrowing or assigning to a field in an object, (EXPR-LVAL-DREAD), (STAT-BORROW) and (STAT-FIELD-ASGN) requires a write permission to the object containing the field in the current context.

Method invocation is a little trickier. If a formal owner parameter requires write access, or that an object is immutable, the calling context must satisfy those requirements (by $\Gamma \vdash \sigma^{\circ}(\overline{\alpha})\ \mathit{perm}$). Furthermore, the current context must also have write permission to every owner in the set of owners to which the method is allowed to write $(E_c \setminus E)$.

### 3.2.1 Type Checking Figure 1

By (CLASS), `addFirst()`, `filter()` and `getFirst()` must be well-formed under an environment $\Gamma = \mathtt{owner+} \prec^* \mathtt{world}, \mathtt{this+} \prec^+ \mathtt{owner}, \mathtt{data-} \succ^+ \mathtt{owner}, \mathtt{this} : \mathtt{owner}\, \mathtt{List}\langle\mathtt{data}\rangle$ for `List` to be a good class. By (METHOD), every statement in a method must be well-formed under $\Gamma'$ equal to $\Gamma$ extended by the formal parameters of the method and possible revocation of write rights. For `addFirst()`, $\Gamma' = \Gamma, \mathtt{obj} : \mathtt{data}\, \mathtt{Object}\langle\rangle$. We now look at the statements in `addFirst()`.

As we do not have constructors, (EXPR-NEW) does not care about permissions and is trivially well-formed with respect to write effects.

By (STAT-FIELD-ASGN), the second line of `addFirst()` requires that the owner of `tmp` is writeable under $\Gamma'$, *i.e.*, it is in $\mathrm{dom}(\Gamma')$. By (EXPR-FIELD), the type of `tmp` is $\mathtt{this}\, \mathtt{Link}\langle\mathtt{owner}\rangle$. As $\mathtt{this+} \in \mathrm{dom}(\Gamma')$, the field update is allowed. The next line follows the same pattern: reading a field is always allowed, and we have already established that we are allowed to assign to fields in `tmp`.

The last line of the method updates a field in `this`. By (STAT-FIELD-ASGN), `owner+` must be in $\mathrm{dom}(\Gamma')$, as the type of `this` is $\mathtt{owner}\, \mathtt{List}\langle\mathtt{data}\rangle$, which it is.

The method `filter()` type checks in a manner similar to that of `addFirst()`. However, `getFirst()` is different as it revokes the right to modify `owner` (and thus self). By (METHOD), the only line in `getFirst()` must type check under $\Gamma\, \mathsf{revoke}\, E$ where $E = \{\mathtt{owner}\}$. This is equivalent to $\mathtt{owner-} \prec^* \mathtt{world}, \mathtt{this-} \prec^+ \mathtt{owner}, \mathtt{data-} \succ^+ \mathtt{owner}, \mathtt{this} : \mathtt{owner}\, \mathtt{List}\langle\mathtt{data}\rangle$—the context has no write permissions. The field access is still allowed as reading fields does not require any write permissions.

### 3.2.2 Trapping Writes in a Read-Only Context

We now show how the system would trap an unpermitted write added to a method in the `List` class of Figure 1. Assume `Object` was defined thus:

```
class Object {
  this:Object state = null;
  void mutate() { this.state = null; }
}
```

and any of the methods in `List` included the line `this.first.data.mutate();`. $\Gamma$ is the same as in the previous section.

The key to trapping this violation of read-only is the 8th clause in (EXPR-INVOKE). By (EXPR-FIELD) (applied two times), the type of `this.first.data`, the receiver of the mutating message, is $\mathtt{data}\, \mathtt{Object}\langle\rangle$.

$E_{\mathtt{Object}} = \{\mathtt{owner}\}$ (remember $E_c$ returns the set of names of owners to which a class has write right) and $E = \epsilon$ as no rights are revoked. Consequently $E_{\mathtt{Object}} \backslash E = E_{\mathtt{Object}}$.

As `mutate` is not owner-polymorphic, $\sigma_a$ is empty and thus $\sigma_2 = \{\mathtt{owner} \mapsto \mathtt{data}\}$ and $\sigma_2(E_{\mathtt{Object}} \backslash E) = \{\mathtt{data}\}$.

Thus, by the 8th clause of (EXPR-INVOKE), $\Gamma \vdash \mathtt{data+}$ must hold. By (GOOD-$\alpha$), this amounts to $\mathtt{data+} \in \mathrm{dom}(\Gamma)$ which it clearly is not as we had $\mathtt{data-} \in \Gamma$ and `data-` and `data+` cannot occur simultaneously in $\Gamma$ when $\Gamma$ is well-formed.

Note that assignment to public fields is not allowed unless the receiver is `this`, which is why the modification had to be done through a method invocation.

## 3.3 Potentially Identical Owners with Different Modes

The list class in Figure 1 requires that the owner of its data objects is strictly outside the owner of the list itself. This allows for a clearer separation of the objects on the heap—for example, the list cannot contain itself.

The downside of defining the list class in this fashion is that it becomes impossible to store representation objects in a list that is also part of the representation. To allow that, the list class head must not use *strictly* outside:

```
class List< data- outside owner > { ... }
```

The less constraining nesting however leads to another problem: data and owner may be instantiated with the same owners. As data is read-only and owner is mutable, at face value, this might seem like a problem.

We choose to allow this situation as the context where the type appears might not care, or might have additional information to determine that the actual owners of data and the list do not overlap. If no such information is available, we could simply issue a warning. Of course, it is always possible to define different lists with different list heads for the two situations.

For immutables, this is actually a non-problem. The only way an immutable owner can be introduced into the system is through borrowing (or regions, see Section 4.2) where the immutable owner is ordered strictly inside any other known owner. As (TYPE) requires that write and immutable owner parameters are instantiated with owners that are write and immutable (respectively) in the context where the type appears, a situation where $p$+ and $q$* could refer to the same owner is impossible. As (TYPE) allows a read owner parameter to be instantiated with any mode, it is possible to have overlapping $p$- and $q$* in a context if a read owner was instantiated by an immutable at some point. Since objects owned by read owners will not be mutated, immutability holds.

### 3.4 Soundness of Joe$_3$

We have not formally proven soundness of Joe$_3$. Modulo omitting inheritance, the formal description of Joe$_3$ is a very simple and straightforward extension of that of Joline [32]. As modes have no run-time semantics, the crucial formal results of Joline should apply to Joe$_3$ as well — type soundness, owners as dominators and external uniqueness as dominating edges. Future work will extend the Joline formalism with modes and object-based regions and do the extra legwork to prove that the extended system is sound.

## 4. Extensions and Encodings

In this section we briefly discuss extensions to our system not included in the formalism, and the encoding of the modes from flexible alias protection.

### 4.1 Immutable Classes

In our system, an object always has permission to write to `owner` and `this` unless this permission is explicitly revoked in an effects clause for a specific method. Consequently, creating an immutable class requires every method to explicitly revoke its right to modify self. To relieve the programmer of this burden and to make a class' semantics clearer in the program text, we can introduce immutable classes through a class modifier:

```
immutable class String ...
```

The immutable class would be checked just as a regular class, but with the weaker permissions `owner*` and `this*` in $\Gamma$. Thus, methods that have write effects on `this` or `owner` would not type check. As fields may not be updated, except through `this`, this makes the object effectively immutable. To allow initialisation of immutable classes, the constructor would be allowed to initialise fields, similar to how final field initialisation is treated in Java.

### 4.2 Regions

In order to increase the precision of effects, we introduce explicitly declared regions, both at object-level and within method bodies. For simplicity, we have excluded regions from the formal account of the system. Object-based regions are similar to the regions of Greenhouse and Boyland [17] and the domains of Aldrich and Chambers [1], but we enable an ordering between them. Our method-scoped region construct is essentially the same as *scoped regions* in Joline [32], which is an object-oriented variant of classical regions [23, 30], adapted for use with ownership types.

***Object-based regions*** As discussed in Section 3.3, defining the list class without the use of *strictly* outside places the burden of determining whether data objects are modified by changes to the list on the client of the list. This is because the list cannot distinguish itself from its data objects, as they (potentially) have the same owner.

By virtue of owners-as-dominators, an object that needs to keep rep objects in a list must include the list in the rep, or the list will not have the necessary permissions to reference the objects. As the list owner and data owner are the same, modifications to the list are indistinguishable from modification to its contents.

To tackle this problem and make our system more expressive, we extend Joe$_3$ with a regions system. A class declaration can contain any number of regions that each introduce a new owner nested strictly inside an owner in the scope. Thus, a class' rep is divided into multiple, disjoint parts (except for nested regions), and an object owned by one region cannot be referenced by another. The syntax for regions is `region` $\alpha$ `{` $e$ `}`. Example:

```
class Example {
  this:Object datum;
  region inner+ strictly-inside this {
    inner:List<this> list;
  }

  void method() { list.add(datum); }
}
```

By virtue of the owner nesting, objects inside the region can be given permission to reference representation objects, but not vice versa as such types would not type check (*e.g.*, `this` is not inside `inner`). Thus, representation objects outside a region cannot reference objects in the region and consequently, effects on objects outside a region cannot propagate to objects in inside the region. In our example above, as there are no references from `datum` to the `list`, changes to `datum` cannot change the `list`.

***Method-scoped regions*** The *scoped regions* construct in Joline [32] can be added to Joe$_3$ to enable the construction of method-scoped regions, which introduces a new owner for a temporary scope within some method body. Scoped regions allow the creation of stack-local objects which can be mutated regardless of what other rights exist in the context, even when `this` is read-only or immutable. Such objects act as local scratch space without requiring that the effects propagate outwards. The effects can be ignored.

The following line illustrates a pattern that occurs several times in the implementation of the Joline compiler:

```
<d- inside world> void method(d:Something arg)
  revoke this {
  region temp+ strictly-inside this {
    temp:Gamma<d> t = new temp:Gamma<d>();
    t.calculationsWithSideEffectsOnTemp(arg);
  }
}
```

Several times in the Joline compiler, we create a temporary object reminiscent of the type environment ($\Gamma$) to check whether certain addtions of owner nestings would be permitted. This object is completely temporary and its sole purpose is throwing an exception on an attempt at adding invalid owner nestings.

### 4.3 Encoding Modes from Flexible Alias Protection

In work [27] that led to the invention of Ownership Types, Noble, Vitek and Potter suggested a set of modes on references to manage the effects of aliasing in object-oriented systems. The modes were *rep*, *free*, *var*, *arg* and *val*. In this section, we indicate how these modes are (partially) encoded in our system.

The *rep* mode denotes a reference to a representation object that should not be leaked outside of the object. All ownership type systems encode *rep*; in ours, it is encoded as `this` $c\langle\sigma\rangle$.

The *free* expression holds a reference that is uncaptured by any variable in the system. This is encoded as `unique`$_p$ $c\langle\sigma\rangle$, a unique type. Any l-value of that type in our system is (externally) free.

The *var* mode denotes a mutable non-rep reference and is encoded as $p\,c\langle\sigma\rangle$, where `this` $\neq p$.

The *arg* mode is the most interesting of the modes. It denotes an argument[3] reference with a guarantee that the underlying object will not be (observably) changed under foot: “that is, *arg* expressions only provide access to the immutable interface of the objects to which they refer. There are no restrictions upon the transfer or use of *arg* expressions around a program” [27]. We support *arg* modes in that we can parameterise a type by an immutable owner in any parameter. It is also possible for a class to declare all its owner parameters as immutable to prevent its instances from ever relying on a mutable argument object that could change under foot. On the other hand, we do not support passing *arg* objects around freely—the program must still respect owners-as-dominators.

The final mode, *val*, is like *arg*, but it is attached to references with value semantics. These are similar to our immutable classes.

## 5. Related Work

Boyland et al.’s Capabilities for sharing [9] generalise the concepts of uniqueness and immutability. The system uses capabilities, which are pointers combined with a set of rights. What really distinguishes this proposal from other work is the exclusive rights which allow the revocation of rights of other references. Boyland et al.’s system can model uniqueness with the ownership capability. However, exclusive rights make the system difficult to check statically.

Table 2 summarises several proposals and their supported features. The systems included in the table represent the state of the art of read-only and immutable. In addition to Joe$_3$, our own proposal, the table includes (in order) SafeJava [4], Universes [24, 16, 15, 25], Jimuva [18], Javari [31], IGJ [35], JAC [21, 20] and ModeJava [28, 29]. SafeJava is probably the closest in spirit to our proposal, but the lack of crucial features, such as borrowing to immutables, makes it less powerful. We now discuss the different features covered in the table.

***Expressiveness*** As discussed in Section 2.1.6, our system allows us to perform staged construction of immutable objects. This is also possible to do in SafeJava.

In our example in Figure 3, we show how we can encode fractional permissions [7]. Boyland suggests that copying rights may lead to observational exposure and proposes that the rights instead be split. Only the one with a complete set of rights may modify an object. SafeJava does not support borrowing to immutables and hence cannot model fractional permissions. It is unclear how allowing borrowing to immutables in SafeJava would affect the system, especially in the presence of inner classes which can break the owners-as-dominators property of deep ownership types.

In order to be able to retrieve writable objects from a read-only list, the elements in the list cannot be part of the list’s representation. Joe$_3$, Universes, Jimuva and SafeJava can express this in a straightforward fashion, by virtue of ownership types. However, only our system, because of owner nesting information, can have two non-sibling lists sharing mutable data elements. Javari and IGJ have taken a more ad hoc course introducing mutable fields. It is possible in those systems to circumvent read-only if an object stores a reference to itself (or an object that does so) in a mutable field.

| **Feature** | Joe$_3$ | SafeJava [4] | Universes [24, 16, 15] | Jimuva [18] | Javari [31] | IGJ [35] | ModeJava [28, 29] | JAC [21, 20] |
|---|---|---|---|---|---|---|---|---|
| *Expressiveness* | | | | | | | | |
| Staged constr. of immutables | √ | √ | × | × | × | × | × | × |
| Fractional permissions | √ | × | × | × | × | × | × | × |
| Non rep fields | √ | √[1] | √[1] | √[1] | ×[2] | ×[2] | × | × |
| *Flexible Alias Protection Modes* | | | | | | | | |
| *arg* | √[3] | ×[4] | × | ×[4] | ×[4] | ×[4] | ×[4] | × |
| *rep* | √ | √ | √ | √ | × | × | × | × |
| *free* | √ | √ | √[9] | × | × | × | × | × |
| *val* [5] | × | × | × | × | × | × | × | × |
| *var* | √ | √ | √ | √ | √ | √ | √ | √ |
| *Immutability* | | | | | | | | |
| Class immutability | √ | × | × | √ | √ | √ | ×[6] | ×[6] |
| Object immutability | √ | √ | × | √ | × | √ | × | × |
| Read-only references | √ | × | √ | × | √ | √ | √ | √ |
| Context-based immutability | √ | × | √ | × | ×[7] | ×[7] | ×[7] | × |
| *Confinement and Alias Control* | | | | | | | | |
| Ownership types | √ | √ | √ | √ | × | × | × | × |
| Owner-polymorphic methods | √ | √ | √ | √ | × | × | × | × |
| Owners-as-modifiers | × | ×[8] | √ | × | × | × | × | × |
| Unique references | √ | √ | √ | × | × | × | × | × |

**Table 2.** Brief overview of related work. [1]) not as powerful as there is no owner nesting; two non sibling lists cannot share *mutable* data elements; [2]) mutable fields can be used to store a reference to `this` and break read-only; [3]) see Section 4.3; [4]) no modes on owners, and hence no immutable parts of objects; [5]) none of the systems deal with value semantics for complex objects; [6]) if all methods of a class are read-only the class is effectively immutable; [7]) limited notion of contexts via `this`-mutability; [8]) allows breaking of owners-as-dominators with inner classes and it is unclear how this interplays with immutables; [9]) support is forthcoming [25].

***Flexible Alias Protection Modes*** The five alias modes proposed by Noble et al [27] were discussed in Section 4.3, where we also describe how these can be (partially) encoded in our system. Here we only describe how the modes have been interpreted for the purpose of the table (Table 2). The *rep* mode denotes a reference belonging to the representation of an object and should not be present in the interface. A defensive interpretation of *arg* is that all systems that have object or class immutability partially support *arg*, but only our system can have parts of an object being immutable. The *free* aliasing mode, interpreted as being equal to uniqueness, is supported by our system and SafeJava. None of the systems handle value semantics for complex objects and thus not the *val* mode (even though Javari include Java’s primitive types in their system). The *var* aliasing mode expresses non-*rep* references which may be aliased and changed freely as long as they do not interfere with the other modes, for example, in assignments.

***Immutability*** Immutability takes on three forms in *class immutability*, where no instance of a specific class can be mutable, *object immutability*, where no reference to a specific object can be mutable and *read-only* or *reference immutability*, where there may be both mutable and read-only aliases to a specific object.

Universes and our system provide what we call context-based immutability. In these two systems it is possible to create a writable list with writable elements and pass it to some other object to whom the elements are read-only. This other object may add new elements to the list which will be writable by the original creator of the list. The other systems in our table do not support this as they cannot allow *e.g.*, a list of writeables to be subsumed into a list of read-only references. In these systems, this practice could lead to standard covariance problems—adding a supertype to a list

[3] An object external to another object.

containing a subtype. Javari, IGJ and ModeJava all have a notion of this-mutable fields which inherit the mode of the accessing reference. This counts for some notion of context, albeit an ad hoc and inflexible one. In ModeJava a read-only list cannot return writable elements. In Javari and IGJ, this is only possible if the elements are stored in mutable fields, which causes other problems as discussed above.

***Confinement and Alias Control*** Joe$_3$, SafeJava, Universes and Jimuva all support ownership types. This is what gives Joe$_3$ and Universes its context-based immutability. SafeJava and Jimuva, despite having ownership types, do not have context-based immutability due to their lack of read-only references. Universes is the only system supporting the owners-as-mutators property, meaning that representation exposure is allowed for read-only references.

Other approaches to confinement and alias control include Confined Types [3, 34], which constrain access to objects to from within their *package*. Bierhoff and Aldrich recently proposed a modular protocol checking approach [2] based on typestates. They partly implement Boyland's fractional permissions [7] in their access permissions.

***Object-Oriented Regions and Effects systems*** Leino [22], Greenhouse and Boyland [17] and (to some degree) Aldrich and Chambers [1] take a similar approach to dividing objects into regions, and using method annotations to specify which parts of an object may be modified by a specific method. Adding effects to ownership, a la Clarke and Drossopoulou's Joe$_1$ [11], gives a stronger notion of encapsulation and enables more accurate description of effects. The addition of scoped regions to our system (*c.f.*, Section 4.2), combines both of these approaches.

Effect systems were first studied by Lucassen and Gifford [23] and Talpin and Jouvelot [30].

## 6. Future Work

### 6.1 Safe Representation Exposure

Müller and Poetzsch-Heffter's Universe system [24] allows representation exposure in a safe way (Boyland might disagree)—an object's representation can be exposed outside the object, but only via read-only references.

Extending our system to allow rep exposure for non-mutables will probably require an additional type that may only appear in contexts where its owner is read-only or immutable, but allows any valid owner in scope to be used as the owner of the type. This is a direction for future work.

### 6.2 Inheritance

Extending our system with inheritance is one of the next directions this research will take. We believe this to be a straightforward extension. Ownership, uniqueness and owner-polymorphic methods are already shown to work in the presence of inheritance [32], subtyping and downcasts [5, 33].

In the simplest way, for our Joe$_3$-specific extensions, an overriding method must revoke (at least) the same rights as the method it overrides, and argument and parameter types must be invariant and modes on owners must be preserved in subclasses.

## 7. Concluding Remarks

We have proposed Joe$_3$, an extension of Joe$_1$ and Joline with access modes on owners that can encode class, object and reference immutability, fractional permissions and context-based ownership with surprisingly little syntactical overhead. Future work will see a complete formalisation of the system, extended with inheritance and regions, including a dynamic semantics, and appropriate immutability invariants and soundness proofs.

## References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, Jan 2004. Springer Verlag.

[2] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. Submitted to OOPSLA 2007.

[3] Boris Bokowski and Jan Vitek. Confined Types. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1999.

[4] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Electrical Engineering and Computer Science, MIT, February 2004.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. In Dave Clarke, editor, *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, UU-CS-2003-030. Utrecht University, July 2003.

[6] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.

[7] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[8] John Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, June 2006. Special issue: ECOOP 2005 Workshop FTfJP.

[9] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, June 2001.

[10] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[11] David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, November 2002.

[12] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1998.

[13] David Clarke and Tobias Wrigstad. External uniqueness. In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA, January 2003.

[14] David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt, Germany, 2003.

[15] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[16] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[17] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[18] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. Immutable objects for a Java-like language. In Rocco De Nicola, editor, *16th European Symposium on Programming (ESOP'07)*,

volume 4421 of *LNCS*, pages 347–362. Springer, March 2007. Won the ETAPS award for the best theory paper at ETAPS.

[19] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, November 1991.

[20] Günter Kniesel and Dirk Theisen. JAC – Java with transitive readonly access control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems*, At ECOOP'99, Lisbon, Portugal, June 1999.

[21] Günter Kniesel and Dirk Theisen. JAC—access right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, May 2001.

[22] K. Rustan M. Leino. Data Groups: Specifying the Modification of Extended State. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1998.

[23] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[24] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[25] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. To appear.

[26] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[27] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1998. Springer-Verlag.

[28] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *Formal Techniques for Java Programs, in Conjunction with ECOOP 2001*, Budapest, Hungary, 2001.

[29] Mats Skoglund and Tobias Wrigstad. Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics*, March 2002.

[30] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[31] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[32] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Department of Computer and Systems Science, Royal Institute of Technology, Kista, Stockholm, May 2006.

[33] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4):141–159, May–June 2007. Available from http://www.jot.fm/issues/issue_2007_03/article5.

[34] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. In *Journal of Functional Programming*, volume 15(6), pages 1–46, 2005.

[35] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 16, 2007.