Ownership, Uniqueness, and Immutability

Johan Östlund¹, Tobias Wrigstad¹, Dave Clarke², and Beatrice Åkerblom³

¹ Purdue University, West Lafayette, Indiana, USA

² CWI, Amsterdam, The Netherlands

³ Stockholm University, Stockholm, Sweden

Abstract. Programming in an object-oriented language demands a fine balance between flexibility and control. At one level, objects need to interact freely to achieve our implementation goals. At a higher level, architectural constraints that ensure the system can be understood by new developers and can evolve as requirements change must be met. To resolve this tension, researchers have developed type systems expressing ownership and behavioural restrictions such as immutability. This work reports on our consolidation of the resulting discoveries into a single programming language. Our language, `Joe3`, 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 [11] 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. Ownership types captures and enforces object nesting statically in the program code by parameterising types over the enclosing objects. Read-only references prevent mutation and have been used in proposals to strengthen object encapsulation and manage aliasing either on its own (e.g., [19,27]) or as parts of other proposals (e.g., [26,18,24]). An immutable object can be thought of as an object to which only read-only references exist. Immutable objects help avoid aliasing problems and data races in multi-threaded programs [4,17], and also enhance program understanding, as read-only or immutable annotations are verified to hold at compile-time [29]. According to Zibin et al. [32], 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.

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.

Contributions. The programming language, `Joe3`, proposed in this paper offers ownership and externally unique references (the only external reference into an aggregate) 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 extension¹ of Clarke and Wrigstad’s *external uniqueness* proposal (Joline) [13,30], 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 [26], which allows aggregates to safely rely on immutable parts of external objects, Fractional Permissions [6], which allows an object to mediate between being unaliased and mutable and aliased and immutable, and the context-based immutability of Universes [24], all the while preserving the owners-as-dominators encapsulation invariant of ownership types which guarantees that all accesses to an object’s internals go via its interface. Furthermore, as our system is based on ownership types, we can distinguish between outgoing aliases to external, non-representation 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.

The Status of Read-Only in the Literature. Boyland [7] 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, so the reference can be safely stored;
 - (b) a read-only reference is unique and thus effectively immutable;
 - (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*, which occurs when changes to state are observed through a read-only reference.

`Joe3` addresses all of these problems. First, `Joe3` 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, `Joe3` supports fractional permissions [6]—converting a mutable unique reference into several immutable references for a certain context. This allows safe representation exposure without the risk for observational exposure.

`Joe3` allows class, object and reference immutability. Unique references, borrowing and owner-polymorphic methods allow us to simulate fractional permissions and staged, external initialisation of immutable objects through auxiliary

¹ For brevity, we omit inheritance. Our `Joe3` compiler however supports it.

methods. As we base modification rights on owners (in the spirit of Joe₁'s [10] effects system), we achieve what we call *context-based* immutability, which is essentially the same kind of read-only found the Universes system [24], a simple ownership system enforcing deep encapsulation modulo read-only references.

Joe₃ 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.

Outline. Section 2 introduces the Joe₃ language through a set of motivating examples. Section 3 gives a brief formal account of Joe₃. Section 4 outlines a few simple but important extensions—immutable classes and Greenhouse and Boyland-style regions [16], describes how they further enhance the system and discusses how to encode the modes of Flexible Alias Protection [26]. Section 5 surveys related work not covered in the previous. Section 6 concludes.

2 Meet Joe₃

In this section we describe Joe₃ using some motivating examples. A more detailed introduction is given in Section 3. Joe₃ is a class-based, object-oriented programming language with deep ownership, owner-polymorphic methods, ownership transfer through external uniqueness, an effects system, and a simple mode system decorating owners with permissions to control how an object may be modified. Beyond the carefully designed combination of features, the annotation of owners with modes is the main novelty in Joe₃. The modes indicate that a reference may be read or written (+), only read (−), or 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 Joe₃ is a simple effects system, expressed through owners, inspired by Clarke and Drossopoulou's Joe₁ [10]. Classes and objects have the rights to read or modify objects belonging to certain owners; only a minor extension to the type system of Clarke and Wrigstad's Joline [13,30] is required to ensure this statically.

The syntax of Joe₃ (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. 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 instantiating the owner parameters. For example, the *type* *this*:*List*<*owner*> denotes an object belonging to the representation of the current *this* and has the right to reference objects owned by *owner*. There are two nesting relations between owners: inside and outside. They

each exist in two forms: one reflexive (*inside/outside*) and one non-reflexive (*strictlyinside/strictlyoutside*). Going back to our list example, a type `this:List<this>` denotes a list object belonging to the current representation (first *this*), storing objects of the current representation (second *this*).

Apart from ownership types, the key ingredients in `Joe3` are the following:

- unique types (written `unique:Object2`), 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 formal owner parameters of classes and methods, though not on types.
- an effects revocation clause on methods that 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.

2.1 Motivating Examples

A Mutable List with Readonly Content. 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 list 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 from 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;
- the list is mutable but the data objects are not, which imposes no additional restrictions, `getFirst()` returns a read-only reference; and
- the data objects are mutable, but the list not, which only allows `getFirst()` to be invoked, and 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 [18], JAC [19], ModeJava [27], Javari [29], and IGJ [32]) cannot express this constraint in a satisfactory way, as these proposals cannot distinguish between an object’s outside and inside.

² Sometimes we use `unique[p]:Object` where the owner `p` acts as a bound for ownership transfer. Uniqueness is *external* [13] meaning we permit *internal* aliases to the unique object.

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 [24,23]. In contrast, however, we do not allow representation exposure via read-only references.

```

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

class List<data- strictlyoutside 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;
    }
}

```

Fig. 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.

```

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

class Reader<o- outside owner, data+ strictlyoutside 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);
  }
}

```

Fig. 2. Different views of the same list can exist 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.

Borrowing Blocks and Owner-polymorphic Methods. Before moving on to the last two examples, we need to introduce borrowing blocks and owner-polymorphic methods [12,30,9], 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 [13,30].) A borrowing block has the following syntax:

$$\text{borrow } lval \text{ as } \alpha x \text{ in } \{ s \}$$

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 block *s*. The block also introduces a fresh stack-local block-scoped 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. As the fresh owner is introduced on the stack, the only way to export it is by passing the owner around as an *owner parameter* (see below). If exported, the owner is still confined to the stack and as all method calls in the borrowing block must have returned when the block exits, the exported permission is invalidated so no residual aliasing can exist. Thus, when the borrowing block exits, the borrowed value can be reinstated and is once again unique.

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

```

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); // ****
    }
  }
}

```

Fig. 3. Fractional permissions using borrowing and unique references

reference. Owner parameters (p^* and p^- in the methods in Figure 3) of 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* [5]).

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 *unique* reference into a variable with immutable type, just as in SafeJava [4]. This allows us to encode fractional permissions using our borrowing construct and do staged construction of immutables.

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 [6]. 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” [6] as we know for sure that all permissions to alias

```

class Client<p* outside owner, data+ strictlyoutside 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+ strictlyoutside p> {
  p:List<data> createList() {
    unique[p]:List<data> list = new p:List<data>();
    borrow list as temp+ 1 in { // 2nd stage of construct.
      1.add(new data:Object());
    }
    return list--; // unique reference returned
  }
}

```

Fig. 4. Staged construction of an immutable list

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.

Initialisation of Immutable Objects. Immutable objects need to be mutated 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 [29,32,17]. Continuing initialisation by calling auxiliary methods *after* the constructor returns is simply not possible. `Joe3`, 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.

Annotating owners at the level of references rather than types 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 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

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

Fig. 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.

is statically checkable in a modular fashion and no run-time representation of modes or owners is required.

3 A Formal Definition of Joe_3

In this section, we formally present the static semantics of Joe_3 .

3.1 Joe_3 's Static Semantics

Joe_3 's type system is a simplification of Joline's [13,30] extended with effects and modes on owners. To simplify the formal account, we omit inheritance and constructors. Both are straightforward, and are supported by our Joe_3 compiler. Furthermore, following Joline, we rely on destructive reads to preserve uniqueness and require that ownership transfer is performed using an explicit operation.

Figure 5 shows Joe_3 's syntax. c, m, f, x are metavariables ranging over names of uniquely named classes, methods, fields and local variables, respectively. Types have the syntax $pc(\overline{p})$ and we use q and p for owner names. We sometimes write $pc(\sigma)$ for some type where σ 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 $\cdot$.

Unique types have the syntax $\text{unique}_p pc(\overline{p})$. 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.

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 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., [16,10]), we use effect annotations on methods to show what is *not* affected by the method—essentially *temporarily revoking* rights. For example, `getFirst()` in the list in Figure 1 does not modify the list object and is thus declared using a *revoke* clause:

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

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

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. $\text{CT}(c)(f) = t$ means that field f in class c has type t . $\text{CT}(c)(m) = \forall \overline{\alpha R q}. \overline{t} \rightarrow t; E$ means that method m in class c have formal owner-parameters declared $\overline{\alpha R q}$, formal parameter types $\overline{t}$, return type t and set of revoked rights E . $E \setminus E'$ denotes set difference.

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

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

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

Good Class. A class is well-formed if all its owner parameters are outside `owner`. This ensures 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 [9]. The environment Γ 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 Γ , and must be respected by field declarations and methods.

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

Good Field, Good Method. The function $\Gamma \text{ rev } E$ is a key player in our system—it revokes the write rights mentioned in E , by converting them to read rights in Γ . It also makes sure that this is not writable whenever owner is not. For ex., given $E = \{p\}$, if $\Gamma \text{ rev } E \vdash s; \Gamma'$, s cannot write to objects owned by p .

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

$$\begin{array}{c} \text{(FIELD)} \\ \frac{\Gamma \vdash e : t}{\Gamma \vdash t f := e} \end{array} \quad \begin{array}{c} \text{(METHOD)} \\ \frac{\Gamma' = \Gamma, \overline{\alpha \text{ R } p} \quad \Gamma' \vdash E \quad (\Gamma' \text{ rev } E), \overline{x : t} \vdash s; \Gamma'' \quad \Gamma'' \vdash e : t}{\Gamma \vdash \langle \overline{\alpha \text{ R } p} \rangle t m(\overline{t x}) \text{ revoke } E \{ s; \text{return } e \}} \end{array}$$

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

Expressions. The expression rules extend Joline’s semantics with permissions.

$$\begin{array}{c} \text{(EXPR-LVAL)} \\ \frac{\Gamma \vdash_{\text{lv}} \text{lval} : t \quad \neg \text{isunique}(t)}{\Gamma \vdash \text{lval} : t} \end{array} \quad \begin{array}{c} \text{(EXPR-LVAL-DREAD)} \\ \frac{\Gamma \vdash_{\text{lv}} \text{lval} : t \quad \text{isunique}(t) \quad \text{lval} \equiv e.f \Rightarrow \Gamma \vdash e : p c(\sigma) \wedge \Gamma \vdash p+ \text{ perm}}{\Gamma \vdash \text{lval}-- : t} \end{array}$$

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

$$\begin{array}{c} \text{(EXPR-VAR)} \\ \frac{x : t \in \Gamma}{\Gamma \vdash_{\text{lv}} x : t} \end{array} \quad \begin{array}{c} \text{(EXPR-FIELD)} \\ \frac{\Gamma \vdash e : p c(\sigma) \quad \text{CT}(c)(f) = t \quad \text{this} \in \text{owners}(t) \Rightarrow e \equiv \text{this}}{\Gamma \vdash_{\text{lv}} e.f : \sigma^p(t)} \end{array}$$

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

In Joline, owner arguments to owner-polymorphic methods must be passed in explicitly. Here, we assume an inference algorithm exists to bind the names of the owner parameters to the actual arguments, σ_a below, at the call site.

$$\begin{array}{c} \text{(EXPR-INVOKE)} \\ \frac{\Gamma \vdash e : p c(\sigma) \quad \text{CT}(c)(m) = \forall \overline{\alpha \text{ R } p}. \bar{t} \rightarrow t; E \quad \sigma' = \sigma^p \uplus \sigma_a \quad \Gamma \vdash \sigma'(\overline{\alpha \text{ R } p}) \quad \Gamma \vdash \sigma'^o(\overline{\alpha}) \text{ perm} \quad \Gamma \vdash \bar{e} : \sigma'(\bar{t}) \quad \Gamma \vdash \sigma'(t) \quad \Gamma \vdash \sigma'(E_c \setminus E) \quad \text{this} \in \text{owners}(\text{CT}(c)(m)) \Rightarrow e \equiv \text{this}}{\Gamma \vdash e.m(\bar{e}) : t} \end{array}$$

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

(σ^p) and the implicit mapping from owner parameter to actual owner (σ_a). $\Gamma \vdash \sigma'^o(\bar{\alpha}) \text{ perm}$ 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 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 [11].

$$\frac{\text{(EXPR-NULL)} \quad \Gamma \vdash t}{\Gamma \vdash \text{null} : t} \quad \frac{\text{(EXPR-NEW)} \quad \Gamma \vdash p \, c\langle \bar{p} \rangle}{\Gamma \vdash \text{new } p \, c\langle \bar{p} \rangle : \text{unique}_p c\langle \bar{p} \rangle}$$

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 [30] for an explanation why adding constructors is not a problem.)

Good Statements

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

Assigning to a field requires write permission to the object containing the field.

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

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.

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

Good Effects Clause revokes write permissions from the current environment.

$$\frac{(\text{GOOD-EFFECT}) \quad \forall p \in E. \Gamma \vdash p+ \text{ perm}}{\Gamma \vdash E}$$

Good Environment. 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, as usual.

$$\frac{(\text{GOOD-EMPTY})}{\epsilon \vdash \diamond} \quad \frac{(\text{GOOD-R}) \quad \Gamma \vdash q \quad p \notin \text{dom}(\Gamma) \quad \dagger \in \{+, -, *\}}{\Gamma, p\dagger Rq \vdash \diamond} \quad \frac{(\text{GOOD-VARTYPE}) \quad \Gamma \vdash t \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : t \vdash \diamond}$$

Good Permissions and Good Owner. By (WORLD), `world` is a good owner and is always writable. By (GOOD- α), 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.

$$\frac{(\text{WORLD}) \quad \Gamma \vdash \diamond}{\Gamma \vdash \text{world}+ \text{ perm}} \quad \frac{(\text{GOOD-}\alpha) \quad \Gamma \vdash \diamond \quad \alpha \in \text{dom}(\Gamma)}{\Gamma \vdash \alpha \text{ perm}} \quad \frac{(\text{GOOD-}p-) \quad \Gamma \vdash p\dagger \text{ perm} \quad \dagger \in \{+, *\}}{\Gamma \vdash p- \text{ perm}} \quad \frac{(\text{GOOD-OWNER}) \quad \Gamma \vdash \alpha \text{ perm}}{\Gamma \vdash \text{nm}(\alpha)}$$

Good Nesting. Judgements $\Gamma \vdash p \prec^* q$ and $\Gamma \vdash p \prec^+ q$ are the reflexive transitive closure and the transitive closure, respectively, of the relation generated from each $\alpha R p \in \Gamma$, where $R, R^{-1} \in \{\prec^*, \prec^+\}$, combined with $p \prec^* \text{world}$ for all p .

Good Type. This rule checks that the owner parameters of the type satisfy the ordering declared in the class header, and that all the permissions are valid in the present context. In addition, 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.

$$\frac{(\text{TYPE}) \quad q* \in \bar{\alpha} \Rightarrow \sigma^\circ(q*) = p*, \text{ for some } p \quad \text{class } c(\overline{\alpha R p}) \{ \dots \} \in P \quad \Gamma \vdash \sigma^p(\overline{\alpha R p}) \quad \Gamma \vdash \sigma^{p^\circ}(\bar{\alpha}) \text{ perm}}{\Gamma \vdash p c\langle \sigma \rangle}$$

$$\frac{(\text{UNIQUE-TYPE}) \quad \Gamma \vdash p c\langle \sigma \rangle}{\Gamma \vdash \text{unique}_p c\langle \sigma \rangle} \quad \frac{(\text{EXPR-LOSE-UNIQUENESS}) \quad \Gamma \vdash e : \text{unique}_q c\langle \sigma \rangle \quad \Gamma \vdash p \prec^* q}{\Gamma \vdash (p) e : p c\langle \sigma \rangle}$$

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 cast-like operation in (EXPR-LOSE-UNIQUENESS). Allowing it to be implicit via subtyping and subsumption would require a destructive read to be inserted. The (EXPR-LVAL) rules ensure that the e is really free, not simply unique.

3.2 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 4.1) 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.

3.3 Invariants and Meta Theory

Due to space restriction, our formalisation of Joe_3 does not include a dynamic semantics, which makes formal invariants difficult to state. However, the formal description of Joe_3 is a small extension of Joline [30] and as modes have no run-time semantics, adapting the soundness results and encapsulation and uniqueness invariants of Joline is relatively straightforward. Here we state our invariants informally and defer their formal definitions and proof to future work:

For immutables, the invariant Joe_3 enforces is that an object referenced by an immutable reference cannot change as a result of evaluating any statement.

For read-only references, Joe_3 guarantees that evaluating a statement can only result in changes to a read-only referenced object if there existed a mutable alias to the object reachable by the statement.

The important encapsulation properties of Joline that carry over to Joe₃ are owners-as-dominators, which says that accesses to an object’s internals must go via its interface and external-uniqueness-as-dominating-edges which says that any path from a root to a unique object’s internals must go via the unique reference to the object.

3.4 Joe₃ Compiler

We have implemented Joe₃ as an extension of our Joline compiler, using the Polyglot compiler framework. The compiler has been tested on moderately sized programs with promising results. Future work will involve further evaluation using larger programs.

4 Extensions and Encodings

4.1 Object-Based Regions and Method-Scoped Regions

To increase the precision of effects, we introduce explicitly declared regions, both at object-level and within method bodies. For simplicity, these are excluded from the formal account of the system. Object-based regions are similar to the regions of Greenhouse and Boyland [16] 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 [30], which is an object-oriented variant of classical regions [22,28], adapted to ownership types.

Object-based regions. As discussed in Section 3.2, 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 might have the same owner. As the (static) 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 Joe₃ with regions that introduce a new owner nested strictly inside an owner in the scope. Thus, a class’ representation 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  $\alpha$  {  $e$  }`. For example:

```
class Example {
  this:Object datum;
  region inner+ strictlyinside this { inner:List<this> list; }
  void method() { list.add(datum); }
}
```

Objects inside the region can be given permission to reference representation objects, but not vice versa (*e.g.*, `this` is not inside `inner`). Thus, `rep` objects outside a region cannot reference objects in the region, so effects on objects outside a region cannot propagate to objects inside it. Above, changes to `datum` cannot change `list`, as there are no references from `datum` to the `list`.

Method-scoped regions. The *scoped regions* construct in Joline [30] can be added to Joe₃ 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. This pattern occurred frequently in the implementation of the Joline compiler. Our compiler supports method-scoped regions thusly:

```
<d- inside world> void method(d:Something arg) revoke this {
    region temp+ strictlyinside this { ... }
}
```

4.2 Encoding Modes from Flexible Alias Protection

In work [26] 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⟨σ⟩`. The *free* expression holds a reference that is uncaptured by any variable in the system. This is encoded as `uniquep c⟨σ⟩`, 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⟨σ⟩`, where `this ≠ p`.

The *arg* mode is the most interesting of the modes. It denotes a reference to an external object with a guarantee that the underlying object will not be 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” [26]. 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 [8] generalise the concepts of uniqueness and immutability through capabilities. Their outstanding feature 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 but exclusive rights make the system difficult to check statically.

Table 1. Brief overview of related work. OT=ownership types, OP=owner polymorphic, OAM=owners as modifiers, EO=Effective Ownership. 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 4.2; 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.

Feature	Joe ₃	SafeJava	Universes	Jimuva	Javari	IGJ	ModeJava	JAC	EO	Feature	Joe ₃	SafeJava	Universes	Jimuva	Javari	IGJ	ModeJava	JAC	EO	
	Expressiveness										Immutability									
Staged const.	✓	✓	×	×	×	×	×	×	×	Class	✓	×	×	✓	✓	✓	×	6	6	×
Fract. perm.	✓	×	×	×	×	×	×	×	×	Object	✓	✓	×	✓	×	✓	×	×	×	
Non-rep fields	✓	✓ ¹	✓ ¹	✓ ¹	×	2	×	2	×	Context-based	✓	×	✓	×	×	7	×	7	×	×
										Read-only ref.	✓	×	✓	×	✓	✓	✓	✓	✓	
	Flexible Alias Protection Modes										Confinement and Alias Control									
arg	✓ ³	×	4	×	×	4	×	4	×	OT	✓	✓	✓	✓	×	×	×	×	✓	
rep	✓	✓	✓	✓	×	×	×	×	✓	OP meths	✓	✓	✓	✓	×	×	×	×	×	
free	✓	✓	✓	×	×	×	×	×	×	OAM	×	×	8	✓	×	×	×	×	✓	
val ⁵	×	×	×	×	×	×	×	×	×	Unique	✓	✓	✓	×	×	×	×	×	×	
var	✓	✓	✓	✓	✓	✓	✓	✓	✓											

Table 1 summarises several proposals and their supported features. The systems included in the table represent the state of the art of read-only and immutable. Except Joe₃, the table includes (in order) SafeJava [4], Universes [24,15,14,25], Jimuva [17], Javari [29], IGJ [32], JAC [19], ModeJava [27] and Effective Ownership [21]. We now discuss the different features in the table.

Expressiveness. Joe₃ and SafeJava support staged construction of immutables.

Figure 3 shows how we can encode fractional permissions [6]. 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 back doors that break encapsulation.

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. Joe₃, Universes, Jimuva and SafeJava can express this through ownership types. Only our system, thanks to owner nesting information, allow two non-sibling lists sharing mutable data elements. Javari and IGJ allow this through ad-hoc mutable fields which can circumvent read-only if an object stores a reference to itself in a mutable field.

Flexible Alias Protection Modes (also see Section 4.2). The *rep* mode denotes a reference belonging to an object's representation and so should not appear in its interface. A defensive interpretation of *arg* is that all systems that have object or class immutability partially support *arg*, but only our system support *parts* of an object being immutable. The *free* mode, interpreted as being equal to uniqueness, is supported by Joe₃ and SafeJava. No system handles *value* semantics except for primitive types. 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: *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. Explicit class immutability could easily be added to Joe₃ by introducing an immutable class annotation that would revoke owner and this.

Universes and Joe₃ provide what we call context-based immutability. Here it is possible to create a writable list with writable elements and pass it to some other context where the elements are read-only. This other context can add 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 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 ad hoc and inflexible. In ModeJava a read-only list cannot return mutable elements. Javari and IGJ support this for elements are stored mutable fields, which opens the back door mentioned above.

Confinement and Alias Control. Joe₃, SafeJava, Universes and Jimuva all support ownership types. This is what gives Joe₃ 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 [3,31], which constrain access to objects to from within their *package*. Bierhoff and Aldrich recently proposed a modular protocol checking approach [2] based on tpestates. They partly implement Boyland's fractional permissions [6] in their access permissions.

Object-Oriented Regions and Effects systems. Effect systems were first studied by Lucassen and Gifford [22] and Talpin and Jouvelot [28]. Leino [20], Greenhouse and Boyland [16] 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, à la Clarke and Drossopoulou's Joe₁ [10], gives a stronger notion of encapsulation and enables more accurate description of

effects. The addition of scoped regions to our system (*cf.* §4.1), combines both of these approaches.

Effective Ownership. Lu and Potter's [21] work on Effective Ownership is similar in spirit to our work, especially wrt. effects on methods. Their goal is making ownership types more flexible, whereas we aim to express immutable and read-only idioms in an ownership types setting. Consequently, their system cannot express two objects owned by the same owner where only one is mutable nor context-based immutability, nor do they consider immutability or uniqueness.

6 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. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Malenfant, J., Østvold, B.M. (eds.) ECOOP 2004. LNCS, vol. 3344. Springer, Heidelberg (2005)
2. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA (2007)
3. Bokowski, B., Vitek, J.: Confined Types. In: OOPSLA (1999)
4. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. PhD thesis, Electrical Engineering and Computer Science. MIT (February 2004)
5. Boyland, J.: Alias burying: Unique variables without destructive reads. *Software — Practice and Experience* (2001)
6. Boyland, J.: Checking interference with fractional permissions. In: *Static Analysis: 10th International Symposium* (2003)
7. Boyland, J.: Why we should not add readonly to Java (yet). *Journal of Object Technology* (2006); Special issue: ECOOP 2005 Workshop FTfJP
8. Boyland, J., Noble, J., Retert, W.: Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072. Springer, Heidelberg (2001)
9. Clarke, D.: Object Ownership and Containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia (2001)
10. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA (2002)
11. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA (1998)
12. Clarke, D., Wrigstad, T.: External uniqueness. In: FOOL (2003)
13. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743. Springer, Heidelberg (2003)

14. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)
15. Dietl, W., Müller, P.: Universes: Lightweight Ownership for JML. *Journal of Object Technology* 4(8), 5–32 (2005)
16. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628. Springer, Heidelberg (1999)
17. Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable Objects for a Java-like Language. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421. Springer, Heidelberg (2007)
18. Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: OOPSLA (November 1991)
19. Kiesel, G., Theisen, D.: JAC—access right based encapsulation for Java. *Software — Practice and Experience* (2001)
20. Leino, K.R.M.: Data Groups: Specifying the Modification of Extended State. In: OOPSLA (1998)
21. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL (2006)
22. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL (1988)
23. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
24. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. Technical report, Fernuniversität Hagen (1999)
25. Müller, P., Rudich, A.: Ownership transfer in Universe Types. In: OOPSLA (2007)
26. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445. Springer, Heidelberg (1998)
27. Skoglund, M., Wrigstad, T.: Alias control with read-only references. In: Sixth Conference on Computer Science and Informatics (March 2002)
28. Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. *Journal of Functional Programming* (1992)
29. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: OOPSLA (2005)
30. Wrigstad, T.: Ownership-Based Alias Management. PhD thesis, Royal Institute of Technology, Kista, Stockholm (May 2006)
31. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. *Journal of Functional Programming* 15(6) (2005)
32. Zibin, Y., Potanin, A., Artzi, S., Kiežun, A., Ernst, M.D.: Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MITCSAIL (2007)