

Comprehending Annotations on Object-Oriented Programs using Fractional Permissions

John Boyland
University of
Wisconsin-Milwaukee, USA
boyland@cs.uwm.edu

William Retert
University of
Wisconsin-Milwaukee, USA
williamr@cs.uwm.edu

Yang Zhao
Nanjing University of Science
and Technology, China
yangzhao@cs.njust.edu.cn

ABSTRACT

Fractional permissions are a general system for managing access to mutable state. We show how fractional permissions can give semantics to a regimen of annotations including “unique,” “non-null,” “read-only,” ownership, and method effects. The unification supports new annotations: “unique-write” and “from”. We also develop a model of object invariants in the presence of inheritance using “nesting,” an extension of “adoption.”

1. INTRODUCTION

Imperative object-oriented programming is a powerful and flexible system of programming, but imperative features can make programs harder to understand and reason about, and thus harder to optimize. Interactions between parts of the program can be hard to track because of *aliasing*, in which the same mutable state can be accessed with different names. The additional possibility that any reference may be null means that almost any operation could potentially throw a null pointer exception. Furthermore, initialization through constructors may include arbitrary code which, combined with dynamic dispatch, can lead to a method being called on an object which has not been fully initialized. All these problems can occur in single-threaded programs; adding multi-threading brings a host of new problems.

Many researchers propose a solution to these problems using *annotations*. Rather than design a new language (and thus a new compiler and run-time system), they instead use annotations on code elements to indicate the *design intent* of the code. These annotations have a dual function: they *describe* the behavior of the code for better understanding by users; they *prescribe* the behavior of the code for implementers and program maintainers. We are interested in purely *non-executable* annotations in which the behavior of the code is entirely unaffected by the annotations. Annotations can be seen as an optional type system [18]; the advantage is that annotations can be checked separately from the main compiler and runtime system.

The host of (mainly non-executable) annotations that have been proposed to describe design intent in object-oriented programs includes *borrowed* [15] (also called *lent* [3, 2]), data groups [34] (also called regions [19, 29]), effects [29, 20], immutability [43, 6], non-nullness [27], ownership [24, 23, 2, 9], *raw* [27], *readonly* [40, 32, 47, 6, 50], and *unique* [31, 11, 38, 2, 22]. While most of the concepts involved are fairly intuitive, the precise semantics for something as “obvious” as *unique* can be difficult to pin down [11]. Moreover, putting all these annotations together can lead to a semantic muddle as each proposal uses its own notation and semantics.

Even so, combining annotations would be straightforward if the annotations did not interact with one another. Unfortunately, the interactions of individually-defined annotations are not immediately obvious, as we have shown for AliasJava [2] with the interaction between uniqueness and borrowing [16]. The undesired interaction in this case (representation exposure) can be addressed by annotating methods with field effects, which adds another annotation with which to interact. Similarly, Leino and others [37] found that an “ownership exclusion” principle was needed in order to maintain object invariants. When considering effects, access to immutable state can be ignored, but access to read-only state cannot be, since that access may interfere with a writer. Likewise, effects through a unique reference may be mapped back to the holder of that reference. All annotations that describe access to mutable state; must interact on some level.

This paper addresses the problem of annotation interaction by providing a foundation on which to ground all the annotations described above. *Fractional permissions* can be used to give a precise meaning to each annotation, and can be checked statically and modularly (one method body at a time). We have already shown [16] how uniqueness and object-oriented effects systems can be modeled using permissions. In this paper, we further add ownership, non-null, read-only, and immutable type qualifiers. Along the way, we show how to model “unary” object invariants (invariants that refer only to one field). Furthermore, (but this is not explored in this paper), fractional permissions can be used to express design intent annotations on multi-threaded programs as well [51].

The theory of (fractional) permissions gets back to the basic issue: mutable state. In the object-oriented paradigm, *fields* of objects serve as loci of mutable state. Each field is associated with exactly one permission. A single permission can be broken into “fractions” each of which permits read access; write access requires the whole permission (possibly

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWACO '09, July 6 2009, Genova, Italy

Copyright 2009 ACM 978-1-60558-546-8/09/07 ...\$10.00.

```

class Square{
  group Location;
  group Size;
  Point ul in Location;
  double side in Size;
  ...
  writes Location
  void translate(Vector d) {...}
}
reads s.Size
writes s.Location
void moveLeft(Square s){
  Vector b = new Vector(-s.side,0);
  s.translate(b);
}

```

Figure 1: Data Groups and Effects

reassembled from fractional segments) [12]. By requiring that the programmer indicate which part of the program has access at any point, all data dependencies can be made apparent by inspection without requiring an auxiliary alias analysis. In this way, large systems can be developed in separate pieces. In the terminology of Aldrich [2], “covert channels” of mutable state are forbidden.

The contributions of this paper are threefold:

- The translation of an annotation to fractional permissions helps understand what the annotation “means” separate from any particular rules proposed for checking it.
- Since all annotations are given semantics in a single system, it is possible to understand interactions precisely.
- It becomes easy to add new annotations that “behave well” with existing annotations and that can be checked by existing checkers.

We give a “semantic” definition of annotations (one that concerns program behavior), rather than a “structural” definition (one that concerns program structure). As a result, the desired properties are maintained by (correct!) compiler optimization, even if the compiler is unaware of the permission system.

In the following section, we introduce proposed annotations and ask some questions on interactions. Section 3 describes the concept of permissions and Section 4 explains how permissions can model the various annotations under discussion. In Section 5, we briefly describe how these ideas can be applied to Java. Section 6 reviews the related work.

2. ANNOTATIONS

In this section we touch on intuitive definitions for several previously-proposed annotations in the context of a Java-like language. We follow previous work only partially as we come up with our meanings for these annotations. Later we will show how fractional permissions can be used to provide each with precise semantics.

2.1 Effects

We start by discussing effects annotations. Each method is annotated with a list of which fields it reads and which fields it writes. In addition to fields, effects may also mention

```

class Node <owner> {
  owner Node next;
  ...
}
class List{
  this Node head;
  ...
}

```

Figure 2: Ownership in a List

data groups. A data group is an abstraction of a collection of fields. One data group may be nested inside another.

Following some earlier proposals (but not others), we determine that write effects include the corresponding read effects. This loses some precision, but since a write effect indicates a possibility not a certainty, distinguishing a write effect from a combine write-read effect has no bearing on data dependencies.

Figure 1 shows a `moveLeft` function that directly reads the field `s.side`. This effect is covered by the declared effect `reads s.Size` because the `side` field is in the data group `Size`. Alternately, we could have declared that `moveLeft` reads `side`, or even that it writes `Size`, since write effects subsume read effects. The effect `writes s.Location` is required because `moveLeft` calls `translate`, which declares that it writes `Location`. Changing the declared effect on `s.Location` from a write to a read would cause an error when checking the call to `translate`.

2.2 Ownership

An ownership system describes objects’ representation beyond the objects themselves. For example, a linked list (Figure 2) is represented by nodes, but the nodes are not stored within the list object on the heap. We say that the list nodes are *owned* by the list (and not by anything else) to represent that the list nodes make up part of the representation of the list. In Figure 2, the field `head` of class `List` is annotated with `this` to indicate that the head is owned by the list. The remaining nodes have no direct way to refer to the list and so we use an *ownership parameter* on class `Node`. In our annotation system, an explicit ownership annotation supplies the value to use for the ownership parameter.

In addition to tracking who owns what, ownership systems attempt to enforce policies, such as *owners-as-dominators*. This means that no references exist from outside the representation of an object to other objects within its representation (no pointers to list nodes from outside of the list). We instead enforce *owners-as-effectors*. One cannot read or write owned state without somewhere declaring the appropriate effect for the owner. Thus, one requires permission to access the `List` to also access the owned list nodes in Figure 2.

The *owners-as-dominators* policy is structural, whereas *owners-as-effectors* is semantic. This matters, for example, when inlining method calls; with *owners-as-dominators*, the inlined code may technically be incorrect since owned fields are being accessed in code outside the owning object. In contrast, using *owners-as-effectors*, if the code before inlining uses permissions correctly, then the caller has all the effects needed by the callee. Thus inlining can never cause a problem.

2.3 Unique and Borrowed

A *unique* pointer is the only pointer to an object. Again, this works well with a singly linked list (Figure 3). However, if one can *never* alias a unique pointer, writing code can be

```

class UNode{
    unique UNode next;
    readonly Object datum;
}

writes n.next
void append(borrowed UNode n, unique UNode m){
    m.next = n.next;
    n.next = m;
}

```

Figure 3: Uniqueness in a Linked List

come awkward. Therefore, in most systems with uniqueness, *borrowed* pointers are allowed to alias unique pointers temporarily. If we call `append` to place a new node after the head of the list (`append(head, new UNode())`), then `head` is not unique for the duration of the call because `n` aliases it. This is allowed because `n` is only a borrowed alias.

There is another point in `append` where uniqueness is temporarily broken. After the first statement but before the second, both `m.next` and `n.next` refer to the same node. This is acceptable in some systems, such as “alias burying,” because `n.next` is set to uniquely point to another node before it is used elsewhere.

As with ownership, then, we have weakened uniqueness to “effective uniqueness”—other references to the “unique” object may exist but they are not used to read or write the state of the object. This enables uniqueness again to be a semantic concept, unaffected by a compiler that may leave an alias of a “unique” pointer in a register for a lengthy time.

2.4 Read-only and Immutable

A *readonly* pointer can be used to read but not write the object it references. The object may be written, however, using other pointers. An immutable object cannot be written by anyone; an *immutable* reference points to an immutable object. Thus an immutable reference can be used where a read-only reference is expected, but the reverse is illegal.

2.5 A Combined System

All of these annotations express design intent about access to mutable state. Fractional permissions enable us to reason rigorously about access to mutable state. Therefore, by expressing each of these annotations in terms of fractional permissions, we get a semantics for these annotations that both adequately expresses their intuitive meaning and enables us to reason about interactions between them. For example, questions such as those below can be answered using permissions.

1. Can a *readonly* pointer be used to write state owned by the object it points to? Uniquely referenced by the object it points to? (Figure 4)
2. If a method has read effects on one parameter and write effects on another, could they be aliases? (Figure 5)

3. PERMISSIONS

In this section, we define a system of linear permissions sufficient to provide a common semantics for all these annotations. Since permissions (and in particular “fractions” and

```

void writeRO(readonly List l){
    l.head.next = null; //? permitted
}
void writeRU(readonly UList l){
    l.head.next = null; //? permitted
}

```

Figure 4: Read-only with Uniqueness/Ownership

```

reads a.side
writes b.Size
void foo(Square a, Square b){
    if(a == b)
        //? reachable
    ...
}

```

Figure 5: Effects and Aliasing

“nesting”) form the core of our system, we explain them in depth. The syntax of permissions is summarized in Figure 6 on page .

3.1 Permissions, Informally

A *permission* (Π) is a token associated with (potentially mutable) state in a running program. The permission is the right to access that state, to read or write what is stored in the state. For simplicity, we restrict mutable state to fields in objects on the heap. Permission checking is done statically; permissions are ignored at runtime.

There is exactly one *unit permission* associated with every field in the heap. If ρ refers to some object on the heap, f is a field and ρ' is the contents of that field, then we can give the syntax of a unit permission as one kind of permission:

$$\Pi ::= \rho.f \rightarrow \rho' \mid \dots$$

A unit permission gives the right to access the state, reading or writing the field. If the field is written, the unit permission changes to reflect the new value stored in the field.

A *compound permission* $\Pi + \Pi'$ gives one all the rights associated with both of the permissions being compounded.

A permission may be *scaled* by a fraction ξ :

$$\Pi ::= \dots \mid \xi \Pi \mid \dots$$

Here ξ represents a positive fraction (rational number). It may be a fraction variable, but will never be zero. Fractions give a way to get multiple permissions for a single state; a permission may be split in two:

$$\Pi \equiv \frac{1}{2}\Pi + \frac{1}{2}\Pi \equiv \frac{1}{2}\Pi + \frac{1}{4}\Pi + \frac{1}{4}\Pi \equiv \frac{3}{4}\Pi + \frac{1}{4}\Pi \equiv \dots$$

Scaling distributes through compounds. A scaled *unit* permission gives *read* access to the field thus referred to.

The linear implication permission $\Pi_1 \multimap \Pi_2$ (“ \multimap ” is pronounced “scepter”) means that one has the rights of the consequent Π_2 , except for the ones of the antecedent Π_1 . The rights implicit in a linear implication cannot be used until it is combined with its antecedent using a form of *modus ponens*:

$$\Pi_1 + (\Pi_1 \multimap \Pi_2) \Rightarrow \Pi_2$$

Both the antecedent and the linear implication are consumed in the process. The $\Pi_1 \Rightarrow \Pi_2$ means that Π_1 can be viewed as Π_2 .

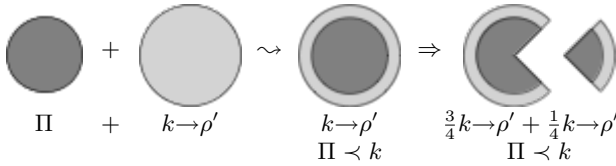
Permission nesting is an extension of Fährdrich and DeLine’s “adoption” [26]. A permission may be *nested* in a field, which means that anyone who has the unit permission (the *nester*) also has the *nested* permission. We represent the information that Π is nested in $\rho.f$ by a formula

$$\Gamma ::= \dots \mid \Pi \prec \rho.f \mid \dots$$

The formula does not represent any permission itself; it merely indicates that the nested permission Π is available if one has the nester permission $\rho.f \rightarrow \rho'$. The nesting *fact* is “nonlinear,” that is, can be duplicated arbitrarily, unlike permissions in general. The next subsection will define some other facts.

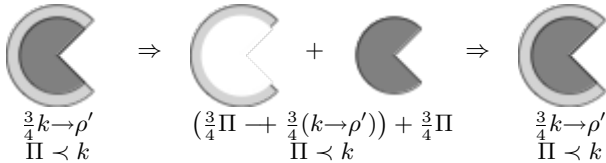
Nesting achieves two purposes: abstraction, in that a number of permissions may be nested in a single “façade” unit permission; and ease of use because nonlinear facts are more easily used by a type system, as explained by Fährdrich and DeLine [26].

The picture here shows the composition of two permissions being put together (\rightsquigarrow) by nesting the darker permission Π in the lighter one k ($k \equiv \rho.f$ refers to a field of some object),



after which case the darker permission is no longer (directly) accessible, but a new nesting fact $\Pi \prec k$ is known. Then this nester permission is split into two pieces, thus implicitly splitting the nested permission. The nesting fact $\Pi \prec k$ is immutable (nonlinear) and thus can be duplicated.

Access to the nested permission requires *carving* it out of the nester permission (Fährdrich and DeLine use the term “focus”). The carve operation leaves a hole in the nester permission. The hole is represented by linear implication (*nested* \dashv *nester*). The nester permission will not be considered complete until the nested permission is “replaced.”



As seen in the picture, permission carving handles fractions transitively: if one has only a fraction of the nesting permission, one can only get a fraction of the nested permission. When one is done with the nested permission, it can be replaced using the linear *modus ponens* rule explained above.

3.2 Permission Syntax

Figure 6 gives the definitions of the various permission-related non-terminals.

Our permission system includes an embedded language of formulae. A *formula* Γ represents a fact whose truth (once known) will hold everywhere and forever afterwards. We say that facts are *nonlinear* in that they can be arbitrarily duplicated or discarded. For this reason, it is easier (both for humans and machines) to reason about facts. This advantage comes at the cost of flexibility because a fact can never change.

$\rho ::= o \mid r$	<i>literal reference, variable</i>
$\xi ::= q \mid z$	<i>literal fraction, variable</i>
$\delta ::= r \mid z \mid v$	<i>any variable</i>
$k ::= \rho.f$	<i>key (field instance)</i>

$\Gamma ::=$	<i>formula:</i>
\top	<i>true</i>
$\Gamma \wedge \Gamma$	<i>conjunction</i>
$\neg \Gamma$	<i>negation</i>
$\exists \delta \cdot \Gamma$	<i>existential</i>
$p(\bar{\rho})$	<i>named predicate</i>
$\rho = \rho$	<i>reference equality</i>
$\Pi \prec k$	<i>nesting</i>
$\rho \in C$	<i>object typing</i>

$\Pi ::=$	<i>permission:</i>
Γ	<i>formula</i>
$k \rightarrow \rho$	<i>unit permission</i>
v	<i>permission variable</i>
\emptyset	<i>no permissions</i>
$\Pi + \Pi$	<i>combination</i>
$\xi \Pi$	<i>fraction scaling</i>
$\Gamma ? \Pi : \Pi$	<i>conditional</i>
$\exists r \cdot \Pi$	<i>existential</i>
$\Pi \dashv \Pi$	<i>implication</i>

$P ::= \{ \dots, p(\bar{r}) = \Gamma, \dots \}$	<i>predicate defs</i>
$\Delta ::= \{ \dots, \delta, \dots \}$	<i>variables</i>
$\alpha ::= \forall \Delta; \Pi \longrightarrow \exists \Delta; \Pi$	<i>procedure type</i>

Figure 6: Permission Syntax.

For this paper, we have just three atomic facts, equality facts ($\rho = \rho'$) which test whether two reference values are identical, nesting facts (described above) and type assertions that check whether an object is of a class or a derived class. Formulae also include the constant true (\top), operators \wedge and \neg and existentials. The constant false (\perp), disjunctions and fact implications (written using \implies) are derived forms. Named predicates are defined in an implicit global P .

The primitives (\emptyset , $+$ and \dashv) used to combine permissions (Π) are analogous to those of multiplicative linear logic ($\mathbf{1}$, \otimes and \multimap) and those of separation logic (**emp**, \star and \multimap). We do not use separation logic’s syntax because the metaphor is wrong (“ \star ” would be addition for fractions) and to avoid confusion in how we treat non-linear terms: “ \top ” corresponds to **emp**, not to **true**. We also use a semantics for \dashv that is more restricted than for \multimap [14].

A permission may include a permission variable v used to refer to an unknown permission treated parametrically by a method. A formula Γ may be treated as a permission with no rights, as long as the formula is true.

We also make use of existential and (in procedure types) universal qualifiers. Unlike both linear logic and separation logic, the permission system does not provide a general disjunction operator, but rather a more limited “conditional” $\Gamma ? \Pi_1 : \Pi_2$ which yields the permission Π_1 if the fact Γ is true and Π_2 if it is false.

A procedure type $\forall \Delta; \Pi \longrightarrow \exists \Delta'; \Pi'$ is polymorphic over variables in Δ . It accepts the permission Π and returns the

permission Π' , using perhaps some new variables Δ' (as well as the existing variables Δ). If either Δ or Δ' is empty, we omit the corresponding qualifier.

4. REPRESENTING ANNOTATIONS

We now discuss how we represent annotations with fractional permissions. We start with the annotations discussed in Section 2, then look at how to add non-null annotations and object-oriented features. Last, we consider other possible annotations derivable from fractional permissions.

4.1 Effects and Data Groups

Effects annotations for a method are directly encoded in the procedure type $(\forall\Delta; \Pi \rightarrow \exists\Delta'; \Pi')$. Permission for each field mentioned in the effects is both passed to and returned from the method; these permissions will be listed in both Π and Π' . A whole permission is used for a write effect, and some fraction, represented with a fraction variable, of a permission is used for a read effect.

We can model a data group using a field, and put other fields in the group using adoption. Because we never use the group as a field, but only for nesting other fields, it can have an uninteresting type. Thus, we give data groups a type indicating that they always point to null.

The example in Figure 1 can be permission-checked as follows. Permissions for the “fields” `s.Size` and `s.Location` are passed into `moveLeft`. The former receives a fractional (in $(0, 1)$) permission, while the latter has fraction 1. To read the field `s.side`, a (read) permission for `s.side` is carved out of the read permission for `s.Size`. After checking the read, this permission can be restored. Then, the write permission for `s.Location` is passed into `translate` because of the effects of that method. Per the procedure type for `translate`, it returns the write permission for `Location`. Therefore, `moveLeft` can return both permissions mentioned in its effects.

In addition, we give every class the special data group `All`, inherited from `Object`. By default, all fields and data groups that do not specify which group they are in are placed in `All`. Thus, the effect `writes s.All` also suffices for `moveLeft`, as both `Size` and `Location` are in `All`. (Use of the `All` data group is not necessary to represent any annotations; it is a convenient holder for all field permissions for an object, but they could always just be listed separately.)

4.2 Pointer Annotations

We can represent each of the pointer annotations from Section 2 using fractional permissions to limit access to the annotated state. For field annotations, the associated permission is passed into and returned out of any method using that field as an effect (as with the effect itself). For parameters, the permission is passed into the method, but not out; for return types, out of the method, but not in. All of this is captured in the permission procedure type (see Section 3.2), which lists all permissions passed in or out of the method.

4.2.1 Ownership

We implement the *owners-as-effectors* policy of Section 2.2 by giving each object a data group `Owned` into which all owned state is nested. For state owned by the world, we adopt it into the `Owned` group of the null pointer. Fields without annotation are treated as if owned by the world.

```
class Foo{
  unique Object u;
  world Object o;
  ...
  writes u, o
  void bar(){
    o = u; /// ERROR
  }
}
```

Figure 7: Compromising unique

```
class Student{
  immutable Name id;
  readonly Grade score;
  ...
}
```

Figure 8: Read-only and Immutable

More precisely, the annotation of `owner` on the `next` node is represented (in the class invariant as described below) with the permission $\exists r \cdot ((r_{\text{this}}.\text{next} \rightarrow r) + (r \neq 0 ? (r.\text{All} \rightarrow 0 \prec r_{\text{owner}}.\text{Owned}) : \emptyset))$, where r_{this} and r_{owner} are location variables referring to the `Node` object and the object passed as its ownership parameter, respectively. The permission states that if the next pointer is not null, the node it points to has its `All` data group nested into the `Owned` data group of its ownership parameter. As all fields of the `next` node are nested in its `All` group, the only way to access the `next` node is to carve the appropriate permissions from its owner (in this case, the `List`). Without permission to the list, we could not access its owned nodes.

4.2.2 Unique and Borrowed

As with ownership, we treat the `unique` annotation as restricting access to the pointed-to state, rather than forbidding the existence of aliasing. That is, no one has permission to access the uniquely pointed-to object except through the unique reference. We accomplish this by enclosing the linear write permission for the object with the unique reference.

For example, the `unique` annotation on the `next` field of the `UNode` in Figure 3 is given a permission like $\exists r \cdot ((r_{\text{this}}.\text{next} \rightarrow r) + (r \neq 0 ? r.\text{All} \rightarrow 0 : \emptyset))$; again, r_{this} represents the node to which this field belongs. The only way to access any of the fields of the next node is to carve them out of the `All` data group. This requires having permission for the `All` group, which requires permission for the unique `next` field.

A `borrowed` reference is represented even more simply; it has no permissions associated with it at all. Instead, it must borrow access from elsewhere. This use is temporary, limited by when the permission will be needed elsewhere.

Thus, for the method `append` in Figure 3, two unique permissions are passed into the method, one for `n.next`, from the effect, and one for the unique parameter `m`. Only the former is returned from the method. Even though `n` is borrowed and is passed with no permission, we can read and write `n.next` because we can borrow the permission that was passed in with the effect. Without the effect declaration, we still have permission to read and write `m.next`, but not `n.next`.

Does controlling access actually enforce uniqueness? Figure 7 shows a function that will not check. After the assignment, both `o` and `u` point to the same object. Because the

```

class Grader{
  unique-write Grade score;
  ...
}

```

Figure 9: A unique-write

method must return its effects, `bar` must return the permissions for `o` and `u`. The former requires its object to nest its `All` group permission in `0.Owned`; the latter requires permission for the `All` group directly. Since both refer to the same object, both require the same linear permission (for the `All` group). Because this permission is linear, it cannot both be adopted by the world and also enclosed with the unique pointer. Therefore, the procedure type for `bar` cannot be satisfied.

4.2.3 Read-only and Immutable

A *readonly* reference can read but not write the object to which it refers; however, there may be other references that can write the object. Expressing *readonly* by including a fraction of the permission for the pointed-to object is therefore unworkable, as it will prevent another reference from assembling a write permission. Instead, we want to place the read permission for the *readonly* reference in a publicly accessible location, where it can be accessed both by the read-only reference and by potential writers. To this end, we nest some fraction of the permission to access the referenced object in `0.Owned`, the state owned by `world`. Thus, the permission for the `score` field in Figure 8 is $\exists r \cdot ((r_{\text{this}}.\text{score} \rightarrow r) + (r \neq 0? (\exists z \cdot zr.\text{All} \rightarrow 0 \prec 0.\text{Owned}) : \emptyset))$ (ignoring class predicates, which are discussed below). If the field is not null, then some read permission for that grade is owned by the *world*. Because the adoption fact can be duplicated, *readonly* fields may be freely aliased.

It is possible for some references to an object to be *readonly* and others to be owned by `world`. Both annotations rely on the same fact—that permission to access the state of the object is nested in `0.Owned`. Because the whole (write) permission is nested, the `world`-owned references can write the object; however, the *readonly* references will not *know* that the whole permission is nested, and so cannot.

We can handle *immutable* fields similarly. Here, we do not want anyone to be able to assemble the whole permission—that would allow them to write immutable state. To this end, we use a separate data group (`0.Immutable`) in place of world ownership, and deliberately discard a fraction of the permission for that group. The associated permission for the `id` field in Figure 8 is $\exists r \cdot ((r_{\text{this}}.\text{id} \rightarrow r) + (r \neq 0? (\exists z \cdot zr.\text{All} \rightarrow 0 \prec 0.\text{Immutable}) : \emptyset))$.

4.3 Nullness

Each pointer annotation describes access to the state of the pointed-to object. If the pointer is `null`, there is no such object and no such state. Therefore, the permission first tests whether the pointer is null using a conditional permission.

If we had *nonnull* annotations, we could “know” that some fields were not null, and we could eliminate the conditional check for nullness for those fields. On the other hand, if the permission omits a conditional check for null (as in $\exists r \cdot ((r_{\text{this}}.\text{next} \rightarrow r) + (r.\text{All} \rightarrow 0 \prec r_{\text{owner}}.\text{Owned}))$), then the permission can only be formed if the field is **not** null. Were `r` null, we could not construct the nesting fact

$r.\text{All} \rightarrow 0 \prec r_{\text{owner}}.\text{Owned}$, because no one has permission for the non-existent field `0.All`. We can, therefore, support *nonnull* and *maybeNull* annotations using fractional permissions: a *maybeNull* reference will have a conditional check for null preceding the permissions associated with its pointer annotation whereas a *nonnull* reference will not.

4.4 Class Invariants

A pointer annotation on a field declaration in a class imposes an invariant on each instance of the field; these are called “unary field invariants.” Together with the structure of the class’ data groups—which fields are in which data groups—these unary field invariants comprise the class invariant. The structure of the data groups is represented with a conjunction of nesting facts. These facts also express the annotation invariants: the nesting fact includes the permission representation of the annotation. In the `UList` in Figure 3 on page , the nesting fact $(\exists r \cdot r_{\text{this}}.\text{head} \rightarrow r + r \neq 0? r.\text{All} \rightarrow 0 : \emptyset) \prec r_{\text{this}}.\text{All}$ expresses both that the `head` field of the `List` is unique and that it is directly nested in the `All` data group (as are all fields not explicitly placed in another group).

The class invariant is represented with a two-argument named predicate. The first argument is the location of the object instance (r_{this}); the second is the value of the ownership parameter. The value of the predicate is the conjunction of nesting facts described above. For simplicity of presentation, we will assume all classes have exactly one ownership parameter; to adjust this number we need only add more arguments to the class invariant predicate.

A class invariant also includes the named predicate for its superclass. This ensures that inherited fields will also have the correct permissions and data group nesting.

The class invariant must be established by the constructor. The constructor is modeled by a method that takes the permissions for each field individually and returns the permission for the “All” data group after establishing the class invariant for its own and all superclasses. The permission to “All” and the class invariant jointly imply that all fields are consistent with their types.

Once the invariant is established in the constructor, it can be broken by carving the field out of its data group and assigning it a value that does not fit. But then the data group permission cannot be restored until the required unary field invariant is restored. Boogie uses a similar semantics for invariants [4]. In JML [33], all invariants of fully-constructed objects must be valid in every method, except that a “helper” method may be called on an object whose invariant is currently broken. With permissions, one has a looser semantics: the only objects whose invariants must be true when the method starts are those whose state can be observed using the permissions passed into the method.

4.4.1 Raw and Cooked

A constructor may pass, directly or indirectly, the object under construction to another method before its class invariant has been established. Following Fähndrich and Leino [27], we call references to objects whose invariants may not have been established *raw*, and references whose invariants have been established through (super-) class `C`, but not necessarily farther *raw[C]*. In direct analogy, we call a reference whose annotation has been fully established *cooked*.

All three possibilities can be described with permissions. A `raw` reference will neither assume nor require any class invariant at all. A `raw[C]` reference includes exactly the class invariant for class C , which recursively contains the invariants for all of its superclasses. If the actual object is of a subtype of C , it is still only assured of the invariant for class C .

When an object is `cooked`, we know that the invariant is true regardless of its dynamic type. We may not know what that type is. Therefore, the `cooked` predicate states that the invariant is true for every dynamic type that the object possesses:

$$\begin{aligned} \text{cooked}(r, o) = & (r \in C_1 \implies C_1(r, o)) \\ & \wedge \dots \wedge (r \in C_n \implies C_n(r, o)) \end{aligned}$$

Here $r \in C$ states the object at location r is of class C or one of its descendants, and $C(r, o)$ is the class invariant for class C (where o is the ownership parameter). The body of the `cooked` predicate ranges over all classes in the system.¹ Thus a method can require that its receiver be “cooked” and permit an overriding method to use the same predicate to provide a stronger invariant. A reference with a `raw[C]` annotation only guarantees those from C up the class hierarchy. For this paper, we assume that a reference without a rawness annotation is `cooked`.

4.5 Questions Answered

At the end of Section 2, we asked some questions about the interactions of annotations. Now that we have a common permission semantics for the annotations, we can examine those questions in more detail.

Question 1: *Can a read-only pointer be used to write state owned by the object it points to?* In Figure 4 on page , the `writeRO` method attempts to write a list node using a `readonly` reference to the `List`. Let us assume the method is given a `write` effect for `world` and that all fields are probably `nonnull`. Read-only means that some fraction (say z_L) of the `List`’s state is nested in `0.owned`. We can carve this fraction of a permission for the `List` ($z_L r_l. \text{All} \rightarrow 0$) from the permission passed in as the world effect. We could carve permission to write the fields of the owned `Node` from a write permission for the `List`’s state as the ownership annotation on `head` means that permission for the `Node` pointed-to by `head` is nested in the `Owned` group of the `List`. However, as we only have read access to the `List`, we can only carve out read access to the `Node` from the `List`’s `Owned` group. Therefore, the attempted assignment cannot be type-checked.

Can a read-only pointer be used to write state uniquely referenced by the object it points to? If we attempt the same thing with a `unique` linked list (`writeRU` in Figure 4), we run into a similar difficulty. Again, we can carve out a fraction (z_L) of the permission for `l`, the `UList`. With a `unique` reference, permission to access the head `UNode` is enclosed with the permission for `l.head`. We can only carve a fraction (not a whole) permission for `l.head` from the fraction (z_L) of a permission for `l`. Then only that fraction of the permission for the `UNode` to which `head` points is enclosed with the partial permission for `l.head`. Thus we can read but not write the fields of the `UNode`. The attempted assignment does not type-check.

¹This implies a closed-world assumption. However, in practice we need not enumerate all possible classes, only those of interest, with other classes being added as necessary.

Question 2: *If a method has read effects on one parameter and write effects on another, could they be aliases?* In Figure 5, the `foo` method receives a read (fractional) permission for the `side` field of `Square` `a` and a write (whole) permission for the `Size` group of `b`, per the effects declaration. As the field `side` is in the group `Size`, we can carve out a whole (fraction is 1) permission for `b.side` from the permission for `b.Size`. Then, if `a` and `b` are aliases, we could combine the permissions for `a.side` and `b.side` and get a permission for the field with a fraction > 1 . As we disallow fractions greater than 1, `a` and `b` cannot alias; the body of the `if` is unreachable code.

With a more complex runtime system, we could create two versions of the method: one for when `a` and `b` are aliases and one for when they are not. If we know the parameters are aliased, we can tailor the procedure annotation accordingly and avoid having too much permission for the field. Doing this is outside the scope of this paper.

4.6 Other Pointer Annotations

Fractional permissions provide a common semantics for pointer annotations concerned with controlling access to mutable state. As seen in Section 2, several such annotations have been proposed. With a common semantics, we can also express meanings not associated with previously proposed annotations. In this section, we briefly describe additional annotations that could be defined in terms of fractional permissions.

4.6.1 Readonly-owner

We express ownership through nesting permission for the owned object in the owner’s `Owned` data group and `readonly` through nesting a fraction of the read-only object’s permission in `0.Owned` (the group for state owned by `world`). It seems natural, then, to allow nesting a fraction of an object’s state in an arbitrary owner ($(\exists z \cdot z r. \text{All} \rightarrow 0) \prec r_{\text{owner}}. \text{Owned}$). We express this with a `readonly-owner` annotation, where `owner` is the particular ownership parameter (or `this`) functioning as the owning object. Read-only ownership gives the owner read access to the state, but some part of the program with sufficient knowledge of where all the fractions are, and permission to get at these fractions (including the fraction nested here) may still be able to mutate the state.

4.6.2 Unique-write

We want other references to an object referred-to with a `readonly` reference to be able to write the object. A `world`-owned reference would be able to do so—it knows that the fraction of state nested in `0.Owned` is really 1. An alternative is to nest a known fraction (usually $\frac{1}{2}$) of the state and keep the remainder in a “unique write” reference (as a unique pointer keeps the whole permission). That reference, and no other, could reassemble the write permission.

In Figure 9, the `Grader` alone can write the `score`, although the `Student` can read it. The permission for the `Grader`’s `score` field is $\exists r. r \text{this.score} \rightarrow r + r \neq 0? \frac{1}{2} r. \text{All} \rightarrow 0 + (\frac{1}{2} r. \text{All} \rightarrow 0 \prec 0. \text{Owned}) : \emptyset$ (again ignoring class invariants). The `Grader` can write the fields of `score` by carving the half permission from `0.Owned` and combining it with the other half permission to get a whole (write) permission. The `Student` can read their score by carving out the world-owned fraction.

Non-null Annotation (na)	Precondition ($\Gamma_{na,r}$)
<code>nonnull</code>	<code>true</code>
<code>maybenull</code>	$r \neq 0$

Figure 10: Translation of Nullity Annotation

Rawness Annotation (ra)	Facts ($\Gamma_{ra,r}$)
<code>raw[C]</code>	$C(r, r_{owner})$
<code>cooked</code>	$cooked(r, r_{owner})$

Figure 11: Translation of Rawness Annotation

4.6.3 From

The effects of a method are the permissions that a method needs to operate; they are returned when the method returns. Sometimes we want an object returned from the method to hold these permissions after the method call. A classic example is an iterator, which needs permission to read and/or write a collection while in use. We can represent this with permissions by carving the iterator’s permission from the permissions passed into the `iterator` method for its effects annotation. This is represented with a `from(effect)` annotation; we discuss `from` in detail in other work [17].

4.7 Formal Translation

In general, the full permission for any annotated reference is composed of three pieces: the rawness modifier (ra), the nullness modifier (na), and the pointer annotation itself (pa). The first two are represented as predicates ($\Gamma_{ra,r}$ and $\Gamma_{na,r}$ respectively), while the pointer annotation is translated as a permission ($\Pi_{pa,r}$). These are determined by both the annotation and the location (r) of the annotated object.

The rawness modifier (ra) is converted into the (class invariant) predicate Γ_{ra} ; this will be the cooked predicate for a fully constructed object, and the literal invariant for class C for an object only constructed through class C . A `raw` reference becomes `raw[Object]` and so is constructed through the `Object` class and no farther.

The nullness modifier (na) will be either `nonnull` or `maybenull`. The latter means we only have permission if the object is not null; the latter that we always have permission (because we know the object cannot be null). For simplicity, we always represent nullity as a precondition, using `true` as the precondition when the field is `nonnull`, as shown in Figure 10.

The various pointer annotations are given semantics by the permissions ($\Pi_{pa,var}$) included in the existential closure along with the field permission. These are summarized in Figure 12. As before, r represents the object to which the field points. The last four annotations in the diagram may annotate method parameters or return values, but not fields.

The rawness, nullness, and pointer annotations are thus represented by this single permission.

$$\Gamma_{na,r} ? (\Pi_{pa,r} + \Gamma_{ra,r} + r \in C) : \emptyset$$

For fields, this permission shows up in the class invariant existentially qualified for r and nested in the appropriate data group. For parameters and return types, it is included in the permission type for the method.

Pointer Annotation (pa)	Permissions ($\Pi_{pa,r}$)
<code>unique</code>	$r.All \rightarrow 0$
<code>shared</code>	$r.All \prec 0.Owned$
<code>immutable</code>	$\exists z.zr.All \prec 0.Immutable$
<code>readonly</code>	$\exists z.zr.All \prec 0.Owned$
<code>unique-write</code>	$\frac{1}{2}r.All \rightarrow 0+$ $\frac{1}{2}r.All \prec 0.Owned$
<code>owner</code>	$r.All \prec r_{owner}.Owned$
<code>readonly-owner</code>	$\exists z.zr.All \prec r_{owner}.Owned$
<code>borrowed</code>	
<code>from(x.f)</code>	$r.All \rightarrow 0+$ $r.All \rightarrow 0 \dashv \xi_{r_x}.f + v$
<code>readonly-from(x.f)</code>	$\frac{1}{2}\xi_{r_x}.All \rightarrow 0+$ $\frac{1}{2}\xi_{r_x}.All \rightarrow 0 \dashv \xi_{r_x}.f + v$

Figure 12: Translation of Pointer Annotations

4.7.1 Class Predicate Example

Figure 13 shows the definition of $UNode(r_t, r_o)$, the class predicate for the `UNode` class from Figure 3. The predicate requires two parameters: the former (r_t) is the location of the `this` object; the latter (r_o) is the location of the owner of the `UNode`. As the `UNode` has no owner, the value of r_o will be null.

The value of the predicate is a conjunction of three facts. The first of these is the class predicate for `Object`, which results from the `UNode` class directly inheriting from `Object`. Anything which is true for `Objects` is also true for `UNodes`.

The second fact is the nesting of the permission for the `next` field into the `All` data group. The memory location (r_n) to which it refers is existentially quantified. In addition to the permission for the `next` field itself, the existential encloses some permissions for the referred-to object. Because the field may be null, these permissions are only available if r_n is proven not to be null. (Conversely, having the permissions would serve as proof of non-nullness.) If r_n is `nonnull`, one has the full permission for its `All` data group and thus sole access to the referred-to object. One also has two facts: one stating that r_n is a `UNode`, and the cooked predicate indicating that it is fully constructed. From these, one may derive the `UNode` predicate for r_n as well. All of this is in the existential closure which is, in turn, nested into the `All` data group of the `this` object (r_t).

The third fact is the nesting fact for the `datum` field. It is very similar to that of `next`, in that there is an existential closure, in which is both the permission for the field and, conditional on the nonnullness of the referred-to object (r_d), several permissions. Here again, the object has a type, and the cooked predicate indicates it is fully constructed. All this is the same because both fields are `maybenull` and `cooked`. However, the last piece is different. As `datum` is `readonly` instead of `unique`, its permission type does not include the permission for $r_d.All$ but rather, the fact that some fraction of that permission is nested in the `0.Owned` data group. As with all fields that do not explicitly belong to a data group, the existential permission for `datum` is nested into the `All` data group of `this`.

4.7.2 Procedure Type Example

Figure 14 provides an example of a procedure type derived from annotations; in this case, it is the procedure type for

$$\begin{aligned}
\text{UNode}(r_t, r_o) = & \text{Object}(r_t, r_o) \\
& + \exists r_{\text{next}} \cdot (r_t.\text{next} \rightarrow r_{\text{next}} + r_{\text{next}} \neq 0 ? (r_{\text{next}}.\text{All} \rightarrow 0 + r_{\text{next}} \in \text{UNode} + \text{cooked}(r_{\text{next}}, 0)) : \emptyset) \prec r_t.\text{All} \\
& + \exists r_d \cdot (r_t.\text{datum} \rightarrow r_d + r_d \neq 0 ? (\exists z \cdot (z r_d.\text{All} \rightarrow 0) \prec 0.\text{Owned} + r_d \in \text{Object} + \text{cooked}(r_d, 0)) : \emptyset) \prec r_t.\text{All}
\end{aligned}$$

Figure 13: Definition of Class Predicate for UNode

$$\forall r_n, r_m \cdot \left(\begin{array}{l} \exists r_{\text{next}} \cdot \left(\begin{array}{l} r_n.\text{next} \rightarrow r_{\text{next}} + \\ r_{\text{next}} \neq 0 ? \left(\begin{array}{l} r_{\text{next}}.\text{All} \rightarrow 0 \\ + r_{\text{next}} \in \text{UNode} \\ + \text{cooked}(r_{\text{next}}, 0) \end{array} \right) : \emptyset \end{array} \right) \\ + r_n \neq 0 ? (r_n \in \text{UNode} + \text{cooked}(r_n, 0)) : \emptyset \\ + r_m \neq 0 ? \left(\begin{array}{l} r_m.\text{All} \rightarrow 0 \\ + r_m \in \text{UNode} + \text{cooked}(r_m, 0) \end{array} \right) : \emptyset \end{array} \right) \longrightarrow \exists r_{\text{next}} \cdot \left(\begin{array}{l} r_n.\text{next} \rightarrow r_{\text{next}} + \\ r_{\text{next}} \neq 0 ? \left(\begin{array}{l} r_{\text{next}}.\text{All} \rightarrow 0 \\ + r_{\text{next}} \in \text{UNode} \\ + \text{cooked}(r_{\text{next}}, 0) \end{array} \right) : \emptyset \end{array} \right)
\end{array}$$

Figure 14: Procedure Type for append

the `append` method in Figure 3. Procedure types have two permissions; those passed into the method and those passed out. The type for `append` also has two location variables (r_n and r_m); these are the locations of the two parameters (`n` and `m`, respectively).

The effect annotation `writes n.next` causes the entire (write) permission for the field `n.next` to be both passed into and returned out of the function. Thus, that permission is listed on both sides of the arrow. This permission is the same existential closure as for the `next` field in the previous example.

Two other permissions are passed into the procedure: one for each parameter. As each parameter is `maybeNull` and `cooked` (by default), each has a conditional test for null, and (if nonnull) the cooked predicate and its type² Because `n` is `borrowed`, no other permissions are passed. In contrast, `m` is `unique`, so its conditional also includes the permission to write `r_m.All`.

Normally, the returned permissions would include an existentially quantified permission for the return value. As `append` is a void function, we need only return the effect.

5. IMPLEMENTATION

We have an implementation of the permission system that checks Java programs with effects and uniqueness annotations. The annotations were originally designed for separate checkers [29, 11] which were implemented earlier (see JSure [30, 49]). By providing a unified semantics, we produced a system that possessed a formal basis lacking previously. The new system was also able to overcome some of the conservativeness of the previous uniqueness system which wasn’t able to use the effects annotations on methods to determine whether a currently aliased unique field might be read.

The object-oriented language here is much smaller than full Java but extending the system to full Java was particularly difficult only for loops (handled by the second author [45]) and for concurrency (handled by the third author [51]). Space precludes discussion of the details.

²In fact, we can always prove that `n` is non-null, because we have passed in the write permission for `n.next`. Were `n` null, the caller would not be able to provide this permission.

6. RELATED WORK

This paper builds directly on previous work of others: the “Adoption and Focus” work of Fähndrich and DeLine [26]; and the work by Fähndrich and Leino on non-null types [27]. Our notion of object invariants is similar to that of Barnett and others [4]. We connect the subjects of these papers by using adoption to help model non-null types in particular, and invariants more generally.

In an earlier paper [15], we unified some of the annotations of the present paper using capabilities. That work introduced the concept of unique-write. It did not solve the problem of aliasing, nor did it have a “semantic” definition of read-only. Nonetheless, it spurred us to continue the work of unification.

Aldrich and others [2] have shown that ownership and uniqueness can be modeled in a single system, as did Clarke and Wrigstad [22] for “external” uniqueness. Clarke and Drossopoulou [20] showed that effects and ownership combine usefully.

Greenhouse [28] and Boyapati [8] each have shown that aggregates of mutable state can be protected by locking mechanisms and that race conditions can be prevented by requiring effects on the aggregate to be properly synchronized. Greenhouse used data groups (“regions”) and effects [29] where Boyapati used ownership and permissions [10].

The term “data group” originates with Leino [34]. In earlier work, we used the moniker “region” [19, 29]. Our data groups aggregate within an object rather than between objects as ownership [21], or “dynamic” data groups [37] do. We achieve the effect of dynamic data groups through the use of existentially closed permissions. Ownership domains [1] can be seen as combining data groups and ownership for more precision. Smith has shown that one can usefully speak of effects on ownership domains [48]. Östlund et al. [42] combine read and write effects with ownership and (external) uniqueness in Joe₃. Ownership parameters can have read-write, read-only or immutability annotations. Unlike previous method effects systems, a method is permitted to modify any accessible state, but permissions may be “revoked” to prevent access.

The JML notation for Java specifications [33] now includes ownership (in the form of Universes [40, 39]), data groups

and write effects (“assignable” clauses). The ESC/Java and ESC/Java2 [25] projects seek to check a subset of the JML syntax at a level roughly equivalent to permission checking. JML and ESC/Java2 could benefit from “accessible” clauses (read effects) because then the invariants would need to be established only for accessible objects, rather than for all existing objects.

We introduced the concept of “fractional permissions” [12] in order to solve the problem of checking interference [46]. Fractional permissions with nesting have a semantics based on fractional heaps [14]. Bornat and others [7] have shown that fractions and generalizations are useful in other areas. Bierhoff and Aldrich’s “full” annotation [5] is similar to our “unique-write” annotation.

Tschantz and Ernst [50] have proposed adding “readonly” to Java, but their proposal does not address issues of ownership and effects [13]. In particular, a method parameter marked “readonly” frequently would be better “borrowed” with the method declaring read effects. The present paper disentangles the concept of a long-term read-only reference from the concept of a reference only intended to be used temporarily for reads.

The “Boogie Methodology” [35, 36] has a similar purpose to our work in providing a unifying foundation for a variety of design-intent annotations. In Boogie, the annotations are converted into quasi-executable forms. In essence, the problem of design-intent checking is reduced to the problem of program verification. The practicality of the system comes from a theorem prover that can usually check the contracts without intervention from the programmer. In contrast, we hope to define a decidable type system of permissions including all the forms needed for annotation translation in this paper.

Despite the different methodology, many of the implications are similar: with Boogie, the invariant is unpacked before it is used, similarly to how we unpack existentials. The restrictions on ownership transfer in Boogie are close to restrictions implied for moving unique objects in our system.

7. CONCLUSIONS

Fractional permissions provide a single semantic model to express a wide variety of proposed Java annotations. In particular, fractions enable us to model immutability, read-only, read effects, unique-write and the novel annotation “from.” Nesting is used to model ownership, data groups and (unary) class invariants. All these annotations are unified in a single formalism.

8. REFERENCES

- [1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP’04 — Object-Oriented Programming, 18th European Conference*, 2004.
- [2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA’02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2002.
- [3] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA’00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2000.
- [4] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3:27–56, 2004.
- [5] K. Bierhoff and J. Aldrich. Cooperative permissions for reasoning about aliased objects. <http://www.cs.cmu.edu/~kbierhof/papers/coop-permissions.pdf>.
- [6] Adrian Birka and Michael Ernst. A practical type system and language for reference immutability. In *OOPSLA’04 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2004.
- [7] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2005.
- [8] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004.
- [9] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA’02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2002.
- [10] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA’01 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2001.
- [11] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31:533–553, 2001.
- [12] John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.
- [13] John Boyland. Why we should not add `readonly` to Java, yet. In *7th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2005.
- [14] John Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin–Milwaukee, Department of EE & CS, 2007.
- [15] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In *ECOOP’01 — Object-Oriented Programming, 15th European Conference*, 2001.
- [16] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2005.
- [17] John Boyland, William Retert, and Yang Zhao. Iterators can be independent “from” their collections. In *IWACO ’07: International Workshop on Aliasing Confinement and Ownership*, 2007.
- [18] Gilad Bracha. Pluggable type systems. OOPSLA 2004 Workshop on Revival of Dynamic Languages, 2004.
- [19] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis

- and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '98)*, 1998.
- [20] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2002.
- [21] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.
- [22] David Clarke and Tobias Wrigstad. External uniqueness. In *Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10)*. 2003.
- [23] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, 2001.
- [24] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 1998.
- [25] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Available on ESC/Java2's web page., 2004.
- [26] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, 2002.
- [27] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2003.
- [28] Aaron Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.
- [29] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, 1999.
- [30] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent Java programs. *Science of Computer Programming*, 58:384–411, 2005.
- [31] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA '91 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 1991.
- [32] Günter Kniesel and Dirk Theisen. JAC – access right based encapsulation for Java. *Software Practice and Experience*, 31, 2001.
- [33] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. 1999.
- [34] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 1998.
- [35] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, 2004.
- [36] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP'06 — Programming Languages and Systems, 15th European Symposium on Programming*, 2006.
- [37] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, 2002.
- [38] Naftaly Minsky. Towards alias-free pointers. In *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, 1996.
- [39] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Fernuniversität Hagen, 2001.
- [40] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *2nd ECOOP Workshop on Formal Techniques for Java Programs*, 2000.
- [41] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Conference Record of POPL 2004: the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2004.
- [42] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness and immutability. In *3rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.
- [43] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Joint ACM-ISCOPE Java Grande Conference*, 2002.
- [44] Frank Pfenning and Carsten Schürmann. Twelf user's guide, version 1.4. Available at <http://www.cs.cm.edu/~twelf>, 2002.
- [45] William S. Retert. *Implementing Permission Analysis*. PhD thesis, University of Wisconsin–Milwaukee, Department of EE & CS, 2009.
- [46] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, 1978.
- [47] Mats Skoglund and Tobias Wrigstad. A mode system for readonly references. In *3rd ECOOP Workshop on Formal Techniques for Java Programs*, 2001.
- [48] Matthew Smith. Toward an effects system for ownership domains. In *7th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2005.
- [49] SureLogic. <http://surelogic.com>.
- [50] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA '05 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, 2005.
- [51] Yang Zhao. *Concurrency Analysis Based on Fractional Permissions*. PhD thesis, University of Wisconsin–Milwaukee, Department of EE & CS, 2007.