Inferring Aliasing and Encapsulation Properties for Java

Kin-Keung Ma
University of Maryland, College Park
kkma@cs.umd.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Abstract

There are many proposals for language techniques to control aliasing and encapsulation in object oriented programs, typically based on notions of object ownership and pointer uniqueness. Most of these systems require extensive manual annotations, and thus there is little experience with these properties in large, existing Java code bases. To remedy this situation, we present Uno, a novel static analysis for automatically inferring ownership, uniqueness, and other aliasing and encapsulation properties in Java. Our analysis requires no annotations, and combines an intraprocedural points-to analysis with an interprocedural, demand-driven predicate resolution algorithm. We have applied Uno to a variety of Java applications and found that some aliasing properties, such as temporarily lending a reference to a method, are common, while others, in particular field and argument ownership, are relatively uncommon. As a result, we believe that Uno can be a valuable tool for discovering and understanding aliasing and encapsulation in Java programs.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.11 [*Software Engineering*]: Software Architectures—Information hiding; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Measurement

Keywords Uno, Java, ownership, uniqueness, lending, encapsulation, aliasing, ownership inference, uniqueness inference

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction

Understanding and controlling aliasing is a fundamental part of building robust software systems. In recent years, researchers have proposed many systems that reason about various aliasing properties in programs, including *uniqueness* [1, 2, 7, 8, 15, 28, 31] and *ownership* [2, 3, 6, 11, 12, 16, 18, 24, 30]. Unique objects are those referred to by only one pointer, and thus are guaranteed unaliased with any other objects in the system. Owned objects are encapsulated inside of their owner, and hence cannot be directly accessed by other components.

Many of these systems include static checking of these properties in Java-like source code, but usually require that the programmer manually add extensive annotations. Moreover, while the properties modeled seem quite useful, it is unclear how often they occur in existing programs. To date, experience with using such systems on large software applications has either been with coarse analysis [16, 30] or via case studies [2].

In this paper, we present a novel tool called Uno[1] that fills this gap. Uno takes as input unannotated Java source code and infers uniqueness of method arguments and results; lending (temporary aliasing) of method arguments and receiver objects; and ownership and non-escaping of parameters and fields. These properties capture key aliasing and encapsulation behavior, and can give important insight into Java code. For example, a programmer might use Uno to check that a factory method always returns a unique object as expected, or that a proxied object is owned by its proxy, which therefore controls all access to it.

Uno performs inference using a novel two-phase algorithm. The first phase is an intraprocedural (within one function) may-alias analysis that computes local points-to information. Our alias analysis is mostly standard, but uses an interesting mix of flow-sensitive and flow-insensitive information. The second phase is a demand-driven interprocedural analysis that computes a set of mutually-recursive *predicates*. For example, for each method m, Uno determines whether the predicate $\textsc{UniqRet}(m)$ holds, meaning that m always returns a unique object when it is called. If m returns its ith argument, then $\textsc{UniqRet}(m)$ holds only

[1] Uniqueness aNd Ownership, http://www.cs.umd.edu/projects/PL/uno

if UNIQPAR(m, i) holds, meaning m is always called with a unique ith argument. Uno incorporates several other interdependent predicates that capture the aspects of aliasing and encapsulation mentioned above.

We have applied Uno to more than one million lines of Java code, including SPEC benchmarks, the DaCapo benchmarks [5], and larger programs found on SourceForge. Our goal was to demonstrate the utility of Uno, and to discover how often the ownership and encapsulation properties it infers actually occur in Java programs. We found that, on average across our benchmarks, the monomorphic ownership inferred by Uno holds for 16% of the private fields and only 2.7% of the arguments of called constructors. Somewhat surprisingly, Uno infers that more than 30% of all methods (constructors not included) that do not return a primitive return a unique value, and approximately 50% of all non-primitive method parameters are lent (i.e., only temporarily aliased by the method and not captured). Our results show that programmers do control aliasing and encapsulation in some ways suggested in the literature but less so in others, modulo the precision of Uno's sound but conservative analysis. To our knowledge, Uno is the first ownership and uniqueness inference tool that has been demonstrated on a wide variety of Java applications.

In summary, the contributions of this paper are:

- We describe a flow-sensitive, intraprocedural points-to analysis algorithm tuned to compute the information needed for evaluating Uno's predicates. (Section 3)
- We present a novel interprocedural algorithm that infers a range of aliasing and encapsulation properties. Our analysis is structured as a set of mutually-recursive predicates. The algorithm is demand-driven, so that only the predicates and points-to information necessary to answer a query are actually computed. (Section 4)
- We describe our implementation, Uno, and apply it to a number of benchmarks. Uno finds that some aliasing and encapsulation properties such as lending of arguments occur often, and other properties, such as monomorphic ownership, occur rarely. As a result, we believe that Uno is a valuable tool for discovering and understanding aliasing and encapsulation in Java. (Section 5)

2. Overview

We begin our presentation by illustrating Uno's core notions of uniqueness and ownership for methods and constructors, and by describing the key predicates Uno computes to perform inference.

Uniqueness We say that a pointer is *unique* if it is the only reference to the object it points to. Uniqueness is a very useful pointer property because its strong notion of non-aliasing permits modular reasoning [1, 2, 7, 8, 15, 28, 31].

Figure 1(a) illustrates one kind of uniqueness Uno infers. In this example, instances of ConcreteSubject (lines 4–7)

```
1  interface Subject {
2      void setData(int d);
3  }
4  class ConcreteSubject implements Subject {
5      private int data;
6      void setData(int d) { data = d; }
7  }
8  class Factory {
9      public Subject getSubject() {        // returns unique
10         Subject r = new ConcreteSubject();
11         Subject s = r;
12         return r;
13     }
14 }
```

(a) Uniqueness of method return

```
15 class Proxy implements Subject {
16     private Subject s;                    // owned by this
17     public Proxy(Subject s) {
18         this.s = s;
19     }
20     public void setData(int d) {
21         s.setData(d*d);
22     }
23 }
24 class Main {
25     public void main(Factory f) {
26         Subject t = f.getSubject();
27         t.setData(1);                     // uses t directly
28         Proxy proxy = new Proxy(t);       // proxy owns t
29         proxy.setData(2);      // t now used through proxy
30     }
31 }
```

(b) Ownership of method argument

Figure 1. Source code example

are created by getSubject (lines 9–12). Notice that when getSubject returns, the only other pointers to its result are r and s, both of which are dead at the method exit. Thus, Uno concludes the return value of getSubject is always unique.

Knowing a return value is unique can be useful because uniqueness typically implies the returned value is "fresh." This is particularly helpful in this example, when we are calling a factory method rather than a constructor, which is at least guaranteed to allocate a new object. Uno also checks uniqueness of constructor return values—a pathological constructor that stores this in a field of another object would violate uniqueness—and we found that all constructors in our experiments return a unique value.

In Section 4, we formally define a predicate UNIQRET(m) that describes the necessary conditions for method m to return a value that is unique when the method exits. Uno's inference algorithm is specified in terms of this and other predicates, and for a given input program and selection of predicates, Uno reports whether those predicates hold for the methods and constructors of interest.

Ownership Uno's notion of ownership is based on the flexible alias protection framework of Noble, Potter, Vitek, and Clarke [12, 24]. Their system uses a notion called *representation containment*, in which if object o *contains* or *owns* object p, then only o may access p. This system is designed to be more flexible than previous proposals [3, 18] in two important ways: First, an object need not own every object it refers to. For example, a container object might own the backbone of the container but not the elements themselves. Second, ownership is *polymorphic* or *context-sensitive*, which allows an owner to grant some objects it owns access to other objects it owns [12].

Uno uses a slight variation of this notion of representation containment for ownership, but in a monomorphic or context-insensitive form. We say that object o *owns* object p if o has the only pointer to p, and if only o may access p. In particular, the only reads and writes to p must either occur inside of o's methods, or inside of other methods that o calls, where those other methods have only transient pointers to p. Moreover, none of o's methods may *leak* p by returning it or storing it in a field of another object.

Ownership is a useful property because it enforces encapsulation. Owned objects are not accessible outside the owner, and thus the owner can safely assume that no other objects can manipulate them. For example, we might use a security proxy to perform access control before delegating to a proxied object. We can help ensure complete mediation by checking that the proxy owns the proxied object.

Figure 1(b) gives an example. This code extends Figure 1(a) with two new classes. Proxy (lines 15–23) has a private field s that stores instances of Subject, and Proxy's setData method delegates to s. Then the class Main has a main method that creates a new ConcreteSubject (line 26), uses it (line 27), creates a Proxy for it (line 28), and then uses it through the Proxy (line 29).

When Uno analyzes this code, it infers several things. By the uniqueness of getSubject's return value, we see that t is unique on line 26. Then on line 27, we invoke a method of t, but that method (line 6) does not change the uniqueness of t. When a unique object o is passed to a method that does not change its uniqueness—meaning the method has only transient pointers to o—we say that o is *lent* to the method. We define the predicate LENTTHIS(m) to mean that calling method m lends the receiver object to m, and LENTTHIS holds for our example method. We also define a corresponding predicate LENTPAR(m, i) to mean parameter number i is lent to method m.

Next, on line 28, we pass the unique pointer t to the Proxy constructor, and t is never used again in the caller. Thus we say that Proxy's first parameter is unique, and we define predicate UNIQPAR(m, i) to mean that method or constructor m is always called with a unique ith argument that is not used after the call. In essence, if UNIQPAR(m, i) holds, then the ith argument may become owned by the receiver object after the call.

Examining the Proxy class further, Uno observes that on line 18, s is stored in a private field. Furthermore, that field value never escapes—in particular, the only instance method, setData, does not leak s. For example, although s is passed on line 21 as the receiver object of ConcreteSubject.setData, that method does not capture the value of this. Thus predicate NESCFIELD(s) holds for Proxy, meaning that none of Proxy's methods leak private field s.

Finally, putting this all together, we see that the Proxy constructor has a unique argument that is stored in a private field and does not escape the method. We define two ownership predicates. First, OWNPAR(m, i) means that after calling the method or constructor m, the receiver object owns the ith argument that was passed in. Here OWNPAR(Proxy(Subject), 1) holds. Second, OWNFIELD(f) means that field f is encapsulated inside its containing object and has a unique reference to its contents, and in our example, OWNFIELD(s) holds.

Notice that ownership critically depends on uniqueness, since without it, we cannot reason about whether we have the sole pointer to an object. Of our two ownership predicates, OWNFIELD(f) is more standard, modeling ownership of a field f no matter where the contents of that field came from, including objects locally constructed within a class. The predicate OWNPAR(m, i), on the other hand, models another important kind of ownership, where we are concerned with which objects coming from outside of a method or constructor become owned. Previous ownership systems support this pattern [12, 24], though they do not call attention to it specifically. Another important aspect of Uno is that it supports flow-sensitivity, to allow unique objects to become owned later on in a method. In our example, line 27 in main uses t directly. Only after t is captured by the Proxy object on line 28 does it becomes owned by proxy. We believe this flexibility is important to allow objects to be initialized before they become owned.

Predicate Violations Next consider Figure 2, a slight modification of Figure 1 that shows several of the ways in which Uno's predicates may fail to hold.

In this code, getSubject (lines 10–14) no longer returns a unique value because on line 12, r is stored in a public field, and hence there are multiple pointers to the return value of getSubject after it returns.

Furthermore, the bad method on lines 21–25 falsifies both NESCFIELD(s) and OWNFIELD(s), in several ways. On line 22, s is stored in some class Other (not shown), and thus s leaks from Proxy. Line 23 accesses the s field of a different Proxy object, causing s to leak again. On line 24, the value of s is returned by the method, and since the method is public, that means s might leak yet again. The middle case illustrates a key difference between ownership and Java's private keyword. In Java, a private field can still

```
1  interface Subject {
2      void setData(int d);
3  }
4  class ConcreteSubject implements Subject {
5      private int data;
6      void setData(int d) { data = d; }
7  }
8  class Factory {
9      public Subject cur;
10     public Subject getSubject() { // should return unique
11         Subject r = new ConcreteSubject();
12         cur = r;                          // violates UniqRet
13         return r;
14     }
15 }
16 class Proxy implements Subject {
17     private Subject s;                    // not owned by this
18     public Proxy(Subject s) {
19         this.s = s;
20     }
21     public Subject bad(Proxy p) {
22         Other.f = s;                      // violates NEscField
23         p.s.setData(1);                   // violates NEscField
24         return s;                         // violates NEscField
25     }
26     public void setData(int d) {
27         s.setData(d*d);
28     }
29 }
30 class Main {
31     public void main(Factory f) {
32         Subject t = f.getSubject();
33         t.setData(1);                                  // valid
34         Proxy proxy = new Proxy(t); // proxy should own t
35         proxy.setData(2);                              // valid
36         t.setData(2);                     // violates UniqPar
37     }
38 }
```

Figure 2. Violations of predicates

```
1  public class ShimpleOptions {
2      private Map options;
3      public ShimpleOptions(Map options) {      // owns options
4          this.options = options;
5      }
6  }
7  public class ShimpleBody extends StmtBody {
8      protected ShimpleOptions options;
9      ShimpleBody(Body body, Map options) {
10         this.options = new ShimpleOptions(options);
11     }
12 }
13 public class Shimple {
14     public static final String PHASE = "shimple";
15     public ShimpleBody newBody(Body b) {              // unique
16         Map options = PhaseOptions.v().getPhaseOptions(PHASE);
17         return new ShimpleBody(b, options);
18     }
19 }
20 public class PhaseOptions {
21     public Map getPhaseOptions(String phaseName) { // unique
22         return getPhaseOptions(getPM().getPhase(phaseName));
23     }
24     public Map getPhaseOptions(HasPhaseOptions phase) {
25         return Collections.unmodifiableMap(ret);       // unique
26     }
27 }
28 public class Collections {
29     public static Map unmodifiableMap(Map m) {       // unique
30         return new UnmodifiableMap(m);
31     }
32 }
```

Figure 3. Example from Soot

be accessed by a different instance of the same class. In our experiments, we found that approximately 40% of private fields may leak in this way, using the fairly coarse analysis discussed in Section 3.

Lastly, on line 36, t is used directly after being passed to the Proxy constructor on line 34, and thus that constructor is no longer always called with a unique argument. Since at least one (in fact several) conditions for ownership are violated, Uno concludes that the Proxy constructor no longer owns its argument.

This example demonstrates that aliasing and encapsulation properties can be subtle to check and are interdependent. In Section 5.3, we present measurements on which predicates most often cause ownership to fail to hold.

Ownership and Uniqueness in Practice Finally, for a more complex example, consider the code in Figure 3, which is extracted and simplified from Soot [29], one of our benchmarks. In this example, Uno infers that the ShimpleOptions constructor (lines 3–5) owns its argument, as follows. First, Uno determines on lines 29–31 that unmodifiableMap returns a unique object. Thus so does getPhaseOptions(HasPhaseOptions) (lines 24–26), and therefore so does getPhaseOptions(String) (lines 21–23). Thus on line 16, Uno infers that options points to a unique object. This unique object is passed to the ShimpleBody constructor declared on line 9, which in turn passes it to the ShimpleOptions constructor declared on line 3, which stores it into a private field on line 4. The remaining code (not shown) does not cause the field to leak. Putting this all together, ShimpleOptions owns its field options, and the ShimpleOptions(Map) constructor owns its argument.

As this example shows, we often need to follow sequences of calls to infer uniqueness and ownership. Section 5 includes sample output from Uno showing this chain of reasoning on the Soot code corresponding to Figure 3.

3. Points-to Analysis

The first step of Uno's inference algorithm is an intraprocedural points-to analysis that determines aliasing information within each method. We use a *may* points-to analysis, so that the *points-to sets* determined by our algorithm describe a superset of all possible run-time objects that may be pointed

RET	Objects returned by this method	FLD(f)	Objects pointed to by field f of this
PAR(ℓ, i)	Objects passed to ith parameter of ℓ	BAD	Objects pointed to externally
THIS(ℓ)	Receiver objects of ℓ	BADFLD	Private fields accessed by other instances of the same class, and all non-private fields
LIVE(ℓ)	Objects live after ℓ		
SUPTHIS(i)	ith parameters of super(...) or this(...)		

Statement	Transfer Function	Points-to Set Constraints
Method entry	$Out(x) = \begin{cases} \{\ell_{pi}\} & x \text{ is } i\text{th param} \\ \{\ell_{this}\} & x \text{ is this} \\ \emptyset & \text{otherwise} \end{cases}$	FLD(f) $\supseteq \{\ell_f\}$ $\forall f$ in class BAD $\supseteq \{\ell_{bad}\}$ BADFLD $\supseteq$ {non-private fields}
$x_1 = x_2$	$Out(x_1) = In(x_2)$	—
$x = \mathsf{new}^\ell\ C(x_1, \ldots, x_n)$	$Out(x) = \{\ell\}$	PAR(ℓ, i) $\supseteq In(x_i)$ $i \in 1..n$ LIVE(ℓ) $\supseteq Out(x)$ $\forall$live x

$x_1.f = x_2$	—	if $x_1 =$ this then
		  FLD(f) $\supseteq In(x_2)$
		else
		  BAD $\supseteq In(x_2)$
		  if *class*(x_1) = *class*(this) then
		  BADFLD $\supseteq \{f\}$

$x_1 = x_2.f$	$Out(x_1) = \begin{cases} \text{FLD}(f) & \text{if } x_2 = \text{this} \\ \text{BAD} & \text{otherwise} \end{cases}$	if $x_2 \neq$ this $\wedge$ *class*(x_2) = *class*(this) then BADFLD $\supseteq \{f\}$
$C.f = x$	—	BAD $\supseteq In(x)$
$x = C.f$	$Out(x) =$ BAD	—
$x_r = x_0.m^\ell(x_1, \ldots, x_n)$	$Out(x_r) = \{\ell\}$	PAR(ℓ, i) $\supseteq In(x_i)$ $i \in 1..n$ THIS(ℓ) $\supseteq In(x_0)$ LIVE(ℓ) $\supseteq Out(x)$ $\forall$live x
super($x_1, \ldots, x_n$)	—	SUPTHIS(i) $\supseteq In(x_i)$ $i \in 1..n$
this($x_1, \ldots, x_n$)	—	SUPTHIS(i) $\supseteq In(x_i)$ $i \in 1..n$
return x	—	RET $\supseteq In(x)$

Figure 4. Intraprocedural points-to analysis

to by variables and fields. Our analysis includes both flow-sensitive and flow-insensitive components, discussed below.

We formalize our points-to analysis by associating each syntactic allocation site in the program with a fresh *label* ℓ that represents objects constructed at that site. Thus points-to sets are sets of labels. For example, if we see an assignment $x = \mathsf{new}^\ell\ C(x_1, \ldots, x_n)$ in the program, we add ℓ to the points-to set for x.

In an interprocedural points-to analysis, we would only need to label occurrences of new, and then propagate those labels throughout the whole program. However, since our points-to analysis is intraprocedural and thus only operates on one method at a time, we also need labels for objects that come from "outside" the method we are analyzing. For each method parameter x_i, we create a label ℓ_{pi} to represent the object initially pointed to by x_i. Similarly, we use label ℓ_f for the initial contents of this.f, label ℓ_{bad} for the initial contents of any other field, and label ℓ_{this} for the object stored in this. Lastly, we also label method invocations as $x.m^\ell(\ldots)$, and use ℓ to represent objects returned by the call.

To infer aliasing and encapsulation properties, we need to determine how objects in the program flow through each method. For example, to decide whether a method returns a unique object, we need to compute the set of objects that may be returned by the method and ensure there are no other pointers to them when the method exits—e.g., the returned object was not stored in a field.

To support this process, our points-to analysis computes a series of flow-insensitive sets, summarized at the top of Figure 4. By *flow-insensitive*, we mean that there is only one copy of the set for the entire method body. This is in contrast to our modeling of local variables, which is flow-sensitive, allowing variables' points-to sets to change from one program point to the next (see below).

The flow-insensitive set RET tracks the objects returned by the current method. For each method invocation labeled ℓ in the current method, PAR(ℓ, i) tracks the objects passed in as the ith argument, THIS(ℓ) tracks the possible receiver objects, and LIVE(ℓ) tracks the objects pointed to by local variables that are live after the call. We do not treat calls to super or this constructors as method invocations, in order to handle certain special cases for constructors. Thus if we see a super or this constructor call, we store the points-to set of the ith parameter in SUPTHIS(i) (rather than as a PAR set). Note that a constructor includes exactly one (possibly implicit) super or this constructor call, and the call must be the first action of the constructor [4].

The flow-insensitive set FLD(f) tracks the objects stored in this.f. Any object written to a field not of this (i.e., the write is not of the form this.$f = \ldots$), or to a static field, is added to the set BAD, rather than trying to track other objects' fields precisely. This greatly simplifies our points-to analysis, since it means that anything stored outside of the current object in the heap is aliased to everything in BAD. In

Section 4, we assume that any object in BAD is not unique and escapes.

The last set, BADFLD, is a set of field names rather than a points-to set. This set contains fields that may be accessed by other instances. It contains all non-private fields, along with private fields that the current method accesses from other instances of the same class as this. Recalling line 24 of Figure 2, we use this set to find fields that are private but are accessed from outside the object containing them. Note that BADFLD is a global set, shared across all points-to analyses.

The bottom of Figure 4 summarizes the dataflow analysis we use to compute points-to information. The various kinds of statement are listed in the left column, and are derived from the Jimple [29] representation of Java bytecode, which we use in our implementation. We omit some language features such as arrays, which are discussed in Section 5. Our analysis tracks information for local variables flow-sensitively, maintaining a mapping *Out* that gives the points-to set for local variables just after each statement executes. We use *In* to stand for the union of the *Out* maps of all preceding statements. For brevity, $\mathit{Out}(x) = \mathit{In}(x)$ for all x unless stated otherwise.

For each statement, we list the corresponding transfer function and constraints on the flow-insensitive points-to sets. At the method entry, we set *Out* so that each formal parameter x_i points to ℓ_{pi}, this points to ℓ_{this}, and other variables' points-to sets are empty. We add ℓ_f to $\text{FLD}(f)$ for each f, we set BAD to contain ℓ_{bad}, and we add any non-private fields to BADFLD.

At an assignment statement, we copy the right-hand side points-to set to the left-hand side, with no effect on the flow-insensitive sets. For a constructor call, we set the left-hand side to point to the label of the call, and we include the points-to set of each argument in PAR, and add to LIVE the set of objects pointed to by live local variables after the call. We omit the live variable computation, since it is standard.

When writing to a field of this, we add the points-to set of the right-hand side to $\text{FLD}(f)$. When writing to a field of any other object, we instead add to the set BAD. Additionally, if the object whose field is written has the same class as this, then we add f to the set BADFLD. When reading from an instance field, we set the points-to set of the left-hand side either to $\text{FLD}(f)$ or BAD, as appropriate, and add f to BADFLD if needed. Static fields are shared by all instances of the class, and so any objects pointed to by them are added to BAD. To keep notation simpler, we assume that fields are not inherited. In our implementation, inherited fields are in BAD, so they always escape and are not unique.

For method and super or this constructor calls, we add the points-to sets of the arguments to PAR or SUPTHIS, as appropriate. For method calls, the points-to set of the left-hand side contains the returned label, and we constrain LIVE and THIS appropriately. Lastly, for method return we add the points-to set of the returned variable to RET.

Resolution Algorithm The points-to analysis described in Figure 4 could be implemented using an iterative fixpoint algorithm. However, we found that the fixpoint algorithm was too slow for large benchmarks, because it computed information that was not necessary to infer Uno's predicates.

Thus our implementation instead uses a lazy, constraint-based solving algorithm. We represent each points-to set A as a *lazy set* $\{\ell_1, \ldots, A_1, \ldots\}$, consisting of labels ℓ_i and other points-to sets A_i. For a transfer function of the form $A = B$, we set A to $\{B\}$. For a constraint $A \supseteq B$, we add B to the lazy set for A (i.e., A is now of the form $\{\ldots, B\}$). For an *In* set, which is a union of the form $\bigcup \mathit{Out}_i$ (recall this is not shown in Figure 4), we set $\mathit{In} = \{\mathit{Out}_1, \ldots\}$.

Later on, when we need the contents of a set, we flatten it on demand, where $\mathit{flatten}(\{\ell_1, \ldots, A_1, \ldots\}) = \{\ell_1, \ldots\} \cup \bigcup_j \mathit{flatten}(A_j)$. Flattening a lazy set can be time consuming, and if we needed to flatten all sets this approach would be too slow. However, we discovered that most sets computed in the points-to analysis are not needed by the predicate resolution algorithm. In particular, any flow sensitive information (e.g., $Out(x)$) is not used directly in predicate resolution. Thus we have found that laziness in the points-to analysis greatly improves the running time of our algorithm.

4. Predicate Inference

The second step of our inference algorithm is an interprocedural analysis that determines aliasing and encapsulation properties of methods and constructors. As discussed in Section 2, we specify our analysis as a set of mutually-recursive predicates. For presentation purposes, we split the predicates into two groups, but in practice, Uno computes all predicates simultaneously.

Our definitions of the predicates depend on the various flow-insensitive points-to sets computed by the algorithm in Section 3. Thus, we believe that any other points-to analysis (e.g., a more precise one [20, 26, 27, 32]) that could be modified to produce the same summary information could be integrated into Uno without difficulty.

4.1 Uniqueness Predicates

Figure 5 defines the predicates related to uniqueness, most of which we saw earlier in Section 2. There are five main predicates, described in the left column. $\text{UNIQRET}(m)$ is the basic uniqueness predicate, which holds if method m always returns a unique object. To cut down on verbiage, throughout the rest of this section we use the word *method* and the symbol m to mean either a method or a constructor. If m is a constructor, $\text{UNIQRET}(m)$ holds if the newly constructed object is unique after the constructor call. $\text{LENTPAR}(m, i, \textit{no-fld})$ holds if calling method m does not affect the uniqueness of its ith argument, i.e., if the ith argument is lent to m. Here the flag *no-fld* (omitted earlier for simplicity) is true when checking ordinary method calls, and is false in certain cases of checking calls to super or this constructors, as discussed

subs(m)	methods that override m	*supthis(m)*	super or this constructor called by m
sups(m)	methods m overrides	*mth(ℓ)*	method invoked by call ℓ
callee(m)	methods called by m	*before*(ℓ)	labels of calls that happen before ℓ

$a \bowtie b \equiv (a \cap b = \emptyset), a \not\bowtie b \equiv (a \cap b \neq \emptyset)$

Predicate	Local Constraints	Non-local Constraints
UNIQRET(m) Method m returns a unique object	(L1) RET $\bowtie$ BAD (L2) RET $\bowtie \{\ell_{this}\}$ (L3) RET $\bowtie$ FLD(f)	(1) UNIQRET(*subs(m)*) (2) UNIQPAR($m, i,$ *true*) if RET $\not\bowtie \{\ell_{pi}\}$ (3) UNIQRET(*mth(ℓ)*) if RET $\not\bowtie \{\ell\}$ (4) LENTPAR(*mth(ℓ)*, j, *true*) if RET $\not\bowtie$ PAR(ℓ, j) (5) LENTTHIS(*mth(ℓ)*) if RET $\not\bowtie$ THIS(ℓ) (6) LENTTHIS(m) if m is constructor
LENTPAR($m, i,$ *no-fld*) Calling method m does not change the uniqueness of its ith argument	(L1) $\{\ell_{pi}\} \bowtie$ BAD (L2) $\{\ell_{pi}\} \bowtie$ RET (L3) If *no-fld* then $\{\ell_{pi}\} \bowtie$ FLD(f)	(1) LENTPAR(*subs(m)*, i, *no-fld*) (2) LENTPAR(*mth(ℓ)*, j, *true*) if $\{\ell_{pi}\} \not\bowtie$ PAR(ℓ, j) (3) LENTTHIS(*mth(ℓ)*) if $\{\ell_{pi}\} \not\bowtie$ THIS(ℓ) (4) LENTPAR(*supthis(m)*, j, *no-fld*) if $\{\ell_{pi}\} \not\bowtie$ SUPTHIS(j)
LENTTHIS(m) Calling method m does not change the uniqueness of the receiver object	(L1) $\{\ell_{this}\} \bowtie$ BAD (L2) $\{\ell_{this}\} \bowtie$ RET (L3) $\{\ell_{this}\} \bowtie$ FLD(f)	(1) LENTTHIS(*subs(m)*) (2) LENTPAR(*mth(ℓ)*, j, *true*) if $\{\ell_{this}\} \not\bowtie$ PAR(ℓ, j) (3) LENTTHIS(*mth(ℓ)*) if $\{\ell_{this}\} \not\bowtie$ THIS(ℓ) (4) LENTPAR(*supthis(m)*, j, *true*) if $\{\ell_{this}\} \not\bowtie$ SUPTHIS(j) (5) LENTTHIS(*supthis(m)*)
UNIQPAR($tgt, i,$ *no-fld*) Method tgt's ith parameter is always unique	—	(1) UNIQPAR(*sups(tgt)*, i, *no-fld*) (2) UNIQPAR-IN($m, i, tgt,$ *true*) if $tgt \in$ *callee(m)* (3) UNIQPAR-IN($m, i, tgt,$ *no-fld*) if $tgt \in$ *supthis(m)*
UNIQPAR-IN($m, i, tgt,$ *no-fld*) Method m always passes a unique object as the ith parameter when it calls tgt	$\forall \ell$ s.t. *mth(ℓ)* $= tgt$ (L1) PAR(ℓ, i) $\bowtie$ BAD (L2) PAR(ℓ, i) $\bowtie \{\ell_{this}\}$ If *no-fld* then (L3) PAR(ℓ, i) $\bowtie$ LIVE(ℓ) (L4) PAR(ℓ, i) $\bowtie$ FLD(f) (L5) PAR(ℓ, i) $\bowtie$ PAR(ℓ, j), $j \neq i$ (L6) PAR(ℓ, i) $\bowtie$ THIS(ℓ)	$\forall \ell$ s.t. *mth(ℓ)* $= tgt$ (1) UNIQPAR($m, j,$ *true*) if PAR(ℓ, i) $\not\bowtie \{\ell_{pj}\}$ (2) UNIQRET(*mth*(ℓ_2)) if PAR(ℓ, i) $\not\bowtie \{\ell_2\}$ $\forall \ell_2 \in$ *before*(ℓ) . (3) LENTPAR(*mth*(ℓ_2), j, *true*) if PAR(ℓ, i) $\not\bowtie$ PAR(ℓ_2, j) (4) LENTTHIS(*mth*(ℓ_2)) if PAR(ℓ, i) $\not\bowtie$ THIS(ℓ_2) (5) LENTPAR(*supthis(m)*, j, *no-fld*) if PAR(ℓ, i) $\not\bowtie$ SUPTHIS(j)

Figure 5. Predicates related to uniqueness

below. LENTTHIS(m) holds if calling m does not change the uniqueness of the receiver object. (We could combine LENTTHIS and LENTPAR into one predicate, but keep them distinct for expository purposes, and because our experimental results show that this is often treated differently than parameters.) Lastly, UNIQPAR($tgt, i,$ *no-fld*) holds if method tgt is always called with a unique ith argument. This predicate is defined in terms of UNIQPAR-IN($m, i, tgt,$ *no-fld*), which checks the same property but for the calls to tgt inside of m. In both cases, *no-fld* is used the same as in LENTPAR.

The middle and right columns of Figure 5 give a set of *conditions* that must hold for the predicate to be true. We use several conventions in the figure to simplify notation. Almost all of the conditions involve checking whether various points-to sets are disjoint, and we write $a \bowtie b$ to mean $a \cap b = \emptyset$, and $a \not\bowtie b$ to mean $a \cap b \neq \emptyset$. In general, we use m and tgt to range over methods and constructors, i and j for parameter numbers, and f for fields. Many of the predicates are parameterized by an argument m, and any points-to set mentioned in a predicate comes from the analysis of m.

We also use a number of sets when defining the predicates, as summarized at the top of Figure 5. We use *subs(m)* and *sups(m)* for the set of methods that override m and that m overrides, respectively. We write *callee(m)* for the set of methods that m may call according to the compile-time types, and we write *supthis(m)* for the super or this constructor called in m; this is only defined if m is a constructor, and otherwise predicates depending on *supthis(m)* are ignored. For a label ℓ corresponding to a method call, we use *mth*(ℓ) to denote the method invoked by the call ℓ, according to the compile-time types. Lastly, we use *before*(ℓ) for the set of labels ℓ' such that there is a path from ℓ' to ℓ in the control-flow graph, i.e., ℓ' may happen before ℓ.

We divide the conditions into two parts. *Local* conditions are those that can be decided just from the intraprocedural alias analysis results for the method in question, and thus do not depend on other methods. For example, UNIQRET(m) requires that any return value of m not be in BAD, since a BAD value might have come from a field of another object. Thus we require RET $\bowtie$ BAD.

Non-local conditions are those that require recursively checking predicates for other methods. For example, if method m_1 returns the result of method m_2, then to decide whether m_1's result is unique, we need to check uniqueness of m_2's result. Each recursively-checked predicate may have side-conditions that describe exactly what must be

checked. For example, UNIQRET(m) recursively checks (3) UNIQRET(*mth*(ℓ)) where side-condition RET $\not\bowtie \{\ell\}$ holds, i.e., UNIQRET is recursively checked for all ℓ that are included in RET. Note the implicit universal quantification here—to keep the conditions readable, we assume that any free variables not defined by the predicate range over all reasonable values (e.g., j ranges over parameter numbers, ℓ ranges over invocations, f ranges over fields, etc.)

For convenience in discussing the predicates and in evaluating Uno, we number all of the conditions. We next discuss the predicates in more depth.

UNIQRET(m) For this predicate to hold, the returned object must not be reachable in any way except via the return value of the call. Thus nothing in BAD (L1) or $\{\ell_{this}\}$ (L2) may be in RET, since objects in BAD may be pointed to by other objects, and we assume this is not unique. The returned object must also not be pointed to by a field (L3). Note the implicit quantification in (L3)—we require this condition holds for all fields f of method m.

We also need to account for objects that come from outside method m. Any parameter ℓ_{pi} in RET must be unique (2), meaning that the caller does not keep a pointer to that object, and similarly any method label ℓ in RET must come from a method that returns a unique object (3). Furthermore, any object in RET that is passed in to a call, either as a method argument (in some PAR(ℓ, j), (4)), or as a receiver object (in THIS(ℓ), (5)) must not have its uniqueness changed by the call, meaning it must have been lent to the callee. If m is a constructor, it must not change the uniqueness of this (6), which is not included in RET. Finally, a call to m at compile time might at run time invoke a method that overrides m. Hence m can have a unique return value only if all methods that override it do also (1).

LENTPAR($m, i,$ *no-fld*) For method m not to change the uniqueness of its ith argument, represented by ℓ_{pi} in the alias analysis, it must be that ℓ_{pi} not appear in BAD (L1), be returned by m (L2), or be stored in a field (L3). If *no-fld* is false, we omit the last check. We use this feature when testing LENTPAR for a call to a super or this constructor. In these cases, the calls by definition are received by the same object as the caller, and so if such a call stores a parameter in a local field, we still treat the argument as if it were lent (since it is stored in the same object).

For the non-local predicates, LENTPAR requires that if ℓ_{pi} is passed to a method (2) or super or this constructor (4), or has one of its methods invoked (3), then those calls must not change its uniqueness. Finally, any methods that override m must also not change parameter i's uniqueness (1).

LENTTHIS(m) This predicate is analogous to LENTPAR, except it checks properties of ℓ_{this} instead of ℓ_{pi}. There is no *no-fld* flag, since storing ℓ_{this} in a field always makes it non-unique, and there is an extra check (5) for a call to a super or this constructor, since such calls are invoked on ℓ_{this} as well.

UNIQPAR($tgt, i,$ *no-fld*) This predicate and UNIQPAR-IN are the most complex of the uniqueness predicates. The base predicate, UNIQPAR, must check that all callers to tgt pass a unique object as tgt's ith argument. Similarly to the other predicates, we first must ensure the same property holds for methods that tgt overrides (1), since tgt may be called in place of methods it overrides. Then we check UNIQPAR-IN($m, i, tgt,$ *no-fld*) for all m that are methods (2) or constructors (3) that may call tgt. For the former, we set *no-fld* to true, since we assume the call may be received by another object, and for the latter *no-fld* remains the same.

In turn, UNIQPAR-IN examines all calls ℓ inside of m that invoke tgt. We want to ensure that each call always passes a unique argument to position i. Thus PAR(ℓ, i) must not intersect BAD (L1) or this (L2), and nothing in PAR(ℓ, i) can be live after the call (L3) or be stored in a field (L4). We relax the last two checks if *no-fld* is false, to allow calls to super or this constructors to retain pointers across a call or store parameters in the current object. We also require that the points-to set PAR(ℓ, i) for parameter i not overlap any other parameter's points-to set (L5) or the receiver object of the call (L6), since then that parameter may be aliased, and therefore not unique, inside of the callee.

UNIQPAR-IN also requires that any parameters ℓ_{pj} or method return values ℓ_2 in PAR(ℓ, i) must themselves have been unique (1–2). We also ensure that for any calls ℓ_2 that happen before the call ℓ, if any labels in PAR(ℓ, i) are passed to those calls—either as a parameter PAR(ℓ_2, j) or as a receiver object THIS(ℓ_2)—then their uniqueness must not have been changed by the call ℓ_2 (3–4). This ensures that only one call in a method can transfer uniqueness of an object to another method. We also check this condition for parameters passed to super or this constructor calls (5), which, if they exist, occur at the beginning of the constructor.

4.2 Ownership Predicates

Figure 6 defines the remaining predicates, which focus on ownership and encapsulation. OWNPAR(m, i) holds if method m owns its ith argument. NESCPAR(m, i) holds if the ith parameter of method m does not leak from the object via a call to m. NESCFIELD(f) holds if field f does not leak from the object, and NESCFIELD-IN(f, m) holds if f does not leak via method m. A refinement of NESCFIELD, OWNFIELD(f) holds if f is owned by the object, meaning it does not escape and contains a unique pointer, and OWNFIELD-IN(f, m) holds if field f is owned locally within method m. Lastly, STORE(m, i) holds if method m stores parameter i in some field. In more detail, the predicates are:

OWNPAR(m, i) As discussed in Section 2, an argument to a method or constructor is owned if it becomes fully encapsulated inside its owner after the call. Thus for ownership of the ith argument to hold, it must be unique (2), so that this method or constructor can acquire ownership. We call

Predicate	Local Constraints	Non-local Constraints
OWNPAR(m, i) Method m owns its ith argument	—	(1) OWNPAR(*subs(m)*, i) (2) UNIQPAR(m, i, *false*) (3) NESCPAR(m, i) If *supthis(m)* $= \emptyset$ or $\nexists j$ s.t. SUPTHIS(j) $\not\bowtie \{\ell_{pi}\}$ (4) then STORE(m, i) (5) else OWNPAR(*supthis(m)*, j) if SUPTHIS(j) $\not\bowtie \{\ell_{pi}\}$
NESCPAR(m, i) The ith parameter of method m does not escape	(L1) $\{\ell_{pi}\} \bowtie$ BAD (L2) $\{\ell_{pi}\} \bowtie$ RET	(1) NESCPAR(*subs(m)*, i) (2) LENTPAR(*mth(ℓ)*, j, *true*) if $\{\ell_{pi}\} \not\bowtie$ PAR(ℓ, j) (3) LENTTHIS(*mth(ℓ)*) if $\{\ell_{pi}\} \not\bowtie$ THIS(ℓ) (4) LENTPAR(*supthis(m)*, j, *false*) if $\{\ell_{pi}\} \not\bowtie$ PAR(ℓ, j) (5) NESCFIELD(f) if $\{\ell_{pi}\} \not\bowtie$ FLD(f)
NESCFIELD(f) Field f does not escape	(L1) f is private	(1) NESCFIELD-IN(f, m) if method m uses f
NESCFIELD-IN(f, m) Field f does not escape in method m	(L1) FLD(f) $\bowtie$ BAD (L2) FLD(f) $\bowtie$ RET (L3) $f \notin$ BADFLD	(1) LENTPAR(*mth(ℓ)*, j, *true*) if FLD(f) $\not\bowtie$ PAR(ℓ, j) (2) LENTTHIS(*mth(ℓ)*) if FLD(f) $\not\bowtie$ THIS(ℓ) (3) LENTPAR(*supthis(m)*, j, *true*) if FLD(f) $\not\bowtie$ SUPTHIS(j)
OWNFIELD(f) Field f is owned by this object	—	(1) NESCFIELD(f) (2) OWNFIELD-IN(f, m) if method m uses f
OWNFIELD-IN(f, m) Field f is owned by this object within method m	(L1) FLD(f) $\bowtie \{\ell_{this}\}$	(1) UNIQPAR(m, j, *false*) if FLD(f) $\not\bowtie \{\ell_{pj}\}$ (2) OWNFIELD(g) if FLD(f) $\not\bowtie$ FLD(g) (3) UNIQRET(*mth(ℓ)*) if FLD(f) $\not\bowtie \{\ell\}$
STORE(m, i) Method m stores its ith parameter in a field	(L1) ($\exists f$ s.t. FLD(f) $\not\bowtie \{\ell_{pi}\}$) $\vee$ ($\exists j$ s.t. SUPTHIS(j) $\not\bowtie \{\ell_{pi}\}$)	(1) STORE(*subs(m)*, i)

Figure 6. Predicates related to ownership

UNIQPAR with *no-fld* set to false to allow m to own its argument even if a super or this caller retains a reference to the argument, since both calls are received by the same object. The ith argument also must not escape (3), so that the owned object is contained inside of the owner.

To suppress some vacuous cases of argument ownership, we also require that the ith parameter is either stored in some field (4) or passed to a super or this constructor that owns it (5), although our check for this is heuristic, as discussed below. Finally, any method that overrides m must have the same ownership behavior (1).

NESCPAR(m, i) This predicate requires that the ith parameter, represented by ℓ_{pi}, not appear in BAD (L1) and not be returned by the method (L2), since either would cause ℓ_{pi} to escape. As usual, this predicate must hold for all methods that override m (1), since they may be called in place of m. If ℓ_{pi} is used in a call, either as an argument (2) or as a receiver (3), then that call must not change its uniqueness—except that a call to a super or this constructor may capture ℓ_{pi}, hence for that case we call LENTPAR with *no-fld* as false (4). Finally, if ℓ_{pi} is stored in a field, then that field cannot escape this object (5).

NESCFIELD(f) This predicate checks that f is private (L1) and checks NESCFIELD-IN(f, m) for all methods m that use f (1). This predicate in turn ensures that nothing in FLD(f) may be in BAD or RET (L1–L2), and that f is not in BADFLD (L3), so that it cannot leak via a different instance of the same class. Furthermore, anything in FLD(f) from a method invocation or constructor call must have been unique (1), and anything in FLD(f) that is passed to a call must not have its uniqueness changed by the call (2–3).

OWNFIELD(f) This predicate requires that field f not escape (1), and also requires that f is locally owned within each method m that refers to f. The latter is checked by predicate OWNFIELD-IN(f, m), which requires that this not be stored in f (L1), and that only unique or otherwise owned objects (1–3) are stored in f.

STORE(m, i) This predicate checks whether ℓ_{pi}, which represents the ith parameter, may be pointed to by some field or was passed to a call to a super or this constructor. (Note that OWNPAR(m, i), which uses this predicate, ensures that the called constructor owns the argument.) Since we use a may-alias analysis, STORE(m, i) is a heuristic—we might think m stores its ith argument when it actually does not at run time. Nevertheless, we have found this predicate useful in practice for eliminating uninteresting cases of ownership.

```
Init: ∀p. visited(p) = false, Val(p) = true

RESOLVE(p) =
  if visited(p)
    return
  end if
  visited(p) = true
  if p can be determined false locally
    Val(p) = false
  else
    for each s ∈ {predicates p depends on} do
      RESOLVE(s)
      if Val(s) = false
        break
      end if
    end for
  end if
  if Val(p) = false
    FALSIFY(p)
  end if

FALSIFY(p) =
  Val(p) = false
  for each q that directly depends on p such that Val(q) = true do
    FALSIFY(q)
  end for
```

Figure 7. Predicate resolution algorithm.

4.3 Predicate Resolution

To compute whether the predicates hold, we can think of each predicate p as a node in a graph, with an edge from p to q if p *depends* on q, meaning that p uses q in one of its non-local conditions. Then we can check whether p is false by performing a forward search from p, looking for a node whose local conditions are false. If such a node exists then p is false, and otherwise it is true.

Figure 7 gives a depth-first search variant RESOLVE(p) to check whether p holds. The algorithm uses a map *Val* from predicates to truth values, and initially *Val*(p) is set to true for all p. The algorithm also keeps a flag *visited*(p) that indicates whether we have already tried to resolve predicate p, to stop the search from revisiting predicates. During resolution, visited nodes have had their local conditions checked, and non-visited nodes have not.

To check whether p holds, the algorithm traverses the predicate dependency graph. If p has been visited before, we exit. Otherwise we mark p as visited and check p's local conditions. If they show that p is false, we update *Val* accordingly. Otherwise we resolve each predicate p depends on. We stop iteration at the first predicate that is false—by the last step of the algorithm (below), that predicate being false has already caused *Val*(p) to be set false. After computing the value of predicate p, if we determine it was false, we invoke FALSIFY(p) to find all other predicates that p depends on and mark them false as well, pruning the graph traversal if we encounter a false predicate. If after this algorithm p is not set to false, it remains true.

The key feature of this algorithm is short-circuiting recursively computing predicates. As soon as one predicate p depends on is discovered to be false, there is no need to check the other predicates p depends on. We found that when Uno is used to compute a partial set of predicates, this feature can improve the running time of predicate resolution. When we attempted to turn off lazy sets in our experiments, we found that the larger experiments no longer completed, even if given several days to run.

5. Implementation and Experiments

Uno is implemented using the Soot Java analysis framework [29]. Soot operates on Java class files, translating them into Jimple, a typed 3-address intermediate representation that uses instructions similar to those in Figure 4.

To analyze the full Java language, Uno needs to handle some language features we have not discussed. We treat arrays and their contents as BAD, which is conservative but sound, because it causes those objects to be treated as non-unique and escaping. We do the same for native method arguments and results, though we cannot be fully sound for native methods since they may carry out arbitrary operations. Java type casts are ignored by our points-to analysis, since they do not change the object stored in a reference. We also make two unsound assumptions in our analysis. First, we analyze code in exception handlers, but do not track aliasing through exceptions, or the uniqueness or ownership of objects that are thrown. We also do not model reflection API calls specially. We leave soundly handling these features to future work; for example, the work of Livshits et al [23] can be used to remove reflection from programs.

Uno begins by performing the points-to analysis from Section 3. Since our points-to analysis is demand-driven, the contents of the points-to sets are not computed until they are demanded by the second step the algorithm, which resolves the predicates from Section 4. Recall that the predicates involve some additional sets. The sets *subs*(m), *sups*(m), *supthis*(m), and *callee*(m) can be determined trivially from the call graph and class hierarchy, and the set *before*(ℓ) can be easily computed from the control-flow graph.

Once we compute these sets, we use the predicate resolution algorithm from Section 4 to infer ownership and uniqueness. As mentioned earlier, that algorithm is demand-driven, so that it does not compute any more points-to sets or predicates than it must. In our experiments, we ran Uno exhaustively, to compute all predicates for all methods and constructors, but Uno can also be used selectively. For example, we could use Uno to find all methods that return unique objects, or to find all constructors that own their arguments.

Since Uno's predicates are somewhat complex and have many interdependencies, understanding why a predicate holds or does not hold is sometimes difficult. Figure 8 gives an example of Uno's output, which is designed to address this problem. This particular output is from the analysis of

```
1          UniqPar of <Shimple: ShimpleBody newBody(SootMethod,Map)> parameter 1 : True
2        UniqPar of <ShimpleBody: ShimpleBody(SootMethod,Map)> parameter 1 in <Shimple: ShimpleBody newBody(SootMethod,Map)> : True
3                UniqRet of <NullPointerException: NullPointerException()> : True
4              UniqRet of <Collections$UnmodifiableMap: UnmodifiableMap(Map)> : True
5            UniqRet of <Collections: Map unmodifiableMap(Map)> : True
6          UniqRet of <PhaseOptions: Map getPhaseOptions(HasPhaseOptions)> : True
7        UniqRet of <PhaseOptions: Map getPhaseOptions(String)> : True
8      UniqPar of <ShimpleBody: ShimpleBody(SootMethod,Map)> parameter 1 in <Shimple: ShimpleBody newBody(SootMethod)> : True
9    UniqPar of <ShimpleBody: ShimpleBody(SootMethod,Map)> parameter 1 : True
10 UniqPar of <ShimpleOptions: ShimpleOptions(Map)> parameter 0 in <ShimpleBody: ShimpleBody(SootMethod,Map)> : True
11       UniqRet of <PhaseOptions: Map getPhaseOptions(String)> : True
12     UniqPar of <ShimpleBody: ShimpleBody(Body,Map)> parameter 1 in <Shimple: ShimpleBody newBody(Body)> : True
13       UniqPar of <Shimple: ShimpleBody newBody(Body,Map)> parameter 1 : True
14     UniqPar of <ShimpleBody: ShimpleBody(Body,Map)> parameter 1 in <Shimple: ShimpleBody newBody(Body,Map)> : True
15   UniqPar of <ShimpleBody: ShimpleBody(Body,Map)> parameter 1 : True
16 UniqPar of <ShimpleOptions: ShimpleOptions(Map)> parameter 0 in <ShimpleBody: ShimpleBody(Body,Map)> : True
17 UniqPar of <ShimpleOptions: ShimpleOptions(Map)> parameter 0 : True
18     NEscField−In of <ShimpleOptions: Map options> in <ShimpleOptions: ShimpleOptions(Map)> : True
19     NEscField−In of <ShimpleOptions: Map options> in <ShimpleOptions: boolean enabled()> : True
20     NEscField−In of <ShimpleOptions: Map options> in <ShimpleOptions: boolean node_elim_opt()> : True
21     NEscField−In of <ShimpleOptions: Map options> in <ShimpleOptions: boolean standard_local_names()> : True
22     NEscField−In of <ShimpleOptions: Map options> in <ShimpleOptions: boolean extended()> : True
23     NEscField−In of <ShimpleOptions: Map options> in <ShimpleOptions: boolean debug()> : True
24   NEscField of <ShimpleOptions: Map options> : True
25 NEscPar of <ShimpleOptions: ShimpleOptions(Map)> parameter 0 : True
26 Store of <ShimpleOptions: ShimpleOptions(Map)> parameter 0 : True
27 OwnPar of <ShimpleOptions: ShimpleOptions(Map)> parameter 0 : True
```

Figure 8. Example output of Uno running on Soot, from Figure 3

Name	Byte code	LoC	Cls + Intfs	Methods	Constrs	Fields	Time (s) Soot	Time (s) Uno
spec201-compress	48k	451	12	44	12	53	816	173
spec209-db	16k	512	3	34	3	10	814	174
spec200-check	92k	1,235	17	107	14	42	823	172
spec205-raytrace	120k	1,429	25	176	37	93	817	171
spec202-jess	688k	4,736	151	690	164	265	861	172
spec222-mpegaudio	272k	–	55	322	55	270	857	184
spec228-jack	300k	–	56	315	57	255	851	181
spec213-javac	964k	–	176	1,190	189	851	861	214
DaCapo-antlr	1,004k	–	294	3,170	363	1,279	912	214
DaCapo-luindex	1,005k	–	411	3,218	502	1,564	476	109
DaCapo-lusearch	1,010k	–	413	3,219	504	1,575	488	112
DaCapo-bloat	1,505k	–	426	4,584	451	1,815	499	140
DaCapo-eclipse	1,557k	–	475	4,242	448	2,431	472	128
DaCapo-hsqldb	1,828k	–	556	6,427	649	4,161	1,137	300
DaCapo-jython	2,113k	–	969	9,130	1,140	3,139	1,146	502
DaCapo-xalan	2,170k	–	700	7,089	854	3,339	1,017	312
DaCapo-chart	3,156k	–	795	9,933	1,186	6,534	1,213	502
DaCapo-pmd	3,549k	–	1,392	11,805	1,609	5,894	1,293	518
DaCapo-fop	5,561k	–	2,389	15,073	2,262	8,342	1,484	497
middleware 2.3.1	1,084k	15,360	218	1,070	208	793	661	128
xui 2.0	1,060k	23,542	187	1,946	178	681	876	195
hsqldb 1.8.0	2,176k	51,362	362	4,973	434	3,343	1,061	310
findbugs 1.2.0	12,564k	60,744	2,297	18,034	2,639	7,125	1,395	759
pooka 1.1	13,248k	108,204	2,603	16,358	3,218	7,507	1,683	602
azureus 2.4.0.2	19,240k	126,398	3,763	22,809	3,574	10,665	1,891	1,032
soot 2.2.3	15,856k	196,858	2,958	23,704	3,205	6,525	2,782	2,091
visad 2.0	18,176k	435,227	2,652	29,082	4,152	19,004	3,886	2,117

Figure 9. Benchmark characteristics

the original version of the code in Figure 3, and the predicate in question is whether ShimpleOptions(Map) owns its first argument. For each predicate, Uno displays the non-local conditions that it depends on and their truth values. We use indentation to match up conditions for the same predicate. For example, the conditions on lines 17, 25, and 26 show that OWNPAR holds on line 27. In this case all the predicates that ownership transitively depends on hold, and so line 27 indicates that OWNPAR is true. Our experience suggests this kind of output is critical in understanding Uno's results. For our experiments we disabled this output, because we found printing all this information noticeably slowed the analysis.

5.1 Experiments

We applied Uno to a number of SPEC JVM98 benchmarks, version 1.03_05, to the DaCapo benchmark suite 2006-10-MR2 [5], and to a selection of programs downloaded from SourceForge. Our goal was to determine how often Uno's

	UNIQ RET	LENT PAR	LENT THIS	UNIQ PAR	OWNPAR All	OWNPAR Call	OWNPAR Cons	NESC PAR	NESC FIELD	OWN FIELD	STORE All	STORE Call
spec201-compress	–	35	100	20	0.0	0.0	0.0	80	77	38	69	73
spec209-db	100	92	100	11	0.0	0.0	–	92	0	0	0	0
spec200-check	0	67	99	40	0.0	0.0	–	92	–	–	0	0
spec205-raytrace	0	65	90	11	0.8	0.8	2.6	73	6	6	27	28
spec202-jess	35	71	99	7	0.0	0.0	0.0	78	50	46	12	12
spec222-mpegaudio	0	79	100	0	0.4	0.0	0.0	86	27	24	13	12
spec228-jack	60	69	100	10	5.4	4.3	30.0	79	43	29	15	14
spec213-javac	27	42	81	2	0.0	0.0	0.0	53	15	0	18	18
Average	**32**	**65**	**96**	**13**	**0.8**	**0.6**	**5.4**	**79**	**31**	**20**	**19**	**20**
DaCapo-antlr	22	46	86	11	0.4	0.4	1.4	55	48	22	21	21
DaCapo-luindex	44	34	93	20	2.2	2.0	4.9	49	40	16	34	33
DaCapo-lusearch	44	34	93	20	2.2	2.0	4.8	49	40	16	35	34
DaCapo-bloat	24	44	76	4	0.6	0.6	3.0	53	52	9	18	18
DaCapo-eclipse	29	42	90	13	0.6	0.7	1.8	49	31	15	23	25
DaCapo-hsqldb	34	53	94	12	0.4	0.4	1.6	61	40	12	19	20
DaCapo-jython	33	32	81	7	0.1	0.1	0.5	43	13	5	23	22
DaCapo-xalan	33	43	91	13	0.6	0.3	1.3	51	38	18	24	25
DaCapo-chart	33	50	87	14	0.2	0.2	0.6	55	10	5	21	17
DaCapo-pmd	27	41	87	11	0.3	0.2	1.2	51	35	16	19	18
DaCapo-fop	37	37	90	12	0.6	0.4	1.1	45	33	10	27	27
Average	**33**	**41**	**88**	**12**	**0.7**	**0.7**	**2.0**	**51**	**35**	**13**	**24**	**24**
middleware 2.3.1	27	73	92	27	0.2	0.3	1.2	81	70	28	19	18
xui 2.0	22	39	92	14	0.6	0.4	2.8	46	41	17	17	16
hsqldb 1.8.0	36	54	95	12	0.1	0.1	0.6	61	33	12	18	18
findbugs 1.2.0	29	46	90	7	1.3	0.4	1.9	55	32	15	20	18
pooka 1.1	31	43	91	14	1.1	0.6	1.6	55	40	13	31	30
azureus 2.4.0.2	23	40	88	14	1.1	1.1	2.8	56	50	9	34	35
soot 2.2.3	31	46	89	6	0.3	0.2	0.9	53	33	11	15	15
visad 2.0	24	59	82	8	0.3	0.2	0.8	66	35	18	19	18
Average	**28**	**50**	**90**	**13**	**0.6**	**0.4**	**1.6**	**59**	**42**	**15**	**22**	**21**
Average of all	**31**	**51**	**91**	**13**	**0.7**	**0.6**	**2.7**	**62**	**36**	**16**	**22**	**22**

Figure 10. Predicate inference results (all numbers are percentages)

aliasing and encapsulating predicates held across a wide variety of Java programs. Currently Uno requires a significant amount of memory, both because it is an early prototype and because it uses Soot, which is memory and time intensive. Accordingly, we ran Uno on a multiprocessor (but Uno is only single-threaded) UltraSparc III 750Mhz machine with 72GB of memory. The maximum Java heap size was set to 15GB, although all except the largest benchmarks required less memory. We believe that by switching front-ends and with more engineering effort, Uno could run comfortably on commodity hardware.

Figure 9 summarizes the characteristics of our benchmark suite and gives the running times for Uno. For each benchmark, we list the size of the bytecode in kilobytes, the number of non-comment, non-blank lines of code, and the number of classes and interfaces. In these and all other counts, we include only code from the benchmarks, although Uno also analyses portions of the Java library. The DaCapo suite and three of the SPEC benchmarks, mpegaudio, jack and javac, include only bytecode but no source. Additionally, the DaCapo benchmarks come in a single large jar file; we identified the class files belonging to each benchmark by name, and also included any other classes in the jar file that are reachable from them based on the compile-time types. This resulted in a different number of classes and a different bytecode size for the version (1.8.0.4) of hsqldb in the DaCapo suite versus the version we downloaded from SourceForge.

Figure 9 also lists the number of methods, constructors, and fields for each benchmark. The right two columns of the figure give the running time for Uno, as the average of five runs. We divide the running time into two parts. The Soot time includes class file loading and conversion into Jimple. The Uno time includes everything else, including computing the class hierarchy (we did not use Soot's hierarchy), performing the points-to analysis, and resolving the predicates. As these results show, Soot consumes a large fraction of the running time.

Figure 10 gives the results of running Uno on the benchmarks. For each predicate, we report the percentage of positions for which it holds. The exact metric depends on the predicate, as we discuss below. Blank entries indicate there are no positions on which to check the predicate. We omit the -IN versions of the predicates, since those are included in their non-IN counterparts. For LENTPAR and UNIQPAR, we evaluated the predicates with *no-fld* set to true. We also compute the average percentage for each predicate, over each group of benchmarks and across the entire benchmark suite. Next we discuss each of the predicates in turn.

UNIQRET – We report what percentage of methods (not constructors) return unique objects. We do not include methods that return primitives, void, or String in our count, either in the numerator or the denominator. (Strings are immutable, so their uniqueness is most likely uninteresting.) These results show that on average, Uno finds uniqueness of a return value for 31% of the methods, which suggests this

property is fairly common. We examined a selection of the results manually, and found that many of these cases were effectively factories—i.e., methods that, at the end, created a new object, initialized it, and returned it. We found that all constructors in our benchmarks returned unique results, although Uno reported that a small number (0.03% of called constructors) did not due to conservatism in its analysis.

LENTPAR – We measured the number of non-primitive parameters that Uno determines are lent, for both methods and constructors. We found that 51% of all parameters are lent on average across all of our benchmarks. This result suggests that lending parameters is common, which makes sense—if methods typically retained pointers to their arguments, it would complicate local reasoning. In this case, we suspect that LENTPAR is even more common, and that conservatism in our analysis may be causing us to underestimate parameter lending.

LENTTHIS – We counted the number of non-static methods (not constructors) whose receiver object is lent. Uno infers that this holds on average 91% of the time, and 96% of the time for the SPEC benchmarks. This result is to be expected, since it is uncommon for Java methods to capture this, although it does happen in some cases. Examples are code with callbacks, e.g., x.addActionListener(this), or when (non-static) inner class instances are created, since they may refer to the outer class object.

UNIQPAR – For this predicate, we report the number of non-primitive, unique parameters of methods and constructors. We only include methods and constructors that are potentially called in the program, according to the compile-time types, since otherwise UNIQPAR is trivially true. We found that only 13% of parameters are unique. Thus Uno discovers relatively few possible handoffs of an object from one method to another (one reason UNIQPAR may hold).

OWNPAR – We counted the number of non-primitive parameters of non-static methods and constructors that are owned. The first column lists the fraction for all methods and constructors; the second column lists the fraction for only those that are called; and the last column lists the fraction for only constructors that are called. Our results show that argument ownership—in the strict, monomorphic sense defined in Uno—is a fairly rare property, holding for only 2.7% of the constructors called in a program, on average. In Section 5.2, we show some examples of argument ownership that we found, and we examine the causes of why ownership does not hold.

NESCPAR – We found that 62% of method and constructor parameters do not escape the callee, on average across all benchmarks. NESCPAR is very similar to LENTPAR, except it is slightly less restrictive—in NESCPAR, a parameter may be written to a field as long as that field does not escape. Thus the difference between the two, roughly 11% on average for all benchmarks, shows how often an argument that is passed to a method may be captured in a non-escaping field.

```
1  public class EventWidget extends ... {
2      private AmandaFile fileData;
3      public EventWidget(AmandaFile fileData, ...) {
4          this.fileData = fileData;
5          slider = buildSlider(fileData.getNumberOfEvents());
6      }
7  }
8  public class NuView extends ... {
9      private static final
10     JPanel buildMainDisplay(..., AmandaFile file ,...) {
11         setRange(xMap,file.getXMin(),file.getXMax(),halfRange);
12         EventWidget eventWidget = new EventWidget(file,...);
13         return panel;
14     }
15     public NuView(String[] args) {
16         AmandaFile file = openFile(fileName);
17         JPanel widgetPanel = buildMainDisplay(..., file ,...);
18     }
19     private static final AmandaFile openFile(String fileName) {
20         AmandaFile file;
21         if (fileName.startsWith("http :// ")) {
22             file = new AmandaFile(new URL(fileName));
23         } else {
24             file = new AmandaFile(fileName);
25         }
26         return file ;
27     }
28 }
```

Figure 11. Argument and field ownership in visad

NESCFIELD – For this predicate, we restricted our measurement to private fields, since the predicate trivially does not hold for non-private fields. We found that about 36% of private fields do not escape. This is another reason argument ownership is rare—many objects stored in private fields are considered shared by Uno, rather than encapsulated.

OWNFIELD – We found that 16% of private fields across our benchmark suite are considered owned by Uno, with a slightly higher percentage in the SPEC benchmarks. Recall that OWNFIELD is a more restrictive version of NESCFIELD, since it further requires that objects stored in fields be unique or only aliased to other owned fields. Our results show that field ownership, while not that common, occurs significantly more often than argument ownership. We discuss some examples of field ownership below in Section 5.2.

STORE – Finally, we measured the number of non-static methods and constructors that store their non-primitive, non-String arguments. We count both all such methods and constructors and only those that are called. Only around a fifth of called methods store their arguments.

5.2 Examples of Ownership

We examined a selection of the ownership results manually to confirm them and to understand the ownership patterns that Uno discovers.

Figure 11 shows a typical example of argument ownership and field ownership, in which an object is generated at a single unique return and then reaches a constructor in a few

```
1  class SynchronizedList extends SynchronizedCollection ... {
2      List list ;
3      SynchronizedList(List list ) {
4          super(list );
5          this. list = list ;
6      }
7  }
8  class SynchronizedCollection implements ... {
9      Collection c;
10     SynchronizedCollection(Collection c) {
11         this.c = c;
12     }
13 }
```

Figure 12. Potential ownership in java.util.Collections

```
1  public class PEMReader extends ...{
2      private PasswordFinder pFinder;
3
4      public PEMReader(..., PasswordFinder pFinder) {
5          this (..., pFinder ,...);
6      }
7      public PEMReader(..., PasswordFinder pFinder, ...) {
8          this.pFinder = pFinder;
9      }
10 }
11 public class ReaderTest extends ...{
12     public void performTest() throws Exception{
13         PasswordFinder pGet =
14             new Password("secret".toCharArray());
15         PEMReader pemRd = new PEMReader(..., pGet);
16     }
17 }
```

Figure 13. Argument and field ownership in pooka

steps. More precisely, in this code (simplified and with some fields and methods not shown), Uno reports that openFile on lines 19–27 returns a unique object. This method is called on line 16, and the result is passed as argument file on line 17 to buildMainDisplay. This method in turn invokes some methods of file that do not capture it (line 11, callee not shown). Then file is passed to the EventWidget constructor, defined on lines 3–5, which stores it in a private field fileData. Since that field is not leaked by the call on line 5 (code not shown), Uno reports that the EventWidget constructor owns its argument, and that fileData is an owned field.

In Section 2, we saw a similar example from Soot, in which the path from unique return to owning constructor was slightly more convoluted but had the same basic pattern. Most of the other cases of ownership that we looked at also fell into this pattern.

Another ownership pattern that Uno sometimes finds involves super or this constructors. Recall from Section 4 that certain predicates use the *no-fld* flag to treat super and this constructor calls specially. We added this flag after running Uno on some sample classes from the Java standard library. We had expected Uno to infer that the SynchronizedList constructor, sketched in Figure 12, owns the list it synchronizes

```
1  public class ASTSynchronizedBlockNode ... {
2      private ValueBox localBox;
3      public void setLocal(Local local){
4          this.localBox = Jimple.v().newLocalBox(local);
5      }
6      public Local getLocal() {
7          return (Local) localBox.getValue();
8      }
9  }
```

Figure 14. Field ownership in Soot

```
1  public final class IPAddressGatekeeper ... {
2      private File m_databaseFile;
3      private IPAddressGatekeeper (String filename) {
4          m_databaseFile = new File(filename);
5      }
6      private void loadDatabase () {
7          long modificationTime = m_databaseFile.lastModified();
8          BufferedReader reader;
9          try {
10             reader = new BufferedReader(
11                 new FileReader(m_databaseFile));
12         } catch (IOException e) { ... }
13     }
14 }
```

Figure 15. Field ownership in Middleware

access to. However, without the *no-fld* flag, this turned out to be false, because both the class SynchronizedList and its superclass, SynchronizedCollection, keep a pointer to the list. Adding the *no-fld* flag fixed this problem, allowing both to point to the list and maintain ownership. Ultimately, however, Uno still does not infer ownership for this example, because both classes store the Collection in a non-private field, causing it to leak.

Figure 13 shows a example of argument ownership that Uno does find that involves a call to a this constructor. In this code, class PEMReader has two constructors, both with a common PasswordFinder argument. One of them simply calls the other one through a this() call (line 5). The other constructor (lines 7–9) stores the PasswordFinder argument into a field. Uno infers that both constructors own their PasswordFinder argument if that argument is unique, which holds because fresh objects are passed in on lines 13 and 15. Uno also infers that pFinder is an owned field.

In general, we found this argument ownership pattern was uncommon in the cases we examined, which mostly matched the example in Figure 11.

We also examined some cases where field ownership but not argument ownership held. Figure 14 shows an example from Soot. On line 4, the field localBox is set to the result of calling newLocalBox, which returns a unique result. Moreover, the call to getValue on line 7 does not change the uniqueness of localBox, and thus that field is owned.

Figure 15 shows a slightly more complex example from Middleware. Here on line 4, the field m_databaseFile is initialized to a fresh object. That field has one of its methods invoked on line 7, and is passed to a constructor on line 11, but neither of those calls affects its uniqueness, and thus m_databaseFile is owned.

Many of the field ownership examples we found in our benchmark suite were similar to these examples, where a field is initialized to a fresh object locally within a class, and then does not escape via method calls of returns. Recall that argument ownership requires that arguments are unique, which is relatively uncommon. We believe this is why field ownership is significantly more common than argument ownership in our benchmarks—local uniqueness of fresh objects is much more likely than a unique argument.

5.3 When Ownership Does Not Hold

Finally, we also investigated the reasons that ownership does not hold in our benchmarks, to try to understand why that property is fairly rare. For each predicate, we computed how often its local and non-local conditions were false, thereby causing the predicate to be false. Figure 16 gives the average percentages for several predicates across our benchmarks. For example, condition 2 of OWNPAR, which is UNIQPAR(m, i, false), did not hold 79% of the time, as shown in the upper-left corner of the figure. Note that to avoid biasing our results by the order of the conditions, we computed the truth value of all conditions of a predicate, rather than stopping at the first one that was false. Thus the columns in Figure 16 can add up to more than 100%, since several conditions may be false simultaneously.

Looking at the results, we see that OWNPAR is most often false because of conditions 2 and 4—the method or constructor does not receive a unique argument, or the method does not store the argument in one of its fields. Clearly not storing an argument is a valid reason for not owning it. Note that our STORE predicate is heuristic, because it only checks storing directly via a field or via a super or this call, and it could miss writes to fields via other means, e.g., calling a setter method.

The other condition, uniqueness of the argument, is more complex. UNIQPAR(m, i, false) only depends on itself (condition 1) and UNIQPAR-IN for regular (condition 2, called with *no-fld* set to *true*) and super or this calls (condition 3, called with *no-fld* set to *false*). Following condition (2) further, we see that calls to UNIQPAR-IN$(\ldots, \mathit{true})$ fail for a variety of reasons. Most often they fail because of local condition L3, i.e., the argument in question is live after the call. Local conditions L1 and L4 (the value is BAD or stored in a field) also contribute somewhat to non-uniqueness of arguments. The last major reason UNIQPAR-IN fails to hold is because the argument is either a non-unique parameter from the caller (condition 1) or a non-unique return value from another call inside this method (condition 2).

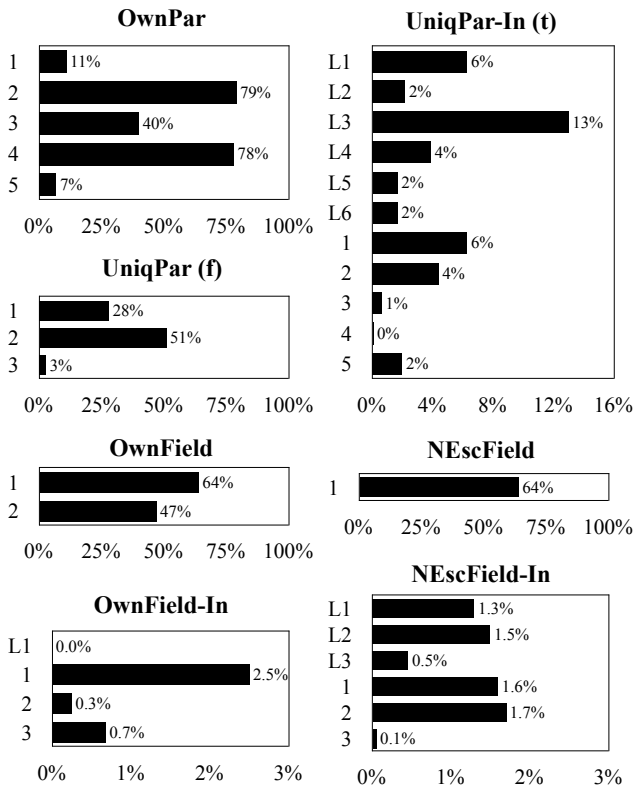

Figure 16. Causes of false predicates

Looking at our other ownership predicate, OWNFIELD, we see that it fails most often because NESCFIELD (condition 1) is false, and almost as often because of OWNFIELD-IN (condition 2). For the latter, we can see that condition 1, having a non-unique parameter stored in a field, is the most common cause the predicate fails to hold. Thus again, uniqueness is a key culprit in lack of ownership. Note that in computing the percentages for OWNFIELD-IN, we measure over all fields and all methods, including methods that do not use a particular field. This lowers the percentages by a constant factor compared to the other predicates, but does not change the relative magnitudes within the histogram.

Lastly, examining NESCFIELD, we ignore condition L1, that the field is private, in our measurements. Thus we concentrate on condition 1, NESCFIELD-IN. We can see that the latter predicate fails for a wide variety of reasons, all roughly equally common. Indeed, the only uncommon conditions to fail are condition L3, a field potentially being accessed from another instance of the same class, and condition 3, parameter capture by a this or super call. Note we have the same measurement issue with NESCFIELD-IN as OWNFIELD-IN, making the percentages low but not changing the relative magnitudes in the histogram.

It is important to recognize that these results only approximate the reasons ownership fails—for example, we measure when UNIQPAR-IN is false over all combinations of

the predicate, not just the ones that cause OWNPAR to be false. However, they do provide some insight. Almost 80% of non-primitive method and constructor arguments are not stored according to Uno, and so clearly they are not owned. One future direction would be implementing a more complete must-store analysis, to find out how many methods truly store their arguments. However, it makes sense that storing arguments is not a property that holds for most parameters, since that would complicate local reasoning. Storing an argument should happen most often for constructors and methods that create fresh objects.

Uniqueness is the other major culprit in preventing ownership, and one way to increase the opportunities for ownership may be to improve the precision of the uniqueness analysis. Another approach may be to relax the uniqueness requirement, e.g., by allowing polymorphic ownership so that an owned object can be pointed to by a field of more than one object. We ran an additional experiment in which we removed the UNIQPAR requirement (condition 2) from OWNPAR, and we found this resulted in nearly 20% of the parameters of called constructors being "owned," as opposed to the 2.7% result with uniqueness. This is a coarse upper bound on how improved uniqueness analysis might improve inference of owned arguments, and we leave pursuing this further as an interesting direction for future work.

5.4 Threats to Validity

There are a number of potential threats to the validity of our evaluation. First, our selection of benchmarks could be unrepresentative, and might be missing certain coding styles that would cause Uno to behave differently. We have tried to address this by applying Uno to a wide variety of applications, include several that are large enough that they should contain many interesting coding patterns.

Second, our algorithm for inferring uniqueness and ownership may not be sound. We have high confidence that the points-to analysis is correct, because that problem has been well-studied, and our analysis is similar to proven approaches. It is harder to be certain that the specification of our uniqueness and ownership predicates is correct and that there are zero implementation bugs. We have tried to address this both by testing Uno on our own small examples and by verifying a selection of Uno's ownership and uniqueness results on our benchmark suite manually. We leave a formal proof of correctness as future work.

Lastly, and perhaps most importantly, our definitions of uniqueness and ownership might not match a programmer's. Hence we might either report ownership and uniqueness that is not interesting, or we might fail to find useful instances of ownership and uniqueness. Indeed, our experiments show that OWNFIELD sometimes holds, and OWNPAR rarely holds. We believe that using a more sophisticated uniqueness analysis may increase the amount of ownership we discover in programs, and we leave that challenge to future work.

6. Related Work

Many researchers have studied encapsulation in object-oriented languages. Islands [18] and Balloons [3] allow objects to be fully encapsulated. Clarke et al [11, 12] and Boyapati et al [6] propose ownership type systems that statically enforce encapsulation. These systems allow rich ownership patterns, including owner parameterization and weakening ownership for inner classes, neither of which is supported by Uno. The key difference between these systems and Uno is that Uno performs inference and can therefore be applied to existing Java programs, whereas the other systems require often extensive user annotations.

AliasJava [2] is a dialect of Java that includes annotations for uniqueness, ownership, and allows classes to be parameterized by owners. AliasJava's annotations are statically checked, and Aldrich et al report on case studies in which annotations were successfully added by hand to Java programs [2]. AliasJava also includes polymorphic annotation inference, but the only reported results are for toy programs and for the Java standard library; and the library results are not reported very precisely. We show that Uno can infer uniqueness and ownership on a wide variety of Java programs, and we report summary statistics of uniqueness and ownership properties, which complements case studies.

Liu and Milanova [22] present static analyses for inferring ownership and immutability in Java. Their ownership inference algorithm constructs a may points-to graph and then uses that to approximate the run-time dominator relationship among objects. Liu and Milanova find that 28% of the reference-valued instance fields are owned across their benchmark suite. As discussed earlier, Uno finds that roughly 16% of private, reference-valued fields are owned. The cause of the difference between the two results is unclear. It may be due to variations in the definition of ownership or due to measurement techniques. One key difference between the Liu and Milanova approach and Uno is that they infer only field ownership, whereas Uno infers many other related properties, including argument ownership, uniqueness, and lending, which their system cannot reason about.

Cherem and Rugina [9] present a lightweight escape and effects analysis for Java. Among other things, their analysis computes which parameters are lent to methods (69% on average), similar to LENTPAR, and what methods return fresh objects (43% on average), similar to UNIQRET. Both percentages were measured on the GNU Classpath library. These numbers are not that close to Uno's averages of 51% and 31%, respectively, but it is unclear whether this is a significant difference, since the GNU library's aliasing behavior might be different than aliasing in other programs.

In follow-up work, Cherem and Rugina develop a field uniqueness analysis for object reclamation in Java. They find that approximately 22% of fields in the SPEC benchmark suite are unique [10]. This corresponds closely to our result that 20% of fields in the SPEC benchmark suite are owned.

These two properties seem to be closely related, suggesting that notions of ownership may be useful in a wide variety of applications in addition to program understanding.

Confined types [30] operate at a coarser granularity than the systems discussed so far. In this approach, static checking ensures that confined types are not exposed outside of their packages. Grothoff et al [16] present an inference algorithm for confined types, and show that over a wide range of benchmarks, many classes can be marked as confined. This system is significantly more scalable than Uno, but also provides much coarser information about encapsulation.

Heine and Lam [17] present Clouseau, a static analysis for finding memory that leaks in C and C++ programs. Clouseau reports a leak when the "owning" pointer of an object is discarded before the object is freed. In this system, an owner has the responsibility to free an object, but does not necessarily have the only pointer to it. In contrast, in Uno, owned objects must be encapsulated inside of their owner.

Uniqueness also has a long research history. Uniqueness is closely related to linear types, which can be used to reason precisely about aliasing in programs [28, 31]. Several researchers have studied making uniqueness and linear types easier to use at the language level [1, 7, 8, 15]. These systems allow a more flexible interpretation of uniqueness than Uno, but mostly focus on checking annotations, and inference is not available for object-oriented languages.

Finally, alias analysis has been extensively studied in the research literature [13, 14, 19, 20, 21, 25, 26, 27, 32] (to name only a few). Our alias analysis is similar to existing systems, but is tuned for inferring Uno's predicates.

7. Conclusion

We have presented a new technique for automatically inferring aliasing and encapsulation in Java programs. Our analysis begins with an intraprocedural points-to analysis that tracks local variables flow-sensitively, and summarizes fields, external data, and method calls flow-insensitively. We then perform a demand-driven, interprocedural algorithm to resolve predicates that describe aliasing and encapsulation of method and constructor arguments and return values, and fields. To test our ideas, we developed a tool called Uno that implements our analysis and applied it to a number of Java applications. Uno discovered many lent method arguments, a moderate number of unique arguments and results, some field ownership, and occasional argument ownership. We believe that Uno is the first ownership and uniqueness inference tool that has been demonstrated on a wide variety of Java applications.

Acknowledgments

We would like to thank Nicholas Chen, Bin Zhao, and Taiga Nakamura for working on predecessors to Uno. We would also like to thank Mike Hicks and the anonymous reviewers for helpful comments on an earlier version of this paper. This research was supported in part by NSF CCF-0346982 and CCF-0430118.

References

[1] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *PLDI'03*, pages 129–140, 2003.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA'02*, pages 311–330, 2002.

[3] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In *ECOOP'97*, pages 32–59, 1997.

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[5] S. M. Blackburn, R. Garner, C. Hoffman, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, 2006.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL'03*, pages 213–223, 2003.

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

[8] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *POPL'05*, pages 283–295, 2005.

[9] S. Cherem and R. Rugina. A Practical Escape and Effect Analysis for Building Lightweight Method Summaries. In *CC'07*, 2007.

[10] S. Cherem and R. Rugina. Uniqueness inference for compile-time object deallocation. In *ISMM'07*, 2007. To appear.

[11] D. G. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA'02*, pages 292–310, 2002.

[12] D. G. Clarke, J. M. Potter, and J. Noble. Onwership Types for Flexible Alias Protection. In *OOPSLA'98*, pages 48–64, 1998.

[13] M. Das. Unification-based Pointer Analysis with Directional Assignments. In *PLDI'00*, pages 35–46, 2000.

[14] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI'94*, pages 242–256, 1994.

[15] M. Fähndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI'02*, pages 13–24, 2002.

[16] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating Objects with Confined Types. In *OOPSLA'01*, pages 241–253, 2001.

[17] D. L. Heine and M. S. Lam. Static Detection of Leaks in Polymorphic Containers. In *ICSE'06*, pages 252–261, 2006.

[18] J. Hogg. Islands: Aliasing Protection In Object-Oriented Languages. In *OOPSLA'91*, pages 271–285, 1991.

[19] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *PLDI'92*, pages 235–248, 1992.

[20] O. Lhoták and L. J. Hendren. Jedd: A BDD-based relational extension of Java. In *PLDI'04*, pages 158–169, 2004.

[21] D. Liang and M. J. Harrold. Efficient Computation of Parametrized Pointer Information for Interprocedural Analyses. In *SAS'01*, pages 279–298, 2001.

[22] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE'07*, pages 323–332, 2007.

[23] V. B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *APLAS'05*, pages 139–160, 2005.

[24] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *ECOOP'98*, pages 158–185, 1998.

[25] A. Rountev and B. G. Ryder. Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries. In *CC'01*, pages 20–36, 2001.

[26] A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP'01*, pages 12–23, 2001.

[27] M. Sridharan and R. Bodik. Refinement-Based Context-Sensitive Points-To Analysis for Java. In *PLDI'06*, pages 387–400, 2006.

[28] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *FPCA'95*, pages 1–11, La Jolla, California, 1995.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON'99*, 1999.

[30] J. Vitek and B. Bokowski. Confined types in java. *Software—Practice and Experience*, 31(6):507–532, 2000.

[31] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *TIC'00*, 2000.

[32] J. Whaley and M. S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI'04*, pages 131–144, 2004.