OwnKit: Inferring Modularly Checkable Ownership Annotations for Java

Constantine Dymnikov
School of Engineering and Computer Science
Victoria University of Wellington
Email: cdymnikov@gmail.com

David J. Pearce
School of Engineering and Computer Science
Victoria University of Wellington
Email: djp@ecs.vuw.ac.nz

Alex Potanin
School of Engineering and Computer Science
Victoria University of Wellington
Email: alex@ecs.vuw.ac.nz

***Abstract*—Ownership and related systems impose restrictions on the object graph that can help improve program structure, exploit concurrency and verify software. Such systems rely on the presence of appropriate ownership annotations in the source code. Unfortunately, manually adding ownership annotations to legacy systems is a tedious process. Previous attempts at automatically inferring such ownership systems do not produce modularly checkable annotations (i.e. which allow classes to be checked in isolation) making them difficult to incorporate into day-to-day development. In this paper, we present OwnKit — a system for automatically inferring ownership annotations which are modularly checkable. We describe and evaluate our approach on a number of real-world benchmarks and compare against an existing system.**

I. Introduction

Modern object-oriented languages permit methods to modify the heap at will. Whilst this provides flexibility of implementation, it presents a number of challenges for both programmers and automated tools [1], [2]. This is because reasoning about such programs is difficult without a strong understanding of where and when aliasing may occur.

An important way of managing this problem is through the use of *ownership types* (e.g. [3], [4], [5], [6], [7], [8], [9]). These provide strong guarantees about when and where aliasing is permitted between objects. Such systems operate by annotating (in some manner) types of the underlying language. For example:

```
public class Rectangle {
  private @Owned List<Point> points;
  ...
}
```

In a *deep ownership* system (e.g. [4], [5], [6], [8]), this annotation might indicate that all objects reachable from `points` are owned by the `Rectangle` instance (i.e. the *owner*). Only the owner of `points` may hold a reference to `points`, leading to a strong encapsulation property.

Aside from managing software complexity, ownership types also have concrete applications in areas such as: parallel and concurrent systems [6], [10], [11], specification languages [12], [13], real-time systems [14], [15], [16], and more.

A. Inferring and Checking Ownership Annotations

Whilst ownership type systems can help developers manage complexity, they also impose a heavy burden: *ownership annotations must be carefully added to programs*. This makes programs more verbose and harder to understand which, in turn, reduces maintainability [17]. The need to retroactively add ownership annotations to large legacy systems may also prohibit wide-spread use [18]. One way of ensuring an ownership system fits within day-to-day development is to divide it into: an *annotation inferer* and an *annotation checker*. The annotation inference operates as a source-to-source translation, taking in existing Java code and adding `@Owned` annotations where appropriate. The annotation checker can then efficiently check these annotations are correct at compile-time. The idea behind this is simple: users take their existing applications, infer the `@Owned` annotations once using the (potentially expensive) annotation inference; then, they *maintain* them using the (efficient) annotation checker.

For this approach to be practical, the annotation checker must be efficient. A sensible way of ensuring this is to require that the annotations be *modularly checkable* [19]. That is, the annotations in one class can be checked in isolation from others and, hence, a whole-program analysis can be avoided. Notice that, whilst the *checker* must be efficient, the *inferer* need not be since it is only operated once on the legacy system. Nevertheless, care must still be taken to ensure inferred annotations are indeed modularly checkable.

Several existing techniques are known for inferring ownership types in OO languages (e.g. [7], [20], [21], [22], [23], [24], [25]). The majority employ interprocedural pointer analysis as the underlying algorithm. While this yields precise results, it has an important drawback: *the produced annotations are not (by themselves) modularly checkable*. That is, checking the annotations are correct requires access to the pointer analysis results. Therefore, we must either: require the annotation checker to perform the interprocedural pointer analysis itself (which clearly goes against the goal of making it efficient); or, embed the pointer analysis results in the source code using annotations. Whilst the latter may appear to be reasonable (at first glance), in practice it is not. This is because points-to sets may contains tens or hundreds of pointer targets, each of which may refer to any class in the whole program (see e.g. [26], [27]). Thus, the annotations themselves would be large and unwieldy, as well as being extremely brittle to unrelated program changes. In contrast, our approach does not rely on interprocedural pointer analysis and produces straightforward `@Owned` annotations which are modularly checkable.

B. Contributions

We present an ownership system that is split into two components: an *ownership inference* (OwnKit — described in this paper) and an *ownership checker* (OIGJ [9]). The ownership inference operates as a source-to-source translation, taking in existing Java code and adding modularly checkable `@Owned` annotations.

The contributions of this paper are as follows:

1) We present a novel ownership inference system which comprises of an *ownership inference* and an *ownership checker*. The former is carefully designed to generate modularly checkable annotations and operates as a source-source translation for annotating legacy code.

2) We report on experiments using our system on a selection of Java programs and compare our results with the closest existing ownership inference tool that is available online [23].

Finally, a more detailed presentation of this work which includes a formalisation of the ownership inference system, and additional technical discussion is available [28].

II. Overview

We start by presenting the underlying notion of ownership types used in our system, and provide an informal argument that they are modularly checkable.

In our system, non-primitive fields may be annotated with the `@Owned` annotation. The following characterises the meaning of this modifier within our system:

Definition 1 (Object Graph): An *object graph*, $\mathtt{O_G}$, is a directed graph capturing a snapshot of the heap at a given moment. Here, $\mathtt{o_1} \xrightarrow{\mathtt{C.f}} \mathtt{o_2} \in \mathtt{O_G}$ denotes that object $\mathtt{o_1}$ refers to object $\mathtt{o_2}$ via the field `f` declared in class `C`.

Definition 2 (Ownership Guarantee): Let `C.f` be a non-primitive field annotated with `@Owned` which is declared in class `C`. Then, for all objects $\mathtt{o_1}, \mathtt{o_2}, \mathtt{o_3}$ where $\mathtt{o_1} \xrightarrow{\mathtt{C.f}} \mathtt{o_3} \in \mathtt{O_G}$ and $\mathtt{o_2} \xrightarrow{\mathtt{C.f}} \mathtt{o_3} \in \mathtt{O_G}$ it follows that $\mathtt{o_1} = \mathtt{o_2}$.

To enforce the ownership guarantee in a modularly checkable fashion, we must type-check annotations for a given class in isolation from others.

A. Objective

The aim of our system is to conservatively determine which fields may be safely annotated with `@Owned` under Definition 2. For simplicity, we do not permit annotations in other places (e.g. parameters, returns or local variables).

In order to infer that a field can be safely annotated with `@Owned`, we must determine that it cannot be *exposed*. A non-primitive variable is said to become *exposed* when external objects gain access to objects it references. Rather than perform an expensive interprocedural flow-analysis, we make several conservative assumptions:

- Parameters and return values for **`public`** or **`protected`** methods are automatically exposed.
- Fields declared **`public`** or **`protected`** are automatically exposed.

These assumptions ensure that our annotations are *modularly checkable*. In other words, source files can be checked in isolation from each other, making it feasible to implement our system within the context of a Java Compiler.

The key challenge in our system lies in determining how values flow within a given class. For example, a field `f` may be declared **`private`** but may potentially flow into the return value of a **`public`** method — in which case `f` is exposed and cannot be annotated with `@Owned`.

B. Variable Exposure

A variable is said to be *read exposed* if its value may be read by external objects. This occurs when the object that is initially referred to by the variable can later become aliased by any of the variables in external objects. Consider the following:

```
public class MyClass {
  private List<String> myList = ...;

  public List<String> getMyList(){
    return myList;
  }
}

public class External {
  public void expose(){
    MyClass mc = ...;
    List<String> alias = mc.getMyList();
    alias.add("bad");
  }
}
```

Here, an external object can potentially obtain the reference to field `myList`, create an alias and proceed to modify elements of the list. Thus, `myList` is read exposed and cannot be annotated with `@Owned`. Note, however, that it is not possible for external code to update `myList` to refer to another object.

A variable is said to be *write exposed* if it may be assigned a value which can be read by external objects. Consider:

```
public class MyClass {
  private List<String> myList = ...;

  public void setMyList(List<String> par){
    myList = par;
} }

public class External {
  public void expose() {
    MyClass mc = ...;
    List<String> alias = ...;
    mc.setMyList(alias);
} }
```

As before, the external object is able to create an alias to field `myList`. This time, however, the alias is created by changing the object that `myList` refers to. Thus, `myList` is write exposed and cannot be annotated with `@Owned`. Note that an external object cannot indirectly read field `myList` and, hence, `myList` itself is not read exposed. This distinction is important because values which flow into `myList` from elsewhere in the class are not exposed by this.

Finally, a variable may have both types of exposure, as illustrated below:

```
public class MyClass {
    public List field = ...;
}
```

Here, the (public) field `field` is both read and write exposed because we must conservatively assume external objects may read from it and/or write to it.

III. Implementation

We now discuss the implementation of our system in more detail. The inference works by constructing a directed graph, called the *class graph*, capturing value flow within a class. Nodes in the graph are coloured according to their exposure, and a procedure (similar to transitive closure) is applied to propagate exposure through the graph.

A. Class-Graph Construction

A class graph $G = (V, E, L)$ is constructed for each class where each field, method parameter, and local variable it contains corresponds uniquely to some vertex $v \in V$. The current exposure for each vertex is maintained using a special map, L, which maps vertices to one of $\{\epsilon, \mathtt{R}, \mathtt{W}, \mathtt{RW}\}$. Here, ϵ indicates no exposure, `R` indicates read exposure, `W` indicates write exposure, and `RW` indicates read-write exposure.

An edge, $v \rightarrow w \in E$, is used to signal a potential value flow from the variable represented by v to that represented by w. The following illustrates a simple class and those edges that would be generated for it:

```
public class MyClass {
  private Object field;

  public Object fun(Object p, MyClass q) {
                                  // fun$p ← $W,fun$q ← $W
    Object tmp = field;    // fun$tmp ← field
    if(p!=null) {
      this.field = p;        // field ← fun$p
    } else {
      this.field = q.field;  // field ← $W
    }
    return tmp;              // fun$ ← fun$tmp
                             // $R ← fun$
} }
```

Here, we can see that variables local to a given method are mangled to avoid potential clashes with same-named variables in other methods. The special variable *fun$* represents the return value of method `fun`. Likewise, the special variable $R (resp. $W) captures all external read (resp. write) accesses and is considered read (resp. write) exposed. Thus, $W is connected to all parameters on **`public`** or **`protected`** methods. Likewise, $R is connected from all return values on **`public`** or **`protected`** methods.

Field accesses on **`this`** are resolved to vertices in the graph, whilst all other field accesses are resolved to either $W or $R (depending on whether they are read from, or assigned to). We ignore assignments through arrays for the moment, and return to them later. Method invocations are treated in a similar way to field accesses. In the case of a method call on **`this`**, the arguments are connected to the corresponding parameters and (if applicable) the invoked method's return is connected to the assigned variable. For all other invocations, the arguments are connected to $R and the returned value taken from $W. The following illustrates the main cases:

```
public class MyClass {
  private Object field;

  public Object f(Object p, MyClass q) {
                              // f$p ← $W,f$q ← $W
    Object t = this.g(p);  // g$x ← f$p,f$t ← g$
    this.g(p).toString();  // g$x ← f$p
    q.field = this.g(p);// g$x ← f$p, $R ← g$
    return q.g(p);           // $R ← f$p,f$ ← $W
                             // $R ← f$
  }

  private Object g(Object x) {
    return x;                //  g$ ← g$x
  }
}
```

Essentially, we are conservatively capturing the flow of information through the variables (i.e. fields, parameters, etc) of the class. In the case of invocations to private methods on **`this`** we include interprocedural flow. However, in cases where the method being invoked is on an arbitrary variable we default to conservatively assuming arguments are read-exposed, and return values are write-exposed.

Finally, in many cases it is possible to optimise the graph by replacing $u \rightarrow v \rightarrow w$ with $u \rightarrow w$, provided that v does not represent a field. Our system does this to reduce graph complexity when it is safe to do so.

B. Class-Graph Propagation

Once the class graph is constructed, we must propagate all exposure information throughout the graph to determine which fields are exposed. Figure 1 shows the rules for inferring the exposure of some node `a`. There are 32 cases in total: 4 possible exposures for each node (no exposure, read exposure, write exposure, read and write exposure) and two possible directions of flow. The rules only specify the exposure of node `a`; if we want to find the exposure of node `b` we would have to swap the order of the nodes involved.

In many cases, the exposure of a given node does not change. In particular, we can see that the only changes to the exposure of node `a` are the additions of new exposure types and never the removal of existing exposure types (note, this

	$[a] \dashrightarrow [b]$				$[a] \dashleftarrow [b]$			
	$[b]$	$[b]^{W}$	$[b]_{R}$	$[b]^{W}_{R}$	$[b]$	$[b]^{W}$	$[b]_{R}$	$[b]^{W}_{R}$
$[a]$	$[a]$	$[a]$	$[a]_{R}$	$[a]_{R}$	$[a]$	$[a]^{W}$	$[a]^{W}$	$[a]^{W}_{R}$
$[a]_{R}$	$[a]_{R}$	$[a]_{R}$	$[a]_{R}$	$[a]_{R}$	$[a]_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$
$[a]^{W}$	$[a]^{W}$	$[a]^{W}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}$	$[a]^{W}$	$[a]^{W}$	$[a]^{W}$
$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$	$[a]^{W}_{R}$
Case:	#0	#4	#1	#1	#0	#2	#3	#2+3

Fig. 1. Case analysis for exposure propagation.

property helps ensure termination). We can further categorise the rules of Figure 1:

- **No Change** - Cases marked #0 (flow to a variable with no exposure) and cases marked #4 (flow to a variable with write exposure) do not change the exposure a.
- **Addition of Read Exposure** - Cases marked #1 (flow to a variable with read exposure) results in variable a becoming read exposed.
- **Addition of Write Exposure** - Cases marked #2 (flow from a variable with write exposure) and cases marked #3 (flow from a variable with read exposure) result variable a becoming write exposed.

We now consider each of these cases in more detail:

Case #0 - node b does not have either type of exposure. This means that any values we read from b cannot be aliased by an external object; writing values into b will also not make them exposed. We can therefore infer that node a will simply keep its current exposure types (if any).

Case #1 - the current node a flows to a read-exposed node b. An example of this case is given in Figure 2. Here the solid lines represent the pointers, and the dashed lines represent the possible value flow between the variables. In our example variable b is directly read exposed (for example, it could be a return value of a public method in class X). In Figure 2, the value of a is possibly assigned to b, and then to c. This potential value transfer exposes the object initially referenced by a. Thus, since a is exposed by virtue of a value escaping from it, it is read exposed.

Case #2 - the current node a has a flow from a write-exposed node b (for example, a parameter of a public method). Figure 3 provides an example object graph with possible value flows. These value flows allow for the possibility of the value of external variable c to be transferred to b and then eventually to a. Similar to the previous example this results in an object that is aliased by both a and c. However, this time the exposure of a occurs by virtue of an aliased value being assigned to it, making node a write exposed.

Case #3 - the current node a has a flow from a read-exposed node b (Figure 4). In this case, the aliasing can occur due to the value of b flowing into both a and the node that is read

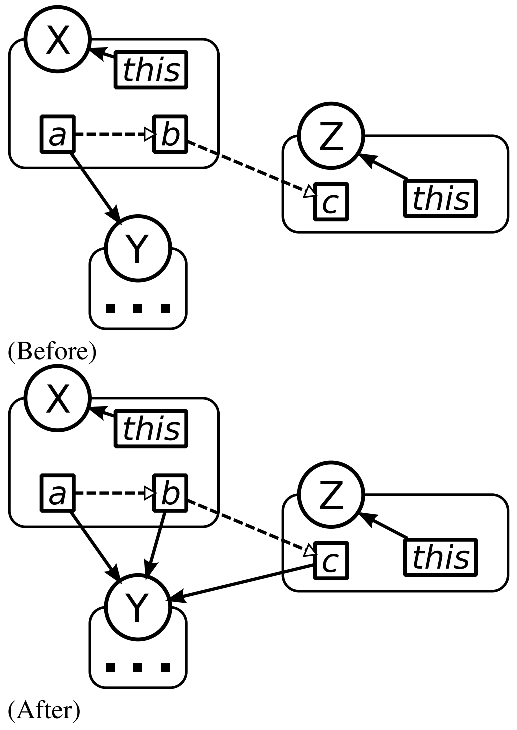


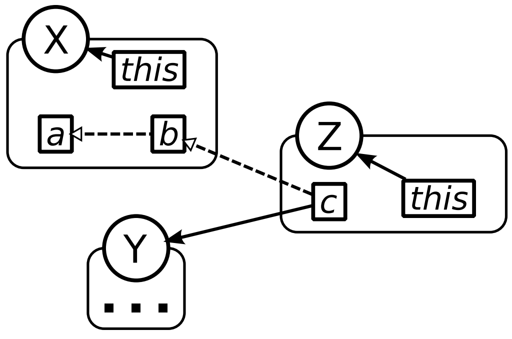
Fig. 2. Case #1

exposing b itself (in our example it is c). As with Case #1, the variable a becomes write exposed, because the exposed value is written into it (from node b).

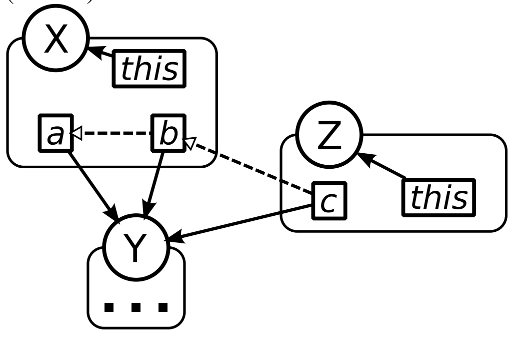
Case #4 - current node a has a flow to a write-exposed node b (Figure 5). Even though the variable a is interacting with an exposed variable b, unlike previous cases, a does not become exposed. Due to the value flows in our example, the only variable that changes its value is b - it can either receive a value from a or a value from c.

While b can potentially point at both Y1 and Y2 during the execution of the program. This means there are only three possible states for the variables: the initial state; the state where b references Y1 (State 1); and the state where b references Y2 (State 2). As we can see, none of these three states can result in a situation where a and c alias an object.

So far we have described the rules for calculating the change in the exposure of a node given a flow to or from its neighbour (i.e. Figure 1), avoiding the question of how and when these

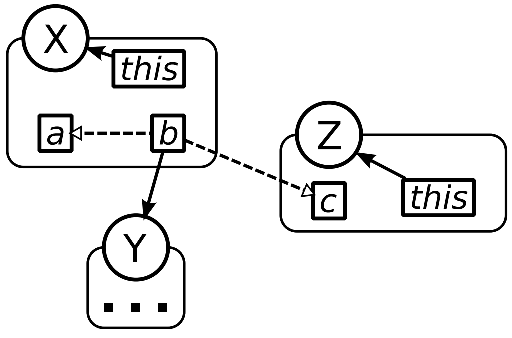


(Before)

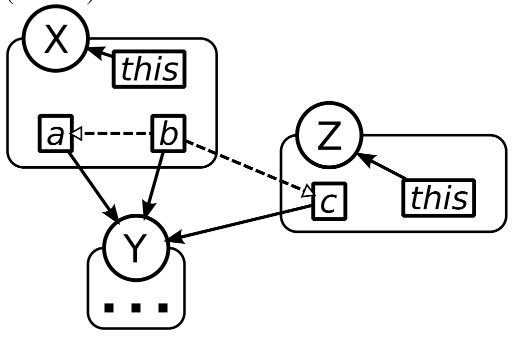


(After)

Fig. 3. Case #2



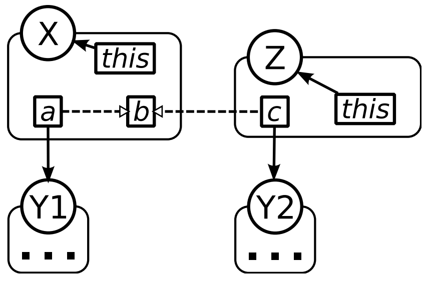
(Before)



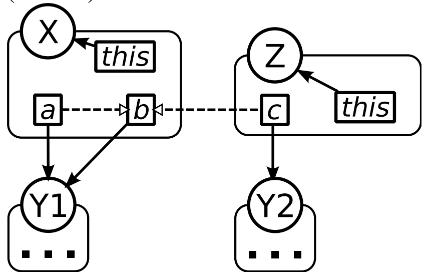
(After)

Fig. 4. Case #3

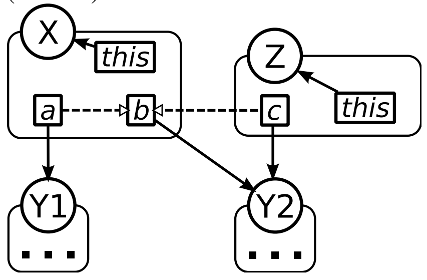
rules should be applied. An important fact to note about our inference rules is that there is no “wrong” time to apply them, since they either leave the exposure of the node the same, or add an additional exposure type — i.e. once an exposure type is inferred it is never removed. This means the order of rule applications does not matter, as long as we keep applying the rules until a fixed point is reached.



(Before)



(State 1)



(State 2)

Fig. 5. Case #4

C. Self Exposure Inference

In most cases our system *infers* field ownership by just looking at one class at a time. There is however a special case in which this cannot be done (note that we can still *check* field ownership modularly). Consider the following example:

```
public class Z {
  public Z(){
    S.staticField = this;
} }

public class S {
  public static Z staticField = ...;
}
```

When an object of class Z is created, it will immediately share its identity with a static field of class S, after which any number of external classes may potentially access it. The consequence of this is that any variables that can contain objects of type Z are read exposed. For example, consider the following:

```
public class MyClass {
  private Z myField = new Z();
}
```

Analysing this class in isolation, we would conclude that `myField` is not exposed and, hence, can be annotated with `@Owned`. Unfortunately, because Z’s constructor is *self-exposing*, we must consider `myField` to be exposed.

In order to address this problem, we first perform *self-exposure inference* before we perform ownership inference. Just as with field exposure inference, self exposure inference has to deal with cases where a value is propagated through a number of intermediate variables before being exposed outside of the instance as follows:

```
public class C {
  private Object f;

  public void m1() { m2(this); } // 1

  private void m2(Object par){ f=par; }// 2

  public Object getF(){ return f; } // 3
}
```

Here, a reference to the current object (`this`) is propagated from the special local variable (1) into a parameter and into `m2`, and then through field `f` into a public return value (2, 3).

We can determine whether a given class is self-exposed by reusing our algorithm for field exposure. The only difference is that instead of looking at whether the fields of the class have either read or write exposure, we only need to know if `this` reference has become read exposed (since it is not possible to assign to `this` in Java). Once the self-exposure of all classes is determined we can proceed with the field ownership inference as usual. The only addition is that all variables whose type allows them to contain objects of self-exposed classes are now marked as read exposed.

This finding has interesting consequences for ownership inference — it is not possible to safely infer ownership of variables inside a class without knowing the self-exposure information of their types. Furthermore, to ensure the inferred annotations are, in fact, modularly checkable we must provide a way to constrain the self-exposure of client classes. That is, if we assume a given class is not self exposed, then we must constrain its subclasses as well.

In fact, this problem is identical to that faced by the Java Compiler for ensuring **`final`** classes are not extended. For every class, the compiler must check that it's super class is not **`final`**. This is done by examining the signature of the immediate super class. Therefore, we can adopt a similar strategy. For example, by providing an annotation such as `@NotExposed` to prohibit subclasses from self exposing. The annotation checker can then easily check this modularly, in much the same way the Java Compiler already checks for the **`final`** modifier.

D. Arrays

The presence of arrays in the Java language introduces many implicit value flows in the target program which serves to complicate the ownership inference process. The simplest approach for dealing with arrays is to simply ignore these implicit flows and declare that all array elements are automatically exposed (this approach is used by UNO [23]). This maintains the safety of annotations because fields marked with `@Owned` are still guaranteed to be owned. This does however result in more conservative field ownership as any other variables that have a flow to or from array elements also become exposed.

Our inference tool uses a more precise approach than that used by UNO. Instead of simply deciding that all array elements are exposed, we ignore the value of the index during array accesses. This results in less conservative annotations while not resulting in major changes to the algorithm or incurring a large computational cost. From the point of view of flow analysis, we are modelling all elements of an array with a single representative. When it comes to generating variable flow information this allows us to represent all interactions with array elements as interactions with a single variable.

Consider the following example:

```
class MyClass {
  private String[] x = ...;

  private String a = ...;
  private String b = ...;

  public void myMethod(){
    x[0] = a;
    b =  x[5];
} }
```

There are two variable flows here: a flow from `a` to "elements of `x`" and from "elements of `x`" to `b`. While these flows may seem to be quite conservative in this case (because, in fact, variable `a` does not flow into variable `b`), such examples where specific indices are known are unusual. As another example, consider the following:

```
public class MyClass {
  private String[] x = ...;
  private String[] y = ...;

  public void myMethod(){
    for(int i = 0; i < x.length; i++){
      y[i] = x[i];
} } }
```

The only variable flow we would generate for the program is from "elements of `x`" to "elements of `y`".

Once the flow information of the array elements has been represented in this way, we can treat them in the same way we treat all other variables in the program. The only special case occurs when the base reference of the array has been exposed. When a base reference is exposed, it means that an object other than `this` has complete read and write access to all of the elements inside the array. Thus, we will have to mark the array elements to be both read and write exposed as well.

IV. Experimental Results

We ran OwnKit on a selection of benchmarks shown in Figure 6. The benchmarks came from the JKit [29] compiler suite and together with OwnKit are available for download[1].

As we present our ownership inference results, we also compare OwnKit to the closest existing ownership inference tool called UNO [23]. UNO uses a whole program analysis which (unlike OwnKit) does not generate modularly checkable

[1]http://homepages.ecs.vuw.ac.nz/~cat/ownkit/

Program	Description	LOC	Total Fields	% of Owned Fields		Classes	
				OwnKit	UNO	Self-Exp. %	Total
java-std	Core Java Standard Library packages (v1.5)	62,508	690	3.77	-	16.0	763
javacc	Parser Generator (v5.0)	36,672	406	4.7	11.8	13.3	150
polyglot	Compiler Framework excluding extensions (v1.3.2)	14,148	421	0.5	2.9	11.0	327
asm	Assembly Simulator (v3.2)	22,474	259	4.2	10.8	14.0	172
jgraph	Graph Visualization Tool (v5.9.2.0)	12,262	178	5.1	3.9	29.2	89
raytracer	From the `SPECjvm98` Suite	1,928	40	12.5	5.0	28.0	25
Average				**5.1**	**6.9**	**18.6**	

Fig. 6. OwnKit Ownership Inference Results. Self-Exp refers to the percentage of classes that are determined to be self-exposed by OwnKit.

annotations. Nonetheless, the comparison provides an interesting insight into the ownership latent in the existing code.

A. Ownership Inference Results

Figure 6 presents the ownership and self-exposure information generated by OwnKit and UNO. Lines of Code (LOC) data was generated using: http://www.dwheeler.com/sloccount/. Total Fields indicates the number of non-primitive fields in the target program. Like UNO, we decided to ignore fields of type `String` and fields of inner classes in order to make the comparison between the two more objective. We were not able to analyse `java-std` with UNO because of a dependency analysis bug in Soot.

Since UNO is not restricted to generated modularly checkable annotations, we would expect its approach to be more precise and, indeed, UNO generally generated more owned annotations. Cases where OwnKit outperformed UNO (`jgraph` and `jvm98_raytracer`) could be attributed to our more precise treatment of array element flows (UNO simply assumes all array elements are exposed).

The number of classes that were inferred to be self-exposing were quite high: around 19% on average, with some benchmarks reaching up to 29%. This could be attributed to the conservative nature of the self-exposure inference algorithm.

In order to gain a better understanding of the ownership and self-exposure patterns we analysed each of the `java-std` packages[2] in isolation as presented in Figure 7. As we can see, both field ownership and self-exposure numbers vary a great amount depending on the package.

B. Causes of Exposure

As part of our experiment we have analysed each of the benchmarks by using OwnKit in "Complete Reasoning Mode". The statistics on the exposure reasons of the fields is given in Figure 8. Note that each of the fields can have multiple exposure reasons. The reasons are as follows:

- **NonPrivate** - The field is not marked as `private` (for example, it is `public` or `protected`).
- **Flow to Read** - A value from the field flows into a `READ` exposed node (Case #1).
- **Flow from Read** - A value flows from a `READ` exposed node into the field (Case #3).
- **Flow from Write** - A value flows from a `WRITE` exposed node into the field (Case #2).
- **Self-Exposed** - The type of the field is a subtype of some self-exposing class.
- **Static** - The field is `static`.
- **Other-Instance** - The field is accessed by a different instance of the containing class.

As we can see the exposure reasons differ a lot from benchmark to benchmark. This could be attributed to the fact that our benchmarks consist of very different classes of code bases. For instance, since `java-std` is a large and general API library, we would expect its coding style and patterns to be quite different from `jvm98_raytracer`, which is a small and narrow purpose tool.

Across all of the benchmarks, we can see that all of the exposure mechanisms played significant roles, with the exception of "Other-Instance" which only occurred in around 6% of the cases. A particularly interesting result is a high proportion of fields that are exposed due to not being declared as `private` — 60% on average, however even as high as 95% in `jgraph`. According to a brief inspection of `jgraph`, most of the non-private exposure comes from the fact that non-primitive fields are often declared as `protected`. Another unexpected result is that while the average number of self-exposed classes

[2]Class `java.util.Collections` was excluded due to JKit type-checking problems.

Package Name	Lines of Code	Total Fields	% Owned Fields	Total Classes	% Self-Exposed
java	62,508	690	3.77	763	15.99
java.lang	16,490	221	2.26	212	10.38
java.lang.reflect	1,482	44	0.00	22	18.18
java.lang.instrument	75	2	0.00	5	0.00
java.lang.ref	246	9	11.11	12	8.33
java.lang.annotation	72	15	0.00	6	0.00
java.lang.management	529	8	0.00	16	0.00
java.io	10,269	101	12.87	113	19.47
java.util	35,749	368	5.98	438	17.81
java.util.concurrent	6,468	69	18.84	90	23.33
java.util.concurrent.locks	853	4	0.00	7	14.29
java.util.concurrent.atomic	1,158	13	0.00	18	22.22
java.util.regex	4,345	12	0.00	86	5.81
java.util.jar	1,378	27	3.70	17	35.29
java.util.prefs	1,879	21	4.76	21	9.52
java.util.logging	2,499	57	14.04	31	16.13
java.util.zip	2,101	27	18.52	21	4.76

Fig. 7. Ownership and Self-Exposure Results for Individual Packages using OwnKit.

is 19%, they result in 39% of all exposures. This demonstrates that self-exposure is indeed a significant problem which cannot be simply ignored during ownership inference.

V. Related Work

Numerous systems exist in the literature for inferring ownership and related encapsulation properties [30], [20], [21], [22], [23], [25], [31], [32], [33], [34]. The vast majority of these (in fact, all except [31], [34]) build on an interprocedural pointer analysis (or a similar analysis, such as type analysis) — making them unsuitable for inferring modularly checkable ownership annotations (as discussed in §I-A). In such cases, the precision and efficiency of the side-effect analysis is largely determined by that of the underlying pointer analysis. Numerous pointer analyses have been developed which offer different precision-time trade-offs (see e.g. [35], [36], [27]).

Ma and Foster developed an ownership and uniqueness inference tool for Java called UNO [23]. This is based on the Soot compiler framework [37]. Similarly to OwnKit, UNO's definition of object ownership is based on flexible alias protection [38] and provides a deep ownership property. The tool uses an intraprocedural points-to analysis followed by a whole-program, on-demand evaluation of ownership, uniqueness, containment and other related predicates. As such, the inferred annotations are not modularly checkable.

Milanova and Vitek developed a tool for inferring deep ownership of Java programs [33]. Their approach employs an interprocedural points-to analysis to approximate the object graph, and then computes dominators on this to identify owners. Their system is not designed to be used in conjunction with a type checker and, in particular, does not generate modularly checkable ownership annotations.

Poetzsch-Heffter *et al.* developed a system for inferring ownership annotations within components [21]. Here, the programmer must provide appropriate ownership annotations at component "boundaries", but the remainder are automatically inferred using a whole-component type analysis. As such, this system explicitly eschews modularly checkable annotations in favour of reducing the overall number of annotations (and, hence, easing the burden on the programmer of maintaining them).

AliasJava [30] present a type system which captures uniqueness and ownership information through a range of annotations, including: `unique`, `lent`, `owned` and `shared`. Whilst their annotation system is more expressive than ours, the system as presented needs additional work to support modular checking.

Generic Universe Types (GUT) [39] is an ownership type system which enforces the *owner-as-modifier* encapsulation discipline (which differs from the *owners-as-dominators* — or deep ownership — approach taken in this paper). Dietl *et al.* developed a tunable static inference for Generic Universe Types [31]. Their approach is unusual and relies on encoding the inference as a boolean satisfiability problem. As with our approach, Dietl *et al.* separate annotation inference from annotation checking, using the latter to ensure inferred annotations are correct. The issue of modular checkability was not explicitly considered, and it remains unclear what extensions would be necessary to support this. In particular, we note that the issue of self-exposure — which is necessary for any modularly checkable system — is not addressed. Finally, to help capture programmer intent, their system supports interaction with the programmer by allowing him/her to modify the inferred annotations and propagate the implications of this.

The recent work of Huang *et al.* presents a general framework for inference and checking of ownership properties [34]. In particular, instantiations of their system for both *owners-as-dominators* and *owners-as-modifiers* are given. In general, their approach appears quite similar to ours. The issue of modular checkability is not considered and it remains unclear what extensions would be necessary to support this. In particular, the issue of self-exposure — necessary for any modularly checkable ownership system — was also not addressed.

Finally, Negara *et al.* consider the issue of *ownership*

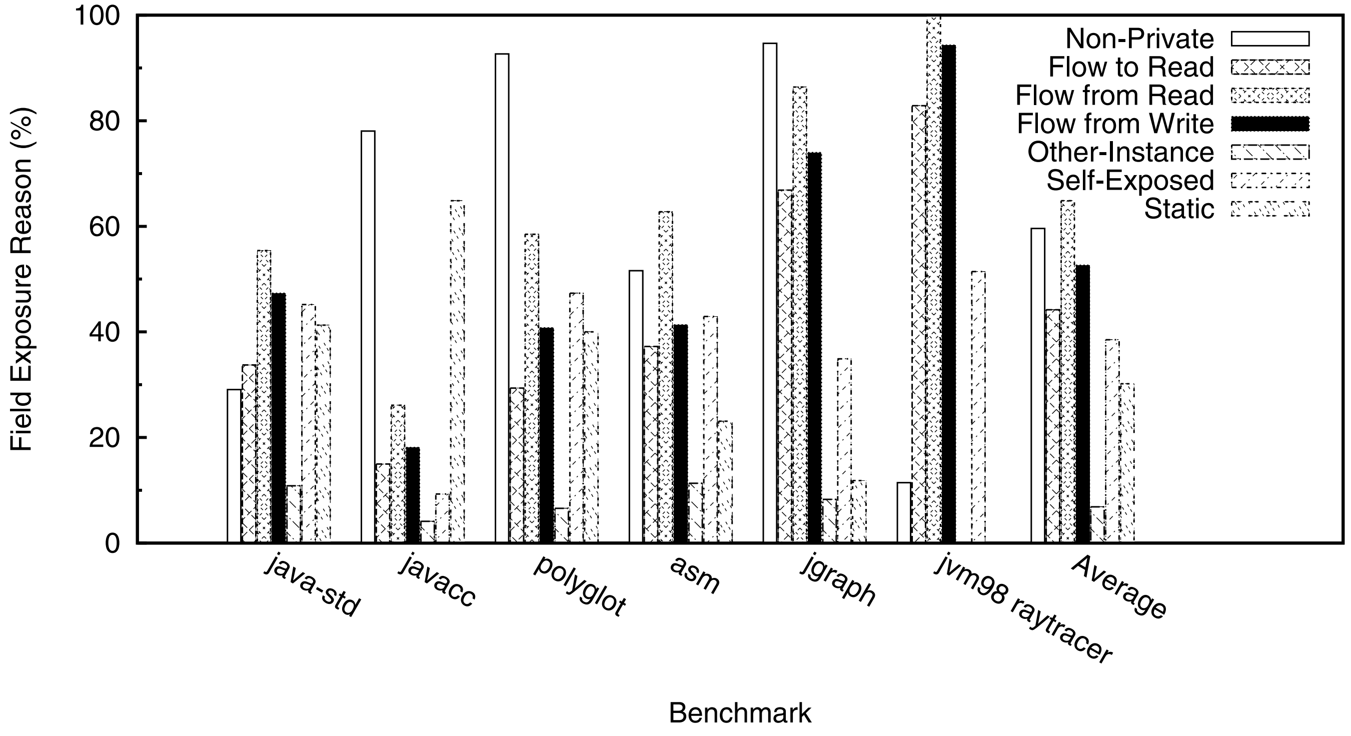


Fig. 8. Field Exposure Reasons

transfer in an actor system to facilitate a call-by-value semantics of message sends whilst avoiding deep clones (as much as possible) [32]. Again, their inference relies on an interprocedural points-to analysis making it unsuited to modular checking of inferred annotations.

A. Other Inference Systems

The inference and checking of *pure methods* presents a similar problem to that discussed in this paper. In particular, the use of pointer analysis as a building block remains critical to many modern side-effect and purity systems (e.g [40], [41], [42]). As such, work on purity systems has been largely unconcerned with the issue of modular checkability. One exception is the JPure system which was explicitly designed to infer modularly checkable annotations [19].

The JQual system infers user-defined type qualifiers for Java programs using a context-insensitive flow analysis similar, in many ways, to an interprocedural pointer analysis [43]. Several works focus on inference of reference immutability annotations to facilitate safe sharing of objects [44], [45]. The work of Huang *et al.* is particular relevant here, as their system divides into separate inference and checking tools [44]. Again, the issue of modular checkability is not explicitly addressed.

Finally, DPJizer infers method effect summaries and annotates the program accordingly [46]. This is used to help porting of Java programs to DPJ — a language for writing safe parallel programs [10]. DPJ provides a type system that guarantees noninterference of parallel tasks.

VI. Conclusions

We have presented a novel ownership inference system which comprises of an *ownership inference* and an *ownership checker*. The ownership inference operates as a source-source translation for annotating legacy code, whilst the ownership checker is designed to be used in the same way as the type checking facility of a modern compiler. Unlike previous work, our ownership inference system generates modularly checkable annotations. This means the annotations in one class can be checked in isolation from others without the need for a whole-program analysis.

We have also reported on experiments using our system on a selection of Java programs, and compared our results with the closest existing ownership inference tool (UNO) available online [23]. Whilst our ownership results are less precise than those of UNO, it is important to remember that UNO is not constrained to generating modularly checkable ownership annotations.

Finally, a more detailed presentation of this work which includes a formalisation of the ownership inference system, and additional technical discussion is available [28].

References

[1] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu, "An empirical investigation of the influence of a type of side effects on program comprehension," *IEEE TSE*, vol. 29, no. 7, pp. 665–670, 2003.

[2] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in Java," in *Proc. CCS*. ACM, 2008, pp. 161–174.

[3] D. Clarke, J. Potter, and J. Noble, "Ownership Types for Flexible Alias Protection," in *OOPSLA*. Vancouver, Canada: ACM, Oct. 1998, pp. 48–64.

[4] D. Clarke and S. Drossopoulou, “Ownership, Encapsulation, and the Disjointness of Type and Effect,” in *Proc. OOPSLA*. Seattle, WA, USA: ACM, Nov. 2002, pp. 292–310.

[5] D. Clarke, “Object Ownership and Containment,” Ph.D. dissertation, School of CSE, UNSW, Australia, 2002.

[6] C. Boyapati, R. Lee, and M. Rinard, “Ownership types for safe programming: Preventing data races and deadlocks,” in *Proc. OOPSLA*. Seattle, WA, USA: ACM, November 2002.

[7] J. Aldrich, V. Kostadinov, and C. Chambers, “Alias Annotations for Program Understanding,” in *Proc. OOPSLA*. Seattle, WA, USA: ACM, Nov. 2002, pp. 311–330.

[8] A. Potanin, J. Noble, D. Clarke, and R. Biddle, “Generic ownership,” in *Proc. OOPSLA*. Portland, OR, USA: ACM, 2006.

[9] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst, “Ownership and immutability in generic java,” in *Proc. OOPSLA*. ACM, 2010, pp. 598–617.

[10] R. L. B. Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, “A type and effect system for deterministic parallel Java,” in *Proc. OOPSLA*. ACM Press, 2009, pp. 97–116.

[11] A. J. Craik and W. A. Kelly, “Using ownership to reason about inherent parallelism in object-oriented programs,” in *Proc. CC*, ser. LNCS. Springer-Verlag, 2010.

[12] P. Müller, *Modular Specification and Verification of Object-Oriented Programs*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2002, vol. 2262.

[13] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte, “Verification of object-oriented programs with invariants,” *JOT*, vol. 3, no. 6, pp. 27–56, 2004.

[14] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard, “Ownership types for safe region-based memory management in Real-Time Java,” in *Proc. PLDI*. ACM Press, 2003, pp. 324–337.

[15] A. Potanin, J. Noble, T. Zhao, and J. Vitek, “A high integrity profile for memory safe programming in real-time Java,” in *The 3rd workshop on Java Technologies for Real-time and Embedded Systems*, San Diego, CA, USA, 2005.

[16] C. Andreae, J. Noble, Y. Coady, C. Gibbs, J. Vitek, and T. Zhao, “Stars: Scoped types and aspects for real-time systems,” in *Proc. ECOOP*. Springer-Verlag, 2006.

[17] A. Potanin, J. Noble, D. Clarke, and R. Biddle, “Defaulting Generic Java to Ownership,” in *Proc. Workshop on Formal Techniques for Java-like Programs*. Oslo, Norway: Springer-Verlag, Jun. 2004.

[18] S. Nägeli, “Ownership in design patterns,” Master’s thesis, Software Component Technology Group, Department of Computer Science, ETH Zurich, 2006.

[19] D. J. Pearce, “JPure: A modular purity system for java,” in *Proc. CC*, ser. LNCS, vol. 6601. Springer-Verlag, 2011, pp. 104–123.

[20] S. E. M. III and A. L. Souter, “An object ownership inference algorithm and its application,” in *MASPLAS*, 2004.

[21] A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer, “Infering ownership types for encapsulated object-oriented program components,” in *Program Analysis and Compilation*, ser. LNCS, vol. 4444. Springer-Verlag, 2006, pp. 120–144.

[22] Y. Liu and A. Milanova, “Ownership and immutability inference for UML-based object access control,” in *Proc. ICSE*. IEEE Computer Society, 2007, pp. 323–332.

[23] K.-K. Ma and J. S. Foster, “Inferring aliasing and encapsulation properties for Java,” in *Proc. OOPSLA*. ACM, 2007, pp. 423–440.

[24] A. Milanova, “Static inference of universe types,” in *IWACO*, 2008.

[25] A. Milanova and Y. Liu, “Static ownership inference for reasoning against concurrency errors,” in *ICSE Companion*, 2009, pp. 279–282.

[26] O. Lhotak and L. Hendren, “Scaling Java points-to analysis using SPARK,” in *Proc. CC*, ser. LNCS, vol. 2622, 2003, pp. 153–169.

[27] D. J. Pearce, P. H. J. Kelly, and C. Hankin, “Efficient field-sensitive pointer analysis for C,” *ACM TOPLAS*, vol. 30, 2007.

[28] C. Dymnikov, “OwnKit: Ownership inference for Java,” Master’s thesis, School of Engineering and Computer Science, Victoria University of Wellington, 2012.

[29] “Java Compiler Kit,” http://homepages.ecs.vuw.ac.nz/~djp/jkit/.

[30] J. Aldrich, V. Kostadinov, and C. Chambers, “Alias annotations for program understanding,” in *Proc. OOPSLA*. ACM Press, 2002, pp. 311–330.

[31] W. Dietl, M. D. Ernst, and P. Müller, “Tunable static inference for generic universe types,” in *Proc. ECOOP*, ser. LNCS, vol. 6813. Springer-Verlag, 2011, pp. 333–357.

[32] S. Negara, R. K. Karmani, and G. A. Agha, “Inferring ownership transfer for efficient message passing,” in *In Proc. PPOPP*. ACM Press, 2011, pp. 81–90.

[33] A. Milanova and J. Vitek, “Static dominance inference,” in *Proc. TOOLS*, ser. LNCS, vol. 6705, 2011, pp. 211–227.

[34] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst, “Inference and checking of object ownership,” in *Proc. ECOOP*, ser. LNCS, vol. 7313. Springer-Verlag, 2012, pp. 181–206.

[35] A. Rountev, A. Milanova, and B. G. Ryder, “Points-to analysis for Java using annotated constraints,” in *Proc. OOPSLA*, 2001, pp. 43–55.

[36] J. Whaley and M. S. Lam, “Cloning-based context-sensitive pointer alias analysis using Binary Decision Diagrams,” in *Proc. PLDI*. ACM Press, 2004, pp. 131–144.

[37] “Soot: a Java Optimization Framework,” http://www.sable.mcgill.ca/soot/.

[38] J. Noble, J. Vitek, and J. Potter, “Flexible Alias Protection,” in *Proc. ECOOP*, ser. LNCS, vol. 1445. Springer-Verlag, Jul. 1998, pp. 158–185.

[39] W. Dietl, S. Drossopoulou, and P. Mller, “Generic Universe Types,” in *Proc. ECOOP*. Springer, 2007, pp. 28–53.

[40] A. Rountev, “Precise identification of side-effect-free methods in Java,” in *Proc. ICSM*. IEEE Computer Society, 2004, pp. 82–91.

[41] A. Salcianu and M. Rinard, “Purity and side effect analysis for Java programs,” in *Proc. VMCAI*, 2005, pp. 199–215.

[42] A. Milanova, A. Rountev, and B. G. Ryder, “Parameterized object sensitivity for points-to and side-effect analyses for Java,” *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 1–11, 2002.

[43] D. Greenfieldboyce and J. S. Foster, “Type qualifier inference for java,” in *Proc. OOPSLA*. ACM, 2007, pp. 321–336.

[44] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, “ReIm & ReImInfer: Checking and inference of reference immutability and method purity,” in *Proc. OOPSLA*, 2011, p. (to appear).

[45] J. Quinonez, M. S. Tschantz, and M. D. Ernst, “Inference of reference immutability,” in *Proc. ECOOP*, ser. LNCS, vol. 5142. Springer-Verlag, 2008, pp. 616–641.

[46] M. Vakilian, D. Dig, R. L. Bocchino, J. Overbey, V. S. Adve, and R. Johnson, “Inferring method effect summaries for nested heap regions,” 2009, pp. 421–432.