OwnKit: Ownership Inference for Java

by

Constantine Dymnikov

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2013

Abstract

Object ownership allows us to statically control run-time aliasing in order to provide a strong notion of object encapsulation. Unfortunately in order to use ownership, code must first be annotated with extra type information. This imposes a heavy burden on the programmer, and has contributed to the slow adoption of ownership. Ownership inference is the process of reconstructing ownership type information based on the existing ownership patterns in code. This thesis presents OwnKit — an automatic ownership inference tool for Java. OwnKit conducts inference in a modular way: by only considering a single class at the time. The modularity makes our algorithm highly scalable in both time and memory usage.

Acknowledgments

I would like to thank Vicky for being extra awesome, my family for being supportive and supervisors Dr Alex Potanin and Dr David Pearce for being a continuous source of guidance and inspiration.

Finally, I would also like to thank coffee for making this thesis possible.

Contents

1 Introduction **1**

1.1 Object Ownership . . . 3
1.2 OwnKit . . . 3
1.3 Contributions . . . 4
1.4 Chapter Outline . . . 4

2 Background **5**

2.1 Aliasing . . . 5
2.2 Ownership . . . 7
2.2.1 Flexible Alias Protection . . . 7
2.2.2 Generic Ownership . . . 8
2.2.3 @Owned . . . 9
2.2.4 Universes . . . 9
2.3 Related Work . . . 11
2.3.1 Points-to Analysis . . . 11
2.3.2 UNO . . . 12
2.3.3 Dominance Inference . . . 12
2.3.4 Generic Universe Inference . . . 13
2.3.5 Boxes Inference . . . 13

3 Ownership Inference with OwnKit **15**

3.1 Software Development with Ownership . . . 15
3.2 Modularity . . . 17
3.2.1 Modularly Checkable Annotations . . . 17
3.3 Value Flow and Exposure . . . 19
3.3.1 Value Flow . . . 19
3.3.2 Variable Exposure . . . 21
3.4 Ownership and Exposure Inference . . . 22
3.4.1 Directly Exposed Variables . . . 23
3.4.2 Indirect Exposure . . . 27
3.5 Self Exposure Inference . . . 29

3.6 Arrays 32

4 Formalisation **35**
4.1 Abstract Syntax 36
4.2 Class Flow Graph 37
4.2.1 CLFG Construction Algorithm 38
4.3 Variable Exposure 44
4.4 General Exposure Inference 45
4.4.1 Initialisation rules 46
4.4.2 Propagation rules 46
4.4.3 Rule application algorithm 49
4.5 Self-Exposure and Field Ownership Inference 52
4.5.1 Self-Exposure 52
4.5.2 Field Ownership 52
4.6 Correctness 53
4.6.1 Termination 53
4.6.2 Ownership Guarantees 55

5 Case Studies **57**
5.1 Methodology 57
5.1.1 Benchmarks 57
5.1.2 Experiment 57
5.2 Results 59
5.2.1 Ownership and Self-Exposure 59
5.2.2 Exposure Reasons 61
5.2.3 Performance 63

6 Conclusions **65**
6.1 Contributions 65
6.2 Future Work 66

Chapter 1

Introduction

Software systems are inherently complex [11]. One of the most common ways to deal with complexity is to make the system modular. In object-oriented systems this principle manifests itself as *object encapsulation* - each object has an interface (public methods and fields) as well as a hidden internal implementation. The idea is that the amount of information that has to be considered at any one time is reduced by the fact that modules only depend on the interfaces and guarantees, not the internal implementation of other modules [24].

Despite the fact that encapsulation is a fundamental concept of object-oriented design, most of the currently used languages don't protect against another fundamental object-oriented concept: *aliasing*. Aliasing occurs whenever two reference fields or variables point at the same object in memory. While aliasing is very useful, it can also have a negative effect on object encapsulation by creating aliases that bypass an object's interface and access the internal implementation [23].

The undesired aliasing cannot be avoided by simply marking fields as `private`. For example, consider the classes in Figure 1.1. The field `tank` of class `Car` is marked as `private`, but due to an error in `driveCars` a single instance of the `FuelTank` becomes aliased by two different `Car` objects. This causes the call of `drive` on `Car a` to change the state of `Car b`, hence breaking the encapsulation property.

Aliasing bugs are notoriously hard to deal with because they create implicit dependencies between various parts of the system. This can make them very hard to detect because individual components can work correctly in isolation but fail when integrated together. Once discovered, the part of code containing the actual bug can often seem unrelated to the part of code that produces the error. For instance in our example the error occurs during the use of one of the `Car` objects, however the actual error is in the code that constructs them.

```
1  public class Main {
2     public void driveCars(){
3         FuelTank tank = new FuelTank(10);
4         Car a = new Car(tank);
5         Car b = new Car(tank);
6
7         a.drive();
8
9         b.drive(); // Out of fuel!
10    }
11 }
12
13 public class Car {
14    private FuelTank tank;
15
16    public Car(FuelTank tank){
17        this.tank = tank;
18    }
19
20    public void drive(){
21        tank.useFuel(10);
22    }
23 }
24
25 public class FuelTank {
26    private int fuel;
27
28    public FuelTank(int fuel){
29        this.fuel = fuel;
30    }
31
32    public void useFuel(int amount) throws OutOfFuelException{
33        if(fuel < amount) throw new OutOfFuelException();
34
35        fuel = fuel - amount;
36    }
37 }
```

Figure 1.1: Aliasing Example

```
1 public class Car {
2     @Owned private FuelTank tank;
3
4     ...
5 }
```

Figure 1.2: Ownership Annotation Example

1.1 Object Ownership

The concept of object ownership provides us with the ability to mark certain fields or variables as *owned* — part of the representation of the containing object [32]. Various ownership checking systems have been developed over the years [32, 14, 39, 12, 13, 8]. Most of these systems work by analysing the code at compile-time in order to ensure that none of the owned fields or variables have aliases outside of the containing object. For instance, in our previous example we could have annotated the field `tank` as owned (see Figure 1.2). This would have allowed us to pinpoint the bug automatically at compile-time, instead of first having to discover it at run-time and then try to find the line of code responsible.

The aliasing constraints provided by ownership mean that it is easier to isolate the influence that one of the program's components has on the other. The improved ability to modularly analyse the program has importance in many areas:

- Concurrent and Parallel Systems [45, 8, 43, 15].
- Real-Time Systems [10, 40, 6]
- Specification Languages [29, 7]
- Program Comprehension [19].

1.2 OwnKit

Most existing approaches to ownership rely on the programmer to add type annotations by hand — a difficult and time-consuming process [37]. This becomes especially problematic when dealing with large legacy systems that lack ownership annotations.

Our proposed solution to this problem is to divide the ownership tool set into two parts: an ownership inferrer and an ownership checker. The ownership inferrer allows us to automatically create ownership type annotations. Since the

inferrer cannot always capture the programmer's intent, the annotations can be manually edited. The checker is then used to ensure that the resulting set of type annotations is consistent.

One of the important distinctions between the ownership annotation schemas is whether or not the annotations can be checked in a modular way — that is, by only considering a single class or method at a time. Checking modularly allows for a faster and more scalable type checking algorithm, as well as making it easy to implement the checking inside the compiler. In this thesis we present OwnKit [3] — an ownership inference tool that creates modularly checkable annotations.

1.3 Contributions

The contributions of this thesis are as follows:

- Implementation of OwnKit — a modular ownership inference tool;
- Extension to the OwnKit inference algorithm in order to facilitate a more accurate analysis of array elements;
- Corpus study comparing the accuracy and performance of OwnKit and UNO [26].

1.4 Chapter Outline

The rest of the thesis is split into the following chapters:

- **Background** - An overview of concepts and algorithms our inference relies on and a survey of existing approaches to ownership inference.
- **Ownership Inference with OwnKit** - An explanation of how OwnKit fits into the software development process as well as a top-down view of the inference algorithm.
- **Formalisation** - A formal and detailed presentation of the inference algorithm, including a discussion of termination and the correctness of inferred annotations.
- **Case Studies** - Explanation and analysis of a small comparative corpus study.
- **Conclusions** - Summary of the thesis and an outlook on future work.

Chapter 2

Background

In this chapter we will present the concepts of aliasing and ownership in object-oriented programming languages, as well as common static analysis techniques and concepts that our inference algorithm relies on.

2.1 Aliasing

We say that an object in memory is *aliased* if there exist two or more variables that reference it at the same time. Consider the following example:

```
1 public void method(){
2     Car a = new Car("Yellow"); // Car 1
3     Car b = new Car("Red"); // Car 2
4
5     b = a;
6
7     b.setColour("Green"); // Car 1 is now green
8     a.getColour(); // returns "Green"
9 }
```

Figure 2.1: Creation and usage of an object alias.

Just before the statement `b = a` is executed the references and objects in memory can be pictured as follows:

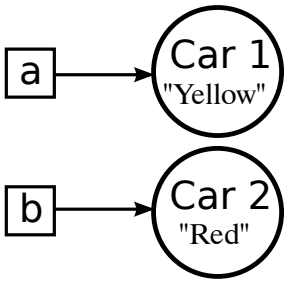


Figure 2.2: Variables `a` and `b` before the assignment statement is executed.

As we can see each of the variables refers to a different object in memory, hence there is no aliasing. However the execution of the assignment statement `b = a;` makes both `a` and `b` refer the same object:

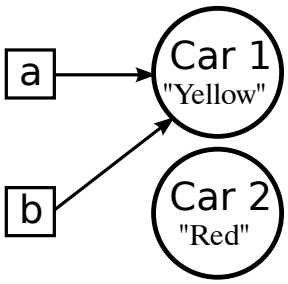


Figure 2.3: Variables `a` and `b` after the assignment statement is executed.

At this point in time we can say that object `Car1` has been *aliased*, and that `a` and `b` are *aliases*. The effect of aliases is that executing the statement `b.setColour(“Green”)` affects the statements that invoke methods on `a`.

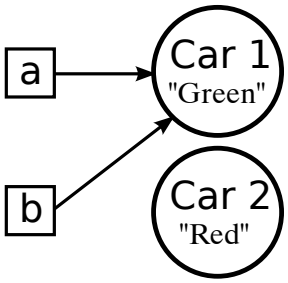


Figure 2.4: Changing state of the aliased object.

Aliasing is an essential feature in object-oriented languages as without it any object graph [41] (a graph where nodes are run-time objects, references are directional edges) would be limited to being a tree, severely restricting expressiveness of the language. Aliasing also allows for major performance improvements because large data structures can simply be referenced by multiple components instead of having to be copied.

2.2 Ownership

The concept of object ownership [32, 14] has emerged in order to address the fundamental aliasing problem. The idea is that since aliasing cannot be removed completely, we need to be able to control the usage of it in order to preserve encapsulation. Ownership type systems allow us to mark certain fields or variables as part of the *internal representation* [32] of an object. We say that such fields or variables are *owned* by the containing object. Once the field or variable has been marked it is possible to use this type information in order to statically ensure that the field cannot have aliases outside of the containing object [9].

In the rest of this section we will give an overview and comparison of different ownership systems that have been proposed in literature.

2.2.1 Flexible Alias Protection

Flexible Alias Protection [32] is the earliest work to put forward the concept of ownership (even though the actual word "ownership" is not used). The paper provides a conceptual model for enforcing object encapsulation based on the separation of roles of objects inside the program.

Encapsulation is presented in terms of an *aggregate object* — that is, an object that acts as an interface for a group of other objects that together form a single conceptual entity. For instance `Car` is an aggregate object in the example from the previous chapter (Figure 1.1).

The main insight of the paper is that there are two types of objects an aggregate object interacts with: the objects that are part of the implementation of the single conceptual entity (*representation objects*) and objects that are not (*argument objects*). The idea is that any *alias protected container* (an aggregate object that is well encapsulated from the point of view of aliasing) must obey the following properties:

- Representation objects must not have aliases outside of the aggregate.
- The aggregate object must not depend on mutable aspects of an argument object's state.
- Each object should only have exactly one of these roles (i.e. representation or argument).

In order to ensure that these properties are followed, the programmer first needs to annotate all of the fields, parameters and return types with special *aliasing modes* indicating the roles each of them play. These aliasing mode annotations

can later be used to automatically ensure the encapsulation properties of the aggregate. The representation objects are identified with the **rep** aliasing mode. The aggregate objects have 4 sub-roles identified with **arg**, **free**, **val** and **var**.

The rules for use of each of the aliasing modes are as follows:

- **rep** - objects that are part of the aggregate representation. These objects are not allowed to have aliases outside of the aggregate.
- **arg** - objects that are arguments to the aggregate (e.g. "contained" by the aggregate). The declared type of the variable is not allowed to expose the mutable aspects of its state.
- **free** - variables or expressions that contain the only reference to an object.
- **val** - primitive values. These values can be used freely by the aggregate since they are immutable (the containing variable may change, but not the value itself).
- **var** - variables containing mutable objects with no aliasing or mutability restrictions.

Like most other ownership systems [14, 39, 12, 13, 8], Flexible Alias Protection enforces the *owner-as-dominator* or *deep ownership* property in the run-time object graph. The property states that if object `A` owns object `B` then `B` is located in a sub-graph that is dominated by `A`.

2.2.2 Generic Ownership

Ownership Generic Java [37, 38, 39] is an extension to the Java language that introduces ownership without imposing additional syntactic or run-time overheads. The language works by allowing generic parameters to not only specify the type of objects, but also their owner.

For example, Figure 2.5 shows code for a simplified `List` collection in the original Java language. The generic parameter `E` allows us to specify the type of objects that can be contained in the list. For example if we wanted to create a new list that can only contain cars, we would provide `Car` as a type parameter (1). Figure 2.6 shows a declaration and usage of `List` in Ownership Generic Java with the additional ownership information added. The `List` class now has two type parameters, the first specifies the type and the owner of the elements, while the second specifies the owner of the `List` object itself. `myMethod` shows the possible combinations of the type arguments that can be used to specify different aliasing restrictions for different instances of the `List` objects. The language uses `World`

```
1  public class List<E> {
2      ...
3      public E get(int pos){
4          ...
5      }
6
7      public void add(E element){
8          ...
9      }
10 }
11
12 public class MyClass {
13     public void myMethod(){
14         // List of cars (1)
15         List<Car> carList = ...;
16     }
17 }
```

Figure 2.5: `List` list declaration and usage in original Java.

as an owner parameter for objects that have no aliasing restrictions and `This` for specifying object owned by containing object.

2.2.3 @Owned

Our ownership type system uses the `@Owned` type annotation in order to indicate which (non-primitive) fields are owned by the containing object. For example, Figure 2.7 shows a field declaration annotated with `@Owned`. The annotation provides the guarantee of deep ownership. The aliases of `myField` will each fall into the one of two categories:

1. Fields of `this` (the containing object with type MyClass) and fields of `myField` itself.

2. Local variables, method parameters and method returns of `this` and `myField`.

The `@Owned` annotation provides the same guarantees as **rep** in Flexible Alias Protection. Inference of other ownership annotations, such as "same owner" [39] is a much harder problem from the point of view of modular inference and is not the focus of this work.

2.2.4 Universes

Universes [30, 31, 18, 16] is a variation on the deep ownership type system. While deep ownership restricts aliasing of the aggregate representation by enforcing

```
1  public class List<E extends Object<EOwner>,
2                                      Owner extends World> {
3      ...
4      public E get(int pos){
5          ...
6      }
7
8      public void add(E element){
9          ...
10     }
11 }
12
13 public class MyClass<Owner extends World> {
14     public void myMethod(){
15         // Public list of any cars (1)
16         List<Car<World>, World> a = ...;
17
18         // Public list of my cars (2)
19         List<Car<This>, World> b = ...;
20
21         // My list of my cars (3)
22         List<Car<This>, This> c = ...;
23     }
24 }
```

Figure 2.6: `List` list declaration and usage in Generic Ownership Java.

```
1  public class MyClass {
2      // Annotation on a field
3      @Owned private Object myField;
4
5      ...
6  }
```

Figure 2.7: `@Owned` example

the owner-as-dominator constraint, the universe approach uses a more flexible *owner-as-modifier* constraint on the object graph. *Owner-as-modifier* only focuses on disallowing the aggregate representation to be modified by external objects, while not making any restrictions on aliasing.

2.3 Related Work

In this section we discuss techniques related to static (compile-time) ownership inference as well provide an overview of other ownership inference approaches.

2.3.1 Points-to Analysis

At their core, ownership inference and checking analyses consist of analysing existing aliasing patterns in the target code. For instance, it is important to be able to determine whether two variables or fields can ever be aliases of each other.

A common technique used to answer this question is called *points-to analysis* [22, 34, 21]. Points-to analysis allows us to analyse a program at compile time in order to determine the possible sets of objects that each variable or a field can point at run-time — these are referred to as *points-to sets*. Once the points-to sets are computed, the task of determining if two variables may be aliases of each other simply becomes the task of checking if their respective points-to sets intersect.

Due to the fact that the general problem is undecidable (it is a special case of The Halting Problem [25, 42]), the points-to analysis algorithm must use some approximations. The most common of these is to approximate groups of runtime objects that are created using the same `new` expression with a single object.

For example, the method in Figure 2.8 creates an arbitrary number of new `Car` objects and assigns them to fields `f` and `g`. In order to determine the exact points-to sets of `f` and `g` we would have to analyse (the potentially very complex) condition of the `while` loop. Instead of this we approximate all of the created `Car` objects with just two: O_1 and O_2, each representing a different creation location (i.e. `new` expression). The resulting points-to sets for `f` and `g` are $\{O_1\}$ and $\{O_2\}$ respectively (assuming there is no other code that interacts with these fields). Since the sets do not intersect, we can conclude that `f` and `g` cannot be aliases of each other.

While our tool does not directly use any of the existing points-to analyses, the Class Flow Graph Construction described in Chapters 3 and 4 uses the similar approach.

```
1 public void method(){
2     while(...){
3         this.f = new Car(); //1
4         this.g = new Car(); //2
5     }
6 }
```

Figure 2.8: `@Owned` example

2.3.2 UNO

UNO [26] is an ownership and uniqueness inference tool for Java based on the Soot compiler framework [5]. Similarly to OwnKit, UNO's definition of object ownership is based on flexible alias protection [32] and provides deep ownership property.

The tool uses intra-procedural points-to analysis followed by a whole-program, on-demand evaluation of ownership, uniqueness, containment and other related predicates. For example in order to evaluate predicate `OwnField(f)` (field is owned by `this`) the system will evaluate `NEscField(f)` (field `f` does not escape `this`) and `OwnFieldIn(f,m)` (`f` is owned in all methods `m` that use it). This approach allows to avoid unnecessary sub-predicate evaluations once it is clear that the top-level predicate can not be true (for instance in the previous example, if `NEscField(f)` fails, then `OwnFieldIn(f,m)` will not be evaluated). Another advantage is the ability to present proof trees for the evaluated predicates.

Both OwnKit and UNO use static analysis in order to determine possible flows of values, with OwnKit using an intra-class flow analysis and UNO using an inter-procedural analysis. The main point of difference between the tools lies in the fact that UNO uses a whole-program approach in order to evaluate its predicates, while OwnKit analyzes a program strictly one class at a time. The effect of this is that in order to check ownership annotations produced by OwnKit we only need to consider a single class, which is not a case with UNO.

2.3.3 Dominance Inference

Dominance Inference [27, 28] is a technique for inference of both (deep) ownership types and universe types [18]. Like UNO and OwnKit, the algorithm also works on unannotated Java source code.

The inference uses a whole-program Anderson-style points-to analysis [20, 44] in order to create an approximation of all possible run-time object graphs. The resulting graph contains edges that can have one or more labels corresponding to the four different types of interactions between objects. These interactions in-

clude: creation of one object by another, passing of an object as a method parameter to another, return of an object out of the method, and exposure of `this` pointer to another object (similar to our concept of self-exposure).

In order to infer ownership properties, the inferrer uses an algorithm that computes a *dominance boundary* for each of the objects. The algorithm uses an iterative work-list approach and maintains `In` and `Out` object sets. The `In` set represents objects that can only be accessed through the currently analysed top-level object. This set initially consists of all objects that are directly created by the top-level object, while the `Out` set initially contains all of the remaining objects. The algorithm then analyses all of the possible connections that may cause objects in `In` to be accessible by an object in `Out`. If an object in the in set `In` is discovered to be exposed, it is moved to `Out` and its connections also analysed. The computed dominance boundary of an object corresponds to all the objects that it owns.

While both OwnKit and the dominance inference algorithm construct a graph structure that shows the possible flows of objects, the two structures are very different in nature. The object graph used in dominance inference describes the object flows for the entire program under analysis and has individual objects as nodes, while our class flow graph has individual variables as nodes and only describes flows inside a single class.

2.3.4 Generic Universe Inference

Tunable Static Inference for Generic Universe Types [17] is an approach for automatic inference of universe types [16, 30, 31]. The inference consists of 3 main stages: identification of type variables (additional type information, similar to aliasing modes in Flexible Alias Protection), generation of constraints on these variables and the solving of constraints in order to obtain concrete values for the type variables.

An interesting aspect of the universe inference is the absence of the most general solution for the type variables. To address this issue the paper presents the use of user-defined heuristics in order to find a solution that has a suitable balance between stronger encapsulation and easier information sharing.

2.3.5 Boxes Inference

Boxes [36] is an alias control schema based on ownership types. The system requires that the program is split into a series of *modules*, each of which is similar to *alias-protected containers* [32]. While containers are only allowed to have a single external interface providing encapsulation, modules can have multiple external

interfaces, potentially making the system more flexible.

Inferring Ownership Types for Encapsulated Object-Oriented Program Components [35] describes a technique for inferring box types. The inference is much more narrow in scope than the other approaches we have described. Before the inference can begin, the user needs to identify each of the modules together with all of the external interfaces. This can sometimes result in cases the where inference process fails due to the inconsistency of user annotations.

Chapter 3

Ownership Inference with OwnKit

In this chapter we will discuss the problem of ownership inference and present our approach. The aim of this chapter is to provide a feel for the problem as well as the basic intuition behind our solution. Formal details of the algorithm will be given in Chapter 4.

3.1 Software Development with Ownership

In order to evaluate the practical usefulness of ownership annotations, two criteria must be used:

1. **Completeness** - The more types augmented with ownership information are found the more it helps us to understand, verify or optimise the code in question.

2. **Correctness** - The type annotations must be correct, otherwise they cannot be used to make reliable judgments regarding the code in question.

To help achieve these requirements we use two tools: an ownership inferrer (called OwnKit) and an ownership checker. OwnKit is used to help create the initial set of type annotations, which then may be manually edited. After this, as code modifications are made the checker is used to ensure that the changes do not violate the annotations. This serves to stop changes to the code from subsequently breaking the ownership protocol.

There are two ways to create annotations in the code: *manual* and *automatic*. On the one hand, in the case of a program being written from scratch, it is easy for a programmer to simply add annotations manually as classes are created. However this approach becomes time consuming and error-prone if a large legacy code base is required to be annotated.

```
1 public class MyClass {
2     private String errorMessage = "Too fast!";
3     // Returns "OK" or an error message
4     public String doCheck(){
5         // Unfinished
6         return "OK";
7     }
8 }
```

Figure 3.1: Unfinished method

```
1 public class MyClass {
2     @Owned private String errorMessage = "Too fast!";
3     // Returns "OK" or an error message
4     public String doCheck(){
5         if(...) {
6             return "OK";
7         }
8         else {
9             // Error: Public method cannot return value of an owned field
10            return errorMessage;
11        }
12    }
13 }
```

Figure 3.2: Type checking error after the modification of the code

On the other hand automatic process is very good for dealing with large code bases, but may result in "overzealous" annotations. This happens because the automatic tool can only calculate de-facto owned fields, not the fields that were *intended* to be owned.

For example, when OwnKit analyses the class in Figure 3.1 it will infer that field `errorMessage` in Figure 3.1 is owned because its value is never given to the outside and therefore will annotate it as `@Owned`. However most programmers looking at this class will understand the comments and realize that `errorMessage` will probably be passed outside of the class once `doCheck` is complete. The problem with overzealous annotations is that once they are in place they will unnecessarily restrict future code by restricting operations on the owned fields. For example when the method `doCheck` is complete the class will fail the type checking process (Figure 3.2).

As we have mentioned in the previous chapter, the main reason for having the checker as a separate tool is to allow for the mixture of the two annotation approaches. For example, a previously unannotated program can be at first an-

notated automatically, then have some of the overzealous annotations removed manually. Then as the program continues to be modified. The ownership checker can be used at any point in time to verify that none of the `@Owned` fields have their values exposed. In this sense the ownership checker is used in a manner similar to a type checker.

3.2 Modularity

All program analysis algorithms can be divided into two broad categories:

- **Modular** - Algorithms that generate a complete solution for a particular unit (for example a class) by only iterating over elements (methods or statements) of the unit in question and using limited type information regarding other units.

- **Whole Program** - Algorithms that generate a complete solution for a unit by iterating over elements of all units in the program.

A particularly important example of a modular program analysis is the Java compiler (javac). In this case the "unit" of the target program is a single Java file; whilst the "complete solution" for a unit is the corresponding class file[1].

In most cases the Java compiler will only consider the current Java file while doing this, and disregard all other Java files in the target program. However, sometimes the current Java file may reference symbols defined in other Java files. If this happens the compiler will analyse the relevant files in order to determine the type information. The important point here is that the compiler will not attempt to compile these files, but only retrieve the type information: a comparatively cheap computational procedure.

3.2.1 Modularly Checkable Annotations

In our system we made the decision to make all of our `@Owned` annotations modularly checkable. This means that if a particular class has a number of `@Owned` annotations, our ownership checker can ensure their validity by only having access to the containing class and some of the basic type information about other classes. The modularity of our annotations sets our system apart from the other (non-modular) schemas (e.g. [26, 28, 16]).

[1]Technically a single Java file can produce multiple classes if it contains inner, anonymous or non-public classes.

```
1 public class MyClass {
2     private String myField = ...;
3     public String getMyField(){
4         return myField;
5     }
6 }
```

Figure 3.3: Single Class Example

One of the main advantages comes from the fact that both our proposed type checking algorithm and the Java compiler work on programs modularly. This means it is feasible to implement the checking of `@Owned` annotations as one of the Java compiler's type checking stages.

From a performance standpoint the modularity of annotations means that both our type checking and inferencing algorithms cannot be constructing and maintain large global data-structures such as call graphs or inter-procedural points-to analysis (e.g. as in [44]). The result of this is that the maximum memory needed to analyse the given program does not depend on the number of classes it has, only the size of the biggest class. Similarly, because the analysis of individual classes does not exchange any data, the analysis time of a whole program grows linearly with respect to the number of classes.

Another advantage of modular algorithms is the small amount of time required to find a new solution after changes to a small number of units. This is important for type checking algorithms as typically only a small number of classes are modified in each iteration of the target program. In particular, since the checker only has to consider the newly changed classes, it makes it easy to implement the checking as one of the stages of the compiler. Since the process of type checking different classes is not dependent on each other, it makes the process of type checking inherently easy to parallelize and distribute. The modular approach also makes it easier to allow the inferrer to be integrated with incremental compiler, such as found in Eclipse [1].

One of the disadvantages of modular inference is that it will typically produce more conservative annotations. This is because a modular algorithm has to assume that any value that can potentially leak outside of the currently analysed class is exposed. The whole-program approach on the other hand can analyse other units in the target program to see if they actually obtain the exposed value.

For example, consider a simple program in Figure 3.3 that consists of only a single class. Here a modular inferrer will start on `MyClass`, and after seeing public `getMyFieldS` method will immediately assume that `myField` is not owned. However a whole-program inferrer may be able to see that even though the value

of `myField` can be obtained by calling `getMyField`, this method is in fact never called.

It is important to note that the precision advantage mostly occurs whenever we can guarantee that no new classes will be loaded at run-time (for example analysis of complete applications). However, if we are analysing library code, we have to make conservative assumptions regarding the (currently unknown) classes of an application that will use the said library. In this case, both modular and whole-program approach would have to operate under a similar set of assumptions.

3.3 Value Flow and Exposure

We will now present the fundamental concepts used in our inference algorithm, *value flow* and *variable exposure*. The actual inference algorithm will be introduced in the next section.

3.3.1 Value Flow

We say that there is a **value flow** from non-primitive variable `A` to a non-primitive variable `B` if during the execution of the program a value of `A` *may* be passed into `B`. This concept allows us to abstract away from the actual mechanism of value transfer, for example assigning to a local variable or passing a field as an argument to a method. Figure 3.4 provides some examples of value flows inside a class.

We also do not concern ourselves with whether the value assignment is guaranteed to happen during the actual execution of a program. For example, consider the method with a non-trivial if statement in Figure 3.5. In this example each execution of `myMethod` may carry out one of two possible assignments, however in any particular execution only one may be taken. Because we define value flow as the the *possibility* of value transfer, we would say that there are two value flows occurring: a flow from `x` to `a` and a flow from `x` to `b`.

Note that value flow is strictly directional, for example an assignment `a = x` will result in a flow from `x` to `a`, but not the other way around. This property of value flows makes a directed graph the natural representation of all the flows inside a program: here nodes are individual variables and edges are the flows. We will present the formal definition of this flow graph in Chapter 4.

```
1  public class MyClass {
2      private String a;
3      private String b;
4
5      public void method(){
6          // Value flow from b to a
7          a = b;
8
9          // Value flow from a to par
10         work1(a);
11
12         // Value flow from the return of work2 into b
13         b = work2();
14     }
15
16     private void work1(String par){
17         ...
18     }
19
20     private String work2(){
21         String tmp = "123";
22
23         // Value flow from tmp to return of work2
24         return tmp;
25     }
26 }
```

Figure 3.4: Value Flow Examples

```
1  public class MyClass {
2      private String x,
3      private String a, b;
4      public int number = ...
5
6      public void myMethod(){
7          if(number > 5){
8              a = x;
9          } else {
10             b = x;
11         }
12     }
13 }
```

Figure 3.5: Inference with Conditional Statements

```
1  public class MyClass {
2      private List<String> myList = ...;
3
4      public List<String> getMyList(){
5          return myList;
6      }
7  }
8
9  public class External {
10     public void expose(){
11         MyClass mc = ...;
12         List<String> alias = mc.getMyList();
13         alias.add("bad");
14     }
15 }
```

Figure 3.6: Read Exposure Example. The value of `myList` can potentially leak through the `getMyList` method and then becomes aliased by a field in an another class.

3.3.2 Variable Exposure

Another important concept is **variable exposure**. A non-primitive variable is said to become exposed when external objects gain access to objects it references. There are two types of exposure a variable can have.

The first type of variable exposure is **Read Exposure**. The variable is said to be read exposed if its value is readable 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 class in Figure 3.6: we can see that an external object can potentially obtain the reference to field `myList`, create an alias and proceed to modify elements of the list. Despite this, it is important to note that in this example no external object can change the field reference `myList`.

The second type is **Write Exposure**. A variable is said to be write exposed if its value can be changed to point to an object that may also be aliased by a variable in an external object.

We present a similar example in Figure 3.7. Just as with the previous example, the external object is able to create an alias to field `myList`. This time however, it can change what list object `myList` references. The external object however, can never create a reference to the original value of `myList`.

It is also possible for a variable to have both or neither types of exposure, as shown in Figure 3.8. Public field `fieldA` is both read and write exposed because

```
1 public class MyClass {
2    private List<String> myList = ...;
3
4    public void setMyList(List<String> par){
5        myList = par;
6    }
7 }
8
9 public class External {
10   public void expose(){
11       MyClass mc = ...;
12       List<String> alias = ...;
13       mc.setMyList(alias);
14   }
15 }
```

Figure 3.7: Write Exposure Example. The value of `myList` can be overwritten by another class with the help of `setMyList` method. If the call of `setMyList` remembers the said value then this will result in the creation of an alias to `myList`.

```
1 public class MyClass {
2    public List fieldA = ...;
3    private List fieldB = ...;
4 }
```

Figure 3.8: Combinations of Exposure Types. The field `fieldA` can be both read and written to by external objects, giving it both types of exposure. Field `fieldB` on the other hand is safe from all external manipulation, making it have neither type of exposure.

we have to conservatively assume that external objects will both copy its values and assign their own values to it. In contrast, a private field that is never used is clearly not exposed (and hence is owned).

3.4 Ownership and Exposure Inference

In our system, exposure and ownership of fields have a simple relationship.

Definition: A variable (field, method parameter or method return value) is considered to be *owned* if it is neither read exposed nor write exposed.

So the problem of ownership inference comes down to identifying exposure types for every variable in the currently analysed class. From the point of view of carrying out the exposure inference, the variables can be separated into two

	Read Exposed	Write Exposed
Parameter of a Non-Private Method		✓
Return of a Non-Private Method	✓	
Non-Private Field	✓	✓
Non-This Method Parameter	✓	
Non-This Method Return		✓
Non-This Field	✓	✓

Figure 3.9: Direct Exposure

categories: *directly* and *indirectly* exposed. The remainder of this section describes how the exposure of both categories are inferred.

3.4.1 Directly Exposed Variables

In order to identify the exposure types of all variables in a class we start by identifying variables that are exposed simply by virtue of their declaration. Figure 3.9 shows all of the cases for this kind of "direct exposure".

The first category refers to variables and methods that belong to `this` object, but do not have the `private` visibility (`public`, `protected` and `package`). Non-private methods can be invoked either by code in external objects (in case of `public` and `package`, see Figure 3.10), or by code in the subclasses (in the case of `protected`, see Figure 3.11).

Due to the modular nature of our analysis, we have to conservatively assume that the values provided as arguments to these methods are exposed - giving the parameters write exposure (case #1 in our code examples). Similarly, we have to assume that the values returned by these methods will be potentially aliased resulting in read exposure (case #2). Non-private fields can potentially be both read and written to, giving them both types of exposure (cases #3-4)

The second category of direct exposure shown in Figure 3.9 is fields and methods that do not belong to `this`. These include both `static` methods and methods of external objects (see Figure 3.12). We consider their exposure types from the point of view of `this` object. This means that the method parameters of another object are read exposed (the external object can read them) as seen in case #1. The return values of other objects are write exposed since the external object can return an aliased value (case #2). Finally, just like in the previous example, fields are both read and write exposed (cases #3-4).

```
1  public class MyClass {
2      public List<String> x;
3
4      public List<String> foo(List<String> par){
5          ...
6      }
7  }
8
9  public class OtherClass {
10     public List<String> exposed;
11
12     public void expose(A a){
13         //1
14         exposed = a.foo(...);
15
16         //2
17         a.foo(exposed);
18
19         //3
20         exposed = a.x;
21
22         //4
23         a.x = exposed;
24     }
25 }
```

Figure 3.10: Exposure of `public` variables by an external object.

```
1  public class MyClass {
2      protected List<String> x;
3
4      protected List<String> foo(List<String> par){
5          ...
6      }
7  }
8
9  public class OtherClass extends MyClass {
10     public List<String> exposed;
11
12     public void expose(){
13         //1
14         exposed = super.foo(...);
15
16         //2
17         super.foo(exposed);
18
19         //3
20         exposed = super.x;
21
22         //4
23         super.x = exposed;
24     }
25 }
```

Figure 3.11: Exposure of `protected` variables by the subclass.

```
1  public class MyClass {
2     private List<String> f = ..;
3
4     public void method(OtherClass a){
5         //1
6         a.set(f);
7
8         //2
9         f = a.get();
10
11        //3
12        a.exposed = f;
13
14        //4
15        f = a.exposed;
16    }
17 }
18
19 public class OtherClass {
20    public List<String> exposed;
21
22    public void set(List<String> p){
23        this.exposed = p;
24    }
25
26    public List<String> get(){
27        return exposed;
28    }
29 }
```

Figure 3.12: Exposure of `other` variables

	[a] --→ [b]				[a] ←-- [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

Figure 3.13: Possible Cases During the Value Flow

3.4.2 Indirect Exposure

Once we know the exposure types for all of the directly exposed variables, we need to consider how their exposure affects other variables in the currently analysed class. To do so we consider all of the node pairs that are directly connected by value flows. Figure 3.13 shows the rules for inferring the exposure of 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 the flow. The rules only specify the exposure of node `a`; if we want to find the exposure of `b` we would have to swap the order of the nodes involved.

We can see that the only changes to the exposure of `a` (highlighted in black) are the additions of new exposure types and never the removal of existing exposure types. Furthermore we can see that the effects of the inference can be easily categorized:

- **No Change** - Cases #0 and #4.
- **Addition of Read Exposure** - Case #1, if node `a` already has write exposure, then its exposure remain unchanged.
- **Addition of Write Exposure** - Cases #2 and #3, again, if node `a` already has read exposure, then its exposure remain unchanged.

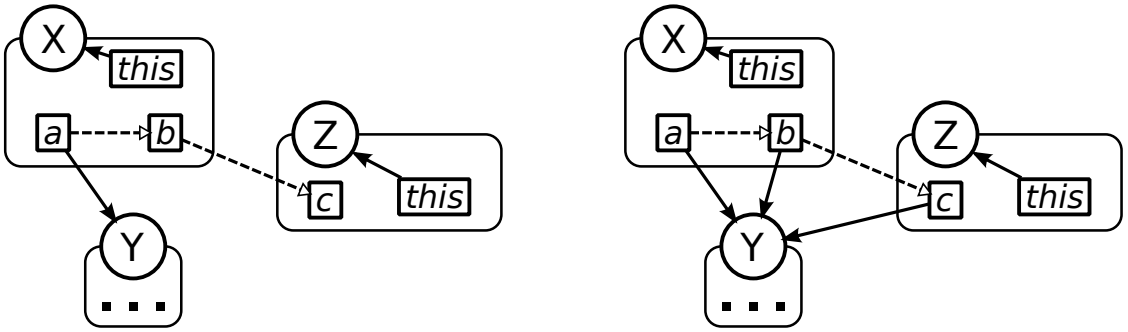
We will now carefully consider each of the cases and explain the reasoning behind them.

Case #0 - neighbour 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` has a flow to a read-exposed node `b`. An example of this case is given in Figure 3.14a. 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 X).

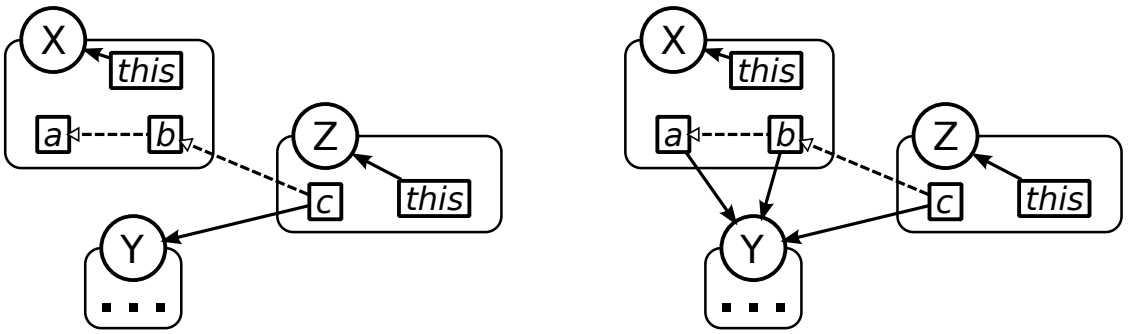
We now consider the possible effects of the value flow between the three variables involved. The value of a is possibly assigned to b, and then to c. Figure 3.14b shows the possibility of value transfer between the variables that results in aliasing. While the object Y has 3 aliases (a, b and c), the only important ones from the perspective of our rule are a and c. The two aliases mean that object Y can not be part of the internal representation of X. In turn that means that the variable a is exposed. According to our definition, since a is exposed by virtue of a value escaping from it, it is read exposed.



(a) Before (b) After

Figure 3.14: Case #1

Case #2 - the current node a has a flow from a write-exposed node b (for example a parameter of a public method). Figure 3.15a provides the initial 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 (Figure 3.15b). 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 a write exposed.

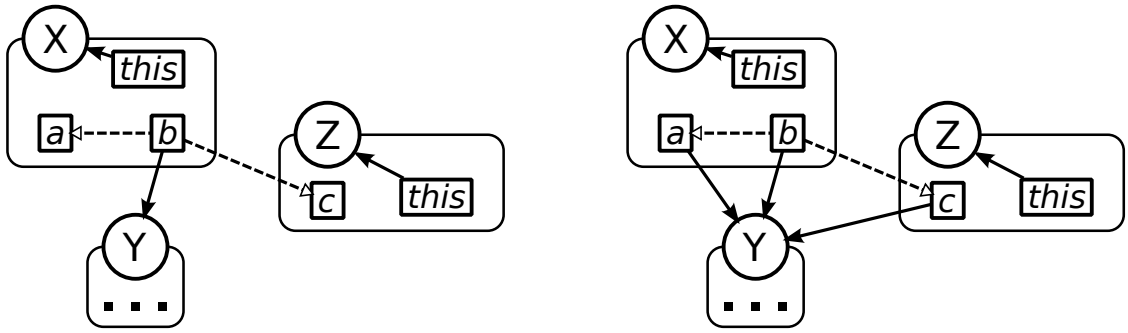


(a) Before (b) After

Figure 3.15: Case #2

Case #3 - the current node a has a flow from a read-exposed node b (Fig-

ure 3.16a). In this case, the aliasing can occur due to the value of `b` flowing into both `a` (Figure 3.16b) and the node that is read 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 `b`).



(a) Before (b) After

Figure 3.16: Case #3

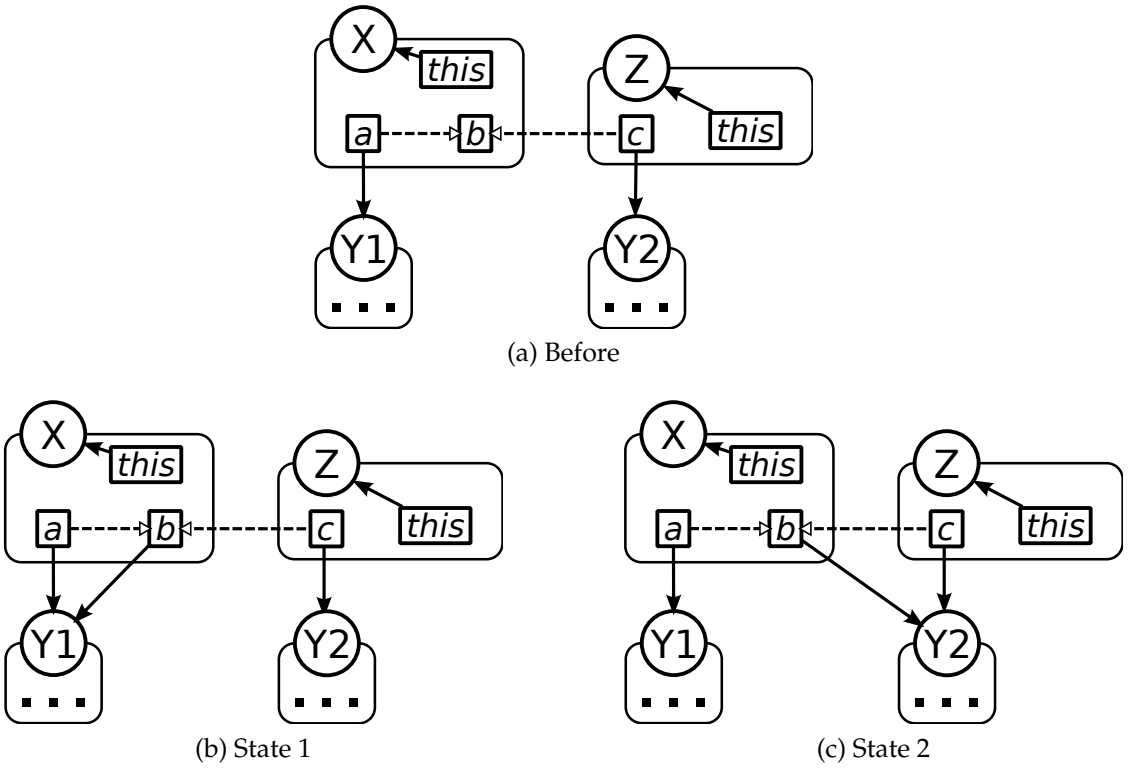
Case #4 - current node `a` has a flow to a write-exposed node `b` (Figure 3.17a). Even though the variable `a` is interacting with an exposed variable `b`, unlike previous cases, `a` does not become exposed. Due to the value flows in our example, the only variable that changes its value is `b` - it can 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 (Figure 3.17a); the state where `b` references `Y1` (Figure 3.17b); and the state where `b` references `Y2` (Figure 3.17c). 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 neighbor, avoiding the question of how and when these 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 - once the 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 all of the nodes have their exposure types inferred. The formal description of the application algorithm will be given in Chapter 4.

3.5 Self Exposure Inference

In most cases it is possible to infer field ownership by just looking at one class at a time. There is however a special case in which this can not be done.



(a) Before

(b) State 1

(c) State 2

Figure 3.17: Case #4

Consider the declaration of the class `Z` in Figure 3.18. 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 as shown in Figure 3.19.

Even though `MyClass` will never expose the value of `myField`, the field is automatically exposed as soon as we assign a value to it. In this situation we say that class `Z` is a *self-exposing class*.

In order to address this problem, we introduce the *Self-Exposure Inference* stage before we perform ownership inference. Just as with field exposure inference, self

```
1 public class Z {
2    public Z(){
3        S.staticField = this;
4    }
5 }
6
7 public class S {
8    public static Z staticField = ...;
9 }
```

Figure 3.18: Self-Exposing Class

```
1public class MyClass {
2    private Z myField = new Z();
3}
```

Figure 3.19: Self-Exposing Field

```
1 public class C {
2    private Object f;
3
4    public void m1(){
5        m2(this); //1
6    }
7
8    private void m2(Object par){
9        f = par; //2
10   }
11
12   public Object getF(){
13       return f; //3
14   }
15 }
```

Figure 3.20: Indirect exposure of `this`

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 can be seen in Figure 3.20. 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 (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 modular ownership inference - it is not possible to safely infer ownership of variables inside a class without knowing the self-exposure information of their types. In general, this is not problematic when dealing with code bases such as applications, where we can infer the self-exposure information for all of the classes. It does however mean that

```
1 class MyClass {
2    private String[] x = ...;
3
4    private String a = ...;
5    private String b = ...;
6
7    public void myMethod(){
8        x[0] = a;
9        b =  x[5];
10   }
11 }
```

Figure 3.21: Array Example

safe ownership annotations cannot be inferred on library code, unless we enforce constraints on the self-exposure of client classes. This issue however, is not specific to our system, but rather a general problem with ownership inference in cases where complete list of classes is not available for analysis.

3.6 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 [26]). This approach maintains the safety of annotations as 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 that is commonly used by other pointer analysis algorithms [22, 34, 21]. 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, this allows us to treat each of the arrays as if they contain a single element. When it comes to generating variable flow information this allows us to represent all interactions with array elements as interactions with a single variable.

Figure 3.21 provides an example. 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

```
1  public class MyClass {
2      private String[] x = ...;
3      private String[] y = ...;
4
5      public void myMethod(){
6          for(int i = 0; i < x.length; i++){
7              y[i] = x[i];
8          }
9      }
10 }
```

Figure 3.22: Loop Example

seem to be quite conservative in this case, we can see that they serve to be a much better approximation when an operation is performed for each of the elements in an array. For example, consider Figure 3.22. 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 new 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. It is important to note that we will have to mark the array elements to be both read and write exposed no matter what the type of exposure is calculated for the base reference. A formal description of this rule will be given in the next chapter.

Chapter 4

Formalisation

This chapter provides a formalisation of the ownership inference algorithm. We will then use this formalisation to sketch a termination proof of the algorithm. In order to aid understanding we present a basic version of the algorithm that will infer ownership on a simplified Java-like language. There are three non-trivial parts to the algorithm:

1. **Class Flow Graph Construction** (`CreateCLFG`) — Creation of a mathematical model (directed graph) that shows *all possible* flows of values between variables (in this thesis we use the term "variables" to mean method parameters, fields, local variables, etc) inside a class.

2. **General Exposure Inference** (`ExposedNodes`) — Once we have obtained the flow graph for a class, we use a series of rules to determine which variables are to be considered to be "exposed". Variable exposure occurs if it is possible for a variable to contain values accessible by objects other than `this`. This stage takes into account only exposures that occur due to code inside the currently analysed class; the self-exposure of the variables is not considered.

3. **Self-Exposure Inference** (`ExposedClasses`) — The goal of this stage is to analyse all of the target classes and determine which ones potentially expose their `this` reference.

4. **Field Ownership Inference** (`OwnedFields`) — Given the set of exposed fields, ownership inference is trivial — if a field is not "exposed" then it is owned by the class given the field is not "self-exposed".

We now proceed to give a detailed formalisation for each of these parts.

4.1 Abstract Syntax

For our formalisation we use a simplified version of the Java language. The two main changes are:

1. **No branches in the control-flow graph:** Method bodies do not contain `if`, `else`, `for`, `while`, `switch` or other conditional statements.

2. **Flat class hierarchy:** This version of the language does not have interfaces or classes extending other classes.

Figure 4.1 shows the syntax of our simplified language. The top level declaration in our language definition is the `Program` — a definition of the whole program, which we represent as a list of class definitions. In turn, each of the class definitions consists of the name, a list of field definitions, and a list of method definitions. Since our simplified language avoids class hierarchies, fields and methods can only be `private` or `public`.

The definition of a method consists of two parts: the header containing type information and the name, followed by the body — simply a list of statements. Each of the statements can either be an assignment or a return. For simplicity we only allow assignment of the results of methods to local variables. There are 7 forms of expression in our language:

- $\texttt{new}_\eta$ `C`() or $\texttt{new}_\eta$ `C`[ ] - Creation sites for objects and arrays. Each of these expressions has a unique η number in order to make it easy to identify. In our implementation we can easily generate this unique identifier by considering file, line and column of the occurrence of the `new` expression.

- `this` - Same as in Java: a reference to the enclosing object.

- `l` - A local variable.

- $\texttt{p}_\texttt{i}$ - A variable referring to the ith formal parameter of the current method. For example, $\texttt{p}_1$ refers to the first, $\texttt{p}_2$ to the second, and so on. To simplify the presentation of our algorithm, our intermediate language does not allow reassignment of the value of the parameter inside the method.

- `v.f` - A field access.

- `e`[`k`] - Array element access.

We also make the following restrictions in order to simplify our formalisation:

$$
\begin{array}{rcl}
\texttt{Program}_{\texttt{def}} & ::= & \overline{\texttt{Class}_{\texttt{def}}} \\
\texttt{Class}_{\texttt{def}} & ::= & \texttt{class C}\ \{\overline{\texttt{Field}_{\texttt{def}}}\ \overline{\texttt{Method}_{\texttt{def}}}\} \\
\texttt{Field}_{\texttt{def}} & ::= & \texttt{Modifier}\ \mathbf{T}\ \texttt{f} \\
\texttt{Method}_{\texttt{def}} & ::= & \texttt{Modifier}\ \mathbf{T}\ \texttt{m}(\mathbf{T}\ \texttt{p}_1, \ldots, \mathbf{T}\ \texttt{p}_n)\ \{\overline{\texttt{Stmt}}\} \\
\texttt{p}_\texttt{i} & ::= & \text{method parameter \#i} \\
\texttt{Stmt} & ::= & \texttt{l} = \texttt{Invk} \mid \texttt{l} = \texttt{e} \mid \texttt{this.f} = \texttt{e} \mid \texttt{return e} \\
\texttt{Invk} & ::= & \texttt{v.m}(\overline{\texttt{e}}) \\
\texttt{e} & ::= & \texttt{new}_\eta\ \texttt{C}() \mid \texttt{new}_\eta\ \texttt{C}[\,] \mid \texttt{v} \mid \texttt{e}[\texttt{k}] \\
\texttt{v} & ::= & \texttt{this} \mid \texttt{l} \mid \texttt{p}_\texttt{i} \mid \texttt{v.f} \\
\texttt{Modifier} & ::= & \textit{private} \mid \textit{public} \\
\mathbf{T} & ::= & \text{types} \\
\texttt{C} & \in & \text{class names} \\
\texttt{m} & \in & \text{method names} \\
\texttt{f} & \in & \text{field names} \\
\texttt{l} & \in & \text{local variable names} \\
\texttt{k} & \in & \text{array indexes} \\
\eta & \in & \{0, 1, \ldots\}
\end{array}
$$

Figure 4.1: Intermediate Language Syntax

- Method invocations are not allowed as arguments. For example: $\texttt{x.m}(\texttt{y.m}())$ is not permitted.
- No two classes have a field or a method with the same name.
- Arrays cannot be elements of other arrays (no multidimensional arrays).

4.2 Class Flow Graph

As we have mentioned in Chapter 3, our inference algorithm relies on information about the flow of values in the target program. In this section we present a construct representing this information: The Class Flow Graph.

A Class Flow Graph (CLFG) is a directed graph where nodes correspond to individual variables and edges correspond to the potential flows of values. In our algorithms we will use $\texttt{nodes}(\texttt{G})$ to refer to the nodes of graph G and $\texttt{edges}(\texttt{G})$ to refer to its edges. We also assume that if we add an extra edge to the graph then its nodes will also be automatically added.

The definition of CLFG and its nodes is given in Figure 4.2. Here the notation $\texttt{new}_\eta$ is used to represent the values generated by a particular new expression. We use the unique identifying number η to easily distinguish the nodes for different new expressions. Fields of the current instance are represented by nodes of the form this.f, fields of all other instances are represented by nodes of the form other.f. Nodes corresponding to formal parameters of methods are of the

$$
\begin{array}{lcl}
\texttt{G} & :: & \{\texttt{N} \mapsto \texttt{N}\} \\
\texttt{N} & ::= & \texttt{new}_{\eta} \mid \texttt{this} \mid \texttt{this.G} \mid \texttt{other.G} \mid \texttt{N}[*] \\
\texttt{G} & ::= & \textit{field\_name} \mid \textit{method\_name}\#\alpha \\
\alpha & \in & \texttt{ret} \cup \{0, 1, ...\}
\end{array}
$$

Figure 4.2: CLFG Definition

```
1 class Foo{
2    public void method(Object a, Object b){
3         ...
4    }
5 }
```

Figure 4.3: CLFG nodes for method parameters. In CLFG, parameter `a` would be represented by node `this.method#0` and parameter `b` would be represented by node `this.method#1`

forms `this.m#i` or `other.m#i` (where `i` is the number of the parameter, see Figure 4.3). The nodes for the return values of the methods are similar, except that they contain `ret` instead of a parameter number.

As we have mentioned in the previous chapter our analysis ignores the indices and sizes of arrays, effectively treating each array as if it only had one element. If `N` is a variable referring to the base of an array, then we use `N`$[*]$ to represent the elements of `N`. We refer to `N` itself as the *base node* and `N`$[*]$ as the *element node*.

Nodes of the CLFG loosely correspond to the variables of the program. There is however an important difference: we do not have separate nodes for variables or fields that belong to objects other than `this`. For example, consider the code in Figure 4.4. The graphical representation of CLFG for `MyClass` is given in Figure 4.5. It contain the following nodes: `this.a`, `this.b`, `other.f`. Notice that node `other.f` represents *both* `x.f` and `y.f`. This results in both `c`$[0]$ and `c`$[4]$ being represented by the single array element node `this.c`$[*]$.

4.2.1 CLFG Construction Algorithm

Now that we have described the structure of the CLFG we will present a simple algorithm that constructs the CLFG for a given class. The top level structure of the algorithm is to first analyse each of the methods in the target class, and then combine the results to create the CLFG for the whole class. The analysis of each method results in a partial flow graph. To combine particular graphs into the preliminary CLFG we simply use a graph union operation. We then proceed to augment the preliminary CLFG with information about the flow between the

```
1  class Foo{
2     private String a;
3     private String b;
4     private String[] c = ..;
5
6     public void method(Bar x, Bar y){
7         this.a = x.f;
8         this.b = y.f;
9
10        this.c[0] = a;
11        this.c[4] = b;
12    }
13 }
```

Figure 4.4: Example class. The CFLG generated for this class is given in Figure 4.5.

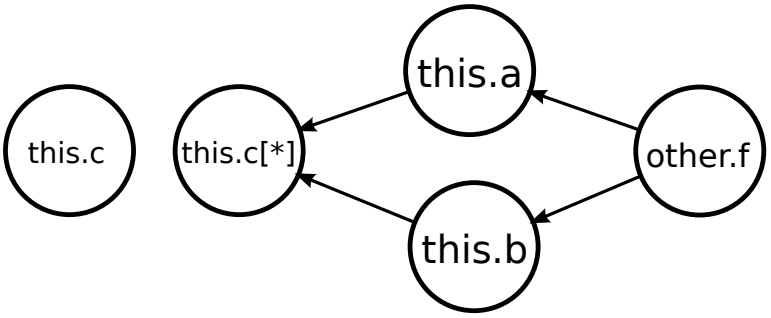


Figure 4.5: CLFG for the class given in Figure 4.4. Notice that `this.c[0]` and `this.c[4]` as well as `x.f` and `y.f` are represented by shared nodes.

element nodes of the different arrays in order to obtain the complete CLFG, the pseudo code for the top level algorithm is presented in Figures 4.6, 4.7 and 4.8.

In order to compute the effect of a particular method on the CLFG, we analyse each of the statements in sequence. As the statements are analysed we maintain the local variable mapping information Γ - an environment that maps local variables to the non-local variables that they currently alias. For each of the statements we use the `cf` function (Figure 4.7) to see what effect the statement has on the CLFG (`G`) and the local variable environment Γ.

The `cf` function (Figure 4.8) identifies the local (inside the current method) value flows that result from the execution of a given statement. For example, an assignment statement of the form `A = B` results in a single flow from `B` to `A`, while a method invocation results in flows from all of the arguments to the corresponding formal parameters of the method. The `node(..)` function is used to identify CLFG nodes corresponding to the expressions in the language.

Before we proceed to describe the array element flow generation function we

Input:
$\texttt{class C}\ \{\texttt{fields}_\texttt{C}\ \texttt{methods}_\texttt{C}\}$ - Definition of class C

Output:
$\texttt{G}_\texttt{C}$ - CLFG for class C

[1] $\texttt{G}_\texttt{C} = \{\}$

[2] **for all** $\texttt{f}_\texttt{i} \in \texttt{fields}_\texttt{C}$ **do**
[3] $\quad \texttt{nodes}(\texttt{G}_\texttt{C}) = \texttt{nodes}(\texttt{G}_\texttt{C}) \cup \{\texttt{node}(\texttt{f}_\texttt{i})\}$

[4] **for all** $\texttt{Modifier}_\texttt{i}\ \texttt{T}_\texttt{i}\ \texttt{m}_\texttt{i}(..)\{\texttt{statements}_\texttt{i}\} \in \texttt{methods}_\texttt{C}$ **do**
[5] $\quad \Gamma = \{\}$
[6] $\quad$ **for all** $\texttt{stmt}_\texttt{j} \in \texttt{statements}_\texttt{i}$ **do**
[7] $\quad\quad (\texttt{G}'_\texttt{C}, \Gamma) = \texttt{cf}(\texttt{stmt}_\texttt{j}, \Gamma)$
[8] $\quad\quad \texttt{G}_\texttt{C} = \texttt{G}_\texttt{C} \cup \texttt{G}'_\texttt{C}$

[9] $\texttt{G}_\texttt{C} = \texttt{AugmentArrayFlow}(\texttt{Def}_\texttt{C}, \texttt{G}_\texttt{C})$

[10] $\Rightarrow$ **output** $\texttt{G}_\texttt{C}$

Figure 4.6: CLFG creation algorithm. The definitions of helper functions are given in Figures 4.7 and 4.8. Since the parameters cannot be reassigned inside the body of the method, the initial environment G is simply an empty mapping.

cf generation rules

$$
\begin{array}{rcl}
\texttt{cf}(\texttt{l}_\texttt{i} = \texttt{Invk}_\texttt{i}, \Gamma, \texttt{m}_\texttt{cur}) & := & (\texttt{arg\_map}(\texttt{Invk}_\texttt{i}), \Gamma \triangleleft \texttt{l}_\texttt{i} \mapsto \texttt{node}(\texttt{Invk}_\texttt{i})) \\
\texttt{cf}(\texttt{l}_\texttt{i} = \texttt{e}_\texttt{i}, \Gamma, \texttt{m}_\texttt{cur}) & := & (\{\}, \Gamma \triangleleft \texttt{l}_\texttt{i} \mapsto \texttt{node}(\texttt{e}_\texttt{i})) \\
\texttt{cf}(\texttt{this.f}_\texttt{i} = \texttt{e}_\texttt{i}, \Gamma, \texttt{m}_\texttt{cur}) & := & (\{\texttt{node}(\texttt{e}_\texttt{i}) \mapsto \texttt{node}(\texttt{this.f}_\texttt{i})\}, \Gamma) \\
\texttt{cf}(\texttt{return e}_\texttt{i}, \Gamma, \texttt{m}_\texttt{cur}) & := & (\{\texttt{node}(\texttt{e}_\texttt{i}) \mapsto \texttt{this.m}_\texttt{cur}\#\texttt{ret}\}, \Gamma) \\
\texttt{arg\_map}(\texttt{this.m}_\texttt{i}(\texttt{e}_1, .., \texttt{e}_\texttt{n}), \Gamma) & := & \{\texttt{node}(\texttt{e}_\texttt{j}) \mapsto \texttt{node}(\texttt{this.m}_\texttt{i}\#\texttt{j}) \mid 0 < \texttt{j} < \texttt{n}\} \\
\texttt{arg\_map}(\texttt{v}_\texttt{i}\texttt{.m}_\texttt{i}(\texttt{e}_1, .., \texttt{e}_\texttt{n}), \Gamma) & := & \{\texttt{node}(\texttt{e}_\texttt{j}) \mapsto \texttt{node}(\texttt{other.m}_\texttt{i}\#\texttt{j}) \mid 0 < \texttt{j} < \texttt{n}\}
\end{array}
$$

Figure 4.7: Statement Flow Effects. The symbol $\triangleleft$ is used to indicate an overwriting update to the environment, for example: $\{\texttt{a} \mapsto \texttt{x}\} \triangleleft (\texttt{a} \mapsto \texttt{y}) = \{\texttt{a} \mapsto \texttt{y}\}$. The helper function arg_map is used to create flows from the arguments of a method invocation to the formal parameters of the method.

$$
\begin{array}{rcl}
\texttt{node} & :: & (\texttt{e}, \texttt{m}, \Gamma) \rightarrow \texttt{N} \\
\texttt{node}(\texttt{new}_\eta\ \texttt{C}(), \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{new}_\eta \\
\texttt{node}(\texttt{new}_\eta\ \texttt{C}[\,], \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{new}_\eta \\
\texttt{node}(\texttt{this}, \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{this} \\
\texttt{node}(\texttt{p}_\texttt{i}, \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{this.m}_{\texttt{cur}}\#\texttt{i} \\
\texttt{node}(\texttt{l}_\texttt{i}, \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \Gamma(\texttt{l}_\texttt{i}) \\
\texttt{node}(\texttt{this.fi}, \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{this.f}_\texttt{i} \\
\texttt{node}(\texttt{v}_\texttt{i}\texttt{.fi}, \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{other.f}_\texttt{i} \\
\texttt{node}(\texttt{this.m}_\texttt{i}(\overline{\texttt{e}}), \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{this.m}_\texttt{i}\#\texttt{ret} \\
\texttt{node}(\texttt{v}_\texttt{i}\texttt{.m}_\texttt{i}(\overline{\texttt{e}}), \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{other.m}_\texttt{i}\#\texttt{ret} \\
\texttt{node}(\texttt{e}_\texttt{i}[\texttt{k}_\texttt{i}], \texttt{m}_{\texttt{cur}}, \Gamma) & ::= & \texttt{node}(\texttt{e}_\texttt{i})[*]
\end{array}
$$

Figure 4.8: Node Creation Function

will present an example of CLFG generation for a small class that does not contain array variables. Class `C` shown in Figure 4.9 contains four fields (`a`, `b`, `c`, `d`) and five methods. Figure 4.10 shows the resulting CLFG where `this.a` flows into the `getA` method, and `this.c` flows into the `setA` method which in turn flows into `this.a`. Similarly, `this.b` has a flow from `f` and `this.d` has a flow from `Foo`.

We now consider the problem of array element flows. Figure 4.11 shows a simple class with two array fields, a method that carries out an assignment between them and a method that transfers the value of field `f` to one of the elements of `a`. Figure 4.12 shows the preliminary CLFG generated before the implicit flow between array elements is added. Consider the effects of the assignment $\texttt{b} = \texttt{a}$. Once the assignment has been completed, accessing elements of `a` will access the exact same elements as accessing the elements of `b`. In our example this means that there must be an edge from the node `this.f` to the node $\texttt{this.b}[*]$.

In order to infer the implicit value flows between the array elements we provide the array augmentation function, which works by looking at all of the edges in the preliminary CLFG. If the edge happens to be between two variables that have array types, it will find the corresponding element nodes and add a bi-directional edge between them. The bi-directional property comes from the fact that both array element nodes are now potentially representing the same set of elements. For instance, in our example class `A` a value flow into elements of `this.a` or elements of `this.b` may potentially affect the elements of the other field. Figure 4.13 has a dashed edge representing the new flows added by the augmentation algorithm.

We can now clearly see that values from `this.f` may potentially flow to the elements of both `a` and `b`. The predicate `IsArrayBase` returns true if the given node is the base reference of an array, or false otherwise based on the type information. The formal description of the augmentation function is given in Figure 4.14.

```
1  class C {
2      private Foo a;
3      private Foo b;
4      private Foo c;
5      private Foo d;
6
7      public Foo getA(){
8          return this.a;
9      }
10
11     public void setA(String par){
12         this.a = par;
13     }
14
15     public void m1(Bar x){
16         this.b = x.f;
17     }
18
19     public void m2(){
20         this.setA(this.c);
21     }
22
23     public void m3(){
24         this.d = new Foo();
25     }
26 }
```

Figure 4.9: Example Class

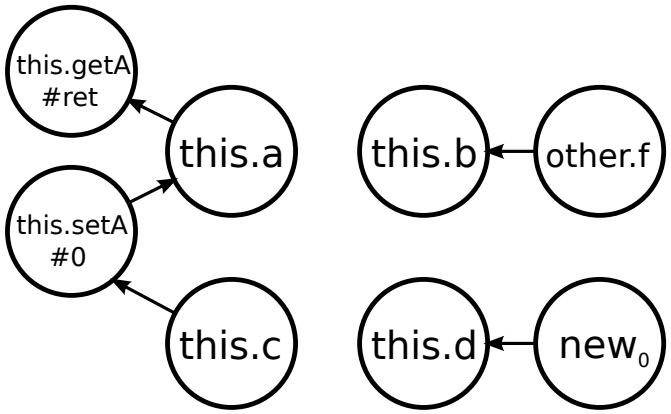


Figure 4.10: Example CLFG

```
1  class A {
2      private Foo a[5];
3      private Foo b[5];
4
5      private Foo f;
6
7      public void m1(){
8          this.b = this.a;
9      }
10
11     public void m2(){
12         this.a[3] = this.f;
13     }
14 }
```

Figure 4.11: Array Assignment Example

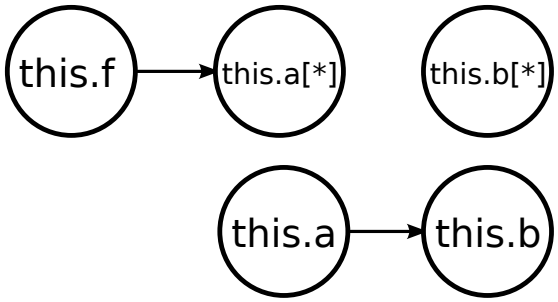


Figure 4.12: Preliminary CLFG for class A

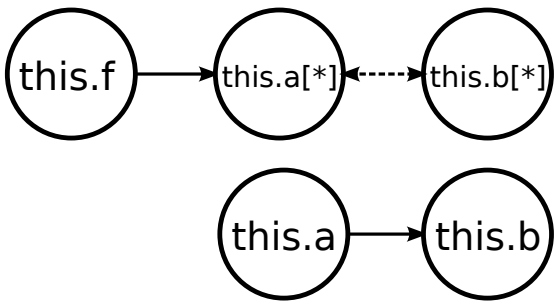


Figure 4.13: Augmented CLFG for class A

Input:
$\mathtt{G_C}$ - Preliminary CLFG for class `C`

Output:
$\mathtt{G'_C}$ - Complete CLFG for class `C`

[1] $\mathtt{G'_C} = \mathtt{G_C}$

[2] **for all** $\mathtt{N_a} \mapsto \mathtt{N_b} \in \mathtt{edges}(\mathtt{G_C})$ **do**
[3] **if** $\mathtt{IsArrayBase}(\mathtt{N_a})$ **then**
[4] $\mathtt{edges}(\mathtt{G'_C}) = \mathtt{edges}(\mathtt{G'_C}) \cup \{\mathtt{N_a}[*] \mapsto \mathtt{N_b}[*]\}$
[5] $\mathtt{edges}(\mathtt{G'_C}) = \mathtt{edges}(\mathtt{G'_C}) \cup \{\mathtt{N_b}[*] \mapsto \mathtt{N_a}[*]\}$

[6] $\Rightarrow$ **output** $\mathtt{G'_C}$

Figure 4.14: Array Augmentation Function

4.3 Variable Exposure

Given a flow graph for a particular class, we perform the exposure propagation algorithm in order to determine how variables inside a class may have their values exposed to the outside.

First we define the concept of exposure. We say that a node in the class flow graph `G` is *exposed* if it may have aliases outside of the containing object. As we have mentioned before in Section 3.3.2, there are two types of exposure:

1. **Read Exposure** - A node is *read exposed* if its value may be read by external objects (and therefore assumed to be aliased).

2. **Write Exposure** - A node is *write exposed* if a previously externally referenced value may flow into it.

Each node can have one or both types of exposure. Consider the example class in Figure 4.15. Here, field `a` has write exposure because an external value can be passed into it through the parameter of method `setA`. However, if we put some value into `a`, this value will not be exposed to the outside, as values of `a` do not propagate to external objects (hence `a` is not read exposed). The case is reversed for field `b`: any value we put into `b` may potentially leak to external objects through method `getB`, but `b` itself will never contain externally generated values (hence `b` is read exposed, but not write exposed). The field `c` is public, and hence external objects can both potentially read and write to it and is therefore has both read and write exposure. Field `d` does not have either type of exposure, as it is private and is never used.

```
1  class C {
2      private String a;
3      private String b;
4      public String c;
5      private String d;
6
7      public void setA(String p){
8          this.a = p;
9      }
10
11     public String getB(){
12         return this.b;
13     }
14 }
```

Figure 4.15: Read and Write Exposure Examples

In order to keep track of exposure types of various nodes, our algorithm maintains a mapping called Δ_C (where C is the name of the currently analysed class). This mapping has a (possibly empty) set of exposure types for each node in the currently analysed class. There are two possible attributes a node can have: WRITE (write exposure) or READ (read exposure). The definition is given in Figure 4.16.

$$\begin{aligned} \Delta_C &:: \{N \mapsto \texttt{ExposureType}\} \\ \texttt{ExposureType} &::= \texttt{READ} \mid \texttt{WRITE} \end{aligned}$$

Figure 4.16: Definition of Exposure Mapping Δ_C

4.4 General Exposure Inference

This section presents the algorithm for finding complete variable exposures. This algorithm is then reused in both self-exposure and ownership inference analyses.

In order to obtain the complete exposure mapping Δ_C we require the definition of the class in question as well as the previously calculated CLFG. We start with an empty Δ_C and proceed to repeatedly apply a series of inference rules. Each rule application adds new exposure information to Δ_C until there are no more applicable rules. At this point, Δ_C is complete.

The first set of rules are the *initialisation rules*. These rules identify nodes that have certain types of exposure independent of the value flow or exposure of other

nodes. Examples of such "inherently" exposed nodes include public fields, parameters of public methods, or return values of public methods.

After the initial set of variable exposures have been established we proceed to repeatedly apply *propagation rules*. These rules look at nodes that already have exposure types, and see how this affects the exposure of other nodes. For example, if node `A` has `READ` exposure and there is a value flow `B` $\rightarrow$ `A`, then `B` also has `READ` exposure.

The rest of this section will describe these rules in greater detail, and the rule application algorithm will be given at the end.

4.4.1 Initialisation rules

The set of initialisation rules is used to identify CLFG nodes that have various exposures simply by virtue of their declaration (e.g. a public field is always both read and write exposed as previously discussed in Section 3.4.1). The formal description of the rules and their helper predicate is provided in Figure 4.17. We use the `isPrivate` predicate to determine if the node in question is either a private field, parameter of a private method or a return value of a private method.

The first three rules (**I-PField**, **I-PPar** and **I-PRet**) address the exposure types of public fields, method parameters and method return values. Public method parameters always have `WRITE` exposure as external classes can invoke public methods with any set of arguments. Similarly, the return values of public methods can always be potentially captured, giving them `READ` exposures. Public fields have both types of exposure.

The next three rules (**I-OField**, **I-OPar** and **I-ORet**) are very similar, except they deal with fields and methods of other (i.e. not `this`) objects. Because all of the exposure attributes are assigned from the perspective of `this` instance, the exposures types are inverted. For example, we consider nodes of the form `other.m#i` (parameter of a method belonging to another instance) to have `READ` exposure (**I-OPar**). This is because if we pass a value as a parameter to an unknown method, we must conservatively assume that this value will become exposed. For a similar reason the return value of a method of another instance contains externally generated values and is therefore a `WRITE` node (**I-ORet**). Fields of external instances are both `WRITE` and `READ` as before (**I-OField**).

4.4.2 Propagation rules

The propagation rules are used to determine the effect of initially exposed nodes on the exposure types of all the other variables inside a class (previously dis-

Initialisation rules

$$\frac{\neg\texttt{isPrivate}(\texttt{this.f})}{\texttt{this.f} \mapsto \texttt{READ}, \texttt{this.f} \mapsto \texttt{WRITE} \in \Delta_c} \quad \text{[I-PField]}$$

$$\frac{\neg\texttt{isPrivate}(\texttt{this.m\#i})}{\texttt{this.m\#i} \mapsto \texttt{WRITE} \in \Delta_c} \quad \text{[I-PPar]}$$

$$\frac{\neg\texttt{isPrivate}(\texttt{this.m\#ret})}{\texttt{this.m\#ret} \mapsto \texttt{READ} \in \Delta_c} \quad \text{[I-PRet]}$$

$$\frac{}{\texttt{other.f} \mapsto \texttt{READ}, \texttt{other.f} \mapsto \texttt{WRITE} \in \Delta_c} \quad \text{[I-OField]}$$

$$\frac{}{\texttt{other.m\#i} \mapsto \texttt{READ} \in \Delta_c} \quad \text{[I-OPar]}$$

$$\frac{}{\texttt{other.m\#ret} \mapsto \texttt{WRITE} \in \Delta_c} \quad \text{[I-ORet]}$$

Figure 4.17: Initialisation rules

cussed in Section 3.4.2). Both initialisation and propagation rules follow the same form, however there are a few important differences between them.

The difference is that unlike the initialisation rules, which consider a single node at a time and have no use for the CLFG, all of the propagation rules operate on pairs of nodes that usually have some sort of value flow between them.

The propagation rules are presented in Figure 4.18. The definition of the helper function `ReflectToThis` can be found in Figure 4.19.

P-WriteFlow - The first rule says that if there is a node $\texttt{N}_\texttt{a}$ whose values may contain externally generated (`WRITE`) values, and there is a flow to $\texttt{N}_\texttt{b}$, then $\texttt{N}_\texttt{b}$ is also `WRITE` exposed. This comes from the transitive nature of the value flow: if there is a flow $\texttt{N}_\texttt{e} \mapsto \texttt{N}_\texttt{a}$ and $\texttt{N}_\texttt{a} \mapsto \texttt{N}_\texttt{b}$, then there is a flow of values $\texttt{N}_\texttt{e} \mapsto \texttt{N}_\texttt{b}$.

P-ReadFlow - Similar to the previous rule, except for a `READ` exposed node. Note that the direction of the flow is different; if there is a value flow $\texttt{N}_\texttt{a} \mapsto \texttt{N}_\texttt{b}$, and $\texttt{N}_\texttt{b}$ has its values leaked to the outside, then so does $\texttt{N}_\texttt{a}$.

P-ReadToWrite - This rule is similar to **P-WriteFlow**, except the initially exposed node $\texttt{N}_\texttt{a}$ has `READ` exposure. Like before, the second node $\texttt{N}_\texttt{b}$ will become `WRITE` exposed. This is best illustrated with the example in Figure 4.20.

P-OtherWriteToRead - In some cases when dealing with other instances of the same class, the exposure of fields can occur indirectly. Consider Figure 4.21. The problem is that even though the return values of `getA` are used externally, the node `this.getA#ret` itself does not gain `READ` exposure from any of the previous

Basic Propagation rules

$$\frac{\mathtt{N_a} \mapsto \mathtt{WRITE} \in \Delta_C \quad \mathtt{N_a} \mapsto \mathtt{N_b} \in G_C}{\mathtt{N_b} \mapsto \mathtt{WRITE} \in \Delta_C} \quad \text{[P-WriteFlow]}$$

$$\frac{\mathtt{N_b} \mapsto \mathtt{READ} \in \Delta_C \quad \mathtt{N_a} \mapsto \mathtt{N_b} \in G_C}{\mathtt{N_a} \mapsto \mathtt{READ} \in \Delta_C} \quad \text{[P-ReadFlow]}$$

$$\frac{\mathtt{N_a} \mapsto \mathtt{READ} \in \Delta_C \quad \mathtt{N_a} \mapsto \mathtt{N_b} \in G_C}{\mathtt{N_b} \mapsto \mathtt{WRITE} \in \Delta_C} \quad \text{[P-ReadToWrite]}$$

$$\frac{\mathtt{N_a} \mapsto \mathtt{WRITE} \in \Delta_C \quad \mathtt{N_b} = \mathtt{ReflectToThis}(\mathtt{N_a})}{\mathtt{N_b} \mapsto \mathtt{READ} \in \Delta_C} \quad \text{[P-OtherWriteToRead]}$$

$$\frac{\mathtt{N_a} \mapsto \mathtt{READ} \in \Delta_C \quad \mathtt{N_b} = \mathtt{ReflectToThis}(\mathtt{N_a})}{\mathtt{N_b} \mapsto \mathtt{WRITE} \in \Delta_C} \quad \text{[P-OtherReadToWrite]}$$

$$\frac{\mathtt{N_{ele}} \text{ is an element node of } \mathtt{N_{base}} \quad \mathtt{N_{base}} \mapsto \mathtt{READ} \in \Delta_C}{\{\mathtt{N_{ele}} \mapsto \mathtt{READ}, \mathtt{N_{ele}} \mapsto \mathtt{WRITE}\} \in \Delta_C} \quad \text{[P-ArrayBaseRead]}$$

$$\frac{\mathtt{N_{ele}} \text{ is an element node of } \mathtt{N_{base}} \quad \mathtt{N_{base}} \mapsto \mathtt{WRITE} \in \Delta_C}{\{\mathtt{N_{ele}} \mapsto \mathtt{READ}, \mathtt{N_{ele}} \mapsto \mathtt{WRITE}\} \in \Delta_C} \quad \text{[P-ArrayBaseWrite]}$$

Figure 4.18: Propagation rules

ReflectToThis

$\mathtt{ReflectToThis} \;::\; \mathtt{N} \rightarrow \mathtt{N}$

$$\frac{\mathtt{SameClass}(\mathtt{other.f})}{\mathtt{ReflectToThis}(\mathtt{other.f}) = \mathtt{this.f}}$$

$$\frac{\mathtt{SameClass}(\mathtt{other.m\#i})}{\mathtt{ReflectToThis}(\mathtt{other.m\#i}) = \mathtt{this.m\#i}}$$

$$\frac{\mathtt{SameClass}(\mathtt{other.m\#ret})}{\mathtt{ReflectToThis}(\mathtt{other.m\#ret}) = \mathtt{this.m\#ret}}$$

Figure 4.19: `ReflectToThis` Helper Function. Function converts all of the nodes of form `other.X` into the form `this.X`. The function $\mathtt{SameClass}(\mathtt{N_i})$ is used to ensure that the node $\mathtt{N_i}$ belongs to the same class as `this` (we omit the definition).

```
1  class MyClass {
2      private Object a;
3      private Object b;
4
5      public Object getA(){
6          // Field a is exposed
7          return this.a;
8      }
9
10     public void method(){
11         // potentially exposed value is written into b
12         this.b = this.a;
13     }
14 }
```

Figure 4.20: **P-ReadToWrite** Example. All of the values stored in `a` can be potentially aliased by external objects. If we were to write any of those values into `b`, then `b` would become exposed as well. We choose to make `b` `WRITE` exposed, because writing other values into it will not cause them to become exposed. Note that in our case the `WRITE` exposure does not mean that external nodes can modify the values of `b`.

rules. When the return value of `getA` is used in `myMethod`, the node is identified as `other.getA#ret` instead of `this.getA#ret`. Since the two nodes do not have an explicit flow between them, we rely on the **P-OtherWriteToRead** rule to transfer the `WRITE` exposure of `other.getA#ret` into the `READ` exposure of `this.getA#ret`.

P-OtherReadToWrite - This rule is very similar to the previous one, the only difference is that exposure types are reversed. Consider example in Figure 4.22.

P-ArrayBaseRead and **P-ArrayBaseWrite** - These rules describe the situation in which the base of the array has been either `READ` or `WRITE` exposed. In either case this means that an external object can potentially have a reference to the array. This means that the external object is free to do both read and write operations on any elements of the arrays using the base reference. For example, in Figure 4.23 we can see that even though the reference to array `myArray` is only `READ` exposed, the elements of `myArray` have both types of exposure. Similarly, a `WRITE` exposure of the base reference exposes the elements to both reading and writing.

4.4.3 Rule application algorithm

Our inference algorithm consists of an initialisation step and an iterative propagation step. The initialisation step determines the initial Δ_C, which contains

```
1class MyClass {
2    private Object a;
3
4    private Object getA(){
5        return this.a;
6    }
7
8    public Object myMethod(MyClass other){
9        Object tmp = other.getA();
10
11       // tmp is read exposed
12       return tmp;
13   }
14}
```

Figure 4.21: **P-OtherWriteToRead** Example. `myMethod` exposes the return value of `getA`, which indirectly exposes the value of field `a`.

```
1class MyClass{
2    private Object a;
3
4    private void setA(Object p){
5        this.a = p;
6    }
7
8    public void myMethod(MyClass other, Object x){
9        other.setA(x);
10   }
11}
```

Figure 4.22: **P-OtherReadToWrite** Example. `myMethod` passes an externally provided value of `x` as an argument to a private method `setA` of another instance.

```
1class MyClass {
2    private Object myArray[] = ...;
3
4    public Object[] getArray(){
5        return myArray;
6    }
7}
8
9class ExternalClass {
10
11    private Object extA, extB;
12
13    public void method(MyClass c){
14        // Reading a value of myArray
15        this.extA = c.getArray()[0];
16
17        // Writing a value to myArray
18        c.getArray()[1] = this.extB;
19    }
20}
```

Figure 4.23: Array Base Exposure

exposure attributes for all inherently exposed nodes (such as public fields, parameters to public methods, etc). The propagation step takes the current Δ_C and information about flow of values G_C to determine which other nodes may contain externally accessible values. Δ_C is then updated with the new results.

Both steps of the inference are performed by application of the inference rules on Δ_C. Each of the rules takes in the information required for inference and returns a set of extra attribute mappings (for example $\{a \mapsto \texttt{READ}, a \mapsto \texttt{WRITE}\}$) which is then added to the existing Δ_C. If the rule is not applicable because its precondition is not satisfied, it simply returns an empty set $\{\}$.

The algorithm tries to apply every initialisation rule to every single node once and then repeatedly tries to apply every propagation rule to every possible pair of nodes.

The algorithm terminates when no further rule applications can add any additional exposures to Δ_C (i.e. it reaches a fix point). Since exposure types are never removed, Δ_C much reach fix point and our algorithm will terminate.

We note that the actual implementation of this algorithm in OwnKit uses a much more efficient work list approach. However both versions use the same rules and will produce the same results, so in the interests of a simpler explanation we will use the unoptimised version.

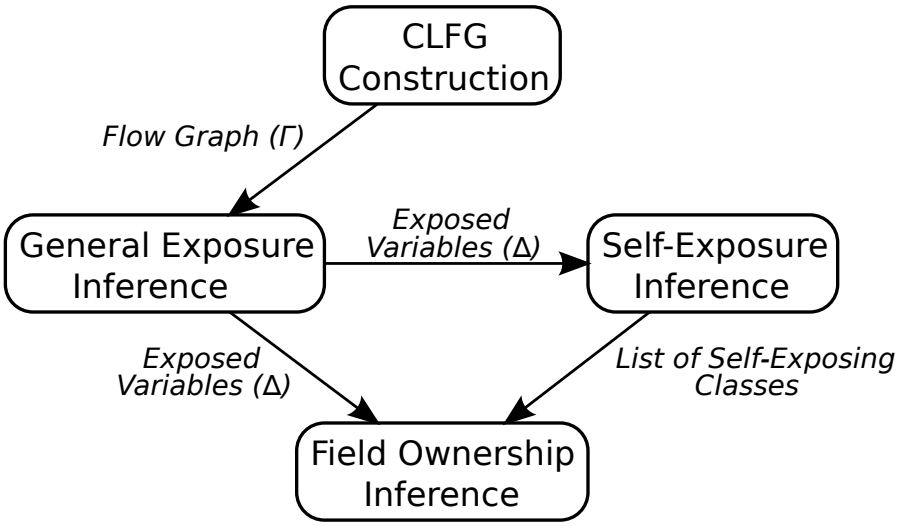


Figure 4.24: Dependencies of sub-algorithms

4.5 Self-Exposure and Field Ownership Inference

We have previously described the two most important parts of our inference algorithm: the CLFG generation step that calculates the value flow inside a class, and the General Exposure Inference step that calculates the exposure of all the variables inside a class. Figure 4.24 shows how these two stages interact with the other parts of the algorithm. Note that General Exposure Inference provides the exposure information to both Self-Exposure and Field Ownership, but does not depend on them.

4.5.1 Self-Exposure

The task of this stage is to determine a set of self-exposed classes out of all the classes in the target program. To do so we compute the complete exposure mapping Δ_C for each of the variables in each of the classes using the algorithm and rules from the previous stage. To know if the class is self-exposing we simply extract the exposure information of the `this` node. A pseudo code description is given in Figure 4.25. Note that we only have to check if `this` is `READ` exposed, as it is impossible to assign a value to it.

4.5.2 Field Ownership

The final stage of the algorithm is to infer the ownership of the actual fields for each of the classes. The approach is very similar to self-exposure inference, except that as well as having a complete exposure mapping Δ_C for each of the classes

Input:

$\overline{\texttt{C}_\texttt{i}}$ - All of the classes in the target program

$\overline{\Delta_\texttt{C}}$ - Corresponding exposure mappings

Output:

$\overline{\texttt{C}_{\texttt{self-exp}}}$ - All of the self-exposing classes

[1] $\overline{\texttt{C}_{\texttt{self-exp}}} = \{\}$

[2] **for all** $\texttt{C}_\texttt{i} \in \overline{\texttt{C}_\texttt{i}}$ **do**

[3] **if** $\texttt{this} \mapsto \texttt{READ} \in \Delta_\texttt{i}$ **then**

[4] $\overline{\texttt{C}_{\texttt{self-exp}}} = \overline{\texttt{C}_{\texttt{self-exp}}} \cup \{\texttt{C}_\texttt{i}\}$

[5] $\Rightarrow$ **output** $\overline{\texttt{C}_{\texttt{self-exp}}}$

Figure 4.25: Self-Exposure Inference Algorithm

we also have a previously calculated set of self-exposed classes. In order for a field to be owned it must not have either type of exposure. The class of the field ($\texttt{ClassOf}(..)$ predicate) must also not belong to the set of self-exposed classes. The pseudo code description of this can be found in Figure 4.26.

4.6 Correctness

In this section we will discuss the two main correctness properties of our inference algorithm: termination and ownership guarantee.

4.6.1 Termination

In order to see why our inference algorithm terminates, we need to show the termination of the four main components shown in Figure 4.24.

CLFG Construction

There are two stages to this part: the construction of a preliminary CLFG, and addition of array element flow information. The construction of a preliminary CLFG (Figure 4.6) simply traverses all of the fields and statements inside the class. Since it does not modify the underlying class, the termination is obvious. The augmentation algorithm (Figure 4.14) iterates over all of the edges in the preliminary CLFG ($\texttt{G}_\texttt{C}$). The $\texttt{G}_\texttt{C}$ is finite and is never modified, therefore providing us with a guarantee of termination.

Input:

$\texttt{class C}\ \{\texttt{fields}_\texttt{C}\ \texttt{methods}_\texttt{C}\}$	-	Definition of class C
$\Delta_\texttt{C}$	-	Corresponding exposure mapping
$\overline{\texttt{C}_{\texttt{self-exp}}}$	-	Self-exposed classes

Output:

$\overline{\texttt{f}_{\texttt{owned}}}$	-	All of the owned fields

[1] $\overline{\texttt{f}_{\texttt{owned}}} = \{\}$

[2] **for all** $\texttt{f}_\texttt{i} \in \texttt{fields}_\texttt{C}$ **do**
[3] **if** $\neg(\texttt{this.f} \mapsto \texttt{READ} \in \Delta_\texttt{C})$ **then**
[4] $\overline{\texttt{f}_{\texttt{owned}}} = \overline{\texttt{f}_{\texttt{owned}}} \cup \{\texttt{f}\}$
[5] **else if** $\neg(\texttt{this.f} \mapsto \texttt{WRITE} \in \Delta_\texttt{C})$ **then**
[6] $\overline{\texttt{f}_{\texttt{owned}}} = \overline{\texttt{f}_{\texttt{owned}}} \cup \{\texttt{f}\}$
[7] **else if** $\neg(\texttt{ClassOf}(\texttt{f}) \in \overline{\texttt{C}_{\texttt{self-exp}}})$ **then**
[8] $\overline{\texttt{f}_{\texttt{owned}}} = \overline{\texttt{f}_{\texttt{owned}}} \cup \{\texttt{f}\}$

[9] $\Rightarrow$ **output** $\overline{\texttt{f}_{\texttt{owned}}}$

Figure 4.26: Field Ownership Inference Algorithm

Self-Exposure Inference and Field Ownership Inference

The pseudo code for these algorithms is given in Figures 4.25 and 4.26. As we can see both of the algorithms simply iterate over collections of fixed sizes that are not being modified, and hence are always guaranteed to terminate.

General Exposure Inference

As before, we will consider the termination of this stage by considering the termination of its two parts: the initialisation step and the propagation step. The termination of the initialisation stage is trivial. We simply try to apply all of the rules from Figure 4.17 to all of the nodes in the CLFG for the current class (the CLFG is not being modified by this process).

During the propagation step, we repeatedly try to apply all of the propagation rules (Figure 4.18). Each of the rules modifies the exposure mapping $\Delta_\texttt{C}$, but does not modify the CLFG. This stage terminates when no further applications of the rules can result in a change of $\Delta_\texttt{C}$.

In order to show the termination of this stage we need to show that $\Delta_\texttt{C}$ has a *least fixed point*. This amounts to showing the following two properties:

1. The sub-typing relation between exposure mappings is a *join-semi-lattice*.

2. The transfer function (in our case, the propagation rules) is *monotonic* with respect to the sub-typing relation.

We can treat our exposure mappings as a set of pairs ($\texttt{node} \mapsto \texttt{exposure}$). This allows us to define the sub-typing relation as a normal subset operation. For example, $\{\texttt{a} \mapsto \texttt{READ}\}$ is a sub-type of $\{\texttt{a} \mapsto \texttt{READ}, \texttt{a} \mapsto \texttt{WRITE}\}$.

The least lower bound between two exposure mappings is the union of pairs. The semi-lattice is also bounded; the greatest element is an exposure mapping where each of the nodes is both `READ` and `WRITE` exposed.

For the second property we need to show that all of our propagation rules are monotonic with relation to the sub-typing relation. In terms of our exposure mappings this means that if a rule starts with an initial Δ_C and produces Δ'_C, then Δ_C has to be a subtype of Δ'_C. Since our sub-typing relation is simply a subset relation, this means that Δ_C has to be a subset of Δ'_C. As we have mentioned before, if we consider the propagation rules in Figure 4.18, we can see that all of them only add extra elements to Δ_C and never remove them, hence satisfying monotonicity.

4.6.2 Ownership Guarantees

The *ownership guarantee theorem* guarantees that inferred ownership properties are correct. It can be informally stated as follows:

If our inference algorithm determines that a field `f` *is owned, then at no point during the execution of the program will* `f` *have external aliases.*

Ideally, we would provide a more formal statement of this theorem and include a rigorous proof of its correctness. However, to do this we would first need to provide operational semantics for our intermediate language — a complicated task that is beyond the scope of this project.

Chapter 5

Case Studies

In this chapter we present the results of a corpus study used to evaluate the performance and accuracy of OwnKit. We use our tool to infer field ownership in a number of open-source programs, including the Java Standard Library. In order to gain a better perspective on the results we then analyse how our tool performs against UNO [26].

5.1 Methodology

In our experiment we have used both tools to infer field ownership in the selected set of benchmarks. In order to compare the performance of the tools, the runtimes for individual benchmarks were also recorded. Since UNO uses the whole program analysis approach and OwnKit uses the modular approach, the general expectation is that UNO will find more owned fields, while OwnKit would have better run times.

5.1.1 Benchmarks

The list and description of our benchmarks are provided in Figure 5.1. As we have mentioned before, OwnKit is built on top of the experimental compiler JKit [2]. Unfortunately JKit is often unable to compile large programs due to limitations with type checking of generics. Due to this fact, the number of benchmarks we were able to analyse is somewhat limited.

5.1.2 Experiment

In order to compare OwnKit and UNO we ran both tools and recorded the number of fields inferred as owned, as well as measuring the time it took for both tools to run. Additionally we used OwnKit to measure the number of self-exposed

Benchmark	Version	Description
java-std	1.5	Core Java Standard Library packages
javacc	5.0	Parser Generator
polyglot	1.3.2	Compiler Framework (excluding extensions)
asm	3.2	Assembly Simulator
jgraph	5.9.2.0	Graph Visualization Tool
jvm98_raytracer	-	From the `SPECjvm98` Suite

Figure 5.1: Corpus. See Figure 5.5 for list of `java-std` packages.

Machine	Optiplex 760
CPU	Intel® Core™ 2 Duo CPU E8400 3.00GHz
RAM	3.2 GB
Operating System	GNU/Linux
Kernel Version	2.6.38-ARCH
JVM	Java™ SE Runtime Environment (build 1.6.0_26-b03)

Figure 5.2: Environment

classes as well as maintaining statistics on exposure reasons. During the design of the experiment we ensured that both tools operated under the same assumptions. For example, by default OwnKit allows for `String` fields to be owned, while UNO does not. Figure 5.2 provides information regarding the environment used to conduct the experimental runs.

When running OwnKit on the benchmarks, we did it twice, one in each of the following modes:

1. **Fast Mode** - The default mode of inference. In this mode OwnKit simply infers the set of owned fields. When applying the rules, we use an efficient work-list algorithm in order to avoid rule applications unless they lead to new exposures. This mode produces correct ownership annotations fast, but does not capture the reasons for the exposure of non-owned fields.

2. **Reasoning Mode** - In this mode OwnKit infers not only the set of owned fields, but also a complete reasoning for why non-owned fields are exposed. This mode is significantly slower and we only use it to gain deeper insight into how the fields gain their exposures.

One of the problems during the experiment was that UNO does not directly present the statistics regarding ownership of the fields in the given benchmark. Instead UNO produces a file containing evaluations of all generated predicates. To deal with this we have created two tools, `uno2xml` and `xmlanalysis`. The first tool allowed us to take the UNO output file and using pattern matching convert

Program	Lines of Code	Total Fields	% of Owned Fields		Classes	
			OwnKit	UNO	Self-Exp. %	Total
java-std	62,508	690	3.77	-	16.0	763
javacc	36,672	406	4.7	11.8	13.3	150
polyglot	14,148	421	0.5	2.9	11.0	327
asm	22,474	259	4.2	10.8	14.0	172
jgraph	12,262	178	5.1	3.9	29.2	89
jvm98_raytracer	1,928	40	12.5	5.0	28.0	25
Average			**5.13**	**6.91**	**19.1**	

Figure 5.3: OwnKit Ownership Inference Results. Lines of code are "Total Physical Source Lines of Code" as generated using David A. Wheeler's 'SLOCCount' [4]. Self-Exp refers to the percentage of classes that are determined to be self-exposed by OwnKit.

it into an XML file that describes the ownership status of each of the fields. The second tool can take the XML outputs of OwnKit itself or `uno2xml` in order to carry out comparisons or to generate ownership statistics.

5.2 Results

5.2.1 Ownership and Self-Exposure

Figure 5.3 presents the ownership and self-exposure information generated from our experiment. "Total Fields" here 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. The total numbers of inferred owned fields for both tools are presented in Figure 5.4. We were not able to analyse `java-std` with UNO because of a dependency analysis bug in Soot.

We would expect whole program analysis to be more precise and this was the case for of the most benchmarks. As expected, the whole-program approach of UNO results in more annotations. However the difference is not very significant, 6.91% vs 5.15% on average. 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 was quite high; 19% on average, with some benchmarks reaching up to 29%. This could be attributed to the conservative nature of the self-exposure inference algorithm; we analyse the effect of this self-exposure on ownership in the next section.

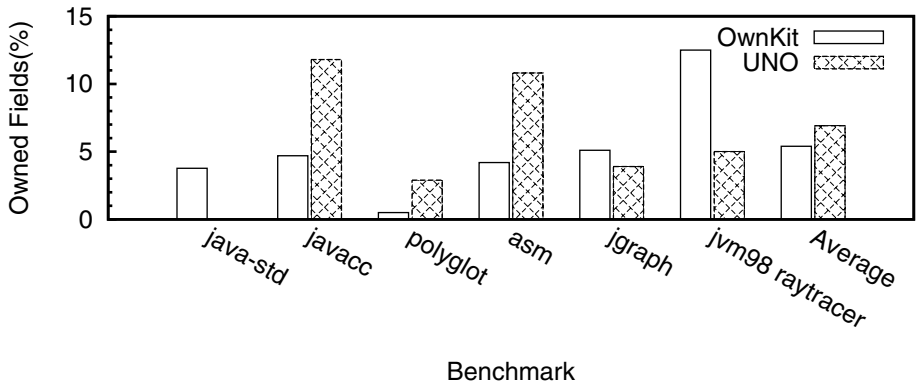


Figure 5.4: Field Ownership. The averages do not include the ownership results for `java-std` by UNO.

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

Figure 5.5: Ownership and Self-Exposure Results for Individual Packages using OwnKit. Class `java.util.Collections` was excluded due to JKit type-checking problems.

```
1  package pac.a;
2  public class MyClass {
3      @Owned private X myField;
4
5      ...
6  }
7
8
9  package pac.b;
10 public class Y extends X {
11     // Self-Exposing
12     ...
13 }
```

Figure 5.6: Self-Exposure and Modularity

In order to gain a better understanding of the ownership and self-exposure patterns we have analysed each of the `java-std` packages in isolation. Figure 5.5 presents the results. As we can see, both field ownership and self-exposure numbers vary a great amount depending on the package.

We have mentioned before that due to self-exposure, the inferred ownership annotations cannot be safe, unless the entire code base is analysed. The isolated analysis of packages demonstrates this fact very clearly. When we analysed the top level package `java`, we inferred 26 owned fields. However, when we analysed `java.lang`, `java.io` and `java.util` separately and added the results, there were 40 owned fields.

This problem is not unique to our algorithm, but is common to all ownership inference systems where full sets of classes are unavailable for analysis. In order to understand the cause of this problem, consider the classes in Figure 5.6. The inference algorithm must either make a very conservative assumption (any field that may contain an object of a non-final class cannot be owned), or make an assumption that the classes that extend it will not have self-exposure. When analysing package `pac.a` we may infer that class `X` is not self-exposing (assuming we use the second assumption), and field `myField` is owned. However, when we consider both `pac.a` and `pac.b` together we would realize that `myField` may in fact contain a self-exposing object of type `Y`, and not mark `myField` as owned.

5.2.2 Exposure Reasons

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

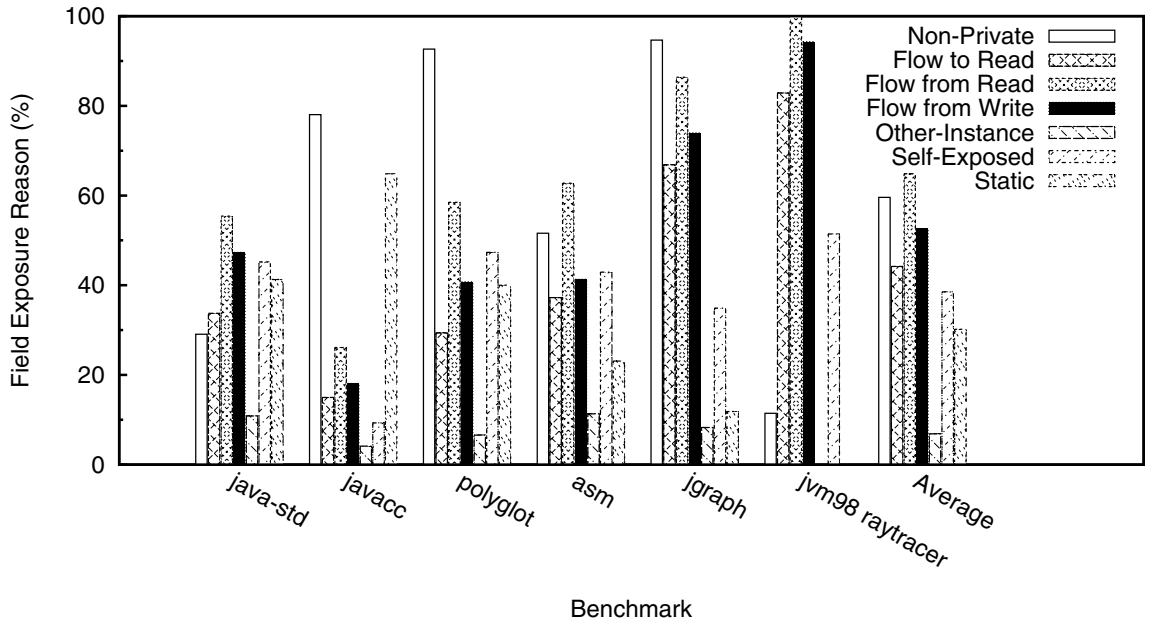


Figure 5.7: Field Exposure Reasons

of the fields is given in Figure 5.7. Note that each of the fields can have multiple exposure reasons. The description for these reasons is 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 described in Chapter 3).
- **Flow from Read** - A value flows from a `READ` exposed node into the field (Case #3 described in Chapter 3).
- **Flow from Write** - A value flows from a `WRITE` exposed node into the field (Case #2 described in Chapter 3).
- **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 `jvm98raytracer`, which is a small and narrow purpose tool.

Benchmark	OwnKit Time (s)	UNO Time (s)
java-std	83.25	-
javacc	20.46	9.55
jgraph	12.24	8.11
asm	26.52	7.18
jvm98_raytracer	0.99	7.99
polyglot	15.07	6.77
Total (excl. java-std)	**75.28**	**39.6**

Figure 5.8: Run Times

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 6.86% 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 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.

5.2.3 Performance

Figure 5.8 presents the times it took both tools to infer annotations for each benchmark. In order to better differentiate the performance of the tools, the given numbers omit the run-times of the underlying compilers (JKit [2] for OwnKit and Soot [46, 5] for UNO) as well as any input/output interactions performed during the analysis.

It is hard to make any definitive conclusions by analysing performance of the two tools as the intermediate languages they work on are provided by different front-ends (JKit and Soot). While the analysis times are comparable, we can see that overall UNO has better time performance on the corpus. One of the possible reasons for this is that the current implementation of OwnKit ends up performing CLFG construction (the most time expensive part of the algorithm) twice for every class. Reworking the top-level of the algorithm to eliminate this could allow the run time of OwnKit to be cut in half. For example, when we disabled the self-exposure inference stage, the run-time of OwnKit on `javacc` reduced from 20.46 seconds to 10.54 seconds.

Chapter 6

Conclusions

Encapsulation is a fundamental concept of object oriented design. Despite this, current programming languages do not offer mechanisms for protecting the internal representation of aggregate objects from being aliased by external entities. The concept of object ownership is designed to address this issue; however, current ownership schemas impose onto programmers the heavy burden of type annotation - a factor that works against the wide adoption of ownership in real-life projects.

In this thesis we have presented an algorithm capable of automatically inferring ownership annotations. Unlike other approaches our algorithm works in a modular fashion - only considering a single class at the time. While this leads to more conservative annotations, it theoretically leads to higher scalability. The run time of the analysis is linear on the number of classes in the program, and the peak space usage is only dependent on the size of the largest class.

6.1 Contributions

The main contributions of this thesis are as follows:

- **Modular Ownership Inference Algorithm** - We have designed and formalised of our modular inference algorithm.
- **Working Implementation** - The algorithm was used to create an automatic inference tool based on the JKit [2] compiler - OwnKit. A large suite of automated tests (89 tests in total) have been created in order to ensure the correctness of the tool.
- **Corpus Study** - The accuracy and performance of the tool has been evaluated by conducting of a small corpus study.

6.2 Future Work

Due to time restrictions, there are a number of projects that have been left outside the scope of this work:

- **Package-Level Analysis** - Our current algorithm only considers a single class at a time. While this makes it highly scalable, it also leads to conservative assumptions about the other classes in the program, producing less @Owned annotations as a result. Analysing the program one package at the time may allow us to achieve more precise analysis, without significant performance loss.

- **Further Corpus Studies** - As we have mentioned before, our choice of benchmarks is limited to those that can be compiled by JKit. The improvement of JKit or use of a different front-end could pave the way for a larger, more detailed corpus study of modular ownership inference.

- **Operational Semantics** - The intermediate language in our current formalisation is lacking a description of the operational semantics. The introduction of this aspect would allow us to provide formal proofs for termination and the ownership guarantee theorem.

- **Using Uniqueness Information** - One of the possible ways to improve the accuracy of our inference would be to use uniqueness information generated by other tools. For example, JPure [33] infers `@Fresh` annotations on method return types, indicating that the returned reference is a unique reference to the object. This information could allow OwnKit to be less conservative by not treating values returned by these methods as exposed.

Bibliography

[1] Eclipse. `http://www.eclipse.org/`.

[2] Java Compiler Kit. `http://homepages.ecs.vuw.ac.nz/~djp/jkit/`.

[3] OwnKit. `http://elvis.ac.nz/Main/OwnKit`.

[4] SLOCCount Homepage. `http://www.dwheeler.com/sloccount/`.

[5] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[6] ANDREAE, C., NOBLE, J., COADY, Y., GIBBS, C., VITEK, J., AND ZHAO, T. Stars: Scoped types and aspects for real-time systems. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2006), Springer-Verlag, Berlin, Heidelberg, Germany, pp. 124–147.

[7] BARNETT, M., DELINE, R., FÄHNDRICH, M., LEINO, K. R. M., AND SCHULTE, W. Verification of object-oriented programs with invariants. *JOT 3*, 6 (2004), 27–56.

[8] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 211–230.

[9] BOYAPATI, C., LISKOV, B., AND SHRIRA, L. Ownership Types for Object Encapsulation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)* (New Orleans, LA, USA, Jan. 2003), ACM Press, New York, NY, USA, pp. 213–223. Invited talk by Barbara Liskov.

[10] BOYAPATI, C., SALCIANU, A., BEEBEE, JR., W., AND RINARD, M. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2003), ACM Press, pp. 324–337.

[11] BROOKS, JR., F. P. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer 20*, 4 (Apr. 1987), 10–19.

[12] CLARKE, D., AND DROSSOPOULOU, S. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 292–310.

[13] CLARKE, D. G. *Object Ownership and Containment*. PhD thesis, New South Wales, Australia, 2002.

[14] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), OOPSLA '98, ACM, pp. 48–64.

[15] CRAIK, A., AND KELLY, W. *Using ownership to reason about inherent parallelism in object-oriented programs*. CC'10/ETAPS'10. Springer-Verlag, Berlin, Heidelberg, 2010.

[16] DIETL, W., DROSSOPOULOU, S., AND MÜLLER, P. Generic Universe Types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2007), Springer, pp. 28–53.

[17] DIETL, W., ERNST, M. D., AND MÜLLER, P. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)* (July 2011).

[18] DIETL, W., AND MÜLLER, P. Universes: Lightweight ownership for JML. *Journal of Object Technology 4*, 8 (2005), 5–32. `http://www.jot.fm/issues/issue_2005_10/article1`.

[19] DOLADO, J. J., HARMAN, M., OTERO, M. C., AND HU, L. An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension. *IEEE Trans. Softw. Eng. 29*, 7 (July 2003), 665–670.

[20] HENDREN, L. Scaling Java points-to analysis using Spark. In *Compiler Construction, 12th International Conference, volume 2622 of LNCS* (2003), Springer, pp. 153–169.

[21] HIRZEL, M., DINCKLAGE, D. V., DIWAN, A., AND HIND, M. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems* (2005).

[22] HIRZEL, M., DIWAN, A., HIND, M., HIRZEL, M., DIWAN, A., AND HIND, M. Pointer analysis in the presence of dynamic class loading. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2004), pp. 96–122.

[23] HOGG, J., LEA, D., WILLS, A., DE CHAMPEAUX, D., AND HOLT, R. The Geneva convention of the treatment of object aliasing. *OOPS Messenger 3*, 2 (April 1992), 11–16.

[24] INGALLS, D. Design Principles Behind Smalltalk. *BYTE*, 6 (1981), 286–298.

[25] LANDI, W. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (Dec. 1992), 323–337.

[26] MA, K.-K., AND FOSTER, J. S. Inferring Aliasing and Encapsulation Properties for Java. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007), ACM Press, New York, NY, USA, pp. 423–440.

[27] MILANOVA, A., AND LIU, Y. Practical static ownership inference, 2009.

[28] MILANOVA, A., AND VITEK, J. Static dominance inference. In *TOOLS Europe 2011* (2011).

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

[30] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming* (1999), A. Poetzsch-Heffter and J. Meyer, Eds., Fernuniversität Hagen, pp. 131–140.

[31] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A type system for alias and dependency control. Tech. Rep. 279, Fernuniversität Hagen, 2001.

[32] NOBLE, J., VITEK, J., AND POTTER, J. Flexible Alias Protection. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (July 1998), vol. 1445 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 158–185.

[33] PEARCE, D. J. JPure: A Modular Purity System for Java. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software* (Berlin, Heidelberg, 2011), CC'11/ETAPS'11, Springer-Verlag, pp. 104–123.

[34] PEARCE, D. J., KELLY, P. H. J., AND HANKIN, C. Efficient field-sensitive pointer analysis for C. In *In ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)* (2004), ACM Press, pp. 37–42.

[35] POETZSCH-HEFFTER, A., GEILMANN, K., AND SCHÄFER, J. Infering ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation* (2006), vol. 4444 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 120–144.

[36] POETZSCH-HEFFTER, A., AND SCHÄFER, J. Modular specification of encapsulated object-oriented components. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects* (Berlin, Heidelberg, 2006), FMCO'05, Springer-Verlag, pp. 313–341.

[37] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Defaulting Generic Java to Ownership. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)* (Oslo, Norway, June 2004), Springer-Verlag, Berlin, Heidelberg, Germany, pp. 311–324.

[38] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Featherweight Generic Ownership. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)* (Glasgow, Scotland, July 2005), Springer-Verlag, Berlin, Heidelberg, Germany.

[39] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Generic Ownership for Generic Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 311–324.

[40] POTANIN, A., NOBLE, J., ZHAO, T., AND VITEK, J. 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), pp. 311–324.

[41] POTTER, J., NOBLE, J., AND CLARKE, D. The ins and outs of objects. In *Australian Software Engineering Conference* (Adelaide, Australia, November 1998), IEEE Press, pp. 80–89.

[42] RAMALINGAM, G. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16* (September 1994), 1467–1471.

[43] ROBERT L. BOCCHINO, J., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel Java. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009), ACM Press, New York, NY, USA, pp. 97–116.

[44] ROUNTEV, A., MILANOVA, A., AND RYDER, B. G. Points-to analysis for Java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), OOPSLA '01, ACM, pp. 43–55.

[45] VAKILIAN, M., DIG, D., BOCCHINO, R., OVERBEY, J., ADVE, V., AND JOHNSON, R. Inferring Method Effect Summaries for Nested Heap Regions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2009), ASE '09, IEEE Computer Society, pp. 421–432.

[46] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), CASCON '99, IBM Press, pp. 13–24.