Collection Disjointness Analysis in Java

by

Hang Chu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Computer Software

Waterloo, Ontario, Canada, 2011

Author's Declaration

Abstract

This thesis presents a collection-disjointness analysis to find disjointness relations between collections in Java. We define the three types of disjointness relations between collections: must-shared, may-shared and not-may-shared. The collection-disjointness analysis is implemented following the way of a forward data-flow analysis using Soot Java bytecode analysis framework. For method calls, which are usually difficult to analyze in static analysis, our analysis provide a way of generating and reading annotations of a method to best approximate the behavior of the calling methods. Finally, this thesis presents the experimental results of the collection-disjointness analysis on several tests.

Acknowledgements

It is a pleasure to thank the many people who made this thesis possible.

It is difficult to express my thanks and gratitude to my supervisor Dr. Patrick Lam. With his enthusiasm, his breadth and depth of knowledge, and his patience, he helped to make research fun for me. I am thankful I had the opportunity to learn from him. Throughout my thesis writing period, he provided encouragement, good advice, and lots of good ideas. I was really moved when he read through my draft, words by words, times by times, to help me correct my spellings and English grammars.

I would also like to thank my thesis committee members Dr. Tan and Dr. Ray-side for their time and effort to read my thesis during this very busy time.

I also appreciate my colleagues Aakarsh Nair, Xavier Nombassi and Jon Eyolfson for taking time to discuss my problems and provide helpful suggestions.

Dedicated to my parents and my girlfriend. I really appreciate their support, understanding, encouragement and sacrifices. Without them I could have not completed it...

Table of Contents

List of Figures

“You can avoid reality, but you cannot avoid the consequences of avoiding reality.”

- Ayn Rand

Chapter 1

Introduction

A collection in Java is a "container" object, which stores multiple elements in a single unit. Containers are widely used to reduce programming effort and increase programming speed and quality by reusing known-good components which have well-understand behaviours. Two collections are disjoint if they do not contain objects pointing to the same heap location. We are interested in the disjointness relations of collections in Java because they enable light-weight specifications, thereby helping program understanding and parallelization. For example, if we store some important information in a collection and we wish to ensure that only operations on this collection can get or change the information, we can just specify that this collection must be disjoint with all other collections and check if this specifications is achieved by analyzing the disjointness relations between this collection and all other collections. Disjointness relation of any two collections therefore helps provide better understanding of our code. For parallelization, if we can find two collections that are completely disjoint in all executions, we can execute operations in parallel.

Programmers can check the disjointness of collections by comparing the contents of collections while running the code with test inputs—a dynamic analysis approach. However, the disadvantage of any dynamic analysis is that checking the disjointness in this way generally will not exhaustively cover all test cases with respect to infinite (or even, large) intractably input spaces. Therefore, we employ static analysis to analyze the disjointness relations of collections in Java. The benefit is that we can estimate the behavior of our code and find disjointness relations while accounting for all possible inputs.

```
Class C {
        public static void main(String [] args) {
                {  }
                List l1 = new LinkedList();
                {  }
                List l2 = new LinkedList();
                {  }
                List l3 = new LinkedList();
                {  }
                Object o1 = new Object();
                {  }
                Object o2 = new Object();
                {  }
                l1.add(o1);
                {(0(l1),3(o1))}
                l2.add(o2);
                {(0(l1),3(o1)),(1(l2),4(o2))}
                l3.add(o1);
                {(0(l1),3(o1)),(1(l2),4(o2)),(2(l3),3(o1)),(0(l1),2(l3)}
                l3.add(o2);
                {(0(l1),3(o1)),(1(l2),4(o2)),(2(l3),3(o1)),
                   (2(l3),4(o2)),(0(l1),2(l3)),(1(l2),2(l3))}
        }
}
```

FIGURE 1.1: A Collection-Disjointness Example.

In this thesis, we describe the design and implementation of a data-flow analysis on Java code to analyze the disjointness relations between collections. Our collection disjointness analysis works on a single method at a time. By using the disjointness analysis, programmers can compute or verify the disjointness relations between any two collections at any program point of the source code. With the information of the disjointness relations, programmers can provide light-weight specifications and better understand their code, and compilers can better optimize concurrent programs.

1.1 Approach

In our approach, we use a data-flow analysis to go through each method, tracking disjointness relations between all collections in the method. Our analysis starts with an initial data-flow set at the start of each method and applies transfer functions according to the statements in the method. Statements that change the data-flow set include collection operations and method calls. The data-flow set contains all of the information we need to find the disjointness relations. We can query the data-flow set at any program point to find the disjointness relations between any two collections at that point, although usually we are most interested in disjointness at the end of a method.

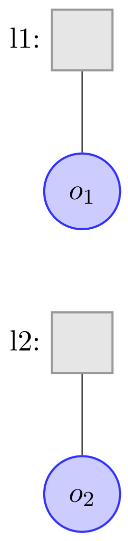


FIGURE 1.2: Result of the collection-disjointness analysis example after line 9

Suppose we wish to find the disjointness relations of collections in the method *main()* for the simple Java program in Figure 1.1. The *main* method of this program runs several *add* operations on three lists to make some of them contain common objects. Chapter 3 presents precise definitions for the contents of the data-flow set. Here we only want to show the basic approach of our collection-disjointness analysis. The data-flow set is initialized to be an empty set at the beginning of the *main()* method. From line 3 to line 7, the statements do not change the contents of the data-flow set, so the data-flow set after line 7 is still an empty set: object instantiations do not provide any disjointness information. After line 8, the *add()* operation on list $l1$ changes the data-flow set to $\{(0(l1), 3(o1))\}$, where 0 and 3 represents objects in the heap while $l1$ and $o1$ are variable references, implying that list $l1$ contains an object reference $o1$. Statements at line 8 to line 11 also change the contents of our data-flow set. The final result of our data-flow set after going through the *main()* method is $\{$(0(l1),3(o1)),(1(l2),4(o2)),(2(l3),3(o1)),(2(l3),4(o2)),(0(l1),2(l3)),(1(l2),2(l3))$\}$. Suppose we are interested in the disjointness relations between $l1$ and $l3$ at the end of the method. If we query the data-flow set after the statement at line 9, we will find that $l1$ and $l3$ are not-may-shared at this point since there is no pair consisting of $l1$ and $l3$ in our data-flow set, which means that no object in $l1$ may alias an object in $l3$. If we query the data-flow set after the statement at line 10, we will find that $l1$ and $l3$ are may-shared since there is a pair $(0(l1), 2(l3))$, which means there is at least one object in $l1$ may alias an object in $l3$.

Figures 1.2 and 1.3 depict graphically the results from the above example after line 9

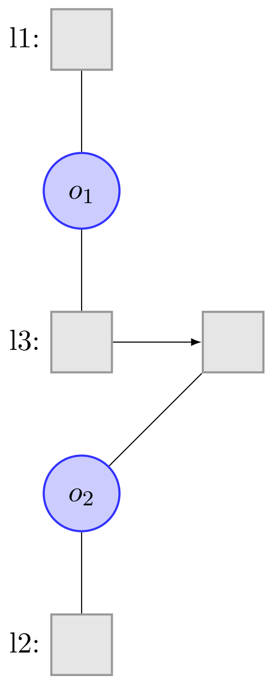


FIGURE 1.3: Result of the collection-disjointness analysis example after line 11

and at the end of the *main()* method respectively, showing the disjointness relations between $l1$, $l2$ and $l3$. the box represents a cell of the linked list, and the line to a round object indicates that this cell contains o_n. Figure 1.2 indicates that $l1$ and $l2$ are disjoint, and Figure 1.3 indicates that $l1$ and $l2$ are shared since they contain common objects.

We have now seen a brief example of the workings of our analysis. In the following chapters we present our implementation, along with more complicated examples.

1.2 Results

We run our collection disjointness analysis on two benchmarks for experiment. We successfully found eight disjoint collections from these two benchmarks. In chapter 4 we will present in more detail about our experimental results.

1.3 Limitations

Since our analysis employs a static data-flow analysis to determine disjointness relations between collections, our analysis has the limitations of all static analyses.

Because our analysis does not actually run the code, it can not provide fully precise results and must use approximations. Those approximations mostly arise from estimating the effects of statements that cause branches, such as the *if* statement.

Another limitation to our analysis is that the correctness of our analysis depends on developer-provided information while analyzing method calls. Although our analysis provides a way of generating annotations automatically, it still allows programmers to write their own annotations for convenience. If the annotations provided by the programmers are incorrect, our analysis may fail to give the correct answers.

1.4 Thesis Contributions

This thesis makes the following contributions.

1) **Collection-disjointness analysis:** This thesis presents a method for determining collection-disjointness which indicates whether two collections contain objects pointing to same heap location. We employ a data-flow analysis to determine the disjointness relations of collections. Our analysis assumes that Java collections are the primary way that programs maintain data structures, and uses this assumption to simplify its task—it processes the collection manipulation operations to understand the contents of collections.Our analysis employs a new type of data-flow set which easily shows the disjointness relations of all collections in the analyzing methods. Our analysis allows users to find disjointness relations between any two collections at any program points.

2) **Annotations for collection disjointness:** The thesis applies developer-provided annotations to our static analysis abstraction and presents a way of statically analyzing methods by generating and reading annotations. Annotations provides information about a program without affects program behaviour. Our analysis generates post-annotations from the pre-annotations of the method and then reads the post-annotations to verify developer-provided annotations and figure out the collection-disjointness relations.

3) **Collection-disjointness Analysis Implementation:** The data-flow analysis is implemented in Java under the Soot framework. The implementation allows developers to check the disjointness relations of any two collection at any program points.

4) **Experimental Results:** The thesis shows the experimental results at the end, which show the correctness of the implementation of our collection disjointness analysis.

1.5 Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 describes some background knowledge related to this thesis. This background knowledge helps understand basic concepts employed in the rest of the thesis.

Chapter 3 presents the details of the implementation of the collection-disjointness analysis. It provides all details of the data-flow analysis.

Chapter 4 presents the experimental results.

Chapter 5 presents works related to our collection-disjointness analysis. Finally, we present the conclusions and future work in Chapter 6.

Chapter 2

Background Knowledge

In this chapter, we present background knowledge and techniques used in our collection disjointness analysis. In particular, we use static analysis as the fundamental analysis method and employ a data-flow analysis, a specific type of static analysis, to build data-flow sets which indicate disjointness. Moreover, we use points-to analysis to define the elements of the data-flow sets, using object representatives as a summary of the heap objects in our points-to analysis. Our implementation of collection disjointness analysis is based on the Soot [18, 19] framework with the SPARK [10] points-to analysis. Therefore, we will present some background information on static analysis, data-flow analysis, points-to analysis, SPARK, object representatives, and Soot in the next few sections.

2.1 Static Analysis

Static Analysis is a technique for analyzing the source code of a program without running it. It estimates the behavior of the given program. Applications of static analysis include providing better understanding of the code for developers during testing and maintenance, as well as program optimization. Contrast static analysis with dynamic analysis. Dynamic analysis examines code by executing the program and considering its behavior on given inputs. Static analysis instead considers the program text independent of inputs (or, for all inputs). Since the input space of even simple programs is usually intractable, dynamic analysis will not generally exhaustively cover all test cases. It is therefore difficult to ensure that a dynamic analysis covers all program behaviors. Static analysis, on the other hand, typically

```
1.      if (...)
2.              x =1;
3.      else
4.              x =0;
```

FIGURE 2.1: Example Java Code.

returns results valid for all inputs without executing the code. However, because it does not actually run the code, static analysis can not provide fully precise results and must use approximations. In principle, this is due to the halting problem, but in practice, approximations mostly arise from estimating the effects of statements that cause branches, such as conditional statements and method calls.

For instance, consider the conditional statements in Figure 2.1. If we execute the code, x is either 0 or 1 because only one branch of the if statement is actually executed. However, without running the code, static analysis is in general not able to determine which branch will be executed. Static analysis therefore usually assumes that either branch may be executed. More generally, static analysis assumes any branch of a program may be executed during analysis. Although there are some techniques which can evaluate some branch conditions, the general treatment of branches introduces imprecision in static analysis results.

Although static analysis is not fully precise, it can still get useful and sound approximate results. The soundness here means that all approximation results provided are true. Data-flow analysis is one of the most useful techniques for performing static analysis.

2.2 Data-Flow Analysis

Data-flow analysis is a technique for calculating the effect of each program statement with respect to the possible sets of values in some abstract domain [13]. We can then gather information from the sets of values that the data-flow analysis computes. This information can be used for understanding, debugging and optimizing the original program. We explain the general concept of data-flow analysis by presenting an analysis to identify variables which are guaranteed to be defined. Figure 2.2 presents the code that we will analyze.

We set up this data-flow analysis in six steps [8]:

```java
void foo () {
  String a;
  int b;
  int c;
  a = "xyz";
  if (a.equals("xyz"))
    b = 2;
  else
    c = 3;
  a = new String();
}
```

FIGURE 2.2: Java Code to be Analyzed.

1. *What is the problem?* In our example, we wish to find all local variables with constant values at a given program point.

2. *Forward or backward analysis?* A forward data-flow analysis starts at the entry statement of a control flow graph and propagates information forward from there. A backward data-flow analysis starts at the exit statement of a control flow graph and propagates information back from that statement. In our example, we use forward analysis since we want to determine how each statement affects our input data-flow set.

3. *What is in the data-flow sets?* Since we are trying to determine if a local variable has a constant value at a given program point, our data-flow set should contains the local variables with constant values at the given program point.

4. *What is the operation at the merge point, union or intersection?* A merge point is a confluence of branches, which are usually caused by conditional statements. In our example, we have an *if* statement with two branches. Since static analysis can not figure out which branch is actually executed, our analysis needs to go through each branch and, at the end of each branch, combine the resulting data-flow sets. In our example, since we want a false negative result—all locals in our data-flow set must be guaranteed to have a constant value at the given program point—we use intersection to merge the sets generated by different program branches.

5. *What are the transfer functions?* Transfer functions calculate the changes of the data-flow set after a program statement.They rely on two sets for each program statement. The Gen set is a set of values to be added to the data-flow set while the Kill set is a set of values to be removed from the data-flow set. In our example, the transfer function is very simple. If a local variable has a constant value (e.g. x=5), we include it into our Gen set. If a local variable had a constant value but does not

```
1.      void foo () {
                                        {}
2.        String a;
                                        {}
3.        int b;
                                        {}
4.        int c;
                                        {}
5.        a = "xyz";
                                        {a}
6.        if (a.equals("xyz"))
7.          b = 2;
                                        {a, b}
8.        else
9.          c = 3;
                                        {a, c}
                                        {a}
10.       a = new String();
                                        {}
11.     }
```

FIGURE 2.3: A simple data-flow analysis example. Sets on the right indicate defined local variables

have it anymore, we include it into our Kill set. Note since we use intersection as the merge point operation in this example, in branches we must include "possibly generated" variables to the Gen set and "definitely killed" variables to the Kill set.

6. *What are the initial values?* The initial value of a data-flow analysis is the contents of the data-flow set at the beginning (or end, for backward analysis) of a method. In our example, it is obviously the empty set since no locals are initialized and assigned with constant values at the beginning of each method.

Figure 2.3 shows an example of operation of our analysis.

At the beginning of this method, before line 1, our data-flow set is initialized to be the empty set. Lines 2–4 are local declarations with no assignments. Our transfer function does not change the data-flow set on these lines. At line 5 we define string *a* to be "xyz". The Gen set is {a} and the Kill set is $\emptyset$. Thus, the data-flow set after line 5 is {a}. Lines 6–9 contain an if statement. As we mentioned previously, our analysis needs to go through each branch and intersect the results at the merge point. At line 7, b is defined to be 2. Thus, the Gen set is {b} while the Kill set is $\emptyset$. The result after line 7 is {a, b}. Similarly, the result after line 9 is {a, c}. Now we compute the intersection of sets generated from both branches and get the result {a}, which means at the end of the if statement, only *a* is guaranteed to have constant value. This result contains false negatives since either *b* or *c* has constant value at this point but our analysis does not know exactly which one has and our abstraction cannot represent such information. At line 10, a is assigned with a new String object, so that a no longer have a constant value. Therefore, the Gen set is $\emptyset$ and the Kill set is {a}.

In our disjointness analysis, we will also employ data-flow analysis to compute disjointness between collections contents. The data-flow set at each program point will contain the results of the analysis at that point.

2.3 Points-to Analysis and SPARK

Many languages such as C and Java use pointers and dynamic storage, which makes static analysis difficult: finding exact run-time values of variables in static analysis is generally uncomputable [9]. In the presence of pointers, variables may alias each other. We define aliasing as follows. Two or more variables alias each other if they point to the same memory location. Changing the contents of one variable will result in the same changes to all other aliased variables. Furthermore, if a new variable $r1$ aliases with another variable $r2$, $r1$ also aliases with all other variables aliasing with $r2$.

Points-to analysis (pointer analysis) is a technique for determining the set of memory locations which a variable may point to [14]. A points-to set for a variable x represents a set of heap locations that x may point to. By examining the points-to sets of different variables, we can find the aliasing relations between them. In particular, there are three types of aliasing: may alias, must alias and may not alias. If two variable x and y point to the same heap location in some execution of the program, we say x and y may alias; if two variables x and y point to the same heap location in all executions of the program, we say x and y must alias; if two variable x and y never point to the same heap location in all execution of the program, we say x and y may not alias.

In the collection disjointness analysis, we need to check not only if two collections contain the same variable but also if two collections contain variables that alias with each other, which makes the points-to analysis very useful. In particular, if two collections contain variables that must-alias with each other, we can conclude that these two collections are must-shared. To get points-to sets of variables in a program, we use the SPARK [10] Java points-to analysis framework, which is integrated into the Soot [18, 19] framework. The Soot framework also employs the object representatives [1], which represents variables abstractly, highlighting their points-to relations. In the next section, we will present some background knowledge about object representatives.

2.4 Object Representatives

Object representatives are a notation which integrate pointer analysis results from flow-insensitive interprocedural analysis with the results from flow-sensitive intraprocedural analysis [1]. A flow-sensitive analysis takes the order of program statements into account while a flow-insensitive analysis does not. By using object representatives, analysis can easily determine whether two variables point to the same heap object or point to different heap objects. The object representative for a variable a is in the form of $m(a)$, where the integer m represents an object in the heap while a represents the variable reference for easy understanding. To determine whether two variable point to the same heap object, we compare object representatives by provided MustAlias and NotMayAlias analysis.

In the collection disjointness analysis, we employ object representatives and use them as the elements in the data-flow set to represent the aliasing relations. Using the object representative simplifies the implementation of our analysis. We will provide more details in chapter 3.

2.5 Soot and Jimple

Soot [18, 19] is a Java optimization framework to analyze and transform Java bytecode using a suite of intermediate representations.

Jimple [18, 19] is a three-address code intermediate representation of Java bytecode used by Soot. It linearizes and names expressions so that most statements only reference at most 3 local variables or constants. It also introduce new local variables for implicit stack locations. The most convenient advantage of Jimple is that an analysis implementation only needs to handle 15 statements in the Jimple representation, compared to more than 200 possible instructions in Java byte code [3].

Figure 2.5 shows the Jimple representation for the example source code shown in Figure 2.4. *temp$0* at line 11 is a local variable introduced by Jimple, as are *temp$1*

```
1.      public static void main (String [] args) {
2.
3.        List <String>l1 = new LinkedList<String>();
4.        String str = "xyz";
5.        l1.add(str);
6.        System.out.println(l1.toString());
7.      }
```

Figure 2.4: Java Method in the Original Form

```
1.    public static void main(java.lang.String[])
2.    {
3.        java.lang.String[] args;
4.        java.util.List l1;
5.        java.util.LinkedList temp$0;
6.        java.lang.String str, temp$3;
7.        boolean temp$1;
8.        java.io.PrintStream temp$2;
9.
10.      args := @parameter0: java.lang.String[];
11.      temp$0 = new java.util.LinkedList;
12.      specialinvoke temp$0.<java.util.LinkedList: void <init>()>();
13.      l1 = temp$0;
14.      str = "xyz";
15.      temp$1 = interfaceinvoke l1.<java.util.List:
                              boolean add(java.lang.Object)>(str);
16.      temp$2 = <java.lang.System: java.io.PrintStream out>;
17.      temp$3 = interfaceinvoke l1.<java.util.List:
                              java.lang.String toString()>();
18.      virtualinvoke temp$2.<java.io.PrintStream:
                              void println(java.lang.String)>(temp$3);
19.      return;
20.   }
```

Figure 2.5: Jimple Representation for Java Method in Figure 2.4

at line 15, *temp$2* at line 16 and *temp$3* at line 17. All variables are given explicit types, like boolean and java.util.List. Jimple converts Java bytecode or source code into its simplified three-address code. For example, method *add()* at line 5 and method *toString()* at line 6 in the original code are represented by the *interfaceinvoke* statement in Jimple at line 15 and line 17 respectively. Our collection disjointness analysis analyzes Jimple statements instead of Java bytecode instructions, which makes our analysis easier to implement. We will explain our treatment of each Jimple statements in more detail in chapter 3.

Chapter 3

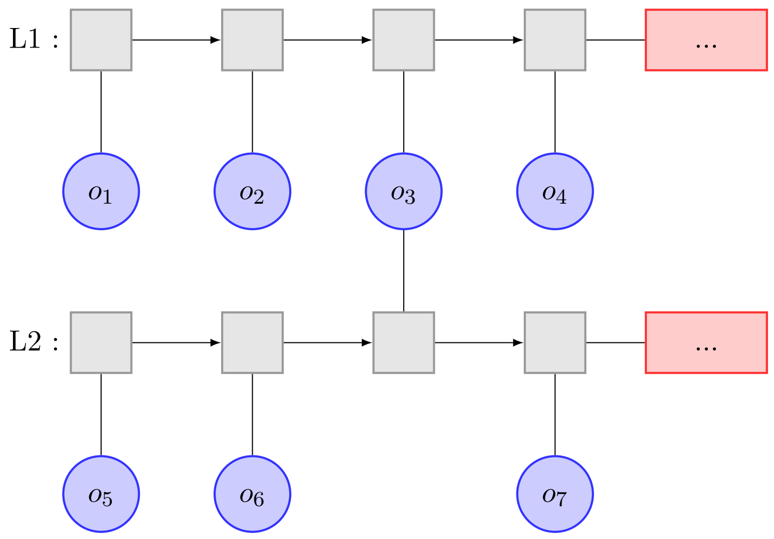
Implementation

In Chapter 2, we introduced some background knowledge related to our collection disjointness analysis, including static analysis, data-flow analysis, points-to analysis and SPARK, object representatives and Soot. In this chapter, we provide a detailed description of our collection disjointness analysis and include details about our implementation.

Since our collection disjointness analysis is implemented in Soot as a data-flow analysis, in this chapter we focus on the six steps of implementing a data-flow analysis. Chapter 3.1 describes the problem definition; chapter 3.2 describes the contents of the data-flow sets; chapter 3.3 shows the merge operation. Chapter 3.4 is the most important section: it provides a detailed description of our transfer functions and discusses their implementations. Finally, chapter 3.5 describes the initial values of the data-flow set.

3.1 Problem Definition

Our collection disjointness analysis checks whether two given collections contain objects that may point to the same heap location (see Figure 3.1). Our heap abstraction operates on a set of abstract objects O. O has an interesting subset L, the subset of collection objects. We only track the containment relation starting with objects in L. Our goal is to find if two objects in L contain any objects that may overlap in the heap.



The blue circles represent the objects, the gray rectangles represent references in the lists and the red rectangles represent the rest of the lists. In this example, L1 and L2 are not disjoint.

FIGURE 3.1: Collection-disjointness abstraction on collections $L1$ and $L2$

Now we introduce the abstraction we use for objects. Since Java does not allow programmers to access a physical memory address in any way (contrast this with languages such as C/C++), programmers can only get references to objects and cannot manipulate a pointer to get different object. It therefore suffices to model the object-typed variables in the program by object representatives.

We define three types of disjointness relations: may-shared, not-may-shared and must-shared. Recall the definitions of aliasing relations in points-to analysis in Chapter 2.3. By using these definitions we can define our disjointness relations as follows: if collections $l1$ and $l2$ contain at least one object that must alias, we say $l1$ and $l2$ are must-shared; if on any execution two collections $l1$ and $l2$ contain at least one objects that may alias, we say $l1$ and $l2$ are may-shared; if two collections $l1$ and $l2$ contain no objects that may alias, we say $l1$ and $l2$ are not-may-shared, i.e. disjoint.

Note that objects in L may contain other objects in L. In general, a collection $l1$ may contain another collection $l1'$. To take this condition into account, we modify our definition of disjointness:if two collections $l1$ and $l2$ contain at least one object that must alias or at least one collection that is must-shared, we say $l1$ and $l2$ are must-shared; if two collections $l1$ and $l2$ contain at least one object that may alias or at least one collection that is may-shared, we say $l1$ and $l2$ are may-shared; if two collections $l1$ and $l2$ contain no objects that may alias or no collections that are may-shared, we say $l1$ and $l2$ are not-may-shared.

Since at any program point, we want to see how the transfer function of current statement takes effect to our input data-flow set which shows the disjointness relation, we implement our disjointness analysis as a forward data-flow analysis. For each method, the analysis runs from the top of the method's control flow graph (entry points) to the bottom of the control flow graph, including all branches of this method. For the whole program, the analysis runs from method to method, organized depending on the caller and callee relations generated by the call graph. Our disjointness analysis should always analyze a callee method prior to its caller method. However, a call graph may contain cycles. For example, imagine that method A calls method B, method B calls method C and method C calls method A. Our analysis applies a fix point technique to this case. The detailed implementation of method ordering is covered in Chapter 3.4.3.2.

Our analysis may return false negatives in the following sense: for instance, if we say that two collections are not-may-shared with each other, it is true that they are not-may-shared with each other no matter what approximations we made during the analysis process. On the other hand, if we say that two collections are may-shared or must-shared with each other, there is a chance that, on some executions, they are not-may-shared with each other because of our approximations, such as the merge point operation. In the next session, we will discuss the contents of our data-flow set, which track the sharedness relation at any program point.

3.2 Data-flow Set

Elements of our data-flow set are "connections". A connection C [5] is a set of pairs $H = O \times O$, where O is a set of abstract objects in the heap. We use object representatives to represent abstract objects. For example, if there is a pair ($o1$, $o2$) of abstract objects, in our representation, the pair is shown as ($m(o1)$, $n(o2)$) where m and n are natural numbers representing the object representatives of $o1$ and $o2$ respectively. Soot computes the object representative for each variable and updates the map automatically, so that we can simply assume that object representatives are available and accurate.

Our abstraction stores two types of pairs H in the connection C:

1) *containment* pairs: If a collection $l1$ has an element $o1$ in it, H includes a containment pair $(m(l1),n(o1))$. $m(l1)$ is the containing element while $n(o1)$ is the contained element. That is, in a containment pair, the containing element is the first element in the pair while the contained element is the second element in the pair. Containment pairs are created after collection operations such as $l1.add(o1)$.

2) *sharedness* pairs: Sharedness pairs describe non-disjointness relations between collections (We call non-disjointness relations as the "sharedness" relations in the rest of this thesis.). For example, a may-shared relation between $l1$ and $l2$ gives rise to a sharedness pair $(m(l1),n(l2))$, which means $l1$ and $l2$ contain at least one object that may alias or at least one collection that is may-shared; The sharedness pairs may be created after each statement by our sharedness method (presented in Chapter 3.4.4.1, which pairs collections if they contain aliasing objects or shared collections. Sharedness pairs may also be created after collection operations such as $l1.addAll(l2)$.

A pair consisting of two collections can be either a containment pair or a sharedness pair. For example, pair $(m(l1),n(l2))$ can be a containment pair if $l1$ contains $l2$ or a sharedness pair if they are must-shared. Since these two types of pairs are created differently, we also store sharedness pairs in a separate list and query the list when we need to figure out if a pair is a containment pair or a sharedness pair.

Since our abstraction operates at the level of object representatives, we only care about the value of the object representative and include the variable names in the parentheses as a convenience to the reader. If two references $o1$ and $o2$ have the same object representative n, we consider the representations $n(o1)$ and $n(o2)$ to be equal since references having the same object representative must point to the same heap location. If two pairs contain the same pair of object representatives (order does not matter), we say these two pairs are equal. For example, pair $(m(a), n(b))$ is equal to pair $(m(c), n(b))$, as is $(n(b), m(c))$. Our connection only contains distinct pairs. For pairs that are equal, we arbitrarily choose one of them and store it in our connection.

The semantics of our pairs are as follows:

1) If there is a containment pair $(m(l),n(o))$ in the connection, collection l may contain object o. Otherwise, l definitely does not contain o.

```
1.      List l1 = new LinkedList();      // l1 has object representative 0.
2.      List l2 = new LinkedList();      // l2 has object representative 1.
3.      Object o1 = new Object();        // o1 has object representative 2.
4.      Object o2 = new Object();        // o2 has object representative 3.
5.      if (...)
6.        l1.add(o1);
              {(0(l1),2(o1))}
7.      else
8.        l1.add(o2);
              {(0(l1),3(o2))}
              {(0(l1),2(o1)), (0(l1),3(o2))}
9.      l2.add(o1);
              {(0(l1),2(o1)), (0(l1),3(o2)), (1(l2),2(o1)), (0(l1),1(l2))}
```

FIGURE 3.2: Merge Point Operation Example

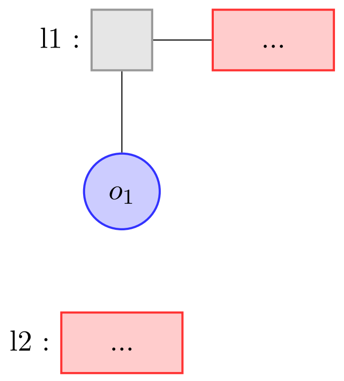
2) If there is a sharedness pair $(m(l1),n(l2))$ in the connection, $l1$ and $l2$ are either may-shared or must-shared. Otherwise, they are not-may-shared. Currently our analysis does not differentiate between a sharedness pair that represents a may-shared relation or a must-shared relation. Our analysis accurately identifies not-may-shared relationships.

Our data-flow sets, connections, contain these two types of pairs. Containment pairs are created by transfer functions and are used for further calculation by transfer functions. Sharedness pairs are created by the sharedness method and sometimes by the transfer functions and they show the disjointness relations we are looking for. In the next section, we will discuss the merge point operation, which combines connections at control-flow merges.

3.3 Merge Point

Since we employ a data-flow analysis, our analysis must run on every branch in the code. Thus we need a merge point operation to compute our result at control flow merges. Because we want to have only false negatives in our result, we choose union to properly gather information from every branch as our merge point operation.

Consider the example in Figure 3.2. Let us only consider the contents of the data-flow set in this example. We will present our transfer functions in more detail in the next section. From line 1 to line 4 we instantiate four objects. Two of them are lists and two of them are objects. From line 5 to line 8, there is an *if* statement resulting in two branches. After line 6, in the first branch, the data-flow set is $\{(1(l1), 3(o1))\}$, reflecting the fact that $o1$ was added to $l1$. After line 8, in the second branch, the



l1 contains o1 in the first branch.

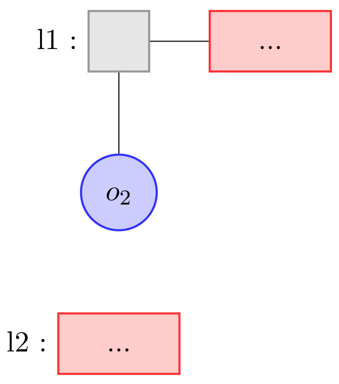
FIGURE 3.3: Result after line 6 in the merge point example.

data-flow set is $\{(1(l1), 4(o2))\}$, reflecting the fact that $o2$ was added to $l1$. Note that these two sets are both calculated from the data-flow set after line 4. The set after line 8 is not calculated from the set after line 6 because the code in line 4 and the code in line 6 are independent. Since we do not know which branch runs during the code execution at line 9, $l1$ may contain $o1$ from line 6 and $l1$ may contains $o2$ from line 8. Thus our data flow set should include all possibilities from each branch to represent that the list $l1$ may contains either $o1$ or $o2$. Therefore, we use union to merge the data-flow sets after line 6 and line 8 and the result is $\{(1(l1), 3(o1)), (1(l1), 4(o2))\}$.

The data-flow set after line 9 illustrates the correctness of our merge point operation. The list $l2$ contains object $o1$ after line 9 and the list $l1$ may contain $o1$ from the previous code. Since $l1$ and $l2$ contain the same object $o1$, we can conclude that $l1$ and $l2$ are may-shared. Therefore, our sharedness method adds a sharedness pair $(1(l1), 2(l2))$ to the data-flow set. The sharedness pair $(1(l1), 2(l2))$ in the data flow set shows the may-shared relation between $l1$ and $l2$, so our analysis correctly indicates that $l1$ and $l2$ are may-shared.

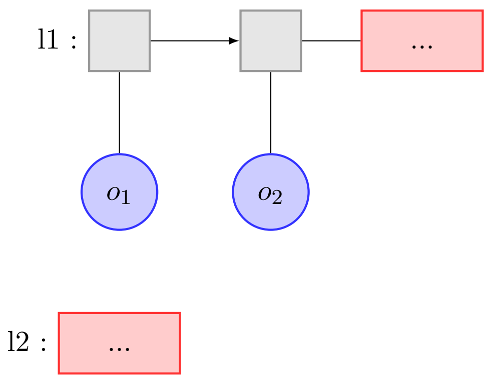
Figures 3.3–3.6 depict graphically the results of the merge point operation example.

In the next section, we will discuss the transfer functions we used to calculate the data-flow set after each statement.



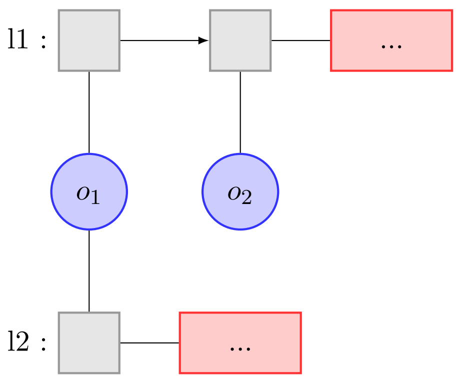
l1 contains o2 in the second branch.

FIGURE 3.4: Result after line 8 in Figure 3.2 before the merge point operation.



l1 may contain o1 and o2 after the merge point operation.

FIGURE 3.5: Result after line 8 in Figure 3.2 after the merge point operation.



l1 and l2 are may-shared or must-shared.

FIGURE 3.6: Result at the end of the method in Figure 3.2

3.4 Transfer Functions

Transfer functions in data-flow analysis are functions used to account for changes of the data-flow set caused by each program statement. In particular, transfer functions compute two sets for each statement: the Gen set and the Kill set. The Gen set contains elements that must be added to the data-flow set after the given statement while the Kill set contains elements that may be removed from the data-flow set after the given statement. In Chapter 3.4.1, we will present transfer functions for basic Java statements including object instantiation and assignment; in Chapter 3.4.2, we will present transfer functions for collection operations, such as add() and remove(); in Chapter 3.4.3, we will present transfer functions for method calls; and in Chapter 3.4.4 , we will present transfer functions for some additional functions such as generation of sharedness pairs.

3.4.1 Basic Operations

We consider the following basic operations:

Instantiations The process of creating an object of a class is called instantiation. An object is always an instance of a class. The instantiation process in Java is almost always followed by an assignment to a variable. This variable can be a newly declared variable or a previously assigned variable. For transfer functions at an object instantiation statement, we consider three cases:

1) If there is an instantiation of an object assigned to a newly declared variable o, since this statement does not change any containment or sharedness relations, the transfer function does not change the data-flow set. Therefore, the Gen set is $\emptyset$ and the Kill set is $\emptyset$.

2) If there is an instantiation of an object, other than a collection, assigned to a previously assigned variable o, since this statement does not change any containment or sharedness relations, the transfer function also does not change the contents of the data-flow set. Therefore, the Gen set is $\emptyset$ while the Kill set is $\emptyset$. Note that if some collection l has contained the previous object o before this statement, we should keep the containment pair $(m(l), n(o))$ in the Connection since the list l still contains a reference pointing to the heap location of the

```
1.      List l1 = new LinkedList();      // l1 has object representative 0.
2.      List l2 = new LinkedList();      // l2 has object representative 1.
3.      Object o = new Object();         // o has object representative 2.
4.      l1.add(o);
        {(0(l1),2(o))}
5.      l2.add(o);
        {(0(l1),2(o)), (1(l2),2(o)), (0(l1),1(l2))}
6.      o = new Object();
        {(0(l1),2(o)), (1(l2),2(o)), (0(l1),1(l2))}
7.      l1 = new LinkedList();
        {(0(l1),2(o)), (1(l2),2(o)), (0(l1),1(l2))}
```

Figure 3.7: Instantiation Example

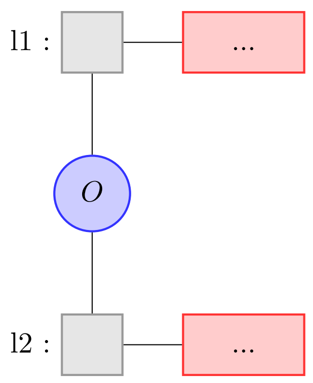
variable o pointed to before this statement. The object representative for o after the instantiation will, however, be a fresh object.

3) If there is an instantiation of a collection assigned to a previously assigned collection l, although this statement makes l points to a fresh collection, it does not change the containment and sharedness relations of the collection object l previously pointed to. Therefore, the Gen set is $\emptyset$, and the Kill set is $\emptyset$, too. Like the previous condition, the object representative for $l1$ after this statemetn will be a fresh object.

Figure 3.7 shows an example of how our analysis works on object initializations and adds.

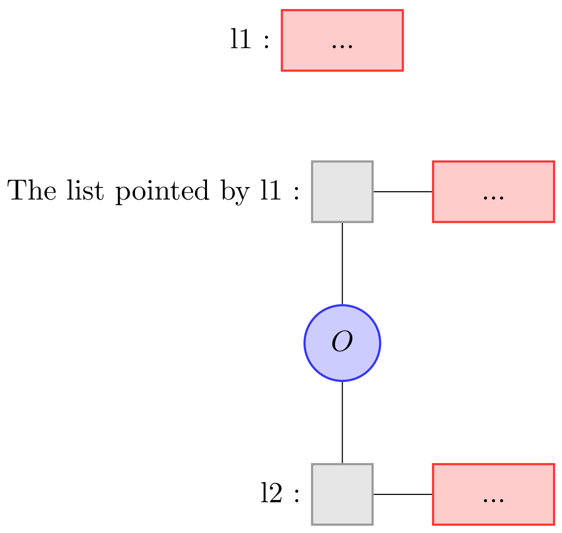
At lines 1–3, three objects are instantiated and assigned to three newly declared variables $l1$, $l2$ and o, respectively. Thus, following condition one, the Gen set is $\emptyset$ and the Kill set is $\emptyset$ after each statement. At line 4, the $add()$ operation on list $l1$ adds containment pair $(0(l1), 2(o))$ to the connection. Similarly, at line 5 the $add()$ statement on list $l2$ adds containment pair $(1(l2), 2(o))$ to the connection. At line 6, since there is an instantiation of an object other than a list to a previously assigned variable o, following Condition two the Gen set is $\emptyset$ and the Kill set is $\emptyset$. Note that after this statement, the object representative of the object that variable o points is not 2 any longer. However, since list $l1$ and $l2$ still contain a reference pointing to an object with object representative 2 that o pointed to previously, our analysis keeps containment pairs $(0(l1), 2(o))$ and $(1(l2), 2(o))$ in the connection. At line 7, there is an instantiation of a list to a previously assigned list $l1$. Following Condition three, the Gen set is $\emptyset$ and the Kill set is $\emptyset$.

Figures 3.8 and 3.9 depict graphically the results of the instantiation example.



Both l1 and l2 contain the reference pointing to the object O, which was also pointed by the reference o before line 6.

FIGURE 3.8: Result after line 6 in Figure 3.7



l2 contains the reference pointing to the object O, which was also also pointed by reference o before line 6. l1 does not contain object O but the collection it previously pointed to (with object representative 0) still contains O.

FIGURE 3.9: Result after line 7 in Figure 3.7

```
1.      List l1 = new LinkedList();
2.      List l2 = new LinkedList();
3.      Object o1 = new Object();
4.      Object o2 = new Object();
5.      l1.add(o1);
6.      l2.add(o2);
        {(0(l1),2(o1)),(1(l2),3(o2))}
7.      o2 = o1;
        {(0(l1),2(o1)),(1(l2),3(o2))}
8.      l2 = l1;
        {(0(l1),2(o1)),(1(l2),3(o2))}
```

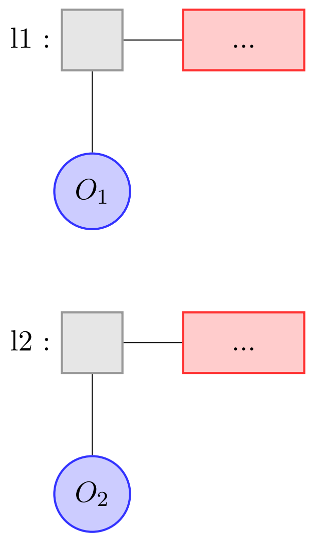
FIGURE 3.10: Assignment Example

Assignment Next, consider assignment statements. An assignment statement in Java replaces a reference to an object with a reference to another object. Assignment statements always have Gen=Kill=$\emptyset$. We will discuss the reason for this below. We classify the transfer functions of assignment statements as follows:

1) For an assignment statement $o2 = o1$, if a non-collection variable $o1$ with object representative m is assigned to another variable $o2$ with object representative n, the Gen set is $\emptyset$ and the Kill set is $\emptyset$ since this assignment does not change any containment or sharedness relations but does change the object representative for $o2$. As in condition two of object instantiation above, note that we still keep all containment pairs $(lkey(l), n(o2))$ where a collection l with object representative $lkey$ contains $o2$, since the collection l still contains a reference pointing to the heap location of the variable $o2$ pointed to before this statement, represented by object representative n. However, the object representative analysis will change the representative of $o2$ to m.

2) For a collection assignment statement $l2 = l1$, if a collection $l1$ with an object representative m is assigned to another collection $l2$ with an object representative n, the object representative of $l2$ is changed to n after this statement. Now reference $l2$ points to the collection object that reference $l1$ points to. Since this statement does not create any new containment and sharedness relations, and the collection object pointed by $l2$ with the object representative n keeps its containment and sharedness relations, the Gen set for a collection assignment is $\emptyset$ and the Kill set is $\emptyset$.

   Figure 3.10 shows an example of how our analysis deals with the above types of assignment statement.



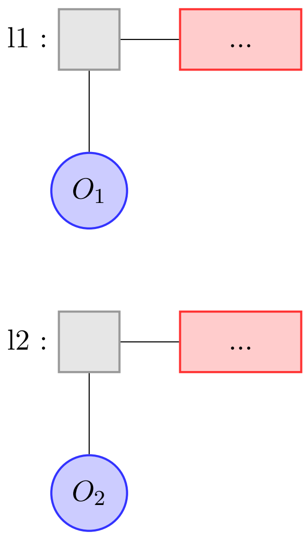
l1 contains the reference pointing the object O_1, which is also pointed by reference $o1$; l2 contains the reference pointing to the object O_2, which is also pointed by reference $o2$.

FIGURE 3.11: Result after line 6 in Figure 3.10

After line 6, the connection is $(0(l1), 2(o1)), (1(l2), 3(o2))$ after the instantiations from line 1 to line 4 and the add operations from line 5 to line 6. At line 7, object $o1$ with object representative 2 is assigned to $o2$ with the object representative 3. Following condition one, the Gen set is $\emptyset$ and the Kill set is $\emptyset$. At line 8, list $l1$ with the object representative 0 is assigned to the list $l2$ with the object representative 1. Following condition two, the Gen set contains the containment pair $(0(l2), 2(o1))$ (because $l1$ contains object $o1$). The Kill set contains the containment pair $(1(l2), 3(o2))$. Since the connection already contains pair $(0(l1), 2(o1)$ which is identical to $(0(l2), 2(o1))$, pair $(0(l2), 2(o1))$ in the Gen set is not actually added to the connection.

Figures 3.11 and 3.12 depict graphically results of the Assignment Example.

3) For a field read $l = o.f$, where l is a collection and $o.f$ is a collection field of an object o, it is difficult to track the containment relations and sharedness relations of l since we must understand other updates of the field $o.f$. Techniques like *Object Histories* [12] or *Thin Slicing* [15] can track the updates of a field $o.f$ but our analysis can not. Figure 3.13 shows an example of a field read. Lines 18–20 define a class *CollectionField* with two collection fields a,b. Lines 1–11 defines a method m in which we write two may-shared collections $l1$ and $l2$ to fields *lf.a* and *lf.b* respectively. $l1$ and $l2$ are may-shared because we add



l1 contains the reference pointing the object O_1, which is also pointed by references $o1$ and $o2$; l2 contains the reference pointing to the object O_2, which was previously pointed by reference $o2$ before the object assignment.

FIGURE 3.12: Result after line 7 in Figure 3.10

object o defined at line 4 to $l1$ and $l2$ at line 5 and 6 respectively. Lines 12–17 defines a method *foo* in which we read fields *lf*.a and *lf*.b and assign them to collections $l3$ and $l4$ respectively. $l3$ and $l4$ should have been may-shared if we call method m immediately before the method *foo*. However, our connection after line 16 does not show the expected result. In order to solve this problem, we designate all collections assigned from field reads are grouped as *external collections*, which we store in a list. We assume that all *external collections* are may-shared with all collections (In fact, *external collections* are may-shared with all collections except collections newly instantiated. Our analysis cannot track if a collection is initialized but never assigned. The assumption does not effect the correctness of our results since our results only contain false negatives to indicate if two collections are not-may-shared.) As with our data-flow sets, our analysis stores object representatives of *external collections* in the list. We express the formal definition as follows: if a collection l with object representative m is assigned from a collection read $o.f$ with object representative n at an assignment statement, we call collection $l1$ a *external collection* and add object representative $n(l)$ to the list of *external collections*. Note that we add $n(l)$ because the object representative of l changes to n after the assignment. (Behind the scenes, since the Jimple codes for $l = o.f$ are $l1 =$ *temp$1*

```
m(CollectionField lf) {
        ...
        Collection l1 = new LinkedList();
        Collection l2 = new LinkedList();
        Object o = new Object();
        l1.add(o);
        {(1(l1),6(o))}
        l2.add(o);
        {(1(l1),6(o)),(2(l2),6(o)),(1(l2),2(l2))}
        lf.a = l1;
        lf.b = l2;
        ...
}

foo(CollectionField lf) {
        Collection l3 = new LinkedList();
        Collection l4 = new LinkedList();
        l3 = lf.a;              // l3 has object representative 1
        l4 = lf.b;              // l4 has object representative 2
        { }
                ...
}

class CollectionField {
        Collection a,b;
}
```

FIGURE 3.13: Field Write and Read Example

and *temp$1* = o.⟨O: java.util.Collection f⟩, where *temp$1* is a local variable introduced by Jimple, the object representative we actually add to the list of *external collections* is $n($*temp$1*$)$. $n(l)$ and $n($*temp$1*$)$ are identical since the numbers in these two object representatives are equal, as we presented in the previous chapter. In our thesis, we use the collection name instead of temp variable for ease of understanding.)

Note since any object can be a collection at the runtime, we extend our *external collections* to include all objects $o1$ in the field read $o1 = o.f$.

In this section, we described the transfer functions on basic Java operations such as instantiation and assignment. In the next section, we will discuss transfer functions on basic collection operations in Java.

3.4.2 Collection Operations

We need only consider collection operations which change the contents of collections, since only those operations affect collection disjointness. Figure 3.14 shows our collection disjointness analysis operating on code that calls the most common collection operations.

```
1.      List l1 = new LinkedList();
2.      List l2 = new LinkedList();
3.      Object o1 = new Object();
4.      Object o2 = new Object();
5.      l1.add(0, o1);
        {(0(l1),2(o1))}
6.      l2.addFirst(o2);
        {(0(l1),2(o1)),(1(l2),3(o2))}
7.      l2.addLast(o1);
        {(0(l1),2(o1)),(1(l2),3(o2)),(1(l2),2(o1)),(0(l2),1(l2))}
8.      l1.addAll(l2);
        {(0(l1),2(o1)),(1(l2),3(o2)),(1(l2),2(o1)),(0(l1),1(l2)),
          (0(l1), 3(o2))}
9.      l1.remove(o1);
        {(0(l1),2(o1)),(1(l2),3(o2)),(1(l2),2(o1)),(0(l1),1(l2)),
          (0(l1), 3(o2))}
10.     l2.removeFirst();
        {(0(l1),2(o1)),(1(l2),3(o2)),(1(l2),2(o1)),(0(l1),1(l2)),
          (0(l1),3(o2))}
11.     l2.removeAll(l1);
        {(0(l1),2(o1)),(0(l1),3(o2))}
12.     l1.clear();
        { }
13.     l2.add(o1);
        {(1(l2),2(o1))}
14.     l2.set(0, o2);
        {(1(l2),2(o1)), (1(l2),3(o2))}
```

FIGURE 3.14: Collection Disjointness Analysis on Collection Operations

We classify the methods in java.lang.Collection as follows: adds, removes and others.

3.4.2.1 Group of Add Methods

There is a group of "add" methods in classes that implement the list interface. They are *add(), addElement(), addFirst(), addLast(), addAll()*.

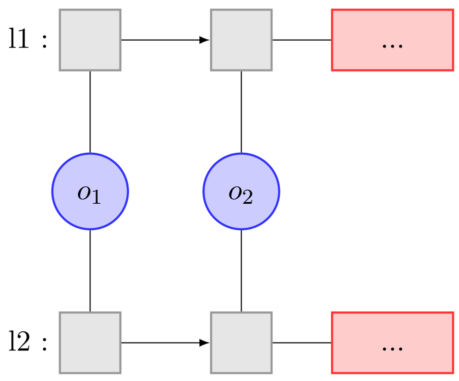
1) add()/addElement(): The most common *add*() method from the *Collection* interface takes an object as its argument: *add(Object)*. If there is a collection l and an object o, the Gen set for $l.add(o)$ is $\{(m(l), n(o))\}$ where m is the object representative of l and n is the object representative of o at the current statement; the Kill set is $\emptyset$.

   There is an additional *add()* method from the *AbstractList* class supplementing the previous one. It is *add(int, Object)* where *int* is the index of the position in the list where the *Object* is to be added. Since we are only interested in whether an object is in the list, not the index of the object, the transfer function for *add(int, Object)* is identical to that of *add(Object)*. The Gen set is also $\{(m(l), n(o))\}$ where m is the object representative of l and n is the object representative of o at the current statement; the Kill set is $\emptyset$. The *addElement()* method from the *Vector* class takes an object o as an argument, in the form of *addElement(Object)*. Our analysis works on this method in exactly the same way our analysis works on the *add(object)* method.

2) addFirst()/addLast(): Both of those methods are from the *LinkedList* class. *addFirst(Object)* adds the object to the head of the list while *addLast(Object)* adds the object to the tail of the list. The transfer functions for these two methods are exactly the same as the regular *add*() method, since the index of the object added does not matter in determining the disjointness of lists. We only care if a list contains an object without considering the position of the object. Therefore, if there is a list l and an object o, the Gen set for *l.addFirst(o)* or *l.addLast()* is $\{(m(l), n(o))\}$ where m is the object representative of l and n is the object representative of o at the current statement; the Kill set is $\emptyset$.

3) addAll(): The *addAll(Collection)* from the *Collection* interface takes a collection as its argument and adds all elements in the collection to the base collection. The transfer function for *addAll*() is more complex than the previous *add*() methods. For two collections $l1$ and $l2$, the Gen set for $l1.addAll(l2)$ consists of all pairs $(m(l1), xkey(x))$ where *xkey(x)* are all elements in containment pairs of the form $(n(l2), xkey(x))$. m is the object representative of $l1$; n is the object representative of $l2$; $xkey$ is the object representative of x which is contained in $l2$. The Kill set is $\emptyset$. In other words, *l1.addAll(l2)* adds all elements in $l2$ to the collection $l1$. There is another *addAll*() method from the *List* interface of the form *addAll(int, Collection)*, which supplements the previous one. Our analysis just ignores the index and analyzes this method exactly just like the plain *addAll()* method, since our analysis only considers the content of the list without considering the order of the contents.

There is one more case to note. Consider two collections $l1$ and $l2$ and two objects $o1$ and $o2$. If $l1$ is aliased with $l2$ and $o1$ is aliased with $o2$, then *l1.add(o1)* not only adds $o1$ to $l1$, but also adds $o2$ to $l1$, $o1$ to $l2$ and $o2$ to $l2$ because of the aliasing relations. This condition shows the benefit of using pairs of object representatives instead of using pairs of objects themselves in the connection. If we use pairs of objects in the data-flow set, to account for aliasing relations we have to add four pairs—$(l1, o1), (l1, o2), (l2, o1), (l2, o2)$—for a single statement *l1.add(o1)*. However, if we use pairs of object representatives of objects, since aliased objects must



Both l1 and l2 contain o1 and o2; l1 and l2 are must-shared.

FIGURE 3.15: Result of the group of *add()* methods after line 8 in Figure 3.14.

have the same object representatives, $(m(l1), n(o1)), (m(l1), n(o2)), (m(l2), n(o1))$ and $(m(l2), n(o2))$ are identical according to the definition in Chapter 3.2. Therefore, for a single statement $l1.add(o1)$, our analysis only adds $(m(l1), n(o1))$ to the connection, which represents all four pairs. Thus, using object representatives makes the connection simpler and easier to read.

Lines 5–8 in Figure 3.14 illustrate our analysis on different types of *add* methods. At line 5, *l1.add(0, o1)* adds $(0(l1), 2(o1))$ to the connection; on line 6, *l2.addFirst(o2)* adds $(1(l2), 3(o2))$ to the connection; at line 7, *l2.addLast(o1)* adds $(1(l2), 2(o1))$ to the connection, and the sharedness pair $(0(l1), 1(l2))$ is generated by the sharedness function and added to the connection (the sharedness function will be explained in Chapter 3.4.4.1); at line 8, *l1.addAll(l2)* adds $(0(l1), 2(o1))$ and $(0(l1), 3(o2))$ to the connection based on the object representative for $l2$ being 1 and because $(1(l2), 2(o1))$ and $(1(l2), 3(o2))$ are in the connection. Since $(0(l1), 2(o1))$ is already in the connection, *l1.addAll(l2)* actually only adds $(0(l1), 3(o2))$ to the connection.

Figure 3.15 depicts graphically the results of the group of *add()* methods for the example in Figure 3.14.

3.4.2.2 Group of Remove Methods

The group of *remove* methods consists of *remove(), removeFirst(), removeLast(), removeElement(), removeElementAt(), removeRange(), removeAll().*

1) remove(): The *remove()* method takes an index or an object as the argument. When it is in the form of *remove(Object)* from the *Set* interface, if there is a set s and an object o, the Gen set for statement *s.remove(o)* is $\emptyset$ and the Kill set is $\{(m(s), n(o))\}$ where m is the object representative of s and n is the object representative of o at the current statement. If the *remove()* method operates on collections other sets, the Kill set is $\emptyset$ since one object may have multiple occurrences in the collection, and our analysis cannot track the number of occurrences of the object in the collection. When $remove()$ takes an index as the argument and intends to remove the element of the given index, our analysis can do nothing, since our analysis does not know the contents of the collection. Therefore, if there is a collection l and an integer i, the Gen set for *l.remove(o)* is $\emptyset$ and the Kill set is also $\emptyset$.

2) removeFirst()/removeLast(): Both methods are from the *LinkedList* class. The *removeFirst()* method removes the first element of the list and the *remove-Last()* method removes the last element of the list. Transfer functions for these two methods are similar to the previous *remove()* method that takes an integer as the argument: since we do not know the order of elements in the list during execution time, our analysis can not do anything. If there is a list l, the Gen set for *l.removeFirst()* or *l.removeLast()* is $\emptyset$ and the Kill set is also $\emptyset$.

3) removeElement()/removeElementAt()/removeRange(): The first two methods are from the *Vector* class. *removeElement(Object)* takes an object as its argument and removes the first occurrence of this object from the vector. Our analysis can not track the number of occurrences of this object in the vector so we do not know whether the vector still contains this object after the *removeElement* statement. Since our analysis only contains false negatives, it does nothing to the connection after this method. $removeElement(int)$ takes an integer as the argument and removes the object of the given index from the vector. Since our analysis does not know the sequence of the contents of the vector, it does nothing to the connection after this method. Similarly, our analysis does nothing to the connection after the method *removeRange(int, int)* from the *AbstractList* class which removes elements from a beginning index to an end index. The Gen set and Kill set for these three methods are $\emptyset$.

4) removeAll(): The *removeAll()* method from the *Collection* interface takes a collection as the argument and removes all elements in the collection from the base collection. If there are two collections $l1$ and $l2$, the Gen set for *l1.removeAll(l2)* is $\emptyset$ and the Kill set consists of all pairs $(m(l1), xkey(x))$ where *xkey(x)* represents all elements in containment pairs in the form of $(n(l2), xkey(x))$. m is the object representative of $l1$; n is the object representative of $l2$; $xkey$ is the object representative of x which is contained in $l2$. The kill set also contains a sharedness pair $(m(l1), n(l2))$. Note that *removeAll* makes $l1$ and $l2$ not-may-shared if $l1$ and $l2$ are connected but not identical previously.

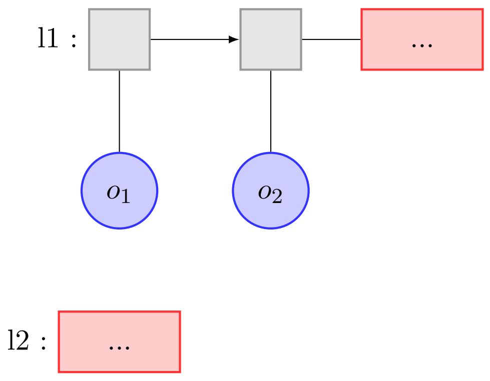
Appendix 7 shows that programmers use *remove* operations less frequently than using *add* operations. Although our transfer functions do not affect the connection for most *remove* operations except *remove()* on sets and *removeAll*, the approximation is still reasonable due to the less frequent use of *remove* methods.

Lines 9–11 in Figure 3.14 show our analysis on different types of *remove* methods. At line 9, *l1.remove(o1)* does nothing to the connection since we do not know the number of occurrences of $o1$ in $l1$. At line 10, *l2.removeFirst()* does nothing to the connection since we do not know the order of the elements in the list. At line 11, *l2.removeAll(l1)* removes containment pairs $(1(l2), 2(o1))$ and $(1(l2), 3(o2))$ from the connection since there are containment pairs $(0(l1), 2(o1))$ and $(0(l1), 3(o2))$ showing $l1$ contains $o1$ and $o2$. Our analysis also removes the sharedness pair $(0(l1), 1(l2))$ from the connection since they no longer have references pointing to the same heap location after this statement.

Figure 3.16 depicts graphically the results of the group of *remove()* methods for the example in Figure 3.14.

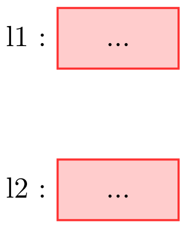
3.4.2.3 Other Methods

The *clear()* method from the *Collection* interface removes all elements from the collection. There is a *removeAllElements()* method from the *Vector* class having similar functionality. For a collection l, the Gen set for *l.clear()* is $\emptyset$ and the Kill set contains all containment pairs $(m(l), okey(o))$ and all sharedness pairs $(m(l), ckey(collection))$ where m is the object representative of l, $okey$ is the object representative of o and $ckey$ is the object representative of *collection* before this statement. If there is a



l1 contains o1 and o2; l1 and l2 are not-may-shared.

FIGURE 3.16: Result of the group of *remove()* methods after line 11 in Figure 3.14.



l1 and l2 are not-may-shared.

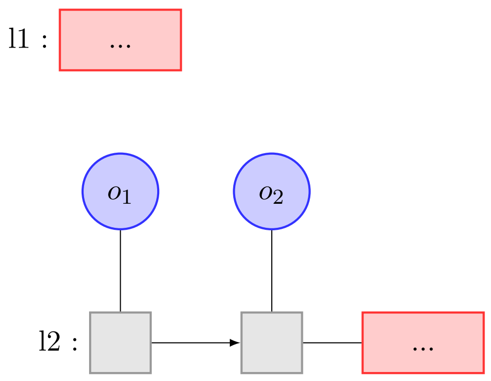
FIGURE 3.17: Result of the *clear()* method after line 12 in Figure 3.14.

vector v, the Gen set for *v.removeAllElements()* is $\emptyset$ and the Kill set contains all containment pairs $(m(v), okey(o))$ and all sharedness pairs $(m(v), ckey(collection))$.

Line 12 in Figure 3.14 shows the result of our analysis on the *clear* method. We remove containment pair $(0(l1), 2(o1))$ and $(0(l1), 3(o2))$ from the connection. Since there is no sharedness pair containing $l1$ in the connection before this statement, our analysis does not remove any sharedness pairs in this example.

Figure 3.17 depicts graphically the result of the *clear()* method for the example in Figure 3.14.

The *set* method from the *List* interface takes an integer and an object as the argument. It replaces the element at the given index in the list with the specified element and retains the replaced element. Since our analysis does not know the sequence of the contents of the list, the identity of the element be removed from the list is unknown. Therefore, our analysis treats the *set* method exactly like the *add* method.For a list l, an integer *index* and an object o, the Gen set for *l.set(index, o)*



l2 contains o1 and o2; l1 and l2 are not-may-shared.

FIGURE 3.18: Result of the *set()* method after line 14 in Figure 3.14.

is $\{m(l), okey(o)\}$ and the Kill set is $\emptyset$ where m is the object representative of l and $okey$ is the object representative of o before this statement.

Line 14 in Figure 3.14 shows our analysis on the *set* method. We add the containment pair $(1(l2), 3(o2))$ to the connection. Although, at run time, $o1$ is the first element in $l2$, and should be replaced by $o2$, our analysis does not know this. Therefore, our analysis will keep the containment pair $(1(l2), 2(o1))$ in the connection. Figure 3.18 depicts graphically the result of the *set()* method for the example in Figure 3.14.

3.4.3 Method Calls

It is usually difficult to analyze method calls in static analysis since we do not know what the callee method does. We must use a call graph to identify potential callee methods. To best approximate changes to the connection resulting from method calls, our collection disjointness analysis tries to read annotations for the callee method, which can be either user-defined annotations written by programmers or generated annotations computed by our analysis itself based on pre-annotations. We use SPARK's call graph [10] to determine the set of possible callee methods. In chapter 3.4.3.1, we will focus on a single method, showing how our analysis reads user defined post-annotations or generated post-annotations and how to compute generated post-annotations from user defined pre-annotations. In chapter 3.4.3.2, we will show how we organize the order of the methods to be analyzed.

```
1.      public @interface ExampleAnnotation {
2.              String pre();
3.              String post();
4.              String generatedPostAnnotations();
5.      }
6.
7.      @ExampleAnnotation(pre = "NotMayShared(l1, l2); MustShared(l2, l3)",
                post = "", generatedPostAnnotation = "MayShared(l1, l2);
                NotMayShared(l2, l3)")
8.      public static void foo (List l1, List l2, List l3) {
9.              l1 = l2;
10.             l3.clear();
11.     }
```

FIGURE 3.19: Annotations Example

3.4.3.1 Analyze Annotations

Annotations provides information about a program without affecting the operation of the program. Static analysis tools such as Soot can exploit annotations by reading their contents to facilitate the analysis process. Since programmers can manually provide modify annotations, annotations enable communication between programmers and static analysis tools.

The annotations we use for our disjointness analysis are multi-valued annotations: they contain multiple values. We also define a domain specific annotation language to represent disjointness information. First, the name and order of values are mandatory. There are three values in the annotation. They are all String types: *pre* is the first value representing the user defined pre-annotations; *post* is the second value representing the user defined post-annotations; *generatedPostAnnotation* is the last value representing the generated post-annotations by our analysis, which is optional. The content of the value must be string nodes in the following three forms: *NotMayShared(l1, l2)*, *MayShared(l1, l2)* and *MustShared(l1, l2)*. A value can contain multiple nodes, separated by ";". We choose these three forms because our analysis only needs to know the disjointness relations after method calls. Figure 3.19 shows an example of annotations of a method foo.

Lines 1–5 show the creation of the annotation *ExampleAnnotation*. Lines 7–8 shows the use the annotation *ExampleAnnotation* of method *foo*. The pre-annotations of *foo* are *NotMayShared(l1, l2)* and *MustShared(l2, l3)*; the user defined post-annotations of *foo* are empty; the generated post-annotations are *MayShared(l1,l2)* and *NotMayShared(l2, l3)*.

A method call in our Jimple intermediate representation implements the Jimple *InvokeExpr* interface. When our analysis finds a method call, it tries to find callee methods in the call graph, then reads the annotations of these methods and changes the content of the connection according to the effect of the method call recorded in the post-annotations (either user defined post-annotations or generated post-annotations). We parse the post-annotations using Java regular expressions.

We describe one last preliminary step. Methods take formal parameters, which are replaced by actual parameters when the method is called. The problem is that annotations of the method only indicate the functionality of the method using formal parameters. In figure 3.19, the pre-annotation tells the not-may-sharedness between parameters $l1$ and $l2$ and the must-sharedness between $l2$ and $l3$. However, if in another method, we want to analyze calls to method *foo*, with locals a, b, c as actual parameters, by reading the annotations our analysis knows nothing about how *foo* take effect on lists a, b and c. Therefore we need to map a to $l1$, b to $l2$ and c to $l3$. Then when the generated post-annotations show *MayDisjoint(l1, l2)*, our analysis knows a and b are may-shared after calling *foo(a, b,c)*. The mapping process is simple. We create a list of locals the method takes as argument in order and a list of parameters of the method. We map elements from the first list to the second list one by one. When our analysis read the post-annotations, it finds the disjointness relations between formal parameters and replaces every formal parameter with the actual parameter mapped to it.

If the post-annotation shows a may-shared or must-shared relation between two locals, we add the sharedness pair of these two locals to the connection. If the post-annotation show a not-may-shared relation between two locals, we remove the pair of these two locals from the connection. Let us recall the annotation example in Figure 3.19. If the analyzing method calls *foo(a, b,c)*, our analysis adds sharedness pair $(m(a), n(b))$ to the connection and remove pair $(n(b), p(c))$ from the connection, where m, n, p are object representatives of a, b, c respectively.

Usually programmers cannot write all post-annotations of all methods since there are too many annotations to write. Our analysis can compute the generated post-annotation if programmers can provide the pre-annotation, which halves the number of annotations required. To generate post annotations, we proceed as follows:

First, before analyzing a method, our analysis checks whether programmers provides pre-annotations for that method. If so, our analysis reads the contents of the pre-annotation. If it mentions the may-sharedness or must-sharedness relation between two collections, our analysis adds a sharedness pair of the two collections to the connection. If it mentions the not-may-sharedness relation, our analysis removes the sharedness pair from the connection.

For example, if the pre-annotation of the method contains a string *MayDisjoint(l1, l2)* where $l1$ and $l2$ are parameters of this method, the sharedness pair $\{(\text{Unknown}(l1), \text{Unknown}(l2))\}$ should be added to the connection. Our analysis reads pre-annotations before analyzing the first Jimple statement of the annotated method. Jimple presents its first statement of any methods with parameter(s) as a parameter assignment statement. Since our transfer functions do nothing to an assignment statement, as mentioned in chapter 3.4.1, adding sharedness pairs according to pre-annotations after the assignment statement only affects the assigned parameter. Thus, the object representative of one collection in the sharedness pair may be updated from Unknown to its value after the assignment statement.

For the above example, if collection $l1$ is the first parameter of the annotated method and its object representative is 0 after the parameter assignment, the sharedness pair added to our data-flow set is $\{0(l1), \text{Unknown}(l2))\}$ instead. However, note that at least one object representative of these two collections is still Unknown at this point. Therefore, we will use an *update* operation to update the value of the object representatives when they are available. The *update* operation will be introduced in chapter 3.4.4.2. After adding the sharedness pairs from the annotations, we apply the standard data-flow analysis for the method as described in this chapter.

Finally, when the analysis finishes computing the connection at the last statement of the method, the analysis must verify or generate post-annotations. To do so, it reads the connection to find collections appearing in the pre-annotations and generates post-annotations, verifying that the connection applies the stated post-annotations. For example, suppose the pre-annotation contains a declaration *MayShared(l1, l2)*, where $l1$ and $l2$ are method parameters. If at the end of the method, the connection does not contain the pair $(m(l1), n(l2))$, indicating that $l1$ and $l2$ are not-may-shared. Then the generated post-annotation should contain *NotMayShared(l1, l2).* Or, if the connection does contain the pair $(m(l1), n(l2))$, meaning $l1$ and $l2$ are may-shared, the generated post-annotation would contain *MayShared(l1, l2).*

Since our analysis reads developer-provided-post-annotations first then the generated-annotations, the effect of reading generated-annotations will overide the effect of reading developer-provided-post-annotations. For instance, if developer-provided-post-annotations state that collections $l1$ and $l2$ are not-may-shared while generated-annotations state that $l1$ and $l2$ are may-shared, our analysis will remove the pair of $l1$ and $l2$ first according to the developer-provided-post-annotations and then add this pair back according to the generated-annotations. In this way, the generated-annotations automatically verify the developer-provided-post-annotations. However, developers can avoid the automatic verification by adjusting the contents of the pre-annotations. As we mentioned above, our analysis only generates annotated pairs that appeares in the pre-annotations. Therefore, developers can keep the provided-post-annotations, by not including the pairs that are in the provided-annotations in pre-annotations. Figure 4.5 shows an example of this condition we met during the tests on a benchmark. We will present this example in more detail in Chapter 4.2.

3.4.3.2 Method Ordering

If method A calls method B, and method B calls method C, what is the order of these methods we need to analyze if we are analyzing method A? We perform a topological sort on methods in the call graph depending on the caller-callee relations to make sure our analysis analyzes a callee prior to its caller. Our analysis stores the result of the topological sort in a list, *sortedMethods*. If there is a cycle in our call graph, we just break the back edge to make our graph acyclic and store the whole cycle in another list, *cycles*.

Then our analysis tries to analyze all methods in the *sortedMethods* list in order. If a method belongs to any cycle in the *cycles* list, we keep analyzing all methods in the cycle until no generated post-annotations of all methods in the cycle change.

3.4.4 Additional Operations

Besides the above operations, our analysis employs two additional operations. The *Sharedness* operation generates sharedness pairs if two collections contains the same object or aliasing objects. The *Update* operation updates objects having unknown object representatives, representing that this object aliases everything. In chapter

3.4.4.1, we will show how our *Sharedness* operation works; in chapter 3.4.4.2, we will show how *Update* operation works.

3.4.4.1 Sharedness

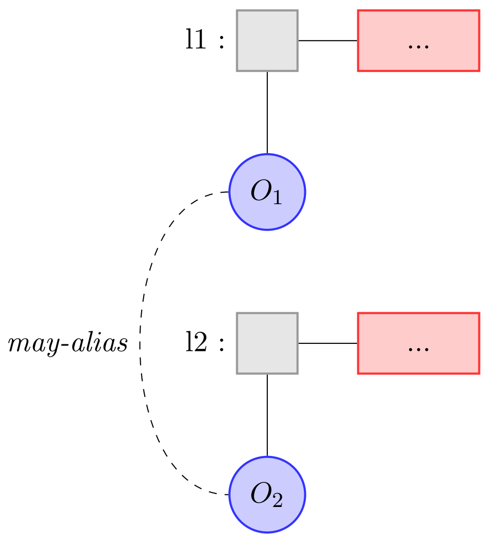
The sharedness pairs describe disjointness relations between collections. Some of the sharedness pairs are created by reading annotations of methods called as seen in chapter 3.4.3. Besides that, they are mainly created by the sharedness operations, which add pairs of lists to the connection when they are may-shared or must-shared, based on the contents of the lists. Recall the definition of disjointness relations from chapter 3.1: if two collections $l1$ and $l2$ contain at least one object that must alias or at least one collection that is must-shared, we say $l1$ and $l2$ are must-shared; if two collections $l1$ and $l2$ contain at least one object that may alias or at least one collection that is may-shared, we say $l1$ and $l2$ are may-shared; if two collections $l1$ and $l2$ contain no objects that may alias and no collections that are may-shared, we say $l1$ and $l2$ are not-may-shared. By this definition, the *Sharedness* operation adds a pair of collections either if they contain at least one object that may alias or at least one collection that is may-shared or if they contain at least one object that must alias or at least one collection that is must-shared.

The implementation of the *Sharedness* operation works in the following way:

1) After each statement, go through every containment pair in the connection and check it with other containment pairs.

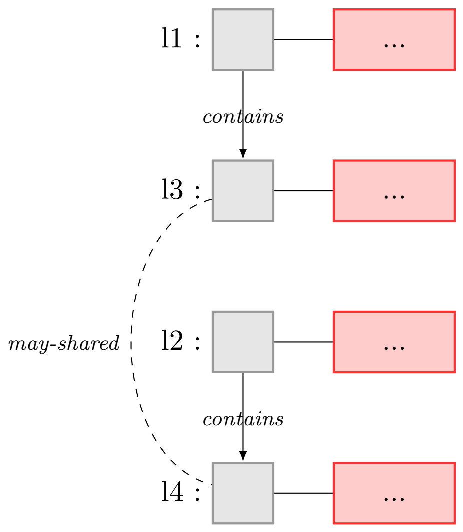
2) *Case 1*: For containment pairs $(m(l1), o1key(o1))$ and $(n(l2), o2key(o2))$ where $l1$ and $l2$ are collections, $o1$ and $o2$ are objects, $m, n, o1key, o2key$ are object representatives of $l1, l2, o1, o2$ respectively, if $o1$ and $o2$ may alias or must alias, a sharedness pair $(m(l1), n(l2))$ is added to the connection (Figure 3.20).

3) *Case 2*: For containment pairs $(m(l1), p(l3))$ and $(n(l2), q(l4))$ where $l1, l2, l3$ and $l4$ are collections, m, n, p, q are object representatives of $l1, l2, l3, l4$ respectively, if there is an sharedness pair $(p(l3), q(l4))$ in the connection, meaning $l3$ and $l4$ are may-shared or must-shared, a sharedness pair $(m(l1), n(l2))$ is added to the connection (Figure 3.21). Note that all *external collections* are may-shared with others, we also check if $p(l3)$ or $q(l4)$ is in the list of *external collections*. If either $p(l3)$ or $q(l4)$ is an *external collection*, the sharedness pair $(m(l1), n(l2))$ should also be added to the connection.



l1 contains o1 and l2 contains o2. If o1 and o2 may alias, l1 and l2 are may-shared.

FIGURE 3.20: Sharedness operation: Case 1



l1 contains l3 and l2 contains l4. If l3 and l4 are may-shared, then l1 and l2 are may-shared.

FIGURE 3.21: Sharedness operation: Case 2

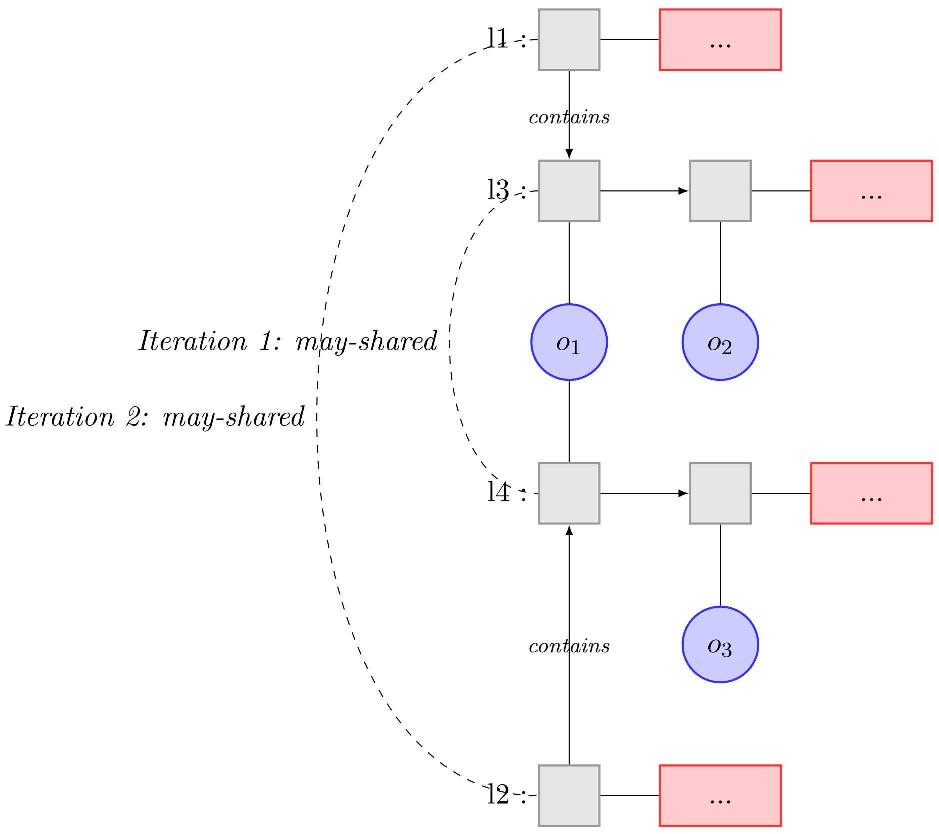
4) *Case 3*: If there is no new sharedness pair added to the connection after going through every containment pairs in the connection, stop. Otherwise, go through every containment pair in the connection again until no new sharedness pair is added to the connection.

The following example motivates our analysis. Consider containment pairs *(m(l1),p(l3))*, *(n(l2),q(l4))*, *(p(l3),o1key(o1))*, *((p(l3),o2key(o2))*, *(q(l4),o1key(o1))* and *(q(l4),o3key(o3))* in the connection where $l1$, $l2$, $l3$ and $l4$ are collections, $o1$, $o2$ and $o3$ are objects and m, n, p, q, *o1key*, empho2key, empho3key are object representatives of $l1$, $l2$, $l3$, $l4$, $o1$, $o2$, $o3$ respectively. Since both $l3$ and $l4$ contain object $o1$, the sharedness pair $((p(l3), q(l4))$ should be added to the connection first. Then another sharedness pair $(m(l1), n(l2))$ should also be added since $l1$ contains $l3$, $l2$ contains $l4$ and $l3, l4$ are may-shared or must-shared (Figure 3.22). However, if our analysis has already gone through containment pairs $(m(l1), p(l3))$ and $(n(l2), q(l4))$, the second sharedness pair will not be added to the connection during the first iteration. That is why the *Sharedness* operation keeps running through the connection until no new sharedness pairs are added to the connection.

3.4.4.2 Update object representatives

Unknown object representatives are arise when analyzing methods with annotations. when our analysis reads the pre-annotation of a method and adds sharedness pairs of collections to the connection, the initial object representatives of these collections is Unknown. We should update the object representatives of these collections when they are available. Since our annotations only contain parameters of the annotating method. These collections with Unknown object representatives are all parameters of the current method. Therefore the object representatives a collection will be available after parameter assignment of this collection. Thus this update operation is simple and obvious: it update the Unknown object representative of a annotated collection after the parameter assignment statement of this collection.

Figure 3.23 shows how our update operation works. In this example we want to analyze the *main()* method in the *UpdateTest* class. The *main()* method calls an annotated method *foo()*, taking two lists as arguments. As presented in chapter 3.4.3.2, a callee method must be analyzed before its calling method, our analysis works through the *foo()* method first. Lines 1–7 show how our analysis works on



l3 and l4 are may-shared at the first step. l1 and l2 are may-shared at the second step.

FIGURE 3.22: Sharedness operation: Case 3

the Jimple code of the *foo()* method. As presented in Chapter 3.4.3.1, our analysis adds sharedness pairs according to the pre-annotations after the first parameter assignment statement for convenience. Therefore, after line 4, a sharedness pair {*0(l1)*, Unknown(*l2*))} is added to the connection because of the pre-annotation *MustShared(l1, l2)*. Since $l1$ has already been updated, our update operation does nothing on this statement. After line 5, our update operation updates the object representative of $l2$ from Unknown to -1. Note that the object representatives for parameters are decreasing negative integers from 0. For example, the object representative is *0* for the first parameter, *-1* for the second parameter and *-2* for the third parameter. After line 6, the sharedness pair $(0(l1), -1(l2))$ is removed from the connection due to the *clear()* operation on $l1$, as presented in chapter 3.4.2.3.

```
public class UpdateTest {

        public static void main (String [] args) {
                List a = new LinkedList();
                List b = new LinkedList();
                Object o = new Object();
                a.add(o);
                b.add(o);
                foo(a,b);
        }

        @Value_Annotation(pre = "MustShared(l1, l2)",post = "")
        public static void foo (List l1, List l2) {
                l1.clear();
        }

        public @interface Value_Annotation {
                String pre();
                String post();
        }
}


1.      public static void foo(java.util.List, java.util.List)
2.      {
3.              java.util.List l1, l2;
                { }
4.              l1 := @parameter0: java.util.List;
                {(0(l1), UNKNOWN(l2))}
5.              l2 := @parameter1: java.util.List;
                {(0(l1), -1(l2))}
6.              interfaceinvoke l1.<java.util.List: void clear()>();
                { }
7.              return;
                { }
        }
```

FIGURE 3.23: Updating Object Representatives Example

3.5 Initial Values

The entry initial value represents the contents of the connection before the first statement at the beginning of the method. Local variables at the beginning of a method have no value until they are initialized. If the initialization gives local a newly-instantiated list, then the new list contains no object. If it loads from a field, we record that we do not know about the contents of that field. When we create a new object, if it does not appear in the annotation, we consider it is fresh and does not related to any exist objects in the heap. Then entry initial value is set to be an empty set. If the analyzing method has pre-annotations, our analysis reads these pre-annotations, adds sharedness pairs to the connection to get the entry initial value.

The new initial value represents the contents of the connection copied to the in-set and out-set of every statement. Since we keep tracking a single connection through the whole method at every program point, the new initial value is set to be this connection.

Chapter 4

Experimental Results

In chapter 3, we provided a detailed description of our collection disjointness analysis and included details about our implementation. In this chapter, we describe our experience using our implementation of our collection disjointness analysis on two open source Java projects.

4.1 Benchmarks

SableCC [4] is an open source parser generator in Java designed by Etienne Gagnon. It is used to build compilers, interpreters and other text parsers by generating object-oriented frameworks. SableCC automatically generates intuitive strictly-typed abstract syntax trees and tree walkers. It also employs a technique that separates machine-generated code and user-written code, which shortens the development cycle. JavaCC [6] (Java Compiler Compiler) is a second open source parser generator in Java. It generates top-down parsers from a formal grammar written in Extended Backus-Naur Form (EBNF). It includes a tree builder, JJTree, which builds trees from the bottom up. Note that we analyze the parser generators rather than any generated parsers. Figure 4.1 summarizes benchmark characteristics.

4.2 Results

Figure 4.2 shows the results of our experiments, including the execution time of our collection disjointness analysis for each of the benchmarks. We ran the tests on

	sableCC	**javaCC**
Version	3.2	4.2
Lines of Code	35408	48162
Number of Classes	285	155

Figure 4.1: Benchmark Information

	sableCC	**javaCC**
Methods analyzed	1824	1066
Total time (seconds)	213	278
Peak memory usage (Mb)	772.86	1754.89
Methods containing Collection operations	549	134
Methods taking Collections as arguments	46	28
Annotated methods	1	6
Not-May-Shared pairs of Collections found	12	0

Figure 4.2: Experimental Results

a desktop computer with a 3.20 GHz Intel Pentium D with 3.5GB memory. Our analysis analyzed 1824 methods in SableCC and 1066 methods in javaCC, taking 213 seconds and 278 seconds respectively. Among those methods, 549 methods in SableCC contain Collection operations while 134 methods in javaCC contain Collection operations. Of these methods, SableCC contains 267 methods and javaCC contains 45 methods, which manipulate only a single collection. In such methods contains no not-may-shared relations, since only one collection appears in it.

4.2.1 Discussion on Annotations

Figure 4.2 also shows that 46 methods in SableCC and 28 methods in javaCC take Collections as arguments. We added pre-annotations to all methods that take at least two collections as arguments. Thus we annotated 1 method in SableCC and 6 methods in javaCC. The others take only one collection as an argument. We also added selected post-annotations to avoid bad approximation from the static analysis. Figures 4.3–4.5 present an example of how pre-annotations and developer-provided post-annotations contribute to our analysis results. Both methods are in the *LookaheadWalk* class in *JavaCC*. Method *genFollowSet()* (lines 13–18) calls the *listSplit()* method (lines 1–12), which splits a given list into two parts by a mask. These two parts are stored separately into two lists.

```
1.      private static void listSplit(List toSplit, List mask,
                                      List partInMask, List rest) {
2.        OuterLoop:
3.        for (int i = 0; i < toSplit.size(); i++) {
4.                for (int j = 0; j < mask.size(); j++) {
5.                        if (toSplit.get(i) == mask.get(j)) {
6.                                partInMask.add(toSplit.get(i));
7.                                continue OuterLoop;
8.                        }
9.                }
10.               rest.add(toSplit.get(i));
11.       }
        {(-3(rest),-2(partInMask)),(-3(rest),6(temp\$7)),
        (-2(partInMask), 5(temp\$6))}
12.     }

13.     public static List genFollowSet(List partialMatches,
                                      Expansion exp, long generation) {
         ...
         {      }
14.      List v = ...
15.      List v1 = new ArrayList();  //v1 has an object representative 5
16.      List v2 = new ArrayList();  //v2 has an object representative 6
17.      listSplit(v, partialMatches, v1, v2);
         {      }
         ...
18.     }
```

FIGURE 4.3: Test Example without Annotations.

Figure 4.3 shows our analysis on the method call in the *genFollowSet()* method without annotations. Our analysis analyzes the *listSplit()* method first due to the caller-callee relation and finds useful results: at the end of the *listSplit()* method, the connection contains two containment pairs *(-3(rest),6(temp$7))* and *(-2(partInMask), 5(temp$6))*, where *temp$7* and *temp$6* are local variables introduced by Jimple, and a sharedness pair *(-3(rest),-2(partInMask))*, indicating that *rest* and *partInMask* are may-shared at the end of the callee method. However, without annotations, our intraprocedural analysis of *genFollowSet()* does not know about the behaviour of its callee, *listSplit()*. Naively, one might say that the connection after line 17 does not change and is still an empty set. However, this naive result is incorrect, since we may conclude wrong disjointness relations from it such as v, *partialMatches*, $v1$ and $v2$ are not-may-shared. Conservatively, we would have to say that all collections are may-shared after every method call. Therefore, we need to add annotations to usefully analyze these methods.

Figure 4.4 shows our analysis on the same example with pre-annotations. For method *listSplit()*, we notice that it uses list *mask* to divide list *toSplit* into two parts: *partInMask* and *rest*. Therefore, before the method call, list *mask* and list *toSplit* are

```
0.  @list_ano(pre="MayShared(toSplit, mask); NotMayShared(partInMask, rest)",
                post="")
1.  private static void listSplit(List toSplit, List mask,
                                     List partInMask, List rest) {
    {(0(toSplit),UNKNOWN(mask)}
2.        OuterLoop:
3.        for (int i = 0; i < toSplit.size(); i++) {
4.                for (int j = 0; j < mask.size(); j++) {
5.                        if (toSplit.get(i) == mask.get(j)) {
6.                               partInMask.add(toSplit.get(i));
7.                               continue OuterLoop;
8.                        }
9.                }
10.               rest.add(toSplit.get(i));
11.       }
    {(-3(rest),-2(partInMask)),(-3(rest),6(temp\$7)),
    (-2(partInMask), 5(temp\$6)),(0(toSplit),-1(mask))}
12.     }

13.  public static List genFollowSet(List partialMatches,
                                     Expansion exp, long generation) {
        ...
14.     List v = ...                         //v has an object representative 3
15.     List v1 = new ArrayList();      //v1 has an object representative 5
16.     List v2 = new ArrayList();      //v2 has an object representative 6
17.     listSplit(v, partialMatches, v1, v2);
        {(3(v),0(partialMatches)),(5(v1), 6(v2))}
        ...
18.     }
```

Figure 4.4: Test Example with only Pre-Annotations.

may-shared since they may contain the same objects, while *partInMask* and *rest* are not-may-shared since they are used to store different parts of *toSplit*. Thus, we add pre-annotations "MayShared(toSplit, mask); NotMayShared(partInMask, rest)" at line 0. Our analysis adds the sharedness pair *(0(toSplit),Unknown(mask))* at the beginning of this method and the connection after line 11 contains one more sharedness pair *(0(toSplit),-1(mask))*. Since we have pre-annotations for this method, our analysis will generate post-annotations based on the connection after line 11. The generated annotations in this example are "MayShared(toSplit, mask); MayShared(partInMask, rest)" since the connection contains these two sharedness pairs, which also appear in the pre-annotations. For the caller method *genFollowSet()*, our analysis reads the generated annotations for method call *listSplit* at line 17 and adds two sharedness pairs *(3(v),0(partialMatches))* and *(5(v1), 6(v2))* to the connection. The pre-annotations therefore improve the analysis of *listSplit()* but not *genFollowSet()*. However, the result after line 17 is still incorrect. Because *v* is divided into two parts and stored in *v1* and *v2*, *v* and *v1* and *v* and *v2* should have been may-shared. But from the connection after line 17, we can conclude that *v* and *v1* are not-may-shared, as are *v* and *v2*. We provide post-annotations to solve

```
0.  @list_ano(pre="MayShared(toSplit, mask); NotMayShared(partInMask, rest)",
             post="MayShared(toSplit, partInMask); MayShared(toSplit, rest);
                   MayShared(mask, partInMask)")
1.  private static void listSplit(List toSplit, List mask,
                                    List partInMask, List rest) {
2.      OuterLoop:
3.      for (int i = 0; i < toSplit.size(); i++) {
4.          for (int j = 0; j < mask.size(); j++) {
5.              if (toSplit.get(i) == mask.get(j)) {
6.                  partInMask.add(toSplit.get(i));
7.                  continue OuterLoop;
8.              }
9.          }
10.         rest.add(toSplit.get(i));
11.     }
    {(-3(rest),-2(partInMask)),(-3(rest),6(temp\$7)),
    (-2(partInMask), 5(temp\$6)),(0(toSplit),-1(mask))}
12.    }

13.  public static List genFollowSet(List partialMatches,
                                    Expansion exp, long generation) {
       ...
14.    List v = ...                      //v has an object representative 3
15.    List v1 = new ArrayList();    //v1 has an object representative 5
16.    List v2 = new ArrayList();    //v2 has an object representative 6
17.    listSplit(v, partialMatches, v1, v2);
       {(3(v),5(v1)),(3(v),6(v2)),(0(partialMatches),5(v1)),
       (3(v),0(partialMatches)),(5(v1), 6(v2))}
       ...
18.    }
```

FIGURE 4.5: Test Example with all annotations

this problem.

Figure 4.5 shows our analysis with both pre-annotations and developer-provided-post-annotations. For method *listSplit*, we notice that at the end of the method, *toSplit* and *partInMask* and *toSplit* and *rest* are may-shared since *toSplit* is possibly stored in both *partInMask* and *rest*. *mask* and *partInMask* are also may-shared since all masked elements in *toSplit* determined by *mask* is possibly stored in *partInMask*. Therefore, we add "MayShared(toSplit, partInMask); MayShared(toSplit, rest); MayShared(mask, partInMask)" to the developer-provided post-annotations at line 0. Therefore, for the method call at line 17, our analysis adds sharedness pairs *(3(v),5(v1)),(3(v),6(v2))*, and *(0(partialMatches),5(v1))* to the connection according to the three statements in developer-provided post-annotations respectively, showing the may-shared relations between *v* and *v1*, *v* and *v2*, and *partialMatches* and *v1*.

Note that for methods that take a single collection as arguments, we conservatively treat this collection as an external collection, which is may-shared with all collections.

For methods that takes more than one collection as arguments (rare in practise), the developer must provide pre-annotations. Otherwise, our analysis may generate incorrect answers.

4.2.2 Discussions on Experimental Results

We did not find any not-may-shared pairs of collections in javaCC. In SableCC, we find 12 not-may-shared pairs of collections. 8 are in the *createParser()* method from the *org.sablecc.sablecc.GenParser* class; the rest 4 are in the *createLexer()* method from the *org.sablecc.sablecc.GenLexer* class. We have verified that these collections are not-may-shared through manual inspection as well as simple tests. Figures 4.6–4.7 and 4.10 show the Jimple code for the above methods. For convenience, we only show the relevant lines of code.

4.2.2.1 *createParser()*

Figure 4.8 depicts graphically the results of our analysis on the *createParser()* method. After analyzing the connection and removing all normalization pairs at the end of this method, we find nine containment pairs *(79(r170),949(r173))*, *(949(r173),962($205))*, *(949(r173),1341($r244))*, *(949(173),1355($r226))*, *(949(173),1398($r254))*, *(87(r277),362(r279))*, *(362(r279),371($r303))*, *(362(r279),562($r319))*, *(99(r351),134($r386))*. None of the above collections appears in the list of external collections. We conclude that there are 8 not-may-shared pairs: *(79(r170),(99(r351))*, *(79(r170),87(r277))*, *(79(r170),362(r279))*, *(949(r173),(99(r351))*, *(949(r173),87(r277))*, *(949(r173),362(r279))*, *(99(r351),87(r277))*, *(99(r351),362(r279))*.

Figures 4.6–4.7 show the Jimple code of the *createParser()* method.

1) Vector *r173* is instantiated at lines 9–11. Then four arrays of int *$r205*, *$r205*, *$r205* and *$r205* are added to *r173* at lines 13, 15, 17 and 19 respectively. Then *r173* is added to another vector *r170* at line 21. After that *r170* calls the method *elements()* to return an enumeration of its contents at line 24. After that the code makes no more changes to vectors *r170* and *r173* and just writes the contents to the output stream.

```
private void createParser()
{      ...
       java.util.Vector r170, r173, r277, r279, r351;
       ...
    label25:
       $r169 = new java.util.Vector;
       specialinvoke $r169.<java.util.Vector: void <init>()>();
       r170 = $r169;
       ...
    label26:
       $r172 = new java.util.Vector;
       specialinvoke $r172.<java.util.Vector: void <init>()>();
       r173 = $r172;
       ...
     label33:
       virtualinvoke r173.<java.util.Vector:
               void addElement(java.lang.Object)>(\$r205);
       ...
     label35:
       virtualinvoke r173.<java.util.Vector:
               void addElement(java.lang.Object)>(\$r226);
       ...
     label36:
       virtualinvoke r173.<java.util.Vector:
               void addElement(java.lang.Object)>(\$r244);
       ...
     label37:
       virtualinvoke r173.<java.util.Vector:
               void addElement(java.lang.Object)>(\$r254);
       ...
     label39:
       virtualinvoke r170.<java.util.Vector:
               void addElement(java.lang.Object)>(r173);
       ...
     label40:
       $i36 = virtualinvoke r170.<java.util.Vector: int size()>();
       r265 = virtualinvoke r170.<java.util.Vector:
               java.util.Enumeration elements()>();
       ...
     label46:
       $r276 = new java.util.Vector;
       specialinvoke $r276.<java.util.Vector: void <init>()>();
       r277 = $r276;
       ...
     label47:
       $r278 = new java.util.Vector;
       specialinvoke $r278.<java.util.Vector: void <init>()>();
       r279 = $r278;
       ...
     label52:
       virtualinvoke r279.<java.util.Vector:
               void addElement(java.lang.Object)>($r303);
       ...
     label53:
       virtualinvoke r279.<java.util.Vector:
               void addElement(java.lang.Object)>($r319);
       ...
     label55:
       virtualinvoke r277.<java.util.Vector:
               void addElement(java.lang.Object)>(r279);
       ...
```

FIGURE 4.6: Jimple code of the *createParser()* method: Part One.

```
39.   label56:
40.     $i58 = virtualinvoke r277.<java.util.Vector: int size()>();
41.     r332 = virtualinvoke r277.<java.util.Vector:
                java.util.Enumeration elements()>();
        ...
42.   label62:
43.     $r350 = new java.util.Vector;
44.     specialinvoke $r350.<java.util.Vector: void <init>()>();
45.     r351 = $r350;
        ...
46.   label69:
47.     virtualinvoke r351.<java.util.Vector:
                void addElement(java.lang.Object)>($r386);
        ...
48.   label71:
49.     $i68 = virtualinvoke r351.<java.util.Vector: int size()>();
50.     r400 = virtualinvoke r351.<java.util.Vector:
                java.util.Enumeration elements()>();
        ...
51.}
```

FIGURE 4.7: Result in the *createParser()* method: Part Two.

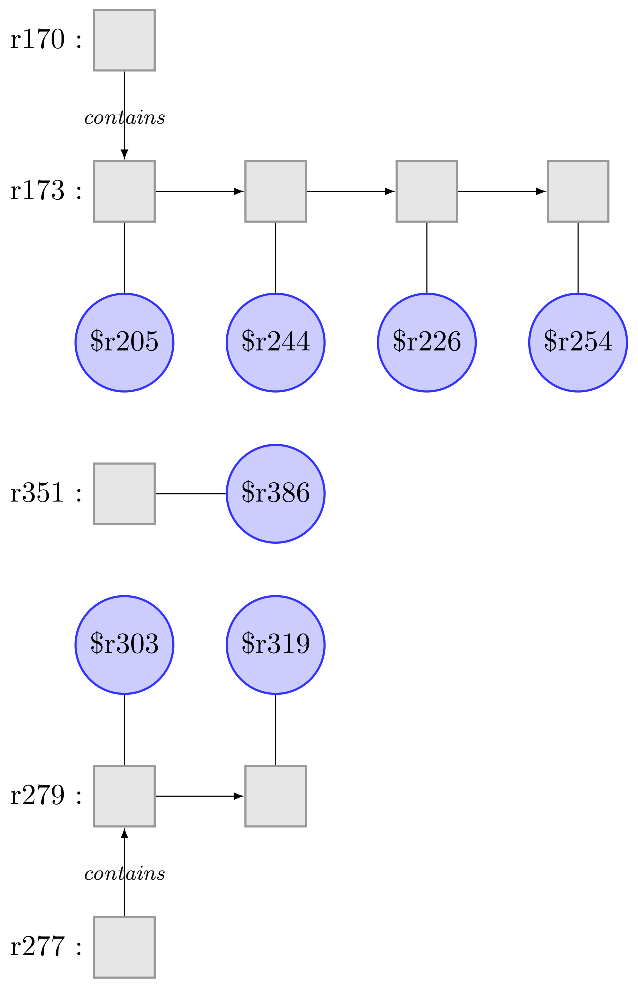
2) Vector *r279* is instantiated at lines 30–32. Then two arrays of int *$r303* and *$319* are added to *r279* at lines 34 and 36 respectively. Then *r279* is added to another vector *r277* at line 38. After that *r277* calls the method *elements()* to return an enumeration of its contents at line 41. After that the code makes no more changes to vectors *r277* and *r279* and just writes the contents to the output stream.

3) Vector *r351* is instantiated at lines 43–45. Then an String object *$r386* is added to it at line 47. After *r351* turns to an enumeration at line 50, the length of its contents is written to the output stream.

After inspecting the above Jimple code, we can verify the results returned by our analysis and depicted in figure 4.8.

4.2.2.2 *createLexer()*

Figure 4.9 depicts graphically the results of our analysis on the *createLexer()* method. After analyzing the connection at the end of this method, we find four containment pairs *(292(r133),502(r136))*, *(502(r136),644($r154))*, *(53(r177),112(r182))* and *(112(r182), 232($r200))*. None of the above collections is in the list of the external collections. We conclude there are four pairs of not-may-shared collections: *(292(r133),53(r177))*, *(292(r133),112(r182))*, *(502(r136),53(r177))* and *(502(r136),112(r182))*.



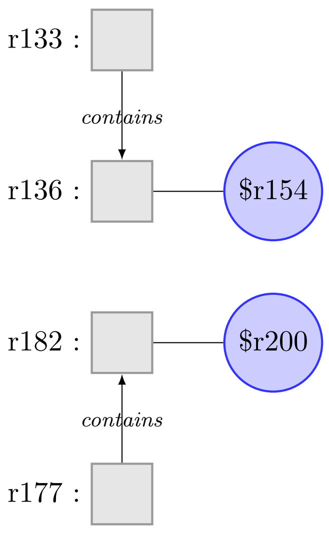
The above five collections are not-may-shared with each other except (r170,r173) *and* (r277,r279).

FIGURE 4.8: Results of the *createParser()* method in Figures 4.6–4.7

Figure 4.10 shows the Jimple code of the *createLexer()* method.

1) Vector *r136* is instantiated at lines 9–11. Then an array of int *$r154* is added to *r136*. Then *r136* is added to another vector *r133* at line 13. After that *r133* calls the method *elements()* to return an enumeration of its contents at line 18. After that the code makes no more changes to vectors *r133* and *r136* and just writes the contents to the output stream.

2) Vector *r182* is instantiated at lines 24–26. Then an Integer object *$r200* is added to *r182*. Then *r182* is added to another vector *r177* at line 30. After that *r177* calls the method *elements()* to return an enumeration of its contents



The above four collections are not-may-shared with each other except (r133,r136) *and* (r177,r182)*.*

FIGURE 4.9: Results of the *createLexer()* method in Figure 4.10

at line 33. After that the code makes no more changes to vectors *r177* and *r182* and just writes the contents to the output stream.

After inspecting the above Jimple code, we can verify the results returned by our analysis and depicted in figure 4.9.

```
private void createLexer()
{
        java.util.Vector r133, r136, r177, r182;
        ...
     label18:
        $r132 = new java.util.Vector;
        specialinvoke $r132.<java.util.Vector: void <init>()>();
        r133 = $r132;
        ...
     label19:
        $r135 = new java.util.Vector;
        specialinvoke $r135.<java.util.Vector: void <init>()>();
        r136 = $r135;
        ...
      label20:
        virtualinvoke r136.<java.util.Vector:
                void addElement(java.lang.Object)>($r154);
        ...
      label21:
        virtualinvoke r133.<java.util.Vector:
                void addElement(java.lang.Object)>(r136);
        ...
      label22:
        $i21 = virtualinvoke r133.<java.util.Vector: int size()>();
        r167 = virtualinvoke r133.<java.util.Vector:
                java.util.Enumeration elements()>();
        ...
      label29:
        $r176 = new java.util.Vector;
        specialinvoke \$r176.<java.util.Vector: void <init>()>();
        r177 = $r176;
        ...
      label30:
        $r181 = new java.util.Vector;
        specialinvoke $r181.<java.util.Vector: void <init>()>();
        r182 = $r181;
        ...
      label31:
        virtualinvoke r182.<java.util.Vector:
                void addElement(java.lang.Object)>(\$r200);
        ...
      label32:
        virtualinvoke r177.<java.util.Vector:
                void addElement(java.lang.Object)>(r182);
        ...
      label33:
        $i31 = virtualinvoke r177.<java.util.Vector: int size()>();
        r206 = virtualinvoke r177.<java.util.Vector:
                java.util.Enumeration elements()>();
        ...
}
```

FIGURE 4.10: Jimple code of the *createLexer()* method.

Chapter 5

Related Work

We discuss five main related works in this chapter: object representatives, heap reachability analysis, conditional must not aliasing analysis, static reasoning about contents of containers and iComments.

5.1 Object Representatives

Our collection-disjointness analysis relies on good points-to analysis information. Object representatives provide precise aliasing information based on the MustAlias and NotMayAlias analysis [1]. Our analysis employ object representatives to represent abstract objects in our data-flow sets, which simplifies the transfer function calculations and makes our data-flow sets easy-to-understand.

Object representatives can actually be used as a disjointness analysis to determine whether two objects are disjoint in the heap. Our disjointness analysis therefore can be seen as a generalization of object representatives from individual objects to collections.

5.2 Disjointness Analysis for Java-like Languages

The disjointness analysis for Java-like languages by Jenista and Demsky provide a method of determining disjointness properties of selected objects in Java-like languages [7]. It finds disjointness relations from the reachability information with static reachability graphs which contain nodes to represent objects and edges to represent heap references. In their approach, two objects are disjoint if they are not reachable in the reachability graph. Therefore. their disjoint analysis can find disjointness relations of data-structures, like collections, by reasoning about whether objects contained in the data-structures are disjoint or not.

Our collection-disjointness analysis focuses on sharedness relations between collections. The main difference is that they keep track all of the structures of the heap via the reachability graph while we use collections methods. Our abstraction uses the containment pairs to represent the containment relations, which say whether a collection contains an object, and sharedness pairs to represent the sharedness relations, which say whether two collections may contain aliased objects or shared collections. Our analysis also defines three types of sharedness relations: may-sharedness, must-sharedness and not-may-sharedness. We use a data-flow analysis to find disjointness relations based on the containment relations and annotations of method calls.

5.3 Conditional Must Not Aliasing Analysis

The conditional must not aliasing analysis by Naik and Aiken provide a method for static race detection. We say a multi-threaded program contains a *race* if two threads can access the same memory location without worrying about the order [11]. Instead of analyzing a piece of code directly by a must-alias analysis, this analysis start with locks in a multi-threaded program and reasons about locations in the locks. If two locks are different then their guarded locations must be disjoint.

Compared to our analysis, Naik and Aiken's analysis provides a complementary method for determining disjointness relations. Our analysis does not need the code to acquire a lock in a multi-threaded program, but instead determines disjointness relations to find whether we need to lock the collection operations. The conditional must not aliasing analysis can find the disjointness relations of any locations guarded by locks, while our analysis can only find disjointness relations of collections. However, our analysis is more flexible since it can find the disjointness relations of collections at any program points without regarding locks.

5.4 Static Reasoning about Contents of Containers

The static technique for reasoning about contents of containers by Dillig, Dillig and Aiken provides a method of precisely and automatically monitoring the contents of containers [2]. Their precise reasoning technique firstly defines a simple statically-typed language with concrete operational semantics to formalize their technique. Then they provide abstract semantics to define the abstract domain and the abstract model of containers. Based on the abstract domain and model, they provide abstract semantics to analyze container operations and iterations.

They classify containers into two types: position-dependent containers and value-dependent containers. The key insight of their technique is to focus on understanding the contents of containers without regarding how those containers are implemented. Therefore, they model any container as a function that converts a key to an abstract index of type integer, and then map the index to a value, which is the value of elements in the container [2]. They present abstract semantics on container operations for reading from, writing to, and allocating containers.

In our analysis, we also track the contents of collections, which are containers, for finding disjointness relations. The key difference between our approach and their static reasoning techniques is that our abstraction is different. In particular, it is more lightweight. We define containment pairs to monitor objects contained in containers and employ object representatives to represent contained objects. Our analysis tracks the contents of collections by analyzing explicit collection operations rather than grouping them into reading from, writing to, and allocating containers. Our analysis on the contents of collections is not as precise as their approach is, due to our lighter-weight abstraction. However, our analysis can still provide sound results in finding not-may-shared collections.

5.5 iComment

iComment by Tan, Yuan, Krishna and Zhou combines technique of Natural Language Processing (NLP), Machine Learning, Statistics and Program Analysis to automatically analyze comments written in natural language and detect inconsistencies between comments and code. Since our analysis reads annotations of callee methods in method calls, we are interested in their method of reading comments to determine disjointness relations. Their technique of reading comments is to build a rule generator to extract "hot" comments that specify certain assumptions and requirements (referred to as rules). The rules can be checked against source code.

Note that the developer-provided annotations in analysis are "rule-like" languages. These annotations specify assumptions and requirements about collection disjointness relations, and the developer-provided-post-annotations can be verified by our generated-post-annotations computed from the source code. The "rule-like" property of annotations may possibly enhance our analysis to read and write annotations in a more flexible way, instead of restricting annotations in only three forms as presented in chapter 3.4.3.1.

Chapter 6

Conclusions and Future Work

In this thesis, we defined the notion of disjointness relations, explained how to compute these relations and implemented a disjointness analysis relying on the collection API in Java. We also presented experimental results. Our collection-disjointness analysis enables light-weight specifications, thereby helping program understanding and parallelization.

We defined three types of disjointness relations between collections: may-shared, not-may-shared and must-shared. If two collections contain at least one object that must alias or at least one collection that is must-shared, we conclude that these two collections are must-shared; if two collections contain at least one object that may alias or at least one collection that is may-shared, we conclude that these two collections are may-shared; and if two collections contain no objects that may alias or no collections that are may-shared, we conclude that these two collections are not-may-shared. These definitions enable us to formally describe the disjointness relations between collections.

After giving the definitions, we presented rules for calculating and implementing our collection-disjointness analysis using the Soot [18, 19] Java bytecode optimization framework. We employed a forward data-flow analysis and used pairs of object representatives as the contents of our data-flow sets. We defined the initial value of our data-flow sets and presented transfer functions on Java statements that change the contents of our data-flow sets appropriately upon object instantiation, object assignment, collection operations and method calls. Method calls are usually difficult to analyze in static analysis since we do not know what the callee method does.

To solve this problem and better approximate our analysis results, we designed and implemented a way of reading developer-provided annotations on the callee methods to provide disjointness information as the result of the method calls. We also provided a way of generating post-annotations from pre-annotations to reduce the work of programmers.

After the implementation, we finally reported experimental results from our collection-disjointness analysis. Our analysis finds not-may-shared collections correctly on two benchmarks.

6.1 Future Work

Our work on collection-disjointness analysis suggests to a number of possible enhancements in the future:

1. A *graphical user interface* for users to query data-flow sets to find disjointness relations of any two collections. Currently, to use our collection-disjointness analysis, users need to add a line of code to their source code to query the disjointness relations. A graphical user interface would make our analysis easier to use, even for those users who know nothing about our analysis and just want to use it to determine disjointness relations between selected collections.

2. A more *precise analysis* to distinguish the must-shared relations from the may-shared relations. Currently, our analysis can not distinguish between a must-shared relation and a may-shared relation due to our current approximation mechanisms. Note our reported results are still sound: any time our analysis reports not-may-shared, this is true. However, an extended analysis can provide additional information, which can enable programmers to better understand theirs programs.

3. A more *flexible analysis* to allow user to write annotations and allow our analysis to read annotations in more "flexible" languages. Currently, contents of the annotations must be string nodes in the following three forms: *NotMayShared(l1, l2)*, *MayShared(l1, l2)* and *MustShared(l1, l2)*. Instead, if contents of the annotations are not restricted by the above forms, it is easier for programmers to write and understand annotations. Tan et al's research

on reading and verifying comments [16] in programming languages provides a potential technique to achieve this goal.

Bibliography

[1] Eric Bodden, Patrick Lam, and Laurie Hendren. Object representatives: a uniform abstraction for pointer information. In *Proceedings of the 1st International Academic Research Conference of the British Computer Society (Visions of Computer Science)*, pages 391–405, 2008.

[2] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, Austin, USA, 2011.

[3] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to Java program analysis with Soot. July 2008.

[4] Etienne Gagnon. SableCC, Last Accessed in Novemver 2010. URL `http://sablecc.org/`.

[5] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[6] java.net. JavaCC. Last Accessed November 2010. URL `https://javacc.dev.java.net/`.

[7] James C. Jenista and Brian Demsky. Disjointness analysis for java-like languages. Technical Report Technical Report UCI-ISR-09-1 REVISED, April 2009.

[8] Patrick Lam. Compiler fundamentals. Lecture notes for ECE750-T5: Static Analysis for Software Engineering, University of Waterloo, September 2008.

[9] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1:323–337, December 1992. ISSN 1057-4514. doi: http://doi.acm.org/10.1145/161494.161501. URL `http://doi.acm.org/10.1145/161494.161501`.

[10] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[11] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 327–338, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: http://doi.acm.org/10.1145/1190216.1190265. URL `http://doi.acm.org/10.1145/1190216.1190265`.

[12] Aakarsh Nair. Object histories in Java. Master's thesis, University of Waterloo, April 2010.

[13] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[14] Michael I. Schwartzbach. Lecture notes on static analysis. University of Aarhus, 2008.

[15] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 112–122. ACM, June 2007.

[16] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. iComment: Bugs or bad comments? In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[17] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of Java code for empirical studies. 2010 Asia Pacific Software Engineering Conference (APSEC2010), December 2010.

[18] Raja Vallée-Rai. Soot: a Java bytecode optimization framework. Master's thesis, McGill University, July 2000.

[19] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot

framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000. URL `www.sable.mcgill.ca/publications`.

Appendix A - Counting the number of ***Add*** and ***Remove*** Methods

With help from Derek Rayside, we examined 99 Java programs and roughly counted the number of statements containing *add()* methods and *remove()* methods by string matching. The purpose of this examination is to prove that programmers use the group of *remove()* methods our analysis cannot handle less often than the group of *add()* methods.

The group of *add()* methods we checked consists of *add()*, *addFirst()*, *addLast()*, *addElement()* and *addAll()*, and the group of *remove()* methods we checked consists of *remove()*, *removeFirst()*, *removeLast()*, *removeElement()*, *removeElementAt()*, and *removeRange().*

Note that the analysis we presented in Chapter 3 does nothing to *removeElement()* statements on vectors and *remove()* statements on collections other than sets since it cannot track the number of occurrences of the object in the collections. Our analysis also does nothing to *removeFirst()*, *removeLast()*, *removeElementAt()* and *removeRange()* since it cannot track during run-time the sequence of the contents of collections.

We just roughly count numbers by string matching to determine the program behaviour. The results not only include the number of *add* and *remove* operations but also include the number of method declarations of *add* and *remove.* For example, if a program has a method declaration "*void add(Object o)* {...}", we just count *add(Object o)* as one occurrence of the *add* operations. The results also ignore the case when programmers declare their own *add* or *remove* methods. For example, if a program has a method "*void addF(Object o)* {**.add(o);*}", all *add()* operations by using method *addF()* are not counted. However, since we only roughly approximate the behaviour of a program, these results give a good idea of the overall trends in our examination.

The results proved our assumption. The number of *add()* method calls occurring in 99 programs is 72782 while the number of *remove()* method calls is 9603, which shows that programmers use the group of *add()* methods about ten times as often as using the group of *remove()* methods.

The list of programs we examined are from the Qualitas Corpus, a standard corpus of open-source Java software for use in empirical studies of software[17]:

`Ant 1.8`	`Antlr 3.2`	`ArgoUML 0.3`
`ArtOfIllusion 2.5.1`	`AspectJ 1.0.6`	`Axion 1.0-M2`
`Azureus 4.3.1.4`	`C-JDBC 2.0.2`	`Checkstyle 4.3`
`Cobertura 1.9`	`Colt 1.2`	`Columba 1.0`
`Compiere 2.5`	`Derby 10.1.1.0`	`DisplayTag 1.1`
`Dr.Java 20050814`	`DrawSWF 1.2.9`	`Eclipse 3.6`
`Emma 2.0.5312`	`FindBugs 1.0`	`FitJava 1.1`
`FitLibraryForFitness 20050923`		`FreeCS 1.2`
`FreeCol 0.9.2`	`GT2 2.2`	`Galleon 1.8`
`Gantt 1.11.1`	`HSQLDB 1.8.0.4`	`HTMLUnit 1.8`
`Heritrix 1.8`	`Hibernate 3.5.3`	`Informa 0.6.5`
`JPF 1.0.2`	`JREFactory 2.9.19`	`JSPWiki 2.2.33`
`Jag 5.0.1`	`James 2.2`	`JasperReports 1.1`
`JavaCC 3.2`	`Jena 2.5.5`	`Jext 5.0`
`Jung 2.2.0.1`	`Log4J 1.2.13`	`Lucene 3.0.1`
`MVNForum 1.0`	`Marauroa 2.5`	`Megamek 20051011`
`MyFaces 1.2`	`NakedObjects 3.0.1`	`NekoHTML 0.9.5`
`OSCache 2.3`	`OpenJMS 0.7.7a3`	`PMD 3.3`
`POI 2.5.1`	`PicoContainer 1.3`	`Pooka 060227`
`Proguard 3.6`	`Quartz 1.5.2`	`QuickServer 1.4.7`
`Quilt 0.6a5`	`RSSOwl 1.2`	`Roller 2.1.1`
`SableCC 3.1`	`Sandmark 3.4`	`SpringFramework 1.2.7`
`SquirrelSQL 2.4`	`Struts 1.2.9`	`Sunflow 0.07.2`
`Tomcat 5.5.17`	`Trove 1.1b5`	`Velocity 1.5`
`WebMail 0.7.10`	`Weka 3.7.1`	`XMojo 5.0`
`Xalan 2.7`	`Xerces 2.8`	`eXoPortal 1.0.2`
`iReport 0.5.2`	`iText 1.4.5`	`iVataGroupware 0.11.3`
`jASML 0.1`	`jChemPaint 2.0.12`	`jEdit 4.3pre14`
`jFinDateMath R1.0`	`jFreeChart 1.0.1`	`jGraph 5.13`
`jGraphPad 5.10.0.2`	`jGraphT 0.7.3`	`jGroups 2.6.2`
`jHotDraw 6.0b1`	`jMeter 2.3.4`	`jMoney 0.4.4`
`jOggPlayer 1.1.4`	`jParse 0.96`	`jRat 0.6`
`jRuby 1.0.1`	`jTOpen 4.9`	`jUnit 4.8.1`
`jsXe 0.4b`		