Collection Disjointness Analysis

Hang Chu

robin.h.chu@gmail.com

Patrick Lam

University of Waterloo
p.lam@ece.uwaterloo.ca

Abstract

We present a static analysis which identifies disjointness relations between collections in Java. We have implemented our analysis as a primarily intraprocedural dataflow analysis framework using Soot. We handle method calls using developer-provided annotations, with some inference support. We include experimental results of the from our disjointness analysis on a pair of benchmarks.

1. Introduction

Almost all modern programming languages provide mutable set data structure implementations, or *containers*, either in their standard libraries (as in Java, .NET, and C++), or within the language itself (e.g. Python). Containers are widely used as they reduce programming effort, and increase programming speed and quality, by reusing known-good components which have well-understand behaviours. A container stores a set, or *collection*, of objects.

Two collections are disjoint if the intersection of their contents is the empty set—in other words, two disjoint collections do not contain objects pointing to the same heap location. Disjointness relations enable lightweight specifications, which are helpful for program understanding, verification, and parallelization. For example, consider a program with two modules, A and B, which each use collection objects to store data. Disjointness between the collections used in A and B enables the developer to assume that changes to objects stored in A's collections will never affect the objects stored in B's collections. Note that aliasing relations between the collection objects themselves are not strong enough to enable this assumption; reachability is required. Our disjointness analysis approximates reachability with a coarse-grained approximation of the set of objects stored in each collection. Disjointness analysis also enables parallelization—if we can find two collections that are completely disjoint in all executions, we can safely execute operations on the collection in parallel.

In this work, we present the design and implementation of an intraprocedural static analysis which verifies disjointness relations of collections in Java. The key insight behind our analysis is that it is possible to rely on the documented semantics of the collections classes in the Java library to dramatically simplify the analysis; see [8, 9] for previous work on verifying collection-manipulating methods. Section 3 presents our analysis in detail; the underlying abstraction is inspired by connection matrices [6]. Note that, since our analysis is primarily intraprocedural, we rely on developer-

provided annotations to obtain method preconditions and postconditions. Disjointness analysis enables developers to compute or verify disjointness relations between any two collections at any program point of the source code. Using these relations, developers can provide light-weight specifications and better understand their code, and compilers can better optimize parallel programs.

To investigate the effectiveness of our algorithm, we executed it on two simple benchmark programs. Our analysis successfully identified twelve collections from these programs as disjoint. Section 4 presents our experimental results in more detail.

Section 2 continues with a motivating example, which explains how our analysis works on a simple case.

2. Motivating Example

Consider the small Java program in Figure 1, which first instantiates two objects and three lists, and then causes sharing by inserting the same object into multiple lists. We next explain the operation of our collection disjointness analysis by presenting the results of the analysis[1] on this code, which we include in the comments alongside the code. The interpretation of these results follows.

First, note that the analysis returns the empty set after lines 3 through 8. Our analysis tracks sharedness (a.k.a non-disjointness) across heap objects, and the empty set means that all objects and collections are disjoint. This is clearly the state upon entry to the program. (Our analysis also accepts annotations which summarize the state of the heap upon entry to a method.) Instantiations do not change the disjointness of the objects already in the heap, so the analysis returns the empty set through the end of line 8.

The subsequent lines create relations between heap objects, and our analysis tracks these relations. Our abstract state consists of a set of pairs, each of which indicates a relationship between (abstract) heap objects. We distinguish heap objects of container type and follow the contents of such objects. Our abstraction tracks two types of pairs—containment pairs, which indicate that a container object may contain some heap object, and sharedness pairs, which record derived information about relationships between containers. We describe the semantics of these pairs fully in Section 3.1. Line 9 adds o_1 to list ℓ_1, and our analysis records that fact with the containment pair (ℓ_1, o_1). The code continues by adding o_2 to ℓ_2, o_1 to ℓ_3, and o_2 to ℓ_3, and our analysis adds the containment pairs $\{(\ell_1, o_1), (\ell_2, o_2), (\ell_3, o_1), (\ell_3, o_2)\}$ and the sharedness pairs $\{(\ell_1, \ell_3), (\ell_2, \ell_3)\}$.

Note that line 11 adds o_1 to list ℓ_3, but that o_1 is already in list ℓ_1. Our analysis adds a *sharedness* pair between lists ℓ_1 and ℓ_3 to record this fact. We can therefore query the analysis results at any point following line 11 to find that ℓ_1 and ℓ_3 are not disjoint.

Figures 2 and 3 depict graphically the results from the above example after line 10 and at the end of the *main()* method respec-

---

[1] For now, we elide object identity information from our analysis results; Section 3.1 describes our use of object representatives [1] to disambiguate heap objects.

```
1  class C {
2   public static void main(String [] args) {
3                                   // { }
4    List l1 = new LinkedList();    // { }
5    List l2 = new LinkedList();    // { }
6    List l3 = new LinkedList();    // { }
7    Object o1 = new Object();      // { }
8    Object o2 = new Object();      // { }
9    l1.add(o1);                    // {(l1,o1)}
10   l2.add(o2);                    // {(l1,o1),(l2,o2)}
11   l3.add(o1);   // {(l1,o1),(l2,o2),(l3,o1),(l1,l3}
12   l3.add(o2);   // {(l1,o1),(l2,o2),(l3,o1),
13                 //    (l3,o2),(l1,l3),(l2,l3)}
14   }
15 }
```

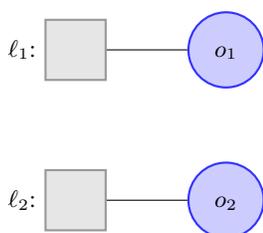**Figure 1.** Collection disjointness results on simple example.

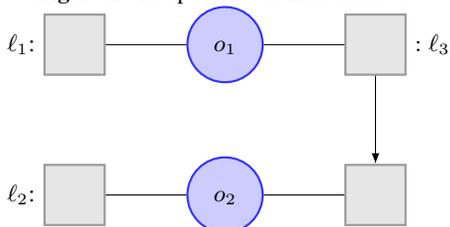**Figure 2.** Heap abstraction after line 10

Figure 3. Heap abstraction after line 13

tively. The Figures show the disjointness relations between ℓ_1, ℓ_2 and ℓ_3; boxes represent linked list cells, and lines indicate that cells contain objects o_i. Figure 2 indicates that ℓ_1 and ℓ_2 are disjoint, while Figure 3 indicates that ℓ_2 and ℓ_3 share the common object o_2, while ℓ_1 and ℓ_3 share the common object o_1.

3. Description of Analysis

We designed our collection disjointness analysis to determine whether two given collections contain objects that may point to the same heap location. The analysis maintains an abstraction based on connection matrices [6]. In particular, the abstraction tracks relationships between object representatives [1], which finitize the heap and encapsulate must-aliasing and may-aliasing relationships between objects. We track two types of relationships: containment relationships, which connect collections and their contained objects, and sharedness relationships, which summarize relationships between different container objects. Our analysis primarily updates relationships when it encounters calls that mutate contents of collection objects, such as `add()` or `clear()`.

If two collections ℓ_1 and ℓ_2 contain no objects that may alias nor collections that may share objects (on any execution), we say ℓ_1 and ℓ_2 are not-may-shared, i.e. disjoint. Not-may-shared collections are unreachable from each other using heap reads. Conservative approximations may result in spurious sharing relationships

between collections. However, if our analysis reports that two collections are disjoint, we guarantee that there is indeed no sharing between them on any execution.

Our disjointness analysis is a primarily-intraprocedural forward dataflow analysis. We use annotations, along with the program's call graph, to enable interprocedural disjointness analysis; see Section 3.3 for details.

3.1 Heap Abstraction

Our dataflow analysis tracks, for each statement in the intermediate representation, two types of pairs of object representatives, where the representatives are drawn from the set of representatives for the method under analysis. Although we always present pairs from our abstraction using local variable names, the analysis itself stores object representatives. Soot computes a map from local variables to object representatives and updates this map automatically; we simply assume that object representatives are available and accurate, with one exception—our analysis manually updates the object representatives map when it encounters a `ParameterRef` assignment in a `DefinitionStmt`. Such statements, which are part of the Jimple intermediate representation, provide explicit definition points for method parameters.

Field Accesses. Collections which are read from fields are challenging to handle. Consider a field read $\ell = o.f$, where ℓ is a collection and $o.f$ is a field of o. We must update containment relations and sharedness relations for ℓ even when updates occur through $o.f$ and its aliases, which requires alias analysis. Fluid updates [3] are probably the best solution for this problem. However, in this work, we propose a conservative solution which trades off accuracy for simplicity.

Our approach designates all collections originating from field reads as *external collections* in our analysis abstraction. As with other objects, we refer to external collections by their object representatives. We assume that all external collections are may-shared with all other collections in scope (except collections instantiated after the read). Note that any object with compatible type may potentially be a collection.

Contents of Heap Abstraction. Our abstraction stores two types of pairs:

1. *containment* pairs: our analysis indicates that a collection ℓ_1 (may) contain an element o_1 with the containment pair (ℓ_1, o_1). Such pairs are ordered. Collection operations like `l1.add(o1)` give rise to containment pairs.

   If the abstraction contains a containment pair (ℓ_1, o_1), then collection ℓ_1 may contain object o_1. Otherwise, ℓ_1 definitely does not contain o_1.

2. *sharedness* pairs: our analysis indicates non-disjointness relations between collections ℓ_1 and ℓ_2 using the unordered sharedness pair (ℓ_1, ℓ_2). Such a pair means that ℓ_1 and ℓ_2 (may) contain, respectively, objects o_1 and o_2, such that o_1 and o_2 may alias. Alternatively, o_1 and o_2 may themselves be may-shared collections. Sharedness pairs are generally created by our analysis based on the containment pairs (see Section 3.4); however, note that it is possible to have a sharedness pair in the abstraction even when there is no witness to the sharing, and such a pair may arise from the analysis of a collection operation like `l1.addAll(l2)`.

   If the abstraction contains a sharedness pair (ℓ_1, ℓ_2), then there may exist objects $o_1 \in \ell_1$ and $o_2 \in \ell_2$ such that o_1 and o_2 may-alias. In the absence of such a pair, our analysis guarantees that ℓ_1 and ℓ_2 do not share any objects. Our analysis does not currently specifically identify the case where ℓ_1 and ℓ_2 must share an object.

```
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(∗))}
8   l2 = l1 ;        //  {(0( l1 ),2( o1 )),(1(∗),3(∗))}
```

Figure 4. Assignment Example; note use of object representatives

A pair (ℓ_1, ℓ_2) consisting of two collections can be either a containment pair or a sharedness pair. Although we hide the distinction between the types of pairs in our example, our abstraction actually tracks whether each pair is a containment pair or a sharedness pair, based on how it was initially created. Disjointness query results are based on the sharedness pairs.

Merge operator. As we are designing a "may" analysis, we combine sets using union at control-flow merges.

Initial value. We start with the empty set of pairs as an initial approximation. When an external collection comes into scope, we note that it is may-shared with all existing collections. We rely on annotations to sharpen the results for collections occurring as method parameters (rather than marking them as external collections, which would be a safe but coarse approximation).

3.2 Transfer Functions

We next describe the transfer functions at the core of our dataflow analysis.

Object instantiation. As mentioned in Section 2, `new` expressions do not add any pairs to our abstraction, since newly-created collections do not contain any objects, and newly-created objects do not belong to any collections. Note that the effect of overwriting a local variable is captured by updates to the object representatives map, and need not be considered here.

Assignment. Assignments also never add pairs to, nor remove pairs from, the abstraction. This is because an assignment never affects the contents of any collection. Once again, object representatives introduce a layer of indirection, such that changing the contents of a local variable does not affect the underlying object representatives for the old contents of that variable.

Examples in this paper use local variables rather than object representatives; we believe that the use of local variables makes the examples easier to follow overall. However, Figure 4 shows an example of how our analysis deals with assignment statements. Here, we have chosen to show both an object representative, represented by an integer, and a local variable containing that representative.

Note that, after line 6, the abstraction connects ℓ_1 with o_1 and ℓ_2 with o_2, as one might expect. Now, after line 7, which assigns o_2 the reference contained in o_1, the same abstraction remains valid; only the mapping from object representative 3 to variable o_2 changes, but the contents of ℓ_2 remain the same. Similarly, the assignment at line 8 does not change the contents of object representative 1, but it does invalidate the mapping between local variable ℓ_2 and object representative 1.

3.2.1 Collection operations

Our analysis does most of its work analyzing calls to collection operations. Figure 5 illustrates the operation of our analysis on a number of common collection operations. We continue by discussing how our analysis handles calls to the following classes of methods from `java.util.Collection`: adds, removes, and other 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 );      // {( l1 , o1 )}
6    l2 . addFirst ( o2 );   // {( l1 , o1 ),( l2 , o2 )}
7    l2 . addLast ( o1 );    // {( l1 , o1 ),( l2 , o2 ),( l2 , o1 ),( l1 , l2 )}
8    l1 . addAll ( l2 );     // {( l1 , o1 ),( l2 , o2 ),( l2 , o1 ),( l1 , l2 ),( l1 , o2 )}
9    l1 . remove ( o1 );     // {( l1 , o1 ),( l2 , o2 ),( l2 , o1 ),( l1 , l2 ),( l1 , o2 )}
10   l2 . removeAll ( l1 ); // {( l1 , o1 ),( l1 , o2 )}
11   l1 . clear ();          // {}
12   l2 . add ( o1 );        // {( l2 , o1 )}
13   l2 . set (0,  o2 );     // {( l2 , o1 ),( l2 , o2 )}
```

Figure 5. Collection Operations Example
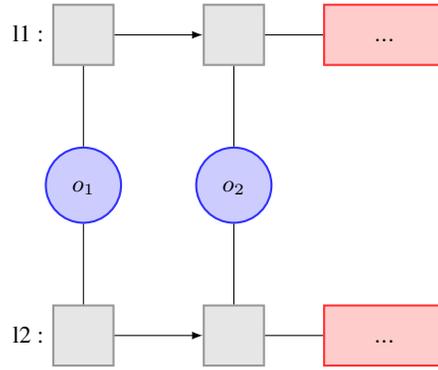
Both l1 and l2 contain o1 and o2.

Figure 6. Analysis state after Line 8 in Figure 5.

Add methods. For an "add" method (i.e. `add()`, `addElement()`, `addFirst()`, `addLast()`, and `addAll()`), we add a containment pair between the receiver object and the object parameter. For instance, `l.add(o)` gives rise to a containment pair (ℓ, o). Some of these methods provide additional information which we are not able to use; for instance, we discard the position information at a call to `add(int, Object)`.

The Java Collections API also provides an `addAll()` method, which adds the contents of a collection to the receiver. At a call to `s: l1.addAll(l2)`, we add the containment pairs from ℓ_2, replacing ℓ_2 by ℓ_1, plus a sharedness pair between ℓ_1 and ℓ_2:

$$\text{OUT}(s) = \text{IN}(s) \cup \{(\ell_1, k) \mid (\ell_2, k) \in \text{IN}(s)\} \cup \{(\ell_1, \ell_2)\}.$$
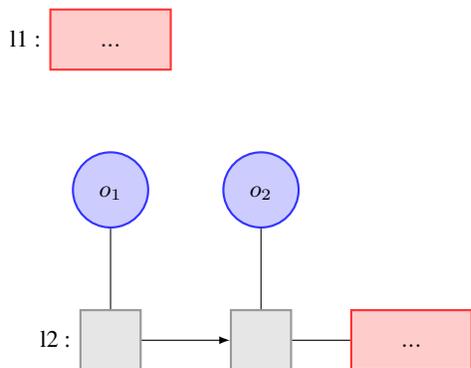
In general, two collections ℓ_1 and ℓ_2 may be aliased. Our use of object representatives transparently handles potential problems due to aliasing: if ℓ_1 and ℓ_2 are may-aliased, a query for objects contained in ℓ_1 will also return objects in ℓ_2.

Lines 5–8 in Figure 5 illustrate our analysis on different types of *add* methods. Observe, for instance, that after line 5, our abstraction contains a pair (ℓ_1, o_1), as one might expect. Also note the sharedness pair (ℓ_1, ℓ_2) added after line 7.

Figure 6 depicts graphically the results of the group of *add()* methods for the example in Figure 5.

Remove methods. Our abstraction contains enough information to remove pairs at calls to `Set.remove(Object)` and `removeAll()`; we must conservatively approximate the effects of the other remove methods by leaving our abstraction unchanged.

At a call to `Set.remove(Object)`, we can remove the pair (ℓ, o) (since a `Set` is guaranteed to contain an object at most once). However, we cannot remove anything following a call to `Collection.remove(Object)`—objects may belong to collec-

ℓ_2 contains o_1 and o_2; ℓ_1 and ℓ_2 are disjoint.

Figure 7. Analysis results after *set()*, Line 13 in Figure 5.

```
1   public @interface DisjointnessAnnotation {
2     String pre();
3     String post();
4     String generatedPostAnnotations();
5   }
6
7   @DisjointnessAnnotation(pre = "NotMayShared(l1,l2);"
8                               "MustShared(l2,l3)",
9           post = "",
10          generatedPostAnnotation = "")
11          /* analysis generates: "MayShared(l1,l2);"
12                                  "NotMayShared(l2,l3)" */
13  public static void foo (List l1, List l2, List l3) {
14    l1 = l2;
15    l3.clear();
16  }
```

Figure 8. Annotations Example

straction. We apply a formal/actual parameter mapping to connect names at the caller and callee sites.

If a callee post-annotation shows a may-shared or must-shared relation between two local variables, we add the corresponding sharedness pair. If the post-annotation shows a not-may-shared relation between two locals, we remove pairs containing these two locals from the abstraction.

Consider the example in Figure 8. If the analyzing method calls *foo(a,b,c)*, our analysis adds sharedness pair $(m(a), n(b))$ to the connection and removes pair $(n(b), p(c))$ from the connection, where m, n, p are object representatives for a, b, c respectively.

Verification and inference of post-annotations. If the developer specifies post-annotations, our analysis will verify that they are correct. To ease the annotation burden, our analysis supports automatic inference of post-annotations. We synthesize post-annotations as follows. As always, we read the pre-annotation and create an initial value based on it. We then analyze the method as usual, which generates sets of pairs at all tails of the method's control-flow graph. We combine all of these pairs to get a single set.

If the developer has manually specified a post-annotation, we verify that the analysis result implies the post-annotation—that is, all sharedness relations in the analysis result must also be in the post-annotation. It is unsound to omit sharedness pairs, since such an omission implies disjointness. Otherwise, we add the generated post-annotations to the metadata for the callee method.

3.4 Sharedness Updates

Recall that a sharedness pair in our abstraction indicates that collections ℓ_1 and ℓ_2 either contain objects o_1 and o_2 which may-alias, or collections ℓ'_1 and ℓ'_2 which are not disjoint. While our transfer function automatically creates some sharedness pairs (for instance after a call to `addAll()`), it will miss pairs that arise as a result of aliasing between objects. Therefore, after every statement, our transfer function ensures that the set of sharedness pairs in the abstraction is complete, as follows.

- Given containment pairs (ℓ_1, o_1) and (ℓ_2, o_2), if o_1 and o_2 may-alias (or must-alias), add sharedness pair (ℓ_1, ℓ_2).
- Given containment pairs (ℓ_1, ℓ'_1) and (ℓ_2, ℓ'_2), if the abstraction contains sharedness pair (ℓ'_1, ℓ'_2), then add sharedness pair (ℓ_1, ℓ_2). (Note that this condition is always true for external collections, as they are shared with all pre-existing collections.)

We repeat these steps until we reach a fixed point.

4. Experience

We next describe our experience using our implementation of our collection disjointness analysis on two open source Java projects,

tions with multiplicities greater than 1. Certain other forms, such as `remove(int)`, `removeFirst()`, and `removeLast()`, all remove objects at given indices from the collection. Since we do not know which objects are at which indices, we cannot update the abstraction to remove any particular object.

We handle a call to `l1.removeAll(l2)` symmetrically to `addAll()`, as follows:

$$\text{OUT}(s) = \text{IN}(s) \setminus \{(\ell_1, k) \mid (\ell_2, k) \in \text{IN}(s)\} \setminus \{(\ell_1, \ell_2)\}.$$

Note that `removeAll()` results in ℓ_1 and ℓ_2 being disjoint, as long as $\ell_1 \neq \ell_2$.

Other methods. As seen in Line 11 from Figure 5, we can handle calls to `l.clear()` by removing all containment and sharedness pairs involving ℓ. The `removeAllElements()` method from `Vector` behaves identically.

The `List.set()` method is analogous to a `remove()` of an unknown element followed by an `add()`. We treat it like an `add()`, since we do not know the identity of the object being replaced.

Line 13 in Figure 5 shows the effect of `set()`; note that we conservatively state that ℓ_2 contains both objects o_1 and o_2, even though it is clear that o_1 was replaced by o_2 at the `set()` call. Figure 7 depicts the state graphically.

3.3 Method Calls

As our analysis is intraprocedural, we primarily use method annotations to capture the effects of method calls. We expect method annotations to be provided by the developer and we trust (but verify) these annotations. Our analysis can also synthesize post-annotations based on pre-annotations. To permit the use of synthesized post-annotations, we visit methods in topological order, starting with the leaves of the call graph, and arbitrarily break cycles. We next discuss the annotation language that our analysis accepts.

Figure 8 shows annotations for method `foo()`. It contains a pre-annotation along with an empty post-annotation and a generated post-annotation. The `generatedPostAnnotation` is blank, to be filled in by the analysis. Note that our annotation language is currently more expressive than our abstraction: while our abstraction can only express may-sharing between collections, the annotation language supports must-sharing. Two collections are must-shared if, on all executions, there is an object o which must belong to both collections. It is sound to represent must-sharing using the may-sharing sharedness pairs in our abstraction.

Our intraprocedural analysis handles method calls by requesting the set of possible callees for the callsite and integrating the post-annotations for all of the callees into the caller's analysis ab-

	SableCC	JavaCC
Total analysis time (seconds)	213	278
Peak memory usage (Mb)	772	1754
Methods analyzed	1824	1066
Methods calling collection operations	549	134
Methods taking collection arguments	46	28
Methods taking 2+ collection arguments	1	6
Single-collection methods	267	45
Not-May-Shared Collections found	12	0

Figure 9. Experimental Results

```
1  private static void listSplit(List toSplit, List mask,
2                                List partInMask, List rest) {
3  OuterLoop:
4   for (int i = 0; i < toSplit.size(); i++) {
5    for (int j = 0; j < mask.size(); j++) {
6     if (toSplit.get(i) == mask.get(j)) {
7      partInMask.add(toSplit.get(i));
8      continue OuterLoop;
9     }
10   }
11   rest.add(toSplit.get(i));
12  }
13  // {(rest, partInMask),(rest, temp$7),(partInMask, temp$6)}
14 }
15
16 public static List genFollowSet(List partialMatches,
17                                Expansion exp,
18                                long generation) {
19  List v = ...;
20  List v1 = new ArrayList(); List v2 = new ArrayList();
21  listSplit(v, partialMatches, v1, v2);
22 }
```

Figure 10. Unannotated JavaCC code snippets.

SableCC and JavaCC, and include interesting results that we found on snippets of code from these projects. SableCC [5] is an open source parser generator in Java designed by Etienne Gagnon; it generates parsers for LALR languages. We analyzed SableCC version 3.2. JavaCC [2] (Java Compiler Compiler) is a second Java open source parser generator; it generates top-down parsers. We analyzed JavaCC version 4.2. Note that we analyze the parser generators themselves rather than any generated parsers.

Figure 9 presents benchmark characteristics, statistics about the running times of our analysis on these benchmarks, and counts of the results that we found. We did not find any disjoint pairs of collections in JavaCC, while we found 12 disjoint pairs of collections in SableCC. We ran the analysis on a desktop computer with a 3.20 GHz Intel Pentium D with 3.5GB of memory. The analysis times (hundreds of seconds) and peak memory usage (772–1754Mb) are large but tractable. Note that 30% (SableCC) or 10% (JavaCC) of methods call at least one collection operation. Note also that many methods work with only a single collection; disjointness between collections is not relevant to such methods.

Disjointness Annotations. Figure 9 also shows that 46 methods in SableCC and 28 methods in JavaCC take collections as arguments. We manually added pre-annotations to all methods that take at least two collections as arguments—1 in SableCC and 6 in JavaCC. We also added selected post-annotations to mitigate coarse approximations from the static analysis. Figures 10–11 illustrate how pre-annotations and developer-provided post-annotations contribute to our analysis results, in the context of two methods from the *LookaheadWalk* class in *JavaCC*. Note that `genFollowSet()` calls `listSplit()`, which splits a given list into two parts relative to a mask. The two parts are stored separately into two lists.

```
1  @list_ano(pre="MayShared(toSplit,mask);"
2             "NotMayShared(partInMask,rest)",
3         post="MayShared(toSplit,partInMask);"
4             "MayShared(toSplit,rest);"
5             "MayShared(mask,partInMask)")
6  private static void listSplit(List toSplit, List mask,
7                                List partInMask,
8                                List rest) {
9   // method code...
10  // {(rest,partInMask),(rest,temp$7),
11  //  (partInMask,temp$6),(toSplit,mask)}
12 }
13
14 public static List genFollowSet(List partialMatches,
15                                Expansion exp,
16                                long generation) {
17  List v = ...
18  List v1 = new ArrayList(); List v2 = new ArrayList();
19  listSplit(v, partialMatches, v1, v2);
20  // {(v,v1),(v,v2),(partialMatches,v1),
21  //  (v,partialMatches),(v1,v2))}
22 }
```

Figure 11. JavaCC snippets with pre- and post-annotations

Figure 10 first shows our analysis results in the absence of annotations. Our analysis follows the call graph and analyzes `listSplit()` first. At the end of `listSplit()`, we find two containment pairs *(rest,temp$7)* and *(partInMask, temp$6)*, where *temp$7* and *temp$6* are local variables introduced by the Jimple intermediate representation, and a sharedness pair *(rest, partInMask)*, indicating that *rest* and *partInMask* may share values 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()`. We must conservatively conclude that all collections may share elements after every method call. Annotations enable us to sharpen our analysis results.

Figure 11 shows both pre-annotations and post-annotations. At the end of `listSplit()`, *toSplit* is may-shared with both *partInMask* and *rest*, since *toSplit* is possibly stored in both *partInMask* and *rest*. *mask* and *partInMask* are also may-shared, since all masked elements in *toSplit* are possibly stored in *partInMask*. We therefore add the post-annotation "MayShared(toSplit, partInMask); MayShared(toSplit, rest); MayShared(mask, partInMask)".

When analyzing the method call at line 19, our analysis computes sharedness pairs *(v,v1),(v,v2)*, and *(partialMatches,v1)*, which reflect the may-shared relation between *v* and *v1*, *v* and *v2*, and *partialMatches* and *v1*.

Remark on single-collection methods. We analyze methods which take a single collection as an argument by conservatively treating this collection as an external collection which is may-shared with all other collections. To generate sound results, we require pre-annotations for methods that takes more than one collection as arguments. Such methods are rare in practice.

4.1 Disjointness results from SableCC

We next present the disjointness results from SableCC, which are in methods `createParser()` (8 pairs) and `createLexer()` (4 pairs), from classes `org.sablecc.sablecc.GenParser` and `org.sablecc.sablecc.GenLexer` respectively. We have verified their disjointness through manual inspection.

Figure 12 depicts the results of our analysis on the *createParser()* method. We briefly discuss the structure of this method.

1. Vector *r173* is instantiated, gets four int arrays, added to another *r170* vector, converted to an enumeration, and written out.

2. Vector *r279* is instantiated, gets two int arrays, added to vector *r277*, converted to an enumeration, and written out.
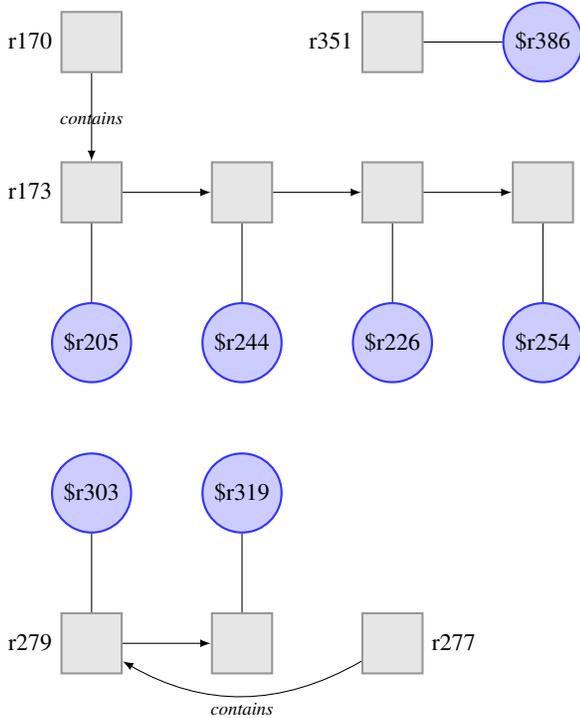
The above five collections are disjoint, except (r170,r173) *and* (r277,r279).

Figure 12. Analysis results for *createParser()*.

3. Vector *r351* is instantiated, gets a String, is converted to an `Enumeration`, and its length written out.

We manually inspected the Jimple code for this method to confirm our analysis results, which contain all expected pairs, such as $\{(r351, r173)\}$, and no unexpected pairs (e.g. $(r170, r173)$). We have not yet sampled arbitrary code to compare our disjointness results with the ground truth.

5. Related Work

In this section, we discuss object representatives, heap reachability analysis, and static reasoning about contents of containers.

Object Representatives. Object representatives [1] provide precise must and not-may alias analysis results. Our analysis employs object representatives to represent abstract objects in our dataflow sets, simplifying the transfer function calculations and making our data-flow sets easy-to-understand. Object representatives determine whether two objects (rather than containers) are disjoint in the heap. Our disjointness analysis can be seen as a generalization of object representatives from individual objects to collections.

Disjointness Analysis for Java-like Languages. The disjointness analysis for Java-like languages by Jenista, Eom and Demsky [7] has the same goal as our analysis. However, it reasons about disjointness by creating a static heap reachability graph and declaring collections disjoint if they are not mutually reachable. We instead use containment and sharedness pairs to reason about the heap; we are insensitive to the details of container implementation, and instead use the Collections API to understand the program. Our approach is therefore limited to a given set of implementations of data structures, but is much less complex than their approach. The

two approaches should be equally effective on their mutual domain. Differences may arise from Jenista et al's use of summary nodes. Note also that their analysis applies to Bamboo, which is a task-based extension to Java that they have developed.

Static Reasoning about Contents of Containers. Dillig, Dillig and Aiken have developed a technique for precisely and automatically monitoring the contents of containers [4]. Their approach classifies containers into two types: position-dependent containers and value-dependent containers. Like us, their analysis focusses on understanding the contents of containers without regard to how those containers are implemented. However, they instead model container contents (not just relationships between containers), using functions that convert a key into an abstract (integer) index, and then map this index to elements in the container. Container operations may read from, write to, and allocate abstract containers.

The key difference between approaches lies in the choice of abstraction. Our abstraction is more lightweight. We define containment pairs to monitor objects contained in containers and employ object representatives to represent contained objects. Our analysis of the contents of collections is not as precise as their approach is, due to the difference in abstractions. However, our analysis can still provide sound results in finding not-may-shared collections. The results are not directly comparable, as their approach works for C and C++, while ours works for Java.

6. Conclusion

We have defined the notion of disjointness between collections and presented an intraprocedural analysis to calculate disjointness relations, along with an annotation language for specifying disjointness at method entry and exit. Our analysis hard-codes the semantics of the Java Collections API into its transfer function. We have implemented our analysis in the context of the Soot framework [10] and presented experimental results. Our collection-disjointness analysis enables light-weight specifications, which contribute to program understanding, verification and parallelization.

References

[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] JavaCC developers. JavaCC. `https://javacc.dev.java.net/`.

[3] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.

[4] 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.

[5] Etienne Gagnon. SableCC. `http://sablecc.org/`.

[6] 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.

[7] James C. Jenista, Yong hun Eom, and Brian Demsky. Using disjoint reachability for parallelization. In *Proceedings of the Int'l Conference on Compiler Construction*, Saarbrcken, Germany, April 2011.

[8] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, MIT, February 2007.

[9] Patrick Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, MIT, February 2007.

[10] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.