Static Inference of Universe Types

Ana Milanova
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Abstract

The Universe type system is an ownership type system which enforces the owners-as-modifiers model. In this paper, we present a *static* analysis for inference of Universe types. We have implemented the analysis and performed preliminary experiments. Our results are promising.

1. Introduction

It is well-known that reasoning about ownership and enforcing ownership has important software engineering benefits. There are many ownership type systems in the literature; however, a disadvantage of these systems is their annotation requirement—developers would need to spend significant effort annotating new and existing software systems in order to realize the benefits of enforcing ownership.

Ownership inference can help alleviate this disadvantage. We believe that inference is important, because it may help bridge the gap between the theory of ownership types and software engineering practice. Therefore, it is important to develop new approaches for ownership inference.

In this paper, we present a static analysis for inference of Universe type modifiers[1]. The Universe type system [9, 8] has three type modifiers: `rep`, which denotes that the current object `this` is the owner of the referenced object, `peer` which denotes that the current object `this` and the referenced object are peers (i.e., they appear in the same context and have the same owner), and `any` which does not give any information. The Universe type system enforces the owners-as-modifiers model by forbidding modifications through `any` references (i.e., allowing modifications only through `rep` and `peer` references).

Our analysis is a lightweight static analysis built as a client of a points-to analysis; it extends our previous work on ownership inference according to the owners-as-dominators model [18]. We have implemented the analysis and present empirical results which we believe are promising.

Note that inference of Universe types has been done before—Dietl and Müller [10] present a dynamic analysis for this purpose. We argue that static analysis has some inherent advantages over dynamic analysis, and therefore our work may be useful as well. First, static analysis is conservative and produces results valid over all program executions, while dynamic analysis reasons only over current executions and produces unsound results. Second, static analysis may be more practical as dynamic analysis requires multiple runs and each run incurs (often significant) instrumentation overhead. Type inference is traditionally static; we believe that static ownership inference is a useful and relatively unexplored direction.

This paper has two possible contributions:

- Broadly, we advocate the usefulness of lightweight static analysis for ownership inference.
- Concretely, we present a novel static analysis for inference of Universe types. We implement the analysis and show promising preliminary results.

Outline. Section 2 presents an overview of our analysis in the context of two running examples. Section 3 presents the details of the analysis. Section 4 presents our experiments. Section 5 presents related work. Section 6 discusses the limitations of our analysis and outlines directions for future work. Section 7 concludes the paper.

[1] For the rest of the paper we refer to type modifiers as types.

2. Analysis Overview

Our static analysis follows the steps outlined in [10]. The general idea is to approximate ownership using the owners-as-dominators model hoping for a deep ownership structure. However, the owners-as-dominators model allows arbitrary modifications which must be handled when inferring Universe types.

The four steps, following [10], are the following:

1. Construct the static object graph.
2. Compute the dominance boundary of each object.
3. Assign types to object graph edges.
4. Assign types to fields and variables.

2.1 Example 1

Figure 1 presents our running example. This is a simplified example of a container (class `Container`) and its iterator (class `Iterator`). There are two contexts of usage of the container. Class `Y` creates a container without creating an iterator, while class `X` creates a container and an iterator over the container.

The code presents one possible assignment of Universe types. Reference `cy` in method `Y.my` is `rep` — that is, the `Y` object is the owner of the container it creates. Formal parameter `zy` is of type `any` which forbids the `Y` object from modifying the `Z` object passed as its argument. Similarly, references `cx` and `itx` in method `X.mx` are `rep` — that is, the creating object `X` is the owner of the container and its iterator. Consequently the container and the iterator are peers as specified by the type assignments at lines 12 and 13. Note that field `data` in `Container` is assigned type `peer` which means that the data array has the same owner as its creating container. This is an acceptable assignment since it allows the container object to modify its data array. However, it may not be the most precise or most intuitive assignment (in fact, in Universes, one would have expected type `rep`). Our choice of type `peer` results from our interpretation of `rep` — intuitively, `rep` implies dominance and in this case we do not have dominance because of the access path through the external iterator. Our current analysis infers exactly the types specified in Figure 1. The assignment of type `rep` to field `Container.data` is essentially a limitation of our analysis; we plan to correct this limitation in future work.

The first step in our analysis is to construct an object graph Ag. Ag is a static approximation of all run-time object graphs. The

```
class Main {
  public static void main(String[] args) {
    X x = new X();                                  //ox
    Z z1 = new Z(); x.mx(z1);                       //oz1
    Y y = new Y();                                  //oy
    Z z2 = new Z(); y.my(z2);                       //oz2
} }
class Y {
  void my(any Z zy) {
    rep Container cy;
    cy = new rep Container(10);                    //ocy
    cy.put(zy,0);
    ...
  }
}
class X {
  void mx(any Z zx) {
    rep Container cx;
    rep Iterator itx;
    cx = new rep Container(10);                    //ocx
    cx.put(zx,0);
    itx = cx.getIt();
  }
}
class Container {
  peer any Object[] data;
  Container(int size) {
    data = new peer Object[size];                  //od
  }
  void put(any Object o, index i) {
    data[i] = o;
  }
  Iterator getIt() {
    return new peer Iterator(this);                //oi
  }
}
class Iterator {
  peer any Object[] data;
  Iterator(peer Container c) {
    data = c.data;
  }
  Object getNext(int i) {
    return data[i];
  }
}
```

Figure 1. Example 1.

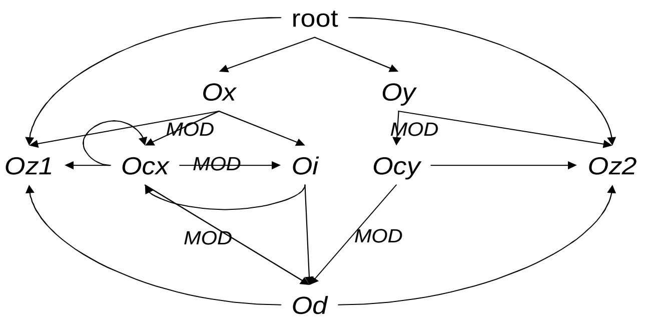

Figure 2. Object graph for Example 1.

object graph for the example in Figure 1 is shown in Figure 2. The nodes in Ag are the object names o — there is an object name for each allocation site in the code. The edges $o \to o'$ represent access relationships. Informally, there are two types of edges: (1) when a field f of o refers to o', and (2) when a method m invoked on receiver o has a local variable that refers to o' (i.e., o has access to o'). Furthermore, some edges $o \to o'$ are annotated with MOD, which denotes that o modifies o' while accessing it. The details on the object graph construction are given in Section 3.1.

The second step in the analysis is to compute dominance boundaries, $Boundary(o)$, for each object name o. $Boundary(o)$ is a subgraph of Ag rooted at o. In our running example, $Boundary(\mathbf{o_x})$ consists of nodes $\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_d}$ and $\mathbf{o_i}$, and the edges between them.[2] Intuitively, the boundary contains paths that are dominated by o. More formally, we conjecture that for every run-time path $o^r \to ... \to o'^r$ whose representative $o \to ... \to o'$ is in $Boundary(o)$, we have that o^r dominates o'^r.[3] The details on the computation of the dominance boundaries are given in Section 3.2.

The third step in the analysis is to assign types to object graph edges. The analysis examines each MOD edge. Initially, it identifies MOD edges that could be assigned `rep` type: these are edges $o \to o'$ such that $o' \in Boundary(o)$ — i.e., o dominates o' and thus o owns o'. In our example, edges $\mathbf{o_x} \to \mathbf{o_{cx}}$, $\mathbf{o_y} \to \mathbf{o_{cy}}$ and $\mathbf{o_{cy}} \to \mathbf{o_d}$ are assigned `rep` type. Edges $\mathbf{o_{cx}} \to \mathbf{o_d}$ and $\mathbf{o_{cx}} \to \mathbf{o_i}$ are not assigned `rep` type because $\mathbf{o_{cx}}$ does not dominate $\mathbf{o_d}$ or $\mathbf{o_i}$ (there are access paths not through $\mathbf{o_{cx}}$ to both $\mathbf{o_i}$ and $\mathbf{o_d}$). The remaining MOD edges (i.e., the ones not assigned `rep` type) are assigned `peer` type. For each such edge the analysis finds the minimal boundaries containing the edge (there might be more than one); intuitively, the minimal boundary gives the closest object which could be the owner of both objects in the `peer` edge. Then it propagates the `peer` type up within each boundary until the owner of the peer objects is reached. In our running example, edge $\mathbf{o_{cx}} \to \mathbf{o_d}$ is identified as `peer`. There is one minimal boundary containing this edge, $Boundary(\mathbf{o_x})$. The analysis propagates the `peer` type up in the boundary; it assigns type `peer` to edges $\mathbf{o_{cx}} \to \mathbf{o_i}$ and $\mathbf{o_i} \to \mathbf{o_{cx}}$, and identifies $\mathbf{o_x}$ as the owner of these peers. The type assignment to object graph edges is explained in detail in Section 3.3.

The fourth step in the analysis is to assign types to fields and variables, resolving conflicting edge types. A field, or a local variable may correspond to edges in different contexts; these edges may have different types. For example, edges $\mathbf{o_{cy}} \to \mathbf{o_d}$ and $\mathbf{o_{cx}} \to \mathbf{o_d}$ which correspond to field `data` in `Container`, have respectively

[2] For the rest of the paper we use bold font (e.g., $\mathbf{o_x}, \mathbf{o_d}, \mathbf{o_i}$) to denote the objects from our running examples. The bold font distinguishes these objects from the generic objects (e.g., o, o_i, o_j) used to for the presentation of the algorithms in Sections 3.1 and 3.2 and throughout the paper.

[3] Although we have good intuition that this statement is true, we have not attempted a formal proof at this point.

```
class Demo {
  public static void main(any any String[] args) {
1   new peer Demo().testA(args.length > 0);   //o_demo
  }
  public void testA(boolean b) {
    rep A a;
2   a = new rep A(b);                         //o_a
  }
}
class A {
  boolean mod;
  peer B b;
  A(boolean m) {
3   mod = m;
4   b = new peer B(this);                     //o_b
  }
  void off() {
5   mod = false;
  }
}
class B {
  peer C c;
  rep D d;
  B(peer A a) {
6   c = new peer C(a);                        //o_c
7   d = new rep D();                          //o_d
  }
}
class C {
  peer A a;
  C(peer A na) {
8   a = na
9   if (a.mod) { a.off(); }
  }
}
class D {
  int i;
  D() {
10  i = 0;
  }
}
```

Figure 3. Example 2.

rep and peer types. In this case, the rep type is changed to peer propagating the new peer type to find the owner of the peers. The type resolution is explained in Section 3.4.

2.2 Example 2

As another example, consider the code in Figure 3; it is taken from [10] with only one minor modification. The corresponding object graph which is constructed from the code during the first step of the analysis, is given in Figure 4.

During the second step, the analysis computes the dominance boundaries: $Boundary(\mathbf{o}_{demo})$ contains all objects, $Boundary(\mathbf{o_a})$ equals $\{\mathbf{o_a}, \mathbf{o_b}, \mathbf{o_c}, \mathbf{o_d}\}$, $Boundary(\mathbf{o_b})$ equals $\{\mathbf{o_b}, \mathbf{o_c}, \mathbf{o_d}\}$, $Boundary(\mathbf{o_c})$ equals $\{\mathbf{o_c}\}$ and $Boundary(\mathbf{o_d})$ equals $\{\mathbf{o_d}\}$.

During the third step, the analysis computes types for object graph edges. Initially, it assigns rep types to all MOD edges but $\mathbf{o_c} \rightarrow \mathbf{o_a}$ (this edge represents modification of an object from an enclosing dominance boundary); edge $\mathbf{o_c} \rightarrow \mathbf{o_a}$ is identified as peer. The analysis then processes peer edge $\mathbf{o_c} \rightarrow \mathbf{o_a}$. There is one minimal boundary that contains this edge, $Boundary(\mathbf{o}_{demo})$. The analysis propagates the peer edge up within this boundary and identifies edges $\mathbf{o_b} \rightarrow \mathbf{o_c}$, $\mathbf{o_a} \rightarrow \mathbf{o_b}$, and $\mathbf{o_b} \rightarrow \mathbf{o_a}$ as peer, and edge $\mathbf{o}_{demo} \rightarrow \mathbf{o_a}$ as rep.

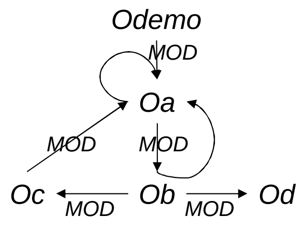

Figure 4. Object graph for Example 2.

There are no conflicting edges and the analysis infers the following types (also shown in Figure 3). Local variable a in method testA is of type rep, field b in class A is of type peer, field c in class B is of type peer and field a in class C is of type peer. Finally, field d in class B is of type rep.

3. Analysis Description

Next, we present the details of our static analysis.

3.1 Object Graph Construction

The object graph construction uses the results of a points-to analysis; specifically, it uses the points-to graph Pt, and the set of methods $PtReach$ identified as reachable by the points-to analysis. Points-to analysis is a well-known static analysis: it computes the set of objects that a reference variable or a reference object field may point to. There are many points-to analyses in the literature. In this work, we use the well-known Andersen's analysis for Java [29, 15].

The analysis constructs Ag, an approximation of all run-time object graphs: if there is a run-time access edge $o_1^r \rightarrow o_2^r$ for some execution, then there is an edge in Ag from the representative of o_1^r to the representative of o_2^r. The nodes in Ag are taken from the set of analysis objects (there is an object name per allocation site), and the edges represent the access relationships.

Figure 5 outlines the construction of Ag. As mentioned earlier, it takes as input Pt and the set of reachable methods $PtReach$. Figure 5 assumes that the program consists of the following kinds of statements: $l = new\ C(r_1)$ (constructor calls), $l = r.n(r_1)$ (virtual calls), $l.f = r$ (instance field writes), $l = r.f$ (instance field reads) and $l = r$ (direct assignments).[4] Notation $\mathcal{RC}_m$ stands for the receivers of method m; sets $\mathcal{RC}_m$ are computed as follows. If m is an instance method, $\mathcal{RC}_m$ equals to the points-to set of the implicit parameter this of m. If m is a static method, $\mathcal{RC}_m$ includes the points-to sets of all implicit parameters this of instance methods n reachable backwards from m on a chain of static calls; if main is reachable backwards from m on a chain of static calls, $\mathcal{RC}_m$ includes the node root.

Lines 1-2 process statements that account for flow due to object creation. New edges are added to Ag from each receiver of the enclosing method m (i.e., $o \in \mathcal{RC}_m$), to the analysis name o_i that represents the newly created object. Intuitively, at object allocation

[4] Assumptions that the program consists of these kinds of statements are often made in program analysis literature in order to simplify the presentation. If necessary, temporary variables may be introduced to achieve this restriction on statements (e.g., $l = r.m().n()$ is transformed into the sequence $r_1 = r.m(); l = r_1.n()$).

```
input  $ReachPt,\ Pt$
output $Ag$
[1] foreach statement $s$ in method $m \in ReachPt$ of kind
      $s_i : l = new\ C(...)$
[2]   add $\{o \to o_i \mid o \in \mathcal{RC}_m\}$ to $Ag$
      // flow into the receiver of $m$ due to object creation
[3] foreach statement $s$ in method $m \in ReachPt$ of kind
      $s : l = r.n(...)$ s.t. $r \neq$ this,
      $s : l = r.f$ s.t. $r \neq$ this
[4]   add $\{o \to o_j \mid o \in \mathcal{RC}_m \wedge o_j \in Pt(l)\}$ to $Ag$
      // flow from callees into the receiver of $m$
[5] foreach statement $s$ in method $m \in ReachPt$ of kind
      $s : l = new\ C(r)$,
      $s : l.n(r)$ s.t. $l \neq$ this,
      $s : l.f = r$ s.t. $l \neq$ this
[6]   add $\{o_i \to o_j \mid o_i \in Pt(l) \wedge o_j \in Pt(r)\}$ to $Ag$
      // flow into the receiver of the callee from $m$
[7] foreach statement $s$ in method $m \in ReachPt$ of kind
      $s : l =$ this,
      $s : r.n($this$)$,
      $s : r.f =$ this
[8] add $\{o_i \to o_i \mid o_i \in Pt($this$)\}$ to $Ag$
      // self-access through a leak of this
```

Figure 5. Construction of Ag. $Pt(l)$ denotes the points-to set of l.

sites (i.e., constructor calls), the newly created object becomes accessible to the receiver of the caller m. Lines 3-4 process statements that account for flow from other objects to the receiver of m. For example, at an instance call not through `this`, new edges are added from each receiver of m (i.e., $o \in \mathcal{RC}_m$) to each returned object o_j. Intuitively, the returned object becomes accessible to the receiver of m. Lines 5-6 process statements that account for flow from m into other objects. For example, at an instance call $l.n(r)$, edges are added from each object o_i in the points-to set of l, to each object o_j in the points-to set of reference argument r. Intuitively, the object passed as actual argument becomes accessible to the receiver of the call. Finally, lines 7-8 take into account that an object may become accessible to itself by accessing implicit parameter `this`.

Next, the analysis infers MOD annotations on object graph edges (i.e., the analysis does not rely on user-provided purity information; it infers this information automatically). Our MOD annotations reflect *direct* modifications. That is, an object graph edge $o_i \to o_j$ is marked as MOD when one of the following is true: (1) a method m called on receiver o_i contains a field write statement $l.f = r$, $l \neq$ `this`, where l refers to o_j, and (2) a method m called on receiver o_i invokes a method n on receiver o_j and n writes o_j directly through a field write statement `this`$.f = r$. Therefore, the analysis examines the following statements:

(1) non-`this` writes $l.f = r$, and

(2) `this` writes `this`$.f = r$.

In case (1), the analysis marks as MOD every edge $o_i \to o_j \in Ag$ such that $o_i \in \mathcal{RC}_m$ and $o_j \in Pt(l)$. In case (2), the analysis first finds all methods m enclosing a call $l.n()$, $l \neq$ `this`, where $l.n()$ leads to the write statement `this`$.f = r$ on a sequence of calls through `this`. Then for each pair m and $l.n()$, the analysis marks as MOD every edge $o_i \to o_j$ such that $o_i \in \mathcal{RC}_m$ and $o_j \in Pt(l)$.

3.1.1 Example 1.

Consider the code in Figure 1 and its corresponding object graph in Figure 2. Object graph edges `root` $\to \mathbf{o_x}$, `root` $\to \mathbf{o_{z1}}$, `root` $\to \mathbf{o_y}$, `root` $\to \mathbf{o_{z2}}$ are due to the object creations at statements 1-4 (lines 1-2 in the algorithm). Edges $\mathbf{o_x} \to \mathbf{o_{z1}}$ and $\mathbf{o_y} \to \mathbf{o_{z2}}$ are due to code lines 2 and 4 respectively (lines 5-6 in the algorithm). Edge $\mathbf{o_x} \to \mathbf{o_i}$ is due to statement 9 (lines 3-4 in the algorithm). Edges $\mathbf{o_{cx}} \to \mathbf{o_{cx}}$, $\mathbf{o_{cx}} \to \mathbf{o_i}$ and $\mathbf{o_i} \to \mathbf{o_{cx}}$ are all due to code line 12. The self-loop edge $\mathbf{o_{cx}} \to \mathbf{o_{cx}}$ is due to the `this` access (lines 7-8 in the algorithm), edge $\mathbf{o_{cx}} \to \mathbf{o_i}$ is due to the creation of the iterator (lines 1-2 in the algorithm), and edge $\mathbf{o_i} \to \mathbf{o_{cx}}$ is due to the parameter passing (lines 5-6 in the algorithm). The rest of the edges are created analogously. Note that statement 14 does not result in any edges because method `getNext` is unreachable from `main`.

Consider the inference of MOD annotations. The analysis infers MOD annotations from program statements 10, 11, and 13. Statement 10 is a `this`$.f = r$ write (case (2)). There are two methods that lead to this write statement: method `my` which contains the constructor call at statement 5, and method `mx` which contains the constructor call at statement 7. Thus, the analysis of the write at statement 10 results in MOD annotations on edges $\mathbf{o_y} \to \mathbf{o_{cy}}$ and $\mathbf{o_x} \to \mathbf{o_{cx}}$. Statement 11 is a non-this write (case (1)). The analysis of this statement leads to MOD annotations on edges $\mathbf{o_{cx}} \to \mathbf{o_d}$ and $\mathbf{o_{cy}} \to \mathbf{o_d}$. Finally, statement 13 is a `this`$.f = r$ write (case (2)). It leads to a MOD annotation on edge $\mathbf{o_{cx}} \to \mathbf{o_i}$. All inferred MOD annotations are shown in Figure 2.

3.1.2 Example 2.

Consider the code in Figure 3 and its corresponding object graph in Figure 4. The object creation at statements 2, 4, 6 and 7 result respectively in edges $\mathbf{o}_{demo} \to \mathbf{o_a}$, $\mathbf{o_a} \to \mathbf{o_b}$, $\mathbf{o_b} \to \mathbf{o_c}$, and $\mathbf{o_b} \to \mathbf{o_d}$ (lines 1-2 in the algorithm in Figure 5). Statement 4 results in two additional edges: edge $\mathbf{o_b} \to \mathbf{o_a}$ due to the parameter passing (lines 5-6 in the algorithm), and self-edge $\mathbf{o_a} \to \mathbf{o_a}$ due to the `this` access (lines 7-8 in the algorithm). Statement 6 results in edge $\mathbf{o_c} \to \mathbf{o_a}$ (lines 5-6 in the algorithm).

Statements 3 and 4, 5, 6 and 7, 8, and finally 10 are field write statements that lead to MOD annotations. Statements 3 and 4 are `this`$.f = r$ writes. There is one method that leads to these writes, method `testA` which contains the constructor call at line 2. These statements result in a MOD annotation on edge $\mathbf{o}_{demo} \to \mathbf{o_a}$. Statement 5 is a `this`$.f = r$ write as well. The constructor `C.C` leads to this statement through the call at line 9. Thus, statement 5 results in a MOD annotation on edge $\mathbf{o_c} \to \mathbf{o_a}$. Statements 6 and 7 are `this`$.f = r$ writes as well. There is one method that leads to these writes, the constructor `A.A` through the constructor call at line 4. Statements 6 and 7 result in a MOD annotation on edge $\mathbf{o_a} \to \mathbf{o_b}$. Statements 8 and 10 result in MOD annotations respectively on edges $\mathbf{o_b} \to \mathbf{o_c}$ and $\mathbf{o_b} \to \mathbf{o_d}$.

3.2 Dominance Boundary Computation

Next, we use the object graph to compute dominance boundaries.

Procedure *findBoundary* in Figure 6 computes $Boundary(o_i)$. It makes use of an auxiliary procedure $computeClosure$, which decides whether a new node o_j is added to the boundary; in addition, $computeClosure$ computes the *closure* of o_j given the current boundary of o_i — essentially, the closure is the set of nodes where o_j could flow to within the boundary of o_i.

Procedure *findBoundary* examines cut edges $o_k \to o_j$ (i.e., edges where $o_k \in Boundary(o_i)$ and $o_j \notin Boundary(o_i)$) — that is, o_j is a node connected to the boundary of o_i, but not yet in the boundary of o_i. While there is such a node o_j, which is not examined yet, *findBoundary* calls $computeClosure$ on o_j; if the result returned by $computeClosure$ is true, the computed closure of o_j, namely $Closure(o_j, o_i)$, is added to the boundary of o_i; otherwise, nothing is added to the boundary of o_i. At the end of procedure *findBoundary*, $pruneBoundary$ removes certain infeasible nodes from $Boundary(o_i)$.

The main idea of our analysis is to track the flow of an object o through *edge triples* $o_1 \to o, o_2 \to o, o_1 \to o_2$ (also denoted by the ordered set of nodes o_1, o, o_2). Intuitively, o can flow from

```
procedure $computeClosure$
input  $Ag,\ o_j,\ o_i,\ Boundary(o_i)$
output $Closure(o_j, o_i)$,  result: boolean
[1]  if $isOutside(o_i \rightarrow o_j)$ return false
[2]  $Closure$=$\{o_j\}$, $W$=$\{o_k \mid o_k \rightarrow o_j \wedge o_k \in Boundary(o_i)\}$
[3]  while $W$ not empty
[4]     take $o_k$ from $W$
[5]     foreach $o_k \rightarrow o_m \in Closure \cup Boundary(o_i)$
[6]        foreach valid triple $o_k, o_m, o_n$, s.t. $o_n$ is new to $W$
[7]           if $isOutside(o_i \rightarrow o_n)$ return false
[8]           else add $o_n$ to $Closure$ and to $W$
[9]     foreach $o_m \rightarrow o_k \in Closure \cup Boundary(o_i)$
[10]       foreach valid triple $o_m, o_k, o_n$, s.t. $o_n$ is new to $W$
[11]          if $isOutside(o_i \rightarrow o_n)$ return false
[12]          else add $o_n$ to $Closure$ and to $W$
[13] $Closure(o_j, o_i)$ = $Closure$;
[14] return true

procedure $findBoundary$
input  $Ag,\ o_i$
output $Boundary(o_i)$
[1] $Boundary(o_i) = \{o_i\}$
[2] while there is new $o_j$ s.t. there is a cut edge $o_k \rightarrow o_j$
[3]    if $computeClosure(o_j, o_i) ==$ true
[4]       add $Closure(o_j, o_i)$ to $Boundary(o_i)$
[5] $pruneBoundary(Boundary(o_i))$
```

Figure 6. Boundary computation. $computeClosure$ computes the closure of o_j given the boundary of o_i, and $findBoundary$ computes the boundary of o_i.

object o_1 to object o_2 only if o_1 has access edge to o_2 (or symmetrically o can flow from o_2 to o_1 only if o_1 had access edge to o_2). Hence, the analysis examines the specified edge triple. In our running example in Figure 2, triple $\mathbf{o_x} \rightarrow \mathbf{o_{z1}}, \mathbf{o_{cx}} \rightarrow \mathbf{o_{z1}}, \mathbf{o_x} \rightarrow \mathbf{o_{cx}}$ denotes that $\mathbf{o_{z1}}$ flows from $\mathbf{o_x}$ to $\mathbf{o_{cx}}$. Essentially, the tracking of triples compensates for the imprecision of the underlying points-to analysis which often uses the same object name for distinct run-time objects. In our first example name $\mathbf{o_d}$ denotes two distinct data arrays (one in container $\mathbf{o_{cx}}$ and the other in container $\mathbf{o_{cy}}$). A naive inference analysis may conclude that $\mathbf{o_d}$ flows from $\mathbf{o_{cx}}$ to $\mathbf{o_{cy}}$ and therefore the dominator of $\mathbf{o_d}$ is `root`. Our analysis concludes that there is no valid "triple path" from $\mathbf{o_{cx}} \rightarrow \mathbf{o_d}$ to $\mathbf{o_{cy}} \rightarrow \mathbf{o_d}$; it infers the precise dominators.

Not all triples in the object graph are triples that represent valid flow. For example, consider triple $\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_i}$. It implies that either (1) $\mathbf{o_{cx}}$ flows from $\mathbf{o_x}$ into $\mathbf{o_i}$, or (2) $\mathbf{o_{cx}}$ flows from $\mathbf{o_i}$ into $\mathbf{o_{cx}}$. It is easy to see that neither is true, and therefore triple $\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_i}$ is not a valid triple. On the other hand, triple $\mathbf{o_{cx}}, \mathbf{o_{cx}}, \mathbf{o_i}$ is valid: it implies that $\mathbf{o_{cx}}$ flows from itself (through the `this`-reference actually) to $\mathbf{o_i}$, which is true. Procedure $computeClosure$ makes use of a predicate (lines 6 and 10) which is able to filter out some of the invalid triples; this is explained in [18].

Consider procedure $computeClosure$. It makes use of a predicate $isOutside(o_i \rightarrow o_j)$ (lines 1, 7 and 11) — an edge $o_i \rightarrow o_j$ is an *outside edge* if there exists a valid triple o_k, o_j, o_i. Intuitively, $isOutside$ conservatively captures the situation when some o_j flows from (or into) an "outside" object o_k and therefore there may be an access path to o_j that does not pass through o_i. In Figure 2, edge $\mathbf{o_{cx}} \rightarrow \mathbf{o_{z1}}$ is an outside edge. Clearly, the `Z` object $\mathbf{o_{z1}}$ is passed from the `X` object to the `Container` object $\mathbf{o_{cx}}$, and $\mathbf{o_{cx}}$ does not dominate $\mathbf{o_{z1}}$. When a new potential node o_j is examined, $computeClosure$ first examines edge $o_i \rightarrow o_j$; if it is not an outside edge, it proceeds to compute the $Closure$ of o_j given o_i and its current boundary. $computeClosure$ finds the paths from o_i to o_j. It examines each edge $o_1 \rightarrow o_2$ on such a path, which is in the union of the current closure and the boundary of o_i (i.e., $Closure \cup Boundary(o_i)$). Then it discovers nodes o_3, such that there is a valid triple $o_1 \rightarrow o_2$, $o_1 \rightarrow o_3$ and $o_3 \rightarrow o_2$, and adds o_3 to $Closure$. If o_2 is added to the union before o_1, o_3 is discovered through lines 5-8; otherwise it is discovered through lines 9-12. If $computeClosure$ detects a path that originates in an outside edge from o_i, it returns false (lines 7 and 11).

Finally, procedure $pruneBoundary$ removes certain nodes from the boundary of o_i — these nodes are added as a result of imprecision in the points-to analysis and contribute only infeasible access paths. The pruning is based on the following observation: an object can be in a boundary *only if* it is *created* in that boundary. Therefore, $pruneBoundary$ examines each node $o_j \in Boundary(o_i)$, $o_j \neq o_i$; then it examines all incoming edge $o_k \rightarrow o_j \in Boundary(o_i)$ — if none of these edges is due to object creation (i.e., due to lines 1-2 in the object graph construction algorithm in Figure 5), node o_j and all incoming and outgoing edges are removed from the boundary.

We conjecture that if the representative of a run-time object graph path $o^r \rightarrow \ldots \rightarrow o'^r$ is in the boundary of o (the representative of o^r), then we have that o^r dominates o'^r; however, we do not have a proof of this statement. We do not discuss static fields; however, the implementation handles this case.

3.2.1 Example 1.

Let us compute the boundary of $\mathbf{o_x}$ in Figure 2.

Let o_j be $\mathbf{o_{z1}}$ in the first iteration of the loop in $findBoundary$. In this case, $computeBoundary$ determines that edge $\mathbf{o_x} \rightarrow \mathbf{o_{z1}}$ is an outside edge and immediately returns false.

Let o_j be $\mathbf{o_{cx}}$ in the second iteration of the loop. At line 2, $computeBoundary$ initializes $Closure = \{\mathbf{o_{cx}}\}$ and $W = \{\mathbf{o_x}\}$. At line 4, it takes $\mathbf{o_x}$ off the worklist. At line 5, it examines edge $\mathbf{o_x} \rightarrow \mathbf{o_{cx}}$ and triples $\mathbf{o_x}, \mathbf{o_{cx}}, o_n$. The first triple is $\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_i}$, but this is not a valid triple and it is discarded without adding $\mathbf{o_i}$ to the closure and worklist. The next triple is $\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_{cx}}$, which is a valid triple and $\mathbf{o_{cx}}$ is added to the worklist. No nodes are added at lines 9-12 as no parents of $\mathbf{o_x}$ are in the current union of $Closure$ and $Boundary(\mathbf{o_x})$. Next, $\mathbf{o_{cx}}$ is taken off the worklist. At line 5, the procedure examines self-edge $\mathbf{o_{cx}} \rightarrow \mathbf{o_{cx}}$ and valid triple $\mathbf{o_{cx}}, \mathbf{o_{cx}}, \mathbf{o_i}$ subsequently adding $\mathbf{o_i}$ to $Closure$ and W. No new nodes are added for $\mathbf{o_{cx}}$ computing $Closure(\mathbf{o_{cx}}, \mathbf{o_x}) = \{\mathbf{o_{cx}}, \mathbf{o_i}\}$, and adding this set to $Boundary(\mathbf{o_x})$.

Let o_j be $\mathbf{o_d}$ in the third iteration of the loop in $findBoundary$. $computeClosure$ returns true and adds $Closure(\mathbf{o_d}, \mathbf{o_x}) = \{\mathbf{o_d}\}$ to $Boundary(\mathbf{o_x})$. The boundary so far is $\{\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_i}, \mathbf{o_d}\}$.

There is only one node connected to the boundary that has not been examined, namely node $\mathbf{o_{z2}}$. The final iteration examines $\mathbf{o_{z2}}$ and adds $Closure(\mathbf{o_{z2}}, \mathbf{o_x}) = \{\mathbf{o_{z2}}\}$ to $Boundary(\mathbf{o_x})$. The resulting $Boundary(\mathbf{o_x})$ is $\{\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_i}, \mathbf{o_d}, \mathbf{o_{z2}}\}$. Note however that the addition of $\mathbf{o_{z2}}$ is infeasible — it is easy to see that there is no run-time access path from $\mathbf{o_x}$ to $\mathbf{o_{z2}}$. The culprit is the imprecision of the underlying points-to analysis which represents the two data arrays with a single analysis object, $\mathbf{o_d}$. The dominance boundary computation adds node $\mathbf{o_{z2}}$ to the boundary of $\mathbf{o_x}$ because it cannot distinguish that connecting edge $\mathbf{o_d} \rightarrow \mathbf{o_{z2}}$ is not valid in the context of $\mathbf{o_x}$ (it is valid in the context of $\mathbf{o_y}$).

Finally, $pruneBoundary$ examines each node (other than $\mathbf{o_x}$) in the final boundary. There are creation edges for nodes $\mathbf{o_{cx}}$, $\mathbf{o_i}$ and $\mathbf{o_d}$ (respectively $\mathbf{o_x} \rightarrow \mathbf{o_{cx}}$, $\mathbf{o_{cx}} \rightarrow \mathbf{o_i}$ and $\mathbf{o_{cx}} \rightarrow \mathbf{o_d}$. There are no creation edges for node $\mathbf{o_{z2}}$ and this node is removed from the boundary. The final boundary, $Boundary(\mathbf{o_x})$ equals $\{\mathbf{o_x}, \mathbf{o_{cx}}, \mathbf{o_d}, \mathbf{o_i}\}$.

3.2.2 Example 2.

As another example, consider the object graph in Figure 4, and let us compute the boundary of $\mathbf{o_a}$. The first iteration of the

```
procedure $propagatePeer$
input   $o_i \to o_j$,   all boundaries
[1]  $minimal$ = compute set of minimal boundaries for $o_i \to o_j$

[2]  foreach boundary $Boundary(o_k) \in minimal$
[3]     $W = \{o_i \to o_j\}$
[4]     while $W$ not empty
[5]       take $o_1 \to o_2$ from $W$
[6]       if $o_1 \to o_2$ is a self-edge continue
[7]       else if $o_1 == o_k$ mark $o_1 \to o_2$ as rep
[8]       else
[9]         mark $o_1 \to o_2$ as peer
[10]        foreach $o_3 \to o_1 \in Boundary(o_k)$, s.t. $o_3 \to o_1$ new to $W$
[11]          add $o_3 \to o_1$ to $W$
```

Figure 7. Propagation of a peer edge.

while loop in *findBoundary* examines $\mathbf{o_b}$ and *computeClosure* adds $\{\mathbf{o_b}, \mathbf{o_c}\}$ to $Boundary(\mathbf{o_a})$. The next iteration examines $\mathbf{o_d}$ and *computeClosure* adds $\{\mathbf{o_d}\}$, resulting in $Boundary(\mathbf{o_a}) = \{\mathbf{o_a}, \mathbf{o_b}, \mathbf{o_c}, \mathbf{o_d}\}$. Nothing can be pruned away.

3.3 Type Assignment to Edges

The type assignment proceeds as follows. If an edge with a MOD annotation is contained in the boundary of its source (i.e., in the boundary of the object that does the direct modification), the edge is identified as `rep`. However, there might be modifications deeper in the boundary (e.g., MOD edge $\mathbf{o_{cx}} \to \mathbf{o_d}$ in Figure 2) and modifications to objects that belong to an enclosing boundary (e.g., MOD edge $\mathbf{o_c} \to \mathbf{o_a}$ in Figure 4); these MOD edges require that the source and the target are peers and the analysis identifies them as `peer`. Subsequently, the analysis invokes procedure $propagatePeer(o_i \to o_j)$ which propagates the `peer` type upward looking for the common owner of o_i and o_j.

Figure 7 outlines procedure $propagatePeer(o_i \to o_j)$. Line 1 identifies the set of minimal boundaries containing $o_i \to o_j$. A minimal boundary is a boundary $Boundary(o_k)$ where $o_k \neq o_j$ and $o_k \neq o_i$ and there is no subset boundary, $Boundary(o_{k'})$, containing the edge. Intuitively, the minimal boundary is the deepest boundary containing the modification that forces the `peer` type, and o_k is the closest possible owner of both o_i and o_j. Note that an edge may appear in different contexts, which would result in different minimal boundaries. For example, suppose that the iterator in Figure 1 modifies the data array and $\mathbf{o_i} \to \mathbf{o_d}$ is a MOD edge; suppose also that the iterator is instantiated in the context of $\mathbf{o_y}$ as well. Then edge $\mathbf{o_i} \to \mathbf{o_d}$ would need to be examined in the boundaries of both $\mathbf{o_x}$ and $\mathbf{o_y}$ (i.e., *minimal* would equal to $\{Boundary(\mathbf{o_x}), Boundary(\mathbf{o_y})\}$. Lines 4-11 propagate the `peer` type within each minimal boundary $Boundary(o_k)$. The propagation starts at edge $o_i \to o_j$ and proceeds backwards to find all paths to o_k. Lines 6-9 identify each visited edge as `peer`, except for self-edges (skipped at line 6) and the first edge on a path (line 7). The first edge is identified as `rep` (therefore identifying the owner of the peers). Note that during this step an edge previously identified as `rep` could be changed to `peer`.

An edge can have one of three possible types: `rep`, `peer`, or `any` (if no type has been assigned or propagated to an edge, we assign the default `any` type to it).

3.3.1 Example 1.

Recall the object graph in Figure 2. First, the analysis identifies MOD edges $\mathbf{o_x} \to \mathbf{o_{cx}}$, $\mathbf{o_y} \to \mathbf{o_{cy}}$ and $\mathbf{o_{cy}} \to \mathbf{o_d}$ as `rep` (these edges are contained in the boundary of their source). MOD edges $\mathbf{o_{cx}} \to \mathbf{o_d}$ and $\mathbf{o_{cx}} \to \mathbf{o_i}$ are identified as `peer`.

Consider MOD edge $\mathbf{o_{cx}} \to \mathbf{o_d}$. It is identified as `peer` and *propagatePeer* is called on it. There is one minimal boundary that contains this edge, namely $Boundary(\mathbf{o_x})$. *propagatePeer* visits edges $\mathbf{o_{cx}} \to \mathbf{o_d}$, $\mathbf{o_i} \to \mathbf{o_{cx}}$ and $\mathbf{o_{cx}} \to \mathbf{o_i}$, likely in this order, and assigns type `peer` to them. It also visits edges $\mathbf{o_x} \to \mathbf{o_{cx}}$ and $\mathbf{o_x} \to \mathbf{o_i}$ and assigns type `rep` to them.

The complete type assignment to the edges of the object graph in Figure 2 is the following. Edges `root` $\to \mathbf{o_{z1}}$, `root` $\to \mathbf{o_{z2}}$, `root` $\to \mathbf{o_x}$ and `root` $\to \mathbf{o_y}$ receive type `any`. Edges $\mathbf{o_x} \to \mathbf{o_{z1}}$, $\mathbf{o_{cx}} \to \mathbf{o_{z1}}$ and $\mathbf{o_d} \to \mathbf{o_{z1}}$ receive type `any`. Edges $\mathbf{o_y} \to \mathbf{o_{z2}}$, $\mathbf{o_{cy}} \to \mathbf{o_{z2}}$ and $\mathbf{o_d} \to \mathbf{o_{z2}}$ receive type `any` as well. Edges $\mathbf{o_x} \to \mathbf{o_{cx}}$ and $\mathbf{o_x} \to \mathbf{o_i}$ receive type `rep`. Edge $\mathbf{o_y} \to \mathbf{o_{cy}}$ receives type `rep` as well. Edges $\mathbf{o_{cx}} \to \mathbf{o_i}$, $\mathbf{o_i} \to \mathbf{o_{cx}}$ and $\mathbf{o_{cx}} \to \mathbf{o_d}$ receive type `peer`. Edge $\mathbf{o_{cy}} \to \mathbf{o_d}$ receives type `rep`. Finally, edge $\mathbf{o_i} \to \mathbf{o_d}$ receives type `any`.

3.3.2 Example 2.

Recall the object graph in Figure 4. Initially, the analysis assigns `rep` type to MOD edges $\mathbf{o}_{demo} \to \mathbf{o_a}$, $\mathbf{o_a} \to \mathbf{o_b}$, $\mathbf{o_b} \to \mathbf{o_c}$ and $\mathbf{o_b} \to \mathbf{o_d}$ (these edges are contained in the boundary of their source). Edge $\mathbf{o_c} \to \mathbf{o_a}$ is identified as `peer` and *propagatePeer* is called on it. *propagatePeer* identifies the only minimal boundary of $\mathbf{o_c} \to \mathbf{o_a}$, namely $Boundary(\mathbf{o}_{demo})$. It visits edges $\mathbf{o_c} \to \mathbf{o_a}$, $\mathbf{o_b} \to \mathbf{o_c}$, $\mathbf{o_a} \to \mathbf{o_b}$ and $\mathbf{o_b} \to \mathbf{o_a}$ and identifies these edges as `peer`. It also visits $\mathbf{o}_{demo} \to \mathbf{o_a}$ and identifies this edge as `rep`.

3.4 Type Assignment to Fields and Variables

The mapping of edge types to field and variable types proceeds as follows. First, the analysis identifies a set of object graph edges that correspond to a given field f, or a local variable l. The set of edges for f, $Edges(f)$ is computed as follows: $Edges(f) = \{o_i \to o_j \in Ag \mid o_i \xrightarrow{f} o_j \in Pt\}$ (i.e., it includes every edge in the object graph which is an f-labeled field edge in the points-to graph). The set of edges for local variable l in instance method m is computed as follows: $Edges(l) = \{o_i \to o_j \in Ag \mid o_i \in Pt(\mathtt{this}_m) \wedge o_j \in Pt(l)\}$. For example, the edges for field `data` in class `Container` are $\mathbf{o_{cx}} \to \mathbf{o_d}$ and $\mathbf{o_{cx}} \to \mathbf{o_d}$. The edges for formal parameter o in method `put` are $\mathbf{o_{cx}} \to \mathbf{o_{z1}}$ and $\mathbf{o_{cy}} \to \mathbf{o_{z2}}$ (o points to $\mathbf{o_{z1}}$ when `put` is invoked on receiver $\mathbf{o_{cx}}$, and to $\mathbf{o_{z2}}$ when `put` is invoked on receiver $\mathbf{o_{cy}}$).

Next, the analysis resolves conflicts. There are three potential cases: (1) `any` and `rep`, (2) `any` and `peer`, and (3) `rep` and `peer`. Consider case (1) and let the two conflicting cases be $o_i \overset{\mathtt{any}}{\to} o_j$ and $o_{i'} \overset{\mathtt{rep}}{\to} o_{j'}$: if $o_j \in Boundary(o_i)$ (i.e., case (1.1)) the analysis assigns type `rep` to both; otherwise (i.e., case (1.2)), it assigns type `peer` to both $o_i \to o_j$ and $o_{i'} \to o_{j'}$, and calls *propagatePeer* on both. Cases (2) and (3) are treated analogously to case (1.2): the two edges are assigned type `peer` and the newly found `peer` edge is propagated using *propagatePeer*. We expect that with additional experiments we would improve our insight into the problem of conflict resolution, and would likely develop a more precise procedure for handling of conflicts.

3.4.1 Example 1.

Consider the inferred types for field `Container.data` and variable `put.o` in Figure 1. We have $Edges(\mathtt{Container.data}) = \{\mathbf{o_{cx}} \overset{\mathtt{peer}}{\to} \mathbf{o_d}, \mathbf{o_{cy}} \overset{\mathtt{rep}}{\to} \mathbf{o_d}\}$; therefore the inferred type for `Container.data` is `peer`. Also, $Edges(\mathtt{put.o}) = \{\mathbf{o_{cx}} \overset{\mathtt{any}}{\to} \mathbf{o_{z1}}, \mathbf{o_{cy}} \overset{\mathtt{any}}{\to} \mathbf{o_{z2}}\}$; therefore the inferred type for `put.o` is `any`.

(1)Component	(2)Functionality	(3)#Classes/ #Functionality	(4)#Methods	(5)Points-to time[sec]	(6)Inference time[sec]
`gzip`	GZIP IO streams	199/6	3481	82	32
`zip`	ZIP IO streams	194/6	3506	84	41
`checked`	IO streams and checksums	189/4	3428	82	22
`collator`	text collation	203/15	3535	83	82
`breaks`	text breaks	193/13	3487	82	30
`number`	number formatting	198/10	3541	85	62

Table 1. Information on Java components.

3.4.2 Example 2.

There are no conflicts in our second example. $Edges(\texttt{testA.a}) = \{\mathbf{o}_{demo} \overset{\texttt{rep}}{\rightarrow} \mathbf{o_a}\}$ and therefore variable `testA.a` receives type `rep`. $Edges(\texttt{A.b}) = \{\mathbf{o_a} \overset{\texttt{peer}}{\rightarrow} \mathbf{o_b}\}$ and therefore field `A.b` receives type `peer`. $Edges(\texttt{B.c}) = \{\mathbf{o_b} \overset{\texttt{peer}}{\rightarrow} \mathbf{o_c}\}$ and field `B.c` receives type `peer`; also, $Edges(\texttt{B.a}) = \{\mathbf{o_b} \overset{\texttt{peer}}{\rightarrow} \mathbf{o_a}\}$ and formal parameter `a` in constructor `B.B` receives type `peer` as well. $Edges(\texttt{B.d}) = \{\mathbf{o_b} \overset{\texttt{rep}}{\rightarrow} \mathbf{o_d}\}$ and field `B.d` receives type `rep`. Finally, $Edges(\texttt{C.a}) = Edges(\texttt{C.na}) = \{\mathbf{o_c} \overset{\texttt{peer}}{\rightarrow} \mathbf{o_a}\}$ and both field `C.a` and formal parameter `na` in constructor `C.C` receive type `peer`.

4. Experiments

We implemented our analysis and performed limited experiments.

4.1 Infrastructure and Benchmarks

The static analysis is implemented in Java using the Soot 2.2.3 [31] and Spark [15] frameworks. It uses the Andersen-style points-to analysis provided by Spark. We performed the analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a 900MHz Sun Fire 380R machine with 4GB of RAM.

We evaluated the analysis on several Java components from the packages `java.util.zip` and `java.text`. We have used these components in previous work [20, 17].[5] Each component contains a set of classes that provide certain functionality (i.e., these are the functionality classes); arbitrary clients utilizing this functionality can be written on top of the component. Information on these components is shown in the first 4 columns in Table 1; column 3 shows the number of reachable classes (including library classes) and the number of functionality classes, and column 4 shows the number of reachable methods (including library methods).

Clearly, the components are incomplete programs and the points-to and object graph construction analyses, which require whole programs, cannot be applied directly. We address this issue by making use of a technique called fragment analysis [30]—the fragment analysis completes the component by attaching a conservative artificial `main` method to the set of component classes, thus allowing whole program analysis. Our analysis is performed on the completed component and the inferred types are valid across all possible clients of the component.

4.2 Results

We applied the inference analyses on reference fields in functionality classes, a total of 46 fields.[6] The last two columns in Table 1 show the running time of the points-to analysis and the running time of the inference analysis (which includes object graph construction, boundary inference, edge type inference and resolution of field types). These results are preliminary and the implementation is far from optimal. Still, we believe that the results are promising and the analysis would scale well to larger programs.

Component	#Fields	`#rep`	`#peer`	`#any`	`#root`
`gzip`	7	3	1	3	0
`zip`	10	5	3	2	0
`checked`	2	0	2	0	0
`collator`	17	8	2	5	2
`breaks`	7	0	2	5	0
`number`	3	1	2	0	0

Table 2. Inferred field types.

Table 2 shows the inferred types for these fields. In the implementation we do not propagate upward `peer` edges whose enclosing boundary is `root`; these edges automatically receive type `root`. The `root` type is designed to capture edges that represent access to static fields (e.g., one may assign a static field to an instance field of an object as in `new A(staticField)`; the object may later modify the instance field). Hence, table 2 includes 4 types: `rep`, `peer`, `any` and `root`.

We examined manually the fields in components `zip` and `collator`, 27 in total. In all but 3 cases, the analysis inferred the most precise type. All cases of imprecision were in class `MergeCollation` in component `collator`: fields `saveEntry` and `lastEntry` were reported to have type `root` instead of `rep`, and field `patterns` was reported to have type `peer` instead of `rep`. We have traced the first two cases, and found that the imprecision originated in imprecision in the underlying context-insensitive points-to analysis. In our future work we will investigate more precise points-to analyses [21, 16] for the purposes of ownership inference.

Although preliminary and limited, we believe that these results are promising: relatively simple, lightweight static analysis could be useful for the purpose of inference of ownership types.

5. Related Work

There is a large body of work on ownership type systems [25, 6, 3, 5, 4, 14, 9, 23, 2, 27]. Unlike our work it focuses on type-theoretic approaches and requires type annotations provided by the programmer. Generally, these approaches require extensions of the language, compiler and run-time environment and therefore, it may be difficult to adopt in software engineering practice. Our approach uses automatic inference and works directly on Java code; we believe that such automatic inference may help advance reasoning about ownership in practice.

Somewhat surprisingly, less has been done on ownership inference. Most inference analyses are dynamic [1, 22, 10, 28], although static analysis has the important advantage of safety. Work on static inference of ownership-related properties includes the confinement

[5] The current set of components does not include `date`, one of the components used in previous work. This is due to the fact that we were unable to run `date` through the points-to analysis from our current Soot-based infrastructure.

[6] We exclude fields of type `String` and `StringBuffer`.

analyses in [12] and [7], the memory leak analysis in [13], and the analysis for inference of AliasJava types in [3]. Poetzsch-Heffter et al. [26] present a new ownership type system with low annotation overhead, and a corresponding inference analysis. Ma and Foster [19] infer ownership and combine reasoning about ownership with reasoning about uniqueness. Our work presents conceptually different analysis from previous work and focuses on inference of Universe types.

The Master's theses in [24] and [11] present analyses and tools specifically for the inference of Universe types; the first presents a static analysis and a tool based on an SAT-solver, and the latter extends the SAT-based static analysis by combining it with a dynamic analysis. This approach is different from ours. It reduces the inference problem to a MAX-SAT problem, while our approach uses an analysis based on points-to information. The advantage of this approach is that it may produce more precise results than ours; also it may benefit from new and efficient SAT solvers. On the other hand, the SAT-based analysis is exponential in the worst case, while our analysis is polynomial (its worst-case complexity is $O(n^5)$, although in practice it seems comparable to the Andersen-style points-to analysis). We note however, that all of these analyses are preliminary. The work in [24] and [11] focuses more on the building of the inference tools and less on the evaluation of the underlying analysis. Our work is preliminary as well; we have not built a usable tool, and while we focus more on the analysis, our evaluation on 6 small Java components is clearly limited. Overall, we believe that ownership inference is still a relatively unexplored direction.

Finally, we contrast this paper with our previous work presented in [18]. The general idea is the same as here—we use lightweight static analysis to reason about the ownership structure of a program. However, the work in [18] focuses on ownership inference according to the owners-as-dominators model and does not attempt to map inference results to types. Our current work focuses on inference of Universe types which requires significant extension of the boundary computation from [18]; also, it attempts to map inference results to Universe types.

6. Discussion of Future Work

We note that this work is preliminary and there are many open questions and opportunities for improvement. Below we discuss some of these issues.

The first issue is the correctness of the boundary computation in Section 3.2 as well as correctness in general; we have intuition that the analysis is correct and the empirical results reinforce this intuition; however, formal reasoning is necessary.

The second issue is that currently, the types of static methods are not handled. However, we believe that the analysis could be easily extended to handle static methods.

The third issue is our choice of MOD (recall that our analysis marks an edge $o \rightarrow o'$ as MOD only if o directly modifies o'—that is, a method called on receiver o calls a method m on receiver o' and m writes `this`). There are several ways to define whether a method m is MOD-free. The most conservative definition, (1), would require that m never writes an object (i.e., it never reaches a field write statement $p.f = ...$). The least conservative definition, (2), would require that m does not write `this` (i.e., there is no direct modification of the receiver of m, but modifications to objects transitively reachable from the receiver are allowed). Yet another definition, (3), in between these two, would require that m does not write the visible state (i.e., m could write temporary objects in its scope, but these objects could not be returned to the caller of m).

Although our choice of MOD was originally motivated by ease of implementation, we now believe that it might be a good choice. Choice (1) is overly conservative—it would propagate modifications up in many dominance boundaries and may lead to imprecise assignment of types. Choice (3) is conservative as well—modifications again would be propagated up in boundaries, far up from the actual modification. In contrast, our choice, choice (2), confines modifications as deep in a boundary as possible, and may lead to more meaningful assignment of types. This is a question that we plan to investigate in the future.

The fourth issue is the precision of the assigned types. Consider edge $\mathbf{o_{cx}} \rightarrow \mathbf{o_d}$ in Figure 2. Our analysis assigns type `peer` to this edge due to the access path to $\mathbf{o_d}$ through $\mathbf{o_i}$—the analysis concludes that $\mathbf{o_{cx}}$ does not dominate $\mathbf{o_d}$ and the owner of $\mathbf{o_{cx}}$ and $\mathbf{o_d}$ appears up in an enclosing boundary. Note however that the path through $\mathbf{o_i}$ does not modify $\mathbf{o_d}$ and the analysis could have done better and assigned type `rep` to edge $\mathbf{o_{cx}} \rightarrow \mathbf{o_d}$ and consequently to field `Container.data`. This is another problem that we will investigate in future work.

The final, and most important issue is analysis scalability. Currently, we have a whole program analysis which in addition to user-defined classes and methods processes hundreds of library classes and thousands of library methods, most of them irrelevant. For example, an unsafe version of the inference analysis which does not process libraries, completes in 2 seconds on `collator`; in contrast, the safe version which processes reachable libraries completes in 82 seconds. In our future work, we would like to safely and efficiently separate relevant library code from irrelevant library code; this may lead to improvements in analysis scalability.

7. Conclusions

This paper presented a static analysis for inference of Universe types. First, the analysis constructs a static object graph, second, it computes dominance boundaries for each object, third, it infers types for object graph edges, and fourth, it infers types for fields and variables. We implemented the analysis and our preliminary results are promising. Finally, we outlined directions for future work.

References

[1] R. Agarwal and S. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, pages 149–160, 2004.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, pages 1–25, 2004.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.

[4] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[5] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[6] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[7] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment time confinement checking. In *OOPSLA*, pages 374–387, 2003.

[8] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, pages 28–53, 2007.

[9] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[10] W. Dietl and P. Müller. Runtime Universe type inference. In *IWACO*, 2007.

[11] A. Fuerer. Combining run-time and static Universe Type Inference. Master's thesis, ETH Zurich, 2007. `http://sct.inf.ethz.ch/projects/students_docs/Andreas_Fuerer`.

[12] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, pages 241–253, 2001.

[13] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.

[14] P. Lam and M. Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *ECOOP*, pages 275–302, 2003.

[15] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.

[16] O. Lhotak and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC*, pages 47–64, 2006.

[17] Y. Liu and A. Milanova. Static analysis for dynamic coupling measures. In *CASCON*, 2006.

[18] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.

[19] K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.

[20] A. Milanova. Precise identification of composition relationships for UML class diagrams. In *ASE*, pages 76–85, 2005.

[21] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 14(1):1–42, 2005.

[22] N. Mitchell. The runtime structure of object ownership. In *ECOOP*, pages 74–98, 2006.

[23] P. Müller and A. Rudich. Ownership transfer in Universe types. In *OOPSLA*, pages 461–478, 2007.

[24] M. Niklaus. Static Universe Type Inference using a SAT-solver. Master's thesis, ETH Zurich, 2006. `http://sct.inf.ethz.ch/projects/students_docs/Mattias_Niklaus`.

[25] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.

[26] A. Poetzsch-Heffter, K. Geilmann, and J. Schafer. Infering ownerhsip types for encapsulated object-oriented program components. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm on the occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 120–144. Springer, 2007.

[27] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, pages 311–324, 2006.

[28] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *ASE*, pages 194–203, 2007.

[29] A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

[30] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE TSE*, 30(6):372–386, 2004.

[31] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, pages 18–34, 2000.