Static Dominance Inference

Ana Milanova[1] and Jan Vitek[2]

Rensselaer Polytechnic Institute [2] Purdue University

Abstract. Dominance, the property that all paths to a given object must go through another object, is at the heart of ownership type disciplines. While ownership types have received abundant attention, ownership inference remains an open problem, and crucial questions about the practical impact of ownership remain unanswered. We argue that a static program analysis that infers dominance is a crucial first step to ownership types inference. This paper describes an algorithm for statically computing dominance relations and shows that it can be used as part of an ownership inference algorithm.

1 Introduction

Dominance is at the heart of virtually every ownership discipline [3, 2, 5], and therefore one would expect dominance inference should be a key part of ownership inference. While there are many ownership disciplines, and there is little question about their benefits, practical adoption is lacking. This is due in part to the lack of software tools that support ownership such as automatic inference and refactoring tools incorporated in IDEs. Dominance inference is the foundation of ownership inference: an algorithm that statically computes dominance relations between objects, allows language designers to prototype ownership inference with respect to different ownership disciplines. Dominance inference has other applications as well. As it subsumes escape analysis, it can be used for lock elimination and deadlock detection [12]. Dominance inference can enable data-centric synchronization [18]. Additionally, dominance inference can be integrated into architecture extraction tools [8], and help enable reasoning about encapsulation properties.

The problem of dominance inference is defined in terms of the notion of *object graph*. Nodes in the graph are objects, and edges capture references between those objects. An edge links object i to object j if i has a field that refers to j, or a local variable in a method invoked on receiver i, refers to j. Fig. 1 shows a program and two object graphs: (1) shows the *concrete object graph* that summarizes the references between objects that arise as the program is evaluated, and (2) shows an *abstract object graph* which is a static approximation of the concrete graph obtained by program analysis. Static analysis entails a loss of precision. In this example, allocation site e is executed twice, resulting in objects e_1 and e_2. A typical static analysis abstraction scheme maps every concrete object to its allocation site, thus e_1 and e_2 map to the same abstract object e. Every object in the concrete object graph has a *dominance boundary*, defined as the maximal subgraph rooted at that object whose nodes are *dominated* by the object. The problem of dominance inference is stated as follows: given an

```
class Main {
  X x;
  Y y;
  static void main(String[] arg) {
     Main m = new Main();   [m]
     m.meth();
  }
  void meth() {
     this.x = new X();  [x]
     x.m();
     this.y = new Y();  [y]
     y.m();
  }
}
class X {
   Contain c;
   void m() {
     this.c = new Contain();  [c]
     c.put(0,1);
     Iter i = c.iter();
   }
}
```

```
class Y {
   Contain d;
   void m() {
     this.d = new Contain();  [d]
     d.put(1,1);
   }
}
class Contain {
   int[] e;
   Contain() {
      this.e = new int[10];  [e]
   }
   void put(int i, int j) { e[i] = j; }
   Iter iter() {
      Iter h = new Iter(e);  [f]
      return h;
} }
class Iter {
   int[] f;
   Iter(int[] f) { this.f = f; }
}
```

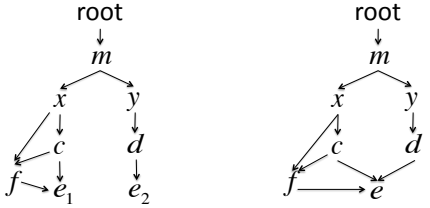

Fig. 1. Concrete (1) and abstract (2) object graphs for the simple program.

abstract object graph $\widehat{G}$ and an object i in $\widehat{G}$, find a subgraph with root i that safely approximates the dominance boundary of all concrete objects mapped to i. Dominance inference using *dynamic* analysis has been studied before [11, 14, 5, 19]. The appeal of dynamic inference lies in its simplicity. During program execution a concrete object graph is maintained by the implementation. However, like all dynamic approaches the results are unsound; there is no guarantee that inferred dominance won't be broken by an unseen execution path. Additionally, scalability and performance overheads limit the applicability of dynamic techniques. Surprisingly, static dominance inference has received almost no attention. Traditional dominator algorithms [6] cannot be applied on an abstract object graph as an abstract node corresponds to multiple nodes in the concrete object graph and straight-forward application of dominator algorithms breaks both precision and correctness. Consider Fig. 1. Clearly, d does not dominate e in the abstract graph, thus the results of the dominator algorithm cannot be used to conclude that concrete d dominates e_2. As we shall see, our dominance inference algorithm determines precisely that the abstract dominance boundary of d includes e, and therefore d does dominate e_2.

2 Formal Account of Object Graphs

We explain our algorithm and, later, ownership types, in terms of core Java-like calculus. Throughout the paper we will use the following notation for graphs. A graph G is a pair (N, E) where N is a set of nodes ranged over by variables i, j, k, l and E is a set of directed edges written $i \triangleright j$. We write $G \cup i \triangleright j$ to denote the addition of i and j to N and $i \triangleright j$ to E.We write $i \in G$ and $i \triangleright j \in G$ to test, respectively node and edge membership. For sets (of nodes, edges, etc.) we write $S \mathrel{+}= S'$ to denote adding S' to set S and $S \mathrel{-}= S'$ for removing S' from S.

2.1 Concrete Semantics

For brevity, we restrict our formal attention to a core calculus in the style of [18] whose syntax appears in Fig. 2. The language models Java with a syntax in A-normal form. Features not strictly necessary were omitted. The semantics operates over configurations of the form $S\ H\ G\ C$ where S is a stack, H is a heap, G is an object graph and C is a creation graph. A stack is a sequence of frames $\langle F\ \mathsf{s}\rangle$ consisting of a mapping F from variables to locations and a statement s. An object $o = \mathsf{C}(\bar{i}\,)$ consists of a class C and values $\bar{i}$ for the object fields. A heap is a mapping from indices, ranged over by meta-variables i, j, k, l, to objects. An object graph G summarizes the references between objects that occur at any time during program execution. A creation graph C records the creator of each object. We write $\bar{i}$ to denote a sequence of indices, $\overline{\tau\ \mathsf{z}}$ for a sequence of local variable declarations, etc. We write 0 to denote the null reference.

Fig. 3 shows the rules of the concrete semantics. Object creation (DNEW) instantiates a new object with all fields set to null and uses a fresh index j to refer to the newly allocated object. The rule adds an edge from i, the receiver of the current frame, to j the newly created object, to G. In addition, it records the edge from i, the creator of j, to j, in creation graph C. Writing to an object field (DWRITE) updates the heap. The value of field f of object $\mathsf{C}(\overline{j'}, j_\mathsf{f}, \overline{j''})$ is j_f. The rule also adds edge from k to j to G. Reading a field into a local variable (DREAD) has the expected semantics. The summary graph records the read by adding a reference from the receiver (i.e. this) to the value of the field. Invoking a method (DCALL) entails pushing a new frame on the stack with local variables initialized to null and formal parameters set to corresponding actual arguments. Function $mbody(\mathsf{C.m})$ retrieves the formal parameters, local variables and method body of the corresponding method. The summary graph records the edges from the receiver of the call (i.e., $F(\mathsf{y}) = k$) to all arguments.

cd	$::= \mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\}$	*class*			
fd	$::= \tau\ \mathsf{f}$	*field*	H	$::= [] \mid H[i \mapsto o]$	*heap*
md	$::= \tau\ \mathsf{m}(\overline{\tau\ \mathsf{x}})\{\overline{\tau\ \mathsf{z}}\ \mathsf{s};\ \mathsf{return\ y}\}$	*method*	S	$::= \epsilon \mid \langle F\ \mathsf{s}\rangle S$	*stack*
s	$::= \mathsf{s; s} \mid \mathsf{x} = \mathsf{new}\ \tau() \mid \mathsf{x} = \mathsf{y.f}$	*statement*	F	$::= [] \mid F[\mathsf{y} \mapsto i]$	*frame*
	$\mid \mathsf{x.f} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})$		o	$::= \mathsf{C}(\bar{i})$	*object*
τ	$::= \mathsf{C}$	*type*			

Fig. 2. Syntax.

$$\frac{o = \mathsf{C}(\overline{0}) \quad j \textit{ fresh} \quad F(\mathsf{this}) = i \quad G' = G \cup i \triangleright j \quad C' = C \cup i \triangleright j}{\langle F\ \mathsf{x} = \mathsf{new}\ \mathsf{C}(); \mathsf{s}\rangle S\ H\ G\ C \ \rightarrow\ \langle F[\mathsf{x} \mapsto j]\ \mathsf{s}\rangle S\ H[j \mapsto o]\ G'\ C'}\ (\textsc{dnew})$$

$$\frac{F(\mathsf{x}) = k \quad H(k) = \mathsf{C}(\overline{j'}, j_\mathsf{f}, \overline{j''}) \quad F(\mathsf{y}) = j \quad H' = H[k \mapsto \mathsf{C}(\overline{j'}, j, \overline{j''})] \quad G' = G \cup k \triangleright j}{\langle F\ \mathsf{x.f} = \mathsf{y}; \mathsf{s}\rangle S\ H\ G\ C \ \rightarrow\ \langle F\ \mathsf{s}\rangle S\ H'\ G'\ C}\ (\textsc{dwrite})$$

$$\frac{F(\mathsf{y}) = i \quad H(i) = \mathsf{C}(\overline{j'}, j_\mathsf{f}, \overline{j''}) \quad F(\mathsf{this}) = k \quad G' = G \cup k \triangleright j_\mathsf{f}}{\langle F\ \mathsf{x} = \mathsf{y.f}; \mathsf{s}\rangle S\ H\ G\ C \ \rightarrow\ \langle F[\mathsf{x} \mapsto j_\mathsf{f}]\ \mathsf{s}\rangle S\ H\ G'\ C}\ (\textsc{dread})$$

$$\frac{\begin{array}{c}F(\mathsf{y}) = k \quad F(\overline{\mathsf{z}}) = \overline{j} \quad H(k) = \mathsf{C}(\ldots) \quad \mathit{mbody}(\mathsf{C.m}) = \overline{\tau_\mathsf{x}\ \mathsf{x}'}; \overline{\tau_\mathsf{y}\ \mathsf{y}'}; \mathsf{s}'; \mathsf{return}\ \mathsf{y}'' \\ F' = [\overline{\mathsf{y}'} \mapsto \overline{0}][\overline{\mathsf{x}'} \mapsto \overline{j}][\mathsf{this} \mapsto k] \quad G' = G \cup \{k \triangleright j \mid j \in \overline{j}\}\end{array}}{\langle F\ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle S\ H\ G\ C \ \rightarrow\ \langle F'\ \mathsf{s}'; \mathsf{return}\ \mathsf{y}''\rangle\langle F\ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle S\ H\ G'\ C}\ (\textsc{dcall})$$

$$\frac{F(\mathsf{this}) = k \quad F'(\mathsf{y}) = j \quad G' = G \cup k \triangleright j}{\langle F'\ \mathsf{return}\ \mathsf{y}\rangle\langle F\ \mathsf{x} = \mathsf{y}'.\mathsf{m}(\overline{\mathsf{z}}); \mathsf{s}\rangle S\ H\ G\ C \ \rightarrow\ \langle F[\mathsf{x} \mapsto j]\ \mathsf{s}\rangle S\ H\ G'\ C}\ (\textsc{dret})$$

Fig. 3. Concrete semantics.

Lemma 1. *The object graph constructed by the above semantics is a superset of the object graph as defined by Clarke et al. [3].*

2.2 Abstract semantics

We assume a may points-to analysis [15, 7] that computes a safe approximation of the heap $\widehat{H}$ and stack $\widehat{S}$. The abstract semantics computes safe approximations of G and C, denoted $\widehat{G}$ and $\widehat{C}$ respectively. As $\widehat{H}$ and $\widehat{S}$ are conservative approximations, the semantics operates on *sets of abstract objects*. Thus, $\widehat{F}(\mathsf{x})$ evaluates to a set of abstract objects, not to a single object. Similarly, fields of an object in $\widehat{H}$ are sets of references (denoted I). We assume that all allocation sites are labelled with an unique identifier.

The abstraction function α is specific to our points-to analysis and is chosen so that $\alpha(i) = i'$ where i' is the index of the allocation site that created i. α acts on G in the obvious way: $\alpha(G) = (N, E)$ where $N = \{\alpha(i) \mid i \in G\}$ and $E = \{\alpha(i) \triangleright \alpha(j) \mid i \triangleright j \in G\}$. As the points-to analysis is safe, the following two conditions hold at every step. The first condition ensures the safety of variables,

$$\frac{\widehat{G}' = \widehat{G} \cup \{i \triangleright j \mid i \in \widehat{F}(\mathsf{this})\} \quad \widehat{C}' = \widehat{C} \cup \{i \triangleright j \mid i \in \widehat{F}(\mathsf{this})\}}{\langle \widehat{F}\ \mathsf{x} = \mathsf{new}^j\ \mathsf{C}(); \mathsf{s}\rangle \widehat{S}\ \widehat{H}\ \widehat{G}\ \widehat{C} \ \rightsquigarrow\ \langle \widehat{F}\ \mathsf{s}\rangle \widehat{S}\ \widehat{H}\ \widehat{G}'\ \widehat{C}'} \ (\textsc{anew})$$

$$\frac{\textit{if } \mathsf{x} \neq \mathsf{this} \textit{ then } \widehat{G}' = \widehat{G} \cup \{k \triangleright j \mid k \in \widehat{F}(\mathsf{x}) \textit{ and } j \in \widehat{F}(\mathsf{y})\} \textit{ else } \widehat{G}' = \widehat{G}}{\langle \widehat{F}\ \mathsf{x.f} = \mathsf{y}; \mathsf{s}\rangle \widehat{S}\ \widehat{H}\ \widehat{G}\ \widehat{C} \quad \rightsquigarrow \quad \langle \widehat{F}\ \mathsf{s}\rangle \widehat{S}\ \widehat{H}\ \widehat{G}'\ \widehat{C}} \ (\textsc{awrite})$$

$$\frac{\begin{array}{c}\textit{if } \mathsf{y} = \mathsf{this} \textit{ then } \widehat{G}' = \widehat{G} \textit{ else} \\ \widehat{G}' = \widehat{G} \cup \{k \triangleright j \mid k \in \widehat{F}(\mathsf{this}) \textit{ and } i \in \widehat{F}(\mathsf{y}) \textit{ and } \widehat{H}(i) = \mathsf{C}(\ldots I_{\mathsf{f}} \ldots) \textit{ and } j \in I_{\mathsf{f}}\}\end{array}}{\langle \widehat{F}\ \mathsf{x} = \mathsf{y.f}; \mathsf{s}\rangle \widehat{S}\ \widehat{H}\ \widehat{G}\ \widehat{C} \quad \rightsquigarrow \quad \langle \widehat{F}\ \mathsf{s}\rangle \widehat{S}\ \widehat{H}\ \widehat{G}'\ \widehat{C}} \ (\textsc{aread})$$

Fig. 4. Abstract Semantics. (Partial)

and the second ensures the safety of fields.

$$\begin{array}{rcl} F(\mathsf{x}) = i & \Rightarrow & \alpha(i) \in \widehat{F}(\mathsf{x}) \\ H(i) = \mathsf{C}(\ldots k_{\mathsf{f}} \ldots) & \Rightarrow & \widehat{H}(\alpha(i)) = \mathsf{C}(\ldots I_{\mathsf{f}} \ldots) \textit{ and } \alpha(k_{\mathsf{f}}) \in I_{\mathsf{f}} \end{array}$$

Fig. 4 shows the rules of the semantics that deal with object creation and field read/write. Rule (ANEW) adds new edges to $\widehat{G}$ and $\widehat{C}$ from every abstract receiver i of current frame $\widehat{F}$, to the abstract object j created at allocation site j. Rule (AWRITE) adds new edges to $\widehat{G}$ from every abstract object k in the points-to set of x to every j in the points-to set of y. The only interesting aspect of this rule is that the edges are added only when $\mathsf{x} \neq \mathsf{this}$. The intuition is that when x is this, the relevant edges are already in $\widehat{G}$ and there is no need to add them again.

Lemma 2. *$\widehat{G}$ and $\widehat{C}$ are safe. That is, $\alpha(G) \subseteq \widehat{G}$ and $\alpha(C) \subseteq \widehat{C}$ hold.*

3 Dominance Inference Analysis

The dominance inference analysis uses the abstract object and creation graphs as constructed by the above abstract semantics. It takes as input an abstract object i, and computes an abstract dominance boundary, which safely approximates the dominance boundaries of the concrete objects represented by i.

3.1 Flow Triples

Let us consider how object references can be transferred. Assume that i has a reference to j. We say that j flows to k from i if k acquires a reference to j from i. This can happen in one of the following four ways:

1. (DWRITE): Local variable y is assigned to a field of local variable x.
2. (DREAD): The field of local variable y is assigned to local variable x.

3. (DCALL): Local variable z is passed as argument to a method of y.
4. (DRET): The local variable y is returned to the receiver of the parent frame.

In each case, the operation adds an edge to the object graph as a side effect. Consider (DWRITE), x.f = y, and let $F(\mathsf{this}, \mathsf{x}, \mathsf{y}) = i, k, j$. Since y holds j, there has to be an edge $i \triangleright j$ in the object graph. Similarly, as x holds k, and there is an edge $i \triangleright k$ in the graph. After the operation, $k \triangleright j$ is added to the graph. We refer to this pattern as a *flow triple* and denote it $\langle i, k, j \rangle$. Consider Fig. 1. Expression i = c.iter() in method X.m causes the iterator object f to flow to x from c. Before the call, c holds c and h in Contain.iter holds f, and thus, $x \triangleright c$ and $c \triangleright f$. After the call, a new edge, $x \triangleright f$ is added to the graph. The pattern is reflected by flow triple $\langle x, c, f \rangle$.

The analysis records flow triples while processing the rules of the abstract semantics, namely (AWRITE), (AREAD), (ACALL) and (ARET). We set relation $isTriple(\langle i, j, k \rangle)$ to true whenever a flow triple is encountered. For example, for (AWRITE), we set $isTriple(\langle i, k, j \rangle)$ to true for every $i \in \widehat{F}(\mathsf{this})$, $k \in \widehat{F}(\mathsf{x})$ and $j \in \widehat{F}(\mathsf{y})$.

A flow triple captures transfer (i.e., exposure) of an object to another object, and is crucial to our analysis. Consider edge $d \triangleright e$ in the abstract graph in Fig. 1. There is no triple that includes this edge, which means that the concrete e referred by d, namely e_2, is not transferred, and therefore it is not exposed to any object but d; the analysis concludes that at runtime d dominates the concrete e it refers to. On the other hand, edge $c \triangleright e$ is part of triple $\langle c, f, e \rangle$ which captures that c's concrete e, e_1, is exposed to f (i.e., we have $f \triangleright e_1$). Edge $c \triangleright f$ is part of triple $\langle x, c, f \rangle$ and thus f is exposed to x (i.e., $x \triangleright f$). The analysis concludes that c does not dominate its run-time e, because said run-time e is exposed to f, and f in turn is exposed to x (i.e., there is a path $x \triangleright f \triangleright e_1$ that does not go through c).

3.2 Analysis Description

We begin with several definitions. The *root* of a graph, is a node j, such that there is a sequence of edges from j to any node i. We assume that G has root root. A *boundary* of a node i is a graph $B_i \subseteq G$ such that i is a root of B_i. A node j *dominates* node j' in boundary B_i if all paths from i to j' go through j. The *dominance boundary* of i in G is the maximal boundary B_i such that for all nodes $j \in B_i$, i *dominates* j in G. We denote the dominance boundary of i in G as D_i. $closure(G, i)$ computes the transitive closure of i inductively:

$$G'_0 = \{i \triangleright j \mid i \triangleright j \in G\} \quad ... \quad G'_n = G'_{n-1} \cup \{j \triangleright k \mid j \in G'_{n-1} \textit{ and } j \triangleright k \in G\}$$

The analysis uses $closure(G, i)$ on the abstract creation graph. $closure(\widehat{C}, i) = \widehat{C}'$ returns the creation dependences from i. We overload the notation slightly, and use $closure(\widehat{C}, i)$ to refer to the nodes in $\widehat{C}'$, that is, the objects created by i, directly or transitively.

The analysis uses a predicate *isOutside*:

$$isOutside(i, j) \quad = \quad \exists k \,.\, isTriple(\langle k, i, j \rangle)$$

Algorithm *computeBoundary*$(i, \widehat{G}, \widehat{C})$
output $\widehat{B}_i$

[1] $Out = \{j \mid isOutside(i, j)\}$
[2] $In = closure(\widehat{C}, i) - Out$
[3] $W = \{i \triangleright j \mid i \triangleright j \in \widehat{G}$ *and* $j \in Out\}$, $W^+ = W$
[4] while $W \neq \emptyset$
[5] $\quad W \mathrel{-}= k \triangleright j$
[6] $\quad$ if $j \in closure(\widehat{C}, i)$
[7] $\quad\quad In \mathrel{-}= closure(\widehat{C}, j)$
[8] $\quad\quad Out \mathrel{+}= closure(\widehat{C}, j)$
[9] $\quad\quad$ foreach $k' \in closure(\widehat{C}, j)$
[10] $\quad\quad\quad$ foreach $k'' \triangleright k' \in \widehat{C}$ *and* $k'' \in In$
[11] $\quad\quad\quad\quad$ if $k'' \triangleright k' \notin W^+$ then $W \mathrel{+}= k'' \triangleright k',\ W^+ \mathrel{+}= k'' \triangleright k'$
[12] $\quad$ foreach $k' \in \widehat{G}$ s.t. $isTriple(\langle k, j, k' \rangle)$
[13] $\quad\quad In \mathrel{-}= k',\ Out \mathrel{+}= k'$
[14] $\quad\quad$ if $k \triangleright k' \notin W^+$ then $W \mathrel{+}= k \triangleright k',\ W^+ \mathrel{+}= k \triangleright k'$
[15] $\quad$ foreach $k' \in \widehat{G}$ s.t. $isTriple(\langle k, k', j \rangle)$ *and* $k' \in In$
[16] $\quad\quad$ if $k' \triangleright j \notin W^+$ then $W \mathrel{+}= k' \triangleright j,\ W^+ \mathrel{+}= k' \triangleright j$
[17] $\quad$ foreach $k' \in \widehat{G}$ s.t. $isTriple(\langle k', k, j \rangle)$ *and* $k' \in In$
[18] $\quad\quad$ if $k' \triangleright j \notin W^+$ then $W \mathrel{+}= k' \triangleright j,\ W^+ \mathrel{+}= k' \triangleright j$
[19] $\widehat{B}_i = \{j \triangleright k \mid j \in In$ *and* $k \in In$ *and* $j \triangleright k \in \widehat{G}\}$

Fig. 5. computeBoundary returns $\widehat{B}_i$.

The predicate captures edges $i \triangleright j$ that are part of a triple $\langle k, i, j \rangle$. Such a triple indicates that there are paths from `root` to j through k that do not go through i, and therefore, i does not dominate j.

The analysis is presented in Fig. 5. It takes as input an abstract object i and uses $\widehat{G}$ and $\widehat{C}$. It computes $\widehat{B}_i$, a boundary of i in $\widehat{G}$. The analysis maintains sets of abstract objects *In* and *Out*. Set *In* contains the current overapproximation of the set of objects in every concrete dominance boundary. Set *Out* contains the current underapproximation of the set of objects in the frontier of the dominance boundary. The analysis starts with initial sets *In*, *Out* and tracks flow of objects using *isTriple*. Eventually, all potentially exposed objects are removed from *In*. The nodes remaining in *In* and the edges between them form boundary $\widehat{B}_i$. The correctness of the analysis is stated by the following theorem:

Theorem 1. *Let G be any object graph and i be any object in G. Let B'_i be any boundary of i in G. If $\alpha(B'_i) \subseteq \widehat{B}_{\alpha(i)}$ then $B'_i \subseteq D_i$.*

The theorem states that the computed boundary $\widehat{B}_{\alpha(i)}$ safely approximates the dominance boundary of every i. That is, for any concrete boundary B'_i represented by $\widehat{B}_{\alpha(i)}$, B'_i is included in D_i; thus i dominates in G all of the nodes in B'_i. In our running example, $\widehat{B}_d$ is the one-edge graph $d \triangleright e$. The theorem states that concrete edge $d \triangleright e_2$ is in the dominance boundary of d, or in other words, d dominates e_2.

4 Application: Ownership Type Inference

We present one application of the dominance inference analysis: ownership type inference. We choose the owner-as-dominator type system of [3] restricted to one ownership parameter. This restriction simplifies the problem; in future work we plan to investigate empirically the necessity for multiple ownership parameters, as well as extend the current analysis with handling of multiple parameters.

4.1 Type System

The type system of [3] assigns an ownership type $\langle p|p'\rangle$ to each local variable, field and allocation site. The type annotation $\mathsf{C}\langle p|p'\rangle\ \mathsf{x}$ (also written as $p\ \mathsf{C}\langle p'\rangle\ \mathsf{x}$), has the following interpretation: p is the owner of the object i referred to by x, and p' is an ownership parameter passed to that object. p takes one of the following three values: rep, own or p (for brevity, we rename owner to own and omit discussion of norep [3]). rep denotes that object i is owned by this, own denotes that i is owned by the owner of this, and p denotes that i's owner is passed from this as an ownership parameter. p' takes the same values. rep is the most precise value, followed by own, followed by p, or in other words, we have rep $<$ own $<$ p. For this paper we impose the following restriction on ownership types $\langle p|p'\rangle : p \leq p'$. Even though types where $p > p'$ (e.g., $\langle \mathsf{p}|\mathsf{rep}\rangle$) are allowed in ownership types, the properties of the system entail that if the program type checks with $\langle p|p'\rangle$, where $p > p'$, it will type check with $\langle p'|p'\rangle$

$$\frac{E(\mathsf{x}) = \mathsf{C}\ t}{E \vdash \mathsf{x} = \mathsf{new}\ \mathsf{C}\ t}\ (\textsc{tnew})$$

$$\frac{\mathsf{x} \neq \mathsf{this} \quad E(\mathsf{x}) = \mathsf{C}\ t_x \quad \mathit{typeof}(\mathsf{C.f}) = \mathsf{D}\ t_f \quad E(\mathsf{y}) = \mathsf{D}\ t_y \quad \mathit{adapt}(t_f, t_x) = t_y}{E \vdash \mathsf{x.f} = \mathsf{y}}\ (\textsc{twrite})$$

$$\frac{E(\mathsf{this}) = \mathsf{C}\ t' \quad \mathit{typeof}(\mathsf{C.f}) = \mathsf{D}\ t \quad E(\mathsf{y}) = \mathsf{D}\ t}{E \vdash \mathsf{this.f} = \mathsf{y}}\ (\textsc{twritethis})$$

$$\frac{\mathsf{y} \neq \mathsf{this} \quad E(\mathsf{y}) = \mathsf{C}\ t_y \quad \mathit{typeof}(\mathsf{C.f}) = \mathsf{D}\ t_f \quad E(\mathsf{x}) = \mathsf{D}\ t_x \quad \mathit{adapt}(t_f, t_y) = t_x}{E \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{E(\mathsf{this}) = \mathsf{C}\ t' \quad \mathit{typeof}(\mathsf{C.f}) = \mathsf{D}\ t \quad E(\mathsf{x}) = \mathsf{D}\ t}{E \vdash \mathsf{x} = \mathsf{this.f}}\ (\textsc{treadthis})$$

$$\frac{E(\mathsf{y}) = \mathsf{C}\ t_y \quad \mathit{typeof}(\mathsf{C.m}) = \overline{\mathsf{D}\ t} \to \mathsf{D}'\ t' \quad \mathsf{y} \neq \mathsf{this} \quad E(\mathsf{x}) = \mathsf{D}'\ t_x \quad E(\overline{\mathsf{z}}) = \overline{\mathsf{D}\ t_z} \quad \mathit{adapt}(\overline{t}, t_y) = \overline{t_z} \quad \mathit{adapt}(t', t_y) = t_x}{E \vdash \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})}\ (\textsc{tcall})$$

$$\frac{E(\mathsf{this}) = \mathsf{C}\ t'' \quad \mathit{typeof}(\mathsf{C.m}) = \overline{\mathsf{D}\ t} \to \mathsf{D}'\ t' \quad E(\mathsf{x}) = \mathsf{D}'\ t' \quad E(\overline{\mathsf{z}}) = \overline{\mathsf{D}\ t}}{E \vdash \mathsf{x} = \mathsf{this.m}(\overline{\mathsf{z}})}\ (\textsc{tcallthis})$$

Fig. 6. Type rules.

as well. Our analysis naturally restricts the inferred types to the following six ownership types, ordered in order of decreasing precision:

$$\langle \mathsf{rep}|\mathsf{rep}\rangle < \langle \mathsf{rep}|\mathsf{own}\rangle < \langle \mathsf{rep}|\mathsf{p}\rangle < \langle \mathsf{own}|\mathsf{own}\rangle < \langle \mathsf{own}|\mathsf{p}\rangle < \langle \mathsf{p}|\mathsf{p}\rangle$$

Note that the above is an ordering relation over the set of types, not a subtyping relation. The ordering relation is necessary to define an inference algorithm based on fixpoint iteration.

The rules for the ownership type system are given in Fig. 6 (see [3] for additional details). The system assigns types $\mathsf{C}\ t$, where C is the class type and t is the ownership type. For brevity, features not strictly necessary are omitted. The viewpoint adaptation function $adapt(t, t')$, gives the view of ownership type t from ownership type t':

$$\begin{aligned} adapt(\langle \mathsf{own}|\mathsf{own}\rangle, \langle p|p'\rangle) &= \langle p|p\rangle \\ adapt(\langle \mathsf{own}|\mathsf{p}\rangle, \langle p|p'\rangle) &= \langle p|p'\rangle \\ adapt(\langle \mathsf{p}|\mathsf{p}\rangle, \langle p|p'\rangle) &= \langle p'|p'\rangle \end{aligned}$$

Viewpoint adaptation originates from work on Universe types [4]. As it is explained in [4], the intuition behind *adapt* is the folowing: if object i sees object j as having ownership type t', and j sees k as having ownership type t, then i sees k as having ownership type t'' where $t'' = adapt(t, t')$ (i.e., t'' is the adapted t from the point of view of t'). In [3] viewpoint adaptation is accomplished through substitution function σ and its inverse ψ; we believe that *adapt* is more intuitive and have taken the liberty to use *adapt*.

adapt is partially defined: no t that contains rep can be viewed from another type t', which accounts for static visibility.

4.2 Type Inference

Fig. 7 shows the ownership type annotations for our example program as inferred by our analysis. The iterator object at allocation site f receives type $\langle \mathsf{own}|\mathsf{own}\rangle$ (written in the code as $\mathsf{own}\ \mathsf{Iter}\langle \mathsf{own}\rangle$). The owner is own which means that the container's owner, x for container c and y for container d, is the owner of the iterator. The ownership parameter passed to the iterator is own as well, but it remains unused, as the analysis infers that the iterator's owner, x or y, owns the corresponding array, e_1 or e_2 respectively. Our prototype reports types for allocation sites and fields. It infers types for local variables as they appear in the intermediate representation but does not map these to Java variables. This is an engineering issue that we plan to address. We stay faithful to the output of our current prototype and show the types it infers.

We infer an ownership type on every edge of $\widehat{G}$. Subsequently, we join these types to compute types for local variables, fields and allocation sites, and show that the computed types type check in the above type system. Each edge $i \triangleright j \in \widehat{G}$ receives an ownership type $T(i \triangleright j) = \langle p|p'\rangle$. p is j's owner from the point of view of i: if p is rep, then i is the owner of j; otherwise, if p is own, then the owner of i is also the owner of j, and finally, if p_0 is p, then i's ownership parameter is the

```
class Main {
  rep X<rep> x; rep Y<rep> y;
  static void main(String[] arg) {
     Main m = new Main();   [m]
     m.meth();
  }
  void meth() {
     x = new rep X<rep>();  [x]
     x.m();
     y = new rep Y<rep>();  [y]
     y.m();
  }
}
class X<p> {
   rep Contain<rep> c;
   void m() {
     this.c = new rep Contain<rep>();  [c]
     c.put(0,1);
     Iter i = c.iter();
   }
}
class Y<p> {
   rep Contain<rep> d;
   void m() {
     this.d = new rep Contain<rep>();  [d]
     d.put(1,1);
   }
}
class Contain<p> {
   own int[] e;
   Contain() { this.e = new own int[10]; }  [e]
   void put(int i, int j) { e[i] = j; }
   Iter iter() {
      Iter h = new own Iter<own>(e);  [f]
      return h;
   }
}
class Iter<p> {
   own int[] f;
   Iter(int[] f) { this.f = f; }
}
```

Fig. 7. Ownership types for simple program.

owner of j. Analogously, p' is j's ownership parameter from the point of view of i. The problem at hand is a constraint problem. We seek type assignment T on the edges of $\widehat{G}$ such that every flow triple $\langle i, j, k\rangle$ in $\widehat{G}$ is well-typed:

$$isTriple(\langle i, j, k\rangle) \Rightarrow adapt(T(j \triangleright k), T(i \triangleright j)) = T(i \triangleright k)$$

These constraints capture the type constraints in Sec. 4.1. Consider rule (TWRITE) which types $\mathsf{x.f = y}$. In the object graph we have flow triple $\langle i, k, j\rangle$ where this holds i, x holds k and y holds j. t_x is the type of x from the point of view of this, and $T(i \triangleright k)$ is the type of k (x) from the point of view of i (this); t_f is the type of field f from the point of view of an object of class C, and $T(k \triangleright j)$ is the type of j from the point of view of k; finally, t_y is the type of y from the point of view of this and $T(i \triangleright j)$ is the type of j from the point of view of i. Our analysis makes $t_x = T(i \triangleright k)$, $t_f = T(k \triangleright j)$ and $t_y = T(i \triangleright j)$. The well-typedness of flow triple $\langle i, k, j\rangle$ (i.e., $adapt(T(k \triangleright j), T(i \triangleright k)) = T(i \triangleright j)$) guarantees the well-typedness of x.f=y (i.e, $adapt(t_f, t_x) = t_y$).

Clearly, there are many assignments that satisfy the *adapt* constraints. For example, a trivial assignment would assign $\langle \mathsf{p}|\mathsf{p}\rangle$ to all edges in $\widehat{G}$ except edges $\mathtt{root} \triangleright i$, to which it would assign $\langle \mathsf{rep}|\mathsf{rep}\rangle$. This assignment is bad however, as it produces a flat (and useless) ownership tree where root is the owner of all objects. A good assignment would assign a large number of rep types.

A *triple typing* is a triple of types $\langle t_{ij}, t_{jk}, t_{ik}\rangle$. A *well-typed triple typing* is a triple typing that meets the *adapt* constraint: $adapt(t_{jk}, t_{ij}) = t_{ik}$. There are

18 well-typed triple typings: t_{jk} ranges over $\langle \mathsf{own}|\mathsf{own}\rangle$, $\langle \mathsf{own}|\mathsf{p}\rangle$ and $\langle \mathsf{p}|\mathsf{p}\rangle$, and t_{ij} ranges over $\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle$ (recall that *adapt* restricts the values of its first argument to account for static visibility). We define an ordering over the set of well-typed triple typings:

$$
\begin{array}{ll}
\langle\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{rep}\rangle\rangle & < \langle\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{rep}\rangle\rangle \quad < \\
\langle\langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{rep}\rangle\rangle & < \langle\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{rep}|\mathsf{rep}\rangle\rangle \quad < \\
\langle\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle\rangle & < \langle\langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{rep}|\mathsf{p}\rangle\rangle \quad < \\
\langle\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{rep}|\mathsf{rep}\rangle\rangle & < \langle\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle\rangle \quad < \\
\langle\langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\rangle & < \langle\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle\rangle \quad < \\
\langle\langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle\rangle & < \langle\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle\rangle \quad < \\
\langle\langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\rangle & < \langle\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle\rangle \quad < \\
\langle\langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\rangle & < \langle\langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\rangle \quad < \\
\langle\langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\rangle & < \langle\langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\rangle
\end{array}
$$

Triple typings with two rep owners are most precise, followed by triple typings with one rep owner, followed by triple typings with three own owners, etc. The least precise typing is the one where all three edges have type $\langle \mathsf{p}|\mathsf{p}\rangle$. Function *raiseTriple* takes a flow triple $\langle i, j, k\rangle$ as an argument and returns the smallest (i.e., most precise) typing $\langle t_{ij}, t_{jk}, t_{ik}\rangle$ in the above ordering, such that $T(i \triangleright j) \leq t_{ij}$ and $T(j \triangleright k) \leq t_{jk}$ and $T(i \triangleright k) \leq t_{ik}$. Intuitively, when the analysis encounters a flow triple $\langle i, j, k\rangle$, which is not well-typed, it invokes *raiseTriple* to find the most precise well-typed typing that is larger than the typing on $\langle i, j, k\rangle$. It then raises the types on $\langle i, j, k\rangle$ to $\langle t_{ij}, t_{jk}, t_{ik}\rangle$ to make $\langle i, j, k\rangle$ well-typed. Function *adjTriples* takes an edge $i \triangleright j$ as an argument and returns the set of all flow triples adjacent to this edge: $\{\langle i, j, k\rangle\} \cup \{\langle i, k', j\rangle\} \cup \{\langle k'', i, j\rangle\}$. If an edge changes its type, the change affects all adjacent triples. The analysis is shown in Fig. 8. It uses the dominance analysis from Sec. 3. Procedure *assignEdgeTypes* assigns an initial type to every edge in $\widehat{G}$ as follows: if the edge is in the dominance boundary of its source, then its initial type is $\langle \mathsf{rep}|\mathsf{rep}\rangle$; otherwise, its type is $\langle \mathsf{own}|\mathsf{own}\rangle$ (lines 1-5). Unfortunately, not all flow triples will be well-typed under this initial assignment. The analysis collects the triples that are not well-typed (lines 6-8), and invokes *resolve* (line 9), which repeatedly raises types until it reaches a fixpoint. Procedure *assignTypes* assigns types on locals and fields. For each variable x, it joins the types of the edges in $\widehat{G}$ that correspond to x (line 8); notation $\bigvee$ has the standard lattice-theoretic interpretation as the join of all values — $E(x)$ is assigned the largest $T(i \triangleright j), i \triangleright j \in M$, according to the ordering of ownership types from Sec. 4.1. If one of the edges in M has a type smaller than $E(\mathsf{x})$, the analysis raises its type to $E(\mathsf{x})$ and places its adjacent triples on the conflict list (lines 5-8). The procedure repeats for fields.

Theorem 2. *Let E be the type assignment for program P computed by the analysis. P is well-typed in the system from Sec. 4.1.*

Discussion. The above analysis, a fixpoint iteration, can be applied to *any* initial type assignment. An optimistic initial assignment would assign a large number of rep types, and a pessimistic assignment would assign less rep types

procedure *assignEdgeTypes*($\widehat{G}$)
output T
[1] foreach $i \triangleright j \in \widehat{G}$
[2] if $i \triangleright j \in \widehat{B}_i$
[3] $T(i \triangleright j) = \langle \mathsf{rep}|\mathsf{rep} \rangle$
[4] else
[5] $T(i \triangleright j) = \langle \mathsf{own}|\mathsf{own} \rangle$
[6] foreach $\langle i,j,k \rangle$ s.t. *isTriple*($\langle i,j,k \rangle$)
[7] if *adapt*($T(j \triangleright k), T(i \triangleright j)) \neq T(i \triangleright k)$
[8] $K \mathrel{+}= \langle i,j,k \rangle$
[9] $T = resolve(\widehat{G}, K, T)$
[10]return T

procedure *resolve*($\widehat{G}, K, T$)
output T
[1] $W = K$
[2] while $W \neq \emptyset$
[3] $W \mathrel{-}= \langle i,j,k \rangle$
[4] $\langle t_{ij}, t_{jk}, t_{ik} \rangle = raiseTriple(\langle i,j,k \rangle)$
[5] if $t_{ij} \neq T(i \triangleright j)$
[6] $T(i \triangleright j) = t_{ij}$
[7] $W \mathrel{+}= adjTriples(i \triangleright j)$
[8] if $t_{jk} \neq T(j \triangleright k)$
[9] $T(j \triangleright k) = t_{jk}$
[10] $W \mathrel{+}= adjTriples(j \triangleright k)$
[11] if $t_{ik} \neq T(i \triangleright k)$
[12] $T(i \triangleright k) = t_{ik}$
[13] $W \mathrel{+}= adjTriples(i \triangleright k)$
[14]return T

procedure *assignTypes*($\widehat{G}, \widehat{H}, \widehat{S}, T$)
output well-typed E, T
[1]*change* = *true*
[2] while *change*
[3] *change* = *false*, $K = \emptyset$
[4] foreach class $\mathsf{C} \in$ program P
[5] foreach method $\mathsf{m} \in \mathsf{C}$
[6] foreach variable $\mathsf{x} \in \mathsf{m}$
[7] $M = \{i \triangleright j \mid i \triangleright j \in \widehat{G}$ *and* $i \in \widehat{F}(\mathsf{this}_\mathsf{m})$ *and* $j \in \widehat{F}(\mathsf{x})\}$
[8] $E(\mathsf{x}) = \bigvee_{i \triangleright j \in M} T(i \triangleright j)$
[9] foreach $i \triangleright j \in M$
[10] if $E(\mathsf{x}) \neq T(i \triangleright j)$
[11] $T(i \triangleright j) = E(\mathsf{x})$
[12] *change* = *true*
[13] $K \mathrel{+}= adjTriples(i \triangleright j)$
[14] foreach field $\mathsf{f} \in \mathsf{C}$
[15] $M = \{i \triangleright j \mid i \triangleright j \in \widehat{G}$ *and* $\widehat{H}(i) = \mathsf{C}(...I_\mathsf{f}...)$ *and* $j \in I_\mathsf{f}\}$
[16] $E(\mathsf{C.f}) = \bigvee_{i \triangleright j \in M} T(i \triangleright j)$
[17] foreach $i \triangleright j \in M$
[18] if $E(\mathsf{C.f}) \neq T(i \triangleright j)$
[19] $T(i \triangleright j) = E(\mathsf{C.f})$
[20] *change* = *true*
[21] $K \mathrel{+}= adjTriples(i \triangleright j)$
[22] $T = resolve(\widehat{G}, K, T)$
[23] return E, T

Fig. 8. Type assignment.

and more own and p types. An unwise initial assignment would affect scalability, precision or both. If the assignment is overly optimistic, the majority of edges would need to be lowered from rep (since most edges are not rep anyway), and this would likely prohibit scaling the analysis beyond small programs. On the other hand, if the assignment is overly pessimistic, the analysis will converge faster to a fixpoint, but will lose precision. We conjecture that our initial assignment, which makes use of dominance inference, is key to the scalability and precision of ownership type inference. It would immediately filter out edges that cannot be rep; as a result, very few edges would change type (predominantly from $\langle \mathsf{own}|\mathsf{own} \rangle$ to $\langle \mathsf{p}|\mathsf{p} \rangle$), and the analysis would scale well. Also, few edges that can be rep, would not be assigned rep in the initial assignment.

5 Implementation

The object graph analysis, dominance inference analysis and type inference analysis are implemented in Java using Soot 2.2.3 [17] and the Andersen-style points-to analysis provided by Spark [7]. We performed whole-program analysis with

Program	Description	Size		Create		Fields		Time	
		#Class	#Meth	#Create	dom	#Field	dom	Pt	Dom
gzip	GZIP IO streams	6	3819	35	31	7	4	25s	2s
zip	ZIP IO streams	6	3844	29	21	10	5	25s	3s
checked	streams/checksums	4	3766	9	8	2	0	96s	2s
collator	text collation	15	3868	40	31	17	9	25s	3s
breaks	iter. over text	13	3822	270	268	7	0	26s	3s
number	number formatting	10	3880	124	119	3	1	25s	4s
jdepend	Quality metrics	17	3962	84	66	29	19	26s	3s
javad	Decompiler	41	3838	48	37	36	19	26s	2s
JATLite	Agent system	45	6279	273	117	142	35	42s	20s
undo	Undo functionality	237	5644	728	313	290	56	50s	31s
soot	Analysis framework	579	6046	703	274	283	64	40s	179s
sablecc	Parser generator	300	7970	1261	865	284	25	49s	34s
polyglot	Compiler	267	7449	1180	278	431	52	141s	365s
antlr	Parser generator	126	5102	596	434	152	38	39s	13s
bloat	Bytecode optimizer	289	6402	1047	453	449	79	41s	95s
jython	Python interpreter	163	5606	520	143	206	41	38s	122s
pmd	Source analyzer	718	8653	374	163	114	46	67s	105s
ps	Postscript engine	200	5396	424	113	19	7	38s	136s

Table 1. Information about benchmarks and dominance inference results.

Create						Fields					
rep|rep	rep|own	rep|p	own|own	own|p	p|p	rep|rep	rep|own	rep|p	own|own	own|p	p|p
6	5	24	1	10	1	2	5	11	1	6	11

Table 2. Type inference results for benchmark javad.

the Sun JDK 1.4.1 libraries. All experiments were done on a MacBook Pro with 4GB of RAM. The implementation, which includes Soot and Spark, was run with a max heap size of 1400MB; however, all benchmarks ran within a memory footprint of 800MB. Native methods are handled by utilizing the models provided by Soot. Reflection is handled by specifying the dynamically loaded classes which Spark uses to appropriately resolve reflection calls.

Our benchmark suite is presented in Table 1. It includes 6 software components (from gzip through number) which we have used in previous work and are familiar with. Each component is transformed into a whole program by attaching an artificial main method to complete it which allows whole-program analysis [16]. In addition, the suite includes 12 whole programs: jdepend, javad, JATLite and undo, benchmarks soot and sablecc from the Ashes suite, polyglot, and antlr, bloat, jython, pmd and ps from the DaCapo benchmark suite version beta051009. **#Class** gives the size of the benchmarks in classes; **#Meth** gives the size of the benchmarks in methods (user and library) reachable by Spark.

5.1 Results

We report dominance inference results on allocation sites and instance fields of reference type. Multicolumn **Create** in Table 1 shows the number of object

creation sites in user classes, excluding String and StringBuffer. Column dom shows the number inferred as dominated by their creating object. Multicolumn **Fields** shows analogous information for instance fields of reference type in user classes, again excluding fields of type String and StringBuffer.

On average, for the 12 large benchmarks, roughly 50% of all creation sites and 30% of all fields were reported as dom. This suggests that *ownership occurs frequently* in real-world object-oriented programs. The high percentage of dom creation sites is not surprising because programs typically create a large number of temporary objects that remain method-local (roughly 30% according to one study [15]). Our analysis captures method-local objects, as well as "object-local" objects (i.e., objects assigned to fields, but remaining in the boundary of their creating "owner" object). These results suggest that the dominance analysis will fare well in another application: escape analysis. Column **Pt** shows the running time for Spark's points-to analysis, **Dom** shows the running time for dominance inference. Except for polyglot, an outlier for all analyses, inference scales well, completing in under 200 seconds.

Additionally, Table 2 shows type inference results for benchmark javad. javad, 4000LOC, was annotated manually and type-checked by a checker built on top of the Checkers framework [13]. Table 2 lists 47 creation sites instead of 48 because one site was static and annotated as norep (see [3]). Interestingly, the additional constraints that ownership types impose on dominance, do not cause dom annotations to become own or p. All but one creation site, and all but one field inferred as dom by dominance inference, stay rep after type inference. We do not report inference results on the other programs, because we have not type checked those programs; we are in the process of integrating the inference analysis with the type checker, which will enable automatic inference and checking.

5.2 Precision

Addressing the issue of precision is highly non-trivial. To the best of our knowledge, there are no large programs annotated with ownership types, that could be used to objectively evaluate an ownership inference analysis. In order to evaluate the precision of our analysis, we performed a study of *absolute precision* [16, 8] on a subset of the fields. Specifically, we considered all fields in components gzip through number and all fields in javad. This accounted for 82 fields. Of these, 38 were reported as dom and 44 were reported as not dom.

To evaluate the precision of the dominance inference, we examined every field f that was not reported as dom, and attempted to prove exposure. That is, we attempted to show that there is an execution such that an object j stored in field f of object i is exposed outside of i, or more formally, that i does not dominate j in the concrete object graph. In *every case*, we were able to prove such exposure. In addition, we examined every dom field. Although the analysis is proven safe and therefore, a dom field must be indeed dominated by its enclosing object, we conducted the detailed examination in order to gain further confidence in the functional correctness of the implementation. In *every case*, the dom field was indeed dominated as expected. Therefore, for this set of 82 fields, the inference analysis achieved very good precision.

6 Related Work

Despite significant effort on ownership types, ownership inference has received much less attention. Work on dynamic ownership inference includes [14, 11, 5, 19]. In their essence, these works take the same approach. They reason about dominance (and hence ownership) on dynamic (i.e., concrete) object graphs by applying well-known dominator algorithms [6] on those graphs. They face challenges such as large concrete object graphs [14, 11] and runtime overhead [19], and they are inherently unsafe since inferred dominance (i.e., ownership) holds only on observed runs. Our dominance inference is fundamentally different: it performs deep semantic analysis on the abstract object graph and avoids the problems inherent in dynamic analysis. The empirical investigation suggests that it avoids the usual pitfall of static analysis (i.e., imprecision), and presents a "sweet spot" in the spectrum: an inexpensive but precise analysis. Ma and Foster present Uno [9], a static analysis-based tool for inference of encapsulation properties in Java programs. Among other things, their analysis computes what fields are *owned*. They report 16% of the fields across their benchmarks as owned, while we report (roughly) 30% as owned. The difference can be explained by the difference in the inferred ownership. Uno infers exclusive ownership: that is, an owned object must be accessed only by its owner. Our model is less-restrictive: an owned object can be passed to other owned objects. We inferred exclusive ownership in our framework and we found that 20% of all fields were exclusively owned. This result is close to Uno's 16%. It suggests that objects often flow to other objects, while remaining encapsulated in their owner and therefore, exclusive ownership may not be enough. We observed multiple such cases in our case studies. Aldrich et al. [1] present an ownership type system which includes annotations for uniqueness, ownership, sharing and parameters. They present a type inference analysis and report preliminary results on small programs and Java library classes. Their inference algorithm is conceptually different from ours; it creates several kinds of constraints at the level of the source, namely equality constraints, component constraints and instantiation constraints; subsequently, it uses a worklist-based procedure to resolve the constraints. Our analysis solves one kind of constraints, essentially equality constraints defined with *adapt*; it relies on dominance inference to start at a "good point" in the solution space, which, we conjecture, speeds-up the resolution procedure. It is difficult to judge which analysis is better because the analysis from [1] is never fully described; [1] focuses on the type system and experience with type checking, not on type inference as our work does. Aldrich has pointed out that reasoning about multiple ownership parameters presents significant difficulty. In this sense, we solve a simpler problem, as for this paper we focus on a system with one ownership parameter; we plan to address multiple ownership parameters in future work. Finally, we contrast this work with our own previous work [8] and [10]. This paper presents a substantial extension in that it computes abstract ownership *boundaries*, while [8] reasoned about specific edges in the object graph. The work in [10] presents a preliminary version of the dominance inference analysis.

7 Conclusion

We have presented a novel static dominance inference analysis. One direction of future work is to build a framework for ownership inference. Different inference analyses, each addressing a specific ownership discipline, can be coded easily on top of dominance inference.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.
2. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.
3. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
4. D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Muller, and A. Summers. Universe types for topology and encapsulation. In *FMCO*, 2008.
5. W. Dietl and P. Müller. Runtime Universe type inference. In *IWACO*, 2007.
6. T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1(1):121–141, May 1979.
7. O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.
8. Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.
9. K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.
10. A. Milanova. Static inference of Universe types. In *IWACO*, 2008.
11. N. Mitchell. The runtime structure of object ownership. In *ECOOP*, pages 74–98, 2006.
12. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.
13. M. Papi, M. Ali, T. Correa Jr., J. Perkins, and M. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 261–272, 2008.
14. A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency - Practice and Experience*, 16(7):671–687, 2004.
15. A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.
16. 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.
17. 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.
18. M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.
19. M. Vetchev, E. Yahav, and G. Yorsh. Phalanx: Parallel checking of expressive heap assertions. In *ISMM*, pages 41–50, 2010.