# Towards Effective Inference and Checking of Ownership Types

Wei Huang and Ana Milanova

Rensselaer Polytechnic Institute, Troy NY, USA

**Abstract.** We present a checker for the owner-as-dominator type system. We add a flow analysis which performs type inference. Our checker allows programmers to annotate a subset of the variables, fields and/or allocation sites; the flow analysis fills in the remaining annotations, and the type checker checks the program. We have type checked two relatively large programs, javad and SPECjbb and present inference and checking results.

## 1 Introduction

Object-oriented languages such as Java only provide name-based mechanisms for protecting an object's internal representations. In practice, however, this mechanism is not strong enough. For example, the method Class.getSigners() in Java 1.1 returns a reference to an internal array. Although the array is protected by the private keyword, clients can still modify it, leading to security problems.

On the other hand, ownership is a control-based protection mechanism. It enforces that only the owner can have full control of (known as *owner-as-dominator*) or modify (known as *owner-as-modifier*) its owned object. There are many ownership type systems in the literature; however, practical adoption is lacking. One reason is that these ownership type systems have a disadvantage — the annotation requirement. Programmers have to put significant effort into annotating new or existing software systems in order to realize the benefits of ownership.

In previous work we have addressed the problem of ownership inference [8, 10]. However, inferred ownership is not checked by a type checker, and it is difficult to verify the correctness of the inferred annotations. In this work, we build a type checker on top of the Checker Framework [12], for the classical owner-as-dominator type system from [3] restricted to one ownership parameter. We go beyond a simple type checker and add a flow analysis which performs type inference. Our checker allows programmers to annotate any subset of fields, locals or allocation sites; the flow analysis fills in the remaining annotations, and the type checker checks the program.

## 2 Type System

The type annotation $\langle q_0|q_1\rangle$ of our type system consists of two parts: $q_0$ is the owner of the object and $q_1$ is the ownership parameter passed to that object.

An annotated field or local variable is written as $\langle q_0|q_1\rangle$ $\mathsf{C}$ $\mathsf{x}$ and allocation site $\mathsf{new}$ $\langle q_0|q_1\rangle$ $\mathsf{C}()$. $q_0$ and $q_1$ can be one of the following values: rep, own, and $p$. rep denotes the object is owned by this and belongs to this's representation; own denotes the object is owned by the owner of this (note that for brevity we use own instead of owner as in [3]); and $p$ denotes the object's owner is the ownership parameter of this.

$$\text{(TNEW)}\ \frac{T(\mathsf{x}) = t\ \mathsf{C}}{T \vdash \mathsf{x} = \mathsf{new}\ t\ \mathsf{C}} \qquad \text{(TASSIGN)}\ \frac{T(\mathsf{x}) = t\ \mathsf{C} \quad T(\mathsf{y}) = t\ \mathsf{C}}{T \vdash \mathsf{x} = \mathsf{y}}$$

$$\text{(TWRITE)}\ \frac{\mathsf{x} \neq \mathsf{this} \quad T(\mathsf{x}) = t_x\ \mathsf{C} \quad \mathit{typeof}(\mathsf{C.f}) = t_f\ \mathsf{D} \quad T(\mathsf{y}) = t_y\ \mathsf{D} \quad \mathit{adapt}(t_f, t_x) = t_y}{T \vdash \mathsf{x.f} = \mathsf{y}} \qquad \text{(TWRITETHIS)}\ \frac{T(\mathsf{this}) = t'\ \mathsf{C} \quad \mathit{typeof}(\mathsf{C.f}) = t\ \mathsf{D} \quad T(\mathsf{y}) = t\ \mathsf{D}}{T \vdash \mathsf{this.f} = \mathsf{y}}$$

$$\text{(TREAD)}\ \frac{\mathsf{y} \neq \mathsf{this} \quad T(\mathsf{y}) = t_y\ \mathsf{C} \quad \mathit{typeof}(\mathsf{C.f}) = t_f\ \mathsf{D} \quad T(\mathsf{x}) = t_x\ \mathsf{D} \quad \mathit{adapt}(t_f, t_y) = t_x}{T \vdash \mathsf{x} = \mathsf{y.f}} \qquad \text{(TREADTHIS)}\ \frac{T(\mathsf{this}) = t'\ \mathsf{C} \quad \mathit{typeof}(\mathsf{C.f}) = t\ \mathsf{D} \quad T(\mathsf{x}) = t\ \mathsf{D}}{T \vdash \mathsf{x} = \mathsf{this.f}}$$

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

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

**Fig. 1.** Typing Rules

Fig. 1 shows the typing rules (see [10] for additional details). $T$ is a type mapping from variables to the annotated types; $t$ $\mathsf{C}$ is an annotated type, in which $t$ can be $\langle \mathsf{rep}|\mathsf{rep}\rangle$, $\langle \mathsf{rep}|\mathsf{own}\rangle$, $\langle \mathsf{rep}|p\rangle$, $\langle \mathsf{own}|\mathsf{own}\rangle$, $\langle \mathsf{own}|p\rangle$ or $\langle p|p\rangle$; *typeof* is a primitive that given a method or field name returns respectively, the method or field type. To better illustrate inference and checking, we ignore features such as static methods and static fields; they will be discussed in Section 5.

Viewpoint adaptation, exemplified in work on Universe types [5], is used to adapt ownership type $t_{in}$ from the point of view of ownership type $t_{rcv}$. $\mathit{adapt}(t_{in}, t_{rcv})$ returns the type $t_{out}$. Here, $t_{in}$ denotes the annotated type of a variable inside its enclosing object, $t_{rcv}$ denotes the receiver whose fields/parameters are being adapted, and $t_{out}$ denotes the adapted ownership type from outside of the receiver. *adapt* is defined below:

$$\begin{aligned}
\mathit{adapt}(\langle \mathsf{own}|\mathsf{own}\rangle, \langle q_0|q_1\rangle) &= \langle q_0|q_0\rangle \\
\mathit{adapt}(\langle \mathsf{own}|p\rangle, \langle q_0|q_1\rangle) &= \langle q_0|q_1\rangle \\
\mathit{adapt}(\langle p|p\rangle, \langle q_0|q_1\rangle) &= \langle q_1|q_1\rangle
\end{aligned}$$

```
class Link<p> {
  <own|p> Link next;
  <p|p> X data;
  Link(<p|p> X d1) {
    next = null;
    data = d1;
  }
}
class XStack<p> {
  <rep|p> Link top;
  XStack() { top = null; }
  void push(<p|p> X d2) {
    <rep|p> Link newTop = new <rep|p> Link(d2);  n
    newTop.next = top;
    top = newTop;
  }
  <p|p> X pop() {
    <rep|p> Link oldTop = top;
    top = oldTop.next;
    return oldTop.data;
  }
  boolean isEmpty() { return top == null; }
  public static void main(String[] args) {
    <rep|rep> XStack s = new <rep|rep> XStack();    s
    <rep|rep> X x = new <rep|rep> X();    x
    s.push(x);
  }
}
```

**Fig. 2.** XStack example

Note that *adapt* is partially defined. The first argument cannot contain rep which accounts for static visibility [3]. Otherwise, it will be considered as illegal and the *adapt* will fail.

This system simplifies [3] as it restricts the number of ownership parameters to one. It uses A-normal form Java syntax, and also we have taken the liberty to use *adapt* instead of substitution function $\sigma$, as *adapt* was somewhat more intuitive to us.

Fig. 2 shows a fully annotated and type checked XStack example, which is rephrased from [3]. The checking process is straightforward in this fully annotated program, as in most cases we only check if the types of both sides are compatible for each assignment. In order to check newTop.next = top, however, we need to use the viewpoint adaptation function. Because newTop has type $\langle \mathsf{rep}|p\rangle$ and next has type $\langle \mathsf{own}|p\rangle$, the result of the viewpoint adaptation $adapt(\langle \mathsf{own}|p\rangle, \langle \mathsf{rep}|p\rangle) = \langle \mathsf{rep}|p\rangle$ is exactly the type of top.

## 3 Flow Analysis

Our type checker works on fully annotated programs, as well as partially annotated programs. We believe that neither fully automatic inference or fully manual annotating are feasible choices. We envision that programmers should provide a small set of annotations on the fields, locals and/or allocation sites that they care about, and let the system fill in the remaining annotations and type check the program. However, a program may have many feasible ownership typings. For instance, a flat ownership structure is one of the possible typings. One major challenge for ownership inference is that there is no notion of an optimal typing. We refer readers to our ongoing work on optimality of ownership type inference [7]. In this work, we conjecture that the checker should infer ownership types that respect dominance as much as possible. Currently, we provide annotations on all allocation sites; in the future, we plan to extend our system and experiment with different settings.

### 3.1 Type Mapping

First we define an annotation mapping $T$ from keys to values. The keys in the mapping are (1) variables, including method parameters, (2) field names, (3) allocation site indices and (4) method names used for return types. The values in the mapping are sets of possible type annotations. For instance, $T(\mathsf{x}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|p\rangle\}$ means the annotation of variable x can be one of $\langle \mathsf{own}|\mathsf{own}\rangle$ or $\langle \mathsf{own}|p\rangle$. For the rest of the paper we use "variables" to refer to all kinds of keys: variables, fields, allocation sites, and method names.

$T$ is initialized as follows. Programmer-annotated variables are initialized to the programmer-provided annotation. In our running example, allocation site $n$ is annotated by the user as $\langle \mathsf{rep}|p\rangle$ and therefore $T(n) = \{\langle \mathsf{rep}|p\rangle\}$. Variables that are not annotated are initialized to the maximal set of annotations $U$, i.e., $T(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{rep}|p\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|p\rangle, \langle p|p\rangle\}$. Static fields, variables of boxed primitive types, string type, as well as variables defined in libraries receive special default annotations which we will discuss later in the paper. The flow analysis iterates over the statements in the program and refines the initial sets until it reaches a fixpoint.

### 3.2 Transfer Functions

Now we describe the transfer functions which take as input the above defined type mapping $T$ and output an updated type mapping $T'$. First, we define three adapt functions that act on sets:

$$\begin{aligned}
Adapt_{out}(S_{in}, S_{rcv}) &= \{t_{out} \mid t_{out} = adapt(t_{in}, t_{rcv}) \text{ where } t_{in} \in S_{in}, t_{rcv} \in S_{rcv}\}\\
Adapt_{rcv}(S_{out}, S_{in}) &= \{t_{rcv} \mid t_{out} = adapt(t_{in}, t_{rcv}) \text{ where } t_{out} \in S_{out}, t_{in} \in S_{in}\}\\
Adapt_{in}(S_{out}, S_{rcv}) &= \{t_{in} \mid t_{out} = adapt(t_{in}, t_{rcv}) \text{ where } t_{out} \in S_{out}, t_{rcv} \in S_{rcv}\}
\end{aligned}$$

What the above functions do is infer the third annotation based on the other two. The first function, $Adapt_{out}$, extends $adapt$ to act on sets in a straightforward manner: it returns all types $t_{out}$ that can be derived from the sets $S_{in}$ and $S_{rcv}$. The second and third functions, $Adapt_{rcv}$ and $Adapt_{in}$ are essentially inverses of $adapt$ that act on sets. $Adapt_{rcv}$ takes as arguments the outside set

$S_{out}$ and the inside set $S_{in}$, and returns the receiver set $S_{rcv}$. Similarly, $Adapt_{in}$ takes as arguments the outside set $S_{out}$ and the receiver set $S_{rcv}$, and returns the inside set $S_{in}$.

Based on these *Adapt* functions, we can define the transfer functions on the kinds of statements shown in Fig. 1.

1. $\mathsf{x} = \mathsf{new}^j\ \mathsf{C}()$
$$f_1(T) = T' = [\mathsf{x} \mapsto S][j \mapsto S]\ T$$
where $S = T(\mathsf{x}) \cap T(j)$. The transfer function takes as argument the current mapping $T$ and returns a new mapping $T'$. $T'$ differs from $T$ only in the mappings for $\mathsf{x}$ and $j$: the sets of $\mathsf{x}$ and allocation site $j$ are unified by taking the intersection of the current sets.
2. $\mathsf{x} = \mathsf{y}$
$$f_2(T) = T' = [\mathsf{x} \mapsto S][\mathsf{y} \mapsto S]\ T$$
where $S = T(\mathsf{x}) \cap T(\mathsf{y})$. The transfer function takes as argument the current mapping $T$ and returns a new mapping $T'$. Again, $T'$ differs from $T$ only in the mappings for $\mathsf{x}$ and $\mathsf{y}$: the sets of $\mathsf{x}$ and $\mathsf{y}$ are unified.
3. $\mathsf{x.f} = \mathsf{y}$ and $\mathsf{y} = \mathsf{x.f}$
$$f_3(T) = T' = [\mathsf{y} \mapsto S''][\mathsf{f} \mapsto S][\mathsf{x} \mapsto S']\ T$$
where $S'' = Adapt_{out}(T(\mathsf{f}), T(\mathsf{x})) \cap T(\mathsf{y})$, $S = Adapt_{in}(S'', T(\mathsf{x})) \cap T(\mathsf{f})$, $S' = Adapt_{rcv}(S'', S) \cap T(\mathsf{x})$. The transfer function first unifies the type of the left-hand-site with the type of the right-hand-side to compute set $S''$. Subsequently, it refines the type for $\mathsf{f}$ and the type for $\mathsf{x}$ based on $S''$.
In our running example, consider applying this transfer function to $\mathsf{newTop.next} = \mathsf{top}$ at line 14. Assume $T(\mathsf{newTop}) = \{\langle \mathsf{rep}|p\rangle\}$, $T(\mathsf{next}) = U$ and $T(\mathsf{top}) = U$. $Adapt_{out}(U, \langle \mathsf{rep}|p\rangle) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|p\rangle, \langle p|p\rangle\}$ (recall that *adapt* restricts the first argument to only three possible values). Therefore, $S'' = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|p\rangle, \langle p|p\rangle\}$. $S = Adapt_{in}(S'', \{\langle \mathsf{rep}|p\rangle\}) \cap T(\mathsf{next}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|p\rangle, \langle p|p\rangle\}$. Finally, $S' = Adapt_{rcv}(S'', S) \cap T(\mathsf{newTop}) = \{\langle \mathsf{rep}|p\rangle\}$. Therefore, the result of the application is $T'(\mathsf{newTop}) = \{\langle \mathsf{rep}|p\rangle\}$, $T'(\mathsf{next}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|p\rangle, \langle p|p\rangle\}$ and $T'(\mathsf{top}) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|p\rangle, \langle p|p\rangle\}$.
4. $\mathsf{this.f} = \mathsf{y}$ and $\mathsf{y} = \mathsf{this.f}$
$$f_4(T) = T' = [\mathsf{y} \mapsto S][\mathsf{f} \mapsto S]\ T$$
where $S = T(\mathsf{f}) \cap T(\mathsf{y})$. No viewpoint adaptation is needed.
Consider applying the transfer function to $\mathsf{top} = \mathsf{newTop}$ at line 15. As a result $T'(\mathsf{top}) = \{\langle \mathsf{rep}|p\rangle\}$. Applying $f_3$ on $\mathsf{newTop.next} = \mathsf{top}$ in a subsequent iteration refines the set for $\mathsf{next}$ to $\{\langle \mathsf{own}|p\rangle\}$.
5. $\mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})$
Let $\mathsf{m}(\overline{\mathsf{p}})$ be the compile-time target at call $\mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})$, and let $\overline{\mathsf{z}} = \mathsf{z}_1, \ldots, \mathsf{z}_n$ and $\overline{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n$.
$$f_5(T) = T' = \overline{[\mathsf{z}_i \mapsto S_{\mathsf{z}_i}][\mathsf{p}_i \mapsto S_{\mathsf{p}_i}]}[\mathsf{x} \mapsto S_\mathsf{x}][\mathsf{m} \mapsto S_\mathsf{m}][\mathsf{y} \mapsto S_\mathsf{y}]\ T$$

where $S_{\mathsf{z}_i} = Adapt_{out}(T(\mathsf{p}_i), T(\mathsf{y})) \cap T(\mathsf{z}_i)$, $S_{\mathsf{p}_i} = Adapt_{in}(S_{\mathsf{z}_i}, T(\mathsf{y})) \cap T(\mathsf{p}_i)$, $S_{\mathsf{x}} = Adapt_{out}(T(\mathsf{m}), T(\mathsf{y})) \cap T(\mathsf{x})$, $S_{\mathsf{m}} = Adapt_{in}(S_{\mathsf{x}}, T(\mathsf{y})) \cap T(\mathsf{m})$, and $S_{\mathsf{y}} = (\cap_{i=1\ldots n} Adapt_{rcv}(S_{\mathsf{p}_i}, S_{\mathsf{z}_i})) \cap Adapt_{rcv}(S_{\mathsf{m}}, S_{\mathsf{x}}) \cap T(\mathsf{y})$.
This function performs viewpoint adaptation by refining the annotation sets for triple $\mathsf{y}$, $\mathsf{m}$, $\mathsf{x}$, and also for each triple $\mathsf{y}, \mathsf{p}_i, \mathsf{z}_i$.

6. $\mathsf{x} = \mathsf{this.m}(\overline{\mathsf{z}})$
Again, let $\mathsf{m}(\overline{\mathsf{p}})$ be the compile-time target at call $\mathsf{x} = \mathsf{this.m}(\overline{\mathsf{z}})$, and let $\overline{\mathsf{z}} = \mathsf{z}_1, \ldots, \mathsf{z}_n$ and $\overline{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n$.

$$f_6(T) = T' = \overline{[\mathsf{z}_i \mapsto S_i][\mathsf{p}_i \mapsto S_i]}[\mathsf{x} \mapsto S][\mathsf{m} \mapsto S]\; T$$

where $S_i = T(\mathsf{p}_i) \cap T(\mathsf{z}_i)$ and $S = T(\mathsf{x}) \cap T(\mathsf{m})$.

### 3.3 Fixpoint Iteration

The flow analysis is a fixpoint iteration that works as follows: it first initializes the mapping $T$ as described earlier in this section, and keeps iterating over the program with the above defined transfer functions until one of the following happens: (1) $T$ reaches a fixpoint, i.e., $T$ remains unchanged from the previous iteration, in which case the analysis terminates successfully, or (2) a key gets assigned the empty set, in which case the analysis terminates with an error.

The algorithm terminates. For each transfer function $f_i$, we have $T'(\mathsf{x}) \subseteq T(\mathsf{x})$ for an arbitrary key $\mathsf{x}$. As a result, $T$ changes at most $O(n * |U|) = O(n)$ times where $n$ is the total number of keys, and $U$ is the maximal set of annotations. Since there are $O(n)$ statements, the overall complexity of the algorithm is $O(n^2)$.

## 4 Type Checking

The type checker can perform type checking of fully annotated programs as well as partially annotated programs. It takes as input the partially annotated source, applies the flow analysis described in the previous section, and type checks with the result of the inference. Note, however, that the result of our inference is a mapping from variables to *sets* of types, not to a single type. When performing type checking, our checker picks the most specific (i.e., smallest) type in the set according to the following ordering:

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

Unfortunately however, this heuristic may not always work. If the programmer-provided annotations are too permissive (i.e., there are none, or too few), type checking with the most specific type may fail. For example, consider the program:

```
x = new A();
y = new <own|own> C();
x.f = y;
```

Applying function $f_3$ on $\mathsf{x.f} = \mathsf{y}$ results in $T(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|p\rangle\}$, $T(\mathsf{f}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|p\rangle, \langle p|p\rangle\}$ and $T(\mathsf{y}) = \{\langle \mathsf{own}|\mathsf{own}\rangle\}$. Type checking with the most specific type from each set obviously does not work because $adapt(\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, not $\langle \mathsf{own}|\mathsf{own}\rangle$.

In our current setting we provide annotations on *all allocation sites*, and inference and checking works well. In practice, the vast majority of variables are mapped to a singleton set and even if they are not, type checking with the most specific type works. In fact, there was only one case where the heuristic did not work and this was after adding norep (see next section); we resolved this case with an explicit annotation to one variable, picking a type other than the most specific one from the variable's set.

It is not surprising that annotating all allocation sites results in unique types. The flow analysis propagates the type annotation at allocation site $j$ to every variable that refers to the object created at $j$. In the future, we will extend the flow analysis to handle an arbitrary set, including the empty set, of programmer-provided annotations.

## 5 Implementation

To better illustrate inference and checking, we ignored some features in Java, including static fields, static methods, libraries and subclassing. However, we have to consider them in the implementation in order to type check programs in the real world.

First, we add a new value norep for $q_0$ and $q_1$ in annotation $\langle q_0|q_1\rangle$. norep denotes that an object is owned by the root environment and might be accessed by any other object (see [3]). We extend $U$ and the ordering with additional types:

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

We add additional viewpoint adaptation rules for norep below:

$$\begin{aligned} adapt(\langle \mathsf{own}|\mathsf{norep}\rangle, \langle q_0|q_1\rangle) &= \langle q_0|\mathsf{norep}\rangle \\ adapt(\langle p|\mathsf{norep}\rangle, \langle q_0|q_1\rangle) &= \langle q_1|\mathsf{norep}\rangle \\ adapt(\langle \mathsf{norep}|\mathsf{norep}\rangle, \langle q_0|q_1\rangle) &= \langle \mathsf{norep}|\mathsf{norep}\rangle \end{aligned}$$

### 5.1 Java Features

*Static fields* In our implementation, all static fields receive default annotation $\langle \mathsf{norep}|\mathsf{norep}\rangle$ as static fields belong to the root environment.

*Static methods* There are no receivers for static methods. In our implementation, we assume static methods have a virtual receiver this. For instance, x=C.sm(y) is treated using transfer functioin $f_6$, as if it were x=this.sm(y).

*Boxed primitives* All variables of type String, StringBuffer, as well as boxed primitives (e.g., Long), receive default type $\langle$norep|norep$\rangle$.

*Subclassing* The way we handle subclassing is to unify the annotations for both subclasses and superclasses. That is, their methods should have the same annotations on parameters and return types. In our implementation, annotations are not only propagated by assignment statements in the transfer functions, but also by overridden methods.

*Arrays* An array object is annotated on the brackets [ ]. For instance, $\langle p|p\rangle$ C $\langle$own$|p\rangle$[] lists declares a variable lists, where the array object is of type $\langle$own|p$\rangle$ and the elements are of type $\langle p|p\rangle$. When adapting an array variable, both the array type and the element type are adapted, e.g. $adapt(\langle p|p\rangle\langle \mathsf{own}|p\rangle[], \langle \mathsf{rep}|\mathsf{own}\rangle)$ gives a compound adapted type $\langle$own|own$\rangle\langle$rep|own$\rangle$[].

*Libraries* Handling of libraries is one of the major challenges for a realistic inference and checking tool. In our checker, static fields, parameters and return types in static library methods receive default annotation $\langle$norep|norep$\rangle$. We take this conservative default typing for static library methods because in general, they can pass arguments to static fields and may return data from static fields. In two cases, we explicitly typecast arguments to $\langle$norep|norep$\rangle$, in order to preserve the typings of arguments. The methods involved were System.arraycopy() and Collections.sort(); these typecasts are safe as those two library methods will never expose their parameters to the outside world.

Parameters and return types of instance library methods map to a set of $\{\langle \mathsf{own}|p\rangle, \langle p|p\rangle\}$ in the type mapping $T$ by default, and the actual annotation would be decided by the flow analysis. Originally, we intended to assign them the annotation set $\{\langle \mathsf{own}|p\rangle\}$. Assuming that (1) all local variables in instance library methods are typed $\langle$own$|p\rangle$, (2) instance methods *do not access* static fields and methods, and (3) subclassing of library classes is handled correctly, we will have that library code type checks and does not break ownership. Unfortunately, $\{\langle \mathsf{own}|p\rangle\}$ does not work well with containers as it forces the container to have the same type as its element (e.g., given that the formal parameter of add is $\langle$own$|p\rangle$, the only way to type check hs.add(e) is to have hs and e have the same ownership type). This is unacceptable because often the element escapes while the container stays confined. Therefore, we typed library variables as $\{\langle \mathsf{own}|p\rangle, \langle p|p\rangle\}$ and let the flow analysis to decide the actual annotation.

### 5.2 Results

The inference analysis and type checking are implemented in Java as a pluggable checker in the Checker Framework [12]. We evaluate the type checker on two benchmarks, javad and SPECjbb. All experiments were done on a desktop with AMD Althlon(tm) II X2 245 Processor @ 2.9GHz and 4GB of RAM, and max heap size is set to 512MB. The software environment consists of Linux 2.6.35, JDK 1.6.0 and the Checker Framework 1.1.2.

| | #LOC | #Classes | #Methods | #Annotations | Ratio | Running Time |
|---|---|---|---|---|---|---|
| javad | 4205 | 45 | 157 | 48 | 12/1KLOC | 10s |
| SPECjbb | 12076 | 61 | 551 | 244 | 20/1KLOC | 47s |

**Table 1.** Benchmarks

Table 1 shows the general information about the benchmarks. **#LOC** gives the total lines of code; **#Classes** gives the total number of user classes; **#Methods** gives the total number of user methods. **#Annotations** gives the total number of annotations that we added, and **Ratio** gives the number of annotations per one thousand lines of code, 12 for javad and 20 for SPECjbb. The annotations that we added were provided by the inference tool described in [10]. Finally, **Running Time** gives the running time for inference and checking; this is an average of three runs.

| | javad | | | | SPECjbb | | | |
|---|---|---|---|---|---|---|---|---|
| | #Locals | #Returns | #Fields | #Allocs | #Locals | #Returns | #Fields | #Allocs |
| ⟨rep\|rep⟩ | 8 | 2 | 2 | 6 | 76 | 1 | 58 | 92 |
| ⟨rep\|own⟩ | 0 | 0 | 5 | 5 | 1 | 0 | 3 | 5 |
| ⟨rep\|$p$⟩ | 6 | 1 | 11 | 24 | 0 | 0 | 0 | 0 |
| ⟨rep\|norep⟩ | 0 | 0 | 0 | 0 | 7 | 0 | 2 | 8 |
| ⟨own\|own⟩ | 1 | 0 | 1 | 1 | 39 | 14 | 19 | 25 |
| ⟨own\|$p$⟩ | 10 | 3 | 6 | 10 | 60 | 29 | 21 | 21 |
| ⟨own\|norep⟩ | 0 | 0 | 0 | 0 | 6 | 1 | 3 | 3 |
| ⟨$p$\|$p$⟩ | 63 | 3 | 11 | 1 | 5 | 0 | 1 | 0 |
| ⟨$p$\|norep⟩ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ⟨norep\|norep⟩ | 70 | 29 | 15 | 9 | 375 | 103 | 185 | 90 |
| ⟨norep\|norep⟩* | 1 | 0 | 1 | 1 | 118 | 12 | 60 | 90 |

**Table 2.** Inferred and checked results for benchmarks javad and SPECjbb

The results of the inference are shown in Table 2. Note that in both benchmarks there are many ⟨norep|norep⟩ annotations. The majority of those are boxed primitives and are assigned ⟨norep|norep⟩ by default. The last row, ⟨norep|norep⟩*, gives the number of ⟨norep|norep⟩ assigned to objects that are not boxed primitives; that is, the ⟨norep|norep⟩ was assigned as a result to flow to or from static fields. The sum of columns **#Allocs** differs from **#Annotations** in Table 1 because we did not provide annotations on allocation sites for boxed primitives (allocation sites counted in row ⟨norep|norep⟩ were assigned by the checker and are excluded from the count in **#Annotations**).

Overall, these results show encouraging precision. For javad, 50% (18 out of 36) of the fields, and 73% (35 out of 48) of the allocation sites have owner rep. We exclude boxed primitives from the total count. For SPECjbb, 38% (63 out of 167) of the fields, and 43% (105 out of 244) of the allocation sites have owner rep. Again, we exclude boxed primitives from the total count.

Finally, we address the question of how important the restriction to one ownership parameter is. One would expect that allowing only one ownership parameter would impose additional constraints and would cause many objects dominated by this in the object graph, to receive owner other than rep; in other words, one would expect that the restriction would flatten the ownership tree. In order to address the above question we compared the type inference results to the dominance inference results from [10]. We examined all fields reported as *dom* (i.e., dominated by their enclosing object) by the analysis in [10] that were inferred as having owner other than rep in their ownership type; similarly, we examined all allocation sites reported as *dom* (i.e., dominated by their creating object) by [10] that were inferred as having owner other than rep.

For javad, only 1 field, and only 1 allocation site, both referring to the same object, were raised from rep to own. In this case, allowing more than one ownership parameter would have allowed the rep typing.

For SPECjbb, 7 fields and 16 allocation sites were raised from rep. In about half of those cases the raising is directly due to passing implicit parameter this as an argument. For example, in cases such as x = new X(this); dominance analysis would infer (correctly) that this dominates the newly created X object. However, type inference raises the type to own because this is of type $\langle \mathsf{own}|p\rangle$ and the only solution to $adapt(t, t') = \langle \mathsf{own}|p\rangle$ is $t = \langle \mathsf{own}|p\rangle, t' = \langle \mathsf{own}|p\rangle$. One can show that for these cases, allowing more than one ownership parameter would not have allowed rep typing either! Although we believe that in some of the other cases raising may have been caused indirectly by this, we estimate that about 4 fields and 10 allocation sites could have been annotated as rep if the system had allowed more than one ownership parameter.

These results are both surprising and encouraging. Surprising, because one would have expected the restriction to one ownership parameter to have a more dramatic negative impact. Encouraging, because the restriction simplifies the ownership type system, which may ease practical adoption. Clearly, more studies are needed in order to draw a definitive conclusion.

## 6 Related Work

We discuss the most recent related work on ownership inference.

There are many work on dynamic inference for ownership-related properties [1, 6, 11, 14]. Although dynamic inference is more precise, it suffers from problems of safety and performance overhead. In contrast, static inference is safer and efficient but less precise. Aldrich et al. [2], and Ma and Foster [9] present work on inferring uniqueness and ownership statically. Aldrich et al. uses a constraint system to infer alias annotations, while Ma and Foster combine an intraprocedural points-to analysis with an interprocedural predicate solution to infer uniqueness and ownership. Poetzsch-Heffter et al. [13] present a ownership type system, as well as a constraint-based inference analysis to lower the annotation burden.

Dielt et al. [4] present a tunable static inference for Generic Universe Types. They encode the Generic Universe Types constraints as a boolean satisfiability (SAT) problem and use a SAT solver to find a correct Universe typing. The time complexity of their inference is exponential. In contrast, our inference is based on flow analysis which gives a polynomial algorithm. In addition, their inference is tunable—programmers can express their preference for certain solutions by providing weights to the SAT solver. Our inference provides similar functionality—the checker accepts partially annotated programs where the programmer can annotate any subset of variables to express his/her preference.

We compare this paper with our recent work on ownership inference presented in [10], in which we first performed dominance inference and then ownership inference based on the dominance results. This work is an improvement over [10] in two ways. First, it allows programmers to express their preference by accepting partially annotated programs while still respecting the dominance relations as much as possible. Second, it provides a type checker which is not available in [10].

## References

1. R. Agarwal. Type inference for parameterized race-free Java. In *Verification, Model Checking, and Abstract*, pages 149—-160, 2004.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. *ACM SIGPLAN Notices*, 37(11):311–330, Nov. 2002.
3. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33(10):48–64, Oct. 1998.
4. W. Dietl, M. D. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2011. To appear.
5. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
6. W. Dietl and P. Muller. Runtime universe type inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 72–80, 2007.
7. W. Huang and A. Milanova. On optimality of ownership type inference. Poster at ECOOP 2011.
8. Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-Based Object Access Control. In *29th International Conference on Software Engineering (ICSE'07)*, pages 323–332. IEEE, May 2007.
9. K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for java. *ACM SIGPLAN Notices*, 42(10):423, Oct. 2007.
10. A. Milanova and J. Vitek. In *Proceedings of TOOLS Europe 2011.* to appear.
11. N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.
12. M. Papi, M. Ali, T. Correa Jr, J. Perkins, and M. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, New York, New York, USA, 2008. ACM.
13. A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Program analysis and compilation, theory and practice. chapter Infering ownership types for encapsulated object-oriented program components, pages 120–144. Springer-Verlag, Berlin, Heidelberg, 2007.
14. Y. Zibin, A. Potanin, P. Li, and M. Ali. Ownership and immutability in generic Java. *Proceedings of the ACM*, pages 598–617, 2010.