Inference and Checking of Object Ownership

Wei Huang[1], Werner Dietl[2], Ana Milanova[1], and Michael D. Ernst[2]

[1] Rensselaer Polytechnic Institute [2] University of Washington

Abstract. Ownership type systems describe a heap topology and enforce an encapsulation discipline; they aid in various program correctness and understanding tasks. However, the annotation overhead of ownership type systems has hindered their widespread use. We present a unified framework for specification, type inference and type checking of ownership type systems, and instantiate the framework for two such systems: Universe Types and Ownership Types. We present an objective metric defining a "best typing" for these type systems, and develop an inference approach that maximizes the metric. The programmer can influence the inference by adding partial annotations to the program. We implemented the approach on top of the Checker Framework and present the results of an experimental evaluation.

1 Introduction

When a type system requires annotations in the source code, the annotation burden on programmers inhibits practical adoption. Therefore, it is important to help programmers transform unannotated or partially-annotated programs to fully-annotated ones. Another benefit of type inference is that it reveals valuable information about how existing programs use the concepts expressed in the type system.

Automatic type inference is especially difficult for type systems that allow multiple valid typings, such as ownership type systems [7]. The notion of the "best typing" is not well-understood or formalized.

This paper presents a unified framework for specifying ownership-like type systems as well as efficient type inference and checking techniques. We give a formal way to define the best typing and design efficient type inference techniques that infer best typings. We have instantiated the framework for two well-known ownership type systems: Universe Types [8], which enforces the *owner-as-modifier* encapsulation discipline, and Ownership Types [4], which enforces the *owner-as-dominator* encapsulation discipline, and present an empirical evaluation.

This paper makes the following contributions:

- A unified framework for specifying the type rules of ownership type systems and instantiations of the framework for two well-known ownership type systems, Universe Types (UT), and Ownership Types (OT). (See Sect. 2.)
- A formalization of the notion of "best typing" for ownership type systems. The programmer specifies a ranking over all valid typings; the highest ranked

cd	::= class C extends D { $\overline{fd}$ $\overline{md}$ }	*class*
fd	::= τ f	*field*
md	::= τ m(τ x) { $\overline{\tau\ \mathsf{y}}$; return y }	*method*
s	::= s; s | x = new τ() | x = y | x.f = y | this.f = y | x = y.f | x = this.f | x = y.m(z) | x = this.m(z)	*statement*
τ	::= q C	*qualified type*
q	$\in$ Q	*qualifier*

Fig. 1. Syntax of a core OO language. The set Q of all qualifiers q is a framework parameter instantiated for specific ownership type systems.

typing is the best typing. The ranking is a heuristic reflecting the desire for deep ownership trees — higher ranked (i.e., "better") typings give rise to deeper runtime ownership trees. Deep ownership trees are desirable, because they expose high degree of encapsulation. (See Sect. 3.)

- A unified type inference approach. The inference reflects programmer intent in two ways: (1) it accepts a programmer-specified ranking over typings, which guides the automatic inference towards the best of many valid typings, and (2) it accepts partially-annotated programs and seamlessly integrates programmer-provided annotations with automatic inference: the programmer may choose to annotate a subset of the variables; the automatic inference fills in the rest, guided by the ranking towards the best typing. (See Sect. 4.)
- A formulation of Universe Types inference as an instance of the unified approach. We infer the "best UT typing", in quadratic time, without annotations. (See Sect. 4.3.)
- A demonstration that while the best UT typing is tractable, the best OT typing is challenging. Our approach cannot always infer the best OT typing without annotations. We scale Ownership Type inference by asking the programmer to provide a small number of annotations (6 per kLOC on average). We infer the "best OT typing" for the partially-annotated program in quadratic time. (See Sect. 4.4.)
- An empirical evaluation which presents type inference results for UT and OT on Java programs of up to 110kLOC, and a comparison of UT and OT. (See Sect. 5.)

2 Unified Framework for Ownership Type Systems

This section describes our unified framework for specifying ownership type systems. The framework can be instantiated to specific ownership type systems. Sect. 2.1 describes the framework's unified typing rules, Sect. 2.2 instantiates the framework for Universe Types, and Sect. 2.3 instantiates it for Ownership Types.

For brevity, we restrict our formal attention to a core calculus in the style of Vaziri et al. [26] whose syntax appears in Fig. 1. The language models Java with a syntax in A-normal form. For brevity, we assume in the presentation that all methods have a single parameter; our implementation handles the general case.

2.1 Framework and Unified Typing Rules

The framework is instantiated to a specific type system by defining three framework parameters: (1) the set of type qualifiers Q with the corresponding subtyping hierarchy, (2) the viewpoint adaptation function $\triangleright$ (described below), and (3) type-system-specific constraints $\mathcal{B}$, enforced in addition to the standard subtyping and viewpoint adaptation constraints.

In contrast to a formalization of pure Java, a type τ has two orthogonal components: ownership type qualifier q and Java class type C. The ownership type system is *orthogonal* (i.e., independent) to the Java type system, which allows us to specify typing rules over type qualifiers q alone.

Framework parameter $\triangleright$ defines *viewpoint adaptation* [8]. For example, the type of $\mathsf{x.f}$ is not just the declared type of field f — it is the type of f adapted from the point of view of x. In ownership type systems, viewpoint adaptation adapts the type of a field, formal parameter, or return type, from the viewpoint of the *receiver* at the corresponding field access or method call to the viewpoint of the current object this. Viewpoint adaptation is performed at field accesses and method calls and is written $q \triangleright q'$, which denotes that type q' is adapted from the point of view of type q to the viewpoint of the current object this. Viewpoint adaptation rules for each type system are given in Sections 2.2 and 2.3.

Fig. 2 shows the unified typing rules over the A-normal-form Java syntax. The figure makes use of the three framework parameters. The environment Γ is used to look up the type qualifier of a variable. Rule (TNEW) ensures that the instantiated type is a subtype of the type of the left-hand side and enforces the additional type-system-specific constraints determined by $\mathcal{B}$. Similarly, rule (TASSIGN) checks the types in assignments. The rules in Fig. 2 separate access through the current object this from other accesses. Rule (TWRITE) adapts the type of the field, and creates the subtype constraint between the type on the right-hand-side and the adapted type of f. Auxiliary function *typeof*(f) retrieves the type of field f from its declaration. Rule (TREAD) ensures that the adapted field type is a subtype of the type of the left-hand-side. Rule (TCALL) uses *typeof*(m) to retrieve the type of method m, namely $q \to q'$, from its declaration. Rule (TCALL) then creates the expected subtyping constraint between the type of the actual argument z and the adapted type of the formal parameter, as well as the subtyping constraint between the adapted return type and the type of the left-hand-side x. Finally, rules (TWRITETHIS), (TREADTHIS), and (TCALLTHIS) perform the corresponding operations, without viewpoint adaptation.

We now instantiate the framework for two well-known ownership type systems, Universe Types (UT) [8, 5] and Ownership Types (OT) [4]. The framework can also be instantiated for a variety of other ownership-like type systems, including EnerJ [22], a type system for energy efficiency, and AJ [26], a type system for data-centric synchronization.

2.2 Universe Types

Universe Types (UT) [8, 5] is a lightweight ownership type system that optionally enforces the *owner-as-modifier* encapsulation discipline. Informally, this means

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad q <: q_\mathsf{x} \quad \mathcal{B}_{(\textsc{tnew})}(q_\mathsf{x}, q)}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\textsc{tnew})$$

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad q_\mathsf{y} <: q_\mathsf{x} \quad \mathcal{B}_{(\textsc{tassign})}(q_\mathsf{x}, q_\mathsf{y})}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\textsc{tassign})$$

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad q_\mathsf{y} <: q_\mathsf{x} \triangleright q_\mathsf{f} \quad \mathcal{B}_{(\textsc{twrite})}(q_\mathsf{x}, q_\mathsf{f}, q_\mathsf{y})}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\textsc{twrite})$$

$$\frac{\mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad q_\mathsf{y} <: q_\mathsf{f} \quad \mathcal{B}_{(\textsc{twritethis})}(q_\mathsf{f}, q_\mathsf{y})}{\Gamma \vdash \mathsf{this.f} = \mathsf{y}}\ (\textsc{twritethis})$$

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad q_\mathsf{y} \triangleright q_\mathsf{f} <: q_\mathsf{x} \quad \mathcal{B}_{(\textsc{tread})}(q_\mathsf{y}, q_\mathsf{f}, q_\mathsf{x})}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{\Gamma(\mathsf{y}) = q_\mathsf{y} \quad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad q_\mathsf{f} <: q_\mathsf{x} \quad \mathcal{B}_{(\textsc{treadthis})}(q_\mathsf{f}, q_\mathsf{x})}{\Gamma \vdash \mathsf{x} = \mathsf{this.f}}\ (\textsc{treadthis})$$

$$\frac{\mathit{typeof}(\mathsf{m}) = q \to q' \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad \Gamma(\mathsf{z}) = q_\mathsf{z} \quad q_\mathsf{z} <: q_\mathsf{y} \triangleright q \quad q_\mathsf{y} \triangleright q' <: q_\mathsf{x} \quad \mathcal{B}_{(\textsc{tcall})}(\mathsf{m}, q_\mathsf{y}, q_\mathsf{x})}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ (\textsc{tcall})$$

$$\frac{\mathit{typeof}(\mathsf{m}) = q \to q' \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{z}) = q_\mathsf{z} \quad q_\mathsf{z} <: q \quad q' <: q_\mathsf{x} \quad \mathcal{B}_{(\textsc{tcallthis})}(\mathsf{m}, q_\mathsf{x})}{\Gamma \vdash \mathsf{x} = \mathsf{this.m(z)}}\ (\textsc{tcallthis})$$

Fig. 2. Unified typing rules. The ownership type system is independent from the Java type system, which allows us to specify the typing rules over qualifiers q alone.

that an object can be modified only by its owner and by its peers, i.e., objects that have the same owner. There are three source-level qualifiers, i.e., $Q_{UT} = \{\mathsf{peer}, \mathsf{rep}, \mathsf{any}\}$:

- peer: an object that is referenced by a peer reference x is part of the same representation as the current object. In other words, the two objects have the same owner.
- rep: an object that is referenced by a rep reference x is part of the current (i.e., this) object's representation. In other words, the current object is the *owner* of the object referenced by x.
- any: the any qualifier does not provide any information about the ownership of the object.

The formalization of Universe Types uses the qualifier lost to express that the result of viewpoint adaptation cannot be expressed statically, that is, a type declaration enforces an ownership constraint, but the constraint is not expressible from the current viewpoint. Qualifier lost is used only internally and users cannot annotate references as lost. In contrast to previous work [5], we type the current object this as peer and use separate rules for accesses through this, instead of adding a self qualifier.

```
1  class XStack {
2    any Link top;
3    XStack() {
4      top = null;
5    }
6    void push(any X d1) {
7      rep Link newTop;
8      newTop = new rep Link(); [l]
9      newTop.init(d1);
10     newTop.next = top;
11     top = newTop;
12   }
13   void main(String[] arg) {
14     rep XStack s;
15     s = new rep XStack(); [s]
16     any X x = new rep X(); [x]
17     s.push(x);
18   }
19 }
20 class Link {
21   any Link next;
22   any X data;
23   void init(any X d2) {
24     next = null;
25     data = d2;
26   }
27 }
```

```
1  class XStack {
2    ⟨rep|p⟩ Link top;
3    XStack() {
4      top = null;
5    }
6    void push(⟨p|p⟩ X d1) {
7      ⟨rep|p⟩ Link newTop;
8      newTop = new ⟨rep|p⟩ Link(); [l]
9      newTop.init(d1);
10     newTop.next = top;
11     top = newTop;
12   }
13   void main(String[] arg) {
14     ⟨rep|rep⟩ XStack s;
15     s = new ⟨rep|rep⟩ XStack(); [s]
16     ⟨rep|rep⟩ X x = new ⟨rep|rep⟩ X(); [x]
17     s.push(x);
18   }
19 }
20 class Link {
21   ⟨own|p⟩ Link next;
22   ⟨p|p⟩ X data;
23   void init(⟨p|p⟩ X d2) {
24     next = null;
25     data = d2;
26   }
27 }
```

Fig. 3. A program with qualifiers for UT (left) and OT (right) as inferred by our tool. The boxed italic letters denote object allocation sites.

The qualifiers form the following subtyping hierarchy:

$$\mathsf{rep} <: \mathsf{lost} \qquad \mathsf{peer} <: \mathsf{lost} \qquad \mathsf{lost} <: \mathsf{any}$$

that is, qualifiers peer and rep are incomparable to each other and are subtypes of lost, and all qualifiers are below any.

Viewpoint adaptation in UT is defined as follows:

$$\begin{array}{llll} \mathsf{peer} & \triangleright\ \mathsf{peer} & = \mathsf{peer} & \\ \mathsf{rep} & \triangleright\ \mathsf{peer} & = \mathsf{rep} & \\ \_ & \triangleright\ \mathsf{any} & = \mathsf{any} & \\ q & \triangleright\ q' & = \mathsf{lost} & \text{otherwise} \end{array}$$

Viewpoint adaptation is applied only when the receiver is not this. The type of the receiver is q_{x} at (TWRITE), q_{y} at (TREAD) and q_{y} at (TCALL). Consider x.f = y. If x is rep, then the current object is the owner of the x object. If the type of f is peer, then the x object and field f object are peers. Therefore, the current object is the

owner of the f object, which is expressed by the fact that the type of f, adapted from the point of view of x's rep, is rep.

UT imposes additional constraints, beyond the standard subtyping and viewpoint adaptation constraints. In our framework, these constraints are expressed by framework parameters $\mathcal{B}$:

$$\begin{array}{ll}
\mathcal{B}_{(\textsc{tnew})}(q_l, q_r) & = \{q_r \neq \mathsf{any}\} \\
\mathcal{B}_{(\textsc{twrite})}(q_r, q_f, q_o) & = \{\underline{q_r \neq \mathsf{any}}, q_r \rhd q_f \neq \mathsf{lost}\} \\
\mathcal{B}_{(\textsc{tcall})}(\mathsf{m}, q_r, q_o) & = \text{let } \mathit{typeof}(\mathsf{m}) = q \to q' \text{ in} \\
 & \quad \text{if } \mathit{impure}(\mathsf{m}) \text{ then } \{\underline{q_r \neq \mathsf{any}}, q_r \rhd q \neq \mathsf{lost}\} \\
 & \quad \text{else } \{q_r \rhd q \neq \mathsf{lost}\}
\end{array}$$

The $\mathcal{B}$ sets for (tassign), (twritethis), (tread), (treadthis), and (tcallthis) are all empty; these rules do not impose additional constraints.

In (tnew), the newly created object needs to be created in a concrete ownership context and therefore needs peer or rep as ownership qualifiers. In (twrite), the adapted field type cannot be lost, and in (tcall), the adapted formal parameter type cannot be lost.

The underlined constraints above enforce the owner-as-modifier encapsulation discipline — they disallow modifications in statically unknown contexts. The receiver cannot be any in (twrite) or in in (tcall) if the method is impure, that is, if the method might have nonlocal side effects. We use our method purity inference tool [13], which relies on a type system for reference immutability and is another instantiation of the unified framework described here. Note that, in contrast to other formalizations [7], we do not need to forbid lost as receiver, because our syntax here is in A-normal form and the programmer cannot explicitly write lost.

Fig. 3 (left) shows a program annotated with Universe types. Variable newTop at line 7 and the Link object l are typed rep, meaning that the XStack object is the owner of the Link object. References top (line 2) and next are any because they are never used to modify the object that they refer to. References d1 and d2 are any as well, as they are never used to modify the object they refer to.

Ownership type systems give rise to a hierarchical ownership structure shown with an *ownership tree*. Fig. 4 shows the object graph and the corresponding ownership tree for the program in Fig. 3. *root* is the owner of objects s and x and s is the owner of l.

2.3 Ownership Types

We now consider the classical Ownership Types (OT) [4], restricted to one ownership parameter. The system enforces the *owner-as-dominator* encapsulation discipline, meaning that an object cannot be exposed outside of the boundary of its owner, or in other words, all access paths to the object go through its owner. There are three base ownership modifiers in Ownership Types:

- rep refers to the current object this.

Fig. 4. Object graph (left) and ownership tree (right) for the example in Fig. 3. UT and OT give rise to the same ownership tree. In the object graph we show all references between objects. In the ownership tree we draw an arrow from the owned object to its owner and put all objects with the same owner into a dashed box.

- own refers to the owner of the current object.
- p is an ownership parameter passed to the current object.

OT qualifiers have the form $\langle q_0|q_1\rangle$, where q_0 and q_1 are one of rep, own, or p. A qualifier $\langle q_0|q_1\rangle$ for reference variable x is interpreted as follows. Let i be the object referenced by x. q_0 is the *owner* of i, from the point of view of the current object, and q_1 is the *ownership parameter* of i, again, from the point of view of the current object. Informally, the ownership parameter q_1 refers to an object, which objects referenced by i might use as owner. For example, $\langle$rep|own$\rangle$ x means that the owner of i is the current object this, and the ownership parameter passed to i is the owner of the current object. Transitively, objects referenced by i, for example, from its fields, can have as owner (1) i itself, by using rep, (2) the current object, by using own, or (3) the owner of the current object, by using p.

There are six type qualifiers:
$Q_{OT} = \{\langle$rep|rep$\rangle, \langle$rep|own$\rangle, \langle$rep|p$\rangle, \langle$own|own$\rangle, \langle$own|p$\rangle, \langle$p|p$\rangle\}$, and there is no subtyping hierarchy. The type of this is $\langle$own|p$\rangle$.

Viewpoint adaptation $\triangleright$ is defined as follows:

$$\begin{aligned}\langle q_0|q_1\rangle \triangleright \langle\text{own}|\text{own}\rangle &= \langle q_0|q_0\rangle\\ \langle q_0|q_1\rangle \triangleright \langle\text{own}|\text{p}\rangle &= \langle q_0|q_1\rangle\\ \langle q_0|q_1\rangle \triangleright \langle\text{p}|\text{p}\rangle &= \langle q_1|q_1\rangle\end{aligned}$$

Viewpoint adaptation disallows the adapted type from containing rep, which accounts for the static visibility constraint [4].

As an example, let us discuss the first rule: the adapted type of $\langle$own|own$\rangle$ from the point of view of $\langle q_0|q_1\rangle$ is $\langle q_0|q_0\rangle$. If an object i has type $\langle q_0|q_1\rangle$ from the point of view of the current this object, this means that the owner of i is q_0. If object j has type $\langle$own|own$\rangle$ from the point of view of i, this means that both j's owner and ownership parameter are instantiated to the *owner* of i. Therefore, j will have type $\langle q_0|q_0\rangle$ from the point of view of this.

As in UT, viewpoint adaptation is applied only when the receiver variable is not this. When the receiver is this, there is no need to adapt, as the object remains in the same context, the context of this.

For example, consider a field read x = y.f. Let y have type $\langle$rep|rep$\rangle$ and let field f have type $\langle$own|p$\rangle$. Then y.f has type $\langle$rep|rep$\rangle$. The first rep in this type

can be explained as follows: Owner own in the type of f gives us that the owner of the f object is the same as the owner of the y object, and owner rep in the type of y gives us that the owner of the y object is the current object. Thus, the owner of the f object, from the point of view of the current object, is the current object.

In OT, all $\mathcal{B}$ sets are empty as the system does not impose additional constraints beyond the standard subtyping and viewpoint adaptation constraints. Note that the subtyping constraints degenerate into equality constraints as OT does not have a subtyping hierarchy.

Fig. 3 (right) shows the XStack program annotated with Ownership Types. The XStack object s is $\langle$rep|rep$\rangle$ meaning that the owner of s is *root* and the ownership parameter passed to s is *root* as well. The Link object l is $\langle$rep|p$\rangle$ meaning that the enclosing XStack object is the owner of l, and the ownership parameter of the XStack object is passed to l as an ownership parameter. Variable next (line 21) has type $\langle$own|p$\rangle$ which means that the next link and the current link have the same owner, the enclosing XStack object. data is typed $\langle$p|p$\rangle$ meaning that its owner is the ownership parameter of Link which resolves to *root*. The resulting ownership tree is shown in Fig. 4. Note that for this program UT and OT give rise to the same ownership tree. In general however, UT and OT capture different ownership structure, as we will discuss in Sect. 5.

We conclude this section with a brief discussion of why we choose to restrict OT to one ownership parameter. As an experiment, we instantiated the unified framework for ownership type systems with 2 and 3 ownership parameters. However, the complexity of annotations was so overwhelming that we could not manually verify the inferred results. We concluded that in order to use Ownership Types in practice, we must restrict the system to one ownership parameter.

3 Heuristic Ranking over Typings

Ownership type systems typically allow many different typings for a given program. The trivial typings that apply to every program (peer in Universe Types, or $\langle$p|p$\rangle$ in Ownership Types) give rise to flat ownership trees where every object is a child of *root*. These typings permit every access and modification, so they do not express the programmer's intent nor detect/prevent coding errors. These goals are better served by inferring deep ownership trees, not trivial flat trees.

This section formalizes the notion of the best typing using a *ranking over all typings*. For ownership types, the ranking is a heuristic/proxy for deep ownership trees — a higher ranked typing would likely give rise to a deeper (i.e., better) runtime ownership tree than a lower ranked typing.

We begin by defining the notion of a valid typing. Let P be a program and F be an ownership type system with universal set of qualifiers Q_F. A *typing* $T_{P,F}$ is a mapping from the variables[1] in P to the type qualifiers in Q_F. A typing $T_{P,F}$ is a *valid typing for P in F* when it renders P well-typed in F. Note that a valid

[1] For the rest of the paper we use "variables" to denote all annotatable types, that is, local variable, parameter, return, allocation site, and field types.

typing $T_{P,F}$ must maintain programmer-provided annotations in P, that is, if a variable v is annotated by the programmer with q, then for every valid typing $T_{P,F}$, we have $T_{P,F}(v) = q$.

We proceed to define an objective function o that can be used to rank valid typings, and instantiations for UT and OT. The objective function o takes a valid typing T and returns a tuple of numbers[2]. The tuples are ordered lexicographically.

To create the tuple, the objective function o assumes that the qualifiers are partitioned and the partitions are ordered. Then, each element of the tuple is the number of variables in T whose type is in the corresponding partition.

3.1 Objective Function for Universe Types

For UT, the function is instantiated as

$$o_{UT}(T) = (|T^{-1}(\mathsf{any})|, |T^{-1}(\mathsf{rep})|, |T^{-1}(\mathsf{peer})|)$$

The partitioning and ordering is

$$\{\mathsf{any}\} > \{\mathsf{rep}\} > \{\mathsf{peer}\}$$

Each qualifier falls in its own partition. This means, informally, that we prefer any over rep and peer, and rep over peer. More formally, the partitioning and ordering gives rise to a preference ranking O_{UT} over all qualifiers:

$$O_{UT} : \mathsf{any} > \mathsf{rep} > \mathsf{peer}$$

Note that this preference ranking is not related to subtyping. We have $T_1 > T_2$ iff T_1 has a larger number of variables typed any than T_2, or T_1 and T_2 have the same number of any variables, but T_1 has a larger number of rep variables than T_2. Function o_{UT} gives a natural ranking over the set of valid typings for UT. In fact, the maximal (i.e., best) typing according to the above ranking, *maximizes the number of allocation sites typed* rep, which is a good proxy for a deep UT ownership tree.

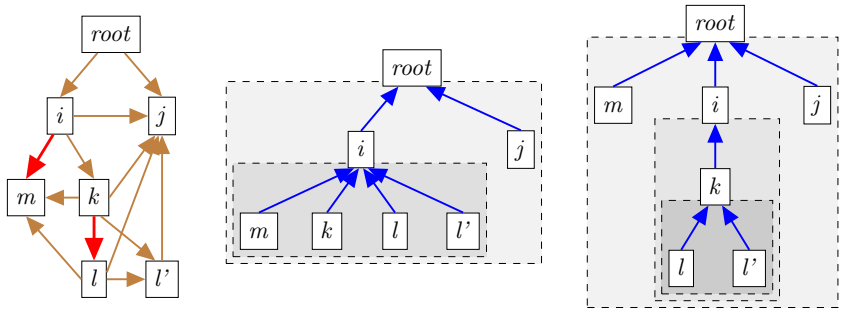
It is interesting to note that an o with exactly one qualifier per partition gives a meaningful heuristic ranking for other type systems, most notably reference immutability [13, 25] and AJ [26].

3.2 Objective Function for Ownership Types

OT cannot use an objective function with one qualifier per partition. Informally, the base modifiers are preference-ranked as

$$\mathsf{rep} > \mathsf{own} > \mathsf{p}$$

[2] Strictly, o and T are defined in terms of a specific type system F and program P; for brevity, we omit the subscripts when they are clear from context.



(a) Object graph (b) OT tree for T_1 (c) OT tree for T_2

Fig. 5. Ownership trees resulting from typings T_1 and T_2. Edges $i \to m$ and $k \to l$ (shown in red in the object graph) cannot be typed with owner rep simultaneously.

but, say, $\langle \mathsf{rep}|\mathsf{rep}\rangle$ should not carry more weight than $\langle \mathsf{rep}|\mathsf{p}\rangle$. The objective function should maximize the number of rep owners *regardless of ownership parameters*.

To illustrate this point, suppose that qualifiers $\langle q_0|q_1\rangle$ were ordered lexicographically based on the ranking of base modifiers, and consider Fig. 5. A variable roughly corresponds to an edge in the object graph [14], and therefore, we use typing of edges instead of typing of variables. Edges $i \to m$ and $k \to l$ cannot be typed with owner rep simultaneously, because of the restriction to one ownership parameter. Thus, one valid typing, call it T_1, types $root \to i$, $root \to j$ and $i \to m$ as $\langle \mathsf{rep}|\mathsf{rep}\rangle$, $i \to k$ as $\langle \mathsf{rep}|\mathsf{own}\rangle$, and the rest of the edges as either $\langle \mathsf{own}|\_\rangle$ or $\langle \mathsf{p}|\mathsf{p}\rangle$. T_1 gives rise to the ownership tree in Fig. 5(b); T_1 flattens the tree at l and l' — the owner of l and l' is i, even though k dominates both l and l' and we would like to have k as the owner of l and l'. Another valid typing, call it T_2, types $root \to i$, $root \to j$ as $\langle \mathsf{rep}|\mathsf{rep}\rangle$, $i \to k$ as $\langle \mathsf{rep}|\mathsf{own}\rangle$, $k \to l$ and $k \to l'$ as $\langle \mathsf{rep}|\mathsf{p}\rangle$, and the rest of the edges as either $\langle \mathsf{own}|\_\rangle$ or $\langle \mathsf{p}|\mathsf{p}\rangle$. T_2 gives rise to the tree in Fig. 5(c); this tree is better than the tree in Fig. 5(b) because it has more dominance. Note that lexicographical ordering ranks T_1 higher than T_2 because it contains 3 $\langle \mathsf{rep}|\mathsf{rep}\rangle$ typings, while T_2 contains only 2 $\langle \mathsf{rep}|\mathsf{rep}\rangle$ typings. However, T_2 is the better typing, because it contains 5 $\langle \mathsf{rep}|\_\rangle$ typings, one more than T_1, and therefore, it preserves more dominance in the ownership tree than T_1.

In OT, valid typings are ranked using the following o:

$$o_{OT}(T) = (|T^{-1}(\langle \mathsf{rep}|\_\rangle)|, |T^{-1}(\langle \mathsf{own}|\_\rangle)|, |T^{-1}(\langle \mathsf{p}|\_\rangle)|)$$

Here $T^{-1}(\langle \mathsf{rep}|\_\rangle)$ is the set of variables typed with owner rep, i.e., typed $\langle \mathsf{rep}|\mathsf{rep}\rangle$, $\langle \mathsf{rep}|\mathsf{own}\rangle$ or $\langle \mathsf{rep}|\mathsf{p}\rangle$. $T^{-1}(\langle \mathsf{own}|\_\rangle)$ is the set of variables typed with owner own, and $T^{-1}(\langle \mathsf{p}|\_\rangle)$ is the set of variables typed with owner p. The primary goal is to maximize the number of variables typed with owner rep (regardless of ownership parameters). Thus, the ranking *maximizes the number of edges in the object graph that are typed* rep, or in other words, the best typing preserves the most dominance (ownership). This is a good proxy for a deep OT ownership tree.

Our type inference approach (Sect. 4) requires that all qualifiers are preference-ranked and the ranking over qualifiers preserves partition ranking. Unlike o_{UT}, o_{OT} does not give rise to such ranking (e.g., $\langle \mathsf{rep}|\mathsf{rep}\rangle$ and $\langle \mathsf{rep}|\mathsf{p}\rangle$ are equally preferred by o_{OT}). We use lexicographical order over the base modifiers:

$$O_{OT} : \langle \mathsf{rep}|\mathsf{rep}\rangle > \langle \mathsf{rep}|\mathsf{own}\rangle > \langle \mathsf{rep}|\mathsf{p}\rangle > \langle \mathsf{own}|\mathsf{own}\rangle > \langle \mathsf{own}|\mathsf{p}\rangle > \langle \mathsf{p}|\mathsf{p}\rangle$$

O_{OT} preserves the partition ranking (e.g., $\langle \mathsf{rep}|\mathsf{p}\rangle > \langle \mathsf{own}|\mathsf{own}\rangle$) and preference-ranks qualifiers within partitions (e.g., $\langle \mathsf{rep}|\mathsf{rep}\rangle > \langle \mathsf{rep}|\mathsf{own}\rangle > \langle \mathsf{rep}|\mathsf{p}\rangle$).

3.3 Maximal Typing

A *maximal typing* is a typing that maximizes o (i.e., the best typing(s) according to the heuristics encoded in o).

Definition 1. Maximal Typing. Given an objective function o over the set of valid typings, a valid typing T is a *maximal typing of P in F* under o, if for every valid typing T', we have $T' \neq T \Rightarrow T \geq T'$.

Perhaps somewhat unexpectedly, for UT, as well as other interesting systems such as reference immutability [13], there exists a *unique maximal typing*. This is discussed in detail in the next section. For OT however, in general, there are multiple maximal typings, i.e., there are multiple typings that maximize o_{OT}. Consider the following program:

```
1  x = new X(); [x]
2  y = new Y(); [y]
3  x.f = y;
```

There are variables x, y, field f, and allocation sites x and y. Typing T_1 types the program as follows: $T_1(\mathsf{x}) = T_1(x) = \langle \mathsf{rep}|\mathsf{own}\rangle$, $T_1(\mathsf{y}) = T_1(y) = \langle \mathsf{rep}|\mathsf{own}\rangle$, and $T_1(\mathsf{f}) = \langle \mathsf{own}|\mathsf{p}\rangle$. Typing T_2 types the program as follows:. $T_2(\mathsf{x}) = T_2(x) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, $T_2(\mathsf{y}) = T_2(y) = \langle \mathsf{rep}|\mathsf{rep}\rangle$, and $T_2(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle$. Clearly, $o_{OT}(T_1) = o_{OT}(T_2) = (4, 1, 0)$. There are other valid typings that maximize o_{OT} as well. There are nontrivial examples as well.

The following section describes a unified type inference approach, which can be used to compute the unique maximal typing for UT, and a maximal typing for OT given user annotations.

4 Unified Type Inference

The unified inference and checking system works on completely unannotated programs, as well as on partially-annotated programs. We believe that neither fully automatic inference nor fully manually annotated programs are feasible choices. In many interesting systems, fully automatic inference is impossible; that is, the programmer must provide initial annotations which typically reflect semantics that is impossible to infer. We envision a cooperative system that fills

in as many annotations as possible and queries the programmer for a small set of annotations on certain variables to resolve ambiguities. The system seamlessly integrates programmer-provided annotations with inferred annotations.

The key idea in our system is to compute a set-based solution S instead of a single typing. S maps variables to *sets* of qualifiers: for every statement s, for every variable v in s, and for every qualifier $q \in S(v)$, there are qualifiers in the sets of the remaining variables in s, such that q and those qualifiers make statement s type check. Interestingly, for some systems such as UT, the set-based solution S, which is inexpensive to compute, implies the unique maximal typing. For other systems, such as OT, where the unique maximal typing does not exist, S pinpoints the places where programmer-provided annotations must be added, and as a result, reduces the number of manual annotations significantly.

Sect. 4.1 describes the computation of the set-based solution S and Sect. 4.2 describes its properties. Then, Sects. 4.3 and 4.4 describe how the type inference is instantiated for the two ownership type systems in our study.

4.1 Set-based Solution

Set Mapping S maps each program variable (annotatable reference) to a *set* of possible type qualifiers. We fix the program P and type system F, and we write S instead of $S_{P,F}$ for brevity.

The initial mapping, S_0, is defined as follows. Programmer-annotated variables are initialized to the singleton set which contains only the programmer-provided annotation. Variables that are not annotated are initialized to the *maximal set* of qualifiers Q_F. The analysis, a fixpoint iteration, iterates over the statements in the program and refines the initial sets, until it reaches the fixpoint.

Transfer Functions We now describe the transfer functions applied by fixpoint iteration. There is a transfer function f_s for each statement s. Statements s can be of kinds as shown in Fig. 2. Each f_s takes as input mapping S and outputs an updated mapping S'. Informally, f_s removes all infeasible qualifiers from the sets of the variables $v \in s$. After the application of f_s, for each variable $v_i \in s$ and each $q_i \in S'(v_i)$, there exist $q_1 \in S'(v_1), \ldots, q_{i-1} \in S'(v_{i-1}), q_{i+i} \in S'(v_{i+1}), \ldots, q_k \in S'(v_k)$, such that $q_1, \ldots, q_k$ type check with the rule for s in Fig. 2. The transfer functions are defined in terms of the typing rules in Fig. 2; making s type check requires that the subtyping, viewpoint adaptation, and $\mathcal{B}$ constraints for s hold.

More formally $f_s : S \to S'$ is defined as follows:

$$
\begin{array}{l}
\text{foreach } v_i \in s \\
\quad S'(v_i) = \{\, q_i \mid q_i \in S(v_i) \text{ and} \\
\qquad \exists q_1 \in S(v_1), \ldots, q_{i-1} \in S(v_{i-1}), q_{i+1} \in S(v_{i+1}), \ldots, q_k \in S(v_k) \\
\qquad\quad \text{s.t. } q_1, \ldots, q_k \text{ type check with the rule for } s \text{ in Fig. 2} \,\}
\end{array}
$$

For example, the transfer function $f_{\mathsf{x=y}} : S \to S'$ for UT is as follows:

$$
\begin{array}{l}
S'(\mathsf{x}) = \{\, q \mid q \in S(\mathsf{x}) \text{ and } \exists q_{\mathsf{y}} \in S(\mathsf{y}) \text{ s.t. } q_{\mathsf{y}} <: q \,\} \\
S'(\mathsf{y}) = \{\, q \mid q \in S(\mathsf{y}) \text{ and } \exists q_{\mathsf{x}} \in S(\mathsf{x}) \text{ s.t. } q <: q_{\mathsf{x}} \,\}
\end{array}
$$

Suppose that we apply transfer function $f_{\mathsf{x=y}}$ for UT on S, where $S(\mathsf{x}) = \{\mathsf{rep}, \mathsf{peer}\}$ and $S(\mathsf{y}) = \{\mathsf{any}, \mathsf{peer}\}$. $f_{\mathsf{x=y}}$ removes rep from $S(\mathsf{x})$ because there does not exist $q_{\mathsf{y}} \in S(\mathsf{y})$ that will make the type constraint for (TASSIGN), namely $q_{\mathsf{y}} <: \mathsf{rep}$, hold. Next, it removes any from $S(\mathsf{y})$ because $\mathsf{any} <: \mathsf{peer}$ does not hold. After the application of the transfer function, $S'(\mathsf{x}) = \{\mathsf{peer}\}$ and $S'(\mathsf{y}) = \{\mathsf{peer}\}$.

As another example, consider $f_{\mathsf{x.f=y}}$ for OT applied on S, where $S(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$, the set for field f, $S(\mathsf{f}) = \{\langle \mathsf{rep}|\mathsf{rep}\rangle, \langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{rep}|\mathsf{p}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ and $S(\mathsf{y}) = \{\langle \mathsf{own}|\mathsf{own}\rangle\}$. $\langle \mathsf{rep}|\mathsf{rep}\rangle$ is removed from $S(\mathsf{x})$ because there does not exist $q \in S(\mathsf{f})$ such that the type constraint for (TWRITE), namely $\langle \mathsf{rep}|\mathsf{rep}\rangle \rhd q = \langle \mathsf{own}|\mathsf{own}\rangle$, holds. $\langle \mathsf{rep}|\mathsf{p}\rangle$ and $\langle \mathsf{p}|\mathsf{p}\rangle$ are removed as well, and $S'(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\}$. Similarly, $\langle \mathsf{rep}|\mathsf{rep}\rangle$, $\langle \mathsf{rep}|\mathsf{own}\rangle$, and $\langle \mathsf{rep}|\mathsf{p}\rangle$ are removed from $S(\mathsf{f})$ (recall that viewpoint adaptation for OT disallows exposed fields from being rep). Thus, $S'(\mathsf{f}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ and $S'(\mathsf{y})$ remains the same.

Fixpoint Iteration The analysis is a fixpoint iteration. It initializes the mapping S_0 as described earlier in this section, and keeps iterating over the program statements, using the above transfer functions, until one of the following happens: (1) S reaches the fixpoint, i.e., S remains unchanged from the previous iteration, in which case the analysis terminates successfully, or (2) a key is assigned the empty set, in which case the analysis terminates indicating the program is untypable.

The computation fits the requirements of a monotone framework [19]. The property space is the standard lattice of subsets, with the set of qualifiers Q_F being the bottom 0, and the empty set $\emptyset$ being the top 1 of the lattice. The transfer functions are monotone. Therefore, the set-based solution S produced by fixpoint iteration is the unique least solution (for historical reasons sometimes this solution is referred to as the "maximal fixpoint solution" [19]).

4.2 Properties of the Set-based Solution

Let us now consider the properties of the set-based solution S. These properties help establish that for certain type systems one can derive a maximal (i.e., best) typing from the set-based solution S.

The first proposition states that if the algorithm removes a qualifier q from the set $S(v)$ for variable v, then there does not exist a valid typing that maps v to q. The notation $T \in S_0$ denotes that for every variable v we have $T(v) \in S_0(v)$.

Proposition 1. *Let S be the set-based solution. Let v be any variable in P and let q be any qualifier in F. If $q \notin S(v)$ then there does not exist a valid typing $T \in S_0$, such that $T(v) = q$.*

Proof. (Sketch) We say that q is a valid qualifier for v if there exists a valid typing T, where $T(v) = q$. Let v be the *first* variable that has a valid qualifier q removed from its set $S(v)$ and let f_s be the transfer function that performs the removal. Since q is a valid qualifier there exist valid qualifiers $q_1, ..., q_k$ that make s type check. If $q_1 \in S(v_1)$ and $q_2 \in S(v_2)$, ..., and $q_k \in S(v_k)$, then by

definition, f_s would not have had q removed from $S(v)$. Thus, one of $v_1, \ldots, v_k$ must have had a valid qualifier removed from its set *before* the application of f_s. This contradicts the assumption that v is the first variable that has a valid qualifier removed.

The second proposition states that if we map every variable v to the maximal qualifier in its set $S(v)$ according to its preference ranking over qualifiers[3], and the typing is valid, then this typing maximizes the objective function.

Proposition 2. *Let o be the objective function over valid typings, and S be the set-based solution. The* maximal typing *T is the following: $T(v) = max(S(v))$ for every variable v in P. If T is a valid typing, then T is a maximal typing of P in F under o.*

Proof. (Sketch) We show that T is a maximal typing. Suppose that there exists a valid typing $T' > T$. Let p_i be the most-preferred partition such that $T'^{-1}(p_i) \neq T^{-1}(p_i)$. Since $T' > T$, there must exist a variable v such that $T'(v) = q' \in p_i$, but $T(v) = q \notin p_i$. In other words, T' types v with $T'(v) = q' \in p_i$, but T types v differently — and lesser in the preference ranking, because $T'^{-1}(p_k) = T^{-1}(p_k)$ for $0 \leq k < i$ (here p_k are the more-preferred partitions than p_i). Since $T(v) = max(S(v))$, it follows that $q' \notin S(v)$. By Proposition 1, if $q' \notin S(v)$ there does not exist a valid typing which maps v to q', which contradicts the assumption that T' is a valid typing.

When each partition in the preference ranking has only a single element, then the weaker assumption "there exists a valid typing $T' \geq T$" can be contradicted, showing that the maximal typing is unique.

The *optimality property* holds for a type system F and a program P if and only if the typing derived from the set-based solution S by typing each variable with the maximally/preferred qualifier from its set, is a valid typing.

Property 1. Optimality Property. Let F be a type system augmented with objective function o and let P be a program. The optimality property holds for F and P iff $T(v) = max(S(v))$, for all variables v, is a valid typing.

The set-based solution is computed in $O(n^2)$ time where n is the size of the program. At each iteration through the program, at least one of the $O(n)$ variables changes its set to a smaller set. Therefore, there are at most $O(|Q_F| * n)$ iterations. At each iteration, the computation goes through $O(n)$ statements. Since $|Q_F|$ is a small constant (3 in UT, 6 in OT), it follows that the complexity is $O(n^2)$. Therefore, for type systems for which the optimality property holds for arbitrarily annotated programs, a maximal typing can be computed in quadratic time, with no manual annotations. If the programmer provides inconsistent initial annotations in P, the computation would terminate within $O(n^2)$ time with a message that there is no valid typing for P.

Remarkably, for several interesting systems (UT, AJ, reference immutability), the optimality property holds for unannotated programs, which means that the

[3] Rankings O_{UT} and O_{OT} ensure that the maximal qualifier is uniquely defined.

Variable	Initial	Iteration 1	Iteration 2
top	all	all	all
d1	all	any, peer	any, peer
newTop	all	rep, peer	rep, peer
new Link()	all	rep, peer	rep, peer
s	all	rep, peer	rep, peer
new XStack()	all	rep, peer	rep, peer
x	all	all	all
new X()	all	rep, peer	rep, peer
next	all	any, peer	any, peer
data	all	any, peer	any, peer
d2	all	any, peer	any, peer

Fig. 6. Inference of Universe Types for the example in Fig. 3.

unique maximal typing can be computed in $O(n^2)$ time with no manual annotations. However, for OT, the property does not hold for unannotated programs. We will discuss each ownership system in turn in the next two subsections.

4.3 Inference of Universe Types

For Universe Types, the preference ranking over all qualifiers is O_{UT} (previously defined in Sect. 3). Libraries receive default type {peer}.

Fig. 6 illustrates the computation of the set-based solution for UT for the example in Fig. 3. Consider statement s.push(x) at line 17. Initially, $S(\mathsf{s}) = S(\mathsf{x}) = S(\mathsf{d1}) = \{\mathsf{any}, \mathsf{rep}, \mathsf{peer}\}$. In iteration 1, the transfer function for s.push(x) removes any from $S(\mathsf{s})$ because push is impure. It also removes rep from $S(\mathsf{d1})$ because $q \triangleright \mathsf{rep} = \mathsf{lost}$ which the type rule for (TCALL) forbids. See Fig. 6. Choosing the maximal type from each set gives us $T(\mathsf{s}) = \mathsf{rep}$, $T(\mathsf{x}) = \mathsf{any}$, and $T(\mathsf{d1}) = \mathsf{any}$, which type checks with the rule for (TCALL).

We show through case analysis that for each statement s, after the application of the transfer function for s, s type checks with the maximal typing:

(TASSIGN) Consider x = y. We must show that after the application of $f_{\mathsf{x=y}}$, x = y type checks with $max(S'(\mathsf{x}))$ and $max(S'(\mathsf{y}))$.

- If $max(S'(\mathsf{x})) = \mathsf{any}$ the statement type checks with any value for $max(S'(\mathsf{y}))$.
- Suppose that $max(S'(\mathsf{x})) = \mathsf{rep}$. Thus, any is not in $S'(\mathsf{x})$, and therefore any cannot be in $S'(\mathsf{y})$. $max(S'(\mathsf{y}))$ cannot be peer; this contradicts the assumption that $max(S'(\mathsf{x})) = \mathsf{rep}$ (rep would have been removed from x's set). Thus, $max(S'(\mathsf{y})) = \mathsf{rep}$ and x = y type checks.
- Suppose now that $max(S'(\mathsf{x})) = \mathsf{peer}$. The only possible value for $max(S'(\mathsf{y}))$ is peer and the statement again type checks.

(TNEW) is shown exactly the same way.

(TREAD) Consider x = y.f. We must show that after the application of the transfer function $f_{\mathsf{x=y.f}}$, the statement will type check with $max(S'(\mathsf{x}))$, $max(S'(\mathsf{f}))$ and $max(S'(\mathsf{y}))$.

Variable	Initial	Iteration 1	Iteration 2	Iteration 3
`top`	all	all	⟨rep|p⟩	⟨rep|p⟩
`d1`	all	⟨p|p⟩	⟨p|p⟩	⟨p|p⟩
`newTop`	all	⟨rep|p⟩	⟨rep|p⟩	⟨rep|p⟩
`new Link()`	⟨rep|p⟩	⟨rep|p⟩	⟨rep|p⟩	⟨rep|p⟩
`s`	all	all	all	all
`new XStack()`	all	all	all	all
`x`	all	all	all	all
`new X()`	all	all	all	all
`next`	all	⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩	⟨own|p⟩	⟨own|p⟩
`data`	all	⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩	⟨p|p⟩	⟨p|p⟩
`d2`	all	⟨own|own⟩, ⟨own|p⟩, ⟨p|p⟩	⟨p|p⟩	⟨p|p⟩

Fig. 7. Inference of Ownership Types for the example in Fig. 3.

- If $max(S'(\mathsf{x})) = \mathsf{any}$ this is clearly true.
- Suppose that $max(S'(\mathsf{x})) = \mathsf{rep}$; then $max(S'(\mathsf{f}))$ must be peer and $max(S'(\mathsf{y}))$ must be rep.
- Finally, when $max(S'(\mathsf{x})) = \mathsf{peer}$, one can easily see that $max(S'(\mathsf{f}))$ must be peer and $max(S'(\mathsf{y}))$ must be peer as well.

(TWRITE) and (TCALL) are analogous; they are omitted for brevity. We implemented an independent type checker which verifies the inferred solution.

4.4 Inference of Ownership Types

For Ownership Types, the preference ranking over all qualifiers is O_{UT} (see Sect. 3). Library variables receive default $\{\langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ as explained in [12].

Fig. 7 shows the computation of the set-based solution for the example program in Fig. 3. Note that this computation assumes annotation ⟨rep|p⟩ at allocation site new Link(); given this annotation, the optimality property holds, and the set-based solution computes the maximal typing for the program.

As mentioned earlier, the optimality property does not always hold in OT. As an example, consider the program:

```
1  x = new A();
2  y = new ⟨own|own⟩ C();
3  x.f = y;
```

The application of transfer functions yields $S(\mathsf{x}) = \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\}$, $S(\mathsf{f}) = \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\}$ and $S(\mathsf{y}) = \{\langle \mathsf{own}|\mathsf{own}\rangle\}$. If we map every variable to the maximal qualifier we have

$$T(\mathsf{x}) = \langle \mathsf{rep}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

which fails to type check because $\langle \mathsf{rep}|\mathsf{own}\rangle \triangleright \langle \mathsf{own}|\mathsf{own}\rangle$ equals $\langle \mathsf{rep}|\mathsf{rep}\rangle$, not $\langle \mathsf{own}|\mathsf{own}\rangle$. The set-based solution contains several valid typings. If we chose the maximal value at x, we will have typing

$$T(\mathsf{x}) = \langle \mathsf{rep}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{p}|\mathsf{p}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

and if we chose the maximal value at f, we will have

$$T(\mathsf{x}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

The set-based solution is valuable for two reasons. First, it restricts the search space significantly. Initially, there are 6 possibilities for each variable and there are n variables, leading to 6^n potential typings. Second, the set-based solution highlights the points of non-determinism where programmer-provided annotations can guide the inference to choose one typing over another. With a small number of programmer-provided annotations, OT inference can scale up to large programs. We explain the process in the remainder of this section.

The points of non-determinism arise at field access and method call statements due to viewpoint adaptation. A statement s is a *conflict* if it does not type check with the maximal assignment derived from the set-based solution. In the example above, statement $\mathsf{x.f} = \mathsf{y}$ is a conflict, because if we map every variable to the maximal qualifier, the statement fails to type check. Our approach performs the following incremental process. Given a program P, which may be unannotated or partially annotated, the tool runs the set-based solver, and if there are conflicts, these conflicts are printed. The programmer selects a subset of conflicts (usually the first 1 to 5), and for each conflict, annotates variables. Then the programmer runs the set-based solver again. This process continues until a program P' is reached, where the optimality property holds for P'. The solver computes a maximal typing for P'.

In the above example, the solver prints conflict $\mathsf{x.f} = \mathsf{y}$ and the set-based solution

$$\begin{aligned} S(\mathsf{x}) &= \{\langle \mathsf{rep}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle\} \\ S(\mathsf{f}) &= \{\langle \mathsf{own}|\mathsf{own}\rangle, \langle \mathsf{own}|\mathsf{p}\rangle, \langle \mathsf{p}|\mathsf{p}\rangle\} \\ S(\mathsf{y}) &= \{\langle \mathsf{own}|\mathsf{own}\rangle\} \end{aligned}$$

If the programmer chooses to annotate x with $\langle \mathsf{rep}|\mathsf{own}\rangle$, this results in typing

$$T(\mathsf{x}) = \langle \mathsf{rep}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{p}|\mathsf{p}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

and if he/she chooses to annotate f with $\langle \mathsf{own}|\mathsf{own}\rangle$ this results in typing

$$T(\mathsf{x}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{f}) = \langle \mathsf{own}|\mathsf{own}\rangle,\ T(\mathsf{y}) = \langle \mathsf{own}|\mathsf{own}\rangle$$

5 Empirical Results

5.1 Implementation

Our inference tool is built on top of the Checker Framework [20, 6]. The tool extends the Checker Framework to specify type system constraints and preference ranking over qualifiers; it generates the constraints for the type systems by traversing the AST and it implements the set-based constraint solver described in Sect. 4. The tool is freely available at http://www.cs.rpi.edu/~huangw5/cf-inference/.

The constraint solver takes as input a number of constraints and it iteratively refines the sets of valid type qualifiers until it reaches a fixpoint. If conflicts (as defined in Sect. 4.4) occur in the solution, the solver prints all conflicts and prompts the user to solve the conflicts by providing manual annotations.

5.2 Results

Benchmarks We evaluated our implementation using eight Java programs of up to 110kLOC (see Fig. 8). The analysis processes only application code; libraries are handled using the defaults specified in Sect. 4.3 and Sect. 4.4. The analysis is modular, in the sense that it can analyze whatever code is available, including libraries with no main method.

All evaluations were conducted on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (all benchmarks run within a memory footprint of 1GB). The software environment consists of JDK 1.6 and GNU/Linux 2.6.38.

Benchmark	#Lines	#Meths	Description
JOlden	6223	326	Benchmark suit of 10 small programs
tinySQL	31980	1597	Database engine
htmlparser	62627	1698	HTML parser
ejc	110822	4734	Compiler of the Eclipse IDE
javad	4207	140	Java class file disassembler
SPECjbb	12076	529	SPEC's benchmark for evaluating server side Java
jdepend	4351	328	Java package dependency analyzer
classycle	8972	440	Java class and package dependency analyzer

Fig. 8. The benchmark programs used in our evaluation.

Universe Types Inference of Universe Types requires information about method side effects. As stated earlier, we used our purity inference tool [13]. The purity inference relies on a type system for reference immutability, which itself instantiates our unified framework. The optimality property holds for unannotated programs for UT, and the set-based solver infers the unique maximal typing.

Fig. 9 shows the inference results for Universe Types. Across all benchmarks, 9%–33% of all variables are inferred as any, the best qualifier. 1% to 10% of all variables are inferred as rep. A relatively large percentage (57%–92%) of the variables are inferred as peer, resulting in a flat ownership structure. This is consistent with previous results [7]. There are several possible reasons that lead to flat ownership structures. One is due to utility methods whose formal parameters are passed to impure methods. This forces the formal parameters to be peer. Another reason is that the inference uses the default peer annotation for libraries.

Compared to previous results [7], our inference reports a larger percentage of any variables. One reason is that there are more pure methods in our inference than in [7]. In our inference, pure methods are inferred automatically while in [7] pure methods are annotated manually. For example in javad, 40 methods

Benchmark	#Pure	#Ref	#any	#rep	#peer	#Manual	Time
JOlden	175	685	227 (33%)	71 (10%)	387 (56%)	0	11.3
tinySQL	965	2711	630 (23%)	104 (4%)	1977 (73%)	0	18.2
htmlparser	642	3269	426 (13%)	153 (5%)	2690 (82%)	0	22.9
ejc	1701	10957	1897 (17%)	122 (1%)	8938 (82%)	0	119.7
javad	60	249	31 (12%)	11 (4%)	207 (83%)	0	4.1
SPECjbb	195	1066	295 (28%)	74 (7%)	697 (65%)	0	13.6
jdepend	102	542	95 (18%)	14 (3%)	433 (80%)	0	7.2
classycle	260	946	87 (9%)	11 (1%)	848 (90%)	0	9.9

Fig. 9. The inference results for Universe Types. Column #Ref gives the total number of references excluding implicit parameters this. Column #Pure gives the number of pure methods inferred automatically based on reference immutability [13]. Columns #any, #rep, and #peer give the number of references inferred as any, rep, and peer, respectively. No user annotations are needed for the inference of Universe Types; therefore, there are only zeros in the #Manual column. Last column Time shows the total running time in seconds including parsing the source code, type inference, and type checking.

were manually annotated as pure in [7] while 60 were inferred automatically in our inference; we verified that the extra 20 methods were indeed pure. Another reason is that our qualifier ranking always prefers any over rep. When a variable is mapped to set {any, rep} in the set-based solution, our tool picks any instead of rep. This happens for variable x in Fig. 6. Although x is assigned by a rep allocation site, the tool still infers x as any because x is readonly in the main method. In contrast, Dietl et al. [7] use a different heuristic which uses program location to preference-rank qualifiers. They choose rep over any in certain cases, which results in a larger percentage of variables reported as rep. It is important to note that the larger percentage of any variables *does not imply a flatter ownership tree* compared to [7]; this is because an any variable can refer to a rep object as is the case with variable x. What matters for ownership structure are the allocation sites, and as we shall see shortly, the inference reports a considerably larger percentage of reps for allocation sites compared to reference variables.

Ownership Types In OT, we add an additional modifier norep, which refers to *root*, as described in detail in [12]. We use norep as the default type for String and boxed primitives such as Boolean, Integer, etc.

Fig. 10 shows the inference results for OT. Note that there are many ⟨norep|_⟩ variables; the majority of these are strings and boxed primitives, e.g. 521 out of 688 ⟨norep|norep⟩ variables in SPECjbb are strings and boxed primitives whose default type is norep.

Compared to UT, a relatively large percentage (4%–24%) of variables are inferred as ⟨rep|_⟩ in OT. Note however, that this *does not imply a deeper ownership tree compared to UT*. In UT, many of the any variables can refer to a rep object (as UT distinguishes readonly access); in contrast, in OT only a rep variable can refer to a rep object. Due to the fact that the optimality property does not hold for OT, as discussed in Sect. 4.4, the inference requires manual annotations. Column #Manual gives the total numbers of manual annotations

Benchmark	#Ref	#⟨rep|_⟩	#⟨own|_⟩	#⟨p|_⟩	#⟨norep|_⟩	#Manual	Time
JOlden	685	67 (10%/**10%**)	497 (73%)	24 (4%)	97 (14%)	13 (2)	10.3
tinySQL	2711	224 (8%/**11%**)	530 (20%)	5 (0%)	1952 (72%)	215 (7)	18.4
htmlparser	3269	330 (10%/**11%**)	629 (19%)	36 (1%)	2274 (70%)	200 (3)	33.6
ejc	10957	467 (4%/ **4%**)	1768 (16%)	50 (0%)	8672 (79%)	592 (5)	122.4
javad	249	44 (18%/**19%**)	27 (11%)	74 (30%)	104 (42%)	46 (10)	5.5
SPECjbb	1066	166 (16%/**16%**)	141 (13%)	71 (7%)	688 (65%)	73 (6)	17.1
jdepend	542	130 (24%/**25%**)	156 (29%)	128 (24%)	128 (24%)	26 (6)	13.7
classycle	946	153 (16%/**20%**)	173 (18%)	28 (3%)	592 (63%)	90 (10)	11.7

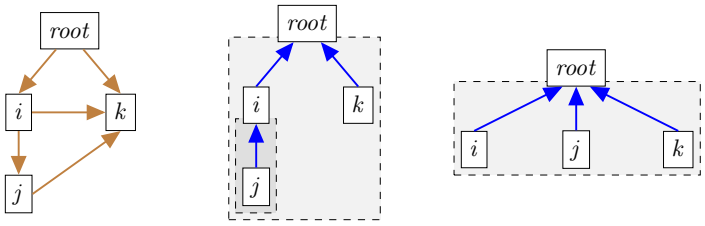
Fig. 10. The inference results for Ownership Types. Column #Ref again gives the total number of references excluding the implicit parameters this. Columns #⟨rep|_⟩, #⟨own|_⟩, #⟨p|_⟩, and #⟨norep|_⟩ give the numbers of variables whose owners are inferred as rep, own, p, and norep, respectively. The boldfaced number in parentheses in column #⟨rep|_⟩ is an upper bound on rep typings; it is discussed in the text. #Manual shows the total number of manual annotations and, in parentheses, the number of annotations per 1kLOC. Time shows the running time in seconds.

that were added and, in parentheses, the number of annotations per 1kLOC. The annotation burden is low — on average, 6 annotations per 1kLOC. Although the set-based solver cannot produce a maximal typing automatically, it is quite valuable, because it reduces the burden of annotations on programmers. The set-based solver prints all conflicts and lets the programmer choose an annotation that resolves the conflict in such a way that it reflects their intent. This process continues until all conflicts are resolved. By doing so, the first author annotated JOlden (6223 LOC) in approximately 10 minutes and SPECjbb in approximately 2 hours. The annotations reflect the intent of the first author, but not necessary the intent of the programmers of these benchmarks. Finally, the last column Time shows the time in seconds to do type inference and type checking *after* the manual annotations. It is approximately equal to the initial run that outputs all conflicts and does not include the time to annotate the benchmark.

The boldfaced percentage shown in parentheses in column #⟨rep|_⟩, is the percentage of all references that contain a ⟨rep|rep⟩, ⟨rep|own⟩ or ⟨rep|p⟩ in their set-based solution. This is an upper bound on the possible rep typings: even an ownership type system with many ownership parameters will be unable to type a larger percentage of variables as rep. The fact that the percentage of #⟨rep|_⟩'s in our typing is close to this bound, has two implications: (1) our typing is precise (at least with respect to the heuristic defined in Sect. 3), and (2) one ownership parameter may be sufficient in practice (again, if the goal is to maximize the number of rep typings).

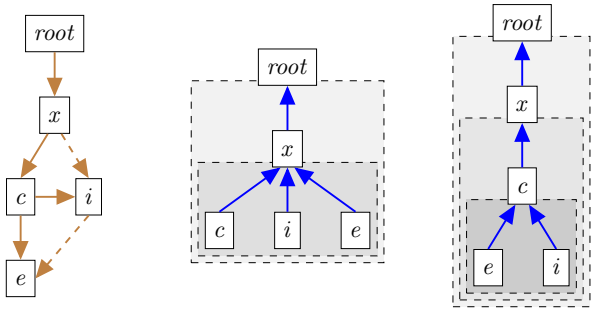
5.3 Comparing Universe Types vs. Ownership Types

In this section, we compare Universe Types, which enforce the owner-as-modifier encapsulation discipline, to Ownership Types, which enforce the owner-as-dominator encapsulation discipline, using examples we observed in the benchmarks.



(a) Object graph (b) OT tree (c) UT tree

Fig. 11. Write access to enclosing context results in flatter structure for UT as compared to OT (The bold edge from j to k highlights the write access).



(a) Object graph (b) OT tree (c) UT tree

Fig. 12. Readonly sharing of internal representation results in flatter structure for OT as compared to UT (The dotted edge from i to e highlights the readonly access).

In some cases, Universe Types inferred flatter structures than Ownership Types. This happens when an object j modifies an object k in an enclosing context. For example, consider Fig. 11. If object j *modifies* k, j and k must be peers in UT, which will force the flat ownership tree in Fig. 11(c). In contrast, OT reflects dominance and produces the deeper ownership tree shown in Fig. 11(b).

In other cases, Ownership Types inferred flatter structures than Universe Types. OT disallows exposure of internal objects outside of the boundary of the owner. UT is more permissive, in the sense that it allows readonly exposure. Consider Fig. 12. which represents a container c, its internal representation e and an iterator i over e. The OT tree is flatter because the iterator i creates a path to e which does not go through c. Therefore, c, e, and i must have x as their owner. In contrast, UT allows the exposure of i to x because this exposure is readonly. Therefore, c remains the owner of both e and i.

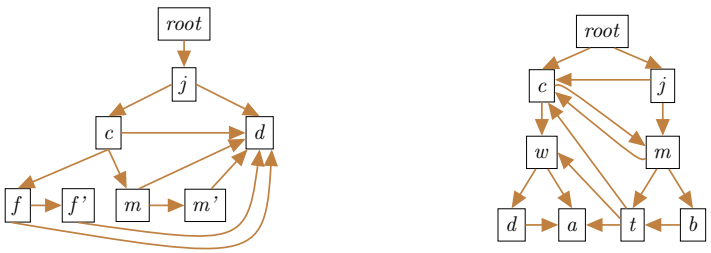
Fig. 13 compares OT and UT on the benchmarks. We consider only allocation sites, excluding strings and boxed primitives. Allocation sites provide the best approximation of ownership structure. On average 25% of the OT $\langle \mathsf{rep}|\_\rangle$ sites are typed rep in UT as well. On the other hand, on average 64% of the UT rep sites are typed $\langle \mathsf{rep}|\_\rangle$ in OT as well. The discrepancy shows that it may be more common to have write access to enclosing context (which lowers rep

Benchmark OT: / UT:	⟨rep|_⟩ / rep	⟨rep|_⟩ / peer	not ⟨rep|_⟩ / rep	not ⟨rep|_⟩ / not rep
JOlden	26 (22%)	8 (7%)	19 (16%)	66 (55%)
tinySQL	32 (6%)	123 (24%)	13 (2%)	355 (68%)
htmlparser	27 (2%)	234 (20%)	16 (1%)	926 (77%)
ejc	44 (2%)	336 (12%)	81 (3%)	2321 (83%)
javad	6 (10%)	38 (66%)	0 (0%)	14 (24%)
SPECjbb	75 (26%)	84 (29%)	25 (9%)	110 (37%)
jdepend	13 (7%)	71 (41%)	1 (1%)	90 (51%)
classycle	1 (0%)	109 (45%)	5 (2%)	128 (53%)

Fig. 13. Ownership Types vs. Universe Types on allocation sites. The four columns give the number of OT/UT pairings and, in parenthesis, the corresponding percentages. For example, column ⟨rep|_⟩/peer shows the number of allocation sites that were inferred as rep in OT and peer in UT.

to peer in UT), than it is to have readonly sharing of internal structure (which allows an object to stay rep in UT while it is not rep in OT). On average 40% of all allocation sites are inferred as rep in OT, and 14% are inferred as rep in UT, which suggests that write access to enclosing context is more common than readonly sharing of internal structure. The results suggest that in general, UT and OT capture distinct ownership structure. Note that as expected, there is a significantly larger percentage of rep allocation sites in UT compared to rep variables.

To further understand the differences between UT and OT, we examined the results of two of the benchmarks, javad and SPECjbb. Fig. 14(a) shows a partial object graph for javad. Here j represents the jvmDump object, c is the classFile object, f and f' are the fieldSection and fieldInfo objects, and m and m' are the methodSection and methodInfo objects. d is the DataStream object. All of c, f, f', m and m' *modify* d, which is an object from enclosing context. This forces all c, f, f', m, m' and d to be peers, and children of j in the UT ownership tree. Edges $c \to f, f \to f', c \to m$ and $m \to m'$ are ⟨rep|_⟩ in OT, but are peer in UT. Now consider Fig. 14(b). It shows a partial object graph for SPECjbb. c is the Company object, w is a Warehouse object, d is a District object, and a is Address object which represents the District's address. j is a JBBmain thread, m is a TransactionManager, t is a Transaction and b is an array that stores transactions.



(a) javad (b) SPECjbb

Fig. 14. Partial object graphs for the javad and SPECjbb case studies.

Edges $w \to a$ and $t \to a$ expose the Address object outside of its creating object d. Therefore, the edge $d \to a$ cannot be $\langle \mathsf{rep}|\_\rangle$ in OT. However, the exposure is readonly, and it remains rep in UT.

6 Related Work

We discuss related work on ownership inference as well as other work on inference of pluggable and extended types.

Several dynamic approaches for ownership inference exist [9, 18, 21, 27]. Although a dynamic approach may produce more precise results, it is inherently unsound and incurs a significant performance overhead. Also, it is difficult to generalize a dynamic approach to different type systems. In contrast, our approach is static and can be applied to multiple type systems.

Aldrich et al. [1] present an ownership type system and a type inference algorithm. Their inference creates equality, component and instantiation constraints and solves these constraints. Our inference solves different kinds of constraints, namely subtyping and adapt constraints.

Ma and Foster [16] propose Uno, a static analysis for automatically inferring ownership, uniqueness, and other aliasing and encapsulation properties in Java. Uno infers "stricter" ownership in which an owned object can only be accessed by its owner. Our inference has a less-restrictive ownership model. Uno's inference is based on Soot and it is difficult to map the inference results back to the source code, subsequently inhibiting type checking. Our type inference is integrated into the Checker Framework; we perform type checking as well.

Greenfieldboyce and Foster [11] present a framework called JQual for inferring user-defined type qualifiers in Java. JQual is effective for *source-sink* type systems, for which programmers need to add annotations to the sources and sinks and JQual infers the intermediate annotations for the rest of the program. Our tool handles more complex type systems such as Ownership type systems. In addition, JQual does not scale well in its field-sensitive mode as reported by Artzi et al.[2]. In contrast, our inference scales to programs of up to 110kLOC.

Chin et al. [3] propose CLARITY for the inference of user-defined qualifiers for C programs based on user-defined rules, which can also be inferred given user-defined invariants. CLARITY infers several type qualifiers, including pos and neg for integers, nonnull for pointers, and tainted and untainted for strings. These type qualifiers are not context-sensitive. Our tool focuses on type systems for Java, and it is context-sensitive (viewpoint adaptation models context sensitivity).

Dietl et al. [7] present a tunable static inference for Generic Universe Types (GUT). Constraints of GUT are encoded as a boolean satisfiability problem, which is solved by a weighted Max-SAT solver. The inference is tunable in the sense that programmers can direct the inference by setting different weights or partially annotating the source code. In contrast, our inference can only be tuned by accepting programmers' manual annotations. However, by defining a ranking over typings, we avoid the exponential SAT solver and manage to scale to larger programs. A detailed comparison is left as future work.

Milanova and Vitek [17] present a static dominance inference analysis, based on which they perform Ownership Type inference. Our current work is an improvement over [17]. First, it accepts manual annotations to direct the inference, while [17] does not. Second, it provides optimality guarantees, while the inference in [17] does not provide guarantees — in theory, it may end up with a solution which produces a flat ownership tree. Third, our work includes a type checker which is not available in [17], and it works on more and larger benchmarks.

Sergey and Clark [23] introduce the notion of *gradual ownership types* and a corresponding consistent-subtyping relation. Their formalism provides a static guarantee of ownership invariants for fully annotated programs, but requires dynamic checks for partially-annotated programs. Their prototype works on non-generic Java programs and they analyzed 8,200 lines of code. In contrast, our inference is static and works on Java programs of up to 110kLOC.

Work on introducing generics to Java [10, 15] solves similar challenges, because leaving every type as raw is a legal typing, but a useless one that expresses no design intent and detects no coding errors. In contrast to our work, Donovan et al. [10] use heuristics to find desirable solutions and their inference requires a pointer analysis. Kieżun et al. [15] make use of type constraints to ensure behavior preservation. They also use heuristics, otherwise user's input is required.

Our algorithm for computing the set-based solution (Sect. 4.1) is similar to the algorithm used by Tip et al. [15, 24]. Both algorithms start with sets containing all possible answers and iteratively remove elements that are inconsistent with the typing rules. Our work differs as we introduce a ranking over valid typings and use the ranking to guide the automatic inference towards a final "best" typing.

Our work, as well as [24], falls in the category of type-based and constraint-based analysis, originally proposed by Palsberg and Schwartzbach [24].

7 Conclusion

We presented a unified framework for type inference and type checking of ownership type systems, and instantiated the framework for two such systems: Universe Types and Ownership Types. We presented a heuristic ranking over valid typings, and an efficient inference approach that produced maximal typings. We implemented the approach on top of the Checker Framework and presented results for Universe Types and Ownership Types on benchmarks of up to 110kLOC.

Acknowledgments. We thank the anonymous reviewers for their extensive feedback. This work was supported by NSF grants CNS-0855252 and CCF-0642911.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.
2. S. Artzi, A. Kieżun, J. Quinonez, and M. D. Ernst. Parameter reference immutability: formal definition, inference tool, and comparison. *ASE*, 16:145–192, 2008.

3. B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, volume 3924 of LNCS, pages 264–278, 2006.
4. D. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33, pages 48–64, 1998.
5. D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe types for topology and encapsulation. In *FMCO*, volume 5382 of LNCS, pages 72–112, 2008.
6. W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In *ICSE*, pages 681–690, 2011.
7. W. Dietl, M. D. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *ECOOP*, 2011.
8. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
9. W. Dietl and P. Müller. Runtime Universe type inference. In *IWACO*, 2007.
10. A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA*, pages 15–34, 2004.
11. D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, 2007.
12. W. Huang and A. Milanova. Towards effective inference and checking of ownership types. In *IWACO*, 2010.
13. W. Huang and A. Milanova. A Type System for Reference Immutability. Technical report, Rensselaer Polytechnic Institute, Department of Computer Science, 2011.
14. W. Huang and A. Milanova. On optimality of ownership type inference. Poster at ECOOP, 2011.
15. A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE*, pages 437–446, 2007.
16. K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.
17. A. Milanova and J. Vitek. Static Dominance Inference. In *TOOLS*, pages 211–227, 2011.
18. N. Mitchell. The Runtime Structure of Object Ownership. In *ECOOP*, volume 4067 of LNCS, pages 74–98, 2006.
19. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag New York, Inc., 1999.
20. M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.
21. A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004.
22. A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *PLDI*, pages 164–174, 2011.
23. I. Sergey and D. Clarke. Gradual ownership types. In *ESOP*, pages 579–599, 2012.
24. F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. D. Sutter. Refactoring using type constraints. *TOPLAS*, 33(3):9:1–9:47, 2011.
25. M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.
26. M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328. Springer, 2010.
27. M. Vetchev, E. Yahav, and G. Yorsh. PHALANX: Parallel Checking of Expressive Heap Assertions. In *ISMM*, pages 41–50, 2010.