Inferring Ownership Domains from Refinements

Ebrahim Khalaj
Department of Computer Science
Wayne State University
Detroit, MI, USA

Marwan Abi-Antoun
Department of Computer Science
Wayne State University
Detroit, MI, USA

Abstract

Ownership type qualifiers clarify aliasing invariants that cannot be directly expressed in mainstream programming languages. Adding qualifiers to code, however, often involves significant overhead and difficult interaction.

We propose an analysis to infer qualifiers in the code based on developer refinements that express strict encapsulation, logical containment and architectural tiers. Refinements include: `makeOwnedBy`, to make an object strictly encapsulated by another; `makePartOf`, to make an object logically contained in another; `makePeer`, to make two objects peers; `makeParam`, to make an object more accessible than the above choices; or `makeShared`, to allow an object to be globally aliased. If the code as-written matches the requested refinements, the analysis generates qualifiers that type-check; otherwise, it reports that the refinements do not match the code, so developers must investigate unexpected aliasing, change their understanding of the code and pick different refinements, or change the code and re-run the analysis.

We implement the analysis and confirm that refinements generate precise qualifiers that express strict encapsulation, logical containment and architectural tiers.

CCS Concepts • **Software and its engineering** → **Formal language definitions**; **Formal software verification**;

Keywords ownership type systems; type inference

ACM Reference Format:
Ebrahim Khalaj and Marwan Abi-Antoun. 2018. Inferring Ownership Domains from Refinements. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3278122.3278128

1 Introduction

Ownership type qualifiers can enforce aliasing invariants and provide stricter encapsulation than visibility modifiers. These qualifiers, when added to existing code, can expose bugs due to unwanted aliasing, and do so in a sound manner, unlike shallow visitor-based analyses such as FindBugs [15] that warn when a field is not encapsulated. Manually adding qualifiers to existing code, however, is a significant burden. Moreover, inference tools are still immature, thus leading to a low adoption of ownership type systems, despite research spanning two decades touting their value at exposing aliasing bugs that can lead to software defects.

We develop an interactive inference tool, OOGRE, to infer qualifiers for a simplified ownership type system. In our approach, developers use *refinements* to express their intent about the strict encapsulation, logical containment and architectural tiers they would like to see in the code. While strict encapsulation can be readily inferred from the code, logical containment is often subtle design intent. If the code as-written supports a requested refinement, the analysis infers qualifiers that satisfy the refinement and the type system rules, marking the refinement as *done*. If the analysis cannot infer qualifiers that are consistent with each other and the code, it marks the refinement as *skipped*, identifying the expressions where type system rules are violated.

When a refinement is skipped, it often points to a mismatch between the developers' understanding and the code, which could be a refactoring opportunity or an aliasing bug. If the developers insist that the refinement be done, they must change the code, e.g., to return a copy (clone) of a collection object instead of an alias, and replay the refinements.

OOGRE can optionally save the inferred qualifiers as annotations in the code to make explicit the object aliasing. Thus, a first output of the approach, in addition to the set of inferred qualifiers, is the attempted refinements and their status (done or skipped). By themselves, the refinements capture important aliasing invariants about the code. After evolving the code to fix bugs or refactor it, developers can replay the refinements on later versions. A second output of the approach is that the generated qualifiers enable a separate extraction analysis [1, 17] to abstractly interpret the code with the qualifiers and statically extract a sound, hierarchical, abstract object graph, that is an abstraction of the system's runtime structure, and conveys information about encapsulation, logical containment and architectural tiers.

Contributions. This paper contributes developer-driven inference of an expressive set of ownership type qualifiers. Novel features of the inference, compared to closely related work, include a larger set of qualifiers that express logical containment in addition to strict encapsulation, as well as object uniqueness and object borrowing. We evaluate the approach on two real code bases and demonstrate that it can generate from developer refinements sound and precise qualifiers that convey important information about aliasing invariants, rather than the default qualifiers where all objects are peers and globally aliased. This paper focuses on the technical details for the sound inference. Details on the implementation and the developer interaction are in an online appendix [11].

Outline. The rest of this paper is structured as follows. Section 2 informally introduces ownership type qualifiers and the refinements on a motivating example. Section 3 gives an overview of our inference analysis, and Section 4 formally describes it. Section 5 evaluates the approach on two systems. The first evaluation reproduces an experiment from closely related work and compares the inferred qualifiers to those from the earlier experiment. The second evaluation focuses on expressing design intent gleaned from external documentation. Finally, we discuss some limitations (Section 6), related work (Section 7), and conclude.

2 Ownership Qualifiers

Ownership types help developers to find and fix aliasing bugs, but require additional modifiers on variables in the code. Some definitions are in order.

Definitions. A *modifier* is an ownership keyword such as `unique`, `lent`, `owned`, or `shared`; we will define the full set later. A *qualifier* is a pair of modifiers $<p, q>$ that is added to a *variable*. A variable is any local variable, parameter, return, object creation, or field that requires a qualifier and is not of a primitive type.

Refinement. A refinement is an abstraction of a qualifier that pins the first part p of the qualifier pair $<p, q>$. If the refinement is done, OOGRE infers the second part q.

```
// Refinement pins down first part of qualifier
  UUID /*makePartOf*/ value = ...;
// vs. OOGRE infers qualifier (pair of modifiers)
  UUID<this.PD, p> value = ...;
```

Expecting developers to provide both parts, p and q, of a qualifier makes ownership types harder to apply, we believe, but evaluating this claim requires a user study. This paper focuses on the technical feasibility of inferring qualifiers based on supplying only the first part p thereof. Moreover, refinements promote thinking at a higher level of abstraction, in particular when allowing developers to directly express logical containment and architectural tiers.

```
1 class MutableClass {
2   private MyDate /*makeOwnedBy*/ d = new MyDate();
3   public MyDate getDate() {
4     return d;
5   }
6 }
```

Figure 1. v1: The refinement `makeOwnedBy` is skipped. A public method returns an alias to a private field.

2.1 Motivating Example

The CERT Oracle Secure Coding Standard for Java [12] stipulates: *Do not return references to private mutable class members.* The developer uses OOGRE refinements to express the above rule.

Act I. The developer requests a refinement by placing the structured comment[1] `/*makeOwnedBy*/` to make the field `d` of type `MyDate` *owned-by* the current `this` object of type `MutableClass` (Fig. 1), and runs OOGRE.

```
OOGRE> run
skipped: makeOwnedBy(d:Date, this:MutableClass)
```

OOGRE, however, indicates that the code does not support such a refinement (it is skipped). It also shows the expression where the analysis fails to infer valid qualifiers. In the first, non-compliant version (v1, Fig. 1), line 4 of the method `getDate()` returns an alias to the `private` field `d` of type `MyDate`, thus enabling a malicious client to directly mutate the object. This code exposes a weakness of standard visibility modifiers; although the field is `private`, the standard Java type system does not prevent developers from declaring a `public` getter method that simply returns an alias to the field. Since timestamps are used to enforce many security properties, a compliant version must return a copy or a clone of the `MyDate` object (Fig. 2, lines 22–23). Any malicious client would then mutate a copy rather than the internal representation of the object, and no harm would be done. The developer fixes the code to return a copy instead of an alias, and re-plays the refinements. This time, the refinement is done.

```
OOGRE> replay
done: makeOwnedBy(d:Date, this:MutableClass)
```

Without an inference tool, a developer would have to add many qualifiers such as `<owned,p>` (Fig. 2), both on each class declaration and on each usage of that class. The meaning of these qualifiers is not obvious to the untrained eye, so we interrupt the example to review them.

2.2 Simple Ownership Domains (SOD)

In this paper, we adopt the Ownership Domains (OD) type system [2], with some simplifications we discuss below. While there are more recent ownership type systems, many

[1] Our implementation uses language support for annotations but annotations tend to be verbose, so we use structured comments in this paper.

```java
// Class decl. takes formal domain parameters
// * owner: the first domain parameter binds
// to the owning domain;
// * p: the second domain parameter binds to any
// domain that needs to be accessed
class MutableClass<owner, p> {
  // Declare a private domain
  private domain owned;

  // At class usage: bind formal domain params
  // to actual domains
  // * MyDate::owner to this class's domain owned
  // * MyDate::p to this class's domain param.
  MyDate<owned, p> d = new MyDate();

  public MyDate<shared,shared> getDate() {
    // Since field d is in a private domain
    // Cannot return alias to field
    // return d;

    // Place copy in global domain shared
    MyDate<shared, shared> cpy = d.clone();
    return cpy;
  }
}
// MyDate::owner not same as MutableClass::owner
class MyDate<owner, p> {
  // MyDate::owned not same as MutableClass::owned
  private domain owned;
}
```

Figure 2. Qualifiers required by the type system.

are paper-only designs that have not been evaluated in practice. Others introduce additional complexities, such as variant ownership, and have not been adopted. Similarly to the related ownership type systems that are even older and that we discuss below, OD has been applied to over 100 KLOC of real Java code [17]. The main benefit of SOD, compared to other systems, is that it can express design intent such as logical containment and architectural tiers.

Key idea: placing objects in domains. The idea behind ownership type systems is to place objects in some ownership contexts, i.e., "place objects in boxes". In most ownership systems, the ownership context, i.e., the owner or the "box", is another object. In OD, objects do not own other objects directly. Instead, the ownership context is an explicit, named *domain*. Each object is exactly in one owning domain that does not change at runtime, and in turn, each object has zero or more child domains, to express object hierarchy.

Domains are declared on a class but are treated like fields, in that fresh domains are created for each instance of that class. For a domain `d` declared on a class `C` and two instances `n1` and `n2` of class `C`, the domains `n1.d` and `n2.d` are distinct, for distinct `n1` and `n2`.

When object *o* has a child domain *d*, *d* can be `private` or `public`. If *d* is private, it enforces strict encapsulation, i.e., all the objects that are inside *d* are not accessible outside of the enclosing object *o*. For example, a `graph` object keeps an object of type `Vector` containing references to `Node` objects in a private domain called `owned`, to prevent client code from removing `Node` objects and breaking the data structure. If *d* is public, it expresses logical containment, i.e., all the objects that are inside *d* are still accessible outside of the enclosing object *o*. For example, a `nd` object of type `Node` is in a public domain called `PD` of the `graph` object. Each `nd` object is conceptually part of the graph, has direct references to the internals of the graph object, but is still accessible outside of the graph to be used by client code.

Aliasing. Domains convey precision about aliasing:
- Objects in the same domain may alias;
- Objects in different domains may not alias.

Syntax. Using OD requires additional syntax on the class declarations and the usage of those classes. A class declares one or more domains using the **`domain`** keyword. A domain declaration can have the `private` or `public` modifier.

A class also takes formal *domain parameters*, e.g., `owner` and `p` on its declaration, e.g., `class C<owner, p> {...}`. The first domain parameter is called `owner` since it serves a special purpose, which we discuss below, and the second domain parameter called `p` can bind to an arbitrary domain. For instance, a collection object may reference objects in some arbitrary domain, that is unrelated to the domains that are within scope. So the collection class takes a domain parameter for the collection elements, and that arbitrary domain containing the elements referenced by the collection is bound to the second parameter.

Each object named `c` that is an instance of the class `C` must bind both domain parameters `owner` and `p` to two *actual* domains `q1` and `q2`, respectively, that are in scope. Therefore, each reference to the object *c* requires a qualifier that is a pair of actual domains `<q1,q2>`: `q1` denotes the *owning domain*, i.e., the domain where the object *c* is, and `q2` denotes the actual domain that binds to the second domain parameter `p` on the class `C`, and means any domain that the object *c* can access. By convention, we name the first domain parameter on the class `C` as `owner`. So when an object *o* defined inside class `C` has the `owner` modifier as its owning domain, this means that *o* is in the same domain as the `this` object.

Modifier `lent`. OD can express the notion of temporary sharing (borrowing) of an object using the `lent` modifier [3]. For example, an object that is passed to a method is used only for the duration of the call. An object can be borrowed from one domain to another as long as the second domain does not create a persistent reference to the borrowed object by storing it in a field. Only formal method parameters and

```
1  class MutableClass {
2    private MyDate d = new MyDate();
3
4    public MyDate getDate() {
5      return d.clone();
6    }
7  }
8  class MyDate {
9    private UUID /*makePartOf*/ value = new UUID();
10   // ... setter/getter elided...
11   public MyDate clone() {
12     final MyDate cpy = new MyDate();
13     cpy.setUUID(this.value);
14     return cpy;
15   }
16 }
```

Figure 3. v2: The refinement makePartOf is skipped. The clone method creates a shallow copy.

local variables can be lent. A field cannot be lent. As a universal sink, a lent variable allows variables with of any other modifier to be assigned to it, which also helps the inference analysis.

Modifier unique. OD can also express the notion of an object to which there is one reference using unique [3]. For example, an object that is returned by a factory method does not belong to a specific domain, so it is unique, thus enabling the caller to place the newly created object into a domain of its choosing. As a universal source, a unique variable can be assigned to another variable that has any other modifier, which also helps the inference analysis.

Simple Ownership Domains (SOD). To make inference of qualifiers tractable, we simplify the OD type system into one we call Simple Ownership Domains (SOD), as follows: 1. a single private domain per class, named owned; 2. a single public domain per class, named PD; 3. an explicit owning domain parameter, named owner; 4. a single free domain parameter per class, named p; and 5. hard-coded object access permissions[2]: (a) objects in private domains are inaccessible to the outside; (b) objects in public domains are accessible; and (c) objects in sibling domains are accessible to each other. In the remainder of this paper, we use SOD.

2.3 Motivating Example (continued)

Act II. The developer wants to ensure that the cloned object is in fact a deep rather than a shallow clone, i.e., it has no references to parts of the original object. So she requests the refinement /*makePartOf*/ to have the field value of type UUID be part-of the object of type MyDate (Fig. 3).

[2]Ownership Domains use explicit *domain links* to specify general object access permissions at the cost of additional syntax. Similarly to Simple Loose Ownership Domains (SLOD) [14], we hard-code simple access permissions.

```
class MyDate<owner, p> {
  private UUID<owner,owner> value = new UUID();

  public UUID<owner,owner> getUUID() {
    return this.val;
  }

  public MyDate<owner,owner> clone() {
    // The copy is peer with this object
    final MyDate<owner,owner> cpy = new MyDate();
    cpy.setUUID(this.value);
    return cpy;
  }
}
```

Figure 4. v2: The inferred qualifiers show that all objects are peers and potentially aliased.

```
1  class MyDate<owner, p> {
2    // field decl., getter, setter same as above
3    public MyDate<unique,p> clone() {
4      final MyDate<unique,p> cpy = new MyDate();
5      UUID<unique,p> uuid = new UUID();
6      // Type system violation:
7      // this.PD cannot be assigned to cpy.PD
8      // cpy.setUUID(this.value);
9      // However, unique can flow to cpy.PD
10     cpy.setUUID(uuid);
11     return cpy;
12   }
13 }
```

Figure 5. v3 creates a deep copy.

```
OOGRE> run
skipped:  makePartOf(value:UUID, this:MyDate)
```

OOGRE indicates that this refinement is skipped. To investigate this, the developer looks at the inferred qualifiers in the code (Fig. 4). She sees mostly owner qualifiers, which indicates that the objects are peers and therefore, may alias freely or share some inner objects (since objects in the same domain may alias), so she suspects a shallow copy.

Indeed, this code is still non-compliant since the clone() method of MyDate returns a shallow copy (Fig. 3, line 13), where both objects of type MyDate share the same inner object of type UUID. Indeed, if OOGRE were to satisfy this refinement, it would infer qualifiers that violate the type system rules. Since OOGRE infers only qualifiers that typecheck, it marks the refinement as skipped.

Act III. The developer fixes the code to return a deep copy (Fig. 5, line 10), and replays the refinements.

```
OOGRE> replay
done:  makePartOf(value:UUID, this:MyDate)
```

```
final MyClass<unique, p> d1 = new MyDate();
UUID<unique, p> uuid1 = new UUID();
d1.setUUID(uuid1);

final MyClass<unique, p> d2 = new MyDate();
UUID<unique, p> uuid2 = new UUID();
d2.setUUID(uuid1); // Typo: meant uuid2
```

Figure 6. `UUID` objects in `d1.PD` and `d2.PD` cannot alias.

This time, the refinement is done, and OOGRE infers the qualifier `unique` on the return of the method `clone`, which indicates that it returns a fresh object.

In this example, a `UUID`, a Universally Unique IDentifier, is used for persistence. So two objects being persisted should not share the same `UUID` object. But the `UUID` object is not strictly encapsulated so it cannot be in a private domain. For such cases, using a public domain is a natural expression of the aliasing invariant. If two distinct objects are made to share the same `UUID` object under certain conditions, the code may run, pass basic unit tests, but the issue will be noticed only when saving to or loading from persistent state and hitting those conditions. Having a sound analysis here is invaluable, as it can identify all cases of aliasing of `UUID`s, including cases that are not exercised by the current test suite. Using public domains identifies unintended sharing of objects, and prevents developers from mixing objects in the domain `n1.PD` with objects in the domain `n2.PD` for distinct `n1` and `n2` (Fig. 6).

2.4 Language of Refinements

A refinement can:

- `makeOwnedBy`: make an object *owned by* the current `this` object by placing it in its private domain `owned`;
- `makePartOf`: make an object conceptually *part of* the current `this` object by placing it in its public domain `PD`;
- `makePeer`: place an object in the same domain as the current `this` object by setting the owning domain to `owner`;
- `makeParam`: place an object in a parameterized domain of another object; this way, a collection object can refer to objects in another domain, for instance;
- `makeShared`: place an object in the global domain `shared`.

In addition, OOGRE supports the following operations:

- **run:** run or replay all the refinements on the current code;
- **runAuto:** without any developer refinements, attempt automated refinements on all variables in the program, guided by a ranking strategy that we discuss later;
- **runAssisted:** in addition to the refinements, attempt automated refinements on the remaining variables.

3 Set-Based Solution

Our analysis computes a set-based solution, by starting with sets containing all possible qualifiers and iteratively removing ones that are inconsistent with the type system rules.

3.1 Positioning

Our inference analysis instantiates the Huang et al. framework [8] by providing a set of qualifiers, adaptation functions, and type-system-specific constraints.

Common definitions. To clarify our contribution in this paper, we reuse the terminology of Huang et al., as follows:
Typing: given a program P and an ownership type system F with a set of possible qualifiers Q_F, a typing T maps each variable in P to a qualifier in Q_F. A typing is *valid* for P in F if it renders P well-typed in F;
Set Mapping: in a set-based solution, a *set mapping* S maps each variable in program P to a set of possible qualifiers in type system F. Crucially, one set mapping may contain several valid typings and if one set becomes empty, the whole set mapping is discarded.
Adaptation: an adaptation function finds the qualifier of the left-hand or right-hand side of an expression, when the receiver of that expression is not `this`. It adapts the qualifier of an inner variable (a field, method parameter or return) from the viewpoint of the receiver, and produces the qualifier of the outer variable.

Our contributions over Huang et al. Our first contribution is about how developers use the analysis. Huang et al. ask developers to provide full qualifiers in the code to guide the inference. In contrast, a refinement pins down just the owner and lets the analysis infer the domain parameter. Our other contribution is having more expressive qualifiers.

First, to support logical containment (`makePartOf` refinements) and the SOD type system, we integrate public domains (`n.PD`) into the set-based solution, which requires new adaptation cases. In SOD, an actual domain can be `n.PD` where `n` can be `this` or a **`final`** field. In contrast, in Ownership Types (OT) [4], `n` is always `this` and there is no `PD`. For SOD, adaptation has to consider the different cases for `n`. Also, SOD has more valid qualifiers. For instance, `<p,owner>` is valid in SOD but prohibited in OT, where an object is accessible only to its owner. Therefore, one set mapping can lead to many more possible typings, and we have bigger initial sets of possible qualifiers.

Second, our inference analysis can infer architectural tiers, thus allowing developers to place objects in two different domains and express two-tiered designs.

Third, our inference analysis infers object borrowing (`lent`) and uniqueness (`unique`), which are needed to express common programming idioms. For example, a factory method returns `unique` objects. As an added bonus, the flexibility of `lent` and `unique` enables the set-based solution to find a typing (discussed in Section 3.6).

3.2 Ranking SOD Qualifiers

We define a ranking over SOD qualifiers, one that discourages aliasing and makes the object ownership tree more hierarchical, as is common in ownership inference [19]. First, we rank the SOD modifiers: `unique` is the highest ranked modifier, since it is the universal source and can be assigned any other modifier. The modifier `lent` is ranked second, since it allows variables with any modifier to be assigned to it. `owned` is ranked next, since an `owned` object is strictly encapsulated. The next ranked modifier is `n.PD`, which is less restrictive. A public domain `PD` can be referred to from outside, similarly to a field, using the `n.PD` modifier. Every object in `owned` can be in `PD`, but the reverse does not hold. Domain parameters `owner` and `p` are ranked next, respectively. The lowest ranked is `shared`, which enables objects to be globally aliased.

We extend the ranking of modifiers to qualifiers by first considering the owning domain (p), then the domain parameter (q) of a qualifier (<p, q>). Therefore, to compare two qualifiers, one is higher ranked that has a higher ranked owning domain. If two qualifiers have the same owning domain, then the one that has a higher ranked domain parameter is ranked higher.

3.3 Starting From an Initial Set Mapping

First, the analysis maps each variable to an *initial set of qualifiers* that contains all the possible qualifiers by considering the kind of a variable and the SOD constraints. In SOD, there are seven modifiers: `unique`, `lent`, `owned`, `n.PD`, `owner`, `p` and `shared`. The modifiers `unique` and `lent` can appear only as the owning domain in a qualifier, so there are 10 qualifiers that contain `unique` or `lent`. The modifier `owned`, internally qualified by the analysis as `this.owned`, can be the owning domain of 5 qualifiers. For `n.PD`, `owner`, `p` as the owning domain, there are 4 qualifiers, since they do not accept `owned` as a domain parameter. Only the qualifier `<shared, shared>` is allowed since the domain parameter cannot be ranked higher than the owning domain `shared`. Therefore, the full set of qualifiers contains 28 qualifiers, but all variables do not map to the full set.

A field cannot be `lent`. We also do not support `unique` fields which require destructive reads. So a field variable maps to an initial set that contains 18 qualifiers. We do not support `unique` method parameters in order to infer more general method signatures. Method returns cannot map to qualifiers containing `lent`. So the initial set of qualifiers of method parameters and returns contain 23 members. Local variables map to the full set of qualifiers. To determine if `unique` is allowed, the transfer functions invoke an intra-procedural standard Live Variables Analysis (LVA).

3.4 Applying Refinements

Each refinement changes the set of qualifiers of one variable in the set mapping into a singleton set. We call that variable the *source variable* of the refinement. The analysis then runs transfer functions to infer qualifiers for all the other variables.

A refinement pins only the owning domain of the qualifier of the source variables. To pick the domain parameter of those qualifiers, the analysis applies a preference strategy, then runs the transfer functions on the set mapping.

The analysis accesses the body of each method declaration and runs the transfer functions on the current set mapping and all the expressions in the program to validate the changes and infer the other changes.

Domain parameter preference strategy. To select the domain parameter, the analysis uses a general strategy that prefers modifiers `p`, `owner`, `PD`, `owned` and `shared` in this order. The analysis prefers a more flexible domain parameter over the others, so it prefers domain parameters over actual domains. As a domain parameter, `p` can bind to any domain, so that is the most flexible solution. The modifier `PD` is more flexible than `owned`, since an object that is in the public domain can be accessed by other objects. This general strategy changes based on the owning domain of the qualifier, where the analysis does not pick the same domain parameter for an owning domain. For example, for a `makePartOf` refinement, the preference strategy picks `owned` over `PD`, if `p`, `owner` do not create a valid qualifier for the refinement.

Respecting the whole set mapping. The analysis applies each new refinement on a set mapping resulting from the previous refinements that are done. This way, a new refinement cannot undo the result of a previous refinement that is done. As a result, after each refinement, the sets of qualifiers get smaller, and the analysis may find a valid typing without too many additional refinements. When more sets of qualifiers are singletons, the set mapping is closer to becoming a typing. One drawback of respecting the whole set mapping is that a previous refinement may remove some qualifiers from the set mapping, but these qualifiers may be needed for a later refinement, so the later refinement is skipped.

3.5 Running Transfer Functions

The analysis applies a transfer function on each expression in the program. Each transfer function takes a set mapping S, an expression and produces an output S'. A transfer function removes infeasible qualifiers from the set of qualifiers of each variable. The transfer functions run until a fixed point when the sets of qualifiers of all variables no longer change. At the fixed point, for each variable x, S contains a set of qualifiers $S[x]$, which can be an empty set. If there exists a variable x for which $S[x]$ is empty, the entire S is discarded and not used any further. If after running the transfer functions, all the instances of S for different domain parameters are discarded, then the refinement is skipped, and the analysis does not save any qualifiers.

3.6 Finding a Valid Typing from a Set Mapping

At the fixed point, each variable in S is mapped to a set of qualifiers that are from multiple valid typings. Using the defined ranking between qualifiers, the analysis extracts a typing T from S by applying the *max* function on the set of qualifiers of each variable x in S. The *max* function picks the highest ranked, i.e., maximal, qualifier in the set of qualifiers of x. The typing T_{max} is the maximal typing. $T_{max}[x] = f(S[x])$ where $f = max$ for all x.

In a set-based solution, extracting a valid typing is not trivial. If the maximal typing T_{max} does not type-check the program, then the optimality property does not hold for the program and SOD. A key finding in Huang et al. is that for certain ownership type systems, one can derive a *unique maximal*, i.e., best, typing T from the set-based solution S. "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."

For programs with arbitrary qualifiers (including partially specified ones, or none) and some ownership type systems, the optimality property may not hold and the analysis may struggle to find a valid typing.

Our insight to make the set-based solution work well for SOD is adding `lent` and `unique`, which act as a universal source and sink, respectively. So `lent` and `unique` variables may be assigned to variables that are mapped to more than one qualifier, which gives the transfer functions more choices of valid qualifiers for the left-hand side or the right-hand side of an assignment. Moreover, `unique` and `lent` are highly ranked, and by having them as the owning domain of qualifiers in the sets of S, the analysis picks those qualifiers using the *max* function to find a valid maximal typing T_{max} very efficiently. For example, for the expression $x = y$, $T[x]$ and $T[y]$ type-check the expression when:

$$\text{(sink)} <\texttt{lent},\_> \in S[x] \implies <\texttt{lent},\_> = max(S[x])$$
$$<\texttt{lent}, q> = T[x] \implies \forall p.<p, q> = T[y] \in S[y]$$
$$\text{(source)} <\texttt{unique},\_> \in S[y] \implies <\texttt{unique},\_> = max(S[y])$$
$$<\texttt{unique}, q> = T[y] \implies \forall p.<p, q> = T[x] \in S[x]$$

To make the optimality property hold, developers provide the analysis with additional information. In Huang et al., this information consists of a small number of qualifiers that developers add manually to selected variables, and that the analysis uses to initialize the sets with singletons and respect the qualifiers. In our approach, this information consists of refinements to express aliasing invariants. The analysis then infers the qualifiers and overwrites existing ones in the code. In Huang et al., when adding a qualifier manually, developers provide the full pair (p and q). A refinement, however, pins only the owning domain (p) and the analysis infers the actual domain for the domain parameter (q).

In order to check the validity of T, the analysis type-checks it. The maximal typing T_{max} is a valid typing if, for each variable x, $T_{max}[x]$ type-checks the program. If T_{max} is a valid typing, the analysis saves its qualifiers to the code. If T_{max} does not type-check the program, it means the optimality property does not hold and there is at least one conflict to resolve. To repeat Huang et al. [8], "a statement s is a conflict if it does *not* type check with the *maximal qualifier* derived from the set-based solution".

3.7 Resolving Conflicts

To resolve conflicts, Huang et al. require that developers guide the inference analysis by adding more qualifiers to the code manually. In the Huang et al. approach, developers do not add manual qualifiers to simply express their design intent. They add qualifiers to resolve conflicts after each iteration of the inference analysis.

Our approach has three modes for resolving conflicts: Manual, Assisted, or automatic (Auto). In the Manual mode, the developers request refinements to either express their design intent or to resolve conflicts. In the Assisted mode, in addition to processing all the developer refinements, OOGRE applies automated refinements on all fields and local variables in the code. Finally, in the Auto mode, the developers do not provide any refinements and OOGRE applies automated refinements as in the Assisted mode.

3.8 Worked Example of the Set-Based Solution

We illustrate on the motivating example how the set-based solution changes the sets of qualifiers for a variable at different steps (Table 1). The current set of qualifiers for a variable x is $S[x]$. After running the transfer functions and applying adaptation functions, the new set is $S'[x]$. The qualifier that is mapped to the variable x in a typing T is shown as $T[x]$.

Consider the method invocation `cpy.setUUID(uuid)` in Fig. 5, line 10. Assume the variables of the method invocation are mapped to the sets of qualifiers shown in the second column at a given time. The variable `par` is the formal method parameter of the method `setUUID`. The third column shows the sets of qualifiers after adaptation. The last column shows the qualifier of each variable in a typing T, which type-checks the method invocation. In Section 4.4, we discuss the transfer function for method invocation expression in detail.

3.9 Time complexity

In our approach and using SOD, the time complexity of running the inference analysis for each refinement is quadratic. The size of the full set of qualifiers (Q_F) for SOD is at most 28, which is a relatively big constant and makes the running time of the set-based solution for SOD a factor of its running time for Ownership Types or Universe Types. If the program contains n variables and the developers attempt m refinements, the time complexity to find a valid typing is $m \times O(|Q_F|^2 * n^2)$. For the Auto and Assisted modes, the overall end-to-end time complexity is $O(n^3)$.

4 Formalization

In this section, we formalize the abstract syntax, and the adaptation functions for SOD qualifiers. Due to space limits,

Table 1. Sets of qualifiers for variables of a method invocation before and after adaptation.

var	S[var]	S'[var]	T[var]	Comments
uuid	{<unique,p>,<owned,p>,<PD,p>}	adapt(S[cpy],S[par]) ∩ S[uuid] = {<unique,p>}	<unique,p>	adapting method argument (outer) qualifiers
par	{<lent,p>}	adapt(S'[uuid],S[cpy]) ∩ S[par] = {<lent,p>}	<lent,p>	adapting method parameter (inner) qualifiers
cpy	{<unique,p>,<owned,p>,<PD,p>}	adapt(S'[uuid],S'[par]) ∩ S[cpy] = S[cpy]	<unique,p>	adapting method receiver qualifiers

we show the transfer function for method invocation. The others are similar in nature. The full formalization of the analysis is in the first author's dissertation [10, Chap. 4].

4.1 Abstract Syntax

We formalize our analysis by adapting Featherweight Domain Java (FDJ), which models a core of a Java-like language with Ownership Domains [2]. To enable comparisons with Huang et al., we simplify FDJ to the A-normal form and assume that each method has a single parameter, and each class has a single field. Of course, our implementation handles the general case. We also simplify FDJ to reflect the SOD simplifications such as hard-coded domain names and default domain links (See Section 2.2).

In our abstract syntax (Fig. 7), C ranges over class names; τ ranges over types; t ranges over qualifiers; f ranges over field names; v ranges over values; e ranges over expressions; x ranges over variable names; n ranges over values and variable names; the set of variables includes the distinguished variable `this` of type τ_{this} used to refer to the receiver of a method invocation, field read or field write; m ranges over method names; q and r range over formal domain parameters, actual domains, or the global domain `shared`; p ranges over `unique`, `lent` and all other possible actual domains; an overbar denotes a sequence; the fixed class table CT maps classes to their definitions; a program is a tuple (CT, e) of a class table and an expression; Γ is the typing context; S defines a map from each variable to a set of qualifiers; T defines a map from each variable to a single qualifier. $S[x]$ denotes reading the set of qualifiers for x in S; and $S' = [x \mapsto Q]S$ denotes updating the set of qualifiers for x in S.

Since in `n.PD`, `n` can be `this`, the adaptation has to distinguish between the inner `this` and the outer `this`. To avoid capture during adaptation, we substitute `that` for the inner `this` using [`that`/`this`]. In the transfer functions, after adaptation, `that` is substituted with `this` ([`this`/`that`]) for the inner `this`, and the outer `this` is substituted with the corresponding object name `n`, using [`n`/`this`], if there is `n.PD` in the resulting set.

4.2 Adaptation Functions

The qualifier of an expression is the result of an adaptation, when the receiver of the expression is not `this`. For each adaptation function, there is an inner qualifier, a receiver qualifier and a result or outer qualifier. More formally, we say t_{out} is the result of adapting t_{in} from the viewpoint of t_{rcv}, and use the notation: $t_{rcv} \triangleright t_{in} = t_{out}$

$$
\begin{array}{rcl}
CT & ::= & \overline{cdef} \\
cdef & ::= & \textbf{class}\ C\texttt{<owner,p>}\ \textbf{extends}\ C'\texttt{<owner,p>} \\
 & & \{\ \textbf{domain}\ \texttt{owned};\ \ dom;\ \ \tau\ f;\ \ md\ \} \\
dom & ::= & \textbf{public domain}\ \texttt{PD}; \\
md & ::= & \tau_R\ m(\tau\ x_m)\ \{\overline{\tau\ y}\ e; \textbf{return}\ y_m;\} \\
e & ::= & e;\ e\ \mid x = \textbf{new}\ C\texttt{<}p,q\texttt{>}()\ \mid y = x.f \\
 & & \mid y = \texttt{this}.f\ \mid x.f = y\ \mid \texttt{this}.f = y\ \mid x = y \\
 & & \mid x = y.m(z)\ \mid x = \texttt{this}.m(z) \\
n & ::= & x\ \mid\ v \\
q,r & ::= & \texttt{owner}\ \mid \texttt{p}\mid\ n.\texttt{PD}\ \mid\ \texttt{this.owned}\ \mid\ \texttt{shared} \\
p & ::= & \texttt{unique}\ \mid \texttt{lent}\mid\ q \\
\tau & ::= & C\texttt{<}p,q\texttt{>}\quad \textit{Type} \\
t & ::= & \texttt{<}p,q\texttt{>}\quad \textit{Qualifier} \\
x,y,z & \in & \textit{Variables} \\
\Gamma & ::= & x\ \rightarrow\ \tau\quad \textit{Static typing context} \\
S & ::= & \emptyset\ \mid\ S \cup \{x \mapsto \{\texttt{<}p,q\texttt{>}\}\}\quad \textit{Set Mapping (SM)} \\
T & ::= & \emptyset\ \mid\ T \cup \{x \mapsto \texttt{<}p,q\texttt{>}\}\quad \textit{Typing}
\end{array}
$$

Figure 7. Abstract syntax for SOD, adapted from Featherweight Domain Java (FDJ) [2].

Adapt-Gen

$$
\frac{\Gamma; n_{this} \vdash n_{this} : C_{this}\texttt{<}p_{this}, q_{this}\texttt{>} \qquad t_{rcv} = \texttt{<}p_0, q_0\texttt{>}}{\Gamma; n_{this}; n \vdash t_{rcv} \triangleright t_{in} = t_{out}}
$$

Figure 8. General rule for the adaptation of `n.PD`, `lent` and `unique`. p_0 and q_0 can be any modifier.

An inner qualifier can be the qualifier of a field, a method parameter, or a method return. Other than `owner`, `p` and `shared`, an inner qualifier can contain `this.PD`, `lent` or `unique`, so we need new adaptation cases to handle them. We show the general adaptation rule in Fig. 8.

Here we explain one of the adaptation cases when t_{in} is <`that.PD`, `p`>. we substitute `this` with `that`, since t_{in} is the inner qualifier, and the corresponding variable is not declared in the class of `this`. Later on, in the transfer functions, `that` is substituted with `this` again. The qualifier of the receiver, t_{rcv}, can be any qualifier, so we show it as $\texttt{<}p_0, q_0\texttt{>}$. For the resulting qualifier, t_{out}, the owning domain is `n.PD` where `n` is a final field of the same type as the receiver declared in the current class. The domain parameter is the domain parameter of t_{rcv} which is q_0. So t_{out} is <`n.PD`, q_0>.

4.3 Set-Level Adaptation

For the set-based solution to handle all the possible combinations of qualifiers, each transfer function uses three types of adaptation functions that operate on sets of qualifiers. First, Adapt-Out ($\triangleright_o$) adapts qualifiers of the outer variable by accepting qualifiers of the inner and the receiver (n_{rcv}) variables as input. Second, Adapt-In ($\triangleright_i$) adapts qualifiers of the inner variable by accepting qualifiers of the outer variable and the receiver variable as input. Third, Adapt-Rcv ($\triangleright_r$) adapts qualifiers of the receiver variable by accepting qualifiers of the outer variable and the inner variable as input.

The judgement form for set-level adaptation is as follows, where Q_i is a set of qualifiers:

$$\Gamma; n_{this}; n_{rcv} \vdash Q_1 \triangleright_X Q_2 = Q \text{ where } X = o \text{ or } X = i \text{ or } X = r$$

4.4 Transfer Functions

Our transfer functions generalize the transfer functions in Huang et al. to handle SOD qualifiers with `PD`, `lent` and `unique`. A transfer function accepts an expression and S and accesses the set of qualifiers of the variables of the expression in S. By intersecting the sets of qualifiers of variables, a transfer function removes the infeasible qualifiers from the set of qualifiers of the variables. Then it updates the sets of qualifiers of the corresponding variables in S and creates S'. Fig. 9 shows the inference rule for the method invocation transfer function. For the transfer functions that require adaptation, we handle qualifiers that contain `n` in `n.PD`. We highlight in gray our extensions to the Huang et al. rules. The judgement form for the transfer function over an expression e is as follows:

$$\Gamma; S; n_{this} \vdash e,\ S'$$

By calling the *mdbody*() auxiliary judgement, the rule extracts x_m that is the formal method parameter. The rule asserts that a method parameter cannot be `unique` as it is unsupported in this version of our work, and a method return cannot be `lent`. First, the rule substitutes `this` with `that` in the sets the qualifiers of x_m and y_m, since they are not declared in the class of `this`. The rule does Adapt-Out using the sets of qualifiers of y and x_m and the resulting set is $Q1_o$. The result of Adapt-Out may contain `lent`. A qualifier with `lent` as the owning domain in $Q1_o$ is valid, if the set of qualifiers of x_m contains `lent`. Otherwise, the rule removes `lent` from $Q1_o$. For each qualifier $t1_o$ in $Q1_o$, if $t1_o$ contains y.`PD` as the owning domain, the domain parameter, or both, if there is a qualifier $t1$ in set of qualifiers of z ($S[z]$) where instead of y.`PD`, $t1$ contains `this.PD`, the rule substitutes `this` with y for $t1$. The set $Q2_o$ is the result of Adapt-Out using the sets of qualifiers of y and y_m. Again, the rule validates `lent` qualifiers in $Q2_o$ and substitutes `this` with y for each qualifier $t2$ in $S[x]$, if there is a corresponding qualifier $t2_o$ in $Q2_o$ containing y.`PD`. By intersecting $Q1_o$ with $S[z]$ and $Q2_o$ with $S[x]$, the rule computes the new sets of qualifiers for z and x, which are Q_z and Q_x, respectively.

By applying Adapt-In on Q_z and $S[y]$, the rule computes a set of qualifiers $Q1_i$ and the result of intersecting it with $S[x_m]$ is the new set of qualifiers for x_m, Q_{x_m}. Again, by applying Adapt-In on Q_x and $S[y]$, and intersecting the result with $S[y_m]$, the rule computes the new set of qualifiers of y_m, Q_{y_m}. For y, the rule does Adapt-Rcv and computes the result using Q_z, Q_f, Q_x and Q_m. Then it intersects the result of Adapt-Rcv with $S[y]$ and computes Q_y, the new set of qualifiers for y. For x_m and y_m, `that` is substituted with `this` in the computed sets of qualifiers.

5 Evaluation

In this section, we evaluate OOGRE on two subject systems: JDepend and JHotDraw. We use "we", this paper's first author, to refer to the developer conducting the evaluation.

The evaluation aims to answer the following questions:

- RQ0–Does a set-based solution make the set of qualifiers of most variables singleton when refinements specify only owning domains?
- RQ1–Does the analysis infer qualifiers that are more precise than `<shared, shared>`?
- RQ2–For the same system, does the analysis infer SOD qualifiers that are more precise than OT or UT?
- RQ3–Does the analysis infer SOD qualifiers that express logical containment and two top-level domains, in addition to strict encapsulation?

5.1 JDepend Evaluation

JDepend (around 4.7 KLOC) traverses packages of a Java project to generate several metrics. We analyze JDepend to reproduce one experiment that inferred OT qualifiers [8]. To familiarize ourselves with the code base, we first use the Auto mode. We then start over using the Manual mode and perform refinements. We measure the qualifiers inferred by OOGRE using Auto and Manual modes.

Manual mode. We attempt 28 refinements in total to express object hierarchy and resolve conflicts, and out of those, 24 are done. Therefore 85% of refinements are done. At first, we apply 16 refinements, and out of those 14 are done. However, those refinements are not enough to find a typing, so we apply 14 more refinements to resolve conflicts, out of which 12 are done, and OOGRE finds a typing (RQ0).

Metrics on inferred qualifiers. Table 2 measures the qualifiers of the inferred typings by OOGRE using the Auto and Manual modes. For the reproduced experiment, we show its numbers (OT-1), as well as the original numbers (OT-0). First, we translate OT qualifiers into the equivalent SOD subset by turning off `PD`, `lent` and `unique`. This way, we reproduce the OT experiment using our tools and compare the inferred OT and UT qualifiers to the SOD ones. In reproducing the experiment, we keep the explicit qualifiers they provide[3] and the analysis infers remaining ones by respecting the

[3] http://www.cs.rpi.edu/~huangw5/cf-inference/

TF-Invk

$$\frac{\begin{array}{c} mdbody(m) = (x_m, y_m) \qquad \texttt{<lent}, q_{y_m}\texttt{>} \notin S[y_m] \qquad \texttt{<unique}, q_{x_m}\texttt{>} \notin S[x_m] \\ \Gamma; n_{this}; y \vdash S[y] \triangleright_o [\texttt{that/this}]S[x_m] = Q1_o \qquad \texttt{<lent}, q_{x_m}\texttt{>} \notin S[x_m] \implies \texttt{<lent}, q1_o\texttt{>} \notin Q1_o \\ \forall t1_o \in Q1_o\ s.t.\ t1_o = \texttt{<}y.\texttt{PD}, q\texttt{>} \text{ or } t1_o = \texttt{<}p, y.\texttt{PD>} \text{ or } t1_o = \texttt{<}y.\texttt{PD}, y.\texttt{PD>}\ \exists t1 \in S[z]\ s.t.\ [y/\texttt{this}]t1 = t1_o \\ Q1_o \cap (S[z] \leftarrow t1_o) = Q_z \qquad \Gamma; n_{this}; y \vdash Q_z \triangleright_i S[y] = Q1_i \qquad Q1_i \cap S[x_m] = Q_{x_m} \\ \Gamma; n_{this}; y \vdash S[y] \triangleright_o [\texttt{that/this}]S[y_m] = Q2_o \qquad \texttt{<lent}, q_{y_m}\texttt{>} \notin S[y_m] \implies \texttt{<lent}, q2_o\texttt{>} \notin Q2_o \\ \forall t2_o \in Q1_o\ s.t.\ t2_o = \texttt{<}y.\texttt{PD}, q\texttt{>} \text{ or } t1_o = \texttt{<}p, y.\texttt{PD>} \text{ or } t2_o = \texttt{<}y.\texttt{PD}, y.\texttt{PD>}\ \exists t2 \in S[x]\ s.t.\ [y/\texttt{this}]t2 = t2_o \\ Q2_o \cap (S[x] \leftarrow t2_o) = Q_x \qquad \Gamma; n_{this}; y \vdash Q_x \triangleright_i S[y] = Q2_i \qquad Q2_i \cap S[y_m] = Q_{y_m} \\ \Gamma; n_{this}; y \vdash Q_z \triangleright_r Q_{x_m} = Q1_r \qquad \Gamma; n_{this}; y \vdash Q_x \triangleright_r Q_{y_m} = Q2_r \qquad Q1_r \cap Q2_r \cap S[y] = Q_y \\ S' = [z \rightarrow Q_z, x_m \rightarrow [\texttt{this/that}]Q_{x_m}, x \rightarrow Q_x, y_m \rightarrow [\texttt{this/that}]Q_{y_m}, y \rightarrow Q_y]S \end{array}}{\Gamma; S; n_{this} \vdash x = y.m(z),\ S'}$$

Figure 9. Transfer functions.

OT ranking. As an aside, the number of variables for Auto, Manual and OT-1 are different from Huang et al. for OT-0 and UT, because we isolate and analyze only the `swingui` entry point and add library stubs.

When discussing Table 2, we compare the owning domain of the inferred qualifiers, referring to them simply as modifiers. By comparing the modifiers of Auto and Manual, Auto tends to respect the ranking more and infer more precise qualifiers. So the number of the local domain modifiers (`owned` or `PD`) is higher in the typing that Auto infers compared to OT-1 and even Manual (RQ2). This feature is supported by design since Auto attempts `makeOwnedBy` first, then `makePartOf` and finally `makePeer`. Therefore, more objects are placed in `owned` or `PD` compared to Manual. By comparing the inferred qualifiers of Auto to the OT-1 ones, Auto infers 19 more `owned` modifiers, which means more precision (RQ1). Moreover, using public domains where strict encapsulation does not hold, Auto infers 59 `PD` modifiers that are also precise. OT does not support public domains, so most of the `PD` modifiers of Auto are replaced by less precise `owner` modifiers in OT-1.

Using the Manual mode, we promote some objects to higher levels of the object hierarchy, so we do not attempt as many `makeOwnedBy` and use `makePeer` instead. Manual infers fewer `owned` modifiers compared to OT-1. When expressing design intent, our goal is not to maximize the number of strictly encapsulated objects. Also OT forces objects to be either owned or peers. In SOD, objects can be in public domains (33 `PD` modifiers), thus confirming RQ3.

For Huang et al.'s UT experiment, the optimality property holds and there is no need to resolve conflicts. However, there are only 14 `owned` modifiers out of 542. Instead, there are 433 `owner` modifiers (`peer` in UT), which make many objects peers. The `owner` modifier is less precise compared to `owned` and does not allow careful reasoning about aliasing (RQ2). Moreover, in the UT experiment, there are 95 `any` modifiers that provide no ownership information and 102 purity modifiers that must be separately inferred.

[4]For UT and OT-1, the number are from [8] Fig. 9 and 10, respectively.

Table 2. JDepend qualifiers for the different experiments. **vars** is the number of variables. Each modifier column measures the qualifiers with the corresponding owning domain. E.g., column `owned` measures the `<owned, _>` qualifiers. Column `<p>` is the number of qualifiers for type variables to support generic types [11]. Row OT-0 refers to the original experiment from Huang et al., and OT-1 is our reproduced experiment. Numbers in parentheses are percentages.

Mode	**vars**	`owned`	`PD`	`owner`	`p`	`shared`	`<p>`	`lent`	`unique`
Auto	562	122 (22)	59 (10)	110 (20)	32 (6)	131 (23)	46 (8)	34 (6)	28 (5)
Manual	562	73 (13)	33 (6)	110 (20)	0 (0)	165 (29)	46 (8)	61 (11)	74 (13)
OT-1	562	103 (18)	n/a (0)	168 (30)	119 (21)	123 (22)	46 (8)	3 (1)	n/a (0)
OT-0	542	130 (24)	n/a (0)	156 (29)	128 (24)	128 (24)	n/a (0)	n/a (0)	n/a (0)
UT[4]	542	14 (3)	n/a (0)	433 (80)	n/a (0)	n/a (0)	n/a (0)	n/a (0)	n/a (0)

5.2 JHotDraw Evaluation

Next, we analyze JHotDraw (JHD), around 15 KLOC, an object-oriented framework for building graphical drawing applications. For the root class, we pick the JavaDrawApp sample application built using the framework.

JHD implements design patterns such as Observer and Composite. It also follows the framework layering technique: it has a framework package of core framework interfaces, a package of abstract base classes that provide default implementations for the framework interfaces, and concrete classes that extend the abstract base classes. So JHD has many cases of multi-level inheritance and overriding that the analysis must handle.

Based on a preliminary understanding of the design, we analyze if the code follows a two-tiered Document-View architecture. We use the two domains, `owned` and `PD`, on the root class `Main` to represent the Document and View tiers, respectively. So we express the two-tiered architecture using `makedOwnedBy` and `makePartOf` refinements that involve the root object.

Procedure to identify refinements. Objects that represent drawings and shapes (figures) that are persisted belong to the Document tier. So we place objects of type `Drawing`, and `Figure` and their subclasses in the `owned` domain on the root class Main. Other objects that display or modify the Document objects belong to the View tier. So we place objects of type `Editor`, `Tool` and `Command` and all of their subclasses in the `PD` domain on the root class `Main`.

Manual refinements. JHD is a framework and parts of its code are unused by the sample application being analyzed. This unused code is under-constrained, which means that OOGRE cannot remove some infeasible qualifiers and reduce the size of the sets of qualifiers for the variables in the unused code. Due to under-constrained code in JHD, OOGRE cannot find a typing. We apply 42 refinements but we do not think too hard about each one. As soon as we decide to express the Document-View architecture, and understand in which domain to place the key objects, we easily identify refinements in terms of the base types like `Figure`, `Editor`, `Tool` and `Command`. We then apply the same refinement on all their subclasses. Out of the 42 refinements, one refinement is skipped, and all the others are done.

We start with one `makeOwned` refinement and move the object of type `JavaDrawApp` into the domain `owned` of the `Main` class. Next, we apply 8 `makeParam` refinements on all the objects of type `Figure` to move them into `PD` domain of `Main` that represents the Document tier. We use `makeParam` to move objects into the `PD` domain of the `Main` class because the very first `makeOwned` refinement picked `PD` as the domain parameter for the object of type `JavaDrawApp`, so applying `makeParam` on that object results in moving objects into `PD` domain of the root object. The remaining 33 refinements are `makeOwner` refinements on the objects of type `Editor`, `Command`, `Tool` and their subclasses to move them to the `owned` domain of the class `Main` that represents the View tier. The refinements are done, therefore, we are able to express the two-tiered architecture (RQ3).

According to Table 3, for the Manual mode of JHD, there are 9,827 variables including `this` variables, for which OOGRE does not save qualifiers in the code. Although JHD is only three times the size of JDepend, it has many more variables. In Table 2, not including `this` variables, the total number of variables is 562 for JDepend. If we were to count `this` variables for JDepend, the total number of variables would be 863. Therefore, JHD has 12 times as many variables.

After applying manual refinements, out of 9,827 variables in for JHD, 6202 (63%) variables are mapped to singleton sets, 2101 (21%) of them are `this` variables that are mapped to sets with 2 members and the remaining 16% have sets with two or more members. Having `this` variables that are mapped to the set of {<`this.owned`>, `p`>, <`owner`>, `p`>} is valid and means that inference analysis can pick any of the two qualifiers. Therefore, we treat them just like singleton sets. Having singleton sets of qualifiers for 84% of variables indicates that using only manual refinements, the set-based solution is able to infer valid qualifiers for majority of the variables (RQ0). Based on Table 3, only 14% of the inferred qualifiers are <`shared`, `shared`>. So the majority the inferred qualifiers are of a higher precision (RQ1). There are 8% of inferred qualifiers with actual domains (`owned` or `PD`) as the owning domain, and around half of that are `PD` domains to express logical containment (RQ2).

Table 3. JHD inferred qualifiers in the Manual and the Assisted modes. Numbers in parentheses are percentages.

Mode	**vars**	owned	PD	owner	p	shared	<p>	lent	unique
Manual	9827	425 (4)	389 (4)	7099 (72)	295 (3)	1358 (14)	94 (1)	82 (1)	85 (1)
Assisted	10075	443 (4)	344 (3)	7441 (74)	184 (2)	1352 (13)	94 (2)	154 (1)	63 (1)

Assisted mode. After applying manual refinements, we run OOGRE in the Assisted mode. OOGRE applies 5532 automated refinements on the fields and local variables in the code. Out of those, 2889 refinements are skipped, which is 52% of the automated refinements. Out of 10075 total number of variables for JHD, 6952 (69%) end up with singleton sets. There are 2170 (22%) `this` variables with two qualifiers in their sets. Therefore, 91% of the variables are mapped to valid sets of qualifiers (RQ0). The remainder 9% of variables are mapped to sets with two or more members. Comparing to manual refinements, using Assisted, there are 6% more singleton sets since the inference analysis applies more refinements. Looking at the metrics on the inferred qualifiers (Table 3), there is no significant change in the number of qualifiers of JHD from Manual to Assisted. The reason is more than half of the automated-refinements applied by Assisted are skipped, so Assisted cannot make a significant improvement over Manual. Moreover, using manual refinements, the analysis already applied 42 refinements (inference analysis reaches fixed point 42 times), so most of the source variables for the automated refinements are already mapped to singleton sets and any automated refinement is skipped. It also indicates that using manual refinements only, developers are able to express object hierarchy in the form of Document-View architecture (RQ3).

6 Limitations

Our current analysis suffers from the following limitations.

One ownership parameter. A standard hashtable typically requires two free domain parameters: one for the key objects and one for the value objects. We support one domain parameter `p`, in addition to `owner`, to keep our inference tractable but this reduces expressiveness.

SOD can still express a hashtable by making the hashtable object, the key objects, or the value objects be peers. One workaround is to specialize a hashtable type for a specific key type or value type and reserve the free parameter for the interesting objects.

Fixed domains. By hard-coding the private and public domains to be `owned` and `PD`, the analysis can enumerate and infer them. As a result, developers cannot define multiple private or public domains per class.

Final fields. In SOD, the code can refer to the public domain of an object through a final field `n`, using the construct `n.PD`. If the variable `n` is not final, it may be re-assigned, and the type system may lose track of the relationship between an object and the objects contained in its public domain. As a result, OOGRE may not infer as many objects in `PD`.

7 Related Work

We organize related approaches based on their output and the kind of program analysis they use.

Static analysis/saves qualifiers. Huang and Milanova [9] present an approach to infer OT qualifiers. An interactive approach that requires developers to add qualifiers for a subset of variables. The approach utilizes a set-based solution and uses transfer functions to analyze all the expressions and eliminate invalid qualifiers. The approach infers one ownership parameter and terminates with an error when there is no solution.

Vakilian et al. [16] propose a universal framework that accepts a type system and produces an inference for it, while requiring the Checker framework [5] for the type system. The inference is interactive and inspired by speculative analysis to help developers decide the next steps, by showing the consequences of their decisions ahead of time.

Dietl et al. [6] build a tunable static inference for Generic UT. It works on Java programs with full, partial, or no qualifiers. By traversing AST, the approach generates constraints for variables and solves them by reusing a max-SAT solver, which limits the approach's scalability. The approach utilizes more than one strategy for multiple solutions: adjusting heuristics by changing the weights, or requiring developers to input partial qualifiers.

Aldrich et al. [3] present a type system called AliasJava and an algorithm to infer its qualifiers. AliasJava is similar to OD, but it does not support public domains. To infer alias parameters for each class, the algorithm builds three sets of constraints, equality, *component*, and *instantiation* that guarantee soundness. The algorithm solves the constraints and integrates the result with other qualifiers based on a ranking. However, over 50% of the inferred qualifiers are `shared`. Moreover, the approach infers many alias parameters up to one for each field of one class.

Dymnikov et al. [7] present an ownership inference that infers `owned` qualifiers for the fields of a class. The inference implements some heuristics to infer strictly encapsulated fields in a class, so it is not a sound approach.

Static analysis/visualizes ownership. In Milanova and Vitek [13], a static analysis infers an ownership tree that follows the owner-as-dominator ownership model. First, it creates points-to sets using a points-to analysis. Second, an object graph is created using transfer functions that create edges to indicate the ownership relation between objects. Next, a dominance boundary analysis creates boundaries as subgraphs of the object graph.

Zhu and Liu [19] present a sound and fully-automated constraint-based ownership inference, Cypress that uses an application of linear programming. Cypress generates a visualized hierarchical decomposition of the heap statically. The hierarchy is based on ownership relations between the objects. Cypress follows the "tall and skinny" principle and favors heap decompositions that are taller and skinnier.

Dynamic analysis/saves qualifiers. The dynamic analysis of Werner and Müller [18] analyzes the execution of programs and infers ownership qualifiers from the executions. First, the approach builds a representation of the object store called the Extended Object Graph, which contains all the objects that ever existed in the store and their modification information. Next, it creates the dominator tree, since in UT, all the modifications of an object should be initiated by its owner. Then it resolves the conflicts with UT and harmonizes different instantiations of a class and outputs the qualifiers. The approach is fully-automated, but it is unsound since it uses dynamic analysis.

8 Conclusion

We propose and implement an approach where developers express refinements, while an analysis generates valid ownership type qualifiers in the code, if the code as-written supports the refinement. The list of refinements provides valuable design intent about the code, some of which, such as logical containment and architectural tiers cannot be directly expressed in mainstream programming languages. We believe refinements offer a more intuitive interaction for applying ownership types to existing code than by typing qualifiers. An empirical evaluation of this claim is left to future work.

Acknowledgments

This work was partially funded by the National Security Agency lablet contract #H98230-14-C-0140, and by the Department of Computer Science at Wayne State University.

References

[1] Marwan Abi-Antoun and Jonathan Aldrich. 2009. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

[2] Jonathan Aldrich and Craig Chambers. 2004. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias Annotations for Program Understanding. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

[4] David Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

[5] Werner Dietl, Stephanie Dietzel, Michael Ernst, Kivanç Muslu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *International Conference on Software Engineering (ICSE)*.

[6] Werner Dietl, Michael Ernst, and Peter Müller. 2011. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*.

[7] Constantine Dymnikov, David J Pearce, and Alex Potanin. 2013. OwnKit: Inferring Modularly Checkable Ownership Annotations for Java. In *Australian Software Engineering Conference (ASWEC)*.

[8] Wei Huang, Werner Dietl, Ana Milanova, and Michael Ernst. 2012. Inference and Checking of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)*.

[9] Wei Huang and Ana Milanova. 2011. Towards effective inference and checking of ownership types. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

[10] Ebrahim Khalaj. 2017. *Automated Refinement of Hierarchical Object Graphs*. Ph.D. Dissertation. Wayne State University. http://www.cs.wayne.edu/~mabianto/students/17_khalaj_phd_thesis.pdf.

[11] Ebrahim Khalaj and Marwan Abi-Antoun. 2018. Online appendix: OOGRE. http://www.cs.wayne.edu/~mabianto/oogre/.

[12] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. 2011. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley.

[13] Ana Milanova and Jan Vitek. 2011. Static Dominance Inference. In *International Conference on Objects, Models, Components, Patterns (TOOLS)*.

[14] Jan Schäfer and Arnd Poetzsch-Heffter. 2007. A Parameterized Type System for Simple Loose Ownership Domains. *Journal of Object Technology* 6, 5 (2007).

[15] University of Maryland. 2007. FindBugs: Find Bugs in Java Programs. http://findbugs.sourceforge.net.

[16] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. 2015. Cascade: a universal programmer-assisted type qualifier inference tool. In *International Conference on Software Engineering (ICSE)*.

[17] Radu Vanciu and Marwan Abi-Antoun. 2013. Object Graphs with Ownership Domains: an Empirical Study. Springer LNCS 7850 (2013), 109–155.

[18] Dietl Werner and Peter Müller. 2007. Runtime Universe Type Inference. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

[19] Haitao Steve Zhu and Yu David Liu. 2013. Heap Decomposition Inference with Linear Programming. In *European Conference on Object-Oriented Programming (ECOOP)*.