1-1-2016

Interactive Refinement Of Hierarchical Object Graphs

Ebrahim Khalaj
Wayne State University,

INTERACTIVE REFINEMENT OF HIERARCHICAL OBJECT GRAPHS

by

EBRAHIM KHALAJ

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2016

MAJOR: COMPUTER SCIENCE

Approved By:

Advisor Date

DEDICATION

To my family for their unconditional support and encouragement

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Marwan Abi-Antoun for believing in me. He cared about this work as much as I did, and this is what makes working with him enjoyable. He was of great help with his constructive feedback and comments on my work. When I did not know what to do, he always had the bright ideas for how to move forward.

Then I would like to thank my other thesis committee members for their valuable feedback on my work and their useful suggestions: Dr. Vaclav Rajlich with his attention to detail and encouraging me to focus on more important parts of my work, Dr. Nathan Fisher with his useful suggestions on how to compare my work with other related work.

I would like to thank all alumni and current members of the SoftwarE Visualization and Evolution REsearch (SEVERE) group: Dr. Radu Vanciu for his patience with me as a first year Ph.D. student who had many questions; he always helped me to learn something new; Sumukhi Chandrashekar for being a great lab mate and for sharing useful ideas about my work; Mohammad Anamul Haque for reading some of my writings and giving me useful feedback; Yibin Wang for his meticulous comments on my drafts; Wesley Trescott for his valuable feedback as the first user of my tool. Also, I would like to thank Dr. Andrian Marcus and his students Dr. Laura Moreno and Oscar Chaparro for being great lab mates. Many thanks to the Department of Computer Science and its chair Dr. Loren Schwiebert for their support.

Next, I would like to thank my parents, Hojjatollah and Mehri for their unconditional love and constant support even from far away. Also, many thanks to my brother Dr. Mohammadreza for being a good friend and a great role model, and to my sisters Maryam and Monir for their support and encouragement. I would like to thank my brothers and sister in law for their support, and my beautiful nephews and nieces of whom I have many sweet memories.

I save my warmest thanks to my wife Nona for her love, support and encouragement. She believed in me and encouraged me to continue, even when I was not sure about my journey as a graduate student.

Funding. This work was supported in part by Wayne State University through the Department of Computer Science, and in part by the National Security Agency lablet contract #H98230-14-C-0140.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

In order to evolve object-oriented code, developers must understand its run-time structure in terms of objects and their relations, as well as they must understand the code structure dealing with source files, classes and packages. For object-oriented code, it is hard to understand the run-time structure from looking at the code. Thus, abstractions of the run-time structure such as points-to graphs or abstract object graphs can be highly complementary to diagrams of the code structure such as class diagrams that are readily extracted by many tools. Unfortunately, tools for object graphs are still immature, compared to tools for the code structure. One reason is that extracting these object graphs from code is difficult.

To support understanding the runtime structure of object-oriented systems, several heap abstractions have been proposed. A heap abstraction can statically approximate the runtime heap by building a points-to graph or abstract away one or more snapshots of the runtime heap using graph manipulation [19] or abstraction techniques [17].

Ideally, the abstraction must be sound, i.e., every runtime object that may occur in any execution must have a representative in the abstract object graph, within known limits of unsoundness such as dynamic code loading. To preserve soundness, developers cannot arbitrarily delete objects or relations between objects, because doing so may not account for the impact of any transitive communication, for example.

Also ideally, the abstraction must be driven by developers, otherwise, they may not recognize automatically extracted abstractions. The developers' input, however, must not involve a significant manual annotation burden.

For the past several years, we have been investigating a statically extracted heap abstraction that is a sound, global, hierarchical points-to graph, the Ownership Object Graph[1]. The object graph uses abstraction by ownership hierarchy and by types,

by abstracting each runtime object to a pair consisting of a type and domain, where a domain is a named, conceptual group of objects. Using abstraction by hierarchy, objects that are data structures are at the lower levels compared to objects from the application domain. The object hierarchy cannot be expressed directly in mainstream languages. Instead, they are expressed using additional annotations in the code. If the annotations are consistent with each other and with the code, the object graph abstraction is proven sound [1]. Previous work evaluated if object graphs convey design intent by comparing them to manually drawn diagrams [3]. The object graph abstraction is guided using annotations that implement a type system, Ownership Domains [6], and these annotations are currently being added manually.

Today, the most significant limitation of extracting object graphs is the effort involved in adding annotations, measured at around 1 hour/KLOC [2]. The effort is due to the high overhead associated with inserting local ownership annotations into the code, then refining the annotations both to get them to type-check, and to ensure that the local annotations capture hierarchy in a way such that the extracted object graph reflects a global hierarchy that matches the developers' mental model.

A related issue is bootstrapping the process of extracting an object graph. Today, developers add most of the annotations and fix all of the high-priority warnings before they can extract an initial object graph. Only then, based on visualizing the extracted object graph, they iterate the process of refining the annotations. In other words, to add better annotations, the developers rely on the knowledge provided by the object graph.

Another issue is that the process of refining the extracted object graph is currently somewhat awkward: developers must notice where the object graph does not match their design intent and identify the cases where there are incorrect annotation in the code, rather than a mismatch between the as-implemented system and the developer's mental model. If the issue is in the annotations, the developers have to change the

annotations consistently to reflect the correct design intent, then re-run the static analysis to extract the object graph.

Today, these issues make the process of extracting and refining object graphs tedious and time-consuming, and make object graphs less useful to developers. This thesis addresses these issues by extracting an initial object graph automatically, then allowing developers to directly and interactively refine the extracted object graph to make it convey their design intent, while preserving the object graph soundness.

1.1 Contributions

This thesis contributes What You See Is What You Get (WYSIWYG) developer-driven inference of ownership type qualifiers. Developers preview an abstract object graph, then manipulate or refine it to express their design intent, and in turn, guide the inference analysis. The approach infers qualifiers that type-check and as a result, preserves the soundness of the graph, and the developers do not modify or add qualifiers directly. This thesis describes the inference analysis behind the graphical refinements, focusing on the technical feasibility of inferring ownership qualifiers that typecheck by refining object graphs.

The contributions of this thesis are:

- An approach that supports different types of refinements on an object graph;
- An inference algorithm that infers valid Ownership Domains type qualifiers that satisfy the requested refinements by developers;
- A formal statement of how the inference algorithm preserves the soundness of the refined inferred qualifiers and makes them type-check.
- A small-scale quantitative evaluation that counts the number of attempted and completed manual refinements and the qualifiers of the best and worst results.

1.2 Thesis statement

The thesis statement is:

Using a visual approach that infers Ownership Domains type qualifiers that express both strict encapsulation and logical containment, developers directly and interactively manipulate or refine an abstract object graph by pushing an abstract object underneath another one. If the code as written supports the refinement, developers make an abstract object owned-by another abstract object, make an abstract object part-of another abstract object, or split a merged abstract object in two distinct abstract objects to express their design intent. Behind the scene, an inference algorithm infers valid Ownership Domains type qualifiers that satisfy the requested refinements and type-check.

1.2.1 Hypotheses

We create three hypotheses subordinate to the main thesis statement.

H1. Using a visual approach, developers are able to interactively refine an abstract object graph.

H2. Developers are able to express two types of hierarchy, strict encapsulation and logical containment.

H3. If the code as written supports the requested refinement, the inference analysis infers valid qualifiers that satisfy the requested refinement and type-check.

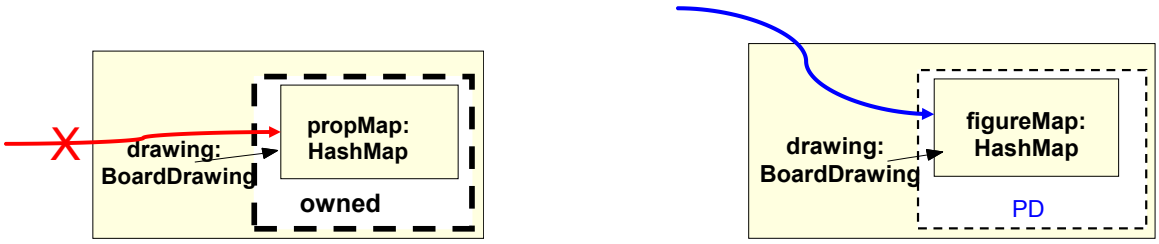
Outline. The rest of this thesis is structured as follows. Chapter 2 provides some motivation for this work and some background on object graphs. Chapter 3 discusses the proposed approach. Chapter 4 positions the work in relation to related ownership type systems. Chapter 5 discusses our inference analysis. Chapter 6 formally describes the inference analysis, Chapter 7 evaluates our approach on small examples. In Chapter 8 we discuss related work. Finally, Chapter 9 discusses implementation

details, limitations, future work and concludes.

Chapter 2: Motivation and Background

We first motivate how we abstract flat object graphs by hierarchy (Section 2.1). Then we give some background on the qualifiers of Ownership Domains (Section 2.2). Next, we motivate why we use object graphs to drive the inference (Section 2.3) instead of asking developers to add qualifiers in the code.

2.1 Motivation: Abstraction by Hierarchy



(A) Strict encapsulation (owned-by). (B) Logical containment (part-of).

FIGURE 2.1: Two ways to create hierarchy in an object graph. Box nesting indicates ownership or containment. The object `propMap` is in the `owned` domain of the object `drawing`, and inaccessible from the outside. The object `figureMap` is inside the domain `PD` and accessible to the outside.

Flat object graphs become very large, and as result, do not convey design intent. One way to make an object graph manageable is to collapse some objects under other objects. One way to do so is to use graph manipulation or transformation [19]. Another way to collapse one object under another is to create an object hierarchy, where one object is the child of another. Instead of letting an object have child objects directly, we introduce an extra level of indirection, a *domain*, which is a named group of objects. So one object has one or more domains and each domain has one or more objects. Two types of domains express two forms of design intent: 1. strict encapsulation; or 2. logical containment;

Strict encapsulation (owned-by). An object `o1` *dominates* object `o2` if all paths from roots in the heap (typically a distinguished object and static fields) to `o2` go through `o1` [8]. In this scenario, `o2` is strictly encapsulated in the abstraction repre-

sented by `o1`, and so we show `o2` as owned-by `o1`, i.e., `o2` is in the private domain `owned` of `o1`. For example, the object `propMap` of type `HashMap` is strictly encapsulated in the object of type `BoardDrawing` (Fig. 2.1a). A private domain has a thick, dashed border. Many ownership type systems enforce this ownership model, also called owner-as-dominator. Moreover, such a property can be inferred fully automatically. An object is either encapsulated or it is not.

Strict encapsulation, however, is too inflexible to make an object graph more hierarchical, after the fact, for code that was written without strict ownership in mind. When developers make an object owned by another, the object becomes inaccessible to other objects that still need to access it. As a result, if developers do not change the code, they have to leave the object at the same hierarchy level as other objects, i.e., more objects will be peers, so they cannot continue making the object graph more hierarchical. So what is really needed is object hierarchy with fewer restrictions to enable developers to make the graph more hierarchical.

Logical containment (part-of). Another object hierarchy that developers express is *logical containment*. Sometimes, one object $o1$ is *conceptually* part-of another object $o2$, even if $o1$ is not dominated or owned by $o2$. For example, we want to consider the `figureMap` of type `HashMap` part-of the `BoardDrawing` object (Fig. 2.1b). A public domain has a thin, dashed border.

In general, it is hard to infer such a relationship from the code, since it is by definition conceptual and reflects design intent rather than any code relationship. Object creation can often hint at logical containment, but not necessarily, as is the case with factory methods. In our approach, developers perform refinements to express this design intent. We also choose an underlying ownership type system that can express logical containment, Ownership Domains [6].

2.2 Background: Ownership Domains

In Ownership Domains, a class can declare one or more domains using the `domain` keyword (Fig. 2.2). Each instance of a class C gets a fresh instance of a domain d declared on the class; for distinct objects n_1 and n_2 of type C, the domains $n_1.d$ and $n_2.d$ are distinct, which means in the object graph there are two distinct domains corresponding $n_1.d$ and $n_2.d$.

In Ownership Domains, a class can take a number of *formal* domain parameters. Here, for simplicity, we allow just two, `owner` and `p`, e.g., `class C<owner, p>` {...}. A type is a class name and two *actual* domains, i.e., `C<p1,q1>`, where `p1` and `q1` are some domains or domain parameters in scope. Given an object that has a type `C<p1,q1>`, the first actual domain `p1` denotes the *owning domain* of the corresponding object. This is why we use the `owner` modifier for the name of the first domain parameter. When used as an actual owning domain on the type of an object o, `owner` means that o is in the same domain as the `this` object. Ownership inference has to infer the *pair* of actual domains `<p1,q1>` for a type, which we call *qualifier*. Given a type qualifier t that is a pair, we access the first actual domain of t with $t.first$, and the second actual domain with $t.second$. For t = `<owned,p>`, $t.first$ returns `owned` and $t.second$ returns `p`.

2.3 Type Qualifiers vs. Object Graphs

To express their design intent, developers can add ownership type qualifiers directly to the code. However, adding qualifiers manually imposes a significant burden since each reference of a non-primitive type in the code needs a qualifier in order to type-check. Therefore, semi-automated or automated approaches for inferring these qualifiers are needed.

It is also hard for developers to directly understand the object structures from

```
1 class C1<owner, p> { // domain parameters
2   private domain owned; // private domain
3   public domain PD; // public domain
4   obj = new C<owner, p>(); // Make peer of this
5   obj = new C<p, p>(); // Place obj inside p
6   obj = new C<owned, p>(); // Make owned-by this
7   obj = new C<PD, p>(); // Make part-of this
8   obj = new C<shared, p>(); // Place inside shared
9 }
```

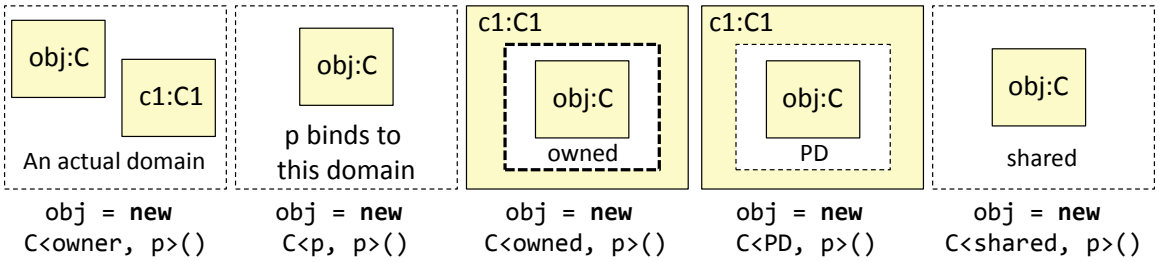


FIGURE 2.2: For the same code, different qualifiers are possible and produce very different object graphs.

looking at code with qualifiers. Almost every single research paper on ownership types uses manually drawn object graphs to explain the object structure that the qualifiers describe or enforce.

To illustrate how hard it is to understand what qualifiers to add to the code, consider an example (Fig. 2.2) with Ownership Domains qualifiers. For the same object creation expression, different qualifiers are possible. At line 4, the object created at the new expression is in the same domain as the object of type `C1` (the declaring class). The object that is created at line 5 is in an actual domain to which the formal domain parameter of `C1` binds. At line 6, the object is in the `owned` domain of `C1`. At line 7, the object is in the `PD` domain of `C1`. The object that is created at line 8 is in the domain `shared`, which is the global context. Each combination of actual domains produces a different object graph (Fig. 2.2). By showing the different object graphs that correspond to the different qualifiers, our WYSIWYG approach makes it easier for developers to choose qualifiers that express their design intent.

Chapter 3: Approach

In this chapter, first, we talk about the proposed approach for the interactive refinement of ownership object graphs (Section 3.1). Then we discuss the refinements we support (Section 3.2), and a more interesting refinement example (Section 3.3).

3.1 Interactive Refinement

Since it is hard to add qualifiers directly to the code without visualizing the object structure being built, we propose a new approach for ownership type inference. Developers use a graphical user interface and interactively manipulate an abstract object graph that is a sound abstraction of the runtime structure. Behind the scenes, an inference analysis infers the corresponding qualifiers that type-check if the code supports this refinement. Otherwise, the inference analysis does not infer any qualifier and leaves the object graph unchanged. Based on the inferred qualifiers, an extraction analysis [1] (not this work's contribution) extracts the updated object graph that the developers manipulate further. The ownership type system provides mathematical guarantees about the soundness of the inferred qualifiers and of the object graph. If the qualifiers type-check, the object graph is sound [1]. In this thesis, we discuss the inference analysis only.

To unclutter an object graph, developers can delete abstract objects, but this makes the object graph unsound, in that it no longer reflects all objects and their communication. Instead, in our approach, developers use abstraction by hierarchy, and push an abstract object they no longer wish to see into a domain of another object.

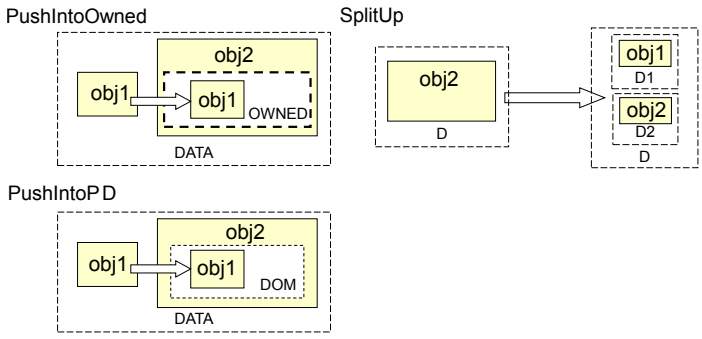


FIGURE 3.1: Three possible refinements, illustrated graphically.

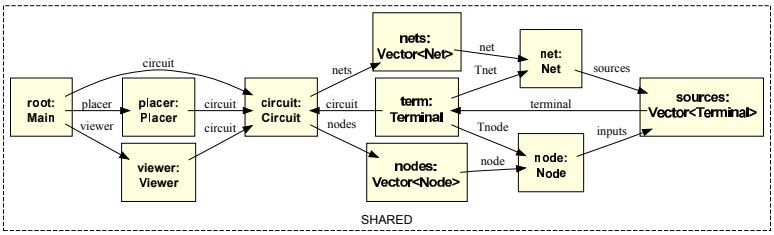
3.2 Supported Refinements

Developers perform the following *refinements* (Fig. 3.1):

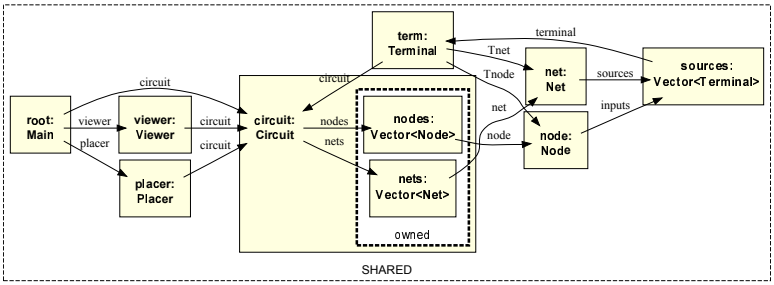
- **PushIntoOwned:** make an abstract object *owned by* another abstract object by pushing it into a private domain;
- **PushIntoPD:** make an abstract object conceptually *part of* another abstract object by pushing it into a public domain;
- **SplitUp:** take one abstract object that merges at least two object creations, and split it into two abstract objects that are in different domains.

3.3 Illustrative Example of the Approach

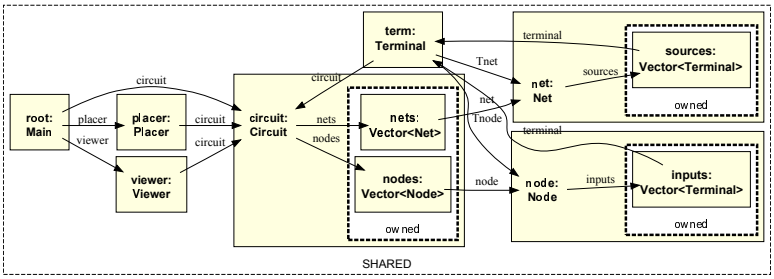
We illustrate the refinement of an object graph using MicroAphyds, a tiny example taken from a larger application, Aphyds [13]. In Fig. 3.2, the edges are points-to edges.



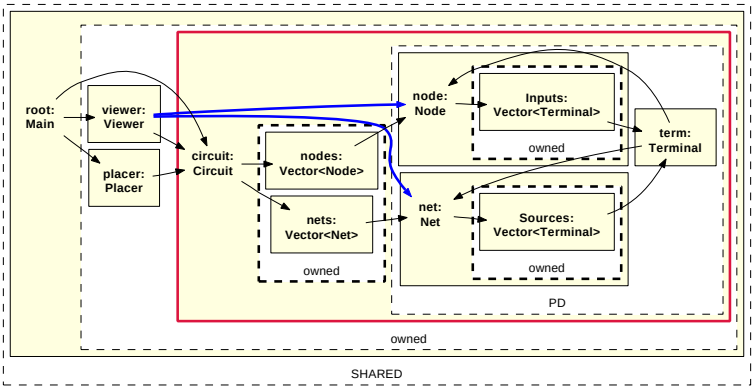
(A) Step 1: The flat graph has all objects in `shared`.



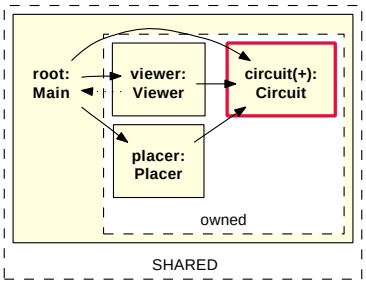
(B) Step 2: Push objects into `owned` of `circuit`.



(C) Step 3: Split abstract objects to avoid excessive merging.



(D) Step 4: Pushing objects into `PD` of `circuit`.



(E) Step 5: Collapsing `circuit`.

FIGURE 3.2: MicroAphyds: refining a hierarchical object graph. Hierarchy enables collapsing several objects underneath the object of type `Circuit`.

Extracting a flat object graph. An initial rough object graph consists of a flat graph that is readily extracted without any developer input, by placing all the abstract objects in the domain `shared` (Fig. 3.2a). While such a flat graph may be useful if it has a small number of abstract objects, flat graphs can become too cluttered for larger systems. In a flat graph, it is hard to find an abstract object or to follow the communication between different abstract objects. In contrast, in a hierarchical graph, developers can collapse objects, and reduce the number of visible objects, as needed.

Another problem of placing objects in the same domain is that it leads the extraction analysis to excessively merge abstract objects of the same type, which makes the graph less precise. For example, in the flat graph, one abstract object of type `Vector<Terminal>` represents two object creation expressions in the code.

Expressing strict encapsulation. Code quality tools such as FindBugs warn when an object returns an alias to a private field to other objects that may mutate it, a code quality issue called "representation exposure". Ownership type qualifiers can express and enforce this design intent and soundly avoid the representation exposure. In MicroAphyds, the developers note that the `Circuit` class has two `Vector` objects, and those objects should not be directly accessible to outside objects, which may mutate them and invalidate data structure invariants. As a result, they push the abstract objects `nodes:Vector<Node>` and `nets:Vector<Net>` into the private domain `owned` of `circuit:Circuit` using two separate PushIntoOwned refinements (Fig. 3.2b). If the code does not suffer from representation exposure, i.e., there is no public method that returns an alias to the fields `nodes` or `nets`, or if it returns a copy or clone of the object, the refinement succeeds. If not, the refinement fails, with a message indicating the expression with the unexpected aliasing. For the refinement to succeed, developers have to fix the code (remove the representation exposure by returning a copy) and re-attempt it.

Splitting abstract objects. The analysis that extracts the object graph merges runtime objects of the same type in the same domain into one abstract object. If an abstract object represents more than one object creation expression in the code, the developers are able to split the abstract object by pushing one of the abstract objects that were merged, in a different domain. In MicroAphyds, the developers split the abstract object of type `Vector<Terminal>` into two abstract objects of the same type and push one of them into `owned` of `node:Node` as one refinement. Using another PushIntoOwned, they push `sources:Vector<Terminal>` into `owned` of `net:Net` (Fig. 3.2c).

Expressing logical containment. Next, developers wish to express that the abstract objects of types `Node`, `Net` and `Terminal` are logically part of `circuit:Circuit`, so they push them into the public domain `PD` of `circuit:Circuit`. They could have equally made them part of some other object, as this is arbitrary design intent.

The key idea is that logically contained objects are still accessible to the objects that have access to the parent. For example, the highlighted edges (appear as thick-/blue) show that the `Viewer` object accesses the objects of type `Node` and `Net` that are part of `Circuit`, i.e., inside its public domain (Fig. 3.2d). With this hierarchy, developers can now collapse the `Circuit` object, as can be seen in Fig 3.2e, thus hiding the objects it contains in its domains, and reducing the number of visible objects in the graph. The (+) on an object label indicates a collapsed object sub-structure.

3.4 Public Domains Create More Hierarchy

In this section, we illustrate by example the benefits of public domains for extracting more hierarchical objects graphs, using the same Aphyds system. We use an experiment where we add alternate sets of qualifiers. One set of qualifiers follows Ownership Types [8], and another follows Ownership Domains with public domains

only. We then compute metrics on the extracted object graphs.

Object Graph Metrics. We compute the following metrics directly on the extracted object graphs:

- **Top-Level Objects (#TLO):** the number of objects in the top-level domain;
- **Objects in PD (#OPD):** the number of objects in a public domain (`PD`);
- **Objects in PrD (#OPrD):** the number of objects in a private domain (`owned`);
- **Object Depth (OD):** the object depth, including the Average (Avg OD), Minimum (Min OD) and Maximum (Max OD);
- **Maximum Depth of Ownership Hierarchy (MXD)**;

Experiment 1: Ownership Types. One can consider Ownership Types to be a subset of Ownership Domains. We modified our implementation to infer the Ownership Types subset of Ownership Domains qualifiers, then extracted the object graph. The graph is very flat (Fig. 3.3). All the following objects are in the same domain: `Placer`, `Viewer`, `Circuit`, `Node`, `Net` and `Terminal`.

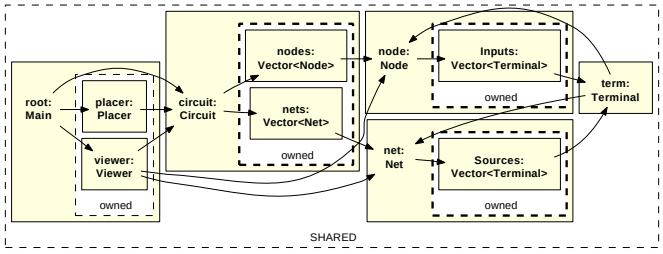


FIGURE 3.3: Aphyds: pure Ownership Types version.

Experiment 2: only public domains. We modified our implementation to infer only public domains, then extracted the object graph. We used heuristics and one PushIntoPD, namely `Net` into `Circuit`. The graph is more hierarchical (Fig. 3.4). The following objects are now children of `Circuit`: `Node`, `Net` and `Terminal`.

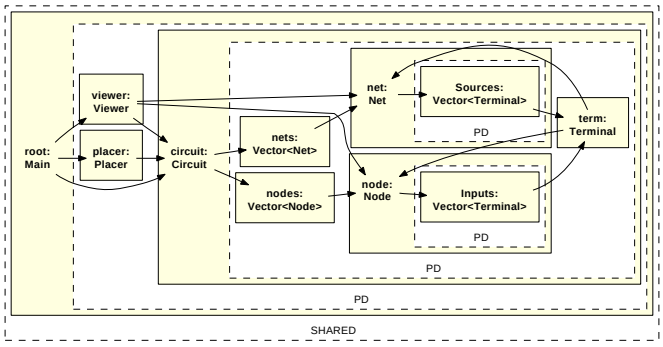


FIGURE 3.4: Aphyds: all PD version.

TABLE 3.1: Metrics on the object graphs.

System	#O	#TLO	#OPD	#OPrD	MXD	AOD	Min OD	Max OD
Ownership Types only	12	6	0	5	3	0.44	0	6
PD only	12	1	10	0	5	0.44	0	5

Metrics. The comparative metrics are in Table 3.1. With pure Ownership Types, MXD is 3. With PD, MXD is 5. In other words, the ownership tree is deeper with public domains. Indeed, public domains create more hierarchy, because putting an object in `owned` restricts its accessibility, so the object has to stay at the same ownership level with more peers.

3.5 Adaptation of a Public Domain

Adding these public domains to the language requires handling additional adaptation cases in the analysis. The qualifier of an expression is the result of adapting its inner qualifier from the view point of its receiver. If a field read, field write or a method invocation expression has a receiver other than `this`, the qualifier of the expression is the result of an adaptation. If an object is trying to access the `PD` domain of another object `n`, the result of adaptation is `n.PD`. If an object accesses its own `PD`, the actual is `this.PD`. Therefore, when accessing another object's `PD`, `this` is substituted with the name of the other object during adaptation. `n.PD` represents the actual from the receiver's viewpoint.

```
1  class C1<owner,p> {
2    public domain PD;
3    C<PD,p> f;
4  }
5  class C2<owner, p> {
6    final C1<owner,p> c1;
7
8    void m() {
9      C<c1.PD,p> c = c1.f;
10   }
11 }
```

FIGURE 3.5: Code to illustrate the adaptation of a public domain.

Fig. 3.5 shows a small example that illustrates adaptation, when the receiver for a field read is not `this`. At line 6, a field `c1` is declared in the class `C2`. Its qualifier is <`owner`, `p`>. At line 9, the field `f` of `c1` is read and assigned to the local variable `c`. In order to find the qualifier of the local variable `c`, which is the qualifier of the field read expression, an adaptation needs to happen. To access the object stored in the field `f`, one should go through `c1:C1`. Therefore, the qualifier of `c` is the adaptation of the qualifier of `f` from the viewpoint of `c1`, as it is the receiver of the field read expression. The qualifier of `f` is <`PD`, `p`> and the qualifier of `c` is <`c1.PD`, `p`>.

Chapter 4: Positioning

In this chapter, we compare the Ownership Domains (OD) type system to two other systems (Section 4.1), then simplify it to make inference tractable (Section 4.2). Then we discuss how our approach applies the framework by Huang et al. [14] (Section 4.3).

4.1 Ownership Domains vs. Other Systems

There are similarities across the three ownership type systems covered in the closely related inference work [14] and this thesis: Ownership Types (OT) [8], Universe Types (UT) [11], and Ownership Domains (OD) that we show in Table 4.1. In the rest of this thesis, we will refer to the type systems by their abbreviated names.

Similarly to OT and UT, OD can express the concept of a strictly encapsulated object, the concept of an object that has the same owning context, and the concept of an object in the global context. Similarly to OT, OD has the notion of an ownership parameter. Compared to OT and UT, OD has the notion of logical containment that is expressed using a public domain and ranks below a strictly encapsulated object and above a peer object (see the last row).

In OT and UT, objects own other objects directly, i.e., the ownership context is an object. In OD, objects do not own other objects directly. Instead, a domain is an explicit, named, ownership context. Explicit contexts are important during the graphical refinement of an object graph. In our approach, developers drag an object and drop it into a named, explicit context. Otherwise, when developers push an object o_1 inside an object o_2, it would be unclear whether o_1 should be owned by o_2, or part of o_2.

This notion of explicit contexts is also useful with public domains. If object o_1 is in a public domain d of o_2, we can refer to the explicit context or domain of o_1 as

TABLE 4.1: Comparison across three ownership type systems. We show the corresponding modifier in the type system or "n/a" if the concept is not available in the type system. The preference/ranking for UT and OT is from Huang et al. [14].

	UT	OT	OD
global owner	`root`	`norep`	`shared`
strict encapsulation	`rep`	`rep`	`owned`
logical containment	n/a	n/a	`n.PD`
same owner as `this`	`peer`	`own`	`owner`
ownership parameter	n/a	`p`	`p`
readonly + pure	`any`	n/a	n/a
preference/ranking	`any` > `rep` > `peer`	`rep` > `own` > `p`	`owned` > `n.PD` > `owner` > `p` > `shared`

$o_2.d$. For example, the iterator of a collection n can be referred to as the object of type `Iterator` inside the domain n.`ITERS`. It makes sense to refer to a public domain of an object o_2.`PD` as an explicit context, but not so for a private domain o_2.`owned`, because the latter is inaccessible from the outside anyway.

In contrast, UT can refer to an object o_1 in some other context but with a reference that cannot be used to mutate the referenced object (the modifier is `any`). As a result, UT requires additional purity qualifiers, which have to be either manually added or inferred using a separate inference analysis. Moreover, the modifier `any` does not provide any information about the actual ownership context of an object. Not knowing the owning context of an object does not suit a visual approach such as ours, since a sound object graph must show each object in its owning domain.

Our previous empirical evaluation using metrics on a corpus of code of 100 KLOC with manually added Ownership Domains qualifiers and their extracted object graphs [22] shows that object hierarchy without encapsulation occurs in practice. In our preliminary testing (Chapter 7), several cases of automated PushIntoPD refinements based on object creation are successful.

4.2 Simple Ownership Domains (SOD)

We simplify the OD type system as follows, and call it Simple Ownership Domains (SOD): 1. a single private domain per class, hard-coded to be `owned`; 2. a single public

domain per class, hard-coded to be `PD`; 3. an implicit domain parameter, `owner`, which is made explicit in the formalization and in the code examples; 4. a single explicit domain parameter per class, hard-coded to be `p`; and 5. default or implicit domain links [6] that make objects in private domains inaccessible to the outside, objects in public domains accessible, and objects in sibling domains accessible to each other. In the rest of this thesis, we use SOD.

Hard-coded domain names. By hard-coding the domain names `owned`, `PD`, and `p`, we cannot define multiple domains per class, or to let the domain name express design intent.

Single parameter. SOD supports a single domain parameter to keep the inference tractable but this reduces expressiveness. Some data structures and programming idioms require more than one explicit ownership parameter. For example, to express a standard Hashtable, two ownership parameters are needed: one for the key objects and one for the value objects [7]. SOD can still express Hashtable with one parameter, by making more objects (the hashtable itself, the key object, or the value object) be peers.

4.3 Relation to Huang et al. [14]

Our inference analysis instantiates the framework of Huang et al. and computes a set-based solution, by starting with sets containing all possible answers and iteratively removing elements that are inconsistent with the typing rules.

4.3.1 Common Definitions

To clarify our contribution in this thesis, we reuse the terminology of Huang et al., as follows:

Variable: Denotes all reference types, i.e., local variable, parameter, return, object

creation, and field types;

Actual Modifier: A member of the set of actual domains or actuals that can be used in SOD, namely `owned`, `n.PD` (`n` can be `this`), `owner`, `p` and `shared`;

Qualifier: A qualifier is a pair of actuals $<p, q>$. The first element, p, is the owning domain and the second element, q, is the actual domain that is supplied for ownership domain parameter;

Maximal Qualifier: A maximal qualifier is the highest ranked qualifier in the set of qualifiers of a variable that type-checks the variable's expressions;

General Qualifier: A high-ranked qualifier that type-checks the variable's expression, but may not express the specific owning domain of an object;

Typing: Given a program and an ownership type system with a set of possible qualifiers, a typing is a mapping from each variable in the program to a qualifier. A valid typing type-checks the program in the type system;

Maximal Typing: A maximal typing is the highest ranked valid typing;

Set Mapping: In a set-based solution, a Set Mapping maps each variable in a program to a set of feasible qualifiers in a type system. One set mapping may contain several valid typings;

Optimality Property: The optimality property holds for a type system and a program if and only if the typing derived from the set-based solution by giving each variable the maximal qualifier from its set, is a valid typing;

Conflict: When assigning the maximal qualifier for each variable does not type-check the program, the expression in the program that does not type-check is a conflict. Conflicts happen when the optimality property does not hold for a program and a type system.

4.3.2 Handling the Lack of Optimality

A key insight 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." Huang et al. show that this property holds for UT, but does not hold for OT.

Unfortunately, this property also does not hold for SOD. Still, we use SOD for the following reasons: (a) We need public domains to get more hierarchical object graphs (See Section 3.4); (b) One could add a maximal modifier to SOD, which is currently lacking, then use it to infer maximal qualifiers. A maximal qualifier that is both the highest ranked and type-checks many expressions is likely to be very general and imprecise, or one that severely restricts what the code can do with a reference, similarly to the `any` modifier in UT that prohibits mutation. Our visual approach, however, requires a precise typing with precise qualifiers rather than general ones, for all the variables, to show each object in the corresponding domain in the object graph, and the edges to or from that object.

Huang et al. support OT, which also lacks this optimality property but they handle it differently. They require that developers guide the inference analysis at certain points. "A statement s is a conflict if it does *not* type check with the *maximal qualifier* derived from the set-based solution. Given a program P, which may be un-annotated 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_0 is reached, where the optimality property holds for P_0. The solver computes a maximal typing for P_0."

In their evaluation, developers provide 2–10 manual annotations per 1 KLOC to

infer OT qualifiers. In contrast, our developers just perform refinements and do not specify qualifiers on variables directly. The optimality property may not hold after any given refinement, but our analysis does not stop and tries to find a valid typing. Our analysis looks for a valid typing from the current qualifiers of each variable. It either infers a valid typing or reports that the refinement is unsupported. In Huang et al., a developer-annotated variable is initialized with a singleton set that contains only the developer-provided qualifier. Similarly to Huang et al., where developers constrain the solution by adding qualifiers to the code, more refinements make more sets of qualifiers to become singletons for more variables. Our preliminary results confirm that after each refinement, there are fewer possible valid typings to find (See Chapter 7).

The value of our WYSIWYG approach, compared to having developers resolve conflicts by adding qualifiers to some expressions, will be evaluated with user studies. We also leave it to future work to seamlessly integrate developer-provided, partial annotations with automatic inference.

4.3.3 Instantiating the Framework

Instantiating the Huang et al. framework for a type system requires the following: (a) the set of possible qualifiers in the type system; (b) the viewpoint adaptation functions, $\triangleright$, which adapt the type of an expression from the viewpoint of its receiver [11]; and (c) type-system specific constraints.

Huang et al. instantiate their framework with UT and OT only. Instantiating their framework to support our graphical refinement approach and the underlying SOD type system has to take into account the following:

• To show each object in a domain in the object graph, our approach needs a precise—rather than a general—typing, which is often the *maximal* typing in Huang et al.;

• We add the public domain `PD` to the initial set of possible qualifiers of each variable, and have fewer restrictions on the *first* and the *second* actual domains of a qualifier t. For example, the qualifier `<PD,owned>` is possible in SOD but impossible in OT (where the actual parameter can never be `rep` due to static visibility constraints). As a result, each variable has a larger initial set of possible qualifiers due to more permutations;

• In SOD, an actual domain can be `n.d` where `n` is the name of an object and `d` is a domain. `n` can be `this` or a `final` field (or a sequence of `final` fields) and `d` can be a private domain `owned` or a public domain `PD`. In contrast, in OT, `n` is always `this` and `d` is always `owned`. For SOD, adaptation has to consider the different values of `n` and `d`.

• Since a qualifier can contain `this`, the adaptation has to be more precise and distinguish between the inner `this` and the outer `this`. To avoid capture during substitution, we rename the inner `this` to `that`, then substitute `that` later on with the corresponding object name `n`;

• There is no subtyping hierarchy between the SOD qualifiers, only equality (similarly to OT);

• There is a different ranking for qualifiers, where `n.PD` is between `owned` and `owner`. Moreover, there is no maximal qualifier that the analysis can always pick from a set of qualifiers for the unique maximal typing, so it has to follow a different strategy to extract a typing from a set mapping;

• We handle type system constraints specific to SOD.

Chapter 5: Set-Based Solution

In this chapter, we give an overview of the inference analysis (Section 5.1). Then we show the pseudo-code for the top-level analysis (Section 5.2). Next, we discuss the ranking of qualifiers (Section 5.3), the trivial qualifiers and the initial object graph (Section 5.4). Then we discuss the default qualifiers (Section 5.5). Next we discuss how a refinement invokes the inference analysis (Section 5.6). In the remainder of this chapter, when we say the analysis for brevity, we mean the inference, rather than the extraction analysis.

5.1 Overview of the Inference Analysis

In this section, we give an informal overview of the analysis, which is a data flow analysis. It builds a Control Flow Graph (CFG) and analyzes all the method declarations of the program. The analysis uses transfer functions for the type of expression being analyzed, namely object creation, assignment, field read, field write and method invocation. In the set-based solution, a Set Mapping (SM), S, maps each variable to a set of qualifiers.

Starting point. In our approach, developers specify the `Main` class of the application to analyze as the starting point. The `Main` class does not declare a domain parameter `p`, and an instance of the class is created in `shared`, as in `new Main<shared>()`. We refer to every class other than the Main class as an application class. Every application class declares a domain parameter `p` and also the local domains `owned` and `PD`.

As the first step, the analysis maps each variable in S to an *initial set of qualifiers* that contains all the possible qualifiers (see Section 5.4). The analysis then saves *trivial qualifiers* (see Section 5.5) for each variable as annotations to the code. The trivial qualifiers are guaranteed to type-check. Based on the trivial qualifiers, the

extraction analysis extracts an initial flat object graph. This may lead to merging abstract objects in the `shared` domain.

Refining the object graph. The developers refine the object graph directly by drag-and-drop of a source object into a domain of a destination object. Each drag-and-drop operation is a refinement.

Inferring qualifiers. A refinement specifies only the owning domain of an abstract object, which can be translated into the *first* element of the qualifier. In contrast, some approaches require the developers to specify the full qualifier at all object creation sites [15] or at a subset thereof [14], which we believe to be a significant burden. Our analysis translates the refinement by changing the *first* element of the qualifier of all the new expressions that the source object traces to. The changed variables are the *target variables* of the refinement. Then, the analysis infers the appropriate *second* element for all the target variables, from all the possible options (`owned`, `PD`, `owner` and `p`). The analysis excludes `shared`, because it is a global context that all the objects can access. For each possible value for the *second* element, the analysis builds a separate S ($S_{\texttt{owned}}$, $S_{\texttt{owner}}$, $S_{\texttt{PD}}$ and $S_{\texttt{p}}$). For S_q, the analysis changes the set of qualifiers of the target variables to be a singleton set, where *first* is dictated by the type of the refinement and *second* is q (see Section 5.6).

Transfer functions. Then, the analysis applies transfer functions on each S and each expression in the program. Each transfer function takes an S, an expression and produces an output S'. A transfer function removes the qualifiers that cannot be in a valid typing (see Section 6.3). The transfer functions run until a fixed point when the sets of qualifiers of variables no longer change. At the fixed point, an S contains only qualifiers of valid typings for each variable or the set of qualifiers for one or more variables is empty. If an S contains an empty set of qualifiers for one or more variable, the S is discarded and not used any further, since it cannot be used to save qualifiers that type-check.

Respect previous refinements. The analysis respects the previous refinements by preserving the set of qualifiers of their target variables. All other variables, which are not target variable of any refinement, are mapped to the initial set of qualifiers, to impose the fewest restrictions.

No solution. After running the transfer functions, all the S's may get discarded. In that case, then the refinement is unsupported, and the analysis informs the developers. For an unsupported refinement, the analysis does not save a new set of qualifiers, so the object graph stays the same.

Multiple solutions. After running the transfer functions, there may be more than one valid solution for the refinement (multiple S's). Each S represents a solution, so the analysis must select one to continue. To select the S, the analysis uses a strategy that prefers the solution S_q where q is `p`, `PD`, `owner` and `owned` in this order. The analysis prefers a more flexible solution over the others. `p` is a domain parameter that can be bound to any domain, so that is the most flexible solution. `PD` is more flexible than `owner` and `owned`, since an object that is in the public domain of an object can be accessed by other objects. `owned` is the least flexible one, since an object in `owned` can be accessed only by the object that declares the domain, or by objects in its sibling `PD` domain. Before we proceed to any refinement, the analysis selects the current S based on the above strategy.

Check overriding. The analysis checks that S respects the rules for method *overriding*, namely that an overriding method has the same qualifiers as the overridden method, for its parameters and for its return type, respectively.

Type-check S. At this point, each variable in the current S maps to a set of qualifiers. Therefore, to be able to pick between them, the analysis defines a ranking between the actuals that can be extended to rank qualifiers (see Section 5.3). The analysis cannot simply save the highest ranked qualifier for each variable. The highest ranked, or maximal, qualifier is the best qualifier for a variable, but there is no

guarantee that the maximal qualifier for each variable type-checks the program (the optimality property, see Section 6.5). In this thesis, we assert that the optimality property may not hold for the un-annotated programs and SOD. To ensure that the maximal qualifiers type-check the program, the analysis does a separate step called *type-checking*. During type-checking, the analysis extracts a typing T from S. To do that, the analysis applies a function f on the set of qualifiers of each variable x in S. The function f initially uses the max function, which picks the highest ranked, i.e., maximal, qualifier in the set of qualifiers of x. Therefore, in the typing T, each variable x receives its maximal qualifier from S. The typing T is a valid typing if, for each variable x, $T[x]$ type-checks the program. If T is valid, it is saved to the code.

Searching for a valid typing. If the analysis cannot find a valid typing during type-checking, it searches for a valid typing T in the current set mapping S by changing the function f. To obtain the qualifier of a variable x in a typing T, for each x in S, the analysis applies f on $S[x]$, which now uses a $next$ function on the set of qualifiers of x. The $next$ function picks a single qualifier from $S[x]$. For any other variable, the function f uses the max function on its set of qualifiers. This step continues until for all each variable x, $T[x]$ is defined. If there is no valid typing T, the refinement is unsupported. If there is more than one valid typing T, the analysis must pick between them as we discuss next.

Finding the best valid typing based on metrics. To select between all the valid typings, the analysis ranks them by computing metrics on the qualifiers in each T. To rank an entire T, the metrics follow the same ranking as for the qualifiers in S (see Section 5.3). A T that contains higher ranked qualifiers also ranks higher. The metrics compute the percentages of actuals in the qualifiers in T. The percentages approximate how hierarchical of an object graph the qualifiers will produce. For example, if many qualifiers contain domain parameters (`owner` or maybe `p`), they will produce objects that are peer in the graph. Alternatively, if many qualifiers are local

domains (`owned` or `PD`), the graph will be more hierarchical. Since extracting the object graph is another whole-program analysis that runs until a fixed point, it is also time consuming to extract object graphs for more than one T. So the metrics are a lightweight strategy to pick the T to save, as opposed to extracting graphs and computing metrics on the graphs. Finally, the analysis picks the highest ranked T and saves it.

Heuristics to increase automation. We define heuristics to automatically suggest refinements based on structural properties in the code. To suggest a PushIntoOwned refinement, an Abstract Syntax Tree visitor identifies variables that are potentially strictly encapsulated, using the standard visibility modifiers. Moreover, when object $o1$ creates another object $o2$, it is likely that $o2$ is conceptually part of $o1$. Therefore, another heuristic suggests a PushIntoPD refinement, namely pushing `o2` into the public domain `PD` of `o1`. If at least a few heuristics succeed, the developers start from an object graph that already has some hierarchy rather than a flat graph and refine it further.

The heuristics can be unsound since they do not perform an alias analysis. So every suggested refinement must be fully validated, as if it were a manual refinement, following the steps above. For example, the heuristics may suggest a PushIntoOwned, but that object may not be strictly encapsulated. This arises, for instance, when the corresponding variable has the `private` visibility, but a `public` method returns an alias to the object, either directly, or through a series of assignments. In that case, the PushIntoOwned refinement will be unsupported. If the developers want that refinement to succeed, they have to manually change the code to fix the representation exposure—return a copy of the object rather than an alias, then re-apply the refinement.

5.2 Top-Level Analysis

We show the pseudo-code (Algorithm 1) for the analysis. The signatures of the functions that are called from the top-level analysis are shown in Algorithm 2. The pseudo-code starts with the initial S, where all the variables are mapped to the initial set of qualifiers. When the developers do a refinement, the analysis changes the qualifiers of the target variables of the refinement and infers the qualifiers for the other variables by running transfer functions. The extraction analysis extracts a new object graph based on the new set of qualifiers inferred by the analysis.

5.3 Ranking of Qualifiers

We define a ranking between the qualifiers that the analysis may infer. The criterion for the ranking is to make the object graph more hierarchical. First we define a ranking between all the actuals of SOD (the domains and domain parameters). So, `owned` is the highest ranked actual, since it creates hierarchy and an object in `owned` is strictly encapsulated. The next ranked domain is `n.PD`, which also creates hierarchy, but is less restrictive. Every object in `owned` can be in `PD`, but the reverse does not hold. In `n.PD`, `n` can be `this`. The third rank is `owner`, which is a domain parameter and makes objects peers. The next ranked domain is `p`, which is also a domain parameter, and it can bind to any domain. The lowest ranked is `shared`, the trivial modifier, used to obtain the initial flat graph.

$$\texttt{owned} > \texttt{n.PD} > \texttt{owner} > \texttt{p} > \texttt{shared}$$

The ranking is extended to the qualifiers. To determine the ranking of a qualifier, the analysis first considers the *first* element, then the *second* element of a qualifier.

Algorithm 1 Pseudo-code for the top-level analysis.

```
function runAnalysis(E e_root, CT _CT) // initial run
    initialSM ← initializeSM() // with initial qualifier set
    saveTrivialQualifiers()
    G ← extractObjectGraph(e_root, _CT)
function runAnalysis(E e_root, CT _CT, OGraph G, Refinement ref) // given G
and ref
    currentSM ← getPreviousSM() // result of the previous ref is the new
currentSM
    currentSM ← resetSM() // preserve previous refs
    Set<S> sm_x // set of SMs with different second
    Set<T> typings // set of valid typings
    sm_x ← applyRefinement(ref, currentSM) // attempt ref
    for all s_i ∈ sm_x do
        if hasEmptySet(s_i) then
            sm_x.remove(s_i) // remove discarded s_i
    currentSM ← pickSM(sm_x) // multiple solutions strategy to set currentSM
    currentSM ← postProcess(currentSM)
    T typing ← typeCheck(currentSM)
    if RunTFs(typing) != null then
        saveQualifiers(typing)
    else
        typings ← findValidTypings(currentSM)
    typingToSave ← applyMetrics(typings) // pick a typing to save
    saveQualifiers(typingToSave)
    G ← extractObjectGraph(e_root, _CT)
function applyHeuristics(E e_root, CT _CT, G)
    if inferHeuristics then
        targetVars ← runVisitors() // finding target vars
        for all var: targetVars do
            heu ← new Heuristics(var)
            runAnalysis(e_root, _CT, G, heu)
function applyRefinement(ref, sm)
    Set<S> smSet
    for all q ∈ {owned, PD, owner, p} do
        sm_q ← sm.clone()
        for all var ∈ sm_q do
            if var is a target variable then
                sm_q[var] ← {<ref,q>} // modify the set
        smSet.add(sm_q)
    for all sm ∈ smSet do
        sm' ← runTFs(sm) // run TFs until fixed point
        smSet.replace(sm,sm') // replace old S with the new S
    return smSet
```

Algorithm 2 Signature of the functions that are called in the top-level analysis.

```
function RESETSM : S // Respect the previous refinements
   // Return:  a reset SM in which all the variables that are not target
variables of the previous refinement are mapped to the initial set of qualifiers
function RUNTFS(S sm) :  S //Run the TFs on a set mapping
   // Input:  a SM
   // Return:  a SM that reflects the set of qualifiers of each variable at
fixed point
function RUNTFS(T typing) :  T // Run the TFs on a typing
   // Input:  a typing
   // Return:  the input typing if it is valid, otherwise returns null
function PICKSM(Set<S> setSs) :  S
   // Input:  a set of SMs
   // Return:  a SM that is the preferred one based on being more reusable
function POSTPROCESS(S s) :  S
   // Input:  a SM
   // Return:  the SM, after post-processing
   sm = checkOverriding(s)
function CHECKOVERRIDING(S s) :  S
   // Input:  a SM
   // Return:  a SM where parameters and return of overridden and overriding
methods have the same set of qualifiers
function TYPECHECK(S s) :  T
   // Input:  a SM
   // Return:  a typing, discarded if it is not a valid type-checked
function FINDVALIDTYPINGS(S s) :  Set<T>
   // Input:  a SM
   // Return:  a set of valid typings
function APPLYMETRICS(Set<T> tSet) :  T
   // Input:  a set of typings
   // Return:  the preferred typing, based on ranking the typings
```

Therefore, to compare two qualifiers, the one that is higher ranked has a higher ranked *first*. If two qualifiers have the same *first*, then the one that has a higher ranked *second* is ranked higher.

5.4 Initial Set Mapping

In the initial S, the analysis maps each variable to an initial set of qualifiers, which contains all the possible qualifiers for a variable, by considering the type system constraints related to SOD. The initial set of qualifiers of a variable in application classes contains 18 members, which are created by the actuals `owned`, `PD`, `owner`, `p`,

and `shared`, as follows:

{<this.owned, this.PD>, <this.owned, owner>, <this.owned, p>, <this.owned, shared>,
<this.PD, this.PD>, <this.PD, owner>, <this.PD, p>, <this.PD, shared>, <owner, this.PD>,
<owner, owner>, <owner, p>, <owner, shared>, <p, this.PD>, <p, owner>, <p, p>, <p, shared>,
<shared, shared>}

In the initial set of qualifiers, we have `this.owned` as the *second* element of a qualifier, if its *first* element is `this.owned`. The reason is that `owned` is the private domain of current object, and using it as a domain parameter is incorrect, since it is inaccessible to the objects that are not in the peer `PD` domain. Moreover, if the *first* element of a qualifier is `shared`, then the *second* element must be `shared`, because all the objects can access an object in `shared`.

For a variable in the `Main` class, the initial set of qualifiers is smaller, because `Main` does not declare the domain parameter `p`. Moreover, the owning domain of the root object is `shared`, so the domain parameter `owner` binds to the domain `shared`. Therefore, there is no need to have qualifiers containing `owner`. The initial set of qualifiers for each variable in the `Main` class is:

{<this.owned, this.owned>, <this.owned, this.PD>, <this.owned, shared>,
<this.PD, this.owned>, <this.PD, this.PD>, <this.PD, shared>, <shared, shared>}

The domain `owned` of `Main` does not have the properties of a private domain. So, the qualifier <this.PD, this.owned> is correct, since we want the root object to describe two top-level domains, to express the design intent of a two-tiered design.

The root object is in `shared`, and there may be other objects in `shared` that may access objects in its `owned`. Therefore, we treat `owned` in `Main` as a public domain, while keeping the other simplifications to OD that we discussed earlier.

5.5 Trivial Qualifiers; Initial Object Graphs

To extract an initial object graph, we need trivial qualifiers that are easy to compute without running the transfer functions and that are guaranteed to type-check. The analysis saves the trivial qualifiers to the code, and the extraction analysis extracts the initial graph based on them. For the variables in the `Main` class, the trivial qualifier is <`shared`, `shared`>, since all the objects in the initial object graph should be in `shared`. The analysis assigns <`p`, `p`> for the variables in each application class where `p` is the ownership domain parameter of that class. Alternatively, we can use <`owner`, `owner`> everywhere. This initial object graph is flat. If some heuristics succeed, however, developers rarely see this flat graph.

5.6 Applying Refinements

There are three refinements that developers use to refine the object graph: PushIntoOwned, PushIntoPD and SplitUp. The first two have similarities so we generalize them into a PushIntoX, where X can be `owned` or `PD`.

Each refinement operates on an OGraph G, and has a source OObject O_{src} and a destination OObject O_{dst} (Fig. 5.1). The detailed representation of an OObject is used only by the extraction analysis [1] and is not needed here. The analysis translates the requested refinement in terms of variables, changes the set of qualifiers for the target variables in S, and runs the transfer functions to infer changes to the qualifiers of other variables.

PushIntoX. The PushIntoX (Fig. 5.2, rule R-PIX) refinement pushes a source object

$G \in$ OGraph
$D \in$ ODomain
$O \in$ OObject

FIGURE 5.1: Data type declarations for the object graph.

O_{src} into the X domain of a destination object O_{dst}. The analysis finds the target variables that O_{src} traces to, $\overline{x}$. Then it creates four instances of S, $S_{\texttt{owned}}$, $S_{\texttt{PD}}$, $S_{\texttt{owner}}$ and $S_{\texttt{p}}$. The set of qualifiers of each changed variable is modified to have one member in which the *first* element is X and the *second* element is the same as the subscript of the corresponding S. For example, in $S_{\texttt{PD}}$, for a PushIntoOwned, the set of qualifiers of a target variable is {<this.owned,this.PD>}. For a PushIntoPD, it is {<this.PD,this.PD>}. The analysis excludes `shared` and does not create $S_{\texttt{shared}}$, since `shared` is a global domain, and all objects can access it. Moreover, if a target variable is in the `Main` class, then the analysis creates only two instances of S, $S_{\texttt{owned}}$ and $S_{\texttt{PD}}$. There is no domain parameter `p` in the `Main` class, and `owner` is the same as `shared` in `Main`, so there is no need to create $S_{\texttt{owner}}$ and $S_{\texttt{p}}$. Using the auxiliary judgement *mdbody*(), the analysis accesses the body of a method declaration (see Section **??**). The analysis runs the transfer functions (highlighted in the rule) on each created S_q and all the expressions of the program to validate the changes and infer the other changes.

SplitUp. In the extracted object graph, objects of the same type and in the same domain get merged in one abstract object. In a flat object graph where all the objects are in `shared`, an abstract object may merge many object creation expressions in the code. The developers may want to split the abstract object into different abstract objects in different domains. To do so, they may select one specific object creation expression and push the object that traces to that expression into another domain D_{dst} of another object, O_{dst}.

In doing so, they invoke a SplitUp (Fig. 5.2, rule R-SPU). The analysis modifies the

set of qualifiers of the selected variable (x_{src}) only to be a singleton set, then it creates four instances of S, $S_{\texttt{owned}}$, $S_{\texttt{PD}}$, $S_{\texttt{owner}}$, and $S_{\texttt{p}}$. It is important for the analysis to not change the set of qualifiers of the variables of the other object creation expressions of the same type, because the goal of split up is to put the abstract objects that trace to them in different domains. In the modified set, the *first* of each qualifier, X can be `owned` or `PD` depending on the destination domain of the refinement. The *second*, q can have four possible values for an actual (`owned`, `PD`, `owner` and `p`).

R-PIX

$$\frac{\begin{array}{c} O_{src} \in G \qquad O_{dst} \in G \\ \overline{x} = getVars(O_{src}) \qquad \forall x_i \in \overline{x} \qquad Q_i = S[x_i] \\ \forall q \in \{\texttt{this.owned}, \texttt{this.PD}, \texttt{owner}, \texttt{p}\} \qquad \exists\, S_q \; s.t. \; Q'_i = \{<X, q>\} \in S_q \\ (x_i \rightarrow Q'_i)S_q \\ \forall C \in CT, md \in C, e \in md, \quad \Gamma; S_q; n_{this} \vdash e, S'_q \end{array}}{S \xrightarrow{PushIntoX(G, O_{src}, O_{dst}, X)} S'_x}$$

R-SPU

$$\frac{\begin{array}{c} Q_{x_{src}} = S[x_{src}] \qquad X = getDomain(O_{dst}, D_{dst}) \\ \forall q \in \{\texttt{this.owned}, \texttt{this.PD}, \texttt{owner}, \texttt{p}\} \\ \exists\, S_q \; s.t. \; Q'_{x_{src}} = \{<X, q>\} \in S_q \\ (x_{src} \rightarrow Q'_{x_{src}})S_q \\ \forall C \in CT, md \in C, e \in md, \quad \Gamma; S_q; n_{this} \vdash e, \; S'_q \end{array}}{S \xrightarrow{SplitUp(G, x_{src}, O_{dst}, D_{dst})} S'_x}$$

FIGURE 5.2: PushIntoX: for PushIntoOwned, X=`this.owned`, and for PushIntoPD, X=`this.PD`. SplitUp splits and pushes the source variable x_{src} to the X domain of the destination object (X=`this.owned` or X=`this.PD`).

Chapter 6: Formalization

First, we show the abstract syntax that the analysis is based on (Section 6.1). Next, we formalize the adaptation functions for the SOD qualifiers (Section 6.2) and then discuss the transfer functions that analyze the different types of expressions (Section 6.3). Then we discuss SOD type system constraints (Section 6.4), properties of set-based solution (Section 6.5) and finding a typing in a set mapping (Section 6.6).

Assumptions. The inference runs, after the fact, on an existing Java-like program that type-checks, and inserts the ownership type qualifiers. In a formalization of SOD, in contrast to a formalization of a Java-like language, a type $T = C<p, q>$ has two orthogonal components: the class name C and the ownership type qualifier $<p, q>$. Please note we overload T to mean either a typing or a type. The meaning should be clear from the context. Similarly to Huang et al., we treat the ownership type system as orthogonal to or independent from the Java type system. As a result, the inference rules for the transfer functions (Fig. 6.5) do not include the Java sub-typing checks. Those are in the type-checking rules (Fig. 6.6). Running our inference analysis on a Java-like program that does not type-check may lead to undefined inference results.

6.1 Abstract Syntax

We formalize our analysis by adapting Featherweight Domain Java (FDJ), which models a core of a Java-like language with Ownership Domains [6]. 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 4.2).

In our abstract syntax (Fig. 6.1), C ranges over class names; T ranges over types;

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

FIGURE 6.1: Abstract syntax for SOD, adapted from Featherweight Domain Java (FDJ) [6].

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 T_{this} used to refer to the receiver of a method invocation, field read or field write; m ranges over method names; p and q range over formal domain parameters, actual domains, or the global domain `shared`, i.e., all the possible values for the actuals used in SOD; 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; and S defines a map from each variable to a set of qualifiers. A qualifier consists an owning domain and a domain parameter, $<p, q>$; $S[x]$ denotes reading the set of qualifiers for x in S; $S' = [x \mapsto Q]S$ 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 transfer functions and after doing 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 of adaptation.

6.2 Adaptation Cases

The qualifier of an expression is the result of an adaptation when the receiver of the expression is not `this`. In each adaptation case, 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}.

$$t_{rcv} \triangleright t_{in} = t_{out}$$

In Fig. 6.2, we include adaptation cases for `owned`, `owner`, `p` and `shared`, which are similar to the cases in Huang and Milanova [15] for OT, and `rep`, `own` and `p` and `norep`, respectively.

$$\frac{t_1 = <\texttt{owner}, \texttt{owner}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <p_0, p_0>} \text{ ADAPT-O-O}$$

$$\frac{t_1 = <\texttt{owner}, \texttt{p}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <p_0, q_0>} \text{ ADAPT-O-A}$$

$$\frac{t_1 = <\texttt{p}, \texttt{p}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <q_0, q_0>} \text{ ADAPT-A-A}$$

$$\frac{t_1 = <\texttt{owner}, \texttt{shared}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <p_0, \texttt{shared}>} \text{ ADAPT-O-S}$$

$$\frac{t_1 = <\texttt{p}, \texttt{shared}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <q_0, \texttt{shared}>} \text{ ADAPT-A-S}$$

$$\frac{t_1 = <\texttt{shared}, \texttt{shared}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <\texttt{shared}, \texttt{shared}>} \text{ ADAPT-S-S}$$

FIGURE 6.2: Adaptation cases for `owner`, `p` and `shared`.

An inner qualifier can be the qualifier of a field, a method parameter or a method return. Qualifiers that contain `this.PD` or `n.PD` can occur as the inner qualifier. Therefore, the analysis handles them with an adaptation case. We show the general case for handling `this.PD` as the inner qualifier in Fig. 6.4. t_1 is the inner qualifier that contains `PD`. For t_1, we substitute `this` with `that`, since t_1 is the inner qualifier, and the corresponding variable is declared in another class. Later on, in the transfer functions, `that` is substituted with `this` again. t_2 is the qualifier of the receiver of

the expression and it can be any qualifier, so we show it as $<p_0, q_0>$. The result of adaptation, t_3 is based on t_1 and t_2. Writing this in inference rule format leads to near identical rules. Instead, we use this tabular form to show t_1, t_3 and the name of the individual cases in Fig. 6.3. Moreover, We do not have any restriction over the values of *first* and *second* of a qualifier, so we add an extra case for adaptation where the inner qualifier is <p, owner>.

For example, in the rule ADAPT-D-D1, t_1 has that.PD as its *first* and *second*. Therefore, independently of the elements of t_2, t_3 is <n.PD, n.PD>, and n is a final field of the same type as the receiver (C_2) declared in current class. In the other rule, ADAPT-D-O1, the *second* element of t_1 is owner, so in the result qualifier, the *first* element of t_2 is selected as *second*, and t_3 is $<$n.PD, $p_0>$.

The judgement form for adaptation is as follows:

$$\Gamma; n_{this}; n_{rcv} \vdash <p_x,\ q_x> \triangleright <p_y,\ q_y> = <p_z,\ q_z>$$

t_1	t_3	rule name
<p, owner>	$<q_0, p_0>$	ADAPT-A-O
<that.PD, that.PD>	$<n$.PD, n.PD>	ADAPT-D-D
<that.PD, owner>	$<n$.PD, $p_0>$	ADAPT-D-O
<that.PD, p>	$<n$.PD, $q_o>$	ADAPT-D-A
<that.PD, shared>	$<n$.PD, shared>	ADAPT-D-S
<owner, that.PD>	$<p_0, n$.PD>	ADAPT-O-D
<p, that.PD>	$<q_0, n$.PD>	ADAPT-A-D
<shared, that.PD>	<shared, n.PD>	ADAPT-S-D

FIGURE 6.3: Different adaptation cases of PD. t_1 is the inner qualifier and t_3 is the result qualifier.

$$\frac{\begin{array}{l}\text{ADAPT-D-X}\\ \Gamma; n_{this} \vdash n_{this} : C_{this}<p_{this}, q_{this}> \\ \qquad t_2 = <p_0, q_0> \qquad final(n)\end{array}}{\Gamma; n_{this}; n \vdash t_2 \triangleright t_1 = t_3}$$

FIGURE 6.4: General rule for the adaptation of n.PD. We show a case analysis for t_1 and t_3 in Fig. 6.3. p_0 and q_0 can be of the form n'.PD.

Set-level adaptation. In the set-based solution, in order to handle all the possible adaptation cases, each transfer function uses three types of adaptation functions that

work 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:

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

6.3 Transfer Functions

In this section, we formalize our transfer functions, some of which generalize the transfer functions in Huang et al. A transfer function accepts an expression and an S. It extracts the variables included in the expression and accesses the set of qualifiers of the extracted variables in S. By intersecting the sets of qualifiers of different variables, a transfer function removes the invalid qualifiers from the set of qualifiers of the variables of the expression. Then it updates the sets of qualifiers of the corresponding variables in S and creates S'. Fig. 6.5 shows the inference rules for each transfer function. We reuse the transfer functions that do not require adaption. For the transfer functions that require adaptation, we handle qualifiers that contain `n`. We visually highlight the key differences in the extended rules. The judgement form for the transfer function over an expression e is as follows:

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

Object creation expression. The rule TF-NEW transfers over a new expression that consists of a left-hand side variable x and a call to the constructor of the class C. The qualifier $<p, q>$ contains the actual owning domain and actual domain parameter, and together with the class C, forms the type of the object being created. The rule intersects the set of qualifiers of x with the qualifier $<p, q>$, which means the qualifier of x is $<p, q>$. At the end, the rule creates S' by updating the set of qualifiers of x in S.

Assignment expression. The rule TF-ASSIGN extracts the set of qualifiers of the left-hand side (x) and right-hand side (y) variables and intersects them. Then, it updates the sets of qualifiers of both variables in S and creates S'.

Field read and write expressions. For field read and write expressions (the rule TF-FIELDRW), there is one transfer function. The rule extracts all the variables in the expression and finds their set of qualifiers in S. First, the rule substitutes `this` with `that` for the qualifiers of the field f, since f is declared in the class of the receiver x. To compute the updated set of qualifiers for y, ADAPT-OUT finds a set of qualifiers, Q_o. For each qualifier t_o in Q_o, if t_o contains x.`PD` as *first* element, or *second* element or both, and if there is a qualifier t in set of qualifiers of y ($S[y]$) where instead of x.`PD`, t contains `this`.`PD`, the rule substitutes `this` with x for t. Then the rule intersects Q_o with the set of qualifiers of y, and the result is the new set of qualifiers for y, Q_y. To compute the new set of qualifiers of the field f, ADAPT-IN finds a set of qualifiers, Q_i using Q_y. Then, the rule intersects Q_i with the set of qualifiers of f. The result is Q_f, which is the new set of qualifiers for f. Next, for the receiver x, ADAPT-RCV finds a set of qualifiers using Q_y and Q_f, which is Q_r. The

rule intersects Q_r with the set of qualifiers of x. The resulting set is the new set of qualifiers for x, named Q_x. In Q_f, the rule substitutes `this` with `that` and updates the set of qualifiers of the variables in S to generate S'.

Field read and write expressions with `this` as the receiver. When the receiver of a field read or a field write expression is `this`, there is no need for adaptation. Therefore, the rule TF-ThisFieldRW extracts the left-hand side variable for field read or the right-hand side variable for the field write (y), along with the field variable, and finds their set of qualifiers in S. Then, it intersects the sets of qualifiers of the left-hand or the right-hand sides with the set of qualifiers of the field. The rule creates S' by updating the set of qualifiers of the variables in S.

Method invocation expression. In the input expression of the rule TF-Invk, x is the left-hand side variable, and y is the receiver. y_m represents the return of a method as a reference type that needs a qualifier. The variable z is the argument of the method invocation. By calling the *mdbody*() auxiliary judgement (see Section **??**), the rule extracts x_m that is the formal method parameter. First, the rule substitutes `this` with `that` in the sets the qualifiers of x_m and y_m, since they declared in the class of the receiver y. The rule does Adapt-Out using the sets of qualifiers of y and x_m and the resulting set is $Q1_o$. For each qualifier $t1_o$ in $Q1_o$, if $t1_o$ contains y.`PD` as *first* element, or *second* element or both, and 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 rule does the same and $Q2_o$ is the result of Adapt-Out using the sets of qualifiers of y and y_m. Again, the rule substitutes `this` with y for each qualifier $t2$ in $S[x]$, when there is a corresponding qualifier $t2_o$ in $Q2_o$ that contains y.`PD`. By intersecting $Q1_o$ with $S[z]$ and $Q2_o$ with $S[x]$, the rules 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 finds 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, which is Q_{x_m}. Again, by applying Adapt-In

on Q_x and $S[y]$, and intersecting the result with $S[y_m]$, the rule finds the new set of qualifiers of y_m, which is Q_{y_m}. For the receiver y, the rule does ADAPT-RCV and finds the result using Q_z, Q_f, Q_x and Q_m. Then, it intersects the result of ADAPT-RCV with the set of qualifiers of y and gets Q_y, the new set of qualifiers for the receiver y. For the x_m and y_m variables, `that` is substituted with `this` in the resulting sets of qualifiers. Then, the rule outputs S' by updating the set of qualifiers of each variable in S.

Method invocation expression with `this` as the receiver. When the receiver of a method invocation is `this`, there is no need for adaptation. Therefore, the rule TF-THISINVK intersects the set of qualifiers of the formal method parameter with the actual method argument and updates the sets of the corresponding variables with the resulting set in S. Also, it intersects the set of qualifiers of the left-hand side and the method return variable and updates their sets of qualifiers in S with the resulting sets and creates S'.

6.4 SOD Type System Constraints

In this section we talk about typing rules that type-check a program in SOD. Typing rules work at the level of expressions. Next, we introduce some rules that work at the level of methods and classes.

6.4.1 Typing Rules

We adapt the typing rules for SOD to this framework (Fig. 6.6). We expand the rules from Ownership Domains [6] to include special cases for when the receiver is `this`, as for the transfer functions. Most crucially, we adapt the rules to use viewpoint adaptation instead of substitution of formals to actuals in FDJ [6]. In SOD, there is no subtyping between qualifiers, just qualifier equality. Also, SOD imposes its

TF-NEW
$$\frac{S' = [x \to (S[x] \cap \{<p, q>\})]S}{\Gamma; S; n_{this} \vdash x = \texttt{new}\ C<p, q>(),\ S'}$$

TF-ASSIGN
$$\frac{S' = [x \to (S[x] \cap S[y]), y \to (S[x] \cap S[y])]S}{\Gamma; S; n_{this} \vdash x = y,\ S'}$$

TF-FIELDRW
$$\frac{\begin{array}{c}\Gamma; n_{this}; x \vdash S[x] \triangleright_o [\texttt{that/this}]S[f] = Q_o \\ \forall t_o \in Q_o\ s.t.\ t_o = <x.\texttt{PD}, q> \text{ or } t_o = <p, x.\texttt{PD}> \text{ or } t_o = <x.\texttt{PD}, x.\texttt{PD}> \exists t \in S[y]\ s.t.\ [x/\texttt{this}]t = t_o \\ Q_o \cap (S[y] \leftarrow t_o) = Q_y \qquad \Gamma; n_{this}; x \vdash Q_y \triangleright_i S[x] = Q_i \qquad Q_i \cap S[f] = Q_f \\ \Gamma; n_{this}; x \vdash Q_y \triangleright_r Q_f = Q_r \qquad Q_r \cap S[x] = Q_x \qquad S' = [y \to Q_y,\ f \to [\texttt{this/that}]Q_f, x \to Q_x]S\end{array}}{\Gamma; S; n_{this} \vdash x.f = y \text{ or } y = x.f,\ S'}$$

TF-THISFIELDRW
$$\frac{S' = [y \to (S[y] \cap S[f]), f \to (S[y] \cap S[f])]S}{\Gamma; S; n_{this} \vdash \texttt{this}.f = y \text{ or } y = \texttt{this}.f,\ S'}$$

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

TF-THISINVK
$$\frac{\begin{array}{c}mdbody(m) = (x_m, y_m) \\ S' = [z \to (S[z] \cap S[x_m]), x_m \to (S[z] \cap S[x_m]), x \to (S[x] \cap S[y_m]), y_m \to (S[x] \cap S[y_m])]S\end{array}}{\Gamma; S; n_{this} \vdash x = \texttt{this}.m(z),\ S'}$$

FIGURE 6.5: Transfer functions. We use ... to break a long premise on multiple lines.

own type system constraints, such as prohibit object creation with an owner being `p` (T-NEW). An object can be created only in a local domain of `this` or its own domain.

6.4.2 Higher Level Rules

Two rules work on a higher level than variables or expressions (Fig. 6.7). METHOK ensures that for an overriding method the qualifier of the method parameter is the same as the qualifier of the method parameter of the overridden method, and similarly, for the return type. Moreover, if method is a `public` method, then its

T-New
$$\frac{\Gamma(x) = C_x<p_x, q_x> \quad C <: C_x \quad <p_x, q_x> = <p, q> \quad p_x \in \{\texttt{owned}, \texttt{PD}, \texttt{owner}, \texttt{shared}\}}{\Gamma \vdash x = \texttt{new}\ C<p, q>()}$$

T-Assign
$$\frac{\Gamma(x) = C_x<p_x, q_x> \quad \Gamma(y) = C_y<p_y, q_y> \quad C_y <: C_x \quad <p_x, q_x> = <p_y, q_y>}{\Gamma \vdash x = y}$$

T-Write
$$\frac{\Gamma(x) = C_x<p_x, q_x> \quad \Gamma(y) = C_y<p_y, q_y> \quad T_f\ f \in CT(C_x) \quad T_f = C_f<p_f, q_f> \quad C_y <: C_f \quad <p_x, q_x> \triangleright <p_f, q_f> = <p_y, q_y> \quad pubsig(f)}{\Gamma \vdash x.f = y}$$

T-ThisWrite
$$\frac{\Gamma(y) = C_y<p_y, q_y> \quad T_f\ f \in CT(C_{this}) \quad T_f = C_f<p_f, q_f> \quad C_y <: C_f \quad <p_f, q_f> = <p_y, q_y>}{\Gamma \vdash \texttt{this}.f = y}$$

T-Read
$$\frac{\Gamma(x) = C_x<p_x, q_x> \quad \Gamma(y) = C_y<p_y, q_y> \quad T_f\ f \in CT(C_y) \quad T_f = C_f<p_f, q_f> \quad C_f <: C_x \quad <p_y, q_y> \triangleright <p_f, q_f> = <p_x, q_x> \quad pubsig(f)}{\Gamma \vdash x = y.f}$$

T-ThisRead
$$\frac{\Gamma(x) = C_x<p_x, q_x> \quad T_f\ f \in CT(C_{this}) \quad T_f = C_f<p_f, q_f> \quad C_f <: C_x \quad <p_f, q_f> = <p_x, q_x>}{\Gamma \vdash x = \texttt{this}.f}$$

T-Invk
$$\frac{mdtype(m) = T_m \to T_r \quad \Gamma(x) = C_x<p_x, q_x> \quad \Gamma(y) = C_y<p_y, q_y> \quad \Gamma(z) = C_z<p_z, q_z> \quad T_m = C_m<p_m, q_m> \quad T_r = C_r<p_r, q_r> \quad C_r <: C_x \quad C_z <: C_m \quad <p_y, q_y> \triangleright <p_m, q_m> = <p_z, q_z> \quad <p_y, q_y> \triangleright <p_r, q_r> = <p_x, q_x> \quad pubsig(m)}{\Gamma \vdash x = y.m(z)}$$

T-ThisInvk
$$\frac{mdtype(m) = T_m \to T_r \quad \Gamma(x) = C_x<p_x, q_x> \quad \Gamma(z) = C_z<p_z, q_z> \quad T_m = C_m<p_m, q_m> \quad T_r = C_r<p_r, q_r> \quad C_r <: C_x \quad C_z <: C_m \quad <p_z, q_z> = <p_m, q_m> \quad <p_r, q_r> = <p_x, q_x> \quad pubsig(m)}{\Gamma \vdash x = \texttt{this}.m(z)}$$

FIGURE 6.6: Typing rules for SOD.

parameter or its return type cannot have `owned` in their qualifiers. ClsOK checks that a `public` field cannot be `owned`, and that all the methods in a class are valid based on MethOK.

The auxiliary judgements $mdType()$ and $mdBody()$ return the type and the body of a method, respectively. The auxiliary judgement $pubsig()$ enforces the SOD constraints on the qualifiers of `public` methods and fields. First, $pubsig(C, f)$ checks

$$\textsc{ClsOK}\quad \frac{T\ f \qquad pubsig(C,f) \qquad md\ OK\ in\ C}{\texttt{class}\ C\texttt{<owner,p> extends}\ C'\texttt{<owner,p>}...\ OK}$$

$$\textsc{MethOK}\quad \frac{\begin{array}{c} CT(C) = \texttt{class}\ C\texttt{<owner,p> extends}\ C'\texttt{<owner,p>}\ldots \\ override(m, C'\texttt{<owner,p>}, T \to T_R) \\ T = C<p_1,q_1> \qquad T_R = C_r<p_2,q_2> \\ mdtype(m) = T' \to T'_R \\ T' = C'<p_3,q_3> \qquad T'_R = C'_r<p_4,q_4> \qquad pubsig(m) \\ <p_1,q_1> = <p_3,q_3> \qquad <p_2,q_2> = <p_4,q_4> \end{array}}{T_R\ m(T\ x)\ \{\ \overline{T\ y}\ e; \texttt{return}\ y_m;\ \}\ OK\ in\ C}$$

FIGURE 6.7: SOD type system constraints.

$$\textsc{Aux-mdtype}\quad \frac{(T_R\ m(T\ x)\ \{\overline{T\ y}\ e;\ \texttt{return}\ y_m;\})\ \in\ md}{mdtype(m) = T \to T_R}$$

$$\textsc{Aux-mdbody}\quad \frac{(T_R\ m(T\ x)\ \{\overline{T\ y}\ e; \texttt{return}\ y_m;\})\ \in\ md}{mdbody(m) = (x,\ y_m)}$$

$$\textsc{Aux-mpublic}\quad \frac{\begin{array}{c} public(m) \qquad mdtype(m) = T \to T' \\ T = C<p,q> \qquad T' = C'<p',q'> \\ p \neq \texttt{owned} \qquad q \neq \texttt{owned} \\ p' \neq \texttt{owned} \qquad q' \neq \texttt{owned} \end{array}}{pubsig(m)}$$

$$\textsc{Aux-fpublic}\quad \frac{\begin{array}{c} T\ f \in CT(C) \qquad public(f) \\ T = C'<p,q> \\ p \neq \texttt{owned} \qquad q \neq \texttt{owned} \end{array}}{pubsig(C,f)}$$

FIGURE 6.8: Auxiliary judgements.

if the field f in the class C has the visibility modifier `public`, its qualifier cannot contain `owned`. Also, $pubsig(m)$ checks if a method is `public`, then the qualifier for its method parameter or its return cannot contain `owned` (Fig. 6.8).

6.5 Properties of Set-Based Solution

Similarly to Huang et al., we define some properties for our set-based solution. We adapt their Proposition 1 and show it holds for our set-based solution. Proposition 1 states if the set-based solution removes a qualifier from the set of qualifiers of a variable, then there is no set of qualifiers that type-checks the program and contains the removed qualifier. We also discuss the *optimality property*, which states that, at the fixed point, the highest ranked qualifier of the set of qualifiers of each variable type-checks the program.

Proposition 1. *Let S be the set-based solution. Let x be any variable in a program P, and let $<p, q>$ be any qualifier in SOD. If $<p_0, q_0> \notin S[x_0]$ for some x_0, then there does not exist a valid typing T for program P in SOD such that $T[x] = <p_x, q_x>$ and $<p_x, q_x> \in S[x]$ for all x and $T[x_0] = <p_0, q_0>$.*

Proof. (Sketch) We say that $<p, q>$ is a *valid qualifier* for x if there exists a valid typing T, where $T[x] = <p, q>$. Let x_0 be the first variable that has a valid qualifier $<p_0, q_0>$ removed from its set $S[x_0]$ and let f_e be the transfer function that performs this removal. Since $<p_0, q_0>$ is a valid qualifier for variable x_0 in expression e, for the other variables in e, there exist other valid qualifiers $<p_1, q_1>$,...,$<p_n, q_n>$ that make e type-check in SOD. If $<p_1, q_1> \in S[x_1], ..., <p_n, q_n> \in S[x_n]$, then by definition of a correct transfer function, f_e would not have removed $<p_0, q_0>$ from $S[x_0]$. So one of $x_1, \ldots, x_n$ must have had a valid qualifier removed from its set *before* the application of f_e. This contradicts the assumption that x_0 is the first variable that had a valid qualifier removed from its set of qualifiers.

Optimality Property. The optimality property does not hold for unannotated programs for SOD, so in our approach, developers refinements provide additional information for the analysis. For the optimality property to hold, the analysis needs enough information, which one refinement may not provide. Therefore, the developers have to do more refinements until the optimality property holds. In contrast with Huang et al., which ask for certain amount of manual qualifiers for optimality property to hold, our approach does not ask for more refinements until the optimality property holds. Our approach tries to find some valid typing even after one refinement, by enumerating the variables and their sets of qualifiers. The reason that our approach cannot save the highest ranked qualifier for each variable is in SOD there are multiple maximal typings. Therefore, if the analysis picks the highest ranked qualifier in the set of qualifiers of each variable, one qualifier may get picked from a maximal typing T_1, and another qualifier may get picked from another maximal typing T_2. The typings

T_1 and T_2 may not be the same, so saving qualifiers based on two typings may lead to programs that do not type-check.

Soundness of the transfer functions. There is a transfer function f_e for each expression e. Each transfer function f_e takes as input a set mapping S, and outputs an updated mapping S'. Let f_e be the transfer function that removes the invalid qualifiers from the set of qualifiers of each variable $x_i \in e$. After the application of f_e, for each variable $x_i \in e$, and each $<p_i, q_i> \in S'[x_i]$, there exists $<p_1, q_1> \in S'[x_1]$,..., $<p_{i-1}, q_{i-1}> \in S'[x_{i-1}]$, $<p_{i+1}, q_{i+1}> \in S'[x_{i+1}]$, ..., $<p_n, q_n> \in S'[x_n]$, such that $<p_1, q_1>$,...,$<p_n, q_n>$ type-check with the rule for e in Fig. 6.6. Making e type check requires that the typing rule for e holds.

$$
\begin{aligned}
&\forall x_i \in e, S'[x_i] = \{<p_i, q_i> | \\
&<p_i, q_i> \in S[x_i] \text{ and} \\
&\exists <p_1, q_1> \in S'[x_1], \ldots, <p_{i-1}, q_{i-1}> \in S'[x_{i-1}], \\
&<p_{i+1}, q_{i+1}> \in S'[x_{i+1}], \ldots, <p_n, q_n> \in S'[x_n] \text{ s.t.} \\
&<p_1, q_1>, \ldots, <p_n, q_n> \text{ type-check with the rule for } e\}
\end{aligned}
$$

6.6 Finding a Typing In a Set Mapping

To extract a typing T from a Set Mapping S, we need a function f such that for any variable x:

$$T[x] = f(S[x])$$

In Huang et al., $f = max$, which selects the maximal typing. In our case, the following holds:

$$T[x] = f(S[x]) \text{ where } f = max \text{ for some } x$$
$$\text{or } f = scalar \text{ if } |S[x]| = 1$$
$$\text{or } f = next \text{ otherwise}$$

where *next* picks one element in $S[x]$, and *scalar* converts a singleton set (of cardinality $|S| = 1$) to a single value.

Chapter 7: Evaluation

In this Chapter, first, we discuss the implementation of the inference analysis tool (Section 7.1). Then, we discuss the method (Section 7.2) and results (Section 7.3) of evaluating our approach on several small examples. At the end of this chapter, we discuss how our evaluation addresses the proposed hypotheses of the thesis (Section 7.4).

7.1 Tool Implementation

We implemented the inference analysis on a dataflow analysis framework, Crystal [20], which handles building the Control Flow Graph, the Three-Address Code representation, and invoking the transfer functions we supply. The analysis saves qualifiers as annotations in the code, using language support for annotations. We have an independent type-checker that reads the annotations and type-checks them, and a separate extraction analysis that uses the annotations to extract the object graph. We manually run the independent type-checker to validate the inferred qualifiers. We also integrated the analysis into a user interface.

Below, we show a screenshot of a working prototype of the refinement tool (Fig. 7.1), on the same MicroAphyds example. The refinement tool is an Eclipse plugin.

The left side shows the ownership tree. Starting from the `SHARED` root domain, each object contains zero or more domains, and each domain contains zero or more objects. By default, we create one private domain, called `owned`, and one public domain, called `PD`, per object.

For illustration purposes, we list below the tree the refinements that have been applied. The first column shows the refinement type, the middle columns show the arguments of the refinement, such as the source object, the destination object, and

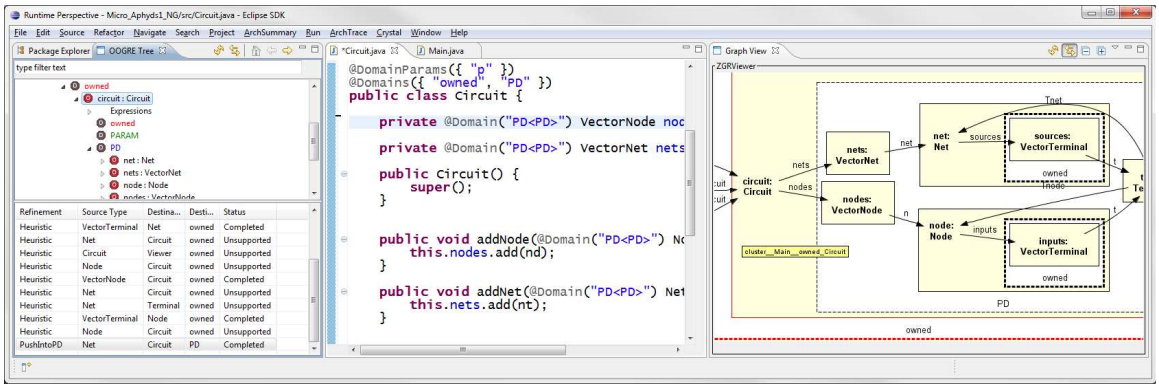


FIGURE 7.1: Snapshot of the current Eclipse prototype. The outcomes of attempted heuristics and proposed refinements (Completed or Unsupported) are shown below the ownership tree (bottom left).

the domain name. Finally, the last column shows the status of applying the refinement and running the inference analysis. If a refinement is successful, it appears as Completed in the Status column. A disallowed refinement has its status set to Unsupported. Refinements can be exported and re-applied to a system.

The middle part of the screen shows in the Eclipse Java editor the code with the annotations that are saved by the inference analysis. The annotations use language support for annotations available in Java 1.5 or later. The `@Domains` keyword lists the domains that a class declares. The `@DomainParams` keyword lists the domain parameters of a class. The `@Domain("p<q>")` annotation on a field, local variable, method parameter, or method return type, saves the inferred qualifier `<p,q>` for that variable. A separate type-checker can optionally check that the code and the annotations are consistent with each other and report any warnings in the Eclipse Problems window. The right side shows the refined graph based on running the graph extraction analysis on the code with annotations. The graph is visualized using nested boxes, which allow expanding or collapsing objects to reveal or hide lower-level objects.

7.2 Evaluation Method

We first explain the evaluation method. We ran the implemented tool on the test cases. When we ran the tool, we turned on the heuristics for inferring `owned` and `PD`. However, for some test cases, the AST visitors do not find any target variable, so there is no applicable heuristics. On the initial graph, we did a few refinements on each test case. We selected the refinements based on the design intent of the test case that we created or reused. After the refinements, we reported the total number of refinements and heuristics, the completed ones and the total number of valid typings that the tool found after all the heuristics and after each refinement. Moreover, we showed the metrics for the best and the worst typing to illustrate the significant differences between them.

7.3 Evaluation Results

In this section, we discuss the results of running our tool on several test cases in detail. The measurements in this section are described in Table 7.1 and Table 7.2. The notation used in the table, `ref(`C_1`>`C_2`)`, means that `ref` is the refinement type, C_1 is the type of the source abstract object, C_2 is the type of destination abstract object. `ref` can be `pio` for PushIntoOwned or `pip` for PushIntoPD. For SplitUp, we use the notation `spu(var,C,domain)` where `var` is the name of the target variable, `C` is the type of the destination object and `domain` is the destination domain, which can be `owned` or `PD`.

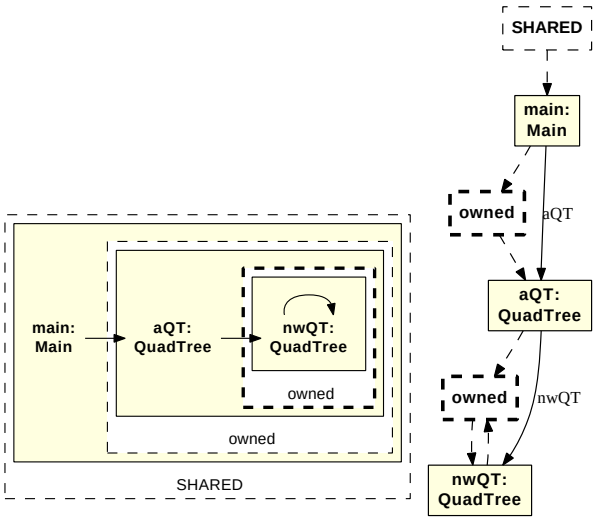
Stack. This test case is inspired by the `XStack` example in [14] and illustrates a stack data structure implemented using a linked list. Each element of the stack has a pointer that points to the next element, and a data item to represent the data that is stored in each element. One difference with the `XStack` example is that we extend the test case to have a wrapper class over the data item of the elements of the stack.

Therefore, the wrapper object should be part of each element, and it should be in the PD domain. Also, the stack data structure has the top of the stack as a field. The approach cannot find any target variable for a heuristic to run for this test case. We refine that initial object graph to push the wrapper object into PD of the abstract object that represents an element (pip(XOwner>Link)). We also push top into the owned domain of the stack (pio(Link>XStack)). As the last refinement, we push the stack object into PD of the root object (pip(XStack>Main)).

By doing pip(XOwner>Link), the approach finds 16 valid typings to save to the code. It means the approach found 16 ways to insert qualifiers to the code to fulfill the refinement. The approach saves the best one that has two more owned qualifiers instead of two shared qualifiers. After doing pio(Link>XStack), there is one valid typing to save and the approach saves that one. Having only one typing means the optimality property holds after the second refinement for the program and SOD. The pip(XStack>Main) refinement leads to one valid typing that has five owned and seven PD modifiers. In the valid typing, there are only two shared qualifiers.

QuadTree. Recursive types are tricky for the extraction analysis, which creates a cycle in the object graph to ensure that the graph is finite. This test case checks that the inference analysis also handles recursive types. The class QuadTree has a field of its own type. By splitting up the abstract object of type QuadTree and pushing the target variable (the field) into owned of QuadTree, the inference analysis pushes the abstract object of type QuadTree into the owned domain of another abstract object of the same type. The recursive type leads to a cycle in the object graph, as expected. In this example there is one valid typing to save.

MicroAphyds. This test case was taken from a larger pedagogical circuit layout application, Aphyds [13]. The test case illustrates a circuit that contains a Vector of Node and a Vector of Net. Each Node and each Net also containing a Vector of Terminal. The design intent is to make the Vector objects strictly encapsulated to



(A) Nested boxes. (B) Internal representation.

FIGURE 7.2: Representations of the object graph for the QuadTree test case.

distinguish between them. `Node` and `Net` and `Terminal` are parts of `Circuit`. We also show two tier architecture by pushing `Circuit` and `Placer` in one top-level domain and `Viewer` in another top-level domain. By having heuristics to infer `owned` and `PD`, the approach attempts seven heuristics to infer `owned`, and four thereof succeed. The completed heuristics push the `Vector` objects into `owned` domain of the abstract objects that traces to declaring class of the `Vector` objects. So, the initial object graph for this test case is not flat. We also refine the object graph by pushing the abstract object of type `Node` into `PD` of the abstract object of type `Circuit`. As a result, abstract objects of types `Net` and `Terminal` are also be pushed into `PD` of `Circuit`. After this refinement, there is one typing to save. We do two more refinements and push the abstract objects of types `Circuit` and `Placer` in `PD` of the root object and push the abstract object of type `Viewer` into `owned` of root object.

CourSys. This larger test case is a fully working example of size 1.4 KLOC that is used in a software architecture class. It is a course registration application: the classes `Student` and `Course` contain information related to students and courses, respectively. The `Student` class has a list of completed, and another list of registered courses. The

TABLE 7.1: Total and completed number of heuristics and refinements for each test case.

test case	Heus				Refs					
	PIO	ComPIO	PIP	ComPIP	PIO	ComPIO	PIP	ComPIP	SPU	ComSPU
Stack	0	0	0	0	1	1	2	2	0	0
Aphyds	7	4	0	0	1	1	2	2	0	0
QuadTree	0	0	0	0	0	0	0	0	1	1
CourSys	33	12	5	5	0	0	3	1	0	0
SM	19	11	0	0	3	0	8	5	3	1

TABLE 7.2: Refinements for each test case and the best and worst S to save. owd=`owned`, owr=`owner`, shd=`shared`, ALPlayer = ArrayListPlayer, LTile = LetterTile, DLSlot= DoubleLetterSlot, TWSlot = TripleWordSlot, TLSlot = TripleLetterSlot, NSlot = NormalSlot, ALSlot = ArrayListSlot

Test case/Refs.	Typings	BestTyping					WorstTyping				
		owd	pd	p	owr	shd	owd	pd	p	owr	shd
Stack											
heuristics	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
pip(XOwner>Link)	16	2	1	9	2	14	0	1	11	0	16
pio(Link>XStack)	1	11	1	13	1	2	11	1	13	1	2
pip(XStack>Main)	1	5	7	13	1	2	5	7	13	1	2
Aphyds											
heuristics	1	14	0	30	24	0	14	0	30	24	0
pip(Node>Circuit)	1	14	12	18	24	0	14	12	18	24	0
pip(Circuit>Main)	1	8	18	18	24	0	8	18	18	24	0
pio(Viewer>Main)	1	9	17	24	18	0	9	17	24	18	0
QuadTree											
heuristics	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
spu(nQT>QuadTree,`owned`)	1	5	0	1	0	2	5	0	1	0	2
CourSys											
heuristics	412	28	10	131	90	182	11	10	109	3	307
pip(Student>Data)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
pip(Course>Data)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
pip(Logic>Main)	415	20	18	130	87	186	7	10	106	3	298
ScrabbleModel											
Heuristics	274	30	0	19	21	428	14	0	15	1	468
pio(Player>ALPlayer)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
spu(tmpAry>Game,`owned`)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
spu(delAry>Dictionary,`owned`)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
spu(plyAry>Game,`owned`)	275	31	0	18	23	428	15	0	18	3	464
pip(LTile>TileBag)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
pip(DLSlot>Board)	266	29	1	21	21	428	15	1	19	3	462
pip(TWSlot>Board)	257	27	2	21	22	428	15	2	20	3	462
pip(TLSlot>Board)	248	25	3	23	21	428	15	3	19	1	460
pip(NSlot>Board)	239	23	4	22	23	428	15	4	22	3	456
pip(ALSlot>Board)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
pip(Game>Main)	236	19	8	23	22	428	15	4	25	6	448

`Course` class has the list of registered students for the course. The idea here is to distinguish between different list objects with different design purposes. There is another class `Data` that provides access to student and course data including reading

the record information and writing registration information. The `Logic` class contains the main logic of the system, and the `Client` class is responsible for accepting user input, executing the corresponding logic methods and displaying the results. There are also classes that provide logging and read/write lock management.

This test case is valuable because this pedagogical system has been used in demonstrations and tutorials for adding Ownership Domains qualifiers manually, and using a type-checker to check the validity of the qualifiers. According to the tutorials, it takes over one hour to inspect the system and add the qualifiers by hand. Using the refinement tool, and by turning on the heuristics, we were able to analyze the system in a few minutes. This system uses many classes from the Java standard library such as the `java.util` collections and `java.io.File`, so we manually generated stubs for those classes to use in the evaluation.

Out of 33 attempted heuristics to infer `owned`, 12 of them are completed. It is interesting to see from the completed heuristics to infer owned that there are several cases of strict encapsulation. E.g., a `log` object encapsulated a `lock` object. However, a high number of attempted heuristics and a low number of completed ones mean many objects are nearly encapsulated, or that our heuristics visitors are not very precise. For example, the collection of `Course` objects is `protected` in the `Data` class, but a `public` method returns an alias to it.

The approach also attempts five heuristics to infer `PD` and all of them are completed. In particular, those heuristics distinguish between different collection objects and split them up, even though they are not strictly encapsulated. The resulting graph distinguishes between the list of students in the `Course` object that indicates the list of students who are registered for the course, and the list of students in the `Data` object that indicates the list of all the students in the system. So, using object creation as a heuristic to infer public domains seems to match the design intent.

By doing `pip(Logic>Main)`, there are 415 valid typings. The metrics show drastic

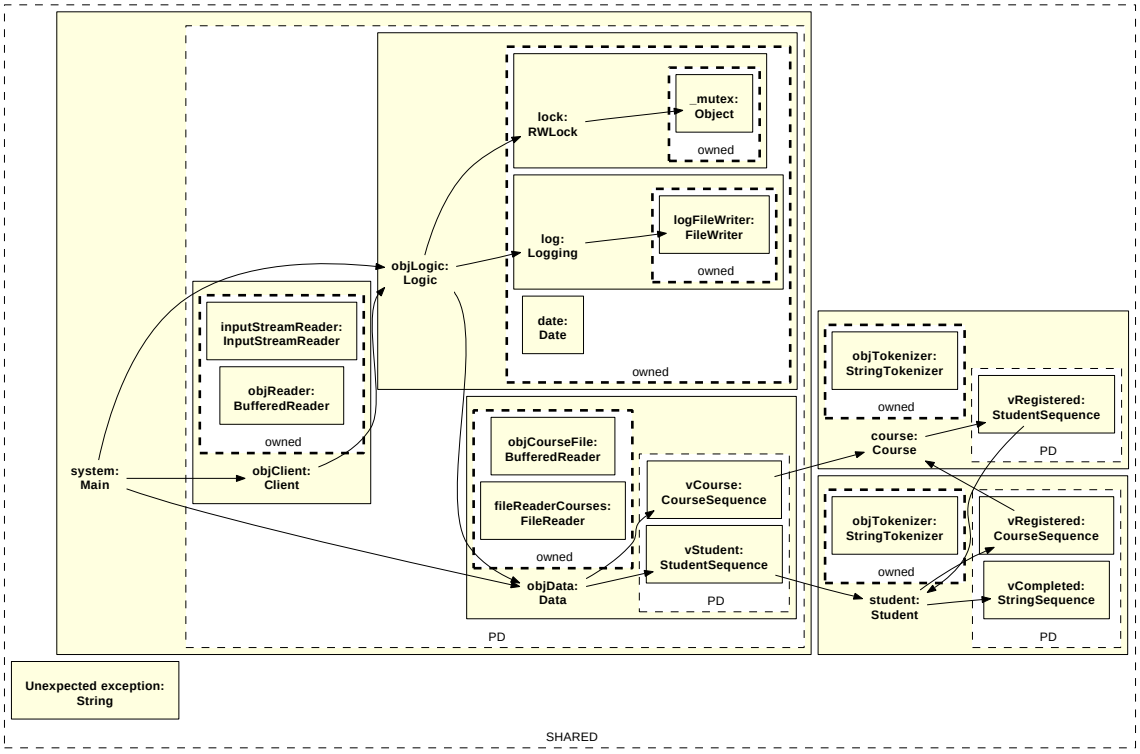
difference on the qualifiers of the best and the worst typing (Table 7.2). There are two false positives, refinements that are possible by adding the qualifiers manually, but the tool does not allow them (`pip(Student>Data)` and `pip(Course>Data)`). The reason is the tool always prefers S_p over S_{PD}, but for those refinements it should select S_{PD}. Heuristics and each refinement take around 1.5 minutes. In total, it takes 5 minutes to complete the test case using the tool.

In summary, using the tool, we were able to infer most of the qualifiers that we typically add manually. The heuristics matched the design intent in most cases. Only a few, manual refinements are needed in addition to the heuristics to match the design intent. Moreover, using the tool was significantly faster than adding the qualifiers by hand, even when using an earlier tool that propagates most of the boilerplate code.

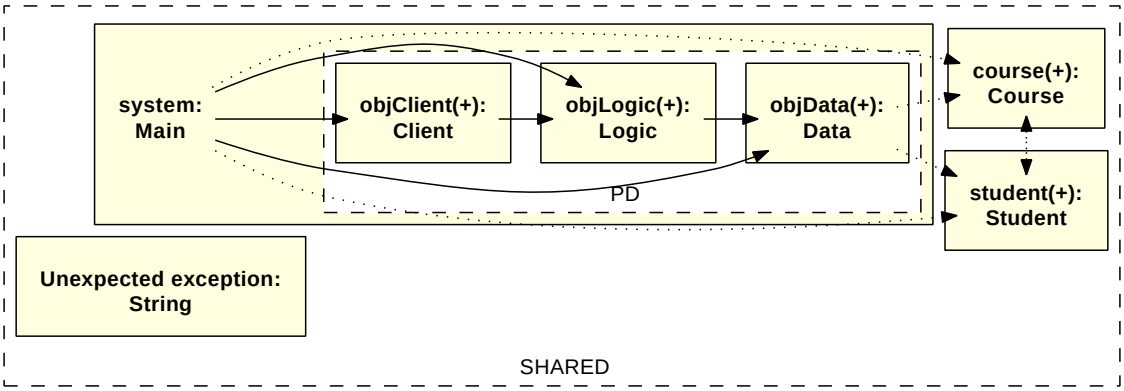
Moreover, we took the manually added qualifiers from the example solution and compared them to the inferred qualifiers. In terms of the quality of the qualifiers inferred by the tool, the tool's qualifiers are equally as precise and in some cases, even more precise when compared to the manual qualifiers. In Fig. 7.3, we show the expanded (Fig. 7.3a) and collapsed (Fig. 7.3b) versions of the final object graph.

ScrableModel. In this next test case, the refinements express the design intent of the original system designer. ScrableModel (SM) is the application logic of a complete game implemented by an undergraduate student as a course project for a software design class, covering object-oriented concepts and design patterns. The game is the Scrabble board game, where players earn points by forming valid words from a set of letter tiles. ScrabbleModel does not include the user interface part of the implementation, which is based on the Java Swing library, which we cannot analyze yet since we do not support all language features, such as anonymous classes. The size of the program is around 2 KLOC.

There is a `Game` class that contains the `Board`, `TileBag`, current `Player` and a `Dictionary` that contains the valid words. The `Board` class implements the logic of



(A) Expanded object graph.



(B) Collapsed substructures of some objects shown with (+).

FIGURE 7.3: Expanded and collapsed object graphs for the CourSys test case.

adding letters to the board and removing letters from it. The `TileBag` class contains the `LetterTile`s that the players use. `Player` owns a `TileTray`.

The student received detailed tutorials on Ownership Domains that had been used in a classroom setting over multiple lectures. We then asked him to practice on his own by manually adding qualifiers to some examples, i.e., do the practice labs associated with the tutorials, while using a type-checker. Next, we asked the student

to add qualifiers to the CourSys system (used above), but he struggled with the task and did not complete it. However, we did not record any information, about the training of the student, or the time that we spent on it.

The student then received some informal instruction on using the interactive refinement tool. He was asked to come up with refinements that reflect his design intent. He was able to draw some object hierarchies by hand. He seemed to understand those concepts, but struggled with some subtle points. For example, at some point, he attempted to push objects into the public domain of a collection object, so he had to be reminded that a collection does not hold on to arbitrary objects in its public domain, since a collection is typically parameterized by the domain of the objects it holds, and is intended to be very reusable. He was then asked to draw the system's conceptual architecture on a whiteboard, which he was able to do. It then became much easier for him to use the tool, and by looking at the object graph, he was able to refine the object graph until it closely matched the conceptual diagram he drew.

The student used an earlier version of the implementation which had some limitations that we have addressed since. For example, he was frustrated when many objects shifted around during a refinement; that is why we respect previous refinements, during each refinement. It was quite interesting that the student, who struggled with adding qualifiers to the code manually, was able to complete the task using the refinement tool and looking at the object graphs, thus reinforcing our belief in the potential benefit of *what you see is what you get object graphs.*

For the results in the table, we re-did the student's refinements, on the latest implementation of the analysis. Out of 19 attempted heuristics to infer `owned`, 11 of them are completed. The 25 refinements provided by the student and the completed heuristics are included in the list. There are several cases of strict encapsulation. For example, the object of type `Board` is encapsulated by the object of type `Game`.

However, there are some cases of near encapsulation, which fail some the heuristics or refinements. To express design intent, there are several PushIntoPD refinements. For example, in this game there are different types of slot that may be placed on the `Board`. Those objects are parts of the object of type `Board`, so they must be in its `PD`. We refine the object graph by doing four PushIntoPDs to push the corresponding object of different types of slot into `PD` of `Board`. Another refinement is to push the top-level object of type `Game` into the `PD` of the root object. By doing that refinement, all the objects underneath `Game` and the other objects underneath them move with `Game`.

Another interesting point is that by doing more refinements, the number of valid typings decreases (as shown in the first column). The reason is that refining the object graph more leads to more constraints on the sets of qualifiers. So, there are fewer qualifiers that are valid for some variables and therefore, for the whole program. Running all the heuristics takes around 2 minutes; each refinement also takes around 2 minutes, with the majority of the time spent in searching for valid typings. In total, it takes around 15 minutes to complete the test case using the tool.

7.4 Discussion of Hypotheses

In this section we revisit the three hypotheses that we introduce in Chapter 1 and discuss how our evaluation addresses each one.

H1. Using a visual approach, developers are able to interactively refine an abstract object graph.

We designed and implemented a tool that is able to interactively refine an abstract object graph. The hypothesis is true, because the tool has interaction with developers in the process of refining the object graph. Developers do a refinement and the tool shows the refined object graph. Developers do the next refinement on the resulting object graph and the tool shows them another refined object graph.

Therefore, developers apply each refinement on a refined object graph that reflects the previous valid refinements.

H2. The developers are able to express two types of hierarchy, strict encapsulation and logical containment.

Our evaluation results show that this hypothesis is true. In Table 7.2 we show that for different test cases, we are able to do different types of refinements to create the two types of hierarchy in the object graph. For example, for the test case `Stack`, we refine the object graph by pushing the object of type `Link` into the `owned` domain of the object of type XStack. That refinement illustrates the strict encapsulation in the object graph. Moreover, we refine the object graph by pushing an object of type `XOwner` into the `PD` domain of the object of type `Link`. That refinement expresses a logical containment relation between those objects. Since Huang et al. [14] do not support public domains, they are not able to express this type of hierarchy in the object graph.

H3. If the code as written supports the requested refinement, the inference analysis infers valid qualifiers that satisfy the requested refinement and type-check.

On all the tests cases, there are no serious warnings but a few minor warnings left. The type-checker warnings that we encountered during development of the tool helped identify bugs in the transfer and adaptation functions.

Chapter 8: Related Work

In this chapter, first we explain the challenges that every ownership inference approach should address (Section 8.1). Next, we discuss specific approaches and argue how they address the challenges (Section 8.2).

8.1 Challenges

Any approach to infer ownership qualifiers must address the following challenges.

- *Soundness:* Sound qualifiers implement a type system. A sound approach must infer qualifiers that type-check [7, 15, 12, 18, 21, 14, 10];
- *Precision:* An approach must select the most precise qualifier between valid qualifiers. The precision can be defined based one a preferred ranking over the qualifiers [15, 14] or the depth of the inferred ownership structure [18, 21, 25, 10].
- *Trivial qualifiers:* An approach must have a way of selecting a trivial qualifier that always type-checks and does not require expensive computation. The trivial qualifiers can be considered as a starting point for an approach, especially the ones that show the results of inference in graphical forms [18, 21, 25].
- *Interactive vs. fully-automated:* An approach can work in a fully-automated mode [7, 18, 12, 25] or in an interactive mode [15, 14, 10, 21]. An interactive approach may accept partial qualifiers and infers the remaining, or accepts graphical interactions.
- *No solution:* An approach must handle the case when it cannot find any solution that type-checks. An approach may not save qualifiers [14, 10], or may produce meaningful error messages [21].
- *Multiple solutions:* An approach may infer more than one valid solution for a program, and it must be able to pick one. An approach may use metrics to pick between different solutions [25]. Another approach may show the different

solutions to the developers and asks them to pick one [21].

- *Reusable code:* Parameters are often introduced to make code more reusable. Although it is hard to infer where the code is intended to be reusable automatically. Some approaches do not infer parameters, but at the cost of restricting the ownership model [12, 18]. Inferring an arbitrary number of parameters is often problematic [7]. For simplicity, an approach may infer one parameter, which is still suitable to express a number of programs in practice [15, 14].

8.2 Specific Approaches

We organize related approaches into four groups based on their output and the type of the program analysis they use.

Static analysis/saves qualifiers. Huang and Milanova [15] present an approach to infer OT. It is an interactive approach, since the developers require adding qualifiers for a subset of variables. The approach utilizes a sound set-based solution and uses transfer functions that analyze all the expressions. The transfer functions eliminate the invalid qualifiers. The approach infers a single ownership parameter. The approach terminates with an error when there is no solution.

Huang et al. [14] present a framework to infer qualifiers of type systems. The framework can be instantiated using three parameters: a set of qualifiers, viewpoint adaptation functions, and type-system-specific constraints. Huang et al. instantiate the approach for OT and UT. They introduce the notion of best qualifier by defining heuristics that rank the qualifiers using objective functions. For OT, the developers add qualifiers for some of the object creation expressions in the code, so the optimality property holds.

Vakilian et al. [21] propose a universal framework that takes a type system as input and produces an inference for that. Although, it requires a checker on top of the Checker framework [9] for the type system. The inference tool is interactive and

is inspired by speculative analysis that helps the developer in the process of decision making, by showing the consequences of their decisions ahead of the time. It builds a tree that consists of two types of nodes: error and change nodes.

Dietl et al. [10] build a tunable static type inference for Generic UT. It can work on fully un-annotated Java codes or on partially annotated ones. By traversing the AST, the approach generates different types of constraints for variables and solves the constraints by reusing a max-SAT solver tool. Although, the max-SAT solver makes the approach does not scale. The approach has more than one strategy where there are multiple solutions: adjusting heuristics by changing the weights, or requiring developers to input partial qualifiers. If there is no solution, the inference produces no result.

Aldrich et al. [7] 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 conducts a constraint system including three sets of constraints, *equality*, *component*, and *instantiation* that guarantee soundness of the approach. The algorithm solves the constraint system and integrates the result with other qualifiers based on a defined ranking. However, over 50% of the inferred qualifiers are `shared`. The main problem of this approach is that it infers many ownership parameters, sometimes one for each field of a class.

Dymnikov et al. [12] present an ownership inference containing an ownership inference and an ownership checker. The ownership inference infers `owned` qualifiers for the fields of a class. The inference system implements some heuristics to infer strictly encapsulated fields in a class, so it is not a sound approach.

Static analysis/visualizes ownership. Milanova and Vitek [18] present a static analysis that infers OT, and the result is an ownership tree that illustrates 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 different

types of edges, which indicate the ownership relation between nodes of the graph. Third, a dominance boundary analysis creates dominance boundaries as subgraphs of the object graph.

Zhu and Liu [25] present a sound constraint-based ownership inference, Cypress that uses an application of linear programming. The goal of Cypress is to generate a hierarchical decomposition of the heap statically. The hierarchy is based on ownership relations between the objects. The approach follows the "tall and skinny" principle and favors heap decompositions that are taller and skinnier. The result of Cypress is a visualized decomposed heap. The approach is fully-automated.

Dynamic analysis/saves qualifiers. Dietl and Müller [24] present an approach that analyzes the execution of programs and infers ownership qualifiers from the executions. The approach consists of five steps. In step 1, it builds the representation of object store that is called Extended Object Graph containing all the objects that ever existed in the store and their modification information. In step 2, it creates the dominator tree using the fact that in UT, all the modifications of an object should be initiated by its owner. In step 3, it resolves the conflicts with UT. In step 4, it harmonizes different instantiations of a class, and at the final step it outputs the qualifiers. The approach is fully-automated, but is not sound and may generate qualifiers that do not type-check.

Chapter 9: Discussion and Conclusion

In this chapter, first, we discuss some implementation details (Section 9.1). Then we discuss some important points about the analysis (Section 9.2). Next, we talk about some limitations of the work (Section 9.3), some future (Section 9.4) work and conclude (Section 9.5).

9.1 Implementation Details

In this section, we talk about some implementations details. First we argue how we handle library code. Next, we discuss how we support generic types.

Library code. The current implementation handles library code in two ways. The first requires generating stubs for library classes. The inference analysis analyzes the stubs and infers qualifiers for them. However, these qualifiers may not be general enough to be reused across multiple applications. The second assumes that each library variable receives the initial set of qualifiers below:

$$<\texttt{owner},\texttt{owner}>, <\texttt{owner},\texttt{p}>, <\texttt{p},\texttt{p}>$$

Support generic types. For expressiveness, we need to support generic types, e.g., a `Vector<T>`, as used in the Aphyds example, e.g., `nets` of type `Vector<Net>` (Fig. 9.1). Our inference analysis infers one additional "inner" parameter for a generic collection class with one generic type parameter. This is needed to express an object of type `Vector<T>` containing objects with the qualifier $<q, w>$. The `Vector` object is in some domain p and has an actual parameter $<q, w>$. Effectively, the qualifier of `Vector` is $<p, q<w>>$

We define a new adaptation case for one generic type parameter. In the rule that

```
1  class Vector<T><owner, p> {
2            // T: generic type parameter
3                // owner, p: ownership parameters
4    T<p> obj; // the "trick" is to use one actual here
5    // obj is virtual/ghost field that summarizes Vector
6  }
7  class Circuit<owner, p> { // domain parameters
8    private domain owned; // private domain
9    public domain PD; // public domain
10   net = new Net<PD,p>();
11   nets = new Vector<Net><owned, PD<p> >();
12   // 1. <PD,p> is qualifier of Net object
13   // 2. owned is actual for Vector's owner
14   // 3. PD<p> is actual for Vector's p
15   ...
16   nets.add( net ); // Add object to collection
17   net = nets.obj; // Field read
18 }
```

FIGURE 9.1: Generic collection with one generic type parameter requires a qualifier with an inner/nested domain.

$$\frac{\begin{array}{c}\textsc{Adapt-X-Gen}\\ t_1 = \texttt{<p>} \qquad t_2 = <p_0, <q_0, w_0>>\\ n_{rcv} : T_{rcv} \qquad isGeneric(T_{rcv})\end{array}}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright_X t_1 = <q_0, w_0>}$$

FIGURE 9.2: Adaptation case to support generic types. X can be o for ADAPT-OUT, i for ADAPT-IN, and r for ADAPT-RCV.

is shown in Fig. 9.2, the type variable qualifier, t_1, has only the *first* element. The reason is the receiver type is of a parameterized type, so it has to have an *inner* element and for the type argument *second* and *inner* work like *first* and *second*. Therefore, for the receiver qualifier, t_2, there are p_0 as *first*, q_0 as *second* and w_0 as *inner*. In order to determine if a receiver type is a generic type, we use the auxiliary judgement $isGeneric(T_{rcv})$ that accepts the type of the receiver.

In Fig. 9.2, we show one parametric rule, ADAPT-X-GEN. When $X = o$, it finds the result qualifier based on the inner and receiver qualifiers. When $X = i$, it finds the inner qualifier based on the result qualifier and the receiver qualifier. When $X = r$, it finds the receiver qualifier based on the result and the inner qualifiers.

9.2 Discussion

We believe it is easier to do refinements using a visual approach on the object graph than to add type qualifiers in the code. So the developers never switch to the code to add qualifiers. Adding type qualifiers is harder due to the fact that the developers need to specify both the owning domain and the domain parameter. We believe this is particularly the case for novice developers who are not familiar with the procedure of type-checking an expression, but they do know about the design intent of their programs. Using our tool, the developers only specify the owning domain by doing a refinement, and the tool finds the correct ownership domain parameter.

The fact that the tool finds at least one valid typing after each refinement means the developers do not have to resolve conflicts by adding qualifiers to the code or doing more refinements. As the designers of the tool, we think it is easier for novice developers to wait for the tool to find a valid typing than resolve conflicts by adding qualifiers manually or doing more refinements. This way, teaching ownership types to undergraduate students can be easier. Of course, this claim requires more careful evaluation with a controlled experiment.

We also think that supporting public domains in SOD enables developers to express more design idioms. Using public domains, the developers are able to impose more hierarchy on the object graph with fewer restrictions. In SOD, the developers can express the part-of relation between objects without making the child object inaccessible to the other objects. Therefore, developers get more hierarchy on the object graph, which makes them more manageable.

We believe the tool should be as flexible as possible in terms of the supported types of refinements. It means the developers should be able to do any type of refinement, as they could add qualifiers to the code. Therefore, the other types of refinements can come into play, such as PushIntoShared, which places an object in the global domain `shared`, or PushIntoOwner, which makes object peer of another object.

Moreover, there are some usability issues with using a tool to infer ownership qualifiers. Using our tool, some of the refinements, which are possible by adding qualifiers to the code manually, are unsupported. One reason is a previous refinement can make a current refinement unsupported. This means the previous refinement conflicts with the current one. We consider a refinement that is doable manually, but cannot be done by the tool to be a false positive. As the designers of the tool, it was a tradeoff for us between respecting previous refinements and having some false positives, or not respecting the previous refinements and having more refinements completed. If we do not respect previous refinements, then after each refinement, objects shift around between different domains and it would be hard for developers to keep track of them. So we chose to respect the previous refinements to make the tool easier to use at the cost of having some false positives.

Another reason is that our strategy to pick one solution between multiple solutions may be introducing more false positives. When the tool selects one solution, the other ones get discarded. Some of the false positives could be avoided by selecting another solution for a previous refinement. To prevent this, the tool can support the notion of *undoing* a refinement, so if a previous refinement conflicts with a current one, the user can prefer the current refinement over the previous one by undoing the previous refinement.

Another usability issue is the fact that the tool should be able to give useful information about the unsupported refinements to developers. When developers do a refinement, most of the time, they believe it should work. By showing the reason of the refinement being unsupported, developers may be able to understand how to fix the problem. For example, if the tool shows the expression in the code that is responsible for the refinement being unsupported, the developers may understand if they have to change the code, or do not apply a previous refinement in the next run of the tool.

9.3 Limitations

Next we discuss some limitations of this work.

One ownership parameter. The approach currently supports the implicit `owner` and the explicit `p` parameter. Extending our approach to infer more than one parameter is left to future work. Huang et al. experimented with instantiating their framework for OT with 2 and 3 ownership parameters, then concluded to restrict the system to one ownership parameter.

Final fields. In SOD, the code can refer to the public domain of an object through a final field `n`, using the construct `n.d`. If the variable `n` is not final, it may be reassigned, and the type system would lose track of the relationship between an object and the objects contained in its public domain. In other ownership type systems, where `n` is always `this`, this is not an issue. In SOD, it also possible to refer to a public domain through a sequence of final fields `n1.n2...d`, though that is not part of our current formalization. The adaptation and the inference analysis also introduce these qualifiers. If `n` is not final, then a type system extension is needed to handle that situation. In particular, the qualifier `n.d` becomes an existential domain since the type system cannot track the instance to which the domain is tied. Adding this existential domain to Ownership Domains is future work.

Side effects of a refinement. Using the set-based solution, when the developers do a refinement, some other objects may also shift into different domains in the resulting object graph. Those changes are asked by the refinement, but they are part of the inferred typing. We call those changes *auto-refinements*. In our current implementation, we do not respect the auto-refinements, so by doing the next refinement, the developers may see different auto-refinements. It may be a good idea to respect auto-refinements, to avoid showing developers object graphs that differ dramatically after each refinement.

9.4 Future Work

Partial annotations. We plan to allow developers to add some partial annotations for some of the variables and the analysis will read the annotations and use them. This feature is available in many inference tools [10, 15, 14].

User study. This thesis shows the technical feasibility and details. Evaluating the WYSIWYG claim and the visual aspect of inference requires a user study, which is left for future work. So far, a preliminary exploratory study with one undergraduate student showed promising results (see the ScrabbleModel test case in Chapter 7).

9.5 Conclusion

We propose and implement an approach where developers express their design intent by refining an object graph directly, while an analysis infers valid ownership type qualifiers in the code. These qualifiers are used by a separate extraction analysis to extract an updated graph. Such a tool can increase the adoptability of ownership type qualifiers, to reap their benefits in improving code quality, such as identifying cases of representation exposure, or exposing shallow versus deep cloning [4]. Such a tool also has pedagogical applications to help novice developers understand object structures and some structural object-oriented design patterns. Also, this work increases the adoptability of reasoning about security policies at the level of object graphs with security properties and constraints [5, 16, 23].

REFERENCES

[1] Abi-Antoun, M., and Aldrich, J. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA* (2009).

[2] Abi-Antoun, M., Ammar, N., and Hailat, Z. Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report. In *QoSA* (2012).

[3] Abi-Antoun, M., and Barnes, J. M. Analyzing Security Architectures. In *ASE* (2010).

[4] Abi-Antoun, M., Giang, A., Chandrashekar, S., and Khalaj, E. The eclipse runtime perspective for object-oriented code exploration and program comprehension. In *Proceedings of the 2014 Workshop on Eclipse Technology eXchange* (2014), ETX '14, pp. 3–8.

[5] Abi-Antoun, M., Khalaj, E., Vanciu, R., and Moghimi, A. Abstract runtime structure for reasoning about security: Poster. In *Proceedings of the Symposium and Bootcamp on the Science of Security* (2016), HotSos '16, pp. 1–3.

[6] Aldrich, J., and Chambers, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP* (2004).

[7] Aldrich, J., Kostadinov, V., and Chambers, C. Alias Annotations for Program Understanding. In *OOPSLA* (2002).

[8] Clarke, D. G., Potter, J. M., and Noble, J. Ownership Types for Flexible Alias Protection. In *OOPSLA* (1998).

[9] Dietl, W., Dietzel, S., Ernst, M., Muslu, K., and Schiller, T. Building and using pluggable type-checkers. In *ICSE* (2011).

[10] Dietl, W., Ernst, M., and Müller, P. Tunable static inference for generic universe types. In *ECOOP* (2011).

[11] Dietl, W., and Müller, P. Universes: Lightweight Ownership for JML. *Journal of Object Technology 4*, 8 (2005).

[12] Dymnikov, C., Pearce, D. J., and Potanin, A. Ownkit: Inferring modularly checkable ownership annotations for java. In *Australian Software Engineering Conference* (2013).

[13] Hauck, S. Aphyds: The academic physical design skeleton. In *Proceedings of the 2003 International Conference on Microelectronics Systems Education* (2003).

[14] Huang, W., Dietl, W., Milanova, A., and Ernst, M. Inference and checking of object ownership. In *ECOOP* (2012).

[15] Huang, W., and Milanova, A. Towards effective inference and checking of ownership types. In *IWACO* (2011).

[16] Khalaj, E., Vanciu, R., and Abi-Antoun, M. Is there value in reasoning about security at the architectural level: A comparative evaluation. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security* (2014), HotSoS '14, pp. 30:1–30:2.

[17] Marron, M., Sanchez, C., Su, Z., and Fahndrich, M. Abstracting Runtime Heaps for Program Understanding. *TSE 39*, 6 (2013), 774–786.

[18] Milanova, A., and Vitek, J. Static dominance inference. In *International Conference on Objects, Models, Components, Patterns (TOOLS)* (2011).

[19] Mitchell, N. The Runtime Structure of Object Ownership. In *ECOOP* (2006).

[20] PLAID Research Group. The Crystal Static Analysis Framework, 2015. `http://code.google.com/p/crystalsaf`.

[21] Vakilian, M., Phaosawasdi, A., Ernst, M. D., and Johnson, R. E. Cascade: a universal programmer-assisted type qualifier inference tool. In *ICSE* (2015).

[22] Vanciu, R., and Abi-Antoun, M. Object Graphs with Ownership Domains: an Empirical Study. 109–155.

[23] Vanciu, R., Khalaj, E., and Abi-Antoun, M. Comparative evaluation of architectural and code-level approaches for finding security vulnerabilities. In *Proceedings of the 2014 ACM Workshop on Security Information Workers* (2014), SIW '14, pp. 27–34.

[24] Werner, D., and Müller, P. Runtime Universe Type Inference. In *IWACO* (2007).

[25] Zhu, H. S., and Liu, Y. D. Heap decomposition inference with linear programming. In *ECOOP* (2013), pp. 104–128.

ABSTRACT

INTERACTIVE REFINEMENT OF HIERARCHICAL OBJECT GRAPHS

by

EBRAHIM KHALAJ

December 2016

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science

Degree: Master of Science

Developers need to understand the runtime structure of object-oriented code, and abstract object graphs can help. To extract abstract object graphs that convey design intent in the form of object hierarchy, additional information is needed to express this hierarchy in the code using ownership types, but adding ownership type qualifiers after the fact involves manual overhead, and requires developers to switch between adding qualifiers in the code and looking at abstract object graphs to understand the object structures that the qualifiers describe. We describe an approach where developers express their design intent by refining an object graph directly, while an inference analysis infers valid qualifiers in the code. A separate extraction analysis then uses these qualifiers and extracts an updated object graph. We implement and test the approach on several small test cases and confirm its feasibility.

AUTOBIOGRAPHICAL STATEMENT

EBRAHIM KHALAJ

EDUCATION

- Master of Science (Computer Science), December 2016
  Wayne State University, Detroit, MI, USA
- Master of Science (Software Engineering), October 2011
  Sharif University of Technology, Iran
- Bachelor of Computer Engineering, November 2008
  Shahid Beheshti University, Iran

PUBLICATIONS

1. Abi-Antoun, M., **Khalaj, E.**, Vanciu, R., and Moghimi, A. Abstract Runtime Structure for Reasoning about Security. *Poster at Symposium and Bootcamp on the Science of Security (HotSoS)* (2016).
2. Abi-Antoun, M., Wang, Y., **Khalaj, E.**, Giang, A., and Rajlich, V Impact Analysis based on a Global Hierarchical Object Graph. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015).
3. **Khalaj, E.**, Vanciu, R., and Abi-Antoun, M. Is There Value in Reasoning about Security at the Architectural Level: a Comparative Evaluation. *Poster at Symposium and Bootcamp on the Science of Security (HotSoS)* (2014).
4. Vanciu, R., **Khalaj, E.** and Abi-Antoun, M. Comparative Evaluation of Architectural and Code-Level Approaches for Finding Security Vulnerabilities. In *Workshop on Security Information Workers (SIW), co-located with the ACM Conference on Computer and Communications Security (CCS)* (2014).
5. Abi-Antoun, M., Giang, A., Chandrashekar, S., and **Khalaj, E.** The Eclipse Runtime Perspective for Object-Oriented Code Exploration and Program Comprehension. In *Eclipse Technology eXchange Workshop (ETX)* (2014).