Extracting Dataflow Objects and other Flow Objects

Radu Vanciu Marwan Abi-Antoun
Department of Computer Science, Wayne State University
{radu, mabiantoun}@wayne.edu

Abstract

Finding architectural flaws in object-oriented code requires a runtime architecture that shows multiple components of the same type that are used in different contexts. Previous work showed that a runtime architecture can be approximated by an abstract object graph that a static analysis extracts from code with Ownership Domain annotations. To find architectural flaws, it is not enough to reason about the presence or absence of communication. Additional work is needed to reason about the content of the communication. The contribution of this paper is a static analysis that extracts a hierarchical object graph with dataflow edges that refer to objects. The extraction analysis combines the aliasing precision provided by Ownership Domains with a domain-sensitive value flow analysis. We evaluate the extraction analysis on an open-source Android application and discuss examples of dataflow edges that refer to objects that are in actual domains or to flow objects that are in domains corresponding to `unique` annotations.

Keywords architectural extraction, static analysis, domain-sensitivity, ownership types, hierarchical object graphs

1. Introduction

While reasoning about security and communication in object-oriented code, developers often use a runtime architecture that shows multiple instances of the same class. In a runtime architecture, a component is an abstract set of objects and a connector represents communication between objects. A connector is more than just a simple relation between objects, and often show "what" information a connector communicate. Our goal is to approximate the runtime architecture as a graph in which a node represents abstract objects, and an edge shows the communicated object.

Consider for example, an Android app that is typically composed of multiple Activity components. In Android, a common communication mechanism is based on objects of type Intent, which are like events or messages for inter- and intra-application communication that enable two Activity objects to communicate with each other. A runtime architecture of an app often shows multiple objects of type Intent, and these objects are shown on the edges (Fig. 1).

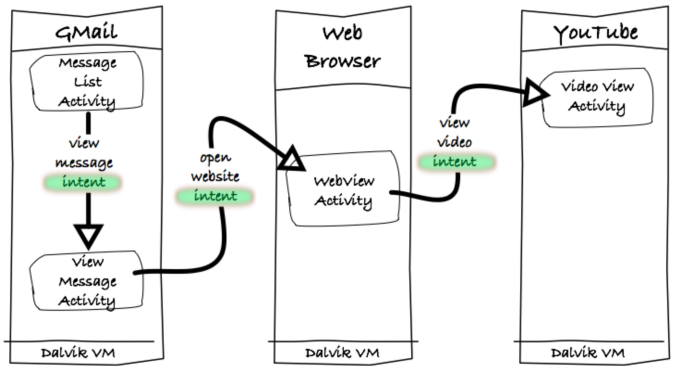

Figure 1: A runtime architecture used for learning the basics of Android applications [5, Fig.4-2]. Highlighted are multiple dataflow objects of the same type Intent.

Reverse engineering an object graph with communication edges is challenging in the presence of aliasing. Security requires a worst-case analysis. A dynamic analysis is therefore unsuitable because it considers only a finite subset of all possible executions and may miss important objects and communication. On the other hand, a static analysis can approximate all possible executions. Various static analyses were proposed to compute a may-alias relation that determines objects that a variable may refer to at runtime. A precise may-alias analysis distinguishes between variables used in different contexts. The context can be a method call in which case the analysis is *call-site context sensitive*, or the context can be an object and the analysis is *object-sensitive*. Ideally, a precise may-alias analysis should be call-site and object-sensitive, but such an analysis does not scale [13].

Another solution to control aliasing is type systems such as Ownership Domains [2] that enhance types using ownership information. Ownership Domains group conceptually related objects into domains. Two variables that refer to objects in different domains cannot be assigned to each other. Since domains are coarser than objects, a *domain-sensitive* analysis that uses the aliasing precision provided by domains scales better than a precise object-sensitive analysis.

In previous work [1, 19], we proposed a static analysis that extracts a hierarchical Ownership Object Graph (OOG) with dataflow edges as an approximation of the runtime architecture from code with Ownership Domain annotations.

However, the dataflow edges showed types rather than objects. In addition, the previous analysis did not handle some extensions to Ownership Domains that increase the expressiveness of the type system and that are used in practice.

For additional expressiveness, Ownership Domains support `lent` for objects that are borrowed in method invocations, and `unique` for objects that are passed linearly from one domain to another [2]. Since we use Ownership Domains and its extensions for architectural extraction and security analysis, we focus here on the static analysis that extracts dataflow edges that refer to objects and handles expressions that involve references declared as `lent` or `unique`.

The extraction is challenging because the analysis needs to preserve soundness and achieve precision in the presence of aliasing. The OOG is sound if and only if each object created at runtime has exactly one abstract object as a representative in the OOG, and every runtime edge has a corresponding abstract edge between the representatives of the source and the destination runtime objects. For variable in domains other than `lent` and `unique`, the analysis uses the ownership types to find objects that a dataflow refers to. For a `lent` variable, the analysis attempts to find the actual domains from where the object is borrowed. For a `unique` variable, the analysis attempts to find the actual domains where the object sinks by analyzing the value flow. If an actual domain is found, the analysis creates a *dataflow object* as a reference from the dataflow edge to an abstract object. If the analysis cannot locate an actual domain, it creates a *flow object* in a fresh domain that is added in the substructure of the object that created the flow object.

Contributions. This paper presents a static analysis that extracts a hierarchical object graph with dataflow edges that refer to objects. The analysis combines the aliasing precision provided by Ownership Domains with a domain-sensitive value flow analysis. The contributions are:

- a formalization of the static analysis that extracts *dataflow objects*;
- a domain-sensitive value flow analysis that attempts to resolve `lent` and `unique` to actual domains;
- a formalization of *flow objects* in the context of dataflow edges.

Outline. Section 2 reviews Ownership Domains and related work. Section 3 defines dataflow communication and flow objects. Section 4 gives a formal description of the extraction analysis. Section 5 discusses preliminary findings by running the analysis on an open-source Android application, and Section 6 concludes.

2. Background and Related Work

We describe the preliminaries using a running example. We review the Ownership Domains type system, which the analysis leverages to obtain aliasing precision. Then, we discuss the differences between a domain-sensitive analysis and variants of the most closely related may-alias analyses.

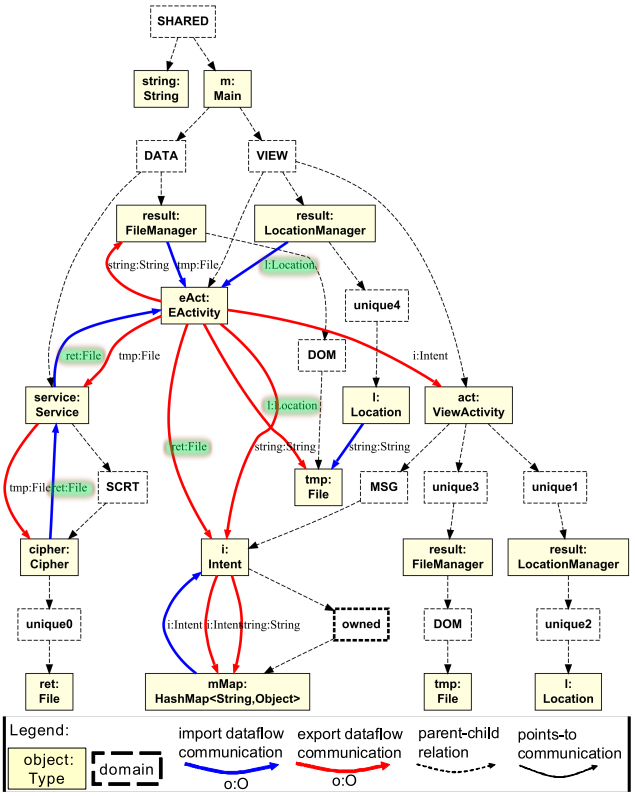

Figure 2: A fragment of the CryptoApp OOG internal representation (excluding self edges). Dataflow edges refer to objects that are nodes in actual domains, or flow objects (highlighted) that are in domains `unique`i. The OOG distinguishes between objects of same type, e.g., the flow object `ret:File` and the dataflow object `tmp:File`.

2.1 Running Example

As a running example, we use CryptoApp, an Android application that searches in a file for the GPS location information, and encrypts the content of the file. The example includes simplified code from the Android framework that the application uses such as `Activity` and `Intent`. The app uses the factory-method design pattern and inheritance, as is common in object-oriented code (Fig. 3).

As a notation, an object labeled `obj:T` indicates a reference obj of type `T`, which we then refer to either as the object `obj` or the `T` object to mean an instance of the `T` class.

2.2 Ownership Domains

An ownership domain is a conceptual group of objects with an explicit name. An ownership domain can convey design intent and represents an architectural tier. Several key features of Ownership Domains are crucial for expressing design intent in code. The first is having explicit "contexts" or domains. Other ownership type systems implicitly treat all objects with the same owner as belonging to one implicit context. For architectural extraction, explicit domains

```
1  class Main<owner> {
2   domain DATA,VIEW;
3   EActivity<DATA, DATA,VIEW> eAct = new EActivity();
4  }
5  class EActivity<owner,V,D> extends Activity<owner,V,D> {
6    void onCreate(){
7      FileMngr<D,V,D> fm = (FileMngr)getManager("FILE");
8      LctMngr<V,V,D> lm = (LctMngr)getManager("LOCATION");
9      Location<lent> loc = lm.getLastLocation();
10     File<lent> tempFile = fm.read("history");
11     if (tempFile.find(loc)) {
12       File<unique> encrypted = serviceRun(tempFile);
13       Intent<unique> i = new Intent(ACTION_SEND);
14       i.putExtra(EXTRA_STREAM,encrypted);
15       Activity<V,V,D> act = new ViewActivity();
16       act.startActivity(i);
17     }
18   }
19   File<unique> serviceRun(File<lent> x) {
20     Service<D> service = new Service();
21     return service.encrypt(x);
22   }
23 }
24 class Activity<owner,V,D> {
25   domain MSG;
26   Intent<MSG> mIntent;
27   //cannot use lent here, since intent flows in a field
28   void startActivity(Intent<MSG,V,D> intent) {
29     mIntent = intent;
30   }
31   Manager<unique,V,D> getManager(String<shared> name) {
32     ...//return FileMngr or LctMngr
33   }
34 }
35 class ViewActivity<owner,V,D> extends Activity<owner,V,D> {}
36 class Intent<owner,V,D>{
37   domain OWNED;
38   Map<OWNED, String<SHARED>,Object<owner>> mMap = ...;
39   void putExtra(String<shared> name, Object<owner> value) {
40     mMap.put(name,value);
41   }
42 }
43 class Service<owner> {
44   domain OWNED;
45   Cipher<OWNED> cipher = new Cipher();
46   File<unique> encrypt(File<lent> x) {
47     return cipher.doFinal(x);
48    }
49 }
50 class Cipher<owner> {
51   File<unique> doFinal(File<lent> x) {
52     File<unique> ret = new File();
53     // encrypt content of x into ret
54     return ret;
55   }
56 }
57 class LctMngr<owner,V,D> extends Manager<owner,V,D> {
58   Location<unique> getLastLocation(){
59     return new Location();
60   }
61 }
62 class FileMngr<owner,V,D> extends Manager<owner,V,D> {
63   domain DOM;
64   File<DOM> read(String<shared> x) {
65     return new File(x);
66   }
67 }
```

Figure 3: CryptoApp: Code fragments with ownership types

are useful, because developers can define multiple domains per object to express their design intent. For CryptoApp, the developer declares two top-level domains or tiers, `DATA` and `VIEW`, and places objects in these domains (Fig. 2).

In Ownership Domains, an object does not have child objects directly, instead an object has domains, which in turn have objects. Domains are declared on a class, but are treated like fields in the sense that fresh domains are created for each instance of that class. As a notation, to distinguish between domains with the same name D, we refer to a domain as o.D, where o is the name of the parent object. For a domain `MSG` declared on a class `Activity`, and two instances o1 and o2 of type `Activity`, the domains `o1.MSG` and `o2.MSG` are distinct for distinct `o1` and `o2`.

Domain parameters propagate ownership information of objects outside the current object. A type declaration has a class name and a list of formal domain parameters. By convention, the first element of the list is the `owner` domain. To bind formal domain parameters to actual domains, the code that instantiates the class supplies the actual domains. In this way, domain parameters allow objects to share state. Typically, a factory object does not own the objects it creates. Instead, the factory object references them through domain parameters. This way, `eAct:EActivity` can create the object `service:Service` in the `DATA` domain that is shared between `eAct:EActivity` and `act:ViewActivity`.

The type system also supports the special annotation `shared`, to indicate an object that can be aliased globally. A global domain `shared` contains such objects, and little reasoning can be done about variables declared `shared`. For example, the variable `m` of type `Main` and the constants of type `String` are declared `shared`.

Graphically, an object is represented with a filled rectangle and a domain with a dashed rectangle. The dataflow edges are solid edges, where the label on a dataflow edge is the object `o:T` that the dataflow edge refers to. The representation distinguishes between export dataflow edges (red) and import dataflow edges (blue). A dashed edge represents a parent-child relations between an object and a domain, while a thin edge represents a points-to relation due to a field reference [1]. Figure 2 shows the raw representation of the OOG because this paper focuses on its extraction. Other representations, e.g., based on nested boxes, are possible. The OOG has dataflow edges that refer to flow objects, and shows the flow objects in separate domains named `unique`i, where i is a fresh identifier. Throughout the paper, we use *actual domain* to refer to a domain that has a domain declaration, and *fresh domain* to refer to a domain that corresponds to a `unique` annotation.

2.3 Extensions of Ownership Domains

For additional expressiveness, Ownership Domains has `lent` and `unique` annotations, which do not correspond to actual domains. A variable declared `unique` refers to an object to which there is only one reference, such as a newly

created object, and can be passed linearly from one domain to another. For example, in a factory method such as `getManager`, the actual domain of the object created by the factory is known only by the client code (Fig. 3, line 31). Therefore, the method returns `unique` objects of various types that extend `Manager`. Next, the client code in `EActivity` assigns these objects to the domain parameter `D` for `FileMngr` object and `V` for `LctMngr` object (Fig. 3, line 7, 8).

Fields are not usually declared `unique` because a field read expression is not usually followed by an object destruction. For example, an encrypted object `ret:File` is created by the object `cipher:Cypher` and is passed linearly to `eAct:EActivity` that stores it into a field of type `Intent` declared in the `MSG` domain (line 26).

A variable declared `lent` refers to an object that is temporarily lent from one domain to another as long as an object in the second domain does not create a persistent reference to the borrowed object, e.g., by storing it in a field. Only method formal parameters and local variables can be `lent`. For example, local variable `tempFile` (line 10), and the formal parameter `x` (line 21) are declared `lent`. A method cannot return a `lent` object and cannot store it in a field, so a `lent` annotation on a field or a return type is prohibited.

A newly created object is often declared `unique`. The client code can assign a `unique` variable to a variable in a locally declared domain, in a domain parameter, in `shared`, or `lent`. The converse is prohibited: a variable declared `lent` can only be assigned to other variables declared `lent`. Therefore, variables declared `unique` are universal sources and can flow to all other variables, while variables declared `lent` are universal sinks [3]. Because a local variable that refers to a newly created object is a source, it cannot be declared `lent`. Since the OOG is an over-approximation, we extract all such objects such that the OOG has a representative for all the objects that are created in any possible execution.

Several variables declared in different domains can alias the same object. An extraction analysis needs to determine a destination variable that is declared in either a domain parameter, a locally declared domain, `shared` or `lent` that the variable declared `unique` may flow into. Similarly, for a variable declared `lent`, the extraction analysis needs to determine a source variable that is declared in either a domain parameter, a locally declared domain, `shared` or `unique`.

For example, the return value of the method `read` in `FileMngr`, the local variable `tempFile`, and the formal parameter `x` refer to the same object `tmp:File`. The method `read` of `FileMngr` returns an object in the domain `DOM` (line 64). The object `eAct:EActivity` borrows the object `tmp:File` (line 10) and passes it to `service:Service` for encryption (line 21). To find the actual domain of the formal parameter `x` that is declared `lent`, the extraction analysis needs to find the variable declared in `DOM`. Finding source and sink variables requires a value flow analysis, which traditionally uses a may-alias analysis.

2.4 May-Alias Points-to Analyses

To extract an object graph, one challenge for static analysis is to handle aliasing. Existing static analysis frameworks such as Soot [7] or WALA [6] provide may-alias points-to analyses that determine all the objects that a variable may refer to. Sensitivity determines the precision of such analyses. An analysis can be for example flow-sensitive or context-sensitive. A flow-sensitive analysis considers the order in which methods are called. In a context-sensitive analysis, the context is either a method call-site (call-site context-sensitive) or an abstract object (object-sensitive) that the receiver of a method may alias. An analysis is call-site context-sensitive if it analyzes a method invocation $r.m(a)$ multiple times based on the call-stack of method invocations that led to $r.m(a)$. In contrast, an analysis is object-sensitive if it uses object allocation expressions to distinguish between different objects that the receiver r may alias [15]. In general, it is not possible to compare the precision of a call-site analysis to that of an object-sensitive analysis [13]. In the literature, a call-site context-sensitive analysis is simply referred to as a context-sensitive analysis.

For example, a context-sensitive analysis distinguishes between invocations of the `makeFile` method (Fig. 4 lines 11 and 12) because the arguments of the method differ. However, it does not distinguish between the invocations at lines 11 and 13 because it considers that `makeFile` is invoked twice with the same arguments. In contrast, an object-sensitive analysis distinguishes between these two call sites because the receivers of `makeFile` alias different objects that correspond to distinct allocation expressions (lines 9 and 10).

Object-sensitivity. In terms of precision, the state-of-the-art analysis is object-sensitive [11, 18]. Such an analysis works well for on-demand based approaches that refine the references analyzed [16], but does not scale for a large number of

```
1  class FileMngr<owner> {
2    File<owner> makeFile(String<shared> s) {
3      return new File(s);
4    }
5  }
6  class Main<owner> {
7    String<shared> s1 = "a.txt";
8    String<shared> s2 = "b.txt";
9    FileMngr<DOM1> c1 = new FileMngr();
10   FileMngr<DOM2> c2 = new FileMngr();
11   File<DOM1> f1 = c1.makeFile(s1);
12   File<DOM1> f2 = c1.makeFile(s2);
13   File<DOM2> f3 = c2.makeFile(s1);
14 }
```

Figure 4: Supporting code to show differences between object- type- and domain-sensitivity.

references. Since in the presence of recursion, the size of the call-stack is unbounded, the analysis is parameterized with a constant k, which determines the maximum size of the call-stack, or the maximum sequence of object allocation expressions considered. In practice, analysis frameworks implement object-sensitive analyses for $k = 1$ or $k = 2$. For example, the Soot framework implements the object-sensitive analysis of Milanova et al. [11] for $k = 1$. That is, the implementation considers one abstract object corresponding to each object allocation expression. Smaragdakis et al. implemented the *2full-1heap* object-sensitive analysis [13] with $k = 2$ by also keeping track of the call-site context. Although the precision increases, the *2full-1heap* analysis does not scale. As an alternative, Smaragdakis et al. proposed an analysis that abstracts objects based on types, rather than object allocation expressions, and keeps track of the type of the calling context. The result is a *type-sensitive* analysis that scales and has a precision similar to *2full-1heap*.

Domain-sensitivity. Seeking a trade-off between soundness and precision, the static analysis that extracts OOG considers domains (groups of objects) as contexts and distinguishes between objects of the same type that are in different domains. Therefore, the OOG analysis is *domain-sensitive* and object- and flow-insensitive. The same way that an object-sensitive analysis looks at the receiver, a domain-sensitive analysis uses the domain of the receiver to distinguish between method invocations. For example, a domain-sensitive analysis distinguishes between the receivers `c1` and `c2` if the domain `DOM1` of `c1` is different from the domain `DOM2` of `c2` (Fig. 4 lines 9 and 10). However, a domain-sensitive analysis does not consider the object allocation expressions to make this distinction, and is independent of the parameter k to handle recursion. One consequence of using a predefined k is that the extracted object graphs has a predefined depth (i.e., the graph is flat for $k = 1$). Instead, a domain-sensitive analysis extracts a hierarchical object graph, where the depth of the graph is not predefined. The object hierarchy may contain cycles for recursive type declarations. Since a domain-sensitive analysis uses the aliasing precision provided by the ownership types rather than a stand-alone points-to analysis, it avoids the scalability problems in a style similar to a type-sensitive analysis.

Domain-sensitivity vs. object-sensitivity. A points-to analysis typically merges all the objects created at the same object creation expression into one equivalence class, attaching an object label h at *each* object creation expression `new` $C()$ (line 1). The domain-sensitive analysis labels an object creation expression with L (line 2). Compared to the label h of a standard points-to analysis, the label L is different as follows. First, multiple object creation expressions of the type C can still be represented by the same L label if the analysis maps the domain parameters to the same actual domains, whereas a basic points-to analysis creates multiple h labels. Second, if the analysis context maps the domain parameters to n different domains, then one object creation expression of type C may create n different L labels.

For example, for the same object allocation expression `new File()` in class `FileMngr` (Fig. 4 line 3), the domain-sensitive analysis maps the `owner` domain parameter to the `DOM1` domain in the context of `c1`, and then to `DOM2` in the context of `c2`. Therefore, the domain-sensitive analysis creates two abstract objects of type `File` for the same object allocation expression whereas a basic object-sensitive points-to analysis ($k = 1$) creates one abstract object.

$$\texttt{new}^{h}\ C() \tag{1}$$

$$\texttt{new}^{L}\ C{<}p_{owner}, p_{params}...{>}() \tag{2}$$

Domain-sensitivity vs. type-sensitivity. A type-sensitive analysis for $k = 2$ identifies the objects based on the type of allocated object, the type of allocator object, and the type of the receiver that instantiates the allocator object. For example, the type-sensitive analysis would distinguish between different objects of type `File` only if `FileMngr` were to be instantiated in different classes `Main1` and `Main2`. For one class `Main` with at least two domains, e.g., `DOM1` and `DOM2`, the domain-sensitive analysis is more precise than the type-sensitive analysis.

3. Flow Objects

Object Graph. An object graph such as an OOG that is statically extracted has nodes representing abstract objects. An abstract object is a representative for a possible unbounded number of objects that may exist at runtime. Edges between abstract objects represent possible edges that may exist during an execution. We are interested in extracting dataflow edges because a dataflow communication may lead to security vulnerabilities such as information disclosure when the object that the dataflow edge refers to contains confidential information, and the destination is untrusted [17].

Dataflow communication means that an object `a:A` has a reference to an object `o:O` and passes it to an object `b:B`, or an object `a:A` has a reference to an object `b:B` and receives a reference to an object `o:O` [14]. The objects `a:A` and `b:B` represent the source or destination objects, and `o:O` is a dataflow object that the dataflow communication refers to. To capture the directionality of the flow an object graph has *import* and *export* edges. An import dataflow edge exists due to the return value of a method invocation or a field read. An export dataflow edge exists due to an argument of a method invocations or a field write.

Flow Object. We define a *flow object* to be an object that is in a domain that corresponds to a `unique` annotation that the analysis cannot resolve to an actual domain. One flow object can be referred to from zero or more dataflow edges, or can be the source or the destination of a dataflow edge. For example, a `unique` variable that refers to a newly created object can be passed as a `lent` parameter of a method; the

called method can pass on the object as a `lent` parameter to other methods but cannot return it or store it in a field.

Examples of dataflow and flow objects. For CryptoApp (Fig. 3), a variable `l` of type `Location` is declared `unique` (line 58) and flows from the method `getLastLocation` of `LctMngr` to the local variable `loc` declared `lent` in `EActivity` (line 9), then it is passed as an argument to the `find` method in the class `File` which uses the location information without storing it. The value flow analysis computes the transitive flow, and attempts to resolve `unique` to an actual domain. Because all the sinks are declared `lent`, the analysis creates the flow object `l:Location`. On the other hand, the reference to `i:Intent` is declared `unique` (line 13), and is passed as an argument to the constructor of the class `Activity` (line 15). Since the formal argument `intent` of the constructor is declared in the domain `MSG` (line 29), the analysis can resolve `unique` to `MSG` such that the dataflow edge from `eAct:EActivity` to `act:ViewActivity` refers to the dataflow object `i:Intent`.

Soundness. An object graph is sound if and only if there is a mapping between any runtime object graph and the OOG, and the mapping has the following properties. Every runtime object has as a unique representative abstract object in the OOG. Every runtime edge between two runtime objects has a corresponding abstract edge between the representatives of the two objects. For a runtime dataflow edge that refers to a runtime object, the corresponding abstract dataflow edge refers to the representative of the runtime object, which can also be a flow object.

4. Formalization of the Extraction Analysis

This section presents the formalization of the extraction analysis using inference rules. We present the abstract syntax, the data types and the parts of the formalization related to the extraction of dataflow edges. In addition, we highlight the parts that are specific to resolving `lent` and `unique`, and the construction of the value flow graph.

4.1 Syntax

We formally describe our static analysis using a three-address code language based on Featherweight Domain Java (FDJ), which models a core of the Java language with Ownership Domains [2]. To keep the language easier to reason about, FDJ ignores advanced Java language constructs such as interfaces and static code.

In Fig. 5, the meta-variable C ranges over class names; T ranges over types; f ranges over field names. As a shorthand, an overbar denotes a sequence. Γ maps variables to their types a store S maps locations ℓ to their contents; the set of variables includes the distinguished variable `this` of type T_{this} used to refer to the receiver of a method; the result of the computation is a location ℓ, which is sometimes referred to as a value v; $S[\ell]$ denotes the store entry of ℓ; $S[\ell, i]$

$$
\begin{array}{rcll}
CT & ::= & \overline{cdef} & \textit{table of class declarations} \\
cdef & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha}, \overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} & \\
 & & \{\ \overline{dom};\ \overline{T_A\ f};\ C(\overline{T'_A\ f'}, \overline{T_A\ f}) & \\
 & & \{\texttt{super}(f'); \texttt{this}.\overline{f} = \overline{f};\}\ \overline{md}\ \} & \textit{class decl.} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; & \textit{domain decl.} \\
md & ::= & T_A\ ret\ m(\overline{T_B\ x})\ T_{this}\ \{ret = e_R; \texttt{return}\ ret;\} & \textit{method decl.} \\
e & ::= & \mid x = \texttt{new}\ C\texttt{<}A, \overline{p}\texttt{>}(\overline{y}) & \\
 & & \mid x = y.f \mid x.f = y \mid x = y \mid x = r.m(\overline{y}) & \\
 & & \mid \ell \mid \ell \triangleright e & \textit{expressions} \\
n & ::= & x \mid v & \textit{values or variable names} \\
p & ::= & \alpha \mid n.d \mid \texttt{SHARED} & \textit{domain name} \\
A & ::= & \texttt{unique} \mid p & \textit{domain may be unique} \\
B & ::= & \texttt{lent} \mid A & \textit{domain may be lent or unique} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} & \textit{precise type} \\
T_A & ::= & C\texttt{<}A, \overline{p}\texttt{>} & \textit{owner domain may be unique} \\
T_B & ::= & C\texttt{<}B, \overline{p}\texttt{>} & \textit{owner domain may be lent or unique} \\
v, \ell, \theta & \in & \textit{locations} & \\
x, y, r, a & \in & \textit{variables} & \\
S & ::= & \ell \rightarrow C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) & \textit{location store} \\
\Sigma & ::= & \ell \rightarrow T & \textit{store typing} \\
\Gamma & ::= & x \rightarrow T_B & \textit{type environment}
\end{array}
$$

Figure 5: FDJ syntax, extended using `lent` and `unique` [2]

denotes the value of i^{th} field of $S[\ell]$; $S[\ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v})]$ denotes adding an entry for location ℓ to S; α and β range over formal domain parameters; the expression form $\ell \triangleright e$ represents a method body e executing with a receiver ℓ; a program is a tuple $(CT,\ e_{root})$ of a class table and an expression that starts the program.

The syntax includes `lent` and `unique`. According to FDJ [3], only the first domain parameter (`owner` domain) of a type can be `lent` or `unique` because the class `Object` takes one domain parameter, and according to *cdef*, only the first domain is mandatory for every type. An object creation expression can have the first domain parameter `unique` (but not `lent`), thus the syntax uses the meta-variable A for a `new` expression. A type T consists of a class C parameterized with a list of domains that are of the following form: a domain parameter α, a declared domain $n.d$, or `shared`. The syntax also includes the meta-variable T_A for types in which the `owner` domain can also be `unique`, and T_B for types in which the `owner` domain can be also `lent`. For example, the type of fields in the class definition cannot be `lent` because a borrowed object cannot be stored in a field, hence the field type is T_A. On the other hand, the first domain in the type of a parameter in a method declaration can be `lent` or `unique` and the parameter type is T_B.

4.2 Data Types

The analysis extracts a hierarchical object graph (OGraph) with nodes that represent abstract objects (OObjects), groups of objects (ODomains), and OEdges that represent dataflow communication between abstract objects (Fig. 6).

$$
\begin{array}{lll}
G \in \mathsf{OGraph} & ::= \langle\ \textbf{Objects} = DO,\ \textbf{DomainMap} = DD, \textbf{Edges} = DE\ \rangle & \\
D \in \mathsf{ODomain} & ::= \langle\ \textbf{Id} = D_{id},\ \textbf{Domain} = C{::}d\ \rangle & \\
O \in \mathsf{OObject} & ::= \langle\ \textbf{Type} = C{<}\overline{D}{>}\ \rangle & \\
E \in \mathsf{OEdge} & ::= \langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst}, & \\
 & \quad \textbf{Label} = O_{label}, \textbf{Flag} = \mathit{Imp}\ |\mathit{Exp}\ \rangle & \\
DD & ::= \emptyset \mid DD \cup \{\ (O, C{::}d) \mapsto D\ \} & \textit{Domain map} \\
DO & ::= \emptyset \mid DO \cup \{\ O\ \} & \textit{Dataflow Object} \\
DE & ::= \emptyset \mid DE \cup \{\ E\ \} & \textit{Dataflow Edge} \\
\Upsilon & ::= \emptyset \mid \Upsilon \cup \{\ C{<}\overline{D}{>}\ \} & \textit{Stack of visited OObjects} \\
FG & ::= \emptyset \mid FG \cup \{\ (O_{src}, x, B_{src}) \overset{annot}{\rightsquigarrow} (O_{dst}, y, B_{dst})\ \} & \\
 & \textit{Value Flow Graph} & \\
annot & ::= (_i \mid)_i \mid \bullet \mid \star & \textit{value flow annotations}
\end{array}
$$

Figure 6: Data type of OGraph, and value flow graph

Each OEdge is a directed edge from a source O_{src} to a destination O_{dst}. The label of an OEdge is the OObject that the dataflow refers to. The flag states whether the OEdge represents an import or an export dataflow communication. The OGraph is a multi-graph, where multiple edges with different labels might exist between the same source and destination.

A value flow graph (FG) represents information flow between two variables x and y. A node in FG is a triplet (O, x, B) that denotes a variable x part of an expression that the analysis interprets in the context of O, and x is of a type T_B where the owner domain B is a domain p, `unique`, or `lent` as defined in Fig. 5. A shorthand the notation $(O, \overline{x}, \overline{B})$ means a list of j triplets (O, x_1, B_1), (O, x_2, B_2), $\ldots$, (O, x_j, B_j) An edge in FG has an $annot$ label to track if the value flow is due to a method invocation ($\overset{(_i}{\rightsquigarrow}$), a method return ($\overset{)_i}{\rightsquigarrow}$). The label $\bullet$ denotes an empty annotation on a value flow edge due an assignment ($\rightsquigarrow$). The label $\star$ denotes an information flow due to a field write. The label on value flow edges tracks call-site sensitivity [8, 9]. By considering O as a part of the node, the flow analysis is also domain-sensitive and has different nodes for the same variable x analyzed in different contexts.

4.3 Specification of the Extraction Analysis

The extraction analysis starts by creating the OObject O_{world} and its owning ODomain $D_{\texttt{SHARED}}$, which constitutes the root of the OGraph. It uses the following initial values:

$$
\begin{array}{l}
D_{\texttt{SHARED}} = \langle D_0, {::}\texttt{SHARED}\ \rangle, O_{world} = \langle C_{dummy} < .\ > \rangle \\
FG_0 = \emptyset,\ DO_0 = \{O_{world}\} \\
DD_0 = \{(O_{world},\ {::}\texttt{shared}) \mapsto D_{\texttt{shared}}\}, DE_0 = \emptyset
\end{array}
$$

Then, the analysis abstractly interprets e_{root} in the context of O_{world}:

$$\emptyset, \emptyset, FG_0, DO_0, DD_0, DE_0 \vdash_{O_{world}} e_{root}$$

We describe the analysis using rules of the following form:

$$\Gamma, \Upsilon, FG, G \vdash_O e$$

The rules consider the types of the variable in scope as provided by Γ, a stack Υ of visited OObjects to avoid non-termination, the value flow graph FG and the OGraph G, from which we can refer to its constituent parts DO, DD, and DE. The O subscript on the turnstile captures the context-sensitivity, and represents the context that the analysis uses to abstractly interpret an expression e.

The analysis uses expressions given in the form of three-address code, where x represents the left-hand-side of the expression. In *Df-New*, the analysis interprets an object allocation expression in the context of O. The analysis first ensures that DO contains an OObject O_C for the newly allocated object. Then, using *dparams*, *Df-New* ensures that each of the actual domain parameters p_i maps to an actual domain D_i in the context of O, where the corresponding formal domain parameter α_i maps to the same D_i but in the context of O_C (Fig. 7). *Df-New* also ensures that the object hierarchy is created such that new ODomains are created for each domain declarations in C according to the auxiliary judgment *ddomains*. Both *dparams* and *ddomains* are recursive auxiliary judgments that consider inheritance (Fig. 9), i.e., the domain may be declared by a class C' that C extends. The base case for the recursion is the class `Object`.

The rule *Df-New* also ensures that FG includes an edge from x to `this` and edges from each of the object allocation arguments $\overline{a}$ to the corresponding fields `this.`$\overline{f}$ (Fig. 7). Next, *Df-New-Unique* handles the case when the owner is `unique`. The rule is similar to *Df-New*, except that it uses the auxiliary judgment $solveUnique$ to find the actual owner domain of O_C. We describe $solveUnique$ later in Fig. 8.

The rules *Df-Read*, *Df-Write* and *Df-Invk* ensure that import and export edges are created using the context O, the receiver O_r and the dataflow OObject as determined by $lookup$ auxiliary judgment (Fig. 8). If the owner domain parameter of T_{label} is `unique`, and $lookup$ cannot find an actual ODomain, the analysis ensures that an OObject is created in a fresh ODomain, as a child of O. All the child OObjects of such an ODomain are flow objects.

Next, *Df-Read*, *Df-Write*, *Df-Invk*, and *Df-Assign* ensure value flow edges are created in FG. For example, *Df-Read*, ensures that the flow graph contains one edge from the receiver r in the context O to the context variable `this` in the context O_r, and a second edge from the field f_k in the context of O_r to x in the context of O. For a method invocation, the rule *Df-Invk* adds annotations to the value flow edges that correspond to the arguments of the invocation and to the return value. For *Df-Read*, *Df-Write*, and *Df-Invk*, the context OObject of the source is different from the context OObject of the destination of a flow edge. On the other hand, for *Df-Assign*, the context OObject remains unchanged for the source and destination.

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ O = C_{\texttt{this}}\texttt{<}\overline{D_O}\texttt{>}\qquad \forall i \in 1..|\overline{p}|\qquad FG, G \vdash_O D_i \in findD(C_{\texttt{this}}{::}p_i) \\ O_C = \langle\ C\texttt{<}\overline{D}\texttt{>}\ \rangle\qquad \{O_C\} \subseteq DO \\ dparams(C, O_C)\qquad \{(O_C, qual(p_i)) \mapsto D_i\} \subseteq DD\qquad G \vdash_O ddomains(C, O_C) \\ \Gamma[\overline{a}] = \overline{T_a}\qquad \{(O, x, p_1) \rightsquigarrow (O_C, \texttt{this}, \alpha_0), (O, \overline{a}, owner(\overline{T_a})) \rightsquigarrow (O_C, \texttt{this}.\overline{f}, owner(\overline{T}))\} \subseteq FG \\ \forall m \in \overline{md}.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, FG, G \vdash_{O_C} e_R \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{a})}\ [\textsc{Df-New}]$$

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ O = C_{\texttt{this}}\texttt{<}\overline{D_O}\texttt{>}\qquad FG, G \vdash_O D_1 \in uniqueDomains(C)\qquad \forall i \in 2..|\overline{p}|\ FG, G \vdash_O D_i \in findD(C_{\texttt{this}}{::}p_i) \\ O_C = \langle\ C\texttt{<}\overline{D}\texttt{>}\ \rangle\qquad \{O_C\} \subseteq DO \\ dparams(C, O_C)\qquad \{(O_C, qual(p_i)) \mapsto D_i\} \subseteq DD\qquad G \vdash_O ddomains(C, O_C) \\ \Gamma[\overline{a}] = \overline{T_a}\qquad \{(O, x, p_1) \rightsquigarrow (O_C, \texttt{this}, \alpha_0), (O, \overline{a}, owner(\overline{T_a})) \rightsquigarrow (O_C, \texttt{this}.\overline{f}, owner(\overline{T}))\} \subseteq FG \\ \forall m \in \overline{md}.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, FG, G \vdash_{O_C} e_R \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = \texttt{new}\ C\texttt{<}\texttt{unique}, \overline{p}\texttt{>}(\overline{a})}\ [\textsc{Df-New-Unique}]$$

$$\frac{\begin{array}{c} CT(C_r) = \texttt{class}\ C_r\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'_r\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ \Gamma[r] = T_r = C_r\texttt{<}\overline{p}\texttt{>}\qquad (T_k\ f_k) \in fields(T_r)\qquad FG, G \vdash_O import(T_r, T_k) \\ FG, G \vdash_O O_r \in lookup(T_r)\qquad (T'_k\ f_k) \in \overline{Tf} \\ \{(O, r, owner(T_r)) \rightsquigarrow (O_r, \texttt{this}, \alpha_0), (O_r, f_k, owner(T'_k)) \rightsquigarrow (O, x, owner(T_k))\} \subseteq FG \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x = r.f_k}\ [\textsc{Df-Read}]$$

$$\frac{\begin{array}{c} CT(C_x) = \texttt{class}\ C_x\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'_x\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ \Gamma[x] = T_x = C_x\texttt{<}\overline{p}\texttt{>}\qquad (T_k\ f_k) \in fields(T_x)\qquad \Gamma[r] = T_r\qquad T_r <: T_k\qquad FG, G \vdash_O export(T_x, T_r) \\ FG, G \vdash_O O_x \in lookup(C_x\texttt{<}\overline{p}\texttt{>})\qquad (T'_k\ f_k) \in \overline{Tf}\qquad \{(O, r, owner(T_r)) \overset{\star}{\rightsquigarrow} (O_x, f_k, owner(T'_k))\} \subseteq FG \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x.f_k = r}\ [\textsc{Df-Write}]$$

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}\qquad G = \langle DO, DD, DE\rangle \\ \Gamma[r_0] = C\texttt{<}\overline{p}\texttt{>}\qquad mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T} \to T_R\qquad FG, DO, DD, DE \vdash_O import(C\texttt{<}\overline{p}\texttt{>}, T_R) \\ \Gamma[\overline{a}] = \overline{T_a}\qquad \overline{T_a} <: \overline{T}\qquad FG, DO, DD, DE \vdash_O export(C\texttt{<}\overline{p}\texttt{>}, \overline{T_a}) \\ FG, G \vdash_O O_r \in lookup(C\texttt{<}\overline{p}\texttt{>})\qquad T'_R\ ret\ m(\overline{T_B}\ \overline{x})\ \ T_{this}\ \{ret = e_R; \texttt{return}\ ret;\} \in \overline{md}\qquad i = fresh_i(O, x_0 = r_0.m(\overline{a})) \\ \Gamma[\overline{x}] = \overline{T_x}\qquad \{(O, r_0, p_0) \overset{(_i}{\rightsquigarrow} (O_r, \texttt{this}, \alpha_0), (O, \overline{a}, owner(\overline{T_a})) \overset{(_i}{\rightsquigarrow} (O_r, \overline{x}, owner(\overline{T_x}))\} \subseteq FG \\ \Gamma[ret] = T'_R\qquad \{(O_r, ret, owner(T'_R)) \overset{)_i}{\rightsquigarrow} (O, x_0, owner(T_R))\} \subseteq FG \end{array}}{\Gamma, \Upsilon, FG, G \vdash_O x_0 = r_0.m(\overline{a})}\ [\textsc{Df-Invk}]$$

$$\frac{\Gamma[r] = T_r\qquad \Gamma[x] = T_x\qquad \{(O, r, owner(T_r)) \rightsquigarrow (O, x, owner(T_x))\} \subseteq FG}{\Gamma, \Upsilon, FG, G \vdash_O x = r}\ [\textsc{Df-Assign}]$$

Figure 7: Static semantics of the extraction analysis. We highlight the parts that construct the value flow graph.

The precision of the analysis is provided by the auxiliary judgments *Df-Lookup* (Fig. 8). For a given type $C'\texttt{<}\overline{p}\texttt{>}$ that includes a list of actual domain parameters, $lookup$ returns those OObjects O_k in DO such the class of O_k is C' or one of its subclasses and each domain D_i of O_k corresponds to D'_i, the domain associated with the pair $(O, C_{this}{::}p_i)$ in DD. The second condition increases the precision of the analysis, because $lookup$ selects all the objects in DO of a class C' or a subclass thereof that are in reachable domains, as opposed to all the objects of a given class in DO.

We introduce the rules *Df-Lookup-Lent*, and *Df-Lookup-Unique* for $lookup$ where the owner domain is `lent` or `unique`. These rules use $solveLent$ and $solveUnique$ to determine the actual domain p' and the context O' where p' is defined. We need to include the context O' in the result to be able to determine the actual domain D'_1 corresponding to `lent` or `unique` because the context might be different from the current context O. For example, a class might be instantiated in the context of O where the owner is `unique`. Next, the reference x in the context O is assigned to y and in another context O'. The destination y is declared in an

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad O = C_{\texttt{this}}\!<\overline{D}\!> \quad O_k \in DO \quad O_k = \langle C\!<\overline{D}\!>\rangle \quad C <: C' \\ \forall i \in 1..|\overline{p'}| \quad FG, G \vdash_O D'_i \in findD(C_{\texttt{this}}\!::\!p'_i) \quad D'_i = D_i \end{array}}{FG, G \vdash_O O_k \in lookup\,(C'\!<\overline{p'}\!>)}\,[\textsc{Df-Lookup}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad O = C_{\texttt{this}}\!<\overline{D}\!> \quad O_k \in DO \quad O_k = \langle C\!<\!D_1, \overline{D}\!>\rangle \quad C <: C' \\ \forall i \in 2..|\overline{p'}| \quad FG, G \vdash_O D'_i \in findD(C_{\texttt{this}}\!::\!p'_i) \quad D'_i = D_i \\ FG \vdash (O', x, p') \in solveLent(O, C') \quad O' = C'_{\texttt{this}}\!<\overline{D'}\!> \\ A = \texttt{unique} \Longrightarrow FG \vdash (O'', x, \texttt{unique}) \in findSrcUnique(O', C') \wedge D'_1 = DD[(O'', C'\!::\!\texttt{unique})] \wedge D'_1 = D_1 \\ A = p' \Longrightarrow FG, G \vdash_{O'} D'_1 \in findD(C'_{\texttt{this}}\!::\!p') \wedge D'_1 = D_1 \end{array}}{FG, G \vdash_O O_k \in lookup\,(C'\!<\texttt{lent}, \overline{p'}\!>)}\,[\textsc{Df-Lookup-Lent}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad O = C_{\texttt{this}}\!<\overline{D}\!> \quad O_k \in DO \quad O_k = \langle C\!<\!D_1, \overline{D}\!>\rangle \quad C <: C' \\ \forall i \in 2..|\overline{p'}| \quad FG, G \vdash_O D'_i \in findD(C_{\texttt{this}}\!::\!p'_i) \quad D'_i = D_i \\ FG \vdash (O', y, A) \in solveUnique(O, C') \quad O' = \langle C'_{\texttt{this}}\!<\overline{D'}\!>\rangle \\ A = \texttt{unique} \Longrightarrow FG \vdash (O'', x, \texttt{unique}) \in findSrcUnique(O', C') \wedge D'_1 = DD[(O'', C'\!::\!\texttt{unique})] \wedge D'_1 = D_1 \\ A = p' \Longrightarrow FG, G \vdash_{O'} D'_1 \in findD(C'_{\texttt{this}}\!::\!p') \wedge D'_1 = D_1 \end{array}}{FG, G \vdash_O O_k \in lookup\,(C'\!<\texttt{unique}, \overline{p'}\!>)}\,[\textsc{Df-Lookup-Unique}]$$

$$\frac{\begin{array}{c} FG_P = propagateAll(FG) \\ (O, s, \texttt{unique}) \rightsquigarrow (O', y, \texttt{unique}) \in FG_P \quad s : C \quad C <: C' \quad \nexists (O'', x, \texttt{unique}) \rightsquigarrow (O, s, \texttt{unique}) \in FG_P \end{array}}{FG \vdash (O, s, \texttt{unique}) \in findSrcUnique(O', C')}\,[\textsc{Aux-Find-Unique}]$$

$$\frac{FG_P = propagateAll(FG) \quad (O, x, \texttt{unique}) \rightsquigarrow (O', y, A) \in FG_P \quad y : C \quad C <: C'}{FG \vdash (O', y, A) \in solveUnique(O, C')}\,[\textsc{Aux-Resolve-Unique}]$$

$$\frac{FG_P = propagateAll(FG) \quad (O', x, A) \rightsquigarrow (O, y, \texttt{lent}) \in FG_P \quad x : C \quad C <: C'}{FG \vdash (O', x, A) \in solveLent(O, C)}\,[\textsc{Aux-Resolve-Lent}]$$

$$\frac{\begin{array}{c} G = \langle DO, DD, DE\rangle \quad FG \vdash (O', y, A) \in solveUnique(O, C) \\ A = \texttt{unique} \Longrightarrow (D = \langle D_{id}, C\!::\!\texttt{unique}\rangle \wedge \{(O, C\!::\!\texttt{unique}) \mapsto D\} \subseteq DD) \\ A = p' \Longrightarrow (O' = \langle C'_{\texttt{this}}\!<\overline{D'}\!>\rangle \wedge FG, G \vdash_{O'} D \in findD(C'_{\texttt{this}}\!::\!p')) \end{array}}{FG, G \vdash_O D \in uniqueDomains(C)}\,[\textsc{Aux-UniqueDom}]$$

$$\frac{\begin{array}{c} FG, G \vdash_O O_i \in lookup\,(T_{src}) \\ FG, G \vdash_{O_i} O_j \in lookup\,(T_{label}) \\ \{\langle O_i, O, O_j, \mathit{Imp}\rangle\} \subseteq DE \; G = \langle DO, DD, DE\rangle \end{array}}{FG, G \vdash_O import\,(T_{src}, T_{label})}\,[\textsc{Aux-Import}]$$

$$\frac{\begin{array}{c} FG, G \vdash_O O_i \in lookup\,(T_{dst}) \\ FG, G \vdash_O O_j \in lookup\,(T_{label}) \\ \{\langle O, O_i, O_j, \mathit{Exp}\rangle\} \subseteq DE \; G = \langle DO, DD, DE\rangle \end{array}}{FG, G \vdash_O export\,(T_{dst}, T_{label})}\,[\textsc{Aux-Export}]$$

Figure 8: Rules for resolving `lent`, and `unique`. Auxiliary judgments *import* and *export* ensure dataflow edges are created.

actual domain p'. To determine the actual ODomain for the domain parameter p', the extraction analysis uses the context of y, namely O', not the context of x.

In CryptoApp (Fig. 2), the extraction analysis creates the OObjects `eAct:EActivity` and `act:ViewActivity`, then it analyzes the object allocation expression `new LctMngr<unique>()` in the base class `Activity`, first, in the context of `eAct:EActivity`, and second, in the context of `act:ViewActivity`. In the first case, $solveUnique$(`eAct:EActivity`, `LctManager`) returns (`eAct:EActivity`, `lm`, `V`), and $uniqueDomains$ returns the ODomain `VIEW`. In the second case, $solveUnique$(`act:ViewActivity`, `LctManager`) returns the empty set and the analysis creates a flow object of type `LctMngr` in a fresh ODomain. Hence, the analysis creates two objects of type `LctMngr`, one is a flow object, the other is an actual OObject in the ODomain `VIEW`. In turn, both objects have a child of type `Location`. When the analysis interprets the method invocation `lm.getLastLocation()` in the context of `eAct:EActivity`, it invokes *lookup* which ensures that only the object `result:LctManager` in the domain `VIEW` is used as a source. The child of `result:LctManager` is then referred to by the dataflow edge from `eAct:Eactivity` to `i:Intent` (Fig. 2).

The analysis is sound. We proved the soundness of the analysis that extracts an OOG with dataflow edges that refer

$$\frac{\forall(\texttt{domain}\ d_j) \in \overline{dom} \qquad D_j = \langle D_{id_j},\ C{::}d_j\rangle \qquad \{(O_C, C{::}d_j) \mapsto D_j\} \subseteq DD \qquad G = \langle DO, DD, DE\rangle \qquad G \vdash_O ddomains(C', O_C)}{G \vdash_O ddomains(C, O_C)}[\text{AUX-DOM}]$$

$$\frac{}{G \vdash_O ddomains(\texttt{Object}, O_C)}[\text{AUX-OBJ1}]$$

$$\frac{CT(C) = \texttt{class}\ C{<}\overline{\alpha}, \overline{\beta}{>}\ \texttt{extends}\ C'{<}\overline{\delta}{>}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \} \qquad G = \langle DO, DD, DE\rangle \qquad O_C = \langle C{<}\overline{D}{>}\rangle \qquad \forall \alpha_j \in params(C)\ \{(O_C, C{::}\alpha_j) \mapsto D_j\} \subseteq DD \qquad G \vdash_O dparams(C', O_C)}{G \vdash_O dparams(C, O_C)}[\text{AUX-ALPHA}]$$

$$\frac{CT(\texttt{Object}) = \texttt{class Object}{<}\alpha_o{>}\ \{\ \}}{G \vdash_O dparams(\texttt{Object}, O_C)}[\text{AUX-ALPHA1}]$$

$$\frac{O = \langle C{<}\overline{D_O}{>}\rangle \qquad n : C_n{<}\overline{p}{>} \qquad FG, G \vdash_O O_i \in lookup\ (C_n{<}\overline{p}{>}) \qquad D_i = DD[(O_i, C_n{::}d)]}{FG, G \vdash_O D_i \in findD(C{::}n.d)}[\text{DF-FINDD-PUBLIC}]$$

$$\frac{O = \langle C{<}\overline{D_O}{>}\rangle \qquad D_i = DD[(O, C{::}d_i)]}{FG, G \vdash_O D_i \in findD\ (C{::}\texttt{this}.d_i)}[\text{DF-FINDD-THIS}] \qquad \frac{O = \langle C{<}\overline{D_O}{>}\rangle \qquad D_i = DD[(O, C{::}\alpha_i)]}{FG, G \vdash_O D_i \in findD\ (C{::}\alpha_i)}[\text{DF-FINDD}]$$

$$\frac{}{FG, G \vdash_O D_{\texttt{SHARED}} \in findD\ (::\texttt{shared})}[\text{DF-FINDD-SHARED}]$$

Figure 9: Auxiliary judgments for inference rules in Fig. 7.

function $summarize(FG)$
  $FG* = FG$
  $WL = \{(O_1, x_1, B_1) \overset{a}{\rightsquigarrow} (O_2, x_2, B_2) \in FG \text{ s.t. } a \text{ is } (_i\}$
  while $WL \neq \emptyset$ **do**
    remove $e_1 : (O_1, x_1, B_1) \overset{a_1}{\rightsquigarrow} (O_2, x_2, B_2)$ from WL
    if a_1 is $(_i$ **then**
      for $e_2 : (O_2, x_2, B_2) \overset{a_2}{\rightsquigarrow} (O_3, x_3, B_3) \in FG*$ **do**
        if $e_3 = concat(e_1, e_2) \notin FG*$ **then**
          add e_3 to $FG*$ and WL
    else
      if a_1 is $\bullet$ or $\star$ **then**
        for $e_2' : (O_0, x_0, B_0) \overset{a_2'}{\rightsquigarrow} (O_1, x_1, B_1) \in FG*$ **do**
          if $e_3' = concat(e_2', e_1) \notin FG*$ **then**
            add e_3' to $FG*$ and WL
  return $FG*$

$concat((O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \rightsquigarrow (O_2, z, B_3)) =$
$= (O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, z, B_3)$

$concat((O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{)_i}{\rightsquigarrow} (O_1, z, B_3)) =$
$(O_1, x, B_1) \rightsquigarrow (O_1, z, B_3)$

$concat((O_1, x, B_1) \overset{(_i}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{\star}{\rightsquigarrow} (O_3, z, B_3)) =$
$(O_1, x, B_1) \overset{\star}{\rightsquigarrow} (O_3, z, B_3)$

Figure 10: The algorithm $summarize$ inspired from [8, Fig. 4.16].

to dataflow objects [20]. Flow objects maintain the unique representative invariant since the analysis creates a fresh ODomain for each flow object. Multiple dataflow edges can then refer to the same flow object. We conjecture but do not prove that the analysis that extracts an OOG with flow objects is also sound.

4.4 Flow Graph Analysis

Since a value might pass linearly through several assignments, the rules $solveLent$ and $solveUnique$ use another flow graph , FG_P, where the transitive flow is propagated, as direct flow edges may not exist in FG. FG_P is computed in two steps. First, an algorithm summarizes FG into a summary graph $FG*$, then another algorithm propagates the transitive flow for nodes in $FG*$ and computes FG_P. The algorithm $summarize$ does not consider the order of the assignments and the Flow Graph Analysis is therefore flow-insensitive. Still, by using the annotations of the flow edges, the $summarize$ matches the parentheses with the same value i and the Flow Graph Analysis is call-site context-sensitive.

In order to better understand the Flow Graph Analysis, we introduce three similar running examples that each creates objects of the same type. The first two examples show how the Flow Graph Analysis distinguishes between these objects, and the extraction analysis avoids creating false positive dataflow edges. The last example highlights one limitation of the extraction analysis if developers overuse `lent` and `unique`. For each example, Figure 11 also shows the extracted OGraph that has OObjects such as $\langle$`A<DATA,DOM1>`$\rangle$ and $\langle$`A<DATA,DOM2>`$\rangle$. The flow graph FG (not shown) has nodes such as ($\langle$`A<DATA,DOM1>`$\rangle$, `f`, `F`) and ($\langle$`A<DATA,DOM2>`$\rangle$, `f`, `F`) and the following flow edges:

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{a1}, \texttt{DATA}) \overset{(_{11}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{this}, \texttt{owner})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n1}, \texttt{DOM1}) \overset{(_{11}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{num}, \texttt{F})$

$(\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{num}, \texttt{F}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{f}, \texttt{F})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{a2}, \texttt{DATA}) \overset{(_{14}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{this}, \texttt{owner})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{DOM2}) \overset{(_{14}}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{num}, \texttt{F})$

$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{num}, \texttt{F}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F})$

$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F}) \rightsquigarrow (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{ret}, \texttt{F})$

$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{ret}, \texttt{F}) \overset{)_{15}}{\rightsquigarrow} (\langle \texttt{Main<SHARED>}\rangle, \texttt{dest}, \texttt{lent})$

The algorithm $summarize$ (Fig. 10) computes $FG*$ such that it concatenates two edges where the first edge has the same destination and the source of the second edge. The algorithm matches pair of edges $(_i$ and $)_i$ that have the same value for i. If the invocation $(_i$ is followed by an assignment, the algorithm propagates the invocation. The $\star$ annotation means that a method stores a value in a field. Because other methods can use the value of the field, the $\star$ annotation cancels the effect of an $(_i$ annotation and the concatenated edge keeps the $\star$ annotation.

For the example in Fig. 11, $summarize$ adds the following edges.

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n1}, \texttt{DOM1}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{f}, \texttt{F})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{DOM2}) \overset{\star}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F})$

The flow graph $FG*$ has a transitive flow:

$(\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F}) \rightsquigarrow \ldots \rightsquigarrow (\langle \texttt{Main<SHARED>}\rangle, \texttt{dest}, \texttt{lent})$

Therefore, by distinguishing between OObjects of the same type, but with different lists of ODomains $\overline{D}$, the Flow Graph Analysis avoids a false positive.

$(\langle \texttt{A<DATA,DOM1>}\rangle, \texttt{f}, \texttt{F}) \rightsquigarrow \ldots \rightsquigarrow (\langle \texttt{Main<SHARED>}\rangle, \texttt{dest}, \texttt{lent})$

According to *Aux-Resolve-Lent*, the Flow Graph Analysis resolves `lent` to `F` in the context of $\langle$`A<DATA,DOM2>`$\rangle$. Then, according to *Aux-Lookup-Lent*, $lookup$ returns only $\langle$`Integer<DOM2>`$\rangle$ and not $\langle$`Integer<DOM1>`$\rangle$, which would introduce a false positive dataflow edge in the OGraph.

Related work [8, Fig. 4.16] treats fields differently from local variables, and requires a separate may-alias analysis for finding variables that may alias the same receiver object to substitute a field `this.f` to `a1.f` or `a2.f`. In a flow node (O, `this.f`, B), the receiver `this` refers to O, so no separate may-alias analysis is required.

To compute the index i used on the value flow edges, a naive flow analysis could use the line number of the method invocation expression in the code. If our analysis were to use such a value for i in the rule *Df-Invk*, it would use the same value of i for different values of O, which would create false positive flow edges. Instead, *Df-Invk* uses $fresh_i$ to generate distinct values for i based on the pair (O, $x = r.m(\overline{y})$) and allows the analysis to distinguish between the same method invocation but in different contexts.

For example, consider Fig. 11 that shows code fragments where an object of type `B` creates an object of type `A` and invokes the method `set` to assign the value for the field f. The analysis creates two objects `a:A` in different domains for the same object allocation expression `new A()`. The assignment of the field value also occurs at the same method invocation `a.set(n)`. Due to different values returned by $fresh_i$, the analysis considers the method invocation `a.set(n)` twice, first in the context of `b1:B` and second in the context of `b2:B`. If the analysis were to consider only the line number of the method invocation as the value of i, $FG*$ would have a false positive transitive flow from ($\langle$`A<DATA,DOM1>`$\rangle$,`f`,`F`) to ($\langle$`Main<SHARED>`$\rangle$,`dest`,`lent`).

To compute the transitive flow, the analysis uses the algorithm $propagate$ (Fig. 12), which takes as input the flow graph $FG*$ returned by the $summarize$ algorithm and a source s. The output of $propagate$ is a flow graph FG_P with the same nodes as FG and $FG*$, but more edges. To compute FG_P, $propagate$ uses a worklist algorithm and the method $concat'$, which concatenates two edges where the destination of the first edge is the source of the first edge. For concatenation, $propagate$ uses two value flow annotation *Call* and *nCall*. During the initialization, the annotation *Call* corresponds to flow edges with annotation $(_i$, and *nCall* correspond to the other annotations. If the second argument of $concat'$ is an edge annotated $(_i$, the result is an edge annotated *Call*. Otherwise, either no edge is added, or the concatenation propagates the *nCall* annotation.

Since we are using the result of the algorithm to resolve `lent` and `unique`, the propagation occurs only if any node of the two edges have a domain B that is `lent`, `unique`, or a public domain $n.d$. We also include $n.d$ because a transitive value flow may exist from (O_1, x,$n.d$) to (O_1, y,`lent`), and further to (O_2, z,`lent`), where the variables n and z are in different contexts O_1 and O_2. Here, the analysis needs to perform an extra step and find the edge from (O,ret,`this`.d) to (O_1, x,$n.d$) such that `this`.d is a domain of O.

For Fig. 11, where the variable `n2` is declared `unique`, the algorithm $propagate$ adds the flow edges:

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{num}, \texttt{F})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{f}, \texttt{F})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{A<DATA,DOM2>}\rangle, \texttt{ret}, \texttt{F})$

$(\langle \texttt{Main<SHARED>}\rangle, \texttt{n2}, \texttt{unique}) \overset{Call}{\rightsquigarrow} (\langle \texttt{Main<SHARED>}\rangle, \texttt{dest}, \texttt{lent})$

Then, *Df-New-Unique* invokes *findD*(`Main::unique`) that in turn invokes *solveUnique*($\langle$`Main<SHARED>`$\rangle$,`Integer`). Here, `unique` is resolved to the domain parameter `F` that is bound to `DOM2` in the context of $\langle$`B<DATA,DOM2>`$\rangle$. Therefore, *Df-New-Unique* creates the OObject $\langle$`Integer<DOM2>`$\rangle$.

```
class A<owner,F> {
  Integer<F> f;
  void set(Integer<F> num) { f = num;}
  Integer<F> get() { return f; }
}
class Main<owner>{
  void main() {
   domain DATA,DOM1,DOM2;
    Integer<DOM1> n1= new Integer(-1);
    A<DATA,DOM1> a1 = new A();
    a1.set(n1);
    Integer<DOM2> n2 = new Integer(2);
    A<DATA,DOM2> a2 = new A();
    a2.set(n2);
    Integer<lent> dest = a2.get();
    dest.compareTo(n1);
}
```

```
class A<owner,F> {
  Integer<F> f;
  void set(Integer<F> num) { f = num;}
  Integer<F> get() { return f; }
}
class B<owner, F> {
  domain OWNED;
  A<OWNED,F> A a = new A();
  void assign(Integer<F> n) {
    a.set(n);
  }
}
class Main<owner>{
  void main() {
   domain DATA,DOM1,DOM2;
   Integer<DOM1> n1 = new Integer(-1);
   B<DATA,DOM1> b1 = new B();
   b1.assign(n1);
   Integer<unique> n2 = new Integer(2);
   B<DATA,DOM2> b2 = new B();
   b2.assign(n2);
   Integer<lent> dest = b2.a.get();
   dest.compareTo(n1);
  }
}
```

```
class A<owner,F> {
  Integer<F> f;
  void set(Integer<F> num) { f = num;}
  Integer<F> get() { return f; }
}
class B<owner, F> {
  domain OWNED;
  A<OWNED,F> A a = new A();
  void assign(Integer<F> n) {
    a.set(n);
  }
}
class Main<owner>{
  void main() {
   domain DATA,DOM1,DOM2;
   Integer<unique> n1 = new Integer(-1);
   B<DATA,DOM1> b1 = new B();
   b1.assign(n1);
   Integer<unique> n2 = new Integer(2);
   B<DATA,DOM2> b2 = new B();
   b2.assign(n2);
   Integer<lent> dest = b2.a.get();
   dest.compareTo(n1);
  }
}
```

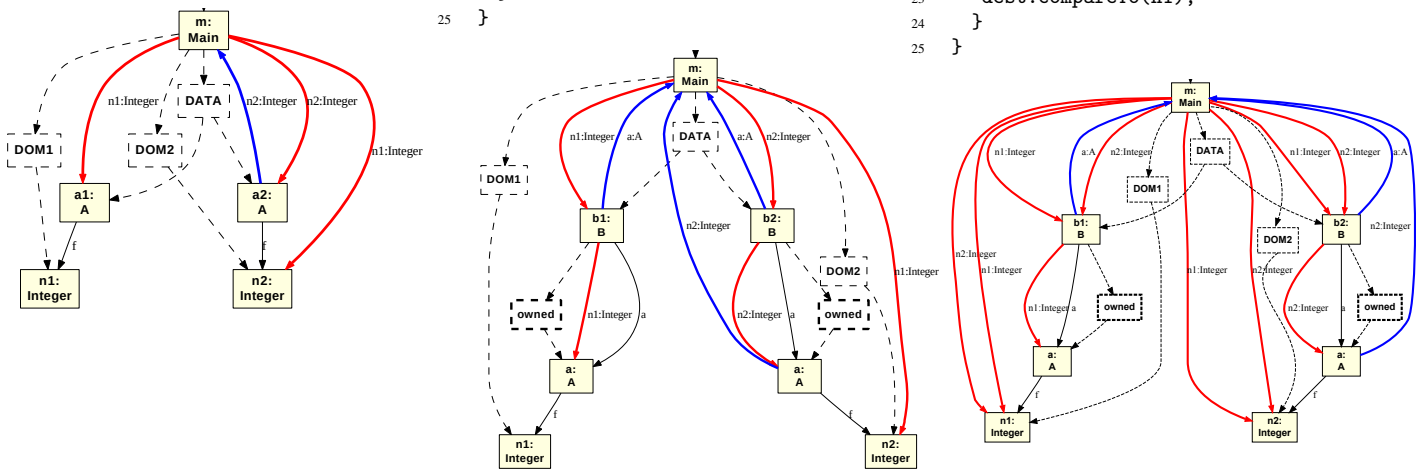

Figure 11: The analysis distinguishes between different instances of the same type `A`, and show a dataflow edge that refers to `n2:Integer` from `a2:A` to `m:Main` and not from `a1:A` to `m:Main`. It is not necessary that the objects of type `A` are created for different object allocation expressions in the code (left vs. middle). However, if developers overuse `lent` and `unique`, the analysis may add false positive edges (right).

4.5 Implementation

The extraction analysis follows the algorithm $runAnalysis$ (Fig. 13) and starts from the root expression that is provided as input along with all the class declarations in the program. For initialization, $runAnalysis$ uses the initial values FG_0, DD_0, DD_0, and DE_0, defined in Section 4.3. The algorithm has two iterations. First, the analysis creates the object hierarchy with OObjects and ODomains only, and collects the nodes and edges of the value flow graph FG. The iteration terminates when it reaches a fixed point, i.e., no flow nodes and flow edges are added to FG, and no new objects and domains are added to the OGraph. In this iteration, the extraction analysis does not compute dataflow edges and flow objects, because computing dataflow edges requires FG, which further depends on DO. Otherwise, the analysis may extract a spurious number of dataflow edges that refer to these flow objects when it is not able to resolve `lent` and `unique` based on the intermediate FG. In the second iteration, the analysis invokes the algorithm $propagateAll$, creates the flow objects, and computes the dataflow edges. The second iteration terminates when the algorithm $propagateAll$ no longer adds edges to FG_P and no objects, domains, or dataflow edges are added to the OGraph.

In Section 4.3, we described the analysis using a constraint-based specification. The implementation uses transfer functions. Each function $accept$ is a transfer function that is equivalent to the corresponding inference rule. It takes as arguments all the elements on the left-hand-side

function $propagate(FG*, s)$
  $FG_P = \emptyset$
  $WL = \emptyset$
  for $e : (O_1, s, B_1) \overset{a}{\rightsquigarrow} (O_2, x_2, B_2) \in FG*$ **do**
    if $\{B1, B2\} \cap \{\texttt{lent}, \texttt{unique}, n.d\} \neq \emptyset$ **then**
      if a_1 is $(_i$ **then**
        add $(O_1, s, B_1) \overset{Call}{\rightsquigarrow} (O_2, x_2, B_2)$ to FG_P and WL
      else
        add $(O_1, s, B_1) \overset{nCall}{\rightsquigarrow} (O_2, x_2, B_2)$ to FG_P and WL
  while $WL \neq \emptyset$ **do**
    remove $e_1 : (O_1, x_1, B_1) \overset{nCall|Call}{\rightsquigarrow} (O_2, x_2, B_2)$ from WL
    for $e_2 : (O_2, x_2, B_2) \overset{a}{\rightsquigarrow} (O_3, x_3, B_3) \in FG*$ **do**
      if $e_3 = concat'(e_1, e_2) \notin FG_P$ **then**
        add e_3 to $FG*$ and WL
  return FG_P

function $propagateAll(FG)$
  $FG* = summarize(FG)$;
  for $(O, x, B) \in FG*$ **do**
    $FG_P = FG_P \cup propagate(FG*, x)$
  return FG_P

$concat'((O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{(_i}{\rightsquigarrow} (O_2, z, B_3)) =$
$= (O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_2, z, B_3)$

$concat'((O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{\star|\bullet|)_i}{\rightsquigarrow} (O_1, z, B_3))$
NO Edge

$concat'((O_1, x, B_1) \overset{nCall}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{(_i}{\rightsquigarrow} (O_3, z, B_3)) =$
$= (O_1, x, B_1) \overset{Call}{\rightsquigarrow} (O_3, z, B_3)$

$concat'((O_1, x, B_1) \overset{nCall}{\rightsquigarrow} (O_2, y, B_2), (O_2, y, B_2) \overset{\star|\bullet|)_i}{\rightsquigarrow} (O_3, z, B_3)) =$
$= (O_1, x, B_1) \overset{nCall}{\rightsquigarrow} (O_3, z, B_3)$

Figure 12: . The algorithm $propagate$ adds more edges for nodes with variables declared `lent` or `unique`.

function $runAnalysis(e_{root}, CT)$
  $DO = DO_0, DD = DD_0$
  $DE = DE_0$ $FG = FG_0, \Gamma = \emptyset, \Upsilon = \emptyset$
  $G = \langle DO, DD, DE\rangle$ $G' = \langle DO', DD', DE'\rangle$
  $G' \subseteq G \Leftrightarrow DO' \subseteq DO \wedge DD' \subseteq DD \wedge DE' \subseteq DE$
// Iteration 1
  $\langle FG', G'\rangle = accept(\langle \Gamma, \Upsilon, FG, G, O_{world}\rangle, e_{root})$
  while $FG' \subseteq FG \wedge G' \subseteq G$ **do**
    $\langle FG, G\rangle = \langle FG', G'\rangle$
    $\langle FG', G'\rangle = accept(\langle \Gamma, \Upsilon, FG, G, O_{world}\rangle, e_{root})$
// summarize FG
  $FG_P = propagateAll(FG)$
// Iteration 2
  $\langle FG', G'\rangle = accept(\langle \Gamma, \Upsilon, FG_P, G, O_{world}\rangle, e_{root})$
  $FG_P = propagateAll(FG')$
  while $FG' \subseteq FG_P \wedge G' \subseteq G$ **do**
    $G = G'$
    $\langle FG', G'\rangle = accept(\langle \Gamma, \Upsilon, FG_P, G, O_{world}\rangle, e_{root})$
    $FG_P = propagateAll(FG')$
  return G

function $accept(\langle \Gamma, \Upsilon, FG, G, O\rangle, x = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}())$
  $\langle DO, DD, DE\rangle = G$
  $O_C = \langle C\texttt{<}\overline{D}\texttt{>}\rangle, DO' = DO \cup \{O_C\}$ . . . //as in *Df-New*
  for $m \in \overline{md}.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$ **do**
    if $C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon$ **then**
      $\Gamma' = \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}$
      $\Upsilon' = \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}$
      $G' = \langle DO', DD', DE'\rangle$
      $\langle FG', G'\rangle = accept(\Gamma', \Upsilon', FG', G', O_C, e_R)$
  return $\langle FG', G'\rangle$
function $accept(\langle \Gamma, \Upsilon, FG, G, O\rangle, x = \ldots)$

Figure 13: . The algorithm $runAnalysis$ describes the steps the analysis follows.

of the turnstile in the constraint-based specification and the context OObject O. For each declarative clause, $subset \subseteq set$ in the constraint-based specification, the transfer function has an statement $set' = subset \cup set$. In particular, for each object allocation expression, the function $accept$ creates a new OObject O_C. Hence the clause that ensures $\{O_C\} \subseteq DO$ in *Df-New* becomes the statement $DO' = \{O_C\} \cup DO$ in the transfer function. Next, $accept$ abstractly interprets each method declaration of the instantiated class, and of the class it extends, recursively. The algorithmic description omits the remaining details of *Df-New* (Fig. 7). The transfer functions for field read, field write, method invocation, and assignment are all similar.

We implemented the extraction analysis using the Crystal framework [12]. The implementation also supports some features of Java that are not part of the FDJ abstract syntax such as interfaces and generics.

5. Discussion

The result of the extraction analysis on CryptoApp is the OGraph in Fig. 2. We also evaluate the analysis on an open-source Android application Universal Password Manager for Android (UPMA) to which one of the authors of the paper added Ownership Domain annotations. UPMA is a 4 KLOC app that allows Android users to manage their passwords using a database encrypted in a file.

The analysis extracts the UPMA OOG from which we show the most interesting fragments (Fig. 14). We elided the substructure of some objects and included only the edges that provide support for the following observations.

The analysis extracts dataflow edges that refer to objects and distinguishes between objects of the same type. The extracted OOG shows distinct objects of type `File`, where `dest:File` represents the file that stores the encrypted database, while `cert:File` represents a file that stores the encryption keys. The analysis shows a dataflow edge from `act:UPMApplication` to `accts:FullAccountList` that refer to `dest:File`. This edge is due to a feature in UPMA that allows the user to copy the database file to the external memory of the phone. If the dataflow edge were to show only the type of the object, it would be misleading because such a dataflow edge would indicate that the file containing the encryption keys is also copied and the encryption keys are exposed, which is not the case.

The analysis extracts several flow objects. While both `File` objects are nodes in the OOG, the analysis also extracts several flow objects. For example, a flow object `b:byte[]` is referred from

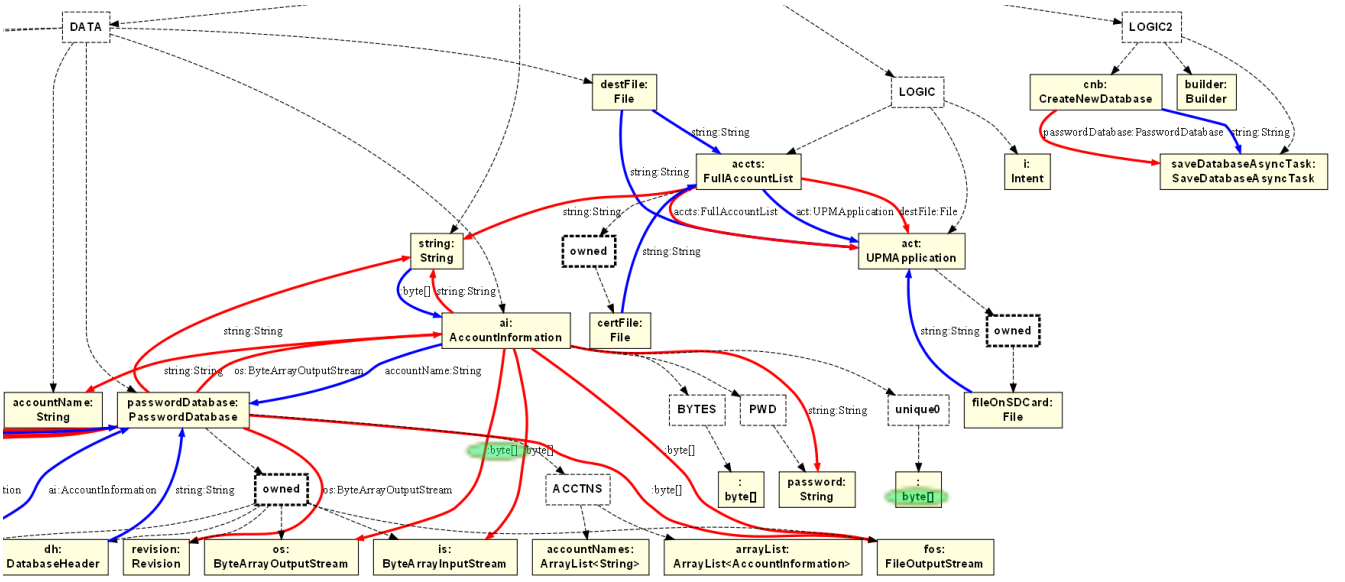

Figure 14: A fragment of the extracted UPMA OOG shows dataflow objects of type `File`, and a flow object of type `byte[]` from `ai:AccountInformation` to `os:ByteArrayOutputStream`.

the dataflow edge from `ai:AccountInformation` to `os:ByteArrayOutputStream`. This flow object is created during encryption when the object `pd:PasswordDatabase` translates `password:String` of `ai:AccountInformation` from plain text to a binary representation.

More precise ownership types produce more precise results. Developers can control the precision of the extracted graphs by placing objects in different domains. For example, the class `SaveDatabaseAsynkTask` has a field of type `Activity`. The field is the receiver of a method invocation `activity.getString()` which returns the flow object. The field is declared in `owner` domain. If all the instances of subclasses that extend `Activity` were in `LOGIC`, the analysis would show dataflow edges to the objects `act:UPMApplication` and `accts:FullAccountList`, which are of types that extend `Activity`. However, such edges would be false positives. One manual workaround is to have the developer define a separate domain, `LOGIC2`, and place these object in that domain, so the analysis does distinguish between these objects and `cnb:CreateNewDatabase`, thus avoiding these false positives (Fig. 14).

Some dataflow edges are missing in the presence of Java reflection. All objects of type `Activity` should have a dataflow self-edge that refers to an object of type `Intent`, which means that each object of type `Activity` sends an object of type `Intent` to the framework which uses reflection to find the correct destination. Since the static analysis does not handle reflection, the UPMA OOG does not show these edges. One possible workaround is for developers to summarize reflective calls as it is commonly done in static analysis tools [4].

In addition to these observations, we also formulate several unanswered questions.

What are the benefits of a value flow graph? The previous extraction analysis focused on points-to edges [1], and we observed that for some of the systems, local variables and formal method parameters are often declared `lent` or `unique` [22]. Therefore, without the value flow graph, the analysis would be unable to extract the dataflow edges for expressions involving these variables. As a result, the OOG with dataflow edges would be unsound. In addition, we believe the analysis presented in this paper can reduce the effort of annotating code. Since previously developed extraction analysis [1] did not show objects for allocation expressions declared `unique`, the annotators spent additional effort replacing `lent` and `unique` with actual domains. However, this can make the annotated code less reusable, in particular when it is part of a framework or a library.

Are the constraints on `lent` and `unique` too strict? The analysis assumes that `lent` and `unique` are used as the first domain parameter only. In practice, we observed that rarely a domain parameter, other than the first one, is `unique`. Such cases occur in the presence of collections such as `HashMap` where the first parameter specifies the owner of the collection, the second parameter specifies the domain of the keys, and the third parameter specifies the domain of the values. Adding Ownership Domains annotations in the presence of the factory-method design pattern is challenging. For example, the factory object creates an object of type `HashMap` that represents a board with figures as values and the positions on the board as keys, the domain of the figures is not known

```
class Factory<owner,D> { // Version 1
  HashMap<unique, Position<D>, Figure<unique>> create() {
    HashMap<unique, Position<D>, Figure<unique>> map;
    map = new HashMap<Position,Figure>();
    Position<D> p = new Position(row,col);
    Figure<unique> fig = new Figure();
    map.put(p,fig);
    return map;
  }
}
class Factory<owner,D> { // Version 2
  //F is a method domain parameter
  HashMap<unique, Position<D>, Figure<F>> create<F>() {
    HashMap<unique, Position<D>, Figure<F>> map;
    Figure<F> fig = new Figure();
  }
}
class Board<owner,D> {
  domain MAPS, FIGS;
  Factory<owner,D> factory;
//unique resolves to MAPS, but third domain param is FIGS
  HashMap<MAPS, Position<D>, Figure<FIGS>> fMap;
  fMap= factory.create();
 // alternative: use method domain parameter
 // fmap = factory.create<FIGS>()
}
```

Figure 15: Code where `unique` is used as the third domain parameter. Instead, a method domain parameter can be used.

to the factory object and is declared `unique`. However, this means that the third domain parameter of the map is also `unique`. This is problematic since both the object `fmap` and the client code have a reference to the values stored in the map. Also, if multiple domain parameters are `unique` it is not necessary that `unique` resolves to the same actual domain for all the domain parameters (Fig. 15). One solution may be to support another extension to Ownership Domains that allows a method to be parameterized with domain parameters [3], and use such a domain parameter instead of `unique` when the list of figures is created. We think we can support this extension in the analysis without major changes.

6. Conclusion and Future Work

The paper presents a static analysis that uses abstract interpretation to extract an OOG with dataflow edges that refer to objects. The analysis also extracts flow objects that are passed linearly. The analysis uses the Ownership Domains type system to achieve aliasing precision and extracts a hierarchy of objects. It also uses a domain-sensitive value flow analysis to resolve the domains of variables declared as `lent` or `unique`. We evaluate the extraction analysis on an open-source Android application and the results indicate that flow objects exist in practice.

This paper is part of ongoing work on extracting a sound approximation of the runtime architecture. In a related paper [21], we use the extracted OOG to support Architectural Risk Analysis [10] and find architectural flaws such as information disclosure. We also plan to evaluate the analysis on more systems.

References

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

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

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

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, 2002.

[5] M. Gargenta. *Learning Android*. O'Reilly Media, 2011.

[6] IBM. T.J. Watson Libraries for Analysis (WALA), 2012.

[7] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[8] Y. Liu. *Practical Static Analysis Framework For Inference Of Security-Related Program Properties*. PhD thesis, Rensselaer Polytechnic Institute, 2010.

[9] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 50–56, 2008.

[10] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[11] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *TOSEM*, 14(1), 2005.

[12] PLAID Research Group. The Crystal Static Analysis Framework, 2009. `http://code.google.com/p/crystalsaf`.

[13] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.

[14] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.

[15] M. Sridharan, S. Chandra, J. Dolby, S. Fink, and E. Yahav. Alias analysis for object-oriented programs. Springer LNCS 7850:156–232, 2013.

[16] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, 2005.

[17] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[18] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[19] R. Vanciu and M. Abi-Antoun. Ownership Object Graphs with Dataflow Edges. In *WCRE*, 2012.

[20] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Objects and other Flow Objects. `http://www.cs.wayne.edu/~mabianto/tech_reports/VA13_TR.pdf`, 2013.

[21] R. Vanciu and M. Abi-Antoun. Finding architectural flaws using constraints. In *ASE*, 2013.

[22] R. Vanciu and M. Abi-Antoun. Object Graphs with Ownership Domains: an Empirical Study. Springer LNCS 7850:109–155, 2013.