Extracting Dataflow Communication from Object-Oriented Code

Radu Vanciu **Marwan Abi-Antoun**

Posted: Oct. 15, 2011; Last updated: Mar. 9, 2012

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

Object graphs help developers understand the runtime structure of an object-oriented system, in terms of objects and their runtime relations (points-to, call, or dataflow, depending on the intent of the diagram). Ideally, an object graph is sound and shows all possible objects and the relations between them. The object graph should also be hierarchical to scale and convey architectural abstraction. Achieving soundness requires a static analysis, but architectural hierarchy is not available in code written in general-purpose programming languages. To achieve hierarchy in a statically extracted object graph, we leverage ownership types in the code. We then abstractly interpret the annotated program and extract a global, sound, hierarchical object graph with dataflow communication edges that show the flow of objects due to field reads, field writes, and method invocations. We formalize the static analysis using a constraint-based specification and prove that the object graph is sound.

Keywords: hierarchical object graphs, dataflow communication, ownership domains

Document History

Date	Version	Description
Oct. 15, 2011	1.0	Initial posting
Feb. 15, 2012	1.1	Added to discussion of recursion (Section 4.2)
		Fixed Df-New
		Included a detailed abstract interpretation of QuadTree
Mar 9, 2012	1.2	Simplified static rules
		Removed O_{id} from OObject definition
		Distinguished import and export edges in OEdge definition

Contents

List of Figures

1 Introduction

During software evolution, reverse-engineered diagrams of the code structure and of the runtime structure help developers to understand the system in order to modify it. Diagrams of the code structure are supported by many tools. Diagrams of the runtime structure, however, are more challenging and less mature.

One challenge with diagrams of the runtime structure is soundness, i.e., showing all possible objects and all possible relations between them. Achieving soundness requires static analysis since, by definition, dynamic analysis shows partial diagrams from a finite number of executions. Another challenge is to create a graph that scales and supports program understanding. A flat object graph, with its profusion of objects, does not meet this challenge. One solution is to use hierarchy, which provides both high-level and detailed understanding.

Architectural hierarchy is not observable in legacy object-oriented code, so we follow a previous approach [2] and use ownership types in the code, specifically, the Ownership Domains type system [3]. To support legacy code, we define annotations that implement the type system, using available language support for annotations. Developers use the annotations and specify, within the code, their design intent in the form of strict encapsulation, logical containment and architectural tiers. These annotations enable a static analysis to extract a sound, global, hierarchical Ownership Object Graph (OOG) [2]. An OOG provides architectural abstraction by ownership hierarchy and by types, where architecturally significant objects appear near the top of the hierarchy and data structures are further down.

In related work, we evaluated in a controlled experiment if OOGs, as diagrams of the runtime structure, help developers with program comprehension during coding activities, and thus complement widely-used class diagrams [5, 4]. We found that developers who used OOGs succeeded on code modification tasks, took less time, or explored less irrelevant code compared to developers who used only class diagrams or who just explored the code. In our previous experiment, developers wondered why the OOG did not show some relations between objects. The OOG showed only points-to edges due to field references that capture persistent relations between objects. In addition to points-to edges, developers need usage edges that capture more transient relations between objects [10]. In this paper, we add to the OOG usage edges that make visually obvious the flow of objects in the program, and that we refer to as dataflow communication.

For instance, in object-oriented code that implements the Observer design pattern, understanding “what” gets notified during a change notification is crucial for understanding the system. “What” does not usually mean a class, “what” means a particular instance. Indeed, with many design patterns, developers need to understand the various instances in the system, and object graphs give insights into instances better than class diagrams. To understand what instances point to what other instances, points-to edges are useful. To understand not just “what” gets notified but also “what kind” of notification the subject of the notification sends to its observers, usage edges may be useful.

Contributions. In this paper, we propose a static analysis to extract a hierarchical object graph with usage edges showing dataflow communication. Our contributions are:

- We formalize the analysis using a constraint-based specification, showing the static and dynamic semantics;
- We prove the soundness of the extracted object graph;
- We evaluate our analysis on an extended example; we compare an OOG with dataflow edges to a diagram of the runtime structure with dataflow communication drawn by an expert, and to an OOG with points-to edges.

Outline. The rest of this report is organized as follows. In Section 2, we describe the challenges of designing a static analysis that extracts a runtime structure. In Section 3, we define dataflow communication. In Section 4, we formalize our analysis, and prove its soundness. We introduce a small example, and describe the analysis on a worked example in Section 6. We discuss related work in Section 7 and conclude.

2 Challenges of Static Analysis

We first discuss the challenges of extracting object graphs statically. At runtime, the structure of an object-oriented program can be represented as a Runtime Object Graph (ROG), where nodes represent objects, i.e., instances of classes, and edges represent relations between objects, such as one object calling another object's methods. A sound static analysis extracts an object graph that approximates all possible ROGs, for any program execution. We represent the extracted object graph as an OGraph, where nodes are OObjects and edges are OEdges. An OObject is a canonical object that represents multiple runtime objects. Similarly, an OEdge is a canonical edge that represents runtime dataflow communication between the corresponding runtime objects. An OGraph has the following requirements:

Object soundness. The OGraph must show a unique representative for each runtime object. While one OObject can represent multiple runtime objects, the same runtime object cannot map to two separate OObjects. It would be misleading to have one runtime entity appear as two boxes (two components) on an architectural diagram. Then one could assign the two components different values for a key trustLevel property and potentially invalidate the analysis results.

Aliasing. The static analysis must soundly handle possible aliasing in the program by enforcing the unique representatives invariant. For two variables in the program that may alias and refer to the same runtime object, the analysis must create a single OObject.

Edge soundness. If there is a runtime dataflow communication between two runtime objects, the OGraph must show an OEdge between the representatives of these objects.

Summarization. An ROG can have an unbounded number of runtime objects. For example, in the presence of recursive types, the ROG might have an unbounded depth. The OGraph must be a finite representation of all ROGs and must have a finite depth. The static analysis must stop creating new nodes in the OGraph at some level, and instead use already created nodes. A common heuristic is for the analysis to stop when it gets to a node of the same type as a node it previously created.

Hierarchy. A global OGraph must convey architectural abstraction by object hierarchy and support both high-level and detailed understanding of the runtime structure. It must show architecturally significant OObjects near the top of the hierarchy and OObjects representing data structures further down.

Precision. The analysis must not merge objects excessively. For example, an OGraph that repre-

```java
class Main{
  A a;  C c;
  void main(){
    main = new Main();
    main.run();
  }
  void run(){
    a = new A(c);
    //no dataflow communication
    a.m1(); //method call
  }
}
```

```java
class A{
  B b; C c; D d; E e;
  A(C c){
   //no dataflow communication
   //field initialization
    this.c = c;
  }
  void m1(){
  //method call (a$\xrightarrow{B}$d)
    d.setB(b);
  }
  B m2(){
  //method call (d$\xrightarrow{B}$a)
    return d.getB();
  }
  C m3(){
  //field read (b$\xrightarrow{C}$a)
    return b.c;
  }
  void m4(){
  //field write (a$\xrightarrow{C}$b)
    b.c = c;
  }
  D m5(){
  //method call(a$\xrightarrow{B}$e,a$\xrightarrow{C}$e,e$\xrightarrow{D}$a)
    return e.me(b,c);
  }
}
```

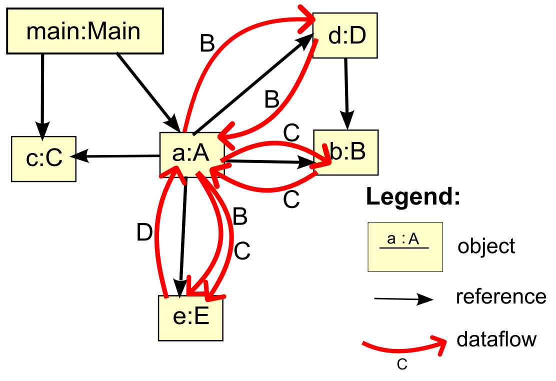

Figure 1: Example of export and import dataflow communication.

sents all the runtime objects with one node is sound but very imprecise. Ideally, the OGraph must have no more OEdges than soundness requires. Like any sound static analysis, however, the OGraph may have false positives and may show OObjects or OEdges that do not correspond to a runtime object or runtime relation, due to infeasible paths in the program.

3 Dataflow Communication

Definition of dataflow communication: *Let* `a` *and* `b` *be two objects. Dataflow communication exists from* `a` *to* `b` *if* `a` *reads or writes to* `b`*'s fields or calls* `b`*'s methods.*

In object-oriented code, a dataflow communication between two references `a` and `b` corresponds to field writes, field reads, or method invocations. Since the flow can be bidirectional, we distinguish between *import* and *export* dataflow.

Import dataflow communication: *An import dataflow communication exists from the source* `b` *of type* `B` *to the destination* `a` *of type* `A` *if* `a` *receives data from* `b`*.* That is, there is a method `ma` of A such that `ma` refers to `b.f` or uses the result returned by a method `mb` of `B`.

Export dataflow communication: *An export dataflow communication exists from the source* `a` *of type* `A` *to the destination* `b` *of type* `B` *if one of* `b`*'s field* `f` *may be modified when one of* `a`*'s methods is invoked.* That is, there is a method `ma` of `a` such that `ma` contains the statement `b.f = c` or `b.mb(c)`, where `c` is in the scope of `ma`, i.e., a field of `A`, an argument of `ma`, an object instantiated by `ma`, or an object returned by another method invoked by `ma`.

To understand the above definitions, consider the example in Figure 1, which has the code for the classes `Main` and `A`, and the corresponding flat object graph. For brevity consider that all the variables are correctly initialized, we do not include the code for the classes `B..E`. In the object graph, the nodes corresponds to objects, and there are two types of edges. Straight arrows means that an object refer another object, while curved arrows correspond to dataflow communications between objects. The curve arrows are labeled with the type of the data communicated between objects.

Dataflow communication exists due to the statements of the methods of `A`, `m1()` to `m5()`. Import dataflow communication exist from `b` to `a` and from `d` to `a` due to the field read and method invocation expressions of `m3()` and `m2()`, respectively. Also, export dataflow communication exist from `a` to `b` and from `a` to `d` due to the field write and method invocation expressions of `m4()` and `m1()`, respectively. Due to only one method invocation expression in `m5()`, two export dataflow communication exist from `a` to `e`, and due to the same expression, an import dataflow communication exists from `e` to `a`.

On the other hand, in the last statement of `run()`, the invocation of `m1()` does not correspond to any import or export dataflow communication, since the method has no arguments, and it returns `void`. Also, there is no dataflow communication from `main` to `a` even though the constructor of `A` has an argument. Dataflow communication definitions ignore object allocation because we consider creation and usage of objects as separate relations, and we distinguish between field initialization in a constructor and field write. That is why, there is no dataflow communication between `main`

$$\frac{\Gamma, \Sigma, \theta \vdash e : T_0 \qquad fields(T_0) = \overline{T}\ \overline{f} \qquad \Gamma, \Sigma, \theta \vdash e' : T \qquad T <: T_i}{\Gamma, \Sigma, \theta \vdash e.f_i = e' : T} \text{[T-WRITE]}$$

$$\frac{S[\ell] = C{<}\overline{p}{>}(\overline{v}) \qquad fields(C{<}\overline{p}{>}) = \overline{T}\ \overline{f} \qquad S' = S[\ell \mapsto C{<}\overline{p}{>}([v/v_i]\overline{v})]}{\ell.f_i = v; S \rightsquigarrow v; S'} \text{[R-WRITE]}$$

$$\frac{\theta \vdash e_0; S \to e_0'; S'}{\theta \vdash e_0.f_i = e_1; S \to e_0'.f_i = e_1; S'} \text{[RC-WRITE-RCV]}$$

$$\frac{\theta \vdash e_1; S \to e_1'; S'}{\theta \vdash v.f_i = e_1; S \to v.f_i = e_1'; S'} \text{[RC-WRITE-ARG]}$$

Figure 2: Field write semantics.

and `a` and between `a` and `c`.

4 Formalization

4.1 Abstract Syntax

We formally describe our static analysis using Featherweight Domain Java (FDJ), which models a core of the Java language with ownership domain annotations [3]. To keep the language simple and easier to reason about, FDJ uses Featherweight Java, which ignores Java language constructs such as interfaces and static code.

We adopt the FDJ abstract syntax (Fig. 3) but with the following changes. We exclude cast expressions and domain links, which are part of FDJ, but not crucial to our discussion. We also include a field write expression $e.f = e'$, which can lead to dataflow communication. (Fig. 2)

In FDJ, C ranges over class names; T ranges over types; f ranges over field names; v ranges over values; d ranges over domain names; e ranges over expressions; x ranges over variable names; n ranges over values and variable names; S ranges over stores; ℓ and θ ranges over locations in a store; θ represents the value of `this`; 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

$$
\begin{array}{rcl}
CT & ::= & \overline{cdef} \\
cdef & ::= & \texttt{class } C{<}\overline{\alpha},\overline{\beta}{>} \texttt{ extends } C'{<}\overline{\alpha}{>} \\
 & & \{\ \overline{dom};\ \ \overline{T}\ \overline{f};\ \ C(\overline{T'}\ \overline{f'},\overline{T}\ \overline{f}) \\
 & & \{\texttt{super}(\overline{f'}); \texttt{this}.\overline{f} = \overline{f};\}\ \ \overline{md}\ \} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; \\
md & ::= & T_R\ m(\overline{T}\ \overline{x})\ \ T_{this}\ \{\texttt{return}\ e_R;\} \\
e & ::= & x\ \mid \texttt{new}\ C{<}\overline{p}{>}(\overline{e})\ \mid e.f\ \mid e.f = e' \\
 & & \mid e.m(\overline{e})\ \mid \ell\ \mid \ell \triangleright e \\
n & ::= & x\ \mid\ v \\
p & ::= & \alpha\ \mid\ n.d\ \mid\ \texttt{SHARED} \\
T & ::= & C{<}\overline{p}{>} \\
v, \ell, \theta & \in & \mathit{locations} \\
S & ::= & \ell\ \to\ C{<}\overline{\ell'.d}{>}(\overline{v}) \\
\Sigma & ::= & \ell\ \to\ T \\
\Gamma & ::= & x\ \to\ T
\end{array}
$$

Figure 3: Simplified FDJ abstract syntax [3].

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; $S[\ell, i]$ denotes the value of i^{th} field of $S[\ell]$; $S[\ell \mapsto C{<}\overline{\ell'.d}{>}(\overline{v})]$ denotes adding an entry for location ℓ to S; α and β range over formal domain parameters; m ranges over method names; p ranges over formal domain parameters, actual domains, or the special domain SHARED; the expression form $\ell \triangleright e$ represents a method body e executing with a receiver ℓ; 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 Σ is the store typing.

4.2 Data Type Declarations

Our analysis produces a hierarchical object graph (OGraph), which has nodes representing objects and domains, and edges representing dataflow communication (Fig. 4). The OGraph is a triplet $G = \langle DO, DD, DE \rangle$, where DO is a set of OObjects, and DD maps a pair $(O, C{::}d)$ to an ODomain D, i.e., DD maintains a mapping from a local domain or a domain parameter d of an OObject O to an actual domain D. Each E in DE is a directed edge from a source O_{src} to a destination O_{dst}, and the label C is the class of the object being communicated. The last label is a flag, and it states whether the OEdge represents an import or an export dataflow communication. Multiple edges with different labels might exists between two OObjects.

$G \in \textsf{OGraph}$	$::= \langle\ \textbf{Objects} = DO,\ \textbf{Domains} = DD, \textbf{Edges} = DE\ \rangle$
$D \in \textsf{ODomain}$	$::= \langle\ \textbf{Id} = D_{id},\ \textbf{Domain} = C{::}d\ \rangle$
$O \in \textsf{OObject}$	$::= \langle\ \textbf{Type} = C\texttt{<}\overline{D}\texttt{>}\ \rangle$
$E \in \textsf{OEdge}$	$::= \langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst}, \textbf{Class} = C, \textbf{Flag} = \mathit{Imp}\ \vert \mathit{Exp}\ \rangle$
DD	$::= \emptyset\ \mid\ DD \cup \{\ (O, C{::}d) \mapsto D\ \}$
DO	$::= \emptyset\ \mid\ DO \cup \{\ O\ \}$
DE	$::= \emptyset\ \mid\ DE \cup \{\ E\ \}$
Υ	$::= \emptyset\ \mid\ \Upsilon \cup \{\ C\texttt{<}\overline{D}\texttt{>}\ \}$
H	$::= \emptyset\ \mid\ H \cup \{\ \ell \mapsto O\ \}$
K	$::= \emptyset\ \mid\ K \cup \{\ \ell.d \mapsto D\ \}$
L_I	$::= \emptyset\ \mid\ L_I \cup \{\ (\ell_{src}, \ell_{dst}) \mapsto \{E\}\}$
L_E	$::= \emptyset\ \mid\ L_E \cup \{\ (\ell_{src}, \ell_{dst}) \mapsto \{E\}\}$

Figure 4: Data type declarations for the OGraph.

Our analysis distinguishes between different instances of the same class C that are in different domains, even if created at the same `new` expression. In addition, the analysis treats an instance of class C with actual parameters $\overline{p}$ differently from another instance that has actual parameters $\overline{p'}$. Hence, the data type of an OObject uses $C\texttt{<}\overline{D}\texttt{>}$ instead of just a type and an owning ODomain. We follow the FDJ convention and consider an OObject's owning ODomain as the first element D_1 of $\overline{D}$. As a result of the aliasing precision provided by ownership domains, our analysis avoids merging objects excessively. It only merges two objects of the same class if all their domains are the same. The context Υ records the combination of class and domain parameters $C\texttt{<}\overline{D}\texttt{>}$ analyzed in the call stack to avoid non-termination of the analysis due to recursive calls.

In addition to the OEdges that have OObjects as source and destination, the OGraph has ownership edges. The OGraph representation is well-formed with respect to the ownership relations declared in the code using the annotations. An ownership edge states that an OObject O is in D_1, or that O owns a domain D. The OGraph captures this hierarchy using the DD map. Given a mapping $\{(O, C'{::}d) \mapsto D\}$ in DD, D is a child of O, i.e., $O \rightarrow D$. Since domains are inherited across classes [3], the class C of O can be a subclass of C' where d is declared. In the presence of recursive types, ownership edges may create cycles.

To invoke the analysis, a developer picks a root class, which is instantiated into a root object. The root class can take only one domain parameter to represent the owning domain. Typically, the root object is in the global ODomain $D_{\texttt{SHARED}}$, the root of the OGraph.

Although a domain d is declared by class C, each instance of C gets its own runtime domain $\ell.d$. For example, if there are two distinct object locations ℓ and ℓ' of class C, then the analysis distinguishes between $\ell.d$ and $\ell'.d$. Since an ODomain represents a runtime domain $\ell_i.d_i$, one domain declaration d in the code can create multiple ODomains D_i in the OGraph. We qualify a domain d by the class that declares it, as C::d. Since no class declares the SHARED domain, we qualify it as ::SHARED.

Instrumentation. The maps H, K, L_I, and L_E are part of the instrumented dynamic semantics (Fig. 4). H maps a location ℓ to the corresponding OObject, and K maps a runtime domain $\ell.d$ to an ODomain. The multi-valued maps L_I and L_E map a pair of locations (ℓ_{src}, ℓ_{dst}) to a set of OEdges $\{E\}$. We use two maps for edges because a pair ($H[\ell_1]$, $H[\ell_2]$) can be associated with an import edge from $H[\ell_1]$ to $H[\ell_2]$, or with an export edge from $H[\ell_1]$ to $H[\ell_2]$.

Notation. For a map M, a key k, and a value v, we use $M[k]$ to denote the lookup of k, and $M' = M[k \mapsto v]$ for adding an entry for k to M. For a multi-valued map M, we use the notation $M' = M[k \mapsto_{\cup} \{v\}]$ for adding an entry for k to M. If the map already has an entry for k, the resulting value is the union of the existing value set and $\{v\}$.

Static Semantics. We formalize our static analysis using a constraint-based specification, as a set of inference rules, then prove that the OGraph is sound, i.e., it has all the required OObjects, ODomains, and OEdges.

In this context, soundness means that we can build a map between a ROG and an OGraph. Soundness consists of object soundness and edge soundness. With object soundness, every runtime object maps to a unique representative OObject in the OGraph. With object soundness, every runtime edge maps to a unique representative OEdge in the OGraph. To build the maps, we instrument the FDJ dynamic semantics. We map every newly created runtime object to an OObject. Also, for every field read, field write, or a method invocation, we map the corresponding runtime edge to an OEdge.

In FDJ, a program is a tuple (CT, e) that consists of a class table CT, which maps classes to their definitions, and an expression e. Our analysis starts with a root expression e_{root}, that explicitly instantiates the root class C_{root}. The analysis result is the least solution $G = \langle DO, DD, DE \rangle$ of the following constraint system:

$$\emptyset, \emptyset, DO, DD, DE \vdash (CT, e_{root})$$

The analysis creates the OObject O_{root} and its owning ODomain $D_{\texttt{SHARED}}$,

$$D_{\texttt{SHARED}} = \langle D_s, ::\texttt{SHARED} \rangle \qquad\qquad O_{root} = \langle O_r, C_{root}{<}D_{\texttt{SHARED}}{>} \rangle$$

then abstractly interprets e_{root} in the context of O_{root}:

$$\emptyset, \emptyset, DO, DD, DE \vdash_{O_{root}} e_{root}$$

The judgement form for expressions is as follows:

$$\Gamma, \Upsilon, DO, DD, DE \vdash_{O,H} e$$

The O subscript on the turnstile captures the context-sensitivity, and represents the context object that the analysis uses to abstractly interpret e. The H subscript is a map used by the dynamic semantics and the store typing rule in the static semantics (not shown). For readability, we omit H when not in use. $CT(C)$ and $CT(\texttt{Object})$ represent a lookup of a class C and the class `Object` in the class table, and is an implicit clause in all the static rules. (We list these clauses once at the top of Fig. 5 to avoid repetition.)

In DF-NEW, the analysis interprets a `new` object allocation in the context of O. The analysis first ensures that DO contains an OObject O_C for the newly allocated object. Then, DF-NEW ensures that DD has a representative ODomain D_i for each domain parameter p_i passed to the constructor of the class C. Based on the binding of each formal domain parameter α_i to actual p_i, DD maps each α_i to a corresponding D_i in the context of O_C $((O_C, \alpha_i) \mapsto D_i)$ (Fig. 5).

$$CT(C) = \texttt{class}\ C<\overline{\alpha},\overline{\beta}>\ \texttt{extends}\ C'<\overline{\alpha}>\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}$$

$$CT(\texttt{Object}) = \texttt{class}\ \texttt{Object}<\alpha_o>\ \{\ \}$$

$$\frac{\begin{array}{c} \forall i \in 1..|\overline{p}| \qquad D_i = DD[(O,p_i)] \qquad params(C)=\overline{\alpha} \\ O_C = \langle\ C<\overline{D}>\ \rangle \qquad \{O_C\}\subseteq DO \qquad \alpha_i\in\overline{\alpha} \\ \{(O_C,\alpha_i)\mapsto D_i\}\subseteq DD \qquad \{(O_C,p_i)\mapsto D_i\}\subseteq DD \\ DO,DD,DE \vdash_O ddomains(C,O_C) \\ \forall m \in \overline{md}\ mbody(m, C<\overline{p}>) = (\overline{x}:\overline{T},\ e_R) \\ C<\overline{D}> \notin \Upsilon \implies \{\overline{x}:\overline{T},\ \texttt{this}: C<\overline{p}>\}, \Upsilon\cup\{C<\overline{D}>\}, DO,DD,DE \vdash_{O_C} e_R \\ \Gamma,\Upsilon,DO,DD,DE\vdash_O \overline{e}\end{array}}{\Gamma,\Upsilon,DO,DD,DE\vdash_O \texttt{new}\ C<\overline{p}>(\overline{e})}\ [\textsc{Df-New}]$$

$$\frac{\begin{array}{c} \forall(\texttt{domain}\ d_j)\in\overline{dom} \qquad D_j = \langle D_{id_j},\ C::d_j\rangle \qquad \{(O_C, C::d_j)\mapsto D_j\}\subseteq DD \\ DO,DD,DE\vdash_O ddomains(C',O_C)\end{array}}{DO,DD,DE\vdash_O ddomains(C,O_C)}\ [\textsc{Aux-Dom}]$$

$$\frac{}{DO,DD,DE\vdash_O ddomains(\texttt{Object},O_C)}\ [\textsc{Aux-Obj1}] \qquad \frac{\begin{array}{c} O_k\in DO \qquad O_k = \langle C<\overline{D}>\ \rangle \qquad C <: C' \\ \forall i\in 1..|\overline{p'}| \qquad D'_i = DD[(O,p'_i)] \qquad D'_i = D_i\end{array}}{DO,DD,DE\vdash_O lookup\ (C'<\overline{p'}>) = \{O_k\}_{k\in 1..sz}}\ [\textsc{Df-Lookup}]$$

$$\frac{\begin{array}{c} e_0 : C<\overline{p}> \qquad (T_k\ f_k)\in fields(C<\overline{p}>) \\ DO,DD,DE\vdash_O import(C<\overline{p}>,T_k) \\ \Gamma,\Upsilon,DO,DD,DE\vdash_O e_0\end{array}}{\Gamma,\Upsilon,DO,DD,DE\vdash_O e_0.f_k}\ [\textsc{Df-Read}] \qquad \frac{\begin{array}{c} e_0 : C<\overline{p}> \qquad (T_k\ f_k)\in fields(C<\overline{p}>) \\ e_1 : C_1<\overline{p''}> \qquad C_1<\overline{p''}> <: T_k \\ DO,DD,DE\vdash_O export(C<\overline{p}>,C_1<\overline{p''}>) \\ \Gamma,\Upsilon,DO,DD,DE\vdash_O e_0 \qquad \Gamma,\Upsilon,DO,DD,DE\vdash_O e_1\end{array}}{\Gamma,\Upsilon,DO,DD,DE\vdash_O e_0.f_k = e_1}\ [\textsc{Df-Write}]$$

$$\frac{\begin{array}{c} DO,DD,DE\vdash_O lookup\ (T_{src}) = \{O_i\}_{i\in 1..sz} \\ DO,DD,DE\vdash_{O_i} lookup\ (T_{label}) = \{O_j\}_{j\in 1..sz'} \\ \forall i\in 1..sz\ \forall j\in 1..sz'\ O_j = \langle C_j<\overline{D}>\rangle\ \{\langle O_i,O,C_j,Imp\rangle\}\subseteq DE\end{array}}{DO,DD,DE\vdash_O import\ (T_{src},T_{label})}\ [\textsc{Aux-Import}]$$

$$\frac{\begin{array}{c} DO,DD,DE\vdash_O lookup\ (T_{dst}) = \{O_i\}_{i\in 1..sz} \\ DO,DD,DE\vdash_O lookup\ (T_{label}) = \{O_j\}_{j\in 1..sz'} \\ \forall i\in 1..sz\ \forall j\in 1..sz'\ O_j = \langle C_j<\overline{D}>\rangle\ \{\langle O,O_i,C_j,Exp\rangle\}\subseteq DE\end{array}}{DO,DD,DE\vdash_O export\ (T_{dst},T_{label})}\ [\textsc{Aux-Export}]$$

$$\frac{\begin{array}{c} e_0 : C<\overline{p}> \qquad mtype(m,C<\overline{p}>) = \overline{T}\rightarrow T_R \\ DO,DD,DE\vdash_O import(C<\overline{p}>,T_R) \\ \forall k\in 1..|\overline{e}|\ e_k : T'_k \qquad T'_k <: T_k \qquad T_k\in\overline{T} \qquad DO,DD,DE\vdash_O export(C<\overline{p}>,T'_k) \\ \Gamma,\Upsilon,DO,DD,DE\vdash_O e_0 \qquad \Gamma,\Upsilon,DO,DD,DE\vdash_O \overline{e}\end{array}}{\Gamma,\Upsilon,DO,DD,DE\vdash_O e_0.m(\overline{e})}\ [\textsc{Df-Invk}]$$

Figure 5: Static semantics. Additional rules (Df-Var, Df-Loc, Df-Context, Df-Sigma) are in [20].

Then, Df-New uses the auxiliary judgement Aux-Dom to ensure that DD has an ODomain corresponding to each domain that C locally declares ($(O_C, C::d_j) \mapsto D_j$). Aux-Dom recursively includes inherited domains from base classes as well. Aux-Obj1, the base case of the recursion, deals with the class `Object`, for which Aux-Obj1 does nothing, because `Object` has no fields, domains, or methods in FDJ.

Df-New then obtains each expression e_R in each method m of C, and recursively processes e_R in the context of the new OObject O_C. To avoid infinite recursion, before Df-New analyzes e_R, it checks if the combination of the class C and actual domains $\overline{D}$ have been previously analyzed by looking for this combination in Υ. If this combination does not exist, Df-New extends Υ with the current combination. As a side note, Υ tracks previously analyzed OObjects only at the call stack level. It does not do so globally across the program because similar combinations of the same class and domain parameters can occur in different contexts, and must be analyzed separately. Finally, Df-New analyzes each argument of the constructor. Since our analysis distinguishes between a field initialization in a constructor and a field write, Df-New does not require dataflow edges in DE.

Df-Lookup defines the auxiliary judgement *lookup* that returns the set of the OObjects O_k in DO such that the class of O_k is C' or one of its subclasses. It also ensures that each domain D_i of O_k corresponds to D'_i, a domain associated with O in DD. The second condition increases the precision of our analysis, because *lookup* returns only a subset of all the objects of class C' or its subclasses in DO. From this subset, our analysis picks the source or destination OObjects, and finds the class representing the label of an OEdge.

The auxiliary judgements Aux-Import and Aux-Export ensure import and export edges between the context OObject O and the OObjects O_i, where O_i is the result of *lookup* (T_{src}), and *lookup* (T_{dst}), respectively. The direction of the edge is from O_i to the context O for Aux-Import, and from the context O to O_i for Aux-Export. To identify an edge's label, Aux-Export calls *lookup* in the context of O, while Aux-Import calls the second *lookup* in the context of O_i. As a result, there could be multiple edges with different labels between the same two OObjects, depending on what *lookup* returns.

Df-Read and Df-Write abstractly interpret field read and field write expressions, respectively, and use Aux-Import and Aux-Export. Both auxiliary judgements take the type e_0 as

$$\frac{}{\Gamma, \Upsilon, DO, DD, DE \vdash_O x}\text{[Df-Var]} \qquad \frac{}{\Gamma, \Upsilon, DO, DD, DE \vdash_O \ell}\text{[Df-Loc]}$$

$$\frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, DO, DD, DE \vdash_{O_C} e}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O,H} \ell \triangleright e}\text{[Df-Context]}$$

$$\frac{\begin{array}{c}\forall \ell \in dom(S), \Sigma[\ell] = C\texttt{<}\overline{p}\texttt{>} \qquad H[\ell] = O = \langle C\texttt{<}\overline{D}\texttt{>}\rangle \in DO \\ \forall m.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \emptyset, DO, DD, DE \vdash_O e_R\end{array}}{DO, DD, DE \vdash_{CT,H} \Sigma}\text{[Df-Sigma]}$$

Figure 6: Static semantics (continued).

the first argument, and pass it to *lookup* to set the source and destination OObjects. For the label, Df-Read uses the type of the field f_k, while Df-Write uses the type of the right-hand side expression e_1. The labels are the classes of these types or one of their subclasses.

Df-Invk abstractly interprets method invocation expressions. First, it ensures the existence of an import edge from the receiver of the method to the context OObject O. The label of the import edge is the class of the return type, or one of its subclasses. Next, for each argument e_k, Df-Invk ensures the existence of an export edge from O to the receiver of the method. The label of each export edge is the class of the argument or one of its subclasses. The rule ensures export edges only for a method invocation with at least one argument. Finally, the rule evaluates recursively the expressions e_0 and $\overline{e}$.

Df-Var, and Df-Loc, and the rest of the rules complete our formalization and make the induction go through (Fig. 6). Df-Context analyzes expressions of the form $\ell \triangleright e$. The context for analyzing e changes from O to O_C, where O_C is the result of looking up the receiver ℓ in H. Finally, the induction requires an augmented store typing rule, Df-Sigma, to ensures that the method bodies have been analyzed for all the locations ℓ in the store, and that every ℓ has a corresponding OObject in DO. To denote all the objects in the store, we use the CT subscript instead of O.

Dynamic Semantics. To complete the formalization, we instrumented the dynamic semantics (Fig. 7). The instrumentation extends the dynamic semantics of FDJ [3] (the common parts are highlighted), but is safe since discarding it produces exactly the FDJ dynamic semantics. The

$$\dfrac{\begin{array}{c}\boxed{\ell \notin dom(S) \qquad S' = S[\ell \mapsto C\texttt{<}\overline{p}\texttt{>}(\overline{v})]} \\ G = \langle DO, DD, DE \rangle \\ \overline{p} = \overline{\ell'.d} \qquad \forall i \in 1..|\overline{\ell'.d}| \; D_i = K[\ell'_i.d_i] \\ O_C = \langle C\texttt{<}\overline{D}\texttt{>}\rangle \qquad O_C \in DO \qquad H' = H[\ell \mapsto O_C] \\ \forall(\texttt{domain}\; d_j) \in domains(C\texttt{<}\overline{p}\texttt{>}) \qquad D_j = DD[(O_C, C{::}d_j)] \qquad K' = K[\ell.d_j \mapsto D_j]\end{array}}{\theta \vdash \boxed{\texttt{new}\; C\texttt{<}\overline{p}\texttt{>}(\overline{v}); S}\,; H; K; L_I; L_E \rightsquigarrow_G \boxed{\ell; S'}\,; H'; K'; L_I; L_E}\text{[IR-NEW]}$$

$$\dfrac{\begin{array}{c}\boxed{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{p}\texttt{>}) = \overline{T}\;\overline{f}} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad T_i = C_i\texttt{<}\overline{p'}\texttt{>} \qquad T_i \in \overline{T} \\ E = \langle O_\ell, O, C_v, Imp \rangle \in DE \qquad C_v <: C_i \qquad L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E\}]\end{array}}{\theta \vdash \boxed{\ell.f_i; S}\,; H; K; L_I; L_E \rightsquigarrow_G \boxed{v_i; S}\,; H; K; L'_I; L_E}\text{[IR-READ]}$$

$$\dfrac{\begin{array}{c}\boxed{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{p}\texttt{>}) = \overline{T}\;\overline{f}} \\ \boxed{S' = S[\ell \mapsto C\texttt{<}\overline{p}\texttt{>}([v/v_i]\overline{v})]} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad T_i = C_i\texttt{<}\overline{p'}\texttt{>} \qquad T_i \in \overline{T} \\ E = \langle O, O_\ell, C_v, Exp \rangle \in DE \qquad C_v <: C_i \qquad L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E\}]\end{array}}{\theta \vdash \boxed{\ell.f_i = v; S}\,; H; K; L_I; L_E \rightsquigarrow_G \boxed{v; S'}\,; H; K; L_I; L'_E}\text{[IR-WRITE]}$$

$$\dfrac{\begin{array}{c}\boxed{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x}, e_R)} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T} \to T_R \qquad T_R = C_R\texttt{<}\overline{p'}\texttt{>} \\ E' = \langle O_\ell, O, C'_R, Imp \rangle \in DE \qquad C'_R <: C_R \qquad L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}] \\ \forall k \in 1..|\overline{T}| \; T_k = C_k\texttt{<}\overline{p''}\texttt{>} \qquad E_k = \langle O, O_\ell, C'_k, Exp \rangle \in DE \qquad C'_k <: C_k \\ L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E_k\}]\end{array}}{\theta \vdash \boxed{\ell.m(\overline{v}); S}\,; H; K; L_I; L_E \rightsquigarrow_G \boxed{\ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S}\,; H; K; L'_I; L'_E}\text{[IR-INVK]}$$

$$\dfrac{}{\theta \vdash \boxed{\ell \triangleright v; S}\,; H; K; L_I; L_E \rightsquigarrow_G \boxed{v;\; S}\,; H; K; L_I; L_E}\text{[IR-CONTEXT]}$$

Figure 7: Instrumented dynamic semantics (core rules).

instrumented evaluation rule is of the following form:

$$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$$

where $G = \langle DO, DD, DE \rangle$ is the statically computed object graph, and $\rightsquigarrow_G$ means that the expression e evaluates to e' in the context of θ, the value of `this`. The dynamic semantics keep G unchanged, but change the store S and the maps H, K, L_I, and L_E.

IR-NEW adds a new location ℓ to the store S, where ℓ maps to an object of type C with the specified ownership domain parameters, and the fields set to the values $\overline{v}$ passed to the constructor. The rule extends H by mapping ℓ and the OObject O_C from DO. The rule requires that each actual domains p_i passed during instantiation corresponds to an actual domain D_i of O_C. Next, the rule extends K such that for all the domains C::d_j, the pair (O_C, C::d_j) has a corresponding D_j in DD.

IR-Read and IR-Write ensure that an OEdge E exists between the context OObject O and the receiver O_ℓ. They use θ and ℓ to lookup these OObjects in H. They also ensure that the edge label C_v is a subclass of the field class C_i. Finally, the rules extend the maps L_I and L_E, respectively, by adding E to the set of edges associated with (ℓ, θ) in L_I, and (θ, ℓ) in L_E.

IR-Invk ensures that an import OEdge E' exists from the receiver O_ℓ to the context O, having as the edge's label a subclass of the return class C_R. IR-Invk also ensures that an export OEdge E_k exist from O to O_ℓ for every parameter, having as edge label a subclass of the method's parameter class C_k. The rule uses θ and ℓ to lookup O and O_ℓ in H . It extends both L_I and L_E by adding E' to the set of import edges between the locations ℓ and θ in L_I, and by adding each E_k to the set of export edges between the locations θ and ℓ in L_E.

When the method expression reduces to a value v, IR-Context propagates v outside of its method context. This rule does not affect the execution of the program.

Finally, the dynamic semantics include standard congruence rules. The congruence rules are similar to those in FDJ [3] (Fig. 8). In addition, there are two congruence rules for field-write: IRC-Write-Rcv and IRC-Write-Arg. IRC-Write-Rcv states that the receiver expression e_0 reduces to e'_0, while IRC-Write-Arg states that the right-hand side expression e_1 reduces to e'_1.

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-New}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash e_0.f_i; S; H; K; L_I; L_E \rightsquigarrow_G \\ e_0'.f_i; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Read}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash e_0.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ e_0'.f_i = e_1; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Write-Rcv}]$$

$$\frac{\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e_1'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash v.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G \\ v.f_i = e_1'; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Write-Arg}]$$

$$\frac{\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash e_0.m(\overline{e}); S; H; K; L_I; L_E \rightsquigarrow_G \\ e_0'.m(\overline{e}); S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-RecvInvk}]$$

$$\frac{\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e_i'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G \\ v.m(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-ArgInvk}]$$

$$\frac{\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L_I'; L_E'}{\begin{array}{c}\theta \vdash \ell \triangleright e; S; H; K; L_I; L_E \rightsquigarrow_G \\ \ell \triangleright e'; S'; H'; K'; L_I'; L_E'\end{array}}[\text{IRC-Context}]$$

Figure 8: Instrumented dynamic semantics (congruence rules).

Recursive Types. The analysis must handle recursive types which lead to unbound number of nodes in the OGraph. As an example, consider a class `QuadTree`, which declares a field `nwQT` of type `QuadTree` in its `OWNED` private domain (Fig. 9). To get a finite OGraph and ensure the analysis terminates, the analysis could stop expanding an OGraph after a certain depth. However, truncating the recursion at an arbitrary depth may fail to show when a child object beyond the visible depth communicates to external objects. Instead, the analysis creates a cycle in the OGraph when it reaches a similar context. There are two possible choices: to unify objects or to unify domains.

```
Main<SHARED> main = new Main<SHARED>();
class Main<OWNER> {
  domain OWNED;
  QuadTree<this.OWNED> aQT;
  ctor() {
    return new QuadTree<this.OWNED>();
  }
}
class QuadTree<M> {
  domain OWNED;
  QuadTree<OWNED> nwQT;
  ctor() {
    return new QuadTree<this.OWNED>();
  }
}
```

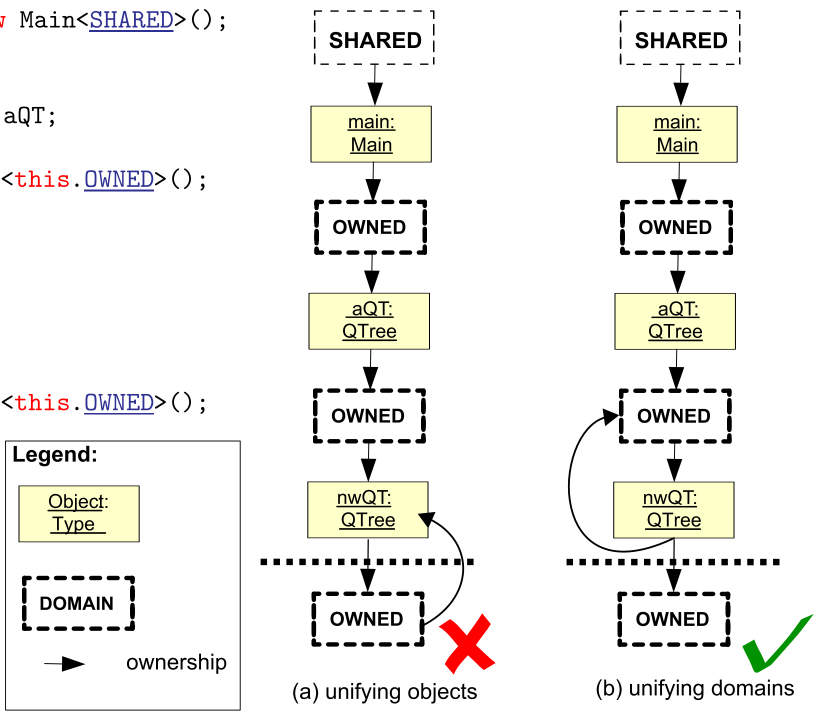

Figure 9: Handling recursive types, revised from [1, Figure 2.22].

The analysis creates objects until it detects that it is creating objects similar to the one it created before. In this case, the analysis uses an existing similar object. One can imagine multiple notion of similarity; it can be any equivalence relation as long as the number of dissimilar objects is finite. We adopt the following similarity relation between two objects a and b: a and b are of the same type, including actual domain parameters ($C{<}\overline{D}{>}$). Unifying objects is problematic, because

O_{root} dummy receiver
{(O_{root}, SHARED)$\mapsto$ SHARED} $\subseteq$ DD
Υ = { }

//Analyzing (line 1) in the context of O_{root}:

$$\frac{}{\emptyset; \emptyset; DO, DD, DE \vdash_{O_{root}} \texttt{new Main} < \texttt{SHARED} >)()}$$

```
CT(Main) = class Main<OWNER> {
  domain OWNED;
  QuadTree<this.OWNED> aQT;
  ctor() { return new QuadTree<this.OWNED>(); }
}
```

In Df-New:
O == O_{root}
this == θ_0
DO == [O_{root}]
DD == [(O_{root}, SHARED)$\mapsto$ SHARED;]
DE == []
C == Main
i == 1
α_i == Main::OWNER
p_i == ::SHARED
D_i == DD[(O_{root}, ::SHARED)] == SHARED
O_C == Main<SHARED>
{ Main<SHARED> } $\subseteq$ DO
{ (Main<SHARED>, Main::OWNER)$\mapsto$ SHARED,
(Main<SHARED>, ::SHARED)$\mapsto$ SHARED } $\subseteq$ DD

// In Aux-Dom:
DD == [(O_{root}, ::SHARED)$\mapsto$ SHARED;
(Main<SHARED>, Main::OWNER)$\mapsto$ SHARED;
(Main<SHARED>, ::SHARED)$\mapsto$ SHARED;]
$\overline{dom}$ == [domain OWNED]
j == 1
C::D_j == Main::OWNED
O_C == Main<SHARED>
D_j == ODomain(main.OWNED, Main::OWNED)
{ (Main<SHARED>, Main::OWNED) $\mapsto$ main.OWNED } $\subseteq$ DD

Figure 10: Applying Df-New on the QuadTree example. First pass: the analysis creates the OObject for main (line 1) and the ODomain main.OWNED.

for two objects to be similar, it is necessary to detect they have the same owning ODomain. But, if the ODomain has a unique owning OObject, the problem is circular. Moreover, in order to add edges, we lookup objects in a given domain by their type.

Since recognizing domains is important, we adopt the solution of unifying domains. It is simpler

```
//Back in Df-New, analyze recursively constructor body as if it were a method:
ϒ = { Main<SHARED> }
// In context O, where O = O_C above, i.e., Main<SHARED>, analyze:
```

$$\frac{}{\Gamma; \Upsilon; DO; DD; DE \vdash_O \texttt{new QuadTree<this.OWNED>()}}$$

```
CT(QuadTree) = class QuadTree<M> {
  domain OWNED;
  QuadTree<OWNED> nwQT;
  ctor() {
    return new QuadTree<this.OWNED>();
  }
}
// In Df-New:
O == Main<SHARED>
this == main
C == QuadTree
DO == [O_root; Main<SHARED>]
DD == [(O_root, ::SHARED)↦ SHARED;
(Main<SHARED>, Main::OWNER) ↦ SHARED;
(Main<SHARED>, ::SHARED )↦ SHARED;
(Main<SHARED>, Main::OWNED) ↦ main.OWNED; ]
i == 1
p_i == Main::OWNED
α_i == QuadTree::M
D_i == DD[(Main<SHARED>, this.OWNED)] = main.OWNED
O_C == QuadTree<main.OWNED>
{ QuadTree<main.OWNED>} ⊆ DO
{ (QuadTree<main.OWNED>, QuadTree::M)↦ main.OWNED,
(QuadTree<main.OWNED>, Main::OWNED)↦ main.OWNED
} ⊆ DD
// In Aux-Dom:
DD == [(O_root, ::SHARED)↦ SHARED;
(Main<SHARED>, Main::OWNER) ↦ SHARED;
(Main<SHARED>, ::SHARED )↦ SHARED;
(Main<SHARED>, Main::OWNED) ↦ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::M)↦ main.OWNED;
(QuadTree<main.OWNED>, Main::OWNED)↦ main.OWNED; ]
dom‾ == [domain OWNED]
j == 1
C::D_j == QuadTree::OWNED
O_C == QuadTree<main.OWNED>
D_j == ODomain(main.OWNED.aQT.OWNED, QuadTree::OWNED)
{(QuadTree<main.OWNED>, QuadTree::OWNED)↦ main.OWNED.aQT.OWNED }⊆ DD
```

Figure 11: Applying DF-NEW on the QuadTree example. Second pass: the analysis creates the OObject for aQT (line 6) and the ODomain main.OWNED.aQT.OWNED.

```
//Back in Df-New, analyze recursively constructor body as if it were a method:
ϒ = { Main<SHARED>, QuadTree<main.OWNED> }
// In context O, where O = O_C above, i.e., QuadTree<main.OWNED>, analyze:
                    Γ; ϒ; DO; DD; DE ⊢_O new QuadTree<this.OWNED>()
CT(QuadTree) = class QuadTree<M> {
  domain OWNED;
  QuadTree<OWNED> nwQT;
  ctor() { return new QuadTree<this.OWNED>(); }
}
// In Df-New:
O == QuadTree<main.OWNED>
this == aQT
DO == [O_root; Main<SHARED>; QuadTree<main.OWNED>]
DD == [(O_root, ::SHARED)↦ SHARED;
(Main<SHARED>, Main::OWNER) ↦ SHARED;
(Main<SHARED>, ::SHARED )↦ SHARED;
(Main<SHARED>, Main::OWNED) ↦ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::M)↦ main.OWNED;
(QuadTree<main.OWNED>, Main::OWNED)↦ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::OWNED)↦ main.OWNED.aQT.OWNED; ]
C == QuadTree
i == 1
p_i == QuadTree::OWNED
α_i == QuadTree::M
D_i == DD[ (QuadTree<main.OWNED>, QuadTree::OWNED)]= main.OWNED.aQT.OWNED
O_C == QuadTree<main.OWNED.aQT.OWNED>
{ QuadTree<main.OWNED.aQT.OWNED> } ⊆ DO
{ (QuadTree<main.OWNED.aQT.OWNED>, QuadTree::M)↦ main.OWNED.aQt.OWNED,
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)↦ main.OWNED.aQt.OWNED } ⊆ DD
// In Aux-Dom:
DD == [(O_root, ::SHARED)↦ SHARED;
(Main<SHARED>, Main::OWNER) ↦ SHARED;
(Main<SHARED>, ::SHARED )↦ SHARED;
(Main<SHARED>, Main::OWNED) ↦ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::M)↦ main.OWNED;
(QuadTree<main.OWNED>, Main::OWNED)↦ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::OWNED)↦ main.OWNED.aQT.OWNED;
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::M)↦ main.OWNED.aQt.OWNED;
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)↦ main.OWNED.aQt.OWNED; ]
dom == [domain OWNED]
j == 1
C::D_j == QuadTree::OWNED
O_C == QuadTree<main.OWNED.aQT.OWNED>
D_j == ODomain(main.OWNED.aQT.OWNED, QuadTree::OWNED) //reuse
{ (QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED) ↦ main.OWNED.aQt.OWNED } ⊆ DD
```

Figure 12: Applying DF-NEW on the QuadTree example. Third pass: the analysis creates the OObject for `nwQT` (line 13), and reuses the ODomain `main.OWNED.aQT.OWNED`

```
//Back in Df-New, analyze recursively constructor body as if it were a method:
$\Upsilon$ = { Main<SHARED>, QuadTree<main.OWNED>, QuadTree<main.OWNED.aQT.OWNED> }
// In context $O$, where $O$ = $O_C$ above, i.e., QuadTree<main.OWNED.aQT.OWNED>, analyze:
                    $\Gamma; \Upsilon; DO; DD; DE \vdash_O$ new QuadTree<this.OWNED>()
// In Df-New:
$O$ == QuadTree<main.OWNED.aQT.OWNED>
this == nwQT
$DO$ == [$O_{root}$; Main<SHARED>; QuadTree<main.OWNED>; QuadTree<main.OWNED.aQT.OWNED>]
$DD$ == [($O_{root}$, ::SHARED)$\mapsto$ SHARED;
(Main<SHARED>, Main::OWNER) $\mapsto$ SHARED;
(Main<SHARED>, ::SHARED )$\mapsto$ SHARED;
(Main<SHARED>, Main::OWNED) $\mapsto$ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::M)$\mapsto$ main.OWNED;
(QuadTree<main.OWNED>, Main::OWNED)$\mapsto$ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::OWNED)$\mapsto$ main.OWNED.aQT.OWNED;
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::M)$\mapsto$ main.OWNED.aQt.OWNED;
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)$\mapsto$ main.OWNED.aQt.OWNED; ]
$C$ == QuadTree
i == 1
$p_i$ == QuadTree::OWNED
$\alpha_i$ == QuadTree::M
$D_i$ == $DD$[ (QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)]= main.OWNED.aQT.OWNED
$O_C$ == QuadTree<main.OWNED.aQT.OWNED> // Reuse
{ QuadTree<main.OWNED.aQT.OWNED > } $\subseteq$ $DO$
{ (QuadTree<main.OWNED.aQT.OWNED >, QuadTree::M)$\mapsto$ main.OWNED.aQT.OWNED,
(QuadTree<main.OWNED.aQT.OWNED >, QuadTree::OWNED)$\mapsto$ main.OWNED.aQT.OWNED} $\subseteq$ $DD$
// In Aux-Dom:
$DD$ == [($O_{root}$, ::SHARED)$\mapsto$ SHARED;
(Main<SHARED>, Main::OWNER) $\mapsto$ SHARED;
(Main<SHARED>, ::SHARED )$\mapsto$ SHARED;
(Main<SHARED>, Main::OWNED) $\mapsto$ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::M)$\mapsto$ main.OWNED;
(QuadTree<main.OWNED>, Main::OWNED)$\mapsto$ main.OWNED;
(QuadTree<main.OWNED>, QuadTree::OWNED)$\mapsto$ main.OWNED.aQT.OWNED;
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::M)$\mapsto$ main.OWNED.aQt.OWNED;
(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)$\mapsto$ main.OWNED.aQt.OWNED; ]
$\overline{dom}$ == [domain OWNED]
j == 1
$C$::$D_j$ == QuadTree::OWNED
$D_j$ == ODomain(main.OWNED.aQT.OWNED.nwQT.OWNED, QuadTree::OWNED)
{ (QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED) $\mapsto$ QuadTree<main.OWNED.aQT.OWNED>} $\subseteq$ $DD$
//Back in Df-New:
$\Upsilon$ = { Main<SHARED>, QuadTree<main.OWNED>, QuadTree<main.OWNED.aQT.OWNED> }
QuadTree<main.OWNED.aQT.OWNED> $\in$ $\Upsilon$ //STOP.
```

Figure 13: Applying Df-New on the QuadTree example. Forth pass: the analysis terminates when `QuadTree<main.OWNED.aQT.OWNED>` is found in Υ

to recognize that two ODomains have the same underlying domain declaration C::d, than to recognize similar objects. The analysis creates a cycle in the OGraph when the same ODomain appears as

```
Main<SHARED> main = new Main<SHARED>();
OObject(main, SHARED, Main)
analyze(main, [Main::OWNER↦SHARED])
this↦main
Main::OWNER↦SHARED
class Main<OWNER> {
  domain OWNED;
  ODomain(main.OWNED, Main::OWNED)
  OObject(main.OWNED.aQT, main.OWNED, QuadTree)
  QuadTree<OWNED> aQT;
  ctor() {
    return = new QuadTree<OWNED>();
  analyze(main.OWNED.aQT, [QuadTree::M↦Main::OWNED])
  }
}
this↦main.OWNED.aQT
[QuadTree::M↦Main::OWNED]
class QuadTree<M> {
  domain OWNED;
  ODomain(main.OWNED.aQT.OWNED, QuadTree::OWNED)
  OObject(main.OWNED.aQT.OWNED.nwQT, main.OWNED.aQT.OWNED, QuadTree)
  QuadTree<OWNED> nwQT;
  ctor() {
    return = new QuadTree<OWNED>();
  analyze(main.OWNED.aQT.OWNED.nwQT, [QuadTree::M↦QuadTree::OWNED])
  }
}
this ↦ main.OWNED.aQT.OWNED.nwQT
[QuadTree::M ↦ QuadTree::OWNED]
class QuadTree<M> {
  domain OWNED;
  ODomain(main.OWNED.aQT.OWNED, QuadTree::OWNED)
  OObject(main.OWNED.aQT.OWNED.nwQT, main.OWNED.aQT.OWNED, QuadTree)
  QuadTree<OWNED> nwQT;
  ctor() {
    return = new QuadTree<OWNED>();
  analyze(main.OWNED.aQT.OWNED.nwQT, [QuadTree::M ↦ QuadTree::OWNED])
  }
}
```

Figure 14: Worked example with recursive types, revised from [1, Figure 2.24].

the child of two OObjects. This justifies an ODomain not having a unique owning OObject (Fig. 4).

In the QuadTree example, the analysis does four passes over DF-NEW and AUX-DOM. In the first two passes, the analysis creates two new OObject instances while interpreting the `new` expressions (Fig 9 line 1 and line 6) and two new ODomain instances while interpreting the domain declarations (Fig 9 line 3 and line 10). The details of each pass are in Fig. 10 and Fig. 11. In each pass, the context O changes to O_C while analyzing the body of `ctor`.

In the third pass, the analysis creates a new OObject `QuadTree<main.OWNED.aQT.OWNED`, but while interpreting the domain declaration `QuadTree::OWNED`, the analysis reuses the existing ODomain `main.OWNED.aQT.OWNED` created during the second pass. Two constraints are ensuring this reuse. First, in the constraint $(O_C, p_i) \mapsto D_i \subseteq DD$ of Df-New O_C is `QuadTree<main.OWNED.aQT.OWNED>` and p_i is the domain `QuadTree::OWNED`. Consequently, the analysis ensures that the pair `(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)` maps to `main.OWNED.aQT.OWNED` in DD. Second, while ensuring the constraint $(O_C, C :: d_j) \mapsto D_j \subseteq DD$ of Aux-Dom, the analysis encounters again the pair `(QuadTree<main.OWNED.aQT.OWNED>, QuadTree::OWNED)`. This justifies why, the analysis reuses the `main.OWNED.aQT.OWNED` ODomain instead of creating a new one (Fig 12). By detecting the same ODomain, the analysis can make the OObject `QuadTree<main.OWNED.aQT.OWNED>` to be both the parent and the child of an ODomain, thus creating a cycle.

In the forth pass, the analysis reuses the existing OObject `QuadTree<main.OWNED.aQT.OWNED>` as O_C to ensure that $\{O_C\} \in DO$ and, similarly to pass 3, the analysis reuses the existing ODomain `main.OWNED.aQT.OWNED`. Next, since $O_C == O$ and the condition $C\langle\overline{D}\rangle \notin \Upsilon$ is unsatisfied, the analysis terminates (Fig. 13).

4.3 Soundness

An OGraph is a *sound* approximation of a ROG, represented by a well-typed store S, if the OGraph relates to the ROG as follows:

Object soundness. There is a map H that maps each object ℓ in S to exactly one representative OObject in the OGraph. Similarly, there is a map K such that each runtime domain $\ell.d$ has exactly one representative ODomain in the OGraph.

Edge soundness. If there is a dataflow communication from an object ℓ_1 to ℓ_2 in a ROG, with their representatives OObjects O_1 and O_2 in the OGraph, then there are two maps L_I and L_E that map the pair (ℓ_1, ℓ_2) to a set of OEdges in the OGraph that represent the dataflow communication between O_1 and O_2.

To relate the dynamic and the static semantics of the analysis, we define an approximation relation (DF-APPROX) between a runtime state (S,H,K,L_I,L_E) and an analysis result (DO, DD, DE). It ensures that the runtime objects, runtime domains and runtime edges are consistent with their representatives in the statically extracted OGraph.

Approximation Relation (Df-Approx).

$$\forall\ \Sigma \vdash S,\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE)$$

$$\Longleftrightarrow$$

$$\forall \ell \in dom(S), \Sigma[\ell] = C{<}\overline{\ell'.d}{>}$$

$$\Longrightarrow$$

$$H[\ell] = O_C = \langle C{<}\overline{D}{>}\rangle \in DO$$

$$and\ \forall \ell'_j.d_j \in \overline{\ell'.d}\ K[\ell'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$$

$$and\ \forall d_i \in domains(C{<}\overline{\ell'.d}{>})$$

$$K[\ell.d_i] = D_i = \langle D_{id_i}, d_i\rangle\ \{(O_C, C{::}d_i) \mapsto D_i\} \in DD$$

$$and\ \forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f}$$

$$\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \rightarrow T_R$$

$$\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p}{>}$$

$$E'_k \in L_I[(\ell_{src}, \ell)]\ E'_k = \langle H[\ell_{src}], H[\ell], C'_k, Imp\rangle \in DE\ C'_k <: C_k$$

$$and\ \forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f}$$

$$\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \rightarrow T_R$$

$$\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k{<}\overline{p}{>}$$

$$E_k \in L_E[(\ell, \ell_{dst})]\ E_k = \langle H[\ell], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$$

Df-Approx states that given a well-typed store S of a program and an OGraph $\langle DO, DD, DE\rangle$ of the same program, there are maps H, K, L_I, and L_E, such that H maps each runtime object ℓ in the store to a unique OObject O_C from DO, K maps each runtime domain $\ell.d_i$ in the store to a unique ODomain D_i, and L_I and L_E map each pair of runtime objects (ℓ_{src}, ℓ) and (ℓ, ℓ_{dst}) to OEdges from DE. Df-Approx ensures the consistency of these mappings with the ownership relation, and with the dataflow communication.

The last two conditions relate runtime dataflow communication back to field reads, field writes, and method invocations that produce the corresponding import and export edges in DE. L_I maps a runtime dataflow communication from a runtime object ℓ_{src} to another runtime object ℓ back to an import OEdge E'_k from DE. By our definition of import dataflow communication, E'_k exists in DE due to a field read or a method invocation expression that has ℓ_{src} as its receiver. The condition also ensures that the edge's label is a subclass of C_k, the class of a field of ℓ_{src}'s class, or the return class of a method of ℓ_{src}'s class.

Similarly, L_E maps a runtime dataflow communication from a runtime object ℓ to another runtime object ℓ_{dst} back to an export OEdge E_k from DE. By our definition of export dataflow communication, E_k exists in DE due to a field write or a method invocation expression that has ℓ_{dst} as its receiver. The condition also ensures that the edge's label is a subclass of C_k, the class of a field of ℓ_{dst}'s class, or the class of a parameter on a method of ℓ_{dst}'s class.

Theorem: Dataflow Object Graph Soundness.

$$\begin{array}{ll}
\textit{If} & G = \langle DO, DD, DE \rangle \\
& DO, DD, DE \vdash (CT, e_{root}) \\
& \forall e,\ \theta_0 \vdash e; \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E \\
& \Sigma \vdash S \\
\textit{then} & DO, DD, DE \vdash_{CT,H} \Sigma \\
& (S, H, K, L_I, L_E) \sim (DO, DD, DE)
\end{array}$$

where $\rightsquigarrow^*_G$ relation is the reflexive and transitive closure of $\rightsquigarrow_G$ relation, and θ_0 is the location of the first object instantiated by e_{root}. To prove the Object Graph Soundness theorem, we prove the Dataflow Preservation and Dataflow Progress theorems, which extend the standard FDJ Preservation and Progress. The common parts are highlighted.

Theorem: Dataflow Preservation (Subject reduction).

$$\begin{array}{l}
\textit{If } \boxed{\emptyset, \Sigma, \theta \vdash e : T} \\
\boxed{\Sigma \vdash S} \\
DO, DD, DE \vdash_{CT,H} \Sigma \\
\emptyset, \emptyset, DO, DD, DE \vdash_O e \\
(S, H, K, L_I, L_E) \sim (DO, DD, DE) \\
\theta \vdash \boxed{\text{e; S}} ; H; K; L_I; L_E \leadsto_G \boxed{\text{e'; S'}} ; H'; K'; L_I'; L_E' \\
\textit{then } \boxed{\textit{there exists } \Sigma' \supseteq \Sigma \textit{ and } T' <: T \textit{ such that}} \\
\boxed{\emptyset, \Sigma', \theta \vdash e' : T' \textit{ and } \Sigma' \vdash S'} \\
(S', H', K', L_I', L_E') \sim (DO, DD, DE) \\
\emptyset, \emptyset, DO, DD, DE \vdash_O e' \\
\textit{and } DO, DD, DE \vdash_{CT,H} \Sigma'
\end{array}$$

Theorem: Dataflow Progress.

$$\begin{array}{l}
\textit{If } \boxed{\emptyset, \Sigma, \theta \vdash e : T} \\
\boxed{\Sigma \vdash S} \\
DO, DD, DE \vdash_{CT,H} \Sigma \\
\emptyset, \emptyset, DO, DD, DE \vdash_O e \\
(S, H, K, L_I, L_E) \sim (DO, DD, DE) \\
\textit{then either } \boxed{\textit{e is a value}} \\
\textit{or else } \theta \vdash \boxed{\text{e}; S} ; H; K; L_I; L_E \leadsto_G \boxed{e'; S'} ; H'; K'; L_I'; L_E'
\end{array}$$

4.4 Theorem: Dataflow Preservation (Subject reduction)

If

$\boxed{\emptyset, \Sigma, \theta \vdash e : T}$

$\boxed{\Sigma \vdash S}$

$DO, DD, DE \vdash_{CT,H} \Sigma$

$\emptyset, \emptyset, DO, DD, DE \vdash_O e$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$

$\theta \vdash \boxed{e; S}; H; K; L_I; L_E \rightsquigarrow_G \boxed{e'; S'}; H'; K'; L'_I; L'_E$

$then$

$\boxed{\textit{there exists } \Sigma' \supseteq \Sigma \textit{ and } T' <: T \textit{ such that } \emptyset, \Sigma', \theta \vdash e' : T' \textit{ and } \Sigma' \vdash S'}$

$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

$\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

$\textit{and } DO, DD, DE \vdash_{CT,H} \Sigma'$

The Dataflow Preservation theorem extends the FDJ Type Preservation theorem (the common parts are highlighted). Those parts are proved by induction over the derivation of the FDJ evaluation relation : $e; S \rightsquigarrow e'; S'$.

Proof: We prove preservation by induction on the instrumented evaluation relation

$$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$$

The most interesting cases are Ir-New, Ir-Read (page 32), Ir-Write (page 33), and Ir-Invk (page 34).

Case Ir-New: e = `new` $C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v})$, and $e' = \ell$.

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$	By assumption
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$	By assumption
$\forall \ell \in dom(S),\ \Sigma(\ell) = C{<}\overline{\ell'.d}{>}$	Since $\Sigma \vdash S$
$H[\theta] = O = \langle C{<}\overline{D}{>} \rangle \in DO$	By Df-Approx
$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j \rangle \in rng(DD)$	By Df-Approx
$\forall d_i \in domains(C{<}\overline{\theta'.d}{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i \rangle$	
$\quad \{(O, d_i) \mapsto D_i\} \in DD$	By Df-Approx
$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$	
$\quad \forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$	
$\qquad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p}{>}$	
$\qquad\quad E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp \rangle \in DE \quad C'_k <: C_k$	By Df-Approx
$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$	
$\quad \forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$	
$\qquad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k{<}\overline{p}{>}$	
$\qquad\quad E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp \rangle \in DE \quad C'_k <: C_k$	By Df-Approx
$O_C = \langle C{<}\overline{D}{>} \rangle \in DO$	By sub-derivation of Ir-New
$S' = S[\ell \mapsto C{<}\overline{p}{>}(\overline{v})]$	By sub-derivation of Ir-New
$H' = H[\ell \mapsto O_C]$	By sub-derivation of Ir-New
$\overline{p} = \overline{\ell'.d}\ \forall i \in 1..\vert\overline{\ell'.d}\vert\ D_i = K[\ell'_i.d_i]$	By sub-derivation of Ir-New
$\forall (\texttt{domain}\ d_j) \in domains(C{<}\overline{p}{>}) \quad D_j = DD[(O_C, d_j)]$	
$K' = K[\ell.d_j \mapsto D_j]$	By sub-derivation of Ir-New
$L'_I = L_I \quad L'_E = L_E$	By sub-derivation of Ir-New
$\exists \Sigma' \supseteq \Sigma\ \ and\ T' <: T\ s.t.\ \emptyset, \Sigma', \theta \vdash e' : T'\ and\ \Sigma' \vdash S'$	By FDJ Type Preservation
$\Sigma'[\ell] = C_\ell{<}\overline{\ell'.d}{>}$	By $\Sigma' \vdash S'$
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$	By Df-Approx

This proves (1).

$\emptyset, \emptyset, DO, DD, DE \vdash_O e'$	By Df-Loc, since $e' = \ell$

This proves (2).

$DO, DD, DE \vdash_{CT,H} \Sigma$	By assumption
$\forall \ell \in dom(S), \Sigma[\ell] = C_\ell{<}\overline{p}{>}$	By sub-derivation of Df-Sigma
$H[\ell] = O_\ell = \langle C_\ell{<}\overline{D_\ell}{>} \rangle \in DO$	By sub-derivation of Df-Sigma
$\forall m.\ mbody(m, C_\ell{<}\overline{p}{>}) = (\overline{x} : \overline{T},\ e_R)$	By sub-derivation of Df-Sigma
$\{\overline{x} : \overline{T},\ \texttt{this} : C_\ell{<}\overline{p}{>}\}, \emptyset, DO, DD, DE \vdash_{O_\ell} e_R$	By sub-derivation of Df-Sigma
$O_C = \langle C{<}\overline{D}{>} \rangle \in DO$	By sub-derivation of Ir-New
$S' = S[\ell \mapsto C{<}\overline{p}{>}(\overline{v})]$	By sub-derivation of Ir-New
$H' = H[\ell \mapsto O_C]$	By sub-derivation of Ir-New
$\emptyset, \emptyset, DO, DD, DE \vdash_O e$	By assumption with e, Υ below
$e = \texttt{new}\ C{<}\overline{\ell'.d}{>}(\overline{v}),\ \text{and}\ \Upsilon = \emptyset$	

$\forall m.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$ By sub-derivation of Df-New

$C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow$

$\{\overline{x} : \overline{T}, \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, DO, DD, DE \vdash_{O_C} e_R$ By sub-derivation of Df-New

$\{\overline{x} : \overline{T}, \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \emptyset, DO, DD, DE \vdash_{O_C} e_R$ By Df-Strengthening Lemma

$\forall \ell \in dom(S'), \Sigma'[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}$

  $H'[\ell] = O_\ell = \langle C_\ell\texttt{<}\overline{D_\ell}\texttt{>}\rangle \in DO$

  $\forall m.\ mbody(m, C_\ell\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$

    $\{\overline{x} : \overline{T},\ \texttt{this} : C_\ell\texttt{<}\overline{p}\texttt{>}\}, \emptyset, DO, DD, DE \vdash_{O_\ell} e_R$ By above

$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By Df-Sigma with above H' and Σ'

This proves (3).

Case Ir-Read: e $= \ell.f_i$, and $e' = v_i$.

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$ By assumption

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption

$\forall \ell \in dom(S),\ \Sigma(\ell) = C\texttt{<}\overline{\ell'.d}\texttt{>}$ Since $\Sigma \vdash S$

$H[\theta] = O = \langle C\texttt{<}\overline{D}\texttt{>}\rangle \in DO$ By Df-Approx

$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$ By Df-Approx

$\forall d_i \in domains(C\texttt{<}\overline{\theta'.d}\texttt{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i\rangle$

  $\{(O, d_i) \mapsto D_i\} \in DD$ By Df-Approx

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$

  $\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \rightarrow T_R$

    $\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\}\quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

      $E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp\rangle \in DE\quad C'_k <: C_k$ By Df-Approx

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

  $\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \rightarrow T_R$

    $\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\}\quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

      $E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$ By Df-Approx

$S' = S, H' = H, K' = K, L'_E = L_E$ By sub-derivation of Ir-Read

$S[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}(\overline{v})\quad fields(C_\ell\texttt{<}\overline{p}\texttt{>}) = \overline{T'}\ \overline{f}$ By sub-derivation of Ir-Read

$O = H[\theta] \quad O_\ell = H[\ell] \quad T'_i = C_i{<}\overline{p'}{>}$ By sub-derivation of Ir-Read
$E' = \langle O_\ell, O, C_v, Imp\rangle \in DE \quad C_v <: C_i$ By sub-derivation of Ir-Read
$L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}]$ By sub-derivation of Ir-Read
$\forall \ell_{src} \in dom(H'),\ fields(\Sigma'[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$
$\quad \forall m.\ mtype(m, \Sigma'[\ell_{src}]) = \overline{T} \to T_R$
$\quad\quad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p}{>}$
$\quad\quad\quad E'_k \in L'_I[(\ell_{src}, \theta)] = \langle H'[\ell_{src}], H'[\theta], C'_k, Imp\rangle \in DE\ C'_k <: C_k$ By above, since $\Sigma' = \Sigma$
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By Df-Approx
This proves (1).

$\emptyset, \emptyset, DO, DD, DE \vdash_O e'$ By Df-Loc, since $e' = v_i$
This proves (2).

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption
$S' = S, H' = H$ By sub-derivation of Ir-Read
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By Df-Sigma with the above H' and $\Sigma' = \Sigma$
This proves (3).

Case Ir-Write: e $= \ell.f_i = v$, and $e' = v$
To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$
(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$ By assumption
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption
$\forall \ell \in dom(S),\ \Sigma(\ell) = C{<}\overline{\ell'.d}{>}$ Since $\Sigma \vdash S$
$H[\theta] = O = \langle C{<}\overline{D}{>}\rangle \in DO$ By Df-Approx
$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$ By Df-Approx
$\forall d_i \in domains(C{<}\overline{\theta'.d}{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i\rangle$
$\quad \{(O, d_i) \mapsto D_i\} \in DD$ By Df-Approx
$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$
$\quad \forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$
$\quad\quad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p}{>}$
$\quad\quad\quad E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ By Df-Approx

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \rightarrow T_R$

$\quad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

$\quad\quad\quad E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$ By Df-Approx

$H' = H, K' = K, L'_I = L_I$ By sub-derivation of Ir-Write

$S[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}(\overline{v}) \quad fields(C_\ell\texttt{<}\overline{p}\texttt{>}) = \overline{T'}\ \overline{f}$ By sub-derivation of Ir-Write

$S' = S[\ell \mapsto C_\ell\texttt{<}\overline{p}\texttt{>}([v/v_i]\overline{v})]$ By sub-derivation of Ir-Write

$O = H[\theta] \quad O_\ell = H[\ell] \quad T'_i = C_i\texttt{<}\overline{p'}\texttt{>}$ By sub-derivation of Ir-Write

$E = \langle O, O_\ell, C_v, Exp\rangle \in DE \quad C_v <: C_i$ By sub-derivation of Ir-Write

$L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E\}]$ By sub-derivation of Ir-Write

$\exists \Sigma' \supseteq \Sigma\ \ and\ T' <: T\ s.t.\ \emptyset, \Sigma', \theta \vdash e' : T'\ and\ \Sigma' \vdash S'$ By FDJ Type Preservation

$\Sigma'[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}$ $\Sigma' \vdash S'$

$\forall \ell_{dst} \in dom(H'),\ fields(\Sigma'[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma'[\ell_{dst}]) = \overline{T} \rightarrow T_R$

$\quad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

$\quad\quad\quad E_k \in L'_E[(\theta, \ell_{dst})] = \langle H'[\theta], H'[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$ By above

$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By Df-Approx

This proves (1).

$\emptyset, \emptyset, DO, DD, DE \vdash_O e'$ By Df-Loc, since $e' = v_i$

This proves (2).

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption

$\forall \ell \in dom(S), \Sigma[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}$

$\quad H[\ell] = O_\ell = \langle C_\ell\texttt{<}\overline{D_\ell}\texttt{>}\rangle \in DO$

$\quad \forall m.\ mbody(m, C_\ell\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$

$\quad\quad \{\overline{x} : \overline{T},\ \texttt{this} : C_\ell\texttt{<}\overline{p}\texttt{>}\}, \emptyset, DO, DD, DE \vdash_{O_\ell} e_R$ By sub-derivation of Df-Sigma

$H' = H$ By sub-derivation of Ir-Write

$S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \quad fields(C\texttt{<}\overline{p}\texttt{>}) = \overline{T}\ \overline{f}$ By sub-derivation of Ir-Write

$S' = S[\ell \mapsto C\texttt{<}\overline{p}\texttt{>}([v/v_i]\overline{v})]$ By sub-derivation of Ir-Write

$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By Df-Sigma with the above H' and $\Sigma' = \Sigma$

This proves (3).

Case Ir-Invk: e $= \ell.m(\overline{v})$, and $e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E$	By assumption
$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$	By assumption
$\forall \ell \in dom(S),\ \Sigma(\ell) = C{<}\overline{\ell'.d}{>}$	Since $\Sigma \vdash S$
$H[\theta] = O = \langle C{<}\overline{D}{>}\rangle \in DO$	By Df-Approx
$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$	By Df-Approx
$\forall d_i \in domains(C{<}\overline{\theta'.d}{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i\rangle$	
$\quad \{(O, d_i) \mapsto D_i\} \in DD$	By Df-Approx
$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$	
$\quad \forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \rightarrow T_R$	
$\quad\quad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p}{>}$	
$\quad\quad\quad E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$	By Df-Approx
$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$	
$\quad \forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \rightarrow T_R$	
$\quad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k{<}\overline{p}{>}$	
$\quad\quad\quad E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$	By Df-Approx
$S' = S \quad H' = H \quad K' = K$	By sub-derivation of Ir-Invk
$S[\ell] = C_\ell{<}\overline{p}{>}(\overline{v}) \quad mbody(m, C_\ell{<}\overline{p}{>}) = (\overline{x}, e_R)$	By sub-derivation of Ir-Invk
$H[\theta] = O \quad H[\ell] = O_\ell$	By sub-derivation of Ir-Invk
$mtype(m, C_\ell{<}\overline{p}{>}) = \overline{T} \rightarrow T_R \quad T_R = C_R{<}\overline{p'}{>}$	By sub-derivation of Ir-Invk
$E' = \langle O_\ell, O, C'_R, Imp\rangle \in DE \quad C'_R <: C_R$	By sub-derivation of Ir-Invk
$L'_I = L_I[(\ell, \theta) \mapsto_\cup \{E'\}]$	By sub-derivation of Ir-Invk
$\forall i \in 1..\lvert\overline{T}\rvert\ T_i = C_i{<}\overline{p''}{>} \quad E_i = \langle O, O_\ell, C'_i, Exp\rangle \in DE \quad C'_i <: C_i$	By sub-derivation of Ir-Invk
$L'_E = L_E[(\theta, \ell) \mapsto_\cup \{E_i\}]$	By sub-derivation of Ir-Invk
$\exists \Sigma' \supseteq \Sigma \quad and\ T' <: T\ s.t.\ \emptyset, \Sigma', \theta \vdash e' : T'\ and\ \Sigma' \vdash S'$	By FDJ Type Preservation
$\Sigma'[\ell] = C_\ell{<}\overline{\ell'.d}{>}$	$\Sigma' \vdash S'$

$\forall \ell_{src} \in dom(H'),\ fields(\Sigma'[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma'[\ell_{src}]) = \overline{T} \to T_R$

$\quad\quad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

$\quad\quad\quad E'_k \in L'_I[(\ell_{src}, \theta)] = \langle H'[\ell_{src}], H'[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ — By above

$\forall \ell_{dst} \in dom(H'),\ fields(\Sigma'[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma'[\ell_{dst}]) = \overline{T} \to T_R$

$\quad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

$\quad\quad\quad E_k \in L'_E[(\theta, \ell_{dst})] = \langle H'[\theta], H'[\ell_{dst}], C'_k, Exp\rangle \in DE \quad C'_k <: C_k$ — By above

$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ — By DF-APPROX

This proves (1).

$\emptyset, \emptyset, DO, DD, DE \vdash_O e$	By assumption
$e = \ell.m(\overline{v}) \quad e_0 = \ell \quad \overline{e} = \overline{v}$	By assumption
$e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$	By assumption
$\emptyset, \Sigma, \theta \vdash e : T$	By assumption
$\exists \Sigma' \supseteq \Sigma$ *and* $T' <: T$ *s.t.* $\emptyset, \Sigma', \theta \vdash e' : T'$ *and* $\Sigma' \vdash S'$	By FDJ Type Preservation
$\quad e_0 : T_0 \quad T_0 = C_\ell\texttt{<}\overline{p}\texttt{>}$	By sub-derivation of DF-INVK
$mtype(m, C_\ell\texttt{<}\overline{p}\texttt{>}) = \overline{T} \to T_R$	By sub-derivation of DF-INVK
$\emptyset, \emptyset, DO, DD, DE \vdash_O e_0$	By sub-derivation of DF-INVK
$\emptyset, \emptyset, DO, DD, DE \vdash_O \overline{e}$	By sub-derivation of DF-INVK
$\{\overline{x} : \overline{T}, \texttt{this} : C_\ell\texttt{<}\alpha, \overline{\beta}\texttt{>}\}, \Sigma, \theta \vdash e_R : T_R \quad T_R <: T$	By FDJ *Meth*OK:
$S[\ell] = C_\ell\texttt{<}d, \overline{d'}\texttt{>}(\overline{v})$	By sub-derivation of IR-INVK
$mbody(m, C_\ell\texttt{<}d, \overline{d'}\texttt{>}) = (\overline{x}, e_R)$	By sub-derivation of IR-INVK
$\Sigma[\ell] = C_\ell\texttt{<}d, \overline{d'}\texttt{>} = T_0$	Since $e_0 = \ell$, by T-Store
$e_0 : C_\ell\texttt{<}d, \overline{d'}\texttt{>}$	Since $e_0 = \ell$, by T-Store
$mtype(m, C_\ell\texttt{<}d, \overline{d'}\texttt{>}) = \overline{T} \to T_R$	Since $e_0 = \ell$, by T-Store
$\overline{v} : \overline{T_a}$	By inversion
$\overline{T_a} <: [\overline{v}/\overline{x}, \ell/\texttt{this}]\overline{T}$	For some $\overline{T_a}$ and $\overline{T}$
there are some $D\texttt{<}\overline{d}\texttt{>}$ and T'_R so that:	By Method Lemma
$T'_R <: T_R$ and $C_\ell\texttt{<}d, \overline{d'}\texttt{>} <: D\texttt{<}\overline{d}\texttt{>}$	By Method Lemma
so that $\{\overline{x} : \overline{T}, \texttt{this} : D\texttt{<}\overline{d}\texttt{>}\}, \Sigma, \theta \vdash e_R : T'_R$	By Method Lemma
there exists $T_S, T_S <: T'_R$ s. t. $[\overline{v}/\overline{x}, \ell/\texttt{this}]e_R : T_S$	Since term substitution preserves typing
$T_S <: T'_R$ and $T'_R <: T_R$	By above
$T_S <: T_R$	By transitivity of $<:$
Take $T = T' = T_R$ in FDJ Preservation	

$\{\overline{x}:\overline{T}, \texttt{this}: C_\ell\texttt{<}d,\overline{d'}\texttt{>}\}, \emptyset, DO, DD, DE \vdash_{O_C} e_R$ By Df-Sigma

$O_C = H[\ell]$ By Df-Sigma

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell$ By Df-Loc

$\emptyset, \emptyset, DO, DD, DE \vdash_{O_C} [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$ By Df-Substitution Lemma

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$ By Df-Context

This proves (2).

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption

$S' = S, H' = H$ By sub-derivation of Ir-Invk

$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By Df-Sigma with the above H' and $\Sigma' = \Sigma$

This proves (3).

Case Ir-Context: e $= \ell \triangleright v$, and $e' = v$

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption

$S' = S, H' = H, K' = K, L'_I = L_I, L'_E = L_E$ By sub-derivation of Ir-Context

This proves (1).

$\emptyset, \emptyset, DO, DD, DE \vdash_O e'$ By Df-Loc, since $e' = v$

This proves (2).

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption

$S' = S, H' = H$ By sub-derivation of Ir-Context

$DO, DD, DE \vdash_{CT,H'} \Sigma'$ Take $\Sigma' = \Sigma$

This proves (3).

Case Irc-New: e $= \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n})$, and $e' = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e'_i, e_{i+1..n})$.

To Show:

(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-New
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-New
$\emptyset, \emptyset, DO, DD, DE \vdash_O e'_i$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e'_i, e_{i+1..n})$ By Df-New
This proves (2).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-New
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Read: e $= e_0.f_k$, and $e' = e'_0.f_k$.
To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$
(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-Read
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-Read
$\emptyset, \emptyset, DO, DD, DE \vdash_O e'_0$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O e'_0.f_k$ By Df-Read
This proves (2).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-Read
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Write-Rcv: e $= (e_0.f_k = e_1)$, and $e' = (e'_0.f_k = e_1)$.
To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$
(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Write-Rcv
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Write-Rcv
$\emptyset, \emptyset, DO, DD, DE \vdash_O e_0'$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O e_1$ By Df-Write
$\emptyset, \emptyset, DO, DD, DE \vdash_O e_0'.f_k = e_1$ By Df-Write
This proves (2).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Write-Rcv
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Write-Arg: e $= (v.f_k = e_1)$, and $e' = (v.f_k = e_1')$.
To Show:
(1) $(S', H', K', L_I', L_E') \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$
(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e_1'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Write-Arg
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e_1'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Write-Arg
$\emptyset, \emptyset, DO, DD, DE \vdash_O e_1'$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O e_0'.f_k = e_1'$ By Df-Write
This proves (2).

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e_1'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Write-Arg
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Recvinvk: e $= e_0.m(\overline{e})$, and $e' = e_0'.m(\overline{e})$.
To Show:
(1) $(S', H', K', L_I', L_E') \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$

(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-RecvInvk
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-RecvInvk
$\emptyset, \emptyset, DO, DD, DE \vdash_O e'_0$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O \overline{e}$ By Df-Invk
$\emptyset, \emptyset, DO, DD, DE \vdash_O e'_0.m(\overline{e})$ By Df-Invk
This proves (2).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-RecvInvk
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Arcinvk: e $= v.m(v_{1..i-1}, e_i, e_{i+1..n})$,and $e' = v.m(v_{1..i-1}, e'_i, e_{i+1..n})$.
To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$
(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$
(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-ArgInvk
$(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-ArgInvk
$\emptyset, \emptyset, DO, DD, DE \vdash_O e'_i$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O v.m(v_{1..i-1}, e'_i, e_{i+1..n})$ By Df-Invk
This proves (2).

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By sub-derivation of Irc-ArgInvk
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

Case Irc-Context: e $= \ell \triangleright e_0$, and $e' = \ell \triangleright e'_0$.
To Show:
(1) $(S', H', K', L'_I, L'_E) \sim (DO, DD, DE)$

(2) $\emptyset, \emptyset, DO, DD, DE \vdash_O e'$
(3) $DO, DD, DE \vdash_{CT,H'} \Sigma'$

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Context
$(S', H', K', L_I', L_E') \sim (DO, DD, DE)$ By induction hypothesis
This proves (1).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Context
$O_\ell = H[\ell]$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_{O_\ell} e_0'$ By induction hypothesis
$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell \triangleright e_0'$ By Df-Context
This proves (2).

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e_0'; S'; H'; K'; L_I'; L_E'$ By sub-derivation of Irc-Context
$DO, DD, DE \vdash_{CT,H'} \Sigma'$ By induction hypothesis, take $\Sigma' = \Sigma$
This proves (3).

■

4.5 Theorem: Dataflow Progress

$$
\begin{array}{l}
\textit{If} \\
\quad \boxed{\emptyset, \Sigma, \theta \vdash e : T} \\
\quad \boxed{\Sigma \vdash S} \\
\quad DO, DD, DE \vdash_{CT,H} \Sigma \\
\quad \emptyset, \emptyset, DO, DD, DE \vdash_{O} e \\
\quad (S, H, K, L_I, L_E) \sim (DO, DD, DE) \\
\textit{then} \\
\quad \textit{either } \boxed{e \textit{ is a value}} \\
\quad \textit{or else } \theta \vdash \boxed{e; S}; H; K; L_I; L_E \leadsto_G \boxed{e'; S'}; H'; K'; L'_I; L'_E
\end{array}
$$

Proof: We prove progress by derivation of $\emptyset, \emptyset, DO, DD, DE \vdash_O e$, with a case analysis on the last typing rule used. The most interesting cases are DF-NEW, DF-READ (page 43), DF-WRITE (page 45), and DF-INVK (page 47).

Case DF-NEW : e = $new\ C{<}\overline{p}{>}(\overline{e})$.

Subcase $\overline{e} = \overline{v}$ that is $e = new\ C{<}\overline{p}{>}(\overline{v})$. Take $e' = \ell$, then IR-NEW can apply.

To show:
(1) $\forall i \in |\overline{\ell'.d}| \quad D_i = K[\ell'_i.d_i]$
(2) $O_C = \langle C{<}\overline{D}{>}\rangle \quad O_C \in DO$
(3) $\forall d_j \in domains(C{<}\overline{\ell'.d}{>}) \quad D_j = DD[(O_C, d_j)]$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$	By assumption
$\forall \ell \in dom(S), \Sigma[\ell] = C{<}\overline{\ell'.d}{>}$	$\Sigma \vdash S$
$H[\ell] = O_C = \langle C{<}\overline{D}{>}\rangle \in DO$	By DF-APPROX
$\forall \ell'_j.d_j \in \overline{\ell'.d} \quad K[\ell'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$	By DF-APPROX
$\forall d_i \in domains(C{<}\overline{\ell'.d}{>})\ K[\ell.d_i] = D_i = \langle D_{id_i}, d_i\rangle$	
$\{(O_C, d_i) \mapsto D_{\ell i}\} \in DD$	By DF-APPROX

This proves (1).

$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}$

$\emptyset, \emptyset, DO, DD, DE \vdash_O e$	By assumption
$\forall i \in 1..\lvert\overline{p}\rvert \quad D_i = DD[(O, p_i)]$	By sub-derivation of Df-New
$params(C) = \overline{\alpha}$	By sub-derivation of Df-New
$O_C = \langle\ C\texttt{<}\overline{D}\texttt{>}\ \rangle \quad \{O_C\} \subseteq DO$	By sub-derivation of Df-New
This proves (2).	
$\{(O_C, \alpha_i) \mapsto D_i\} \subseteq DD$	By sub-derivation of Df-New
$\{(O_C, p_i) \mapsto D_i\} \subseteq DD$	By sub-derivation of Df-New
$DO, DD, DE \vdash_O ddomains(C, O_C)$	By sub-derivation of Df-New
This proves (3).	By Df-Domains Lemma

Subcase $e = new\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n})$. Then Irc-New can apply.

$\Gamma, \Upsilon, DO, DD, DE \vdash_O e_i$	By sub-derivation of Df-New
$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G S'; H'; K'; L'_I; L'_E$	By induction hypothesis
$\theta \vdash \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n})S; H; K; L_I; L_E \rightsquigarrow_G$	
$\quad \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E$	By Irc-New
This proves (2).	
$\{(O_C, \alpha_i) \mapsto D_i\} \subseteq DD$	By sub-derivation of Df-New
$\{(O_C, p_i) \mapsto D_i\} \subseteq DD$	By sub-derivation of Df-New
$DO, DD, DE \vdash_O ddomains(C, O_C)$	By sub-derivation of Df-New
Take $e' = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e'_i, e_{i+1..n})$	By Df-Domains Lemma

Case DF-VAR : $e = x$.
Not applicable since variable is not a closed term.

Case DF-LOC : $e = \ell$.
e is a value.

Case DF-READ : e = $e_0.f_i$. There are two subcases to consider depending on whether the receiver e_0 is a value.

Subcase $e_0 = \ell$. Then $e = \ell.f_i$
To show:
(1) $O = H[\theta]$
(2) $O_\ell = H[\ell]$
(3) $E = \langle O_\ell, O, C_v, Imp\rangle \in DE \quad C_v <: C_i$

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption

$\forall \ell' \in dom(S), \Sigma[\ell'] = C'\texttt{<}\overline{p}\texttt{>}$ By sub-derivation of DF-SIGMA

$H[\ell'] = O' = \langle C'\texttt{<}\overline{D'}\texttt{>}\rangle \in DO$ By sub-derivation of DF-SIGMA

$H[\theta] = O = \langle O_{\theta id}, C\texttt{<}\overline{D}\texttt{>}\rangle \in DO$ Since $\theta \in dom(S)$

$H[\ell] = O_\ell = \langle O_{\ell id}, C_\ell\texttt{<}\overline{D_\ell}\texttt{>}\rangle \in DO$ Since $\ell \in dom(S)$

this proves (1), and (2).

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption

$\forall \ell \in dom(S),\ \Sigma(\ell) = C\texttt{<}\overline{\ell'.d}\texttt{>}$ Since $\Sigma \vdash S$

$H[\theta] = O = \langle C\texttt{<}\overline{D}\texttt{>}\rangle \in DO$ By DF-APPROX

$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$ By DF-APPROX

$\forall d_i \in domains(C\texttt{<}\overline{\theta'.d}\texttt{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i\rangle$

  $\{(O, d_i) \mapsto D_i\} \in DD$ By DF-APPROX

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$

  $\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \to T_R$

    $\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

      $E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ By DF-APPROX

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

  $\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \to T_R$

    $\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

      $E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$ By DF-APPROX

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell.f_i$ By assumption

$fields(\Sigma[\ell]) = \overline{T'}\ \overline{f}$ By FDJ T-Store

Since $e_0 = \ell \in dom(H)$

$\ell : \Sigma[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}\ (T'_i\ f_i) \in fields(C_\ell\texttt{<}\overline{p}\texttt{>})\ T'_i = C_i\texttt{<}\overline{p'}\texttt{>}$ By sub-derivation of DF-READ

$DO, DD, DE \vdash_O import(\Sigma[\ell], T'_i)$ By sub-derivation of DF-READ

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell$ By sub-derivation of DF-READ

Take $\ell_{src} = \ell$.

$\ell : \Sigma[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}\ (T'_i\ f_i) \in fields(C_\ell\texttt{<}\overline{p}\texttt{>})\ T'_i = C_i\texttt{<}\overline{p'}\texttt{>}$ By above sub-derivation

$\forall m.\ mtype(m, \Sigma[\ell]) = \overline{T} \rightarrow T_R$

$\forall T_k \in \{\overline{T'}\} \cup \{T_R\} \quad T_k = C_k\texttt{<}\overline{p''}\texttt{>}$

$\langle H[\ell], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ By above Df-Approx

Take $T_k = T'_i \in \overline{T'}, C_i = C_k$, and $C_v = C'_k$, this proves (3).

Subcase $e_0 = e'_0.f_i$. That is, e_0 is not a value

From Irc-Read:

$\theta \vdash e'_0; S; H; K; L_I; L_E \rightsquigarrow_G e''_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash e'_0.f_i; S; H; K; L_I; L_E \rightsquigarrow_G e''_0.f_i; S'; H'; K'; L'_I; L'_E$ By Irc-Read

Take $e' = e''_0.f_i$.

Case DF-WRITE : e = $(e_0.f_i = e_1)$. There are three subcases to consider depending on whether the receiver e_0, and e_1 are values.

Subcase $e_0 = \ell$, and $e_1 = v$. Then $e = (\ell.f_i = v)$

To show:

(1) $O = H[\theta]$

(2) $O_\ell = H[\ell]$

(3) $E = \langle O, O_\ell, C_v, Exp\rangle \in DE \quad C_v <: C_i$

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption

$\forall \ell' \in dom(S), \Sigma[\ell'] = C'\texttt{<}\overline{p}\texttt{>}$ By sub-derivation of DF-SIGMA

$H[\ell'] = O' = \langle C'\texttt{<}\overline{D'}\texttt{>}\rangle \in DO$ By sub-derivation of DF-SIGMA

$H[\theta] = O = \langle O_{\theta id}, C\texttt{<}\overline{D}\texttt{>}\rangle \in DO$ Since $\theta \in dom(S)$

$H[\ell] = O_\ell = \langle O_{\ell id}, C_\ell\texttt{<}\overline{D_\ell}\texttt{>}\rangle \in DO$ Since $\ell \in dom(S)$

this proves (1), and (2).

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption

$\forall \ell \in dom(S),\ \Sigma(\ell) = C\texttt{<}\overline{\ell'.d}\texttt{>}$ Since $\Sigma \vdash S$

$H[\theta] = O = \langle C\texttt{<}\overline{D}\texttt{>}\rangle \in DO$ By DF-APPROX

$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$ By DF-APPROX

$\forall d_i \in domains(C\texttt{<}\overline{\theta'.d}\texttt{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i\rangle$

  $\{(O, d_i) \mapsto D_i\} \in DD$ By DF-APPROX

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$

  $\forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \rightarrow T_R$

    $\forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

      $E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ By DF-APPROX

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

  $\forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \rightarrow T_R$

    $\forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k\texttt{<}\overline{p}\texttt{>}$

      $E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$ By DF-APPROX

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell.f_i = v$ By assumption:

Since $e_0 = \ell \in dom(H) \quad e_1 = v$:

$\ell : \Sigma[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}\ (T'_i\ f_i) \in fields(C_\ell\texttt{<}\overline{p}\texttt{>}) = \overline{T'}\ \overline{f}\ T'_i = C_i\texttt{<}\overline{p'}\texttt{>}$ By sub-derivation of DF-WRITE

$v : \Sigma[v] = C_v\texttt{<}\overline{p''}\texttt{>} \quad \Sigma[v] <: T'_i$ By sub-derivation of DF-WRITE

$DO, DD, DE \vdash_O export(\Sigma[\ell], \Sigma[v])$ By sub-derivation of DF-WRITE

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell$ By sub-derivation of DF-WRITE

$\emptyset, \emptyset, DO, DD, DE \vdash_O v$ By sub-derivation of DF-WRITE

Take $\ell_{dst} = \ell$.

$\ell : \Sigma[\ell] = C_\ell\texttt{<}\overline{p}\texttt{>}\ (T'_i\ f_i) \in fields(C_\ell\texttt{<}\overline{p}\texttt{>}) = \overline{T'}\ \overline{f}\ T'_i = C_i\texttt{<}\overline{p'}\texttt{>}$ By the sub-derivation above

$\forall m.\ mtype(m, \Sigma[\ell]) = \overline{T} \rightarrow T_R$

  $\forall T_k \in \{\overline{T'}\} \cup \{\overline{T}\} \quad T_k = C_k\texttt{<}\overline{p'''}\texttt{>}$

    $\langle H[\theta], H[\ell], C'_k, Exp\rangle \in DE \quad C'_k <: C_k$ By above DF-APPROX

Take $T_k = T'_i \in \overline{T'}, C_i = C_k$, and $C_v = C'_k$, this proves (3).

Subcase $e_0 = e'_0$. Then $e = (e'_0.f_i = e_1)$

From IRC-Write-Rcv:

$\theta \vdash e'_0; S; H; K; L_I; L_E \rightsquigarrow_G e''_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash e'_0.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G e''_0.f_i = e_1; S'; H'; K'; L'_I; L'_E$ By IRC-Write-Rcv

Take $e' = (e''_0.f_i = e_1)$.

Subcase $e_0 = v$, and $e_1 = e'_1$. Then $e = (v.f_i = e'_1)$

From IRC-Write-Arg:

$\Gamma, \Upsilon, DO, DD, DE \vdash_O e_1$ By sub-derivation of Df-Write

$\theta \vdash e_1; S; H; K; L_I; L_E \rightsquigarrow_G e'_1; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash v.f_i = e_1; S; H; K; L_I; L_E \rightsquigarrow_G v.f_i = e'_1; S'; H'; K'; L'_I; L'_E$ By IRC-Write-Arg

Take $e' = (v.f_i = e'_1)$.

Case DF-INVK : e $= e_0.m(\overline{e})$. There are three subcases to consider, depending on whether the receiver e_0, or the arguments $\overline{e}$ are values.

Subcase $e_0 = \ell$, and $\overline{e} = \overline{v}$ that is $e = \ell.m(\overline{v})$

To show:

(1) $O = H[\theta]$

(2) $O_\ell = H[\ell]$

(3) $mtype(m, C_\ell\texttt{<}\overline{p}\texttt{>}) = \overline{T} \rightarrow T_R$ $T_R = C_R\texttt{<}\overline{p'}\texttt{>}$

$E' = \langle O_\ell, O, C'_R, Imp\rangle \in DE$ $C'_R <: C_R$

(4) $\forall i \in 1..|\overline{T}|$ $T_i = C_i\texttt{<}\overline{p''}\texttt{>}$ $E_i = \langle O, O_\ell, C'_i, Exp\rangle \in DE$ $C'_i <: C_i$

$DO, DD, DE \vdash_{CT,H} \Sigma$ By assumption

$\forall \ell' \in dom(S), \Sigma[\ell'] = C'{<}\overline{p}{>}$ By sub-derivation of DF-SIGMA

$H[\ell'] = O' = \langle C'{<}\overline{D'}{>}\rangle \in DO$ By sub-derivation of DF-SIGMA

$H[\theta] = O = \langle O_{\theta id}, C{<}\overline{D}{>}\rangle \in DO$ Since $\theta \in dom(S)$

$H[\ell] = O_\ell = \langle O_{\ell id}, C_\ell{<}\overline{D_\ell}{>}\rangle \in DO$ Since $\ell \in dom(S)$

this proves (1), and (2).

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$ By assumption

$\forall \ell \in dom(S),\ \Sigma(\ell) = C{<}\overline{\ell'.d}{>}$ Since $\Sigma \vdash S$

$H[\theta] = O = \langle C{<}\overline{D}{>}\rangle \in DO$ By DF-APPROX

$\forall \theta'_j.d_j \in \overline{\theta'.d}\ K[\theta'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(DD)$ By DF-APPROX

$\forall d_i \in domains(C{<}\overline{\theta'.d}{>})\ K[\theta.d_i] = D_i = \langle D_{id_i}, d_i\rangle$

$\quad \{(O, d_i) \mapsto D_i\} \in DD$ By DF-APPROX

$\forall \ell_{src} \in dom(H),\ fields(\Sigma[\ell_{src}]) = \overline{T_{src}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma[\ell_{src}]) = \overline{T} \rightarrow T_R$

$\quad\quad \forall T_k \in \{\overline{T_{src}}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p}{>}$

$\quad\quad\quad E'_k \in L_I[(\ell_{src}, \theta)] = \langle H[\ell_{src}], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ By DF-APPROX

$\forall \ell_{dst} \in dom(H),\ fields(\Sigma[\ell_{dst}]) = \overline{T_{dst}}\ \overline{f},$

$\quad \forall m.\ mtype(m, \Sigma[\ell_{dst}]) = \overline{T} \rightarrow T_R$

$\quad\quad \forall T_k \in \{\overline{T_{dst}}\} \cup \{\overline{T}\} \quad T_k = C_k{<}\overline{p}{>}$

$\quad\quad\quad E_k \in L_E[(\theta, \ell_{dst})] = \langle H[\theta], H[\ell_{dst}], C'_k, Exp\rangle \in DE\ C'_k <: C_k$ By DF-APPROX

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell.m(\overline{v})$ By assumption

$\ell : \Sigma[\ell] = C_\ell{<}\overline{\ell'.d}{>}$ By sub-derivation of DF-INVK

$mtype(m, C_\ell{<}\overline{\ell'.d}{>}) = \overline{T} \rightarrow T_R \quad T_R = C_R{<}\overline{p'}{>}$ By sub-derivation of DF-INVK

$DO, DD, DE \vdash_O import(\Sigma[\ell], T_R)$ By sub-derivation of DF-INVK

$\forall i \in 1..|\overline{v}|\ v_i : \Sigma[v_i]\ \Sigma[v_i] <: T_i\ DO, DD, DE \vdash_O export(\Sigma[\ell], \Sigma[v_i])$ By sub-derivation of DF-INVK

$\emptyset, \emptyset, DO, DD, DE \vdash_O \ell$ By sub-derivation of DF-INVK

$\emptyset, \emptyset, DO, DD, DE \vdash_O \overline{v}$ By sub-derivation of DF-INVK

Take $\ell_{src} = \ell$.

$mtype(m, \Sigma[\ell]) = \overline{T} \rightarrow T_R \quad T_R = C_R{<}\overline{p'}{>}$ By above sub-derivation

$fields(\Sigma[\ell]) = \overline{T'}\ \overline{f}$ By FDJ T-Store

$\forall T_k \in \{\overline{T'}\} \cup \{T_R\} \quad T_k = C_k{<}\overline{p'''}{>}$

$\quad \langle H[\ell], H[\theta], C'_k, Imp\rangle \in DE \quad C'_k <: C_k$ By above DF-APPROX

Take $T_k = T_R, C_R = C_k$ and $C'_R = C'_k$, this proves (3).

Take $\ell_{dst} = \ell$.

$mtype(m, \Sigma[\ell]) = \overline{T} \rightarrow T_R \quad T_i = C_i\text{<}\overline{p''}\text{>}$ By above sub-derivation

$fields(\Sigma[\ell]) = \overline{T'}\ \overline{f}$ By FDJ T-Store

$\forall T_k \in \{\overline{T'}\} \cup \{\overline{T}\} \quad T_k = C_k\text{<}\overline{p'''}\text{>}$

$\langle H[\theta], H[\ell], C'_k, Exp\rangle \in DE \quad C'_k <: C_k$ By above Df-Approx

Take $\forall i \in 1..\overline{T}.\ T_k = T_i \in \overline{T}, C_i = C_k$ and $C'_i = C'_k$. This proves (4).

Subcase $e_0 = e'_0$ that is $e = e'_0.m(\overline{e})$.

From IRC-RecvInvk

$\theta \vdash e'_0; S; H; K; L_I; L_E \rightsquigarrow_G e''_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash e'_0.m(\overline{e}); S; H; K; L_I; L_E \rightsquigarrow_G e''_0.m(\overline{e}); S'; H'; K'; L'_I; L'_E$ By IRC-RecvInvk

Take $e' = e''_0.m(\overline{e})$.

Subcase $e_0 = v$ that is $e = v.m(v_{1..i-1}, e_i, e_{i+1..n})$.

From IRC-ArgInvk:

$\Gamma, \Upsilon, DO, DD, DE \vdash_O e_i$ By sub-derivation of Df-Invk

$\theta \vdash e_i; S; H; K; L_I; L_E \rightsquigarrow_G e'_i; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K; L_I; L_E \rightsquigarrow_G$
$v.m(v_{1..i-1}, e'_i, e_{i+1..n}); S'; H'; K'; L'_I; L'_E$ By IRC-ArgInvk

Take $e' = v.m(v_{1..i-1}, e'_i, e_{i+1..n})$.

Case DF-CONTEXT : e = $\ell \triangleright e_0$. there are two subcases to consider, depending on whether e_0 is a value

Subcase e_0 is a value that is $e = \ell \triangleright v$.

From Ir-Context:

Then Ir-Context can apply. Take $e' = v$.

Subcase e_0 is a value that is $e = \ell \triangleright e'_0$.

From Irc-Context:

$\theta \vdash e_0; S; H; K; L_I; L_E \rightsquigarrow_G e'_0; S'; H'; K'; L'_I; L'_E$ By induction hypothesis

$\theta \vdash \ell \triangleright e_0; S; H; K; L_I; L_E \rightsquigarrow_G \ell \triangleright e'_0; S'; H'; K'; L'_I; L'_E$ By Irc-Context

Take $e' = \ell \triangleright e'_0$.

■

$$\frac{}{\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow^*_G e; S; H; K; L_I; L_E}\text{[DF-REFLEX]}$$

$$\frac{\begin{array}{c}\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow^*_G e''; S''; H''; K''; L''_I; L''_E \\ \theta \vdash e''; S''; H''; K''; L''_I; L''_E \rightsquigarrow_G e'; S'; H'; K'; L'_I; L'_E\end{array}}{\theta \vdash e; S; H; K; L_I; L_E \rightsquigarrow^*_G e'; S'; H'; K'; L'_I; L'_E}\text{[DF-TRANS]}$$

Figure 15: Reflexive, transitive closure of the instrumented evaluation relation

4.6 Theorem: Object Graph Soundness

If

$G = \langle DO, DD, DE \rangle$

$DO, DD, DE \vdash (CT, e_{root})$

$\forall e,\ \theta_0 \vdash e; \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E$

$\Sigma \vdash S$

then

$DO, DD, DE \vdash_{CT,H} \Sigma$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$

where $\rightsquigarrow^*_G$ relation is the reflexive and transitive closure of $\rightsquigarrow_G$ relation (Fig. 15). θ_0 is the location of the first object instantiated by e_{root}.

To prove the Object Graph Soundness theorem, we need to show:

(1) $DO, DD, DE \vdash_{CT,H} \Sigma$

(2) $(S, H, K, L_I, L_E) \sim (DO, DD, DE)$

Proof: The proof is by induction on the $\rightsquigarrow^*_G$ relation. There are two cases to consider: [1]

Case Df-Reflex :

Since $S = \emptyset$:

$(S, H, K, L_I, L_E) \sim G$

Immediately, from DF-SIGMA store constraint with $S = \emptyset$:

$DO, DD, DE \vdash_{CT,H} \Sigma$

Case Df-Trans :

By assumption:

$\theta_0 \vdash e; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e; S; H; K; L_I; L_E$

Since $S = \emptyset$:

$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \sim G$

By inversion of DF-TRANS:

[1] The soundness proof follows similar steps to the one of points-to analysis [1].

$\theta_0 \vdash e;\emptyset;\emptyset;\emptyset;\emptyset;\emptyset \rightsquigarrow^*_G e';S';H';K';L'_I;L'_E$
By induction hypothesis:
$(S';H';K';L'_I;L'_E) \sim G$
By inversion of Df-Trans:
$\theta_0 \vdash e';S';H';K';L'_I;L'_E \rightsquigarrow_G e;S;H;K;L_I;L_E$
By preservation:
$(S;H;K;L_I;L_E) \sim G$

By assumption:
$\theta_0 \vdash e;\emptyset;\emptyset;\emptyset;\emptyset;\emptyset \rightsquigarrow^*_G e;S;H;K;L_I;L_E$
Since $S = \emptyset$:
$(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset) \sim G$
By inversion of Df-Trans:
$\theta_0 \vdash e;\emptyset;\emptyset;\emptyset;\emptyset;\emptyset \rightsquigarrow^*_G e';S';H';K';L'_I;L'_E$
By induction hypothesis:
$DO, DD, DE \vdash_{CT,H'} \Sigma'$
By inversion of Df-Trans:
$\theta_0 \vdash e';S';H';K';L'_I;L'_E \rightsquigarrow_G e;S;H;K;L_I;L_E$
By preservation:
$DO, DD, DE \vdash_{CT,H} \Sigma$

■

4.6.1 Lemmas

To prove the Progress and Preservation theorems, we use the following lemmas. We intended to use the first four lemmas, (i.e. the import and export lemmas) in the Progress theorem proof. However, we complete the Progress proof without their use. We keep them for backward compatibility with the previous version of this report.

Df-Substitution Lemma.
If
$\Gamma \cup \{\overline{x} : \overline{T_f}\}, \Sigma, \theta \vdash e : T$
$\Gamma \cup \{\overline{x} : \overline{T_f}\}, \Upsilon, DO, DD, DE \vdash_O e$
$\Gamma, \Sigma, \theta \vdash \overline{v} : \overline{T_a}$ where $\overline{T_a} <: [\overline{v}/\overline{x}]\overline{T_f}$
then
$\Gamma, \Sigma, \theta \vdash [\overline{v}/\overline{x}]e : T'$ *for some* $T' <: [\overline{v}/\overline{x}]T$
$\Gamma, \Upsilon, DO, DD, DE \vdash_O [\overline{v}/\overline{x}]e$
Proof: By induction on the $\Gamma, \Upsilon, DO, DD, DE \vdash_O e$ relation. ■

Df-Weakening Lemma.
If
$\Gamma, \Upsilon, DO, DD, DE \vdash_O e$
then
$\Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, DO, DD, DE, \vdash_O e$

Proof: By induction on the $\Gamma, \Upsilon, DO, DD, DE \vdash_O e$ relation. ■

Df-Strengthening Lemma.

If

$\Gamma, \emptyset, DO, DD, DE \vdash_O \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v)$

$\forall i \in 1..|\overline{p}| \quad D_i = DD[(O, p_i)]$

$\Gamma, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, DO, DD, DE, \vdash_{O'} e'$

then

$\Gamma, \Upsilon, DO, DD, DE, \vdash_O e$

Proof: By induction on the $\Gamma, \Upsilon, DO, DD, DE \vdash_O e$ relation. ■

Df-Domains Lemma.

If

$\emptyset, \Sigma, \theta \vdash e : T$

$\Sigma \vdash S$

$DO, DD, DE \vdash_{CT,H} \Sigma$

$\emptyset, \emptyset, DO, DD, DE \vdash_O\ new\ C\texttt{<}\overline{p}\texttt{>}(\overline{v})$

$(S, H, K, L_I, L_E) \sim (DO, DD, DE)$

$DO, DD, DE \vdash_O ddomains(C, O_C)$

$\forall i \in 1..|\overline{p}| \quad D_i = DD[(O, p_i)]$

$O_C = \langle C\texttt{<}\overline{D}\texttt{>}\rangle \quad \{O_C\} \subseteq DO$

then

$\forall d_j \in domains(C\texttt{<}\overline{p}\texttt{>}) \quad D_j = DD[(O_C, d_j)]$

Proof: By induction on the $DO, DD, DE \vdash_O ddomains(C, O_C)$ relation. ■

Differences with earlier versions of this work. Our formalization refines the one in Rawshdeh's thesis [16]. Our analysis does not consider creational edges, it focuses on usage edges only. We completed the formalization of the static and dynamic semantics, and proved progress, preservation, and soundness. We defined the approximation relation, and used the additional maps L_I and L_E to track import and export edges.

Differences with points-to analysis. Our formalization is similar to the one for the points-to analysis [1, Section 3.2 and 3.3]. The two analyses create the same object-domain hierarchy, but the analysis in this paper shows additional edges that are missing from an OOG with points-to edges. The key differences in the formalization deal with generating the dataflow edges and the soundness proof. The previous work made a simplistic assumption about dataflow edges, namely that they can be approximated by reverting points-to edges, but the assumption turned out to be imprecise.

5 Addressed Challenges

We discuss how our analysis addressed each challenge in Section 2.

Object soundness. The OGraph shows a unique representative for each runtime object. We built the map H from runtime object to an OObject, and proved that each runtime object ℓ has a unique representative OObject (Section 4.6). We detailed the steps of building the map in the IR-NEW inference rule of dynamic semantics (Fig. 7).

Aliasing. Ownership domains ensures the aliasing invariant: two objects in different domains cannot alias. The analysis create separate OObjects for different parent ODomain. Although the code may show only one domain declaration `domain d` followed by a `new` expression `new`$C{<}d, ...{>}$, the analysis creates distinct ODomains depending on the context `OObject` O in which the domain declaration is analyzed. For two variables of the same type, that may refer to the same runtime object, the analysis shows a unique representative $C{<}\overline{D}{>}$ in DO. The condition $O_C = \langle C{<}\overline{D}{>}\rangle \quad \{O_C\} \subseteq DO$ in DF-NEW ensures this.

Edge soundness. If there is a runtime dataflow communication between two runtime objects, the OGraph must show an OEdge between the representatives of these objects. We built the maps L_I and L_E that map each pair of runtime objects (ℓ_1, ℓ_2) to an OEdge between the representative of ℓ_1 and ℓ_2 in the map H, and we provided the soundness proof (Section 4.6). We detailed the steps of building the maps in the IR-READ, IR-WRITE, IR-INVK inference rules of dynamic semantics(Fig. 7).

Summarization. In the presence of recursive types, the analysis stops creating new ODomains when the a domain declaration is analyzed using the same context. The analysis reuses an existing ODomain and creates a cycle with respect to ownership edges in the OGraph, which ensures a finite representation and a finite depth for the ROG (Fig. 5[DF-NEW, AUX-DOM]). In an extended example, we show the detailed steps of the analysis while analyzing a QuadTree recursive type (Section 4.2).

Hierarchy. The extracted OGraph contains a hierarchical organization of objects given by the ownership edges. A OObject has domains, which contain other objects to form its substructure, and so on. The OGraph provides abstraction by ownership hierarchy when it shows architecturally significant objects near the top of the hierarchy, and instances of data structures further down.

Precision. To avoid excessive merging of objects, the analysis relies on the aliasing precision provided by ownership domains. We map runtime objects in distinct runtime domains are mapped to distinct OObjects. The analysis also uses ownership domains to increase the precision of dataflow communication edges. While extracting OEdges, the analysis relies on Df-Lookup. For a variable of type $C'<\overline{p}>$, Df-Lookup returns only a subset of all the objects of class C' or its subclasses in DO. The analysis uses these results to determine the source and destination OObjects, and the class representing the edge label (Fig. 5[DF-LOOKUP]).

5.1 Analysis Invariants

- The analysis might create multiple ODomains for a single domain declaration $C::d$. As an exception, for top-level domain declarations, the analysis creates a single ODomain.
- The analysis might create multiple OObjects for a single expression `new`.
- The analysis might create a single OObject for multiple `new` expressions in the code, when the same class is instantiated multiple times. The `new` expressions might be scattered over multiple methods in different classes.
- For one field expression the analysis might create multiple import edges, depending on the result of *lookup* on the type of the receiver.
- The analysis does not create an export dataflow communication edge for a method invocation expression when the method has no arguments.
- The analysis does not create an import dataflow communication edge for a method invocation expression when the method returns `void`.
- In the presence of recursive types, the analysis reuses ODomains and creates a cycle in the ownership structure of the OGraph.

6 Evaluation

6.1 Running Example

As a running example, we use a small system that follows a Document-View architecture and implements the Observer design pattern. We refer to this example as Listeners, and we selected it because empirical data shows that developers often struggle while understanding listeners in object-oriented code [11].

Code structure. The code consists of several classes and uses various base classes, as is common in object-oriented code. The entry point of the application is the `Main` class, which instantiates `Model`, `PieChart`, and `BarChart`. `Model` extends the `Listener`, and contains the information displayed in the charts. `BarChart` and `PieChart` extend the `BaseChart` abstract class, which subsequently extends `Listener`. `BaseChart` and `Model` have a field of type `List` that represents a collection of objects of type `Listener` (Fig. 16). An instance of `Model` exchange messages of type `MsgMtoV` and `MsgVtoM` with instances of `BarChart` and `PieChart`.

Ownership domain annotations To express the Document-View architecture, `Main` defines two domains `DOC` and `VIEW`. The object `model` is in `DOC`, while the objects `barChart` and `pieChart` are in `VIEW`. Next, `Listener` has a declaration of a public domain `DATA` for messages. `Model` and `BaseChart` declare the private domain `OWNED` for collections of `Listener` objects registered for notification. `BarChart` and `PieChart` inherit the `OWNED` domain from `BaseChart`. As a public domain, `DATA` gives access to messages, while the collections in `OWNED` are strictly encapsulated.

Objects and domains. `DATA` is inherited from the abstract class `Listener` by all its subclasses. Therefore, the OOG shows three distinct domains: `barChart.DATA`, `pieChart.DATA`, and `model.DATA`. Similarly, the OOG shows three distinct domains: `barChart.OWNED`, `pieChart.OWNED`, and `model.OWNED`. Since each object is in exactly one domain, the OOG shows three distinct `List` objects, where two of them correspond to one `new` expression in `BaseChart`.

Dataflow Communication Edges. The dataflow edges in the OOG correspond to field reads, field writes, and method invocations. An export dataflow edge exists from `main` to `model` due to the method invocation `model.addListener(barChart)`, and the edge is labeled with the type of the argument, in this case `BarChart`. An export dataflow edge exists from `model` to `barChart` and to `pieChart` due to the method invocation `l.update(mTOv)`. In the code the type of the receiver `l` is `Listener`; however, the OOG shows only the objects in `VIEW` as destinations. The OOG also shows two export dataflow edges corresponding to the field write `listeners.value = l`. The source is `model`, the destination is `listeners1`, while the labels are `BarChart` and `PieChart`, respectively. The import dataflow edges from `listeners1` to `model` correspond to the field read `listeners.value` in the `Model` class. Similarly to the export dataflow edges the edge labels are `BarChart` and `PieChart`.

According to our definition, the analysis does not create an export edge if the method has no argument. The reason for this decision is that the analysis should create the same edges for a field read expression and for a method invocation that simply returns an alias to a field. Also, the analysis does not create import edges if the return type of the method is `void`. That is why, the analysis does not create an edge for the method invocation `model.notifyObservers()`. An illustration of the analysis on the Listeners example is available in the companion technical report [20].

Our static analysis extracts a hierarchical object graph that distinguishes between different instances of `Listener`, and depicts the dataflow communication between objects. The object graph conveys architectural abstraction by organizing objects hierarchically with the architecturally significant objects such as `BarChart`, `PieChart` and `Model` at a higher level of the hierarchy, and low-level objects such as collections and messages at a lower level (Fig. 17(a)).

Discussion. The resulting object graph makes visually obvious the dataflow communication occurring in the program. For example, the object graph shows two dataflow edges labeled `MsgMtoV` from `model` to `barChart` and `pieChart`, and two edges labeled `MsgVtoM`, from `barChart` and `pieChart` to `model`. Indeed, such communication is common in a Document-View architecture, followed by this example. An object graph with points-to edges does not show this communication since `model` does not have a field of type `BarChart` or `PieChart`. Although the objects representing messages appear in the object graph, namely `mTOv:MsgMtoV` and `vTOm:MsgVtoM`, they have no incoming or outgoing dataflow edges. The graph still shows the classes of the messages as edge labels, which makes visually obvious the transient relations between `model`, `barChart`, and `pieChart` (the highlighted

```
class Main<OWNER> {
  public domain DOC, VIEW;
  BarChart<VIEW, DOC> barChart = new BarChart();
  PieChart<VIEW, DOC> pieChart = new PieChart();
  Model<DOC, VIEW> pieChart = new PieChart();
  void run(){
   model.addListener(barChart); //(main $\xrightarrow{BarChart}$ model)
   model.notifyObservers(); //no dataflow
  }
}
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {}
class PieChart<OWNER, M> extends BaseChart<OWNER, M> {}
class BaseChart<OWNER, M> extends Listener<OWNER> {
  domain OWNED;
  List<OWNED, Listener<M>> listeners = new List();
}
abstract class Listener<OWNER> {
  public domain DATA;
  abstract void update(Msg<DATA> msg);
}
class Model<OWNER, V> extends Listener<OWNER> {
  domain OWNED;
  List<OWNED, Listener<V>> listeners = new List();
  void addListener(Listener<V> l) {
//(model $\xrightarrow{BarChart}$ listeners1, model $\xrightarrow{PieChart}$ listeners1)
   listeners.value = l;
  }
  void notifyObservers() {
   MsgMtoV<DATA> mTOv = new MsgMtoV();
//(listeners1 $\xrightarrow{BarChart}$ model, listeners1 $\xrightarrow{PieChart}$ model)
   Listener<V> l = listeners.value;
//(model $\xrightarrow{MsgMtoV}$ barChart, model $\xrightarrow{MsgMtoV}$ pieChart)
   l.update(mTOv);
  }
}
class List<OWNER, T<ELTS>> {//generic type T
  T<ELTS> value; // ELTS is a domain parameter
}
```

Figure 16: Listeners code fragments. The full code is in the companion technical report [20].

edges in Fig. 17).

Furthermore, the object graph shows three distinct objects of type `List`, which contain objects of the abstract type `Listener`. The dataflow edges to the `List` objects result from the analysis of the method invocations `listeners.add(l)` inside the methods `addListener(Listener l)`, in `BaseChart` and `Model`, respectively. From just reading these statements, it is not clear "what kind" of objects are added to each collection. But the object graph makes visually obvious (as edge labels) that in `BaseChart`, the reference `l` represents `Model` objects, while in `Model`, `l` represents `BarChart`

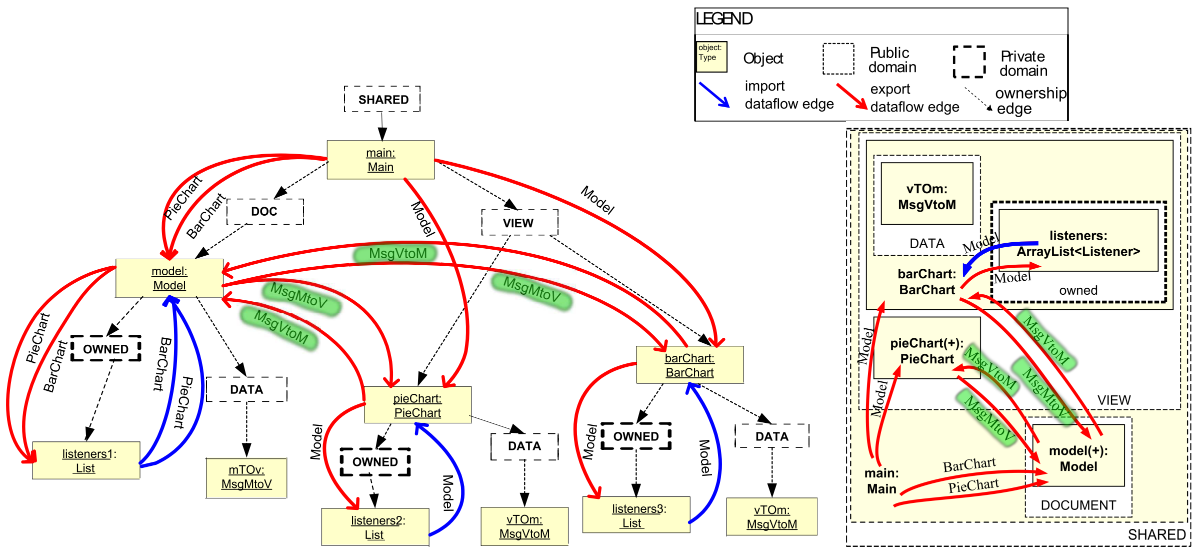

Figure 17: Dataflow communication for Listeners. Interesting edges are highlighted.

or `PieChart` objects. These labels are more precise than the base class `Listener`, the declared type of `l` in the code.

Collapsed OOG. Having a hierarchical representation allows expanding or collapsing the substructure of selected objects to control the level of visual detail. For example, only the substructure of `barChart` is visible, while the substructures of `pieChart` and `model` are collapsed (Fig. 17(b)). A (+) symbol indicates than an object has a collapsed substructure. While collapsing, the visualization also lifts the parent-child dataflow edges to the nearest visible ancestor, which makes the graph less cluttered. That is, if there is an edge between low-level objects that are elided, the edge is recursively lifted to the nearest visible ancestor. The nested box visualization allows us to collapse an object's substructure and reduce the number of objects at the top level [1, Section 3.4]. We also show the underlying representation because we explain how to extract it.

6.2 Notation.

We use the following notation: `obj.DOM` refers to either a public or a private domain `DOM` inside object `obj`. We effectively treat a domain as a field of an object, e.g., `main.DOC`; `obj1.DOM.obj2` refers to the object `obj2` inside the domain `DOM` of `obj1`, e.g., `main.DOC.model`; $C::d$ refers to a domain d qualified by the class C that declares it. The first domain parameter corresponds to the owning domain, and we call it `OWNER`. We use capital letters for domain names to distinguish them from other program identifiers.

The analysis calls the *analyze* method for every expression e, with the bindings $p_i \mapsto D_i$, and the context O:

$$analyze(e, [..., p_i \mapsto D_i, ...], O)$$

When it encounters inheritance, *analyze* recursively analyzes the base class of the current class (in that case, we do not show the parameter e).

A boxed statement represents the analysis step performed while analyzing the preceding statement. When a boxed statement precedes a class declaration, it includes the mapping of formal domain parameters to actual domains, and the context OObject O. Consecutive boxed statements correspond to consecutive analysis steps.

For example, when the analysis encounters a method invocation expression, it calls *lookup* multiple times to find the type of the receiver expression, the type of method arguments, and the return type. Next, it creates dataflow edges, and continues in the body of m. For brevity, we include only the most interesting *lookup* calls.

6.3 Worked Example.

Our analysis starts with the developer selecting the root type, in this case, the `Main` class (Fig. 18). The analysis creates the OObject (O0) for the `main` object allocation. Then, it analyzes `Main` in the context of `main`. Before analyzing `Main`, the analysis maps all formal domain parameters, if any, to their corresponding ODomains in the OGraph. In this case, the analysis maps `MAIN`::`OWNER` to the global domain `::SHARED`. The analysis also tracks the context OObject O.

The analysis continues inside `Main` and finds that the first statement is a domain declaration. In response, it creates two ODomains: `DOC` and `VIEW`. Next, for the object allocation statement of the `barChart` object, the analysis creates an OObject (O1), and proceeds to analyze the class `BarChart`, mapping O to `main.VIEW.barChart`. It also maps the domain parameter `BarChart`::`OWNER` to `main.VIEW`, and `BarChart`::`M` to `main.DOC`.

Next, the analysis covers `BarChart` and its base class `BaseChart` in the context of the OObject `barChart` (Fig. 19). The analysis proceeds into the base class `BaseChart` map-

```
Main<SHARED> main = new Main();
OObject(main, Main<SHARED>)  (O0)
analyze ( new Main<OWNER>(), [Main::OWNER ↦ SHARED], O ↦ main )

[Main::OWNER↦ ::SHARED], O↦ main
class Main<OWNER> {
  public domain DOC, VIEW;
  ODomain(main.DOC, Main::DOC)  (D1)
  ODomain(main.VIEW, Main::VIEW)  (D2)

  BarChart<VIEW, DOC> barChart = new BarChart();
  OObject(main.VIEW.barChart, BarChart<main.VIEW, main.DOC>)  (O1)
  analyze(new BarChart<OWNER, M>(), [BarChart::OWNER ↦ main.VIEW, BarChart::M ↦ main.DOC], O ↦
  main.VIEW.barChart)
  // continue to Fig. 19

  PieChart<VIEW, DOC> pieChart = new PieChart();
  OObject(main.VIEW.pieChart, PieChart<main.VIEW, main.DOC>)  (O2)
  analyze(new PieChart<OWNER, M>(), [PieChart::OWNER ↦ main.VIEW, PieChart::M ↦ main.DOC], O ↦
  main.VIEW.pieChart)
  // The analysis is similar to barChart, omitted for brevity

  Model<DOC, VIEW> model = new Model();
  OObject(main.DOC.model, Model<main.DOC, main.VIEW>)  (O3)
  analyze(new Model<OWNER, V>(), [Model::OWNER ↦ main.DOC, Model::V ↦ main.VIEW],O ↦
  main.DOC.model)
   // continue to Fig. 20
    ...
```

Figure 18: Abstractly interpreting the program, starting with the root class `Main`.

ping `BaseChart::OWNER` to `main.VIEW`, and `BaseChart::M` to `main.DOC` while the context O remains unchanged. Inside `BaseChart`, the analysis encounters two domain declarations: `domain OWNED` and `public domain DATA`. As a result, it creates a private and a public ODomains for `barChart`. Next statement is an instantiation of the class `List`, the analysis creates the OObject `main.VIEW.barChart.OWNED.listeners` (O4) inside the `barChart.OWNED` domain. Since `List` has no domain declarations or `new` statements, the analysis backtracks to `BaseChart`, and then further on to `Main`.

The analysis of class `PieChart`, its base class `BaseChart`, and its `List` is similar to `BarChart`, so we omitted it for brevity.

```
[BarChart::OWNER ↦ main.VIEW, BarChart::M ↦ main.DOC], O ↦ main.VIEW.barChart
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  analyze([BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦ main.DOC], O ↦ main.VIEW.barChart)

  public void update(Msg<DATA> msg) {...}
}

[BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦ main.DOC], O ↦ main.VIEW.barChart
class BaseChart<OWNER, M> extends Listener<OWNER> {
  domain OWNED;
  ODomain(main.VIEW.barChart.OWNED, BaseChart::OWNED)  (D3)

  public domain DATA;
  ODomain(main.VIEW.barChart.DATA, BaseChart::DATA)  (D4)

  List<OWNED, Listener<M>> listeners = new List();
  OObject(main.VIEW.barChart.OWNED.listeners, List<main.VIEW.barChart.OWNED,
  Listener<main.DOC>)                                                          (O4)
  analyze(new List<OWNER, ELTS>(), [List::OWNER ↦ main.VIEW.barChart.OWNED, List::ELTS
  ↦ main.DOC], O ↦ main.VIEW.barChart.OWNED.listeners)
  ...
}

[List::OWNER ↦ main.VIEW.barChart.OWNED, List::ELTS ↦ main.DOC], O
↦main.VIEW.barChart.OWNED.listeners
T = Listener  //generic type
class List<OWNER, T<ELTS>> {
  T<ELTS> value; // ELTS is a domain parameter for list elements
  ...
}
```

Figure 19: Abstractly interpreting the program (continued): BarChart, BaseChart and List.

Back in Main (Fig. 18), the analysis creates the OObject main.DOC.model (O3) corresponding to the instantiation of the class Model inside the ODomain DOC. Then it proceeds to analyze Model in the context of main.DOC.model (Fig. 20). The analysis maps the domain parameter Model::OWNER to main.DOC, and Model::V to main.VIEW. Similar to BaseChart the analysis creates a private and a public ODomain OWNED and DATA, and an OObject main.DOC.model.OWNED.listeners. Note that the analysis distinguishes between the listeners object owned by model, and the one owned by barChart although they are both instances of List.

Next, the analysis encounters the method invocation main.run() (Fig. 21). In this case, O correspond to main. Inside run(), the analysis encounters model.addListener(barChart), and it

```
[ Model::OWNER ↦ main.DOC, Model::V ↦ main.VIEW], O ↦ main.DOC.model
class Model<OWNER, V> extends Listener<OWNER> {
  domain OWNED;
  ODomain(main.DOC.model.OWNED, Model::OWNED)  (D9)

  public domain DATA;
  ODomain(main.DOC.model.DATA, Model::DATA)  (D10)

  List<OWNED, Listener<V>> listeners = new List();
  OObject(main.DOC.model.OWNED.listeners, List<main.DOC.model.OWNED,
  Listener<main.VIEW>>)                                                  (O6)
  analyze(new List<OWNER, ELTS>(), [List::ELTS ↦ main.VIEW, List::OWNER ↦
  main.DOC.model.OWNED], O ↦ main.DOC.model.OWNED.listeners)
  ...
} [List::OWNER ↦ main.DOC.model.OWNED, List::ELTS ↦ main.VIEW], O
  ↦main.DOC.model.OWNED.listeners
 T = Listener  //generic type
 class List<OWNER, T<ELTS>> {

  T<ELTS> value; // ELTS is a domain parameter for list elements
  ...
}
```

Figure 20: Abstractly interpreting the program (continued): Model and List.

changes the context to main.DOC.model. This method invocation introduces an export edge (E1) from main to model because main exports an object of type BarChart to model as the argument of addListener(Listener). For edge label, the analysis calls *lookup* and finds one OObject of type BarChart<main.VIEW, main.DOC>, main.VIEW.barChart.

Inside addListener(Listener) of Model, the analysis encounters listeners.add(l). A first *lookup* call returns the OObject listeners of type List<main.DOC.model.OWNED, Listener<main.VIEW>>, i.e., the object of type List of the ODomain model.OWNED, which is a collection of elements of type Listener, and each element of the collection is of the ODomain main.VIEW (Fig. 22). A second *lookup* returns the OObject barChart and, the analysis adds an OEdge (E2) between model and listeners labeled using BarChart. At this point the analysis backtracks to Main (Fig. 21).

Similar to the previous analyzed statement, the analysis of the method invocation model.addListener(pieChart) creates two edges labeled PieChart: from main to model (E4),

```
main.run();
analyze(main.run(), [Main::OWNER ↦ SHARED], O ↦ main)

public class Main<OWNER> {
  ...
  public void run() {

    model.addListener(barChart);
    analyze(model.addListener(barChart), [Model::OWNER ↦ main.DOC, Model::V ↦
    main.VIEW], O ↦ main.DOC.model
    OObject(main.DOC.model, Model<main.DOC, main.VIEW>) ∈ lookup(Model<main.DOC,
    main.VIEW>)
    OEdge(main, main.DOC.model, BarChart, Exp) (E1)
    // continue to Fig. 22

    model.addListener(pieChart);
    // The analysis is similar to model.addListener(barChart)
    // Omitted for brevity
    OEdge(main, main.DOC.model, PieChart, Exp) (E4)

    barChart.addListener(model);
    analyze(barChart.addListener(model), [BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦
    main.DOC], O ↦ main.VIEW.barChart)
    OObject(main.VIEW.barchart, BarChart<main.VIEW, main.DOC>) ∈
    lookup(BarChart<main.VIEW, main.DOC>)
    OEdge(main, main.VIEW.barChart, Model, Exp) (E7)
    // continue to Fig. 23

    pieChart.addListener(model);
    // The analysis is similar to barChart.addListener(model)
    // Omitted for brevity
    OEdge(main, main.VIEW.pieChart, Model, Exp) (E9)

    model.notifyObservers();
    analyze(model.notifyObservers(), [Model::OWNER ↦ main.DOC, Model::V ↦ main.VIEW], O
    ↦ main.DOC.model)
    // continue to Fig. 24

    barChart.notifyObservers();
    analyze(barChart.notifyObservers(), [BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦
    main.DOC], O ↦ main.VIEW.barChart)
    // continue to Fig. 25

    pieChart.notifyObservers();
    // The analysis is similar to barChart.notifyObservers()
    // Omitted for brevity
  }
}
```

Figure 21: Abstractly interpreting the program, class `Main`.

```
[Model::OWNER ↦ main.DOC, Model::V ↦ main.VIEW], O ↦ main.DOC.model
class Model<OWNER,V> extends Listener<OWNER> {
  ...
  l:BarChart<main.VIEW, main.DOC>
  public void addListener(Listener<V> l) {
    listeners.add(l);
    analyze(listeners.add(l),[List::OWNER ↦ main.DOC.model.OWNED, List::ELTS ↦
    main.VIEW], O ↦ main.DOC.model.OWNED.listeners)
    OObject(main.DOC.model.OWNED.listeners, List<main.DOC.model.OWNED,
    Listener<main.VIEW>>) ∈ lookup(List<main.DOC.model.OWNED, Listener<main.VIEW>>)
    OEdge(main.DOC.model, main.DOC.model.OWNED.listeners, BarChart, Exp)(E2)
  }
} [List::OWNER ↦ main.DOC.model.OWNED, List::ELTS ↦ main.VIEW], O
  ↦main.DOC.model.OWNED.listeners
 T = Listener //generic type
 class List<OWNER, T<ELTS>> {
    T<ELTS> value; // ELTS is a domain parameter for list elements
    public void add(T<ELTS> value) {...}
    ...
}
```

Figure 22: Abstractly interpreting the program (continued): `Model` addListener method.

```
[BarChart::OWNER ↦ main.VIEW, BarChart::M ↦ main.DOC], O ↦ main.VIEW.barChart
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  analyze([BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦ main.DOC], O ↦
  main.VIEW.barChart)

  public void update(Msg<DATA> msg) {...}
}

[BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦ main.DOC], O ↦ main.VIEW.barChart
class BaseChart<OWNER, M> extends Listener<OWNER> {
  ...
  l :  Model<main.DOC, main.VIEW>
  public void addListener(Listener<M> l) {
    listeners.value = l; // field write - export dataflow communication
    OObject(main.VIEW.barChart.OWNED.listeners, List<main.VIEW.barChart.OWNED,
    Listener<main.DOC>>) ∈ lookup(List<main.VIEW.barChart.OWNED, Listener<main.DOC>>)
    OEdge(main.VIEW.barChart, main.VIEW.barChart.OWNED.listeners, Model, Exp)(E8)
  }
}
```

Figure 23: Abstractly interpreting the program (continued): `BaseChart` addListener method.

```
[Model::OWNER ↦ main.DOC, Model::V ↦ main.VIEW], O ↦ main.DOC.model
class Model<OWNER,V> extends Listener<OWNER> {
  ...
  public void notifyObservers() {
    MsgMtoV<DATA> mTOv = new MsgMtoV();
    OObject(main.DOC.model.DATA.mTOv, MsgMtoV<main.DOC.model.DATA>) (O7)
    analyze(new MsgMtoV<OWNER>(), [MsgMtoV::OWNER ↦ main.DOC.model.DATA], O ↦
    main.DOC.model.DATA.mTOv)

    Listener<V> l = listeners.value; //field read - import dataflow communication
    OObject(main.DOC.model.OWNED.listeners, List<main.DOC.model.OWNED,
    Listener<main.VIEW>>) ∈ lookup(List<main.DOC.model.OWNED, Listener<main.VIEW>>)
    OObject(main.VIEW.barChart, BarChart<main.VIEW, main.DOC>) ∈
    lookup(Listener<main.VIEW>)
    OObject(main.VIEW.pieChart, PieChart<main.VIEW, main.DOC>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.DOC.model.OWNED.listeners, main.DOC.model, BarChart, Imp) (E11)
    OEdge(main.DOC.model.OWNED.listeners, main.DOC.model, PieChart, Imp) (E12)

    l.update(mTOv);
    analyze(l.update(vTOm),[BarChart::OWNER ↦ main.VIEW, BarChart::M ↦ main.DOC], O
    ↦ main.VIEW.barChart)
    OObject(main.VIEW.barChart, BarChart<main.VIEW, main.DOC>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.DOC.model, main.VIEW.barChart, MsgMtoV, Exp)  (E13)

    analyze(l.update(vTOm),[PieChart::OWNER ↦ main.VIEW, PieChart::M ↦ main.DOC], O
    ↦ main.VIEW.pieChart)
    OObject(main.VIEW.pieChart, PieChart<main.VIEW, main.DOC>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.DOC.model, main.VIEW.pieChart, MsgMtoV, Exp)  (E14)
    }
  }
}
```

Figure 24: Abstractly interpreting the program (continued): `Model` notifyObservers method.

from `model` to `listeners` (E5).

For `barChart.addListener(model)` and `pieChart.addListener(model)`, the analysis creates two OEdges labeled with `Model`: from `main` to `barChart` (E7), and from `main` to `pieChart` (E9). In both cases, the analysis encounters statement `listeners.value = l` in the `addListener` method of the class `BaseChart`. In response to this field write statement, the analysis

```
[BarChart::OWNER ↦ main.VIEW, BarChart::M ↦ main.DOC], O ↦ main.VIEW.barChart
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  analyze([BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦ main.DOC], O ↦
  main.VIEW.barChart)

  public void update(Msg<DATA> msg) {...}
} [BaseChart::OWNER ↦ main.VIEW, BaseChart::M ↦ main.DOC], O ↦ main.VIEW.barChart
class BaseChart<OWNER, M> extends Listener<OWNER> {
  ...
  public void notifyObservers() {
    MsgVtoM<DATA> vTOm = new MsgVtoM();
    OObject(main.VIEW.barChart.DATA.vTOm, MsgVtoM<main.VIEW.barChart.DATA>) (O8)
    analyze(new MsgVtoM<OWNER>(), [MsgVtoM::OWNER ↦ main.VIEW.barChart.DATA], O ↦
    main.VIEW.barChart.DATA.vTOm)

    Listener<M> l = listeners.getFirst(); //generates import dataflow communication
    analyze(listeners.getFirst(), [List::OWNER ↦ main.VIEW.barChartl.OWNED, List::ELTS
    ↦ main.DOC], O ↦ main.VIEW.barChart.OWNED.listeners)
    OObject(main.VIEW.barChart.OWNED.listeners, List<main.VIEW.barChart.OWNED,
    Listener<main.DOC>>) ∈ lookup(List<main.VIEW.barChart.OWNED, Listener<main.DOC>>)
    OObject(main.DOC.model, Model<main.DOC, main.VIEW>) ∈ lookup(Listener<main.DOC>)
    OEdge(main.VIEW.barChart.OWNED.listeners, main.VIEW.barChart, Model, Imp) (E15)

    l.update(vTOm);
    analyze(l.update(vTOm),[Model::OWNER ↦ main.DOC, Model::V ↦ main.VIEW], O ↦
    main.DOC.model)
    OObject(main.DOC.model, Model<main.DOC, main.VIEW>) ∈ lookup(Listener<main.DOC>)
    OEdge(main.VIEW.barChart, main.DOC.model, MsgVtoM, Exp)  (E16)
  }
}

[List::OWNER ↦ main.VIEW.barChart.OWNED, List::ELTS ↦ main.DOC], O
↦main.VIEW.barChart.OWNED.listeners
 T = Listener  //generic type
 class List<OWNER, T<ELTS>> {
    T<ELTS> value; // ELTS is a domain parameter for list elements
    public T<ELTS> getFirst() {  return value;  }
}
```

Figure 25: Abstractly interpreting the program (continued): BaseChart notifyObservers method.

calls *lookup* with the parameter `List<main.VIEW.barChart.OWNED, Listener<main.DOC>>`, and `List<main.VIEW.pieChart.OWNED, Listener<main.DOC>>`, respectively. The resulting OObjects are the two `listeners` owned by `barChart` and `pieChart`, respectively, and in each case, the analysis creates an OEdge labeled with `Model` (i.e., the actual type of `l`): from `barChart` to `listeners`

(E8), and from `pieChart` to its owned `listeners` (E10) (Fig. 23).

Back in `run()`, the analysis processes three method invocations `notifyObservers()` with different receivers (Fig. 21). Since the method has no parameters, the analysis does not include additional `OEdge` from `main` to the receivers. The analysis continues in the `notifyObservers()` methods of `Model`, `BarChart`, and `PieChart`.

While analyzing `notifyObservers()` of `Model` in the context of `model` (O), the analysis encounter the first statement, a class instantiation, and it creates a new OObject (O7) `mTOv` in the public domain `DATA` of `model` (Fig. 24).

The second statement contains the field read expression `listener.value`. In response, the analysis calls *lookup* twice. First *lookup* searches for object of type `List` in `model.OWNED`, while the second *lookup* searches for objects of type `Listener<main.VIEW>`. For the first call *lookup* returns `listeners`, while for the later call, *lookup* returns two OObjects: barChart and pieChart. As a result, the analysis creates two OEdges from `listeners` to `model`, one labeled with `BarChart` (E11), and the other labeled with `PieChart` (E12). Note that if the second *lookup* would not have been performed, the label would be `Listener`, which can be interpreted as any of the five classes extending `Listener`: `Model`, `BarChart`, `PieChart` or `BaseChart`. Therefore, by calling *lookup*, the analysis produces more accurate labels. Another observation is that the field read expression introduces an import edge, and O is the destination of the edge, while in the previous cases O was the source.

The third and last statement contains the method invocation `l.update(mTOv)`. The analysis calls again *lookup* searching for OObjects of type `Listener<main.VIEW>`. The result are the two OObjects `barChart` and `pieChart`. The analysis includes two OEdges labeled as MsgMtoV: from `model` to `barChart` (E13), and from `model` to `pieChart` (E14).

Since the `update` methods are empty, the analysis returns to `Main` and proceeds to `notifyObservers()` with `barChart` as receiver (O) (Fig. 25). The method is implemented in the superclass `BaseChart`, and the analysis performs the similar steps as previously discussed with two major differences. First, the second statement is the method invocation `listeners.getFirst()` instead of field read. The `getFirst()` method returns an alias to the `value` field. After a first *lookup* identifies `listeners` as the receiver of `getFirst()`, the analysis calls a second *lookup* searching for OObjects of type `Listener<main.DOC>`, which corresponds to the returned type of `getFirst()`. The result of the second *lookup* is `model`, and its class constitutes the edge label. As for field read, O is the destination of the OEdge from `listeners` to `barChart` (E15). Second, the analysis looks up the OObjects of a subtype of the local variable `l` in the method invocation `l.update(vTOm)`, and it finds only one OObject in the `main.DOC` domain (i.e., `model`). Therefore, it creates the OEdge from `barChart` to `model` labeled with `MsgVtoM` (E16).

The analysis concludes with the method invocation `pieChart.notifyObservers()`, and its corresponding implementation from `BaseChart`. The analysis performs the same steps previously discussed in the context of `pieChart` (O).

6.4 Graphical notation.

In the visualization of the OOG, we graphically distinguish between objects and domains by using a rectangle-shape to represent an object and a dashed rectangle-shape to represent a domain. We further distinguish between public and private domains using a thin dashed border for a public domain, and a bold dashed border for a private domain. In all cases, we label each rectangle with the name of the object or domain that corresponds to it. We represent the dataflow edges with a red solid arrow, and the ownership edges from domains to objects and vice-versa with a black arrow. We display as a root the `SHARED` domain, which constitutes the root node of the graph. The only object `SHARED` we display is the first object instantiated when the application starts, in our case `main`. On the dataflow edges, we also display the class of the object passed through the dataflow (Fig. 26). Developer can also opt for the hierarchical view (Fig. 27), hide the low level objects, and visualize only the dataflow communication between the high level objects (Fig. 28).

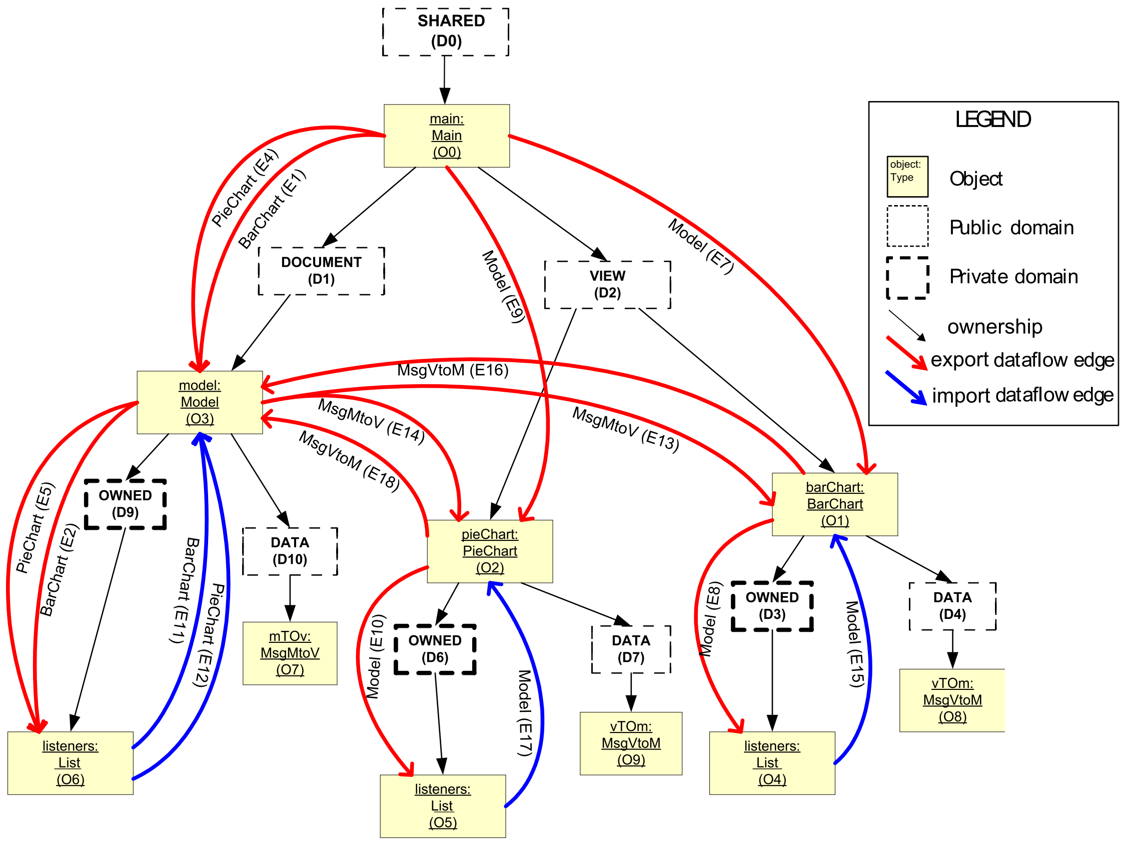

Figure 26: OOG extracted for Listeners

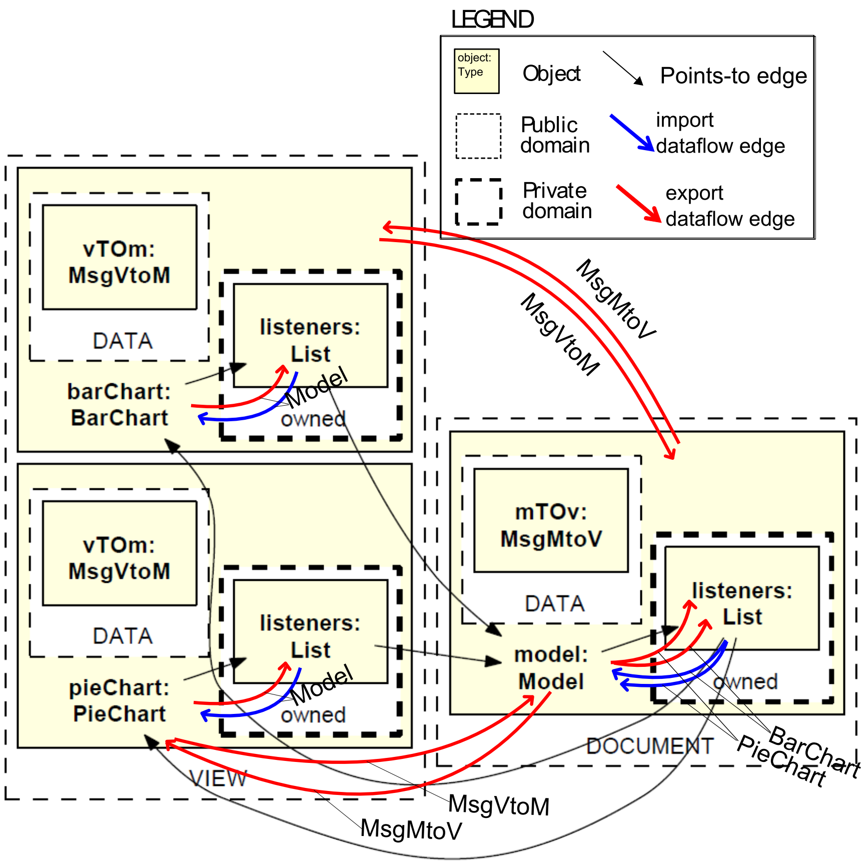

Figure 27: Display Graph for Listeners, with both points-to and dataflow edges.

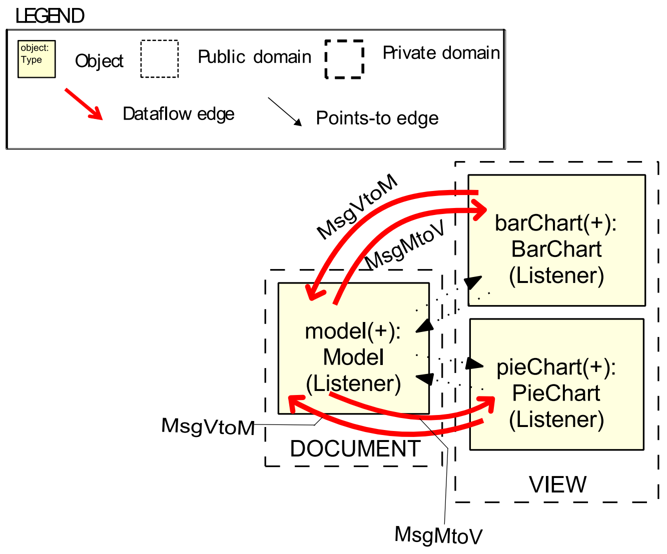

Figure 28: Display Graph for Listeners, after collapsing the sub-structures of top-level objects.

7 Related Work

Our work focuses on extracting one type of information, namely usage relationships, i.e., an object is using a method or a field of another object. We discuss how the related analyses address the challenges we listed above (Section 2).

Dataflow Communication. Andersen's static analysis extracted dataflow and points-to information from programs written in C [6], and was extended to object oriented code [13, 19] including Java [17]. These analyses determined the memory locations that may be modified by the execution of a statement. A dataflow edge means that *an object a owns a reference to an object c, and passes it to an object b*, or *an object a owns a reference to an object b, from which it receives a reference to an object c which only b knew before* [17]. However, the results of these analyses are flat graphs [17, 8] and the analyses did not attempt to be sound. In contrast, our analysis extracts hierarchical object graphs, with a relatively small number of top-level objects [1].

Sensitivity. An object graph analysis can be flow-, context-, or object-sensitive. A flow-sensitive analysis considers the order in which methods are called. A context-sensitive analysis analyzes the methods for each context under which a method is invoked. Object-sensitive analyses for points-to and dataflow edges addressed the aliasing and precision challenges [19, 13]. However, the analysis might not scale for a large number of references. Such an analysis worked well for on-demand based approaches which refined the references analyzed [18]. Seeking a tradeoff between soundness and precision, our analysis considers ownership domains as contexts and distinguishes objects of the same type but in different domains. That is, our analysis is *domain-sensitive*, and object- and flow-insensitive.

Dynamic analyses. Object graphs were extracted by analyzing heap snapshots [14, 15], and execution traces [12]. Lienhard analyzed execution traces and extracted an Object Flow Graph (OFG) in which edges represent objects, and nodes represent code structures: classes, and groups of classes [12]. OFG analysis addressed aliasing challenge, and linked objects to field read, field write, and method invocation expressions in the code, the same expressions used by our analysis. Since one class corresponds to one OFG node, an OFG is unable to show the communication between different instances of the same class and does not meet the soundness challenge. One advantage of dynamic analysis is that it does not require annotations. However, it can only infer a strict, owner-as-dominator hierarchy, which is limited in representing some design idioms [3]. Ownership domains support both strict encapsulation and logical containment (through public domains) and thus can express arbitrary design intent without restricting accessibility. In addition, a dynamic analysis requires extensive graph summarization to obtain an abstracted object graph [15, 7].

Annotation-based static analyses. Lam and Rinard [9] proposed a type system and a static analysis where by developer-specified annotations guide the static abstraction of an object model by merging objects based on tokens. Their approach supports a fixed set of statically declared global tokens, and their analysis shows a graph indicating which objects appear in which tokens. Since there is a statically fixed number of tokens, all of which are at the top level, an extracted object model is a top-level architecture that does not support hierarchical decomposition, thus limiting the scalability of the object model. In addition to their object model, Lam and Rinard extract models for "subsystem access", "call/return interaction", and "heap interaction", which is similar to the dataflow information our analysis extracts. From the challenges we listed in Section 2, they addressed aliasing, summarization in the presence of recursive types, and precision supported by tokens. Our approach extends Lam and Rinard's both to handle hierarchical object graphs and to support object-oriented language constructs such as inheritance.

8 Conclusion

We proposed a static analysis to extract a hierarchical object graph with dataflow communication edges that show usage relations between objects. We formalized the analysis following ownership domains and Featherweight Domain Java, and proved its soundness. We evaluated our analysis on an extended example and showed that the dataflow edges extracted by our analysis are similar to the ones drawn by developers who are reasoning about security.

Acknowledgements

The authors thank Suhib Rawshdeh for his contributions to an earlier version of this work.

References

[1] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, CMU, 2010.

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

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

[4] N. Ammar. Evaluation of the Usefulness of Diagrams of the Run-Time Structure for Coding Activities. Master's thesis, WSU, 2011.

[5] N. Ammar and M. Abi-Antoun. Evaluation of Global Hierarchical Object Graphs for Coding Activities: a Controlled Experiment. Under review at ECOOP, 2012.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Visual Lang. and Comp.*, 13(3), 2002.

[8] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.

[9] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.

[10] T. LaToza and B. Myers. Hard-to-answer questions about code. In *PLATEAU*, 2010.

[11] S. Lee, G. C. Murphy, T. Fritz, and M. Allen. How Can Diagramming Tools Help Support Programming Activities? In *VL/HCC*, 2008.

[12] A. Lienhard, S. Ducasse, and T. Grba. Taking an object-centric view on dynamic information with object flow analysis. *Journal of Computer Languages, Systems and Structures (COMLAN)*, 35:63–79, 2009.

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

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

[15] N. Mitchell, E. Schonberg, and G. Sevitsky. Making Sense of Large Heaps. In *ECOOP*, 2009.

[16] S. Rawshdeh and M. Abi-Antoun. A static analysis to extract dataflow edges from object-oriented programs with ownership domain annotations. Technical report, WSU, 2011.

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

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

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

[20] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Communication from Object-Oriented Code. Technical report, WSU, 2011. `www.cs.wayne.edu/~mabianto/tech_reports/VA11_TR.pdf`.

APPENDIX

A Source Code of Listeners Example

```

class Main<OWNER> {

  public domain DOC, VIEW;
  BarChart<VIEW, DOC> barChart = new BarChart();
  PieChart<VIEW, DOC> pieChart = new PieChart();
  Model<DOC, VIEW> model = new Model();

  public void run() {
    model.addListener(barChart);
    model.addListener(pieChart);
    barChart.addListener(model);
    pieChart.addListener(model);

    model.notifyObservers();
    barChart.notifyObservers();
    pieChart.notifyObservers();
  }

  public static void main(String[]<SHARED[SHARED]> args){
    Main<SHARED> main = new Main();
    main.run();
  }
}

class BaseChart<OWNER, M> extends Listener<OWNER> {
  domain OWNED;
  List<OWNED, Listener<M>> listeners = new List();

  public void addListener(Listener<M> l) {
    listeners.value = l;
  }

  public void notifyObservers() {
    MsgVtoM<DATA> vTOm = new MsgVtoM();
    Listener<M> l = listeners.getFirst();
    l.update(vTOm);
  }
}

class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  public void update(Msg<DATA> msg) {...}
}

class PieChart<OWNER, M> extends BaseChart<OWNER, M> {
  public void update(Msg<DATA> msg) {...}
}



```

$$\frac{p = n.d \qquad \Gamma;\Sigma;\theta \vdash n : C\texttt{<}\overline{p'}\texttt{>} \qquad d \in \mathit{domains}(C\texttt{<}\overline{p'}\texttt{>})}{\Gamma;\Sigma;\theta \vdash \mathit{qual}(p) == C{::}d}\text{[QUAL-VAR]}$$

$$\frac{\Gamma;\Sigma;\theta \vdash \texttt{this} : C\texttt{<}\overline{p'}\texttt{>} \qquad \alpha \in \mathit{params}(C) \qquad p = \alpha}{\Gamma;\Sigma;\theta \vdash \mathit{qual}(p) = C{::}\alpha}\text{[QUAL-PARAM]}$$

$$\frac{}{\Gamma;\Sigma;\theta \vdash \mathit{qual}(\texttt{SHARED}) = {::}\texttt{SHARED}}\text{[QUAL-SHARED]}$$

Figure 29: Qualify domains rules.

```
//generic type T
class List<OWNER, T<ELTS>> {
  T<ELTS> value; // ELTS is a domain parameter for list elements
  public T<ELTS> getFirst() {  return value;  }
  ...
}

class Model<OWNER, V> extends Listener<OWNER> {
  domain OWNED;
  List<OWNED, Listener<V>> listeners = new List();

  public void addListener(Listener<V> l) {
    listeners.add(l);
  }

  public void notifyObservers() {
    MsgMtoV<DATA> mTOv = new MsgMtoV();
    Listener<V> l = listeners.value;
    l.update(mTOv);
    }
  }

}

abstract class Listener<OWNER> {
  public domain DATA;
  public abstract void update(Msg<DATA> msg);
}
```

B Auxiliary judgements

Figure 29 shows the definitions we use to qualify a domain p by the class C that declares it. In the context of Γ, Σ, and θ, QUAL-VAR qualifies $n.d$ as $C{::}d$. This judgement also applies to the case when n is `this` and $p = \texttt{this}.d$. QUAL-PARAM qualifies a formal domain parameter α as $C{::}\alpha$, where C is the class of `this`. Since no class declares the `SHARED` domain, QUAL-SHARED qualifies it as ::`SHARED`. We use these rules implicitly in the static and dynamic semantics to ensure that $(O, C{::}d) \mapsto D$ is in DD and for the lookup operations $D = DD[(O,p)]$.

Figure 30 shows the definitions of many auxiliary judgments used earlier in the semantics. These definitions are the auxiliary judgments from ownership domains [3]. The Aux-Public rule checks whether a domain is public. The next few rules define the *domains*, and *fields* functions

$$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha}, \overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma} \rightarrow \overline{\delta}\ \{\ \overline{D};\ \overline{L};\ \overline{F};\ K\ \overline{M};\ \}$$

$$\frac{(\texttt{public domain}\ d) \in \overline{D}}{public(d)}\ \textit{Aux-Public}$$

$$\frac{\overline{D} = \overline{\texttt{public}_{opt}\ \texttt{domain}\ d_C} \qquad domains(C'\texttt{<}\overline{p}\texttt{>}) = \overline{d'}}{domains(C\texttt{<}\overline{p}, \overline{p'}\texttt{>}) = \overline{this.d_C}, \overline{d'}}\ \textit{Aux-Domains}$$

$$\frac{}{domains(\texttt{Object<}\overline{\alpha_0}\texttt{>}) = \emptyset}\ \textit{Aux-Domains-Obj}$$

$$\frac{\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}}{params(C) = \overline{\alpha}}\ \textit{Aux-Params}$$

$$\frac{\overline{F} = \overline{T}\ \overline{f} \qquad fields(C'\texttt{<}\overline{p}\texttt{>}) = \overline{T'}\ \overline{f'}}{fields(C\texttt{<}\overline{p}, \overline{p'}\texttt{>}) = ([\overline{p}/\overline{\alpha}, \overline{p'}/\overline{\beta}]\ \overline{T}\ \overline{f}), \overline{T'}\ \overline{f'}}\ \textit{Aux-Fields}$$

$$\frac{}{fields(\texttt{Object<}\overline{\alpha_0}\texttt{>}) = \emptyset}\ \textit{Aux-Fields-Obj}$$

$$\frac{}{owner(C\texttt{<}\overline{p}\texttt{>}) = p_1}\ \textit{Aux-Owner}$$

$$\frac{(T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mtype(m, C\texttt{<}\overline{p}\texttt{>}) = [\overline{p}/\overline{\alpha}]\ \overline{T} \rightarrow T_R}\ \textit{Aux-MType1}$$

$$\frac{m\ is\ not\ defined\ in\ \overline{M}}{mtype(m, C\texttt{<}\overline{p}, \overline{p'}\texttt{>}) = mtype(m, C'\texttt{<}\overline{p}\texttt{>})}\ \textit{Aux-MType2}$$

$$\frac{(T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mbody(m, C\texttt{<}\overline{p}\texttt{>}) = [\overline{p}/\overline{\alpha}]\ (\overline{x},\ e)}\ \textit{Aux-MBody1}$$

$$\frac{m\ is\ not\ defined\ in\ \overline{M}}{mbody(m, C\texttt{<}\overline{p}, \overline{p'}\texttt{>}) = mbody(m, C'\texttt{<}\overline{p}\texttt{>})}\ \textit{Aux-MBody2}$$

$$\frac{(mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T'} \rightarrow T') \Longrightarrow (\overline{T} = \overline{T'} \wedge T = T')}{override(m, C\texttt{<}\overline{p}\texttt{>}, \overline{T} \rightarrow T)}\ \textit{Aux-Override}$$

Figure 30: Auxiliary Judgments. Source: [3].

by looking up the declarations in the class and adding them to the declarations in the base classes. The *owner* function just returns the first domain parameter (which represents the owning domain in our formal system).

The *mtype* function looks up the type of a method in the class; if the method is not present, it looks in the superclass instead. The *mbody* function looks up the body of a method in a similar way. Finally, the *override* function verifies that if a superclass defines method m, it has the same type as the definition of m in a subclass.