Inference and Checking of Object Immutability

Ana Milanova Yao Dong
Rensselaer Polytechnic Institute
110 8th Street, Troy NY
{milanova, dongy6}@cs.rpi.edu

ABSTRACT

Reference immutability guarantees that a reference is not used to modify the referenced object. It is well-understood and there are several scalable inference systems. Object immutability, a stronger immutability guarantee, guarantees that an object is not modified. Unfortunately, object immutability is not as well-understood; specifically, we are unaware of an inference system that infers object immutability across large Java programs and libraries.

It is tempting to use reference immutability to reason about object immutability. However, representation exposure and object initialization pose significant challenges.

In this paper we present a novel type system and a corresponding inference analysis. We leverage reference immutability to infer object immutability overcoming the challenges due to representation exposure and object initialization.

We have implemented our object immutability system for Java. Evaluation on the standard Dacapo benchmarks demonstrates precision and scalability. Nearly 40% of all static objects are inferred immutable. Analysis completes in under 2 minutes on all benchmarks.

CCS Concepts

•Software and its engineering → Object oriented languages;

Keywords

immutability, object immutability, reference immutability

1. INTRODUCTION

Immutability information benefits a wide variety of compiler optimizations and software engineering tasks, including redundant synchronization elimination, automatic test generation, refactoring, and program understanding [19].

There are several notions of immutability in object-oriented programming. *Reference immutability* guarantees that an immutable reference cannot be used to modify its referent object; however, there are normal references that can be used to modify that same object. *Object immutability* guarantees that an immutable object cannot be modified through any reference; however there are mutable objects of the same class. *Class immutability* guarantees that if a class is immutable, then every object of that class is immutable, e.g., String and Integer in Java.

Reference immutability is relatively well-understood and there are several inference systems that work on large Java programs, including Javarifier [20] and ReimInfer [17]. Object immutability is not well-understood, in the sense that there are no practical inference systems akin to Javarifier and ReimInfer (to the best of our knowledge). The main challenges are representation exposure and object initialization. Existing type systems for object immutability [5, 24, 12, 11] typically rely on *ownership* [6] to make their immutability guarantees, and ownership is notoriously difficult to reason about.

We present a new type system for object immutability, Oim, and an efficient inference analysis. Oim supports deep object immutability, i.e., it disallows mutation not only to the object but to all of its components, transitively. In addition, it supports delayed object initialization, i.e., it allows for initialization after the constructor call has completed. The key novelty is a combination of reference immutability and light-weight escape analysis. Next, we outline the challenges, approach, and contributions.

1.1 Challenges

It is tempting to try to use reference immutability to prove that the object created at y = new C(z), which we'll call o, is immutable. The reasoning proceeds as follows: if y, which holds the canonical reference to o, is immutable according to reference immutability, then o must be immutable too.

Unfortunately, this reasoning is flawed. First, y may not be the *only* reference to o. This is because implicit parameter this of constructor C may "escape", leading to outstanding references to o, other than y. For example, if C contains

```
public C(D p) { p.f = this; ... }
```

then after y = new C(z) completes, z.f and y both refer to o. A modification to o may occur through z.f rendering o mutable, even if y remains an immutable reference.

Second, there may be representation exposure at object creation. If C contains code

```
public C(D p) { this.f = p; ... }
```

which is often the case in constructors, then argument z references parts of o and modification of o may happen

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

DOI: http://dx.doi.org/10.1145/2972206.2972208

through z. Such modification renders o mutable (we are interested in deep immutability), and may happen even as the canonical reference y remains immutable. An additional challenge is that there may be outstanding mutable references to the object z refers to and proving z an immutable reference will not suffice.

Yet another challenge is object initialization. Reference immutability systems (e.g., [17]) break constructor calls y = new C(z) into a sequence of object creation followed by an explicit constructor call:

```
y = new C;
y.C(z);
```

Reference immutability systems treat constructor calls y.C(z) just like regular instance calls and propagate the mutability of implicit parameter this to the receiver y. Since constructors practically always modify their this parameter, since they perform object initialization, this is mutable. Reference immutability systems capture the "link" from y to this via subtyping constraint $q_{\mathsf{y}} <: q_{\mathsf{this}}$, which forces y to be mutable (here q_{y} and q_{this} are the reference immutability qualifiers associated with y and this). Therefore, y is inferred mutable, even though it may be a unique reference to o that is never modified beyond the constructor call.

Our solution builds upon the success of reference immutability. The key idea is simple: we *skip* constraint $q_{\mathsf{y}} <: q_{\mathsf{this}}$ thus *endorsing* modifications through this during object initialization. In the same time, we ensure safety and precision.

1.2 Contributions

We make the following contributions:

- Oim, a new type system for object immutability. Oim supports deep object immutability and delayed object initialization. The key novelty is a combination of reference immutability and light-weight escape analysis.
- Efficient inference system that requires no annotations.
- Empirical evaluation on the Dacapo benchmarks.

The rest of the paper is organized as follows. Section 2 gives an overview of our system. Section 3 presents a generalized flow type system, which is the foundation for reference immutability, escape analysis and object immutability. Section 4 presents the escape type systems and Section 5 presents the object immutability type system. Section 6 details our experiments. Section 7 discusses related work, and Section 8 concludes.

2. OVERVIEW

Section 2.1 introduces ReIm, a type system for reference immutability we developed in earlier work [17]. Section 2.2 and Section 2.3 outline how we build Oim on top of ReIm.

2.1 Reference Immutability

ReIm has two main immutability qualifiers: mutable and readonly.

- mutable: A mutable reference can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages.
- readonly: A readonly reference x cannot be used to mutate the referenced object nor anything it references. For example, all of the following are forbidden:
  - x.f = z
  - x.setField(z) where setField sets a field of its receiver
  - y = id(x); y.f = z where id is a function that returns its argument
  - x.f.g = z
  - y = x.f; y.g = z

mutable is a subtype of readonly. This means that a mutable reference can be assigned to a readonly one, but a readonly reference cannot be assigned to a mutable one. ReIm, similarly to other reference immutability systems, tracks flow of values through subtyping. For example, assignment x = y entails subtyping constraint $q_{\mathsf{y}} <: q_{\mathsf{x}}$. The subtyping constraint captures the flow dependence from y to x: y $\longrightarrow$ x. If x is mutable, due to, for example, x.f = z or z = x.f; z.g = 0;, subtyping constraint $q_{\mathsf{y}} <: q_{\mathsf{x}}$ entails that y is mutable as well. A method call entails subtyping constraints that track flow of values from actual arguments to formal parameters, and from return value to the left-hand-side of the call assignment. For example, call x = y.m(z) creates $q_{\mathsf{y}} <: q_{\mathsf{this}}$ and $q_{\mathsf{z}} <: q_{\mathsf{p}}$, which connect receiver y to implicit parameter this of m, and argument z to parameter p. If this or p is mutable, then y or z, respectively, become mutable. In addition, call x = y.m(z) creates constraint $q_{\mathsf{ret}} <: q_{\mathsf{x}}$. (This explanation of calls is simplified; in reality, ReIm creates constraints that treat calls context-sensitively.)

2.2 Endorse Qualifiers for Delayed Object Initialization

Object initialization is a thorny issue for object immutability type systems [24, 11, 19] as well as other type systems [10]. Clearly, even immutable objects must be mutated during their construction phase. The main issue is whether to restrict initialization to constructors, or to allow *delayed initialization*, i.e., initialization writes and calls, issued after the constructor has returned. Delayed initialization is necessary to handle *circular initialization*, a fairly common idiom. (Figure 1 shows circular initialization; we elaborate upon this example shortly.)

Another issue, specific to our approach, is that reference immutability propagates mutation of this in constructors (and delayed initializers) to the receiver as we described in Section 1.1.

We propose endorse qualifiers which enable delayed initialization and in the same time handle our reference-immutability-specific issue. An endorse-ed *statement* allows modification to happen to the receiver while the statement is in scope. endorse can be applied to (1) instance field/array writes and (2) instance calls.

Consider this example of array initialization (from [11]):

```
static int[] copy(int[] a) {
  immutable int[] r = new int[a.length];
  for (int i = 0; i < r.length; i++) {
      int v = a[i];
      endorse r[i] = v;
  }
  return r;
}
```

ReIm renders r **mutable** because of the write r[i] = v in line 5. When r[i] = v is endorsed, the write to r does not contribute mutation (in this case mutation is part of object initialization) and both reference r and the array object are determined immutable. Oim easily handles endorsed field writes. When a field write x.f = y is endorsed, it drops the standard ReIm requirement that $q_{\mathsf{x}} = \mathsf{mutable}$, i.e., that receiver x is mutable.

Handling of method calls is more involved yet still carried out with only a minor modification to ReIm. Consider the example below stylized from a popular multithreaded benchmark, SpecJBB:

```
1  public static JBBmain extends Thread {
2    static Company company;
3    public static void main() {
4       Company immutable c = new Company;   c
5       endorse c.Company();
6       endorse c.prepareForStart();
7       company = c;
8       for (...) {
9         JBBmain j = new JBBmain;
10        j.start();
11      }
12   }
13   public void run() { ... }
14 }
```

In SpecJBB, the Company object c is shared among multiple threads. Threads, spawned inside the loop at line 10, operate on c. Importantly, modification of c occurs only at lines 5-6; this includes creation of several objects that are part of the representation of c. The "simultaneous" multithreaded accesses to c, triggered by run, are readonly. Therefore, for all purposes c is immutable, and synchronization on c can be eliminated resulting in substantial performance improvement [9].

Above, the constructor call at lines 5 and the instance call at line 6 are endorse-ed. Thus, mutations to receiver c happening during the call to Company and during the call to prapareForStart do not render c mutable the way they do in ReIm. Intuitively, mutation during Company and prepareForStart is initialization mutation. Our system handles endorsed instance calls by *dropping the subtyping constraint* that links the receiver to this. For example, in line 5, the system drops the standard ReIm constraint $q_{\mathsf{c}} <: q_{\mathsf{this_{Company}}}$. This "cuts" propagation from $\mathsf{this_{Company}}$, which is mutable, to c.

Endorsement is restricted to field writes and method calls whose receiver is a *canonical reference*, i.e., a reference at the left-hand side of an object creation statement, e.g., x in x = new A. Additionally, endorsed calls cannot return references.

The Raw type qualifier of OIGJ [24] and the fresh(n) initialization block of Haack and Poll [11] have similar goal to endorse. They are designed to allow for mutation during initialization and delayed initialization. However, Raw applies to variables, while endorse applies to statements. fresh(n) has tighter parallels to endorse. It applies to a block of statements. For example, in Figure 1, the entire block at the end of the code snippet must be declared fresh. In contrast, our system naturally endorses writes and calls on canonical references. This simplifies inference and increases flexibility — our system allows for "parts" bob and alice to be created and initialized separately from couple, possibly in a different method. In addition, endorse is tightly coupled to reference immutability.

Like Raw and fresh, endorse "suspends" soundness — that is, immutability does not hold during the endorse-ed statement but is restored after the endorse-ed statement completes.

```
class Person {
  Person partner;
}
class Couple {
  Person husband;
  Person wife;
  Couple(noEsc Couple this, noOesc Person h, noOesc Person w) {
     this.husband = h;
     this.wife = w;
  }
}
Person immutable alice = new Person;
Person immutable bob = new Person;
endorse alice.partner = bob;
endorse bob.partner = alice;
Couple immutable couple = new Couple;
endorse couple.Couple(bob, alice);
```

Figure 1: Circular initialization.

2.3 Escape Qualifiers for Safe Endorsement

Unfortunately, simply dropping constraints $q_{\mathsf{y}} <: q_{\mathsf{this}}$ may be unsound. Suppose implicit parameter this escaped its enclosing method m. "Cutting" the link from y to this may fail to reflect modifications to y that happen after the call to m has returned. This would violate the property of endorsement which "endorses" only those modifications that happen during the call to m.

Another challenge is representation exposure at object construction. Figure 1 illustrates this case: the alice and bob objects, and references to them, do exist before the couple object; alice and bob become parts of the couple object during its construction. Disallowing representation exposure at construction (as other approaches do) would have rendered experiments with real applications impossible, because this is a very common idiom in practice. We must guarantee however, that when representation exposure at construction occurs, then all preexisting references to parts of the newly created object, remain immutable.

Let us return to the example from Section 1:

y = new C; $\boxed{o}$
endorse y.C(z);

Suppose that z becomes part of y, which refers to object o, just as alice and bob become part of couple in Figure 1. Once such a connection between argument z and receiver y exist, the mutability of object o depends not only on y but on z as well (recall that we are interested in deep object immutability). To account for this, our type system imposes a new subtyping constraint, $q_{\mathsf{y}} <: q_{\mathsf{z}}$, thus connecting y to argument z: $\mathsf{y} \longrightarrow \mathsf{z}$. This constraint entails that if z is mutable then y must be mutable as well.

An important observation is that when (1) formal parameter p of constructor C *escapes exclusively* through this of C *and* (2) p is not modified in C, we can *drop the standard ReIm constraint* $q_{\mathsf{z}} <: q_{\mathsf{p}}$ (which connects actual argument z to p). Dropping this constraint is necessary for precision; if it remains, over-approximation in ReIm (and other reference immutability systems) causes p to be mutable, which propagates to z due to $q_{\mathsf{z}} <: q_{\mathsf{p}}$ and subsequently to y due to $q_{\mathsf{y}} <: q_{\mathsf{z}}$. Forgoing $q_{\mathsf{z}} <: q_{\mathsf{p}}$ is good for precision, but poses a soundness issue when p flows to this (e.g., through this.f

$= \mathsf{p}$). This is because z may get modified through y, along a flow path such as $\mathsf{z} \longrightarrow \mathsf{p} \longrightarrow \mathsf{this.f} \longrightarrow \mathsf{y.f}$; "cutting" the link from z to p would violate soundness when y.f is modified after the call. Therefore, when z is assigned into a field of y, we add constraint $q_{\mathsf{z}} <: q_{\mathsf{y}}$ which entails that if y is mutable then z must be mutable as well. Constraints $q_{\mathsf{y}} <: q_{\mathsf{z}}$ and $q_{\mathsf{z}} <: q_{\mathsf{y}}$ imply $q_{\mathsf{y}} = q_{\mathsf{z}}$.

We make use of *escape qualifiers*. In Figure 1 (again, from Haack and Poll [11]), this of constructor Couple is qualified as noEsc, which guarantees that no part of this escapes. Parameter h on the other hand is not noEsc because it escapes to this through this.husband = h. We need one other kind of escape qualifiers, *other-escape qualifiers*. noOesc x guarantees that (1) if x escapes, it escapes exclusively through this, and (2) x is not modified during the call to its enclosing method. In Figure 1, parameters h and w are noOesc. The precise guarantees of noEsc and noOesc are given in Section 4. For simplicity, the presentation ignores static fields; our implementation handles them correctly.

In summary, light-weight escape analysis enables precise and safe handling of endorsed calls. Oim defines a rule for handling endorsed calls y.m(z), which, essentially, is the only change from ReIm. Oim drops constraint $q_{\mathsf{y}} <: q_{\mathsf{this}}$. It considers 3 cases for the parameter:

1. Formal parameter p is noEsc and noOesc. Oim imposes the standard constraint $q_{\mathsf{z}} <: q_{\mathsf{p}}$.

2. p is noOesc, i.e., it escapes exclusively through this and it is not modified. Oim imposes $q_{\mathsf{y}} = q_{\mathsf{z}}$ because z has become a part of y, and thus, the mutabilities of z and y are interdependent.

3. p escapes differently (e.g., to a formal parameter) or p escapes through this, or p is modified. Oim enforces both the subtyping constraint $q_{\mathsf{z}} <: q_{\mathsf{p}}$, to capture mutation of z through p, and $q_{\mathsf{y}} = q_{\mathsf{z}}$, to account that z may have become part of y.

Let us return to Figure 1. Parameters h and w of constructor Couple are noEsc and the system enforces constraints $q_{\mathsf{couple}} = q_{\mathsf{bob}}$ and $q_{\mathsf{couple}} = q_{\mathsf{alice}}$. Since all mutations are endorsed, variables couple, alice and bob remain immutable. Note that if there were mutation to, say, alice down the road, the above constraints would render couple mutable (justifiably) as well as bob mutable (unjustifiably). Our analysis reflects mutation of parts into mutation to the enclosing object, but also, it conservatively propagates this mutation to other parts of the enclosing object. In practice, it is rare that one part of an objects is mutable while the other parts remain immutable.

3. GENERALIZED FLOW TYPE SYSTEM

In this section, we describe a class of type systems, which we refer to as *flow type systems*. The type systems we consider in this paper, ReIm, Escape and oEscape, which reason about escape properties (detailed in Section 4) and Oim, the object immutability type system (Section 5) are all flow type systems. The purpose of this section is twofold: (1) it sets a framework that fits the above type systems, and (2) it generalizes previous work, most notably on reference immutability [17] and information flow [16]; using the framework, one can easily specify new flow systems (as we do with Escape and oEscape), as well as reason about soundness.

A flow system has two main type qualifiers, a *positive* qualifier and a *negative* one. In information flow terms, positive variables are *sources* and negative variables are *sinks*. A flow type system enforces subtyping constraints which essentially track flow of values in the program. The goal is to prevent flow from positive variables (sources) to negative ones (sinks).

In ReIm, readonly is the positive qualifier and mutable is the negative one. Variables that appear at field writes are mutable sinks; for example, x at x.f = y is a sink. Variables designated by the programmer as readonly are sources. The goal of ReIm is to prevent flow from readonly sources to mutable sinks.

Section 3.1 describes the dynamic semantics of flow. Section 3.2 describes the static semantics, including the flow type qualifiers, context sensitivity, typing rules and type inference. Finally, Section 3.3 defines the soundness framework.

After setting the framework, we proceed to define the flow type systems of interest — Escape and oEscape are described in Section 4, and Oim is described in Section 5.

3.1 Dynamic Semantics of Flow

The dynamic semantics builds *flow paths*, which represent flow dependences between variables. It is defined over a syntax in named form. For readability we simplify the description; for a much more formal description we refer the reader to our technical report [15]. We assume that local variables and heap objects have unique fresh identifier. (Since this is a dynamic semantics, there is no conflation per method or allocation site as with a static semantics.)

An assignment statement contributes a link as follows:

$$\mathsf{x} = \mathsf{y} \quad \Rightarrow \quad \mathsf{y} \longrightarrow \mathsf{x}$$

The new link, $\mathsf{y} \longrightarrow \mathsf{x}$, represents the flow from variable y to variable x. The semantics appends link $\mathsf{y} \longrightarrow \mathsf{x}$ to every flow path that ends at y.

A pair of field write x.f = y and field read $\mathsf{y}' = \mathsf{x}'.\mathsf{f}$, where x and x′ refer to the same object o and x.f = y was the last write to o.f, contribute a link as follows:

$$\mathsf{x.f} = \mathsf{y},\ \mathsf{y}' = \mathsf{x}'.\mathsf{f} \quad \Rightarrow \quad \mathsf{y} \longrightarrow \mathsf{y}'$$

That is, the pair creates a link from y to y′. We deliberately avoid heap objects in both our dynamic and static semantics. We are interested in flow paths from one variable to another. Thus, demonstrating that a static semantics safely captures structure-transmitted links $\mathsf{y} \longrightarrow \mathsf{y}'$, is enough.

A method call (method entry) creates the expected links from actual arguments to formal parameters:

$$\mathsf{x} = \mathsf{y.m(z)} \quad \Rightarrow \quad \mathsf{y} \longrightarrow \mathsf{this} \quad \mathsf{z} \longrightarrow \mathsf{p}$$

A method return (method exit) creates the link from return value to left-hand-side of the call assignment:

$$\mathsf{x} = \mathsf{y.m(z)} \quad \Rightarrow \quad \mathsf{ret} \longrightarrow \mathsf{x}$$

Since we are interested in flow paths from one variable to another, we eschew object creation statements.

For our purposes we need one more rule:

$$\mathsf{y}' = \mathsf{x}'.\mathsf{f} \quad \Rightarrow \quad \mathsf{x}' \longrightarrow \mathsf{y}'$$

This link denotes a flow dependence from the receiver x′ of the field read to y′. We elaborate upon this shortly.

To illustrate the concept of the flow path, consider the code in Figure 2. For readability, code throughout this section

```
class DateCell {
  Date date;
  Date(DateCell this, Date date) { this.date = date; }
  Date getDate(DateCell this) { return this.date; }
  void cellSetHours(DateCell this) {
    Date md = this.getDate();
    md.setHours(1);              // md is mutated
  }
  int cellGetHours(DateCell this) {
    Date rd = this.getDate();
    int hour = rd.getHours();  // rd is readonly
    return hour;
  }
}
...
main() {
  Date d = new Date;
  d.Date(0); // Jan 1, 1970, 00:00:00
  DateCell dc = new DateCell;
  dc.DateCell(d);
  ... = dc.cellGetHours();
  dc.cellSetHours();
}
```

Figure 2: DateCell example.

makes the formal parameter this explicit. There is a flow path from this of cellSetHours to md in cellSetHours:

$$\mathsf{this}_{\mathsf{cellSetHours}} \longrightarrow \mathsf{this}_{\mathsf{getDate}} \longrightarrow \mathsf{ret}_{\mathsf{getDate}} \longrightarrow \mathsf{md}$$

Note that the link $\mathsf{this}_{\mathsf{getDate}} \longrightarrow \mathsf{ret}_{\mathsf{getDate}}$ is due to the field read statement in getDate.

The goal of a flow type system is to disallow all flow paths that start at a source and end at a sink. In the above flow path, variable $\mathsf{this}_{\mathsf{cellSetHours}}$ cannot be readonly (if it were readonly, it would be a source), because there is a flow path from $\mathsf{this}_{\mathsf{cellSetHours}}$ to the mutable md.

An example of a structure-transmitted link is:

$$\mathsf{d}_{\mathsf{main}} \longrightarrow \mathsf{date}_{\mathsf{DateCell}} \longrightarrow \mathsf{ret}_{\mathsf{getDate}} \longrightarrow \mathsf{rd}_{\mathsf{cellGetHours}}$$

The link from the date parameter in constructor DateCell to ret in getDate is a structure-transmitted link.

3.2 Static Semantics of Flow

We now describe the static semantics of flow.

3.2.1 Flow Type Qualifiers

A flow system has three type qualifiers: negative, positive, and poly.

- negative: The negative qualifier designates sinks. In ReIm, mutable is the negative qualifier.
- positive: The positive qualifier designates sources. Flow from a positive variable x to a negative variable y is forbidden. In ReIm, readonly is the positive qualifier.
- poly: This qualifier expresses polymorphism over flow qualifiers. The poly qualifier is interpreted (instantiated) as positive in positive contexts and as negative in negative contexts. In ReIm, the poly qualifier denotes that a reference is readonly within its enclosing method, but it may be mutable outside of the enclosing method, depending on the context. We elaborate on the poly qualifier in Section 3.2.2.

The subtyping relation between the qualifiers is

$$\mathsf{negative} <: \mathsf{poly} <: \mathsf{positive}$$

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2. Thus, in ReIm, it is allowed to assign a mutable reference to a poly or readonly one, but it is not allowed to assign a readonly reference to a poly or mutable one.

3.2.2 Context Sensitivity

There are two dimensions of context sensitivity: (1) in call-transmitted dependences and (2) in structure-transmitted dependences [21]. Context-sensitive handling of call-transmitted dependences ensures, roughly speaking, that in a = id(b); ... c = id(d); the analysis properly records flow dependences from b to a and from d to c, but does not record spurious dependences from b to c and from d to a. Context-sensitive handling of structure-transmitted dependences ensures, again roughly speaking, that in x.f = a; ... b = y.f; the analysis records flow dependence from a to b when x.f and y.f are aliases, but avoids spurious dependences when x.f and y.f are not aliases.

We first illustrate handling of call-transmitted dependences. Return to the code in Figure 2. Implicit parameter this of cellGetHours flows to rd due to the call to getDate(), which establishes flow path

$$\mathsf{this}_{\mathsf{cellGetHours}} \rightarrow \mathsf{this}_{\mathsf{getDate}} \rightarrow \mathsf{ret}_{\mathsf{getDate}} \rightarrow \mathsf{rd}$$

On the other hand, this of cellSetHours flows to md, again due to a call to getDate(), which establishes the analogous flow path

$$\mathsf{this}_{\mathsf{cellSetHours}} \rightarrow \mathsf{this}'_{\mathsf{getDate}} \rightarrow \mathsf{ret}'_{\mathsf{getDate}} \rightarrow \mathsf{md}$$

(The $'$ symbol designates different run-time instances of this and ret in getDate.) Clearly, getDate() must be polymorphic as it may appear in positive contexts as well as in negative ones (i.e., on a path to a sink).

The flow systems use polymorphic qualifier poly and *viewpoint adaptation* to handle calls context-sensitively. (The role of viewpoint adaptation is to interpret poly according to the right context.) In the above example, this and the return type of getDate are poly:

```
poly Date getDate(poly DateCell this) {
  return this.date;
}
```

This means that getDate is interpreted as negative in negative context and as positive in positive context.

Consider ReIm. In Figure 2, this of cellGetHours is readonly. Having this of cellGetHours readonly is advantageous because then cellGetHours can be called on any argument.

The return value of method DateCell.getDate is used in a mutable (i.e., negative) context in cellSetHours and is used in a readonly (i.e., positive) context in cellGetHours. A context-insensitive type system would give the return type of getDate one specific type, which would have to be mutable. This would cause rd to be mutable, and then this of cellGetHours would have to be mutable as well (if this.date is of type mutable, this means that the current object was modified using this, which forces this to become mutable). This violates our goal that this of cellGetHours is readonly.

The polymorphic qualifier poly expresses context sensitivity. Viewpoint adaptation instantiates poly to mutable in the context of cellSetHours, and to readonly in the context of cellGetHours. The call this.getDate on line 6 returns a mutable Date, and the call this.getDate on line 10 returns a readonly Date. As a result, the mutability of md propagates only to this of cellSetHours; it does not propagate to this of cellGetHours, which remains readonly.

Next, we illustrate handling of structure-transmitted dependences. Viewpoint adaptation instantiates the type of a poly field f to the type of the receiver. If the receiver x is negative, then x.f is negative. If the receiver x is positive, then x.f is positive. If the receiver x is poly, then x.f is poly.

Reps has shown that handling both call-transmitted and structure-transmitted dependences precisely is undecidable [21]. Therefore, like all static analyses, our system must approximate. We handle call-transmitted dependences precisely, but handle structure-transmitted dependences approximately. The approximation becomes clear in Section 3.2.3.

Viewpoint adaptation.

Viewpoint adaptation is a concept from Universe Types [8, 7]. Viewpoint adaptation of a type q' from the point of view of another type q, results in the adapted type q''. This is written as $q \rhd q' = q''$.

Below, we explain viewpoint adaptation for flow systems. In flow systems, $\rhd$ adapts formal parameters and return types at method calls from the point of view of the *call-site context* at the call; it adapts fields at field accesses from the point of view of the *receiver* of the access.

We define $\rhd$ as:

$$\begin{array}{lcl} \_ \rhd \mathsf{negative} & = & \mathsf{negative} \\ \_ \rhd \mathsf{positive} & = & \mathsf{positive} \\ q \rhd \mathsf{poly} & = & q \end{array}$$

The underscore denotes a "don't care" value. Qualifiers negative and positive do not depend on the viewpoint. poly depends on the viewpoint — it becomes that viewpoint. As we mention already, the role of viewpoint adaptation is to instantiate (interpret) the poly qualifier, according to the context.

For a method call $i\colon \mathsf{x} = \mathsf{y.m(z)}$, viewpoint adaptation $q_i \rhd q$ adapts q, the declared qualifier of a formal parameter/return of m, from the point of view of q_i, which is the call-site qualifier at i. If a formal parameter/return is readonly or mutable, its adapted type remains the same regardless of q_i. However, if the formal parameter/return is poly, the adapted type depends on q_i — it becomes q_i (i.e., the poly type is the polymorphic type, and it is instantiated to q_i).

For a field access, viewpoint adaptation $q \rhd q_\mathsf{f}$ adapts the declared field qualifier q_f from the point of view of the receiver qualifier q. In field access y.f where the field f is positive, the type of y.f is positive. In field access y.g where the field g is poly, y.g takes the type of y. If y is positive, then y.g must be positive as well. For example, in ReIm, having y.g readonly disallows modifications of y's object through y.g. If y is poly then y.g is poly as well, propagating the context-dependency. As part of our approximation, we disallow negative fields for all flow systems. In ReIm this means that fields cannot be mutable.

3.2.3 Typing Rules

Figure 3 contains the core of the flow type system defined over the same normal-form syntax we used in Section 3.1. Rule (TASSIGN) is straightforward. It requires that the left-hand-side is a supertype of the right-hand-side thus tracking flow dependence $\mathsf{y} \longrightarrow \mathsf{x}$. Rules (TWRITE) and (TREAD), together, handle structure-transmitted dependences. Figure 3 shows (TWRITE) and (TREAD) for ReIm, which requires that q_x at (TWRITE) is mutable (x is a sink). Other type systems in this paper have slightly different rules. It is required of a type system to prove that (TWRITE) and (TREAD) correctly handle structure-transmitted dependences.

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad q_\mathsf{y} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\text{TASSIGN})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \\ q_\mathsf{x} = \mathsf{mutable} \qquad q_\mathsf{y} <: q_\mathsf{x} \rhd q_\mathsf{f}\end{array}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\text{TWRITE})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \\ q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\text{TREAD})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \Gamma(\mathsf{z}) = q_\mathsf{z} \\ \mathit{typeof}(\mathsf{m}) = q_\mathsf{this}, q_\mathsf{p} \rightarrow q_\mathsf{ret} \\ q_\mathsf{y} <: q_i \rhd q_\mathsf{this} \qquad q_\mathsf{z} <: q_i \rhd q_\mathsf{p} \qquad q_i \rhd q_\mathsf{ret} <: q_\mathsf{x}\end{array}}{\Gamma \vdash i : \mathsf{x} = \mathsf{y.m(z)}}\ (\text{TCALL})$$

Figure 3: Typing rules. Function *typeof* retrieves the declared immutability qualifiers of fields and methods. Γ is a type environment that maps variables to their immutability qualifiers.

Rule (TCALL) handles call-transmitted dependences. Function *typeof* retrieves the type of m. q_this is the type of parameter this, q_p is the type of the formal parameter, and q_ret is the type of the return. Rule (TCALL) requires $q_i \rhd q_\mathsf{ret} <: q_\mathsf{x}$. This constraint disallows the return value of m from being positive when there is a call to m, $\mathsf{x} = \mathsf{y.m(z)}$, where left-hand-side x is negative. Only if the left-hand-sides of all call assignments to m are positive, can the return type of m be positive; otherwise, it is poly.

In addition, the rule requires $q_\mathsf{y} <: q_i \rhd q_\mathsf{this}$. When q_this is positive or negative, its adapted value is the same. In ReIm, when q_this is mutable (e.g., due to this.f = 0 in m),

$$q_\mathsf{y} <: q_\mathsf{x} \rhd q_\mathsf{this} \quad \text{becomes} \quad q_\mathsf{y} <: \mathsf{mutable}$$

which disallows q_y from being anything but mutable, as expected. The most interesting case arises when q_this is poly. There could be a dependence between this and ret such as

$$\mathsf{X\ m()\ \{\ ...\ z = this.f; w = z.g; return\ w; ...\ \}}$$

which allows the this object to be modified in caller context, after m's return. Well-formedness guarantees that whenever there is a flow path (dependence) between this and ret, as in the above example, the following constraint holds:

$$q_\mathsf{this} <: q_\mathsf{ret}$$

It is a theorem that for every q_i if $q_\mathsf{this} <: q_\mathsf{ret}$ then

$$q_i \rhd q_\mathsf{this} <: q_i \rhd q_\mathsf{ret}$$

At (TCALL), the type system requires $q_{\mathsf{y}} <: q_i \rhd q_{\mathsf{this}}$ and $q_i \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}$. These two constraints combined with the above constraint imply

$$q_{\mathsf{y}} <: q_{\mathsf{x}}$$

In other words, the type system transmits the dependence between this and ret into the appropriate dependence between y and x. If x is negative, then y is negative, as expected.

Our inference tool, ReImInfer, types the DateCell class from Section 3.2.2 as follows:

```
1  class DateCell {
2    poly Date date;
3    poly Date getDate(poly DateCell this) {
4      return this.date;
5    }
6    void cellSetHours(mutable DateCell this) {
7      mutable Date md = this.getDate();
8      md.setHours(1);
9    }
10   void cellGetHours(readonly DateCell this) {
11     readonly Date rd = this.getDate();
12     int hour = rd.getHours();
13   }
14 }
```

Field date is poly because it is mutated indirectly in method cellSetHours. Because the type of this of getDate is poly, it is instantiated to mutable in cellSetHours as follows:

$$q_7 \rhd q_{\mathsf{this}} \quad = \quad \mathsf{mutable} \rhd \mathsf{poly} \quad = \quad \mathsf{mutable}$$

It is instantiated to readonly in cellGetHours:

$$q_{11} \rhd q_{\mathsf{this}} \quad = \quad \mathsf{readonly} \rhd \mathsf{poly} \quad = \quad \mathsf{readonly}$$

Thus, this of cellGetHours can be typed readonly.

Method overriding is handled by the standard constraints for function subtyping. If m' overrides m we require

$$\mathit{typeof}(\mathsf{m}') <: \mathit{typeof}(\mathsf{m})$$

and thus,

$$(q_{\mathsf{this}_{\mathsf{m}'}}, q_{p_{\mathsf{m}'}} \rightarrow q_{\mathsf{ret}_{\mathsf{m}'}}) \;\; <: \;\; (q_{\mathsf{this}_{\mathsf{m}}}, q_{p_{\mathsf{m}}} \rightarrow q_{\mathsf{ret}_{\mathsf{m}}})$$

This entails $q_{\mathsf{this}_{\mathsf{m}}} <: q_{\mathsf{this}_{\mathsf{m}'}}$, $q_{p_{\mathsf{m}}} <: q_{p_{\mathsf{m}'}}$, and $q_{\mathsf{ret}_{\mathsf{m}'}} <: q_{\mathsf{ret}_{\mathsf{m}}}$.

3.2.4 Type Inference

Type inference operates on mappings from keys to values S. The keys in the mapping are (1) local variables and parameters, including parameters this, (2) field names, (3) method returns and (4) call-site contexts q_i. The values in the mapping are *sets* of type qualifiers. For instance, $S(\mathsf{x}) = \{\mathsf{poly}, \mathsf{negative}\}$ means the type of reference x can be poly or negative. For the rest of the paper we use "reference" and "variable" to refer to all kinds of keys: local variables, fields, method returns, and call-site qualifiers.

S is initialized as follows. Programmer-annotated references, if any, are initialized to the singleton set that contains the programmer-provided type. Method returns are initialized $S(\mathsf{ret}) = \{\mathsf{positive}, \mathsf{poly}\}$ for each method m. Fields are initialized $S(\mathsf{f}) = \{\mathsf{positive}, \mathsf{poly}\}$. (Recall that we forbid a negatively-qualified field.) All other references are initialized to the maximal set of qualifiers, i.e., $S(\mathsf{x}) = \{\mathsf{positive}, \mathsf{poly}, \mathsf{negative}\}$.

The inference analysis iterates over the typing rules removing infeasible types from $S(\mathsf{x})$ until it reaches a fixpoint. For example, consider x = y.f and corresponding rule (TREAD). Suppose that $S(\mathsf{x}) = \{\mathsf{poly}\}$, $S(\mathsf{y}) = \{\mathsf{positive}, \mathsf{poly}, \mathsf{negative}\}$ and $S(\mathsf{f}) = \{\mathsf{positive}, \mathsf{poly}\}$. Processing (TREAD) removes positive from $S(\mathsf{y})$ because there does not exist $q_{\mathsf{f}} \in S(\mathsf{f})$ and $q_{\mathsf{x}} \in S(\mathsf{x})$ that satisfies $\mathsf{positive} \rhd q_{\mathsf{f}} \;<:\; q_{\mathsf{x}}$. Similarly, it removes positive from $S(\mathsf{f})$ because there does not exist $q_{\mathsf{y}} \in S(\mathsf{y})$ and $q_{\mathsf{x}} \in S(\mathsf{x})$ that satisfies $q_{\mathsf{y}} \rhd \mathsf{positive} \;<:\; q_{\mathsf{x}}$. After processing (TREAD), S' is as follows: $S'(\mathsf{x}) = \{\mathsf{poly}\}$, $S'(\mathsf{y}) = \{\mathsf{poly}, \mathsf{negative}\}$, and $S'(\mathsf{f}) = \{\mathsf{poly}\}$.

Note that the result of fixpoint iteration is a mapping from references to *sets* of qualifiers. The actual mapping from references to qualifiers is derived as follows: for each reference x we pick the maximal element of $S(\mathsf{x})$ according to the subtyping relation, which we also call the "preference ranking" when we use it for this purpose. There are two important theorems: (1) picking the maximal element *always type checks* (for the type systems in this paper), and (2) the resulting typing *maximizes* the number of positive references. Inference is $O(n^2)$ where n is the size of the program. For details, see [17]. [1]

3.3 Soundness

Soundness connects the static semantics to the dynamic semantics.

Runtime interpretation.

This connection relies on the notion of runtime interpretation [15]. Intuitively, a static type qualifier receives a runtime interpretation, depending on the runtime context. The runtime interpretation of the static type of x is defined as follows:

$$RI(q_{\mathsf{x}}) = q_0 \rhd ... q_{k-1} \rhd q_k \rhd q_{\mathsf{x}}$$

Here q_k is the call-site context at the call that pushed x's frame, F_k, on the stack, q_{k-1} is the call-site context at the call that pushed F_k's parent frame on the stack, and so on. q_0 is the call-site context in main that triggered the stack configuration leading to F_k. Consider Figure 2. The runtime interpretation of ret of getDate in the context that starts at callsite 22 is as follows:

$$RI(q_{\mathsf{ret}}) = q_{22} \rhd q_6 \rhd q_{\mathsf{ret}}$$

q_6 is mutable because md is mutable, and q_{ret} is poly. Therefore $RI(q_{\mathsf{ret}}) = \mathsf{mutable}$. Note that the RI of a mutable or readonly qualifier remain the same. The RI is interesting when interpreting a poly qualifier; it interprets it as mutable in mutable contexts and as readonly in readonly ones. The runtime interpretation is either mutable or readonly; we guarantee this by forbidding poly call-site contexts in main.

Well-formedness.

Let configuration C consist of all (dynamic) flow paths. C is well-formed if for every flow path $\mathsf{x} \longrightarrow^* \mathsf{y} \in C$, $RI(q_{\mathsf{x}}) <: RI(q_{\mathsf{y}})$. Therefore, if $q_{\mathsf{y}} = \mathsf{negative}$, then q_{x} must be poly or negative establishing the fact that there is no flow path from a positive x to a negative y.

The preservation theorem is standard:

THEOREM 3.1. *If C is well-formed and $C \xrightarrow{\mathsf{s}} C'$, then C' is well-formed.*

[1] ReImInfer [17] uses the left-hand-side of the call assignment q_{x} in lieu of call-site qualifier q_i. The theorems still hold.

It is an onus on an individual type system to establish preservation. This is done by structural induction with s ranging over assignment, field write, field read, method call and return. All type systems capture flow paths, however they define sinks differently, and may propagate flow dependences differently.

We sketch the proof for field read in ReIm, perhaps the most interesting case. We must show that when we have x.f = a ... b = y.f such that x.f is the update read at y.f, then $RI(q_{a}) <: RI(q_{b})$ (since these establish link a $\longrightarrow$ b). If $RI(q_{b}) =$ readonly the subtyping trivially holds. Conversely, if $RI(q_{b}) =$ mutable, then q_{b} must be mutable or poly, which combined with the typing rule for (TREAD) $q_{y} \rhd q_{f} <: q_{b}$ entails that $q_{f} =$ poly (we forbid mutable fields). Rule (TWRITE) enforces that $q_{x} =$ mutable, which combined with $q_{f} =$ poly and the other part of the rule, $q_{a} <: q_{x} \rhd q_{f}$, entails that $q_{a} =$ mutable. Therefore $RI(q_{a}) <: RI(q_{b})$ holds.

4. ESCAPE TYPE SYSTEMS

This section presents two flow type systems that track *escapes* of references. In particular, we are interested in escapes of parameters. The first type system, Escape, tracks all escapes. For example, if there is this.f = p, or q = p.f; r.g = q, then p escapes. The second system, oEscape, tracks what we call *other* escapes. Roughly, it allows for escapes through this. In the above example, p does not escape in this.f = p; however, it does escape in q = p.f; r.g = q. Type inference for both Escape and oEscape proceeds as described in Section 3.2.4.

4.1 Escape Type System

Let x be a local reference. In the terminology of Section 3, the Escape type system tracks flow paths from x to z, where z is a sink y.f = z. If x is marked as source, Escape disallows such paths.

There are 3 escape qualifiers.

- esc: This is the negative qualifier. A reference x escapes if x is "responsible" for the creation of heap references to the object x references or to one of its components. For example, x in y.f = x escapes.
- noEsc: This is the positive qualifier. Thus, a noEsc reference cannot escape. For example, if x is noEsc all of the following are forbidden:
  - y.f = x
  - y.setField(x)
  - y = id(x); z.f = y
  - y.f.g = x
  - y = x.f; z.g = y
- poly: This qualifier expresses polymorphism over escape qualifiers. Consider the id function which takes a poly parameter and returns a poly result. Thus, x is noEsc in context y = id(x), where y does not escape, while z is esc in context w = id(z); v.f = w, where clearly it escapes through w. poly is instantiated to noEsc in the former context and to esc in the latter context.

Figure 4 shows rules (TWRITE) and (TREAD) for the Escape type system. All other rules are as in Figure 3. Rule (TWRITE) designates sinks, differently from ReIm. All other

$$\frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad q_{\mathsf{y}} = \mathsf{esc}}{\Gamma \vdash \mathsf{x.f = y}}\ \text{(TWRITE)} \qquad \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x = y.f}}\ \text{(TREAD)}$$

Figure 4: Typing rules for Escape. All remaining rules are as shown in Figure 3.

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x = y}}\ \text{(TASSIGN)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{this}) = q_{\mathsf{this}} \qquad \Gamma(\mathsf{p}) = q_{\mathsf{p}} \\ q_{\mathsf{y}} <: q_{\mathsf{this}} \qquad q_{\mathsf{this}} <: q_{\mathsf{p}}\end{array}}{\Gamma \vdash \mathsf{this.f = y}}\ \text{(TWRITETHIS)}$$

$$\frac{\begin{array}{c}\mathsf{y} \neq \mathsf{this} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \\ q_{\mathsf{y}} = q_{\mathsf{x}} = \mathsf{oEsc}\end{array}}{\Gamma \vdash \mathsf{x.f = y}}\ \text{(TWRITE)}$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x = y.f}}\ \text{(TREAD)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{this}) = q_{\mathsf{this}} \qquad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ \mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}_{\mathsf{m}}}, q_{\mathsf{p}} \rightarrow q_{\mathsf{ret}} \\ q_{\mathsf{this}} = q_i \rhd q_{\mathsf{this}_{\mathsf{m}}} \qquad q_{\mathsf{z}} <: q_i \rhd q_{\mathsf{p}} \\ q_i \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}\end{array}}{\Gamma \vdash i : \mathsf{x = this.m(z)}}\ \text{(TCALLTHIS)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ \mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_{\mathsf{p}} \rightarrow q_{\mathsf{ret}} \\ q_{\mathsf{y}} = \mathsf{oEsc} \qquad q_i \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}} \\ \mathit{escape}(\mathsf{p}) \Rightarrow q_{\mathsf{z}} = \mathsf{oEsc} \\ !\mathit{escape}(\mathsf{p}) \Rightarrow q_{\mathsf{z}} <: q_i \rhd q_{\mathsf{p}}\end{array}}{\Gamma \vdash i : \mathsf{x = y.m(z)}}\ \text{(TCALL)}$$

Figure 5: Typing rules for the oEscape type system. ***escape*(p) returns true if p is esc in Escape; it returns false otherwise.**

rules track flow as described in Section 3. Soundness is easy to argue, particularly because all field writes are sinks and therefore one need not worry about structure-transmitted dependences.

4.2 oEscape Type System

Let p be a parameter in method m. The oEscape type system has two goals. First, it tracks flow paths from p to a sink z, y.f = z where y refers to an object *other* than the receiver of m. Second, it tracks flow paths from p to z′, where z′ is a sink z′.f = ..., i.e., p is being mutated. Importantly, oEscape tracks paths during the call to m, i.e., while m's frame is active on the stack. We are interested in escapes and modifications that happen during the call to m, not ones that may happen after the call has returned.

Type qualifiers and typing rules.

The oEscape type system has the following 3 qualifiers:

- oEsc: This is the negative qualifier. A parameter p in method m is oEsc if p flows to an object other than the receiver of m, or if p is mutated during the call to m. For example, p is oEsc in p'.f = p. In another example, p is oEsc in p.f = ..., because of the mutation to p. Similarly, p is oEsc in this.f = p; y = this.f; y.g = z;, but not because of the assignment to this, but rather because of the mutation to y.
- noOesc: This is the positive qualifier. A noOesc reference x cannot escape to an object other than this and cannot be mutated. For example, all of the following are forbidden:
  - p'.f = p
  - p'.setField(p)
  - y = id(p); y.f = z

  However, this.f = p is legal.
- poly has the same semantics as Escape.

Figure 5 shows the typing rules for oEscape. Rules (TASSIGN) and (TCALLTHIS) are as in Figure 3. (We show all rules for completeness.) Rules (TASSIGN) and (TREAD) are straightforward; for example, at (TREAD), oEscape enforces constraint $q_y <: q_x$, which models link $y \longrightarrow x$.

Rule (TWRITE) marks both y and x as oEsc sinks. y is a sink because it flows to x.f, which may create a heap reference from an object other than this. x is a sink because it is being modified.

Rule (TCALL) conservatively marks receiver y as oEsc sink because m may modify its implicit parameter this and oEscape does not track such modification. If parameter p escapes according to the Escape type system, oEscape marks argument z as oEsc (the escape may happen because p flows to this as in this.f = p, or because p flows somewhere else). If p does not escape, oEscape creates the standard constraint $q_z <: q_i \rhd q_p$ which propagates modification of parameter p to argument z.

Consider rule (TCALLTHIS). Since the receiver context does not change, all it does is propagate dependences from actuals to formals and from return type to the left-hand-side of the call assignment. This rule captures the case when a constructor initializes fields with a call to its superclass constructor, super(p). A parameter may be noOesc in the context of one subclass, and oEsc in the context of another subclass.

Rule (TWRITETHIS) does not mark sinks. Instead it propagates dependence from y to this through constraint $q_y <: q_{this}$. (TWRITETHIS) creates one additional constraint, $q_{this} <: q_p$, where p is the formal parameter of the enclosing method; this constraint is needed to establish.

Soundness.

We now sketch the preservation argument. oEscape tracks flow paths from sources x to oEsc sinks y with the restriction that such flow path happens while x's frame is still active on the stack (x may flow to an oEsc sink y after x's frame has returned). This restriction is important when arguing preservation. This is unlike ReIm, where we are interested whether x is mutated after its enclosing method has returned.

The most interesting argument is at field read. To argue preservation we have to show that if there is x.f = a ... y.f = b such that x.f is the update, read out at y.f, then

$$RI(q_a) <: RI(q_b).$$

If $x \neq this$, then the statement trivially holds as (TWRITE) sets q_a to sink type oEsc. However, if x is this, then a is not explicitly set to oEsc. Since receiver o is created outside of the method, it must flow into the method through a parameter: we have either flow path $this \longrightarrow^* y$ or $p \longrightarrow^* y$. The former entails $RI(q_{this}) <: RI(q_y)$ and the letter entails $RI(q_p) <: RI(q_y)$ (by the inductive hypothesis of preservation). Consider the latter case. (TWRITETHIS) entails $q_a <: q_{this}$ and $q_{this} <: q_p$, which gives us

$$RI(q_a) <: RI(q_{this}) <: RI(q_p)$$

which combined with $RI(q_p) <: RI(q_y)$ gives us

$$RI(q_a) <: RI(q_y)$$

(TREAD) entails $q_y <: q_b$ which gives us

$$RI(q_y) <: RI(q_b)$$

Therefore, $RI(q_a) <: RI(q_b)$ holds.

$$\frac{\begin{array}{c}\Gamma(y) = q_x \quad \Gamma(x) = q_x \quad typeof(f) = q_f \\ q_y <: q_x \rhd q_f \\ q_x <: q_y\end{array}}{\Gamma \vdash i : x.f = y}\ \text{(TWRITE-ENDORSED)}$$

$$\frac{\begin{array}{c}\Gamma(x) = q_x \quad \Gamma(y) = q_y \quad \Gamma(z) = q_z \\ typeof(m) = q_{this}, q_p \to q_{ret} \\ !escape(p) \wedge !oEscape(p) \Rightarrow q_z <: q_i \rhd q_p \\ escape(p) \wedge !oEscape(p) \Rightarrow q_y = q_z \\ oEscape(p) \Rightarrow q_z <: q_i \rhd q_p \quad q_y = q_z\end{array}}{\Gamma \vdash i : x = y.m(z)}\ \text{(TCALL-ENDORSED)}$$

Figure 6: Typing rules for Oim. $escape(p)$ returns true if the Escape qualifier q_p is esc; it returns false otherwise. $oEscape(p)$ returns true if the oEscape qualifier q_p is oEsc; it returns false otherwise. Endorsement requires that receivers are canonical references and calls do not return (Section 2.2).

5. OBJECT IMMUTABILITY

We are now ready to present Oim, the object immutability type system. Qualifiers mutable, readonly and polyread have the same meaning as in ReIm.

Oim deviates slightly from the typical flow system by adding the immutable type qualifier to the hierarchy:

$$immutable <: readonly$$

An immutable reference x points to an immutable object. That is, in addition to x being readonly, all other references to the object x refers to, are readonly as well.

immutable is a subtype of readonly. Clearly, an immutable reference can be assigned to a readonly one, but not vice-versa, because there might be outstanding mutable references to the object. OIGJ defines a similar type hierarchy [24], except that it has mutable <: Raw <: readonly (recall that Raw's purpose is to allow for mutation at initialization, similarly to our endorsed statements).

Type inference for Oim is as described in Section 3.2.4. The ranking over the four qualifiers is:

$$\mathsf{immutable} > \mathsf{readonly} > \mathsf{poly} > \mathsf{mutable}$$

thus inferring a maximal amount of immutable references.

Typing Rules.

There are two changes from ReIm, necessary to handle endorsement. The new rules, (TWRITE-ENDORSED) and (TCALL-ENDORSED) are shown in Figure 6. All other rules, (TASSIGN), regular (TWRITE), (TREAD) and regular (TCALL) are as in Figure 3.

Consider (TWRITE-ENDORSE). When a field write x.f = y is endorsed, the receiver x does not become mutable due to this field write. (Note that it may become mutable due to a subsequent field write, which is not endorsed.) Instead, the receiver x now depends on y: $q_{\mathsf{x}} <: q_{\mathsf{y}}$ reflecting the fact that y is now part of the representation of x and mutation on y influences the object immutability status of x. A corollary of this rule is that x is immutable only if y is immutable. Consider this code snippet:

```
A[] r = new A[10];
A a = null;
for (int i=0; i<10; i++) {
    a = new A;
    endorse r[i] = a;
}
b = a;
b.f = 0;
```

Since an A object, part of array r, is modified at b.f = 0, a is mutable, and therefore r is mutable as well. To account for this Oim creates constraint $q_{\mathsf{r}} <: q_{\mathsf{a}}$ at the endorsed array write, which entails that r is mutable. The requirement that the receiver at endorsed field writes is a canonical reference (Section 2.2), is necessary for correctness.

Now consider rule (TCALL-ENDORSED). As with (TWRITE-ENDORSED) when an instance call is endorsed, we forgo modifications to the receiver made during the call.

Rule (TCALL-ENDORSED) enumerates all possible cases and creates appropriate constraints on argument z. In the first case the formal parameter p *does not escape* the call and it is *not modified* during the call (guaranteed by $!\mathit{oEscape}(\mathsf{p})$). Therefore, there is no "connection" between the receiver object and the parameter object (e.g., there is no this.f = p in which case the argument object becomes part of the receiver object; as another example, there is no code such as x = new X; this.f = x; p.g = x; in which case the receiver and parameter share representation).

In the second case, p may escape, however the escape is strictly through this (e.g., there is this.f = p). Furthermore, p is not modified during the call. The rule creates constraint $q_{\mathsf{y}} = q_{\mathsf{z}}$ with twofold intention: (1) this constraint accounts for the fact that z is now part of the representation of y: it entails $q_{\mathsf{y}} <: q_{\mathsf{z}}$ and therefore y is immutable only if z is immutable, and (2) it accounts for mutation to z through y: it entails $q_{\mathsf{z}} <: q_{\mathsf{y}}$ — since we forgo the standard constraint $q_{\mathsf{z}} <: q_i \rhd q_{\mathsf{p}}$, mutation to z outside of the call to m will be accounted for by $q_{\mathsf{z}} <: q_{\mathsf{y}}$. Note that there is no mutation during the call to m, or otherwise p would be oEsc. Additionally, p does not escape otherwise but to this, and therefore, all mutations to z outside of the call will occur through y.

In the last case, when p may escape (through this or not through this), or p may be modified during the call to m. Rule (TCALL-ENDORSED) demands both $q_{\mathsf{y}} = q_{\mathsf{z}}$ and $q_{\mathsf{z}} <: q_i \rhd q_{\mathsf{p}}$. Constraint $q_{\mathsf{y}} = q_{\mathsf{z}}$ accounts for the fact that part of p may have become part of this. This may happen because p escaped to this, or because a new object was created during the call to m and this new object escaped both through this and through p. Constraint $q_{\mathsf{z}} <: q_i \rhd q_{\mathsf{p}}$ accounts for mutation on z through p during the call.

Note that this last case is especially penalizing. If a formal parameter is modified during an endorsed call, this practically cancels endorsement. This includes the case when this escapes: this escapes only if a parameter is modified. Parameter modification triggers $q_{\mathsf{z}} <: q_i \rhd q_{\mathsf{p}}$ and since q_{p} is mutable q_{z} becomes mutable. It triggers $q_{\mathsf{y}} = q_{\mathsf{z}}$, and therefore q_{y} becomes mutable. Fortunately, modification of parameters during initialization calls is rare.

Recall the Couple example from Figure 1, part of it shown below for convenience:

```
Person immutable alice = new Person;
Person immutable bob = new Person;
endorse alice.partner = bob;
endorse bob.partner = alice;
Couple immutable couple = new Couple;
endorse couple.Couple(bob, alice);
```

The endorsed field writes account for $q_{\mathsf{bob}} = q_{\mathsf{alice}}$. The endorsed constructor call accounts for constraints $q_{\mathsf{bob}} = q_{\mathsf{couple}}$ and $q_{\mathsf{alice}} = q_{\mathsf{couple}}$. Since neither alice nor bob nor couple is mutated outside of endorsed statements, all remain immutable.

As another example, consider DateCell in Figure 2. Assume the statements at lines 18, 20 and 22 are endorsed. Since the parameter of constructor Date is an integer, line 18 triggers no constraints. Line 20 triggers $q_{\mathsf{dc}} = q_{\mathsf{d}}$ since parameter p of constructor DateCell is found noOesc. Line 22 triggers no constraints. As a result, dc and d both remain immutable.

Soundness.

We first give several definitions.

An endorsed call y.m(z) triggers an *endorsement block*. An *endorsement block* consists of the stack frame of m, as well as all other frames that are pushed and popped off the stack while the frame of m is active. Returning to Figure 2, assume that the call to setCellHours at line 22 is endorsed. The endorsement block triggered on dc at the call consists of the frame for setCellHours, the frame triggered by call this.getDate at line 6, and finally, the frame triggered by md.setHours(1) at line 7. An endorsed write y.f = x triggers a degenerate endorsement block which consists of the statement itself.

Recall that the preservation argument demands we prove that for each dynamic flow path $\mathsf{x} \longrightarrow^* \mathsf{y}$, $\mathit{RI}(q_{\mathsf{x}}) <: \mathit{RI}(q_{\mathsf{y}})$. A corollary of the preservation theorem is:

COROLLARY 5.1. *For every canonical reference* x*, such that* $q_{\mathsf{x}} = \mathsf{immutable}$*, there does not exist a flow path* $\mathsf{x} \longrightarrow^* \mathsf{y}$, y.f = ...*, where field write* y.f = ... *occurs outside of an endorsement block triggered on* y.

To reason about deep immutability we need to define the *Closure* of canonical reference x:

DEFINITION 5.2. *Let* x *be the canonical reference that corresponds to object* o*. Let* Y *be the set of all canonical references* y *that correspond to objects* o' *at field writes* $o.\mathsf{f} = o'$*.*

The closure of x *is defined as follows:*

$$\mathit{Closure}(\mathsf{x}) = \{\mathsf{x}\} \cup \bigcup_{\mathsf{y} \in Y} \mathit{Closure}(\mathsf{y})$$

Intuitively, the closure consists of x and all transitive components of x. (We use x and the object that corresponds to x interchangeably.) For example, in Figure 2, the closure of DateCell dc includes dc and d. In Figure 1 the closure of bob is {bob, alice}, and the closure of couple is {couble, bob, alice}.

Further, we divide canonical references $y \in \mathit{Closure}(x)$ into two categories: *internal* and *external*. An internal reference occurs within an endorsement scope triggered by x. An external reference occurs outside such a scope. For example, bob and alice are both external references with respect to couple. We can now state our immutability guarantee.

LEMMA 5.3. *If* x *is* immutable *then*

(1) for each internal $z \in \mathit{Closure}(x)$ *there does not exist a flow path* $z \longrightarrow^* y$, $y.f = \ldots$, *where* $y.f = \ldots$ *occurs outside of an endorsement block triggered on* x, *and*

(2) for each external $z \in \mathit{Closure}(x)$, *there does not exist a flow path* $z \longrightarrow^* y$, $y.f = \ldots$, *where* $y.f = \ldots$ *occurs outside of an endorsement block triggered on some external* $z' \in \mathit{Closure}(x)$.

Informally, the lemma states that if x is immutable, all components of x, including transitive components, are never modified after x's initialization has completed.

If x is not part of a strongly-connected structure then all of its components are immutable (in the sense of Corollary 5.1) and their endorsement blocks *happen before* x's endorsement block. This is the case with couple. Its components bob and alice are immutable, and their respective endorsement blocks happen before couple's initialization.

Again informally, suppose that x is part of a strongly-connected structure S. The lemma states that all of x's components are immutable, and their endorsement blocks happen before the *last* endorsement block triggered on a reference from S. In our running example, bob and alice form a strongly-connected structure. bob becomes part of alice first, but alice's endorsement block happens before bob's. Initialization of alice completes when the *last* endorsement block, in this case bob's, completes.

We are preparing a technical report which formalizes all theorems and proofs.

6. EXPERIMENTS

We have implemented Escape, oEscape and Oim in our framework for inference and checking of pluggable types, which we have used to instantiate many non-trivial and useful type systems [14, 17, 16]. We have used the Soot-based Java bytecode front end of our framework. (In addition, we have an Android bytecode and Java source front ends.) Our framework is publicly available online (https://github.com/proganalysis/type-inference). We will release the type systems from this paper too, once we have finalized all proofs.

One important advantage of our type-based analysis is that it is modular and compositional. That is, inference and checking can be done on *any set of classes L*. Unknown (i.e., unanalyzed) libraries called from L are typed using the worst-case type. For example, in reference immutability, the default type for library parameters is mutable. User code written on top of L can be composed with L by applying the method overriding constraints stated at the end of Section 3.2.3, on all classes that override classes from L.

Benchmark	#Allocs	#Immutable Allocs	Time
antlr	1264	779 (61.6%)	34.1
bloat	1878	451 (24.0%)	37.7
chart	6937	1931 (27.8%)	85.4
eclipse	2148	787 (36.6%)	38.5
fop	6671	2200 (33.0%)	119.2
hsqldb	3374	1193 (35.4%)	67.5
luindex	706	260 (36.8%)	17.2
lusearch	869	347 (39.9%)	20.4
pmd	4265	1850 (43.4%)	84.7
xalan	4027	1946 (48.3%)	57.7
Average		38.7%(am) 36.5%(gm)	

Table 1: Inference results for object immutability. #Allocs shows the number of object allocations excluding String, StringBuffer and immutable boxed primitives (e.g., Integer). #Immutable Allocs gives the number and percentage of those allocations inferred as immutable. The last column Time shows the total running time, in seconds, including Escape, oEscape and Oim inference. Row Average shows the arithmetic mean and geometric mean.

We take advantage of the modularity and compositionality. We analyze the standard Dacapo benchmarks, but apply default types on libraries for each specific type system. In Escape, we type the implicit parameter this poly and all other parameters esc. While the typing of this is not the worst-case esc, this escapes through parameters extremely rarely and thus the less conservative poly type is justified. Note that the poly type captures the case when this escapes through a return (e.g., A get() { return this.f; }). In oEscape, we type all parameters oEsc. In Oim, we use the annotated JDK from our previous work [17] and [16] (also used in Javari [22]). Only a small number of parameters are annotated readonly. The vast majority of parameters are mutable.

We applied the analysis on the standard Dacapo benchmarks, Dacapo-2006-10MR2 [1]. All experiments run on a MacBook Pro with a 2.8 GHz Intel Core i7 processor and 16GB of RAM. However, we restrict the max heap size to 2GB. The software infrastructure consists of Soot and Eclipse Luna configured with Java 7.

The results are presented in Table 1. Column **#Allocs** counts all new sites, including newInstance sites across all classes in a benchmark. (We do analyze *all* classes included in a Dacapo jar.) **#Allocs** excludes new sites that allocate Strings or StringBuffers as well as new sites that allocate immutable boxed primitives (e.g., Integer, Long, Boolean, etc.). We automatically endorse all calls on canonical references that do not return references, as well as all field/array writes on canonical references. More than 40% of the allocation sites allocate immutable objects, which attests to the precision of our light-weight analysis. Running times, which include type inference for Escape, oEscape and Oim, run in this order to account for appropriate dependences, are all under 2 minutes, which attests to scalability. In summary, the experiments confirm that the analysis is precise and scalable and can be applied in compilers and software tools.

7. RELATED WORK

Immutability has been studied extensively [19]. There is a number of reference immutability systems that have been implemented and successfully run on large real-world Java applications [22, 17].

Unfortunately, we are not aware of an object immutability system, which has been run on real-world Java programs to infer object immutability. State of the art in object immutability includes Joe_3 [5], IGJ [23], OIGJ [24] and IOJ by Haack et al. [12, 11]. These systems are more powerful than ours but also more complex. Most notably, they make use of ownership to disallow representation exposure at object construction. The complexity of ownership makes inference particularly difficult, which is the reason why neither system has been applied to infer object immutability in real-world programs. Our system eschews ownership; it captures representation exposure with the novel combination of light-weight, targeted escape analysis and reference immutability. As a result, Oim scales well.

There are similarity between existing object immutability systems and ours. The Raw type qualifier of IGJ and OIGJ and the fresh(n) initialization block of IOJ [11] have similar semantics to our endorse qualifier, as we point out earlier.

Potanin et al. [19] present and excellent survey on immutability, including object immutability.

Our work is related to work on borrowing permissions [18, 2, 3, 4, 13]. In a sense, our type system deals with aliasing created close to object initialization. Work in the space of borrowing permissions deals with aliasing created later in the object lifetime. It will be interesting to explore the synergy between these complementary lines of work.

8. CONCLUSION AND FUTURE WORK

We presented a novel type system for object immutability that leverages reference immutability. Our system handles deep object immutability and delayed object initialization. We have implemented the system. Nearly 40% of all static objects are inferred immutable.

There are many avenues for future work. We will fully formalize and publish our soundness proofs. In this vein, we are especially interested in developing more elegant and less restrictive oEscape reasoning. We will explore case studies on real-world codes. Finally, we will develop dynamic analysis to serve as "ground truth" for our evaluation — while 40% immutable objects is significant, at this point we do not what the "ground truth" percentage of immutable objects is.

9. ACKNOWLEDGMENTS

We thank the PPPJ'16 reviewers for their detailed and valuable comments, which we tried to incorporate in the paper. This work was supported by NSF grant CCF-1319384.

10. REFERENCES

[1] S. M. Blackburn, S. Z. Guyer, M. Hirzel, and et. al. The Dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[2] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, May 2001.

[3] J. Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.

[4] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27, 2001.

[5] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, volume 37, pages 292–310, 2002.

[6] D. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[7] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. Universe types for topology and encapsulation. In *FMCO*, pages 72–112, 2008.

[8] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[9] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):1–48, Apr. 2012.

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312, 2003.

[11] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In *ECOOP*, pages 520–545, 2009.

[12] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In *ESOP*, pages 347–362, 2007.

[13] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, pages 354–378, 2010.

[14] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.

[15] W. Huang, Y. Dong, and A. Milanova. Type-based Taint Analysis for Java Web Applications. Technical report, Rensselaer Polytechnic Institute, 2013.

[16] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for Android. In *ISSTA*, July 2015.

[17] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, pages 879–896, 2012.

[18] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL*, pages 557–570, 2012.

[19] A. Potanin, J. Östlund, Y. Zibin, and M. D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 233–269. Springer-Verlag, Apr. 2013.

[20] J. Quinonez, M. S. Tschantz, and M. D. Ernst. Inference of reference immutability. In *ECOOP*, pages 616–641, 2008.

[21] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[22] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[23] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and Reference Immutability using Java Generics. In *ESEC/FSE*, pages 75–84, 2007.

[24] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. In *OOPSLA*, pages 598–617, 2010.