ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity

Wei Huang Ana Milanova
Rensselaer Polytechnic Institute
{huangw5, milanova}@cs.rpi.edu

Werner Dietl Michael D. Ernst
University of Washington
{wmdietl, mernst}@cs.washington.edu

Abstract

Reference immutability ensures that a reference is not used to modify the referenced object, and enables the safe sharing of object structures. A pure method does not cause side-effects on the objects that existed in the pre-state of the method execution. Checking and inference of reference immutability and method purity enables a variety of program analyses and optimizations.

We present a type system for reference immutability and a corresponding type inference analysis, ReIm and ReImInfer, respectively. The type system is concise and context-sensitive. The type inference analysis is precise and scalable, and requires no manual annotations. In addition, we present a novel application of the reference immutability type system: method purity inference.

To support our theoretical results, we implemented the type system and the type inference analysis for Java. We include a type checker to verify the correctness of the inference result. Empirical results on Java applications and libraries of up to 384kLOC show that our approach achieves both scalability and precision.

1. Introduction

An immutable, or readonly, reference cannot modify the state of an object, including the transitively reachable state. For instance, in the following code, the Date object cannot be modified by using the immutable reference rd, but the same Date object can be modified through the mutable reference md:

```
Date md = new Date(); // mutable by default
readonly Date rd = md; // an immutable reference
md.setHours(1); // OK, md is mutable
rd.setHours(1); // compile−time error, rd is immutable
```


OOPSLA '12 October 19-26, 2012, Tucson, AZ.

The qualifier readonly denotes that rd is an immutable reference. By contrast to reference immutability, object immutability enforces a stronger guarantee that no reference in the system can modify a particular object. Each variety of immutability is preferable in certain situations; neither dominates the other. This paper only deals with reference immutability.

As a motivating example, consider a simplification of the Class.getSigners method implemented in JDK 1.1:

```
class Class {
  private Object[] signers;
  public Object[] getSigners() {
    return signers;
  }
}
```

This implementation is not safe because a malicious client can obtain a reference to the signers array by invoking the getSigners method and can then side-effect the array to add an arbitrary trusted signer. Even though the field is declared private, the referenced object is still modifiable from the outside. There is no language support for preventing outside modifications, and the programmer must manually ensure that the code only returns clones of internal data.

A solution is to use reference immutability and annotate the return value of getSigners as readonly. (A readonly array is expressed, following Java 8 syntax [12], as Object readonly [].) As a result, mutations of the array through the returned reference will be disallowed:

```
Object readonly [] getSigners() {
  return signers;
}
...
getSigners()[0] = maliciousClass; // compile−time error
```

A type system enforcing reference immutability has a number of benefits. It improves the expressiveness of interface design by specifying the immutability of parameters and return values; it helps prevent and detect errors caused by unwanted object mutations; it facilitates reasoning about and proving other properties such as object immutability

and method purity; it supports enforcement of the owner-as-modifier discipline [9, 10].

This paper presents a context-sensitive type system for reference immutability, ReIm, and a novel and efficient inference analysis, ReImInfer. We implemented our system for Java and performed case studies of applications and libraries of up to 384kLOC.

ReIm is related to Javari [27], the state-of-the-art in reference immutability, but also differs in important points of design and implementation. ReIm's design was motivated by a particular application: method purity inference. As a result, ReIm is simpler than Javari, if less expressive in some respects that are irrelevant to purity inference. ReIm treats every structure as a whole and assigns a single mutability to the structure. By contrast, Javari contains multiple features for excluding certain fields or generic type arguments from the immutability guarantee. Another difference is that ReIm encodes context sensitivity using the concept of *viewpoint adaptation* from Universe Types [9, 10], while Javari uses templating. These design decisions result in a more compact and scalable type system, particularly suitable for reasoning about method purity.

Our system allows programmers to annotate only references they care about (programmers may choose to annotate no references at all). The inference analysis fills in the remaining types, and the system performs type checking. The inference is *precise* in the sense that it infers the maximal number of immutable references. It has $O(n^2)$ worst-case complexity and scales linearly in practice. Our inference system, ReImInfer, has two advantages over Javarifier [22], the state of the art reference immutability inference tool. First, as with ReIm, it models context sensitivity using the concept of viewpoint adaptation from Universe Types. Javarifier handles context sensitivity by replicating methods. Viewpoint adaptation contributes to the better scalability of ReImInfer compared to Javarifier. Second, our tool relies entirely on the Checker Framework [11, 19], which provides better integration of programmer-provided annotations, type inference, and type checking.

In addition, we present method purity inference built as an application of reference immutability. Purity information facilitates compiler optimization [6, 16, 31], model checking [26], Universe types inference [9], and memorization of procedure calls [14]. Purity inference (also known as side-effect analysis) has a long history. Most existing purity or side effect analyses are whole-program analyses that are based on points-to analysis and/or escape analysis and therefore scale poorly. We know of no purity inference tool that scales to large Java codes and analyzes both whole programs and libraries.

Our reference immutability inference and purity inference are *modular* and *compositional*. They are modular in the sense that they can analyze any given set of classes L. Unknown callees in L are handled using appropriate defaults. Callers of L can be analyzed separately and composed with L without re-analysis of L.

In summary, we make the following contributions:

- ReIm, a context-sensitive type system for reference immutability. A key novelty in ReIm is the use of *viewpoint adaptation* to encode context sensitivity.
- ReImInfer, a type inference algorithm for reference immutability.
- A novel application of reference immutability: method purity inference.
- An implementation for Java.
- An empirical evaluation of reference immutability inference and purity inference on programs of up to 348,229 lines of code, including widely used Java applications and libraries, comprising 766,053 lines of code in total.

The rest of this paper is organized as follows. Section 2 describes the type system and the inference analysis. Section 3 presents purity inference, and Section 4 describes our experiments. Section 5 discusses related work, and Section 6 concludes. The appendix in Section A formalizes the concrete semantics, the well-formedness properties, and type soundness.

2. ReIm Reference Immutability Types

In this section, we describe the immutability types (Section 2.1) and then explain context sensitivity (Section 2.2). We proceed to define the type system for reference immutability (Section 2.3), followed by the inference analysis (Section 2.4).

2.1 Immutability Qualifiers

There are three immutability qualifiers in our type system:

- 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:
  - $\mathsf{x.f} = \mathsf{z}$
  - $\mathsf{x.set(z)}$ where set sets a field of its receiver
  - $\mathsf{y} = \mathsf{id(x)};\ \mathsf{y.f} = \mathsf{z}$ where id is a function that returns its argument
  - $\mathsf{x.f.g} = \mathsf{z}$
  - $\mathsf{y} = \mathsf{x.f};\ \mathsf{y.g} = \mathsf{z}$
- polyread: A polyread reference x cannot be used to mutate the referenced object. A method may return x to the caller, and the caller may mutate the object. Programmers should use polyread when the reference is readonly in the scope of the enclosing method, but depends on the context of the caller of the method. For example,

- x.f = 0, where x is polyread, is not allowed, but
- z = id(y); z.f = 0, where id is
  polyread X id(polyread X x) { return x; },
  is allowed when y and z are mutable.

polyread can be applied to parameters and local variables of both instance and static methods. It is important to note that polyread cannot be applied to fields, or in other words, our system is context-sensitive in the method-transmitted data dependences, but is approximate in the structure-transmitted data dependences. This is necessitated by a fundamental result by Reps [23] which states that (fully) context-sensitive, structure-transmitted data-dependence analysis is undecidable. Additional discussion of polyread follows in Section 2.2.

The subtyping relation between the qualifiers is

$$\text{mutable} <: \text{polyread} <: \text{readonly}$$

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2. For example, it is allowed to assign a mutable reference to a polyread or readonly one, but it is not allowed to assign a readonly reference to a polyread or mutable one.

2.2 Context Sensitivity

Context sensitivity is important for precision and we use a variant of viewpoint adaptation [9] to express context-sensitivity in our system. Consider the following code:

```
1  class DateCell {
2    Date date;
3    Date getDate() { return this.date; }
4    void m1() {
5      Date md = this.getDate(); // md is mutable
6      md.setHours(1);
7    }
8    int m2() {
9      Date rd = this.getDate(); // rd is readonly
10     int hour = rd.getHours();
11     return hour;
12   }
13 }
```

The return value of method getDate has different mutabilities in different contexts m1 and m2. A context-insensitive type system would force the return of getDate to be mutable due to m1 at line 5:

```
Date md = this.getDate();
```

The left-hand-side of the call is the mutable reference md. Therefore, field access expression this.date would become mutable, which would force this of getDate to become 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). The mutability of this of getDate will propagate, and force this of m1 as well as this of m2 to become mutable.

However, this of m2 is readonly, and we would like our type system to express this fact. We use qualifier polyread. We annotate this and the return of getDate as polyread:

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

For readability, the above code, and other code throughout this section, makes formal parameter this explicit.

We use viewpoint adaptation to adapt polyread from the point of view of left-hand-side md (which is mutable) in the context of m1, and from the point of view of rd (which is readonly) in the context of m2. Intuitively, viewpoint adaptation instantiates polyread to mutable in the context of m1, and to readonly in the context of m2. As a result, the mutability of md propagates only to this of m1; it does not propagate to this of m2 which remains readonly.

Viewpoint adaptation is a concept from Universe Types [7, 9, 10], which can be adapted to Ownership Types [5] and ownership-like type systems such as AJ [29]. 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 $q \triangleright q' = q''$. Traditional viewpoint adaptation from Universe Types defines one viewpoint adaptation operation $\triangleright$; it uses $\triangleright$ to adapt fields, formal parameters, and method returns from the point of view of the *receiver* at the field access or method call.

Below, we explain viewpoint adaptation for reference immutability. Unlike traditional viewpoint adaptation from Universe Types, there are two viewpoint adaptation operations, $\triangleright_f$ for field accesses, and $\triangleright_m$ for method calls. Furthermore, while $\triangleright_f$ adapts from the point of view of the receiver at the field access, $\triangleright_m$ does not adapt from the point of view of the receiver at the call — it adapts from the point of view of the *left-hand side* at the call.

Viewpoint adaptation $q \triangleright_f q_\mathsf{f}$ is applied at field accesses. It adapts declared field qualifier q_f from the point of view of receiver qualifier q. We define $\triangleright_f$ as:

$$\begin{array}{lcl} \_ \triangleright_f \text{readonly} & = & \text{readonly} \\ q \triangleright_f \text{mutable} & = & q \end{array}$$

The underscore denotes a "don't care" value. Consider field access y.f. If the type of receiver y is readonly and the declared type of field f is mutable, then the type of y.f is readonly $\triangleright_f$ mutable $=$ readonly. A field access y.f is mutable if and only if both the receiver y and field f are mutable. If the receiver or the field is readonly, y.f is readonly. Finally, if the receiver is polyread and the field is mutable, y.f is polyread.

Viewpoint adaptation $q_\mathsf{x} \triangleright_m q$ is applied at method calls $\mathsf{x} = \mathsf{y.m(z)}$. It adapts q, the declared qualifier of a formal parameter/return of m, from the point of view of q_x, the

$$
\begin{array}{lll}
cd & ::= \mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\} & \textit{class} \\
fd & ::= t_f\ \mathsf{f} & \textit{field} \\
md & ::= t\ \mathsf{m}(t\ \mathsf{x})\ \{\ \overline{t\ \mathsf{y}};\ \mathsf{return\ y}\ \} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s;s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} & \textit{statement} \\
 & \mid \mathsf{x} = \mathsf{y.f} \mid \mathsf{x.f} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.m(z)} & \\
t_f & ::= q_f\ \mathsf{C} & \textit{field type} \\
q_f & ::= \mathsf{readonly} \mid \mathsf{mutable} & \textit{field qualifier} \\
t & ::= q\ \mathsf{C} & \textit{qualified type} \\
q & ::= \mathsf{readonly} \mid \mathsf{polyread} \mid \mathsf{mutable} & \textit{qualifier}
\end{array}
$$

Figure 1. Syntax. C and D are class names, f is a field name, m is a method name, and x, y, and z are names of local variables or parameters. For simplicity, we assume all names are unique. Note that we distinguish the type qualifiers used for fields from those used for local variables/parameters.

qualifier at the left-hand side x. We define $\triangleright_m$ as:

$$
\begin{array}{lcl}
\_ \triangleright_m \mathsf{mutable} & = & \mathsf{mutable} \\
\_ \triangleright_m \mathsf{readonly} & = & \mathsf{readonly} \\
q_\mathsf{x} \triangleright_m \mathsf{polyread} & = & q_\mathsf{x}
\end{array}
$$

If a formal parameter/return is readonly or mutable, its adapted type remains the same regardless of q_x. However, if q is polyread, the adapted type depends on q_x — it becomes q_x (i.e., the polyread type is the polymorphic type, and it is instantiated to q_x).

This is a generalization of traditional viewpoint adaptation in two ways. (1) we allow for two different viewpoint adaptation operations, one for field accesses, and one for method calls. (2) we allow for adaptation from *other* points of view, not only the point of view of the receiver as in traditional viewpoint adaptation We use viewpoint adaptation to encode context sensitivity. Thus, (1) can be interpreted as encoding context sensitivity at field-transmitted dependences *differently* from context sensitivity at call-transmitted dependences. (2) can be viewed as allowing different *abstractions of context*. For example, adaptation from the point of view of the receiver amounts to object sensitivity [18]. Adaptation from the point of view of the left-hand side of a call amounts to call-site context sensitivity. We note that the purpose of this paper is to develop reference immutability and method purity. The precise relation between context sensitivity in dataflow analysis, CFL-reachability [23], and viewpoint adaptation is left for future work.

2.3 Typing Rules

For brevity, we restrict our formal attention to a core calculus in the style of Vaziri et al. [29] whose syntax appears in Figure 1. The language models Java with a syntax in a "named form", where the results of field accesses, method calls and instantiations are immediately stored in a variable. Without loss of generality, we assume that methods have exactly one parameter. Features not strictly necessary are omitted from the formalism, but they are handled correctly in the implementation. We write $\overline{t\ \mathsf{y}}$ for a sequence of local variable declarations.

$$
\frac{}{\Gamma \vdash \mathsf{x} = \mathsf{new\ mutable\ C}}\ (\textsc{tnew})
$$

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

$$
\frac{\Gamma(\mathsf{x}) = \mathsf{mutable} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \qquad q_\mathsf{y} <: \mathsf{mutable} \triangleright_f q_\mathsf{f}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\textsc{twrite})
$$

$$
\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \mathit{typeof}(\mathsf{f}) = q_\mathsf{f} \qquad q_\mathsf{y} \triangleright_f q_\mathsf{f} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})
$$

$$
\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \Gamma(\mathsf{z}) = q_\mathsf{z} \\ \mathit{typeof}(\mathsf{m}) = q_\mathsf{this}, q_p \rightarrow q_\mathsf{ret} \\ q_\mathsf{y} <: q_\mathsf{x} \triangleright_m q_\mathsf{this} \qquad q_\mathsf{z} <: q_\mathsf{x} \triangleright_m q_p \\ q_\mathsf{x} \triangleright_m q_\mathsf{ret} <: q_\mathsf{x}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ (\textsc{tcall})
$$

Figure 2. 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.

In contrast to a formalization of pure Java, a type t has two orthogonal components: type qualifier q and Java class type C. The type system is *orthogonal* to (i.e., independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

The type system is presented in Figure 2. Appendix A defines the operational semantics of reference immutability and proves soundness of the type system.

Rules (TNEW) and (TASSIGN) are straightforward. They require that the left-hand-side is a supertype of the right-hand-side. The system does not enforce object immutability and, for simplicity, only mutable objects are created. Rule (TWRITE) requires $\Gamma(\mathsf{x})$ to be mutable because x's field is updated in the statement. The adapt rules for field access are used in both (TWRITE) and (TREAD).

Rule (TCALL) demands a detailed explanation. Function *typeof* retrieves the type of m. q_this is the type of implicit parameter this, q_p is the type of the formal parameter, and q_ret is the type of the return. The rule requires $q_\mathsf{y} <: q_\mathsf{x} \triangleright_m q_\mathsf{this}$. When q_this is readonly or mutable, its adapted value is the

same. Thus, when q_{this} is mutable due to a statement this.f = 0 for example,

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

which disallows q_{y} from being anything but mutable, as expected. If q_{this} is polyread, this expresses a dependence between this and ret of m (e.g., due to z = this.f; return z;). Therefore, the mutability of y depends on the mutability of x: if x is mutable then y must be mutable, if x is readonly, then readonly y is allowed, given that y is not mutated due to another statement. Viewpoint adaptation transfers the dependence between this and ret in the callee, into a dependence between y and x in the caller. When this is polyread, constraint $q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_m q_{\mathsf{this}}$ becomes

$$q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_m \mathsf{polyread} \quad \text{and then} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}$$

When q_{x} is mutable, the constraint disallows y from being anything but mutable. When q_{x} is readonly, y can be readonly as well. Exactly the same argument applies to constraint $q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd_m q_p$. When q_p is polyread, this expresses a dependence between the formal parameter p of m and ret, and the mutability of z depends on the mutability of x. Again, viewpoint adaptation transfers the dependence between p and ret in the callee, into a dependence between z and x in the caller.

In addition, (TCALL) requires $q_{\mathsf{x}} \rhd_m q_{\mathsf{ret}} <: q_{\mathsf{x}}$. This constraint disallows the return value of m from being readonly when there is a call to m, x = y.m(z), where left-hand-side x is mutable. Only if the left-hand sides of all calls to m are readonly, can the return type of m be readonly; otherwise, it is polyread. Note that it is allowed to annotate the return type of m as mutable. However, this typing is pointless, because it unnecessarily forces local variables and parameters in m to become mutable when they can be polyread.

Our inference tool, ReImInfer, types the DateCell class from Section 2.2 as follows:

```
class DateCell {
  mutable Date date;
  polyread Date getDate(polyread DateCell this) {
    return this.date;
  }
  void m1(mutable DateCell this) {
    mutable Date md = this.getDate();
    md.setHours(1);
  }
  void m2(readonly DateCell this) {
    readonly Date rd = this.getDate();
    int hour = rd.getHours();
  }
}
```

Field date is mutable because it is mutated indirectly in method m1. Because the type of this of getDate is polyread, it is instantiated to mutable in m1 as follows:

$$q_{\mathsf{md}} \rhd_m q_{\mathsf{this}} \quad = \quad \mathsf{mutable} \rhd_m \mathsf{polyread} \quad = \quad \mathsf{mutable}$$

It is instantiated to readonly in m2:

$$q_{\mathsf{rd}} \rhd_m q_{\mathsf{this}} \quad = \quad \mathsf{readonly} \rhd_m \mathsf{polyread} \quad = \quad \mathsf{readonly}$$

This allows this of m2 to be typed readonly.

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

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

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 $q_{\mathsf{ret}_{\mathsf{m}'}} <: q_{\mathsf{ret}_{\mathsf{m}}}$.

2.4 Type Inference

The type inference algorithm operates on mappings from keys to values S. The keys in the mapping are (1) local variables and parameters, including implicit parameters this, (2) field names and (3) method returns. The values in the mapping are *sets* of types. For instance, $S(\mathsf{x}) = \{\mathsf{polyread}, \mathsf{mutable}\}$ means the type of reference x can be polyread or mutable. For the rest of the paper we use "reference" to refer to all kinds of keys: variables, fields and method returns.

S is initialized as follows. Programmer-annotated references are initialized to the singleton set that contains the programmer-provided type. Note that there may be no programmer-annotated variables at all. Method returns are initialized $S(\mathsf{ret}) = \{\mathsf{readonly}, \mathsf{polyread}\}$ for each method m. Fields are initialized $S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{mutable}\}$. All other references are initialized to the maximal set of types, i.e., $S(\mathsf{x}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$.

There is a transfer function f_s for each statement s. Each f_s takes as input the current mapping S and outputs an updated mapping S'. f_s refines the set of each reference that participates in s as follows. Let $\mathsf{x}, \mathsf{y}, \mathsf{z}$ be the references in s. For each reference, say x, f_s removes each $t_{\mathsf{x}} \in S(\mathsf{x})$ from $S(\mathsf{x})$, if there does not exist a pair $q_{\mathsf{y}} \in S(\mathsf{y})$, $q_{\mathsf{z}} \in S(\mathsf{z})$ such that $q_{\mathsf{x}}, q_{\mathsf{y}}, q_{\mathsf{z}}$ type check under the type rule for s from Figure 2. For example, consider statement x = y.f and corresponding rule (TREAD). Suppose that $S(\mathsf{x}) = \{\mathsf{polyread}\}$, $S(\mathsf{y}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$ and $S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{mutable}\}$ before the application of the transfer function. The transfer function removes readonly from $S(\mathsf{y})$ because there does not exist $q_{\mathsf{f}} \in S(\mathsf{f})$ that satisfies $\mathsf{readonly} \rhd_f q_{\mathsf{f}} <: \mathsf{polyread}$. Similarly, it removes readonly from $S(\mathsf{f})$ because $\_ \rhd_f \mathsf{readonly} = \mathsf{readonly}$, is not a subtype of polyread as rule (TREAD) requires. After the application of the transfer function, S' is as follows: $S'(\mathsf{x}) = \{\mathsf{polyread}\}$, $S'(\mathsf{y}) = \{\mathsf{polyread}, \mathsf{mutable}\}$, and $S'(\mathsf{f}) = \{\mathsf{mutable}\}$.

The inference analysis iterates over the statements in the program and refines the sets until either (1) a reference is assigned the empty set in which case the analysis terminates with an error, or (2) the iteration reaches a fixpoint.

Note that the result of fixpoint iteration is a mapping from references to *sets*. The actual mapping from references to types is derived as follows: for each reference x we pick the largest element of $S(\mathsf{x})$ according to the preference ranking readonly $>$ polyread $>$ mutable, because we want to maximize the number of readonly references. Note that leaving all references as mutable is also a valid typing but a useless one, as it expresses nothing about immutability. The following rather interesting propositions hold:

Proposition 2.1. *The type assignment type checks under the rules from Figure 2.*

Proof. (Sketch) The proof is a case-by-case analysis which shows that after the application of each transfer function, the rule type checks with the maximal assignment. We show (TCALL) $\mathsf{x} = \mathsf{y.m(z)}$. The rest of the cases are straightforward.

- Let $max(S(q_{\mathsf{x}}))$ be readonly. $q_{\mathsf{x}} \rhd_m q_{\mathsf{ret}} <: q_{\mathsf{x}}$ holds for any value of $max(S(\mathsf{ret}))$. If $max(S(q_{\mathsf{this}}))$ is readonly or polyread, $q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_m q_{\mathsf{this}}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(q_{\mathsf{this}}))$ is mutable, the only possible max for y would be mutable (the others would have been removed by the transfer function for (TCALL)).
- Let $max(S(q_{\mathsf{x}}))$ be mutable. If $max(S(\mathsf{ret}))$ is polyread, clearly $q_{\mathsf{x}} \rhd_m q_{\mathsf{ret}} <: q_{\mathsf{x}}$ holds. $max(S(\mathsf{ret}))$ cannot be readonly, readonly would have been removed by the transfer function.
  If $max(S(q_{\mathsf{this}}))$ is readonly, $q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_m q_{\mathsf{this}}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(q_{\mathsf{this}}))$ is polyread, the only possible value for $max(S(\mathsf{y}))$ would be mutable. If $max(S(q_{\mathsf{this}}))$ is mutable, the only possible max for y would be mutable as well (the others would have been removed by the transfer function for (TCALL)).
- Let $max(S(q_{\mathsf{x}}))$ be polyread. If $max(S(\mathsf{ret}))$ is polyread, clearly $q_{\mathsf{x}} \rhd_m q_{\mathsf{ret}} <: q_{\mathsf{x}}$ holds. $max(S(\mathsf{ret}))$ cannot be readonly, readonly would have been removed by the transfer function.
  If $max(S(q_{\mathsf{this}}))$ is readonly, $q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_m q_{\mathsf{this}}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(q_{\mathsf{this}}))$ is polyread, the only possible value for $max(S(\mathsf{y}))$ would be polyread or mutable. If $max(S(q_{\mathsf{this}}))$ is mutable, the only possible max for y would be mutable.

□

Proposition 2.2. *The type assignment is precise. That is, all references that can be* readonly*, are assigned* readonly.

Proof. (Sketch) A valid typing is an assignment T from variables to qualifiers, such that the program type checks with the rules from Figure 2. We say that q is a *valid qualifier* for x if there exists a valid typing T, where $T(\mathsf{x}) = q$. Let x be the *first* variable that has a valid qualifier q removed from its set $S(\mathsf{x})$ and let f_s be the transfer function that performs the removal. Since q is a valid qualifier there exists valid qualifiers y, z that make s type check. If $q \in S(\mathsf{x})$, $q_{\mathsf{y}} \in S(\mathsf{y})$, and $q_{\mathsf{z}} \in S(\mathsf{z})$, then by definition, f_s would not have had q removed from $S(\mathsf{x})$. Thus, one of y of z must have had a valid qualifier removed from its set *before* the application of f_s. This contradicts the assumption that x is the first variable that has a valid qualifier removed. Thus, if readonly is not in the set for x, this means that there does not exist a valid typing that types x readonly. Or in other words, if x can be assigned readonly, it is assigned readonly in our typing. □

```
class A {
  X f;                                  S(f) = {mutable}
  X get(A this, Y y) {                  S(this_get) = {polyread, mutable}
    ... = y.h;                          S(y_get) = {readonly, polyread, mutable}
    X x = this.getX();                  S(x_get) = {polyread, mutable}
    return x;                           S(ret_get) = {polyread}
  }
  X getX(A this) {                      S(this_getX) = {polyread, mutable}
    X x = this.f;                       S(x_getX) = {polyread, mutable}
    return x;                           S(ret_getX) = {polyread}
  }
}
void m1() {
  A a = ...                             S(a_m1) = {mutable}
  Y y = ...                             S(y_m1) = {readonly, polyread, mutable}
  X x = a.get(y);                       S(x_m1) = {mutable}
  x.g = null;
}
void m2() {
  A a = ...                             S(a_m2) = {readonly, polyread, mutable}
  Y y = ...                             S(y_m2) = {readonly, polyread, mutable}
  X x = a.get(y);                       S(x_m2) = {readonly, polyread, mutable}
  ... = x.g;
}
```

Figure 3. Polymorphic methods. A.get(Y) has different mutabilities in the contexts of m1 and m2. A.getX() which is called from A.get(Y) has different mutabilities as well. The box beside each statement shows the set-based solution; the underlined qualifiers are the final type picked by ReImInfer.

These propositions are validated empirically as detailed in Section 4. To validate Proposition 2.1, we build an independent type checker in the Checker Framework and type check the inferred types. To validate Proposition 2.2, we perform detailed comparison with Javarifier, the state-of-the-art tool for inference of reference immutability.

Consider the example in Figure 3. We use $\mathsf{x}_{\mathsf{get}}$ to denote the reference x in method get. Initially, all references are initialized to the sets as described above. The analysis iterates

over all statements in class A, methods m1 and m2. In the first iteration, the analysis changes nothing until it processes x.g = null in m1. $S(\mathsf{x}_{\mathsf{m1}})$ is updated to {mutable}. In the second iteration, when the analysis processes x = a.get(y), $S(\mathsf{ret}_{\mathsf{get}})$ becomes {polyread}. In the third iteration, $S(\mathsf{x}_{\mathsf{get}})$ becomes {polyread, mutable} because x has to be a subtype of $S(\mathsf{ret}_{\mathsf{get}})$. This in turn forces $S(\mathsf{ret}_{\mathsf{getX}})$ and subsequently $S(\mathsf{this}_{\mathsf{getX}})$ to become {polyread, mutable}. The iteration continues until it reaches the fixpoint as shown in the boxes in Figure 3. For brevity, some references are not shown in the boxes. The underlined qualifiers are the largest element according to the preference ranking defined before.

The fixpoint will be reached in $O(n^2)$ time where n is the size of the program. In each iteration, at least one of the $O(n)$ references is updated to point to a smaller set. Hence, there are at most $O(3n)$ iterations (recall that each set has at most 3 qualifiers), resulting in the $O(n^2)$ time complexity.

3. Method Purity

A method is *pure* (or side-effect free) when it has no visible side effects. Knowing which methods are pure has a number of practical applications. It can facilitate compiler optimization [6, 16, 31], model checking [26], Universe types inference [9], memorization of function calls [14] and so on.

We adopt the definition of purity given by Sălcianu and Rinard [25]: a method is *pure* if it does not mutate any object that exists in *prestates*. Thus, a method is pure if (1) it does not mutate prestates reachable through parameters, and (2) it does not mutate prestates reachable through static fields. The definition allows a pure method to create and mutate local objects, as well as return a newly constructed object as a result.

For a method that does not access static fields, the prestates it can reach are the objects reachable from the actual arguments and the method receiver. Therefore, if any of the formal parameters of m or implicit parameter this, is inferred as mutable by reference immutability inference, m is impure. Otherwise, i.e., if none of the parameters is inferred as mutable, m is pure. Consider the implementation of List in the left column of Figure 4. For method add, reference immutability inference infers that both n and this are mutable, i.e. the objects referred by them are mutated in add. When there is a method invocation lst.add(node), we know that the prestates referred by the actual argument node and the receiver lst are mutated. As a result, we can infer that method add is impure. We can also infer that method reset is impure because implicit parameter this is inferred as mutable by reference immutability inference. Method size is inferred as pure because its implicit parameter this is inferred as readonly and it has no formal parameters.

However, the prestates can also come from static fields. A method is impure if it mutates (directly, or indirectly through callees), a static field, or objects reachable from a static field. We introduce a *static immutability type* q_{m} for each method

```
class List {
  Node head;
  int len;
  void add(Node n) {
    n.next = this.head;
    this.head = n;
    this.len++;
  }
  void reset() {
    this.head = null;
    this.size = 0;
  }
  int size() {
    return this.len;
  }
}
```

```
class Main {
  static List sLst;
  void m1() {
    List lst = ...
    Node node = ...
    lst.add(node);
    Main.sLst = lst;
  }
  void m2() {
    int len = sLst.size();
    PrintStream o = System.out;
    o.print(len);
  }
  void m3() {
    m2();
  }
}
```

Figure 4. A simple linked list

m. q_{m} can be readonly or mutable but not polyread. Roughly, q_{m} is mutable when m accesses static state through some static field, and then mutates this static state; q_{m} is readonly otherwise. Static immutability types are computed using reference immutability. We introduce a function *statictypeof* which retrieves the static immutability type of m:

$$\mathit{statictypeof}(\mathsf{m}) = q_{\mathsf{m}}$$

We extend the program syntax with two additional statements (TSWRITE) sf = x for static field write, and (TSREAD) x = sf for static field read. Here x denotes a local variable and sf denotes a static field.

Figure 5 extends the typing rules from Figure 2 with constraints on static immutability types. If method m contains a static field write sf = x, then its static immutability type is mutable (see rule (TSWRITE)). If m contains a static field read x = sf where x is inferred as mutable or polyread, q_{m} becomes mutable as well (see rule (TSREAD)). While the handling of (TSWRITE) is expected, the handling of (TSREAD) may be unexpected. If sf is read in m, using x = sf, then m or one of its callees can access and mutate the fields of sf through x. If m or one of its callees writes a field of sf through x, then x will be mutable. If m does not write x, but returns x to a caller, which subsequently writes a field of sf, then x will be polyread. x being readonly guarantees that x is immutable in the scope of m and after m's return, and sf is not mutated through x. Note that aliasing is handled by the type system which disallows assignment from readonly to mutable or polyread. Consider the code:

```
void m() {
  ...
  x = sf; // a static field read
  y = x.f;
  z = id(y);
  z.g = 0;
```

$$\frac{methodof(\mathsf{sf} = \mathsf{x}) = \mathsf{m} \quad statictypeof(\mathsf{m}) = q_{\mathsf{m}} \quad q_{\mathsf{m}} = \mathsf{mutable}}{\Gamma \vdash \mathsf{sf} = \mathsf{x}} \text{(TSWRITE)}$$

$$\frac{methodof(\mathsf{x} = \mathsf{sf}) = \mathsf{m} \quad statictypeof(\mathsf{m}) = q_{\mathsf{m}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{m}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{sf}} \text{(TSREAD)}$$

$$\frac{methodof(\mathsf{x} = \mathsf{y.m(z)}) = \mathsf{m}' \quad statictypeof(\mathsf{m}) = q_{\mathsf{m}} \quad statictypeof(\mathsf{m}') = q_{\mathsf{m}'} \quad q_{\mathsf{m}'} <: q_{\mathsf{m}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}} \text{(TCALL)}$$

Figure 5. Extended typing rules for static fields (See Figure 2 for the base type system). Function $methodof$ returns the enclosing method of the statement. $statictypeof(\mathsf{m})$ returns the static immutability type of m. Static immutability types can be readonly or mutable, but not polyread.

```
  ...
}
```

Here static field sf has its field f aliased to local z, which is mutated. The type system propagates the mutation of z to x; thus, the constraints in Figure 5 set the static immutability type of m to mutable. In addition, q_{m} becomes mutable if the static immutability type of one of its callees is mutable (i.e., a static field may be mutated in a callee of m). This is expressed by rule (TCALL).

Method overriding is handled by an additional constraint. If m' overrides m we must have

$$q_{\mathsf{m}} <: q_{\mathsf{m}'}$$

In other words, if m' mutates static state, q_{m} must be mutable, even if m itself does not mutate static state. This constraint ensures that m' is a behavioral subtype of m and is essential for modularity.

Static immutability types are inferred in the same fashion as reference immutability types. The analysis initializes every $S(\mathsf{m})$ to $\{\mathsf{readonly}, \mathsf{mutable}\}$ and iterates over the statements in Figure 5 and the overriding constraints, until it reaches the fixpoint. If readonly remains in $S(\mathsf{m})$ at the end, the static immutability type of m is readonly; otherwise, it is mutable.

Consider the right column of Figure 4. q_{m1} becomes mutable because m1 assigns lst to the static field sLst. q_{m2} is mutable as well, because it mutates the PrintStream object referred by System.out by invoking the print method on it, and local variable o is mutable. q_{m3} becomes mutable as well, because it invokes method m2 and q_{m2} is mutable.

The observant reader has likely noticed that $q_{\mathsf{m}} = \mathsf{mutable}$ does not account for all mutations of static state in m. In particular, static state may be aliased to parameters and be accessed and mutated in m through parameters:

```
void m(X p) {
  p.g = 0;
}
...
void n() {
  X x = sf; // a static field read (TSREAD)
  m(x);
}
```

In the above example, q_{m} is readonly, even though m mutates static state. Interestingly, this is not unsound. Parameter and static mutability types capture precisely the information needed to infer purity as we shall see shortly.

We infer that a method m is pure if all of its parameters, including implicit parameter this are not mutable (i.e., they are readonly or polyread), and its static immutability type is not mutable (i.e., it is readonly). More formally, let $typeof(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}}$ and $statictypeof(\mathsf{m}) = q_{\mathsf{m}}$. We have:

$$pure(\mathsf{m}) = \begin{cases} \textbf{false} & \textbf{if } q_{\mathsf{this}} = \mathsf{mutable} \textbf{ or} \\ & \quad q_p = \mathsf{mutable} \textbf{ or} \\ & \quad q_{\mathsf{m}} = \mathsf{mutable} \\ \textbf{true} & \textbf{otherwise} \end{cases}$$

As discussed earlier, a method m can be impure because: (1) prestates are mutated through parameters, or (2) prestates are mutated through static fields. If prestates are mutated through parameters, then this will be captured by the mutability of this and p. Now, suppose that prestates are *not* mutated through parameters, but are mutated after access through a static field. In this case, there must be an access in m to a static field sf through (TSREAD) or (TSWRITE), and the mutation is captured by the static immutability type q_{m}.

4. Experiments

The inference of reference immutability, and the type checker that verifies the inferred types, are implemented in the Checker Framework (CF) [11, 19]. The purity inference is implemented on top of the CF as well. The tool called ReImInfer is publicly available at http://www.cs.rpi.edu/~huangw5/cf-inference/.

4.1 Benchmarks

The implementation is evaluated on 13 large Java benchmarks, including 4 whole-program applications and 9 Java libraries.

Whole programs:

- **Java Olden** (JOlden) is a benchmark suite of 10 small programs.
- **ejc-3.2.0** is the Java Compiler for the Eclipse IDE.
- **javad** is a Java class file disassembler.
- **SPECjbb 2005** is SPEC's benchmark for evaluating server side Java.

Libraries:

- **tinySQL-1.1** is a database engine.[1]
- **htmlparser-1.4** is a library for parsing HTML.
- **jdbm-1.0** is a lightweight transactional persistence engine.
- **jdbf-0.0.1** is an object-relational mapping system.
- **commons-pool-1.2** is a generic object-pooling library.
- **jtds-1.0** is a JDBC driver for Microsoft SQL Server and Sybase.
- **java.lang** is the package from JDK 1.6
- **java.util** is the package from JDK 1.6.
- **xalan-2.7.1** is a library for transforming XML documents to HTML from the DaCapo 9.12 benchmark suite.

We run our inference tool, called ReImInfer, on the above benchmarks on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (the maximal heap size is set to 2 GB). The software environment consists of Sun JDK 1.6 and the Checker Framework 1.1.5 on GNU/Linux 2.6.38.

4.2 Reference Immutability Inference

In this section, we present our results on reference immutability inference. We treat the this parameters of java.lang.Object's hashCode, equal and toString as readonly, even though these methods may mutate internal fields (these fields are used only for caching and can be excluded from the object state). This handling is consistent with the notion of *observational purity* discussed in [3] as well as other related analyses such as JPPA [25]; these methods are intended to be observationally pure. Our analysis does not detect bugs due to unintended mutation in these methods.

ReImInfer treats private fields f that are read or written through this in exactly one method m, as if they were local variables. Precisely, this means that for these fields we allow qualifier polyread, and treat field reads $x = this.f$ and writes $this.f = x$ as if they were assignments $x = f$ and $f = x$. One such field and method are current and nextElement() in class Enumerate shown in Figure 7. We preserve the dependence between this and f, by using an additional constraint: $q_{this} <: q_f$. Thus, when f is mutated in m, f and this are inferred as mutable. When f is readonly in the scope of m, but depends on the context of the caller, f is polyread and this is polyread or mutable. Otherwise, f is readonly. As an example, current and this of nextElement() in Figure 7 are both inferred polyread. The motivation behind this optimization is precisely the Enumeration class in Figure 7. The goal is to transfer the dependence from the element stored in the container, to the container itself, which is important for purity inference. If current was treated as a field, it would be mutable, and therefore, this of elements would be mutable, which entails that every container that creates an enumeration is mutable, even if its elements were not mutated. If current was excluded from abstract sate, then this of nextElement would have been readonly and mutation from elements would not have been transferred to the container. Our optimization allows this of nextElement and elements to be polyread, which is important for purity inference, as we discuss shortly. The optimization affected 8 nextElement and elements methods and 12 other methods that call nextElement and elements throughout all of our benchmarks.

Recall that reference immutability inference is modular. Thus, it is able to analyze any given set of classes L. If there are unknown callees in L, the analysis assumes default typing $\mathsf{mutable}, \mathsf{mutable} \rightarrow \mathsf{polyread}$. The mutable parameters assume worst-case behavior of the unknown callee — the unknown callee mutates its arguments. The polyread return is an appropriate choice because readonly would have been too restrictive for the caller. User code U, which uses previously analyzed library L, is analyzed separately using the result of the analysis of L. In our case, when analyzing user code U, we use the annotated JDK available with Javarifier in CF; the similarities between Javari and ReIm justify this use. Correctness of the composition is ensured by the check that the function subtyping constraints hold: for every m' in U that overrides an m from L, $typeof(m') <: typeof(m)$ must hold. For example, suppose that L contains code x.m() where $this_m$, is inferred as readonly. The typing is correct even in the presence of callbacks. If x.m() results in a callback to m' in U (m' overrides m), constraint $typeof(m') <: typeof(m)$ which entails $this_m <: this_{m'}$, ensures that $this_{m'}$ is readonly as well.

Of course, it is possible that U violates the subtyping expected by L. Interestingly however, in our experiments the only violations were on special-cased methods of Object: equals, hashCode and toString. Furthermore, the vast majority of violations occurred in the java.util library. As with other analyses (JPPA), we report these violations as warnings.

Below, we present our results on inference of reference immutability. Sections 4.2.1–4.2.3 evaluate our tool in terms of scalability and precision.

4.2.1 Inference output

Table 1 presents the result of running our inference tool ReImInfer on all benchmarks.

In all benchmarks, about 41% to 69% of references are reported as readonly, less than 10% are reported as polyread and 27% to 55% are reported as mutable.

To summarize our findings, ReImInfer is more scalable than Javarifier (Section 4.2.2). Furthermore, ReImInfer produces equally precise results (Section 4.2.3).

4.2.2 Timing results

Figure 6 compares the running times of ReImInfer and Javarifier on the first 5 benchmarks in Table 1. ReImInfer and Javarifier analyze exactly the same set of classes (given

[1] We added 392 empty methods in tinySQL in order to compile it with Java 1.6.

Benchmark	Code size			Annotatable References				Time (in seconds)		
	#Line	#Meth	#Pure	#Ref	#Readonly	#Polyread	#Mutable	Infer	Purity	Total
JOlden	6223	326	175 (54%)	949	453 (48%)	95 (10%)	401 (42%)	6.6	0.15	9.6
tinySQL	31980	1597	965 (60%)	4247	2644 (62%)	287 (7%)	1316 (31%)	14.9	0.54	22.1
htmlparser	62627	1698	642 (38%)	4853	2711 (56%)	230 (5%)	1912 (39%)	16.6	1.24	25.9
ejc	110822	4734	1701 (36%)	15434	6163 (40%)	1106 (7%)	8165 (53%)	100.2	1.83	150.6
xalan	348229	10386	3942 (38%)	41192	25233 (61%)	2105 (5%)	13854 (34%)	172.8	6.65	220.7
javad	4207	140	60 (43%)	363	249 (69%)	3 (1%)	111 (31%)	2.3	0.16	4.1
SPECjbb	28333	529	195 (37%)	1537	836 (54%)	98 (6%)	603 (39%)	11.7	0.18	15.4
commons-pool	4755	275	94 (34%)	602	266 (44%)	3 (0%)	333 (55%)	3.4	0.11	5.4
jdbm	11610	446	136 (30%)	1161	468 (40%)	88 (8%)	605 (52%)	6.6	0.19	9.6
jdbf	15961	707	304 (43%)	2510	1677 (67%)	161 (6%)	672 (27%)	9.5	0.26	13.9
jtds	38064	1882	671 (36%)	5048	2817 (56%)	144 (3%)	2087 (41%)	16.6	1.08	26.0
java.lang	43282	1642	1101 (67%)	2970	1957 (66%)	109 (4%)	904 (30%)	12.5	0.39	17.8
java.util	59960	2727	1027 (38%)	6925	2805 (41%)	772 (11%)	3348 (48%)	28.1	0.77	42.0

Table 1. Inference results of reference immutability. **#Line** shows the line number of the benchmarks, including blank lines and comments. **#Meth** gives the number of methods of the benchmarks. **#Pure** is the number of pure methods inferred by our purity analysis. *Annotatable References* include all references, including fields, local variables, return values, formal parameters and implicit parameters this. It does not include references of primitive type. **#Ref** is the total number of annotatable references, **#Readonly**, **#Polyread** and **#Mutable** are the number of references inferred as readonly, polyread and mutable, respectively. We also include the running time for the benchmarks. **Infer** is the running time of ReImInfer for inferring reference immutability. **Purity** is the time for inferring method purity based on the result of reference immutability. The last column **Total** shows the total running time, including source processing, reference immutability inference, purity inference and type checking.

at the command-line), and use stubs for the JDK. That is, both ReImInfer and Javarifier generate and solve constraints for the exact same set of classes, and neither analyzes the JDK.

ReImInfer scales better than Javarifier. In contrast, ReImInfer appears to scale approximately linearly. As the applications grow larger, the difference between ReImInfer and Javarifier becomes more significant. This results are consistent with the results reported by Quinonez et al. [22] where, Javarifier posts significant nonlinear growth in running time, when program size goes from 62kLOC to 110kLOC.

4.2.3 Correctness and precision evaluation

To evaluate the correctness and precision of our analysis, we compared our result with Javarifier on the first four benchmarks from Table 1. We do not compare the numbers directly because we use a different notion of annotatable reference from Javarifier (e.g., Javarifier counts List<Date> twice while we only count it once). In our comparison, we examine only fields, return values, formal parameters and this parameters; we call these references *identifiable references*. We exclude local variables because Javarifier does not give identifiable names for local variables (it only shows local 0, local 1 and so on).

JOlden We examined all programs in the Java Olden (JOlden) benchmark suite. We found 34 differences between our result and Javarifier's, out of 758 identifiable references. We exclude the following difference from the count: the this parameters of constructors are as readonly by Javarifier, while they are reported as mutable if this is mutated, by our inference. Differences due to the annotated JDK are also excluded because Javarifier treated variables from library methods as mutable even though we have specified the annotated JDK. 8 out of the 34 differences are the nextElement() method that implements the Enumeration interface (Figure 7). Javarifier infers the return value as readonly. This is correct with respect to the semantics of Javari and Javarifier, which separates a structure from the elements stored in it; thus, a mutation on

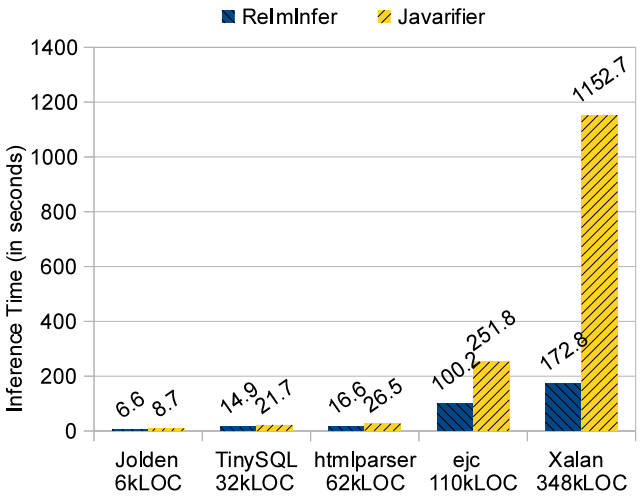

Figure 6. Runtime Performance Comparison. Note that the running time for purity inference and type checking is excluded for ReImInfer.

```
public class Body{
  Body next;
  public final Enumeration elements() {
    class Enumerate implements Enumeration {
      private Body current;
      public Enumerate() { current = Body.this; }
      public Object nextElement() {
        Object retval = current;
        current = current.next;
        return retval;
      }
    }
    return new Enumerate();
  }
}
```

Figure 7. The elements() method in JOlden/BH

an element, should not necessarily affect the data structure itself.

The semantics of ReIm and ReImInfer demands that the return of nextElement should be polyread, because there are cases when the retrieved element is mutated. ReImInfer reports that nextElement()'s return is polyread. Also Javarifier infers the this parameter of nextElement() as mutable while ReImInfer reports that it is polyread. This is possible because ReImInfer treats field current in Figure 7 as a local variable as discussed earlier. There are 4 nextElement() methods in the JOlden benchmark suite, causing 8 differences in total.

These 8 differences directly or indirectly lead to the remaining 26 differences. First, these 8 differences directly lead to 8 differences in the current field and the elements() method in the Enumerate class, which is shown in Figure 7. Our analysis infers retval as polyread because the return value of nextElement() is polyread as discussed earlier. This causes field current to be inferred as polyread in statement Object retval=current since current is a field but treated as a local variable. As a result, the this parameter of elements() becomes polyread due to the assignment current=Body.this. Because Javarifier infers the return value of nextElement() as readonly, it reports both the current field and the this of elements() are readonly, which leads to 8 differences in total. The treatment of Javarifier reflects the expected semantics of Javari — the container that calls elements should not be affected by the data stored in it. ReIm and ReImInfer's semantics demands that a mutation on the element is propagated to the container.

Second, these 8 differences on current and elements() propagate to the other 18 differences. The following code shows an example:

```
Body bodyTab = ...;
for(Enumeration e = bodyTab.elements();
                        e.hasMoreElements();){
  Body b = (Body)e.nextElement();
  ...
  b.setProcNext(prev);
}
```

Here b is mutable since the this parameter of setProcNext(Body) is mutable. Because bodyTab is indirectly assigned to b through the Enumeration instance referred by e, bodyTab should be mutable as well. Javarifier reports bodyTab is readonly because the this parameter of elements() is inferred as readonly. ReImInfer reports bodyTab as mutable. This is important for purity — e.g., if bodyTab is a parameter, its mutability entails that the enclosing method is impure.

Other benchmarks For the remaining three benchmarks, tinySQL, htmlparser and ejc, we examined 4 randomly selected classes from each (a total of 12 classes). We found 2 differences out of 868 identifiable references. The 2 differences are caused by the fact that Javarifier infers a parameter of String type as polyread, which causes an actual argument to become polyread or mutable; ReImInfer infers this parameter as readonly.

Overall, the differences are very minor. Most are attributable to the different semantics of ReIm and Javari, and the few others are due to an apparent bug in a corner case of Javarifier's handling of the annotated JDK.

4.3 Purity Inference

This section presents our results on purity inference. We treat methods equals, hashCode, toString in java.lang.Object, as well as java.util.Comparable.compareTo, as observationally pure. This is analogous to previous work [25].

Our purity inference is modular. Reference immutability assumptions for unknown callees are exactly as before. Static immutability types, which we discussed in Section 3, are not available in Javari's annotated JDK. We ran ReImInfer on the java.lang and java.util packages, and we assumed that other library methods have not mutated static fields. JPPA, a Java Pointer and Purity Analysis tool by Sălcianu and Rinard [25], makes the same assumption for unknown library methods, and our decision to use q_{m} = readonly as default, is motivated by this, in order to facilitate comparison with JPPA.

When composing previously analyzed libraries L with user code U for purity inference, we need one additional check: for every m' in U that overrides m in L, we must have $q_{\mathsf{m}} <: q_{\mathsf{m}'}$. In particular, if q_{m} is inferred as readonly, then $q_{\mathsf{m}'}$ must be readonly as well. As with reference immutability, it is possible that user code violates this constraint. In the first 11 benchmarks in Table 1, we found 205 out of 22,720 user methods that violate the inferred $statictypeof$ on java.lang and java.util packages, and the vast majority of the violations are on the special-cased methods, equals, hashCode and toString. These violations are reported as warnings.

The results of purity inference by ReImInfer are shown in Table 1, column **#Pure**. To evaluate analysis precision, we compared with JPPA by Sălcianu and Rinard [25] and JPure by Pearce [20]. We ran JPPA and JPure on the JOlden

Program	#Meth	JPPA	JPure	ReImInfer
BH	69	20 (29%)	N/A	33 (48%)
BiSort	13	4 (31%)	3 (23%)	5 (38%)
Em3d	19	4 (21%)	1 (5%)	8 (42%)
Health	26	6 (23%)	2 (8%)	11 (42%)
MST	33	15 (45%)	12 (36%)	16 (48%)
Perimeter	42	27 (64%)	31 (74%)	38 (90%)
Power	29	4 (14%)	2 (7%)	10 (34%)
TSP	14	4 (29%)	0 (0%)	1 (7%)
TreeAdd	10	1 (10%)	1 (10%)	6 (60%)
Voronoi	71	40 (56%)	30 (42%)	47 (66%)

Table 2. Pure methods in Java Olden benchmarks

benchmark suite and directly compared its output with ours. Table 2 presents the comparison result.

To summarize our results, ReImInfer scales well to large programs and shows good precision compared to JPPA and JPure. Furthermore, ReImInfer, which is based on the stable and well-maintained CF, appears to be more robust than JPPA and JPure, both of which are based on custom compilers. These results suggest that ReImInfer can be useful in practice as a wide variety of clients require purity analysis.

4.3.1 Comparison with JPPA

JOlden There are 59 differences out of 326 user methods between ReImInfer's result and JPPA's. Of these differences, (a) 4 are due to differences in definitions/assumptions, (b) 51 are due to limitations/bugs in JPPA and (c) 4 are due to limitations in ReImInfer.

4 differences are due to JPPA's assumption about unknown library methods. For example, JPPA reports as pure the method median in Jolden/TSP, which invokes new java.lang.Random(). The constructor Random should not be pure because it mutates a static field seedUniquifier. ReImInfer precomputes static immutability types q_{m} on the JDK library and thus reports method median as impure.

51 differences are due to limitations/bugs of JPPA. 38 differences are the constructors, which ReImInfer reports as pure but JPPA does not. According to [25], JPPA follows the JML convention and constructors that mutate only fields of the this object are pure. Thus, JPPA should have inferred them as pure. ReImInfer follows the same definition and reports these constructors as pure. There are 3 differences on methods that are inferred as pure by ReImInfer but impure by JPPA. These 3 methods that return newly-constructed objects, which are mutated later. According to the definition in [25], JPPA should have inferred them as pure. There is 1 difference on method loadTree in Jolden/BH. It is likely a bug in JPPA because the this parameter is passed to another object's field which is mutated later, but JPPA reports loadTree as pure. ReImInfers detects the this parameter is mutated and reports the method as impure. There are 9 methods reported as pure by ReImInfer but not covered by JPPA. This is because JPPA is a whole-program analysis and these methods are not reachable, resulting in 9 differences in the comparison.

The remaining 4 differences are the nextElement method discussed in Section 4.2.3. Because ReImInfer considers the current field as a local variable, it infers these 4 methods as pure while JPPA considers they are impure.

Other benchmarks We attempted to run JPPA and compare on benchmarks tinySQL, htmlparser and ejc as we did with Javarifier. tinySQL is a library and there is no main method. htmlparser, which is a library as well, comes with a main, which exercises a portion of its functionality; JPPA threw an exception on htmlparser which we were unable to correct. JPPA completed on ejc. Due to the fact that it is a whole-program analysis, it analyzed 3790 reachable user methods; ReImInfer covered all 4734 user methods.

We examined 4 randomly selected classes from ejc and found 22 differences out of 163 methods in total. 9 methods are not reachable according to JPPA. Of the remaining 13 differences, (a) 2 are due to limitations/bugs in JPPA and (b) 11 are due to limitations/bugs in ReImInfer. 1 constructor that should have been pure according to the JML convention was reported as impure by JPPA. In addition, 1 method which we believe is pure because it does not mutate any prestate, was reported as impure by JPPA. The remaining 11 methods are reported as pure by JPPA but impure by ReImInfer; this is imprecision in ReImInfer. 6 of these methods are inferred as impure by ReImInfer because they are overridden by impure methods. This is an insurmountable imprecision for ReImInfer. ReImInfer reports the other 5 methods as impure because they return static fields which are mutated later. This imprecision can be corrected by allowing q_{m} to have value polyread. In the current formulation and implementation of purity inference, for simplicity, we allow q_{m} to be only mutable or readonly. We will address this issue in the next version of ReImInfer.

4.3.2 Comparison with JPure

JOlden There are 60 differences out of 257 user methods between ReImInfer's result and JPure's, excluding the BH program (JPure could not compile BH). Of these, (a) 29 differences are caused by different definitions/assumptions, (b) 2 are caused by limitations/bugs in ReImInfer, and (c) 29 differences are caused by limitations/bugs in JPure.

29 differences are caused by different definitions of pure constructors. We follow the JML convention that a constructor is pure if it only mutates its own fields. JPure has different definition of a pure constructor and that leads to these differences. 2 differences are the nextElement method where ReImInfer considers the current field as a local variable as discussed above. There are 8 differences in toString methods, which are inferred as impure by JPure. Our examination shows that those methods are pure; it appears that they should be pure, but are inferred as impure due to imprecision in JPure, according to [20]. 16 differences are caused by meth-

ods that return fresh local references. JPure should have been able to identify them as @Fresh, but it did not. The remaining 5 differences are due to the static methods in java.lang.Math. JPure infers all methods that invoke the static methods in java.lang.Math as impure, while ReImInfer identifies that these methods satisfy q_{m} is readonly by using the inference result from the java.lang package.

Other benchmarks We attempted to run JPure on the libraries from JDK 1.6, but that caused problem with the underlying compiler in JPure. We attempted to run JPure on tinySQL, htmlparser and ejc. In all three cases, the tool issued an error. We were unable to perform direct comparison on larger benchmarks.

5. Related Work

We begin by comparison with Javari [27] and its inference tool Javarifier [22], which represent the state-of-the-art in reference immutability. Although the type systems have similarities, they also differ in important points of design and implementation. The corresponding inference tools implement substantially different inference algorithms. Section 5.1 compares our type system with Javari, Section 5.2 compares our inference approach with Javarifier. Section 5.3 discusses related work on purity inference, and Section 5.4 discusses other related works.

5.1 Comparison with Javari

There are two essential differences between our type system and Javari [27]. First, Javari allows programmers to exclude fields from abstract state by designating fields as assignable. Therefore, the assignable field may be assigned or mutated even through a readonly reference. An example is a field used for caching (e.g., hashCode) — modifying it should not be considered mutation from the client's point of view. As expected however, this expressive power complicates Javari: to prevent converting an immutable reference to a mutable reference, Javari requires the access to an assignable field through a readonly reference, to have different mutabilities depending on whether it is an l-value or an r-value of an assignment expression. ReIm does not allow assignable fields and therefore it is simpler. This decision is motivated by our intended application: purity inference. Including assignable in the type system would have complicated purity inference.

Second, Javari treats generics and arrays differently. Javari permits annotating the type arguments when instantiating a parametric class but disallows annotating the type parameters (as mutable) within a parameterized class. ReIm handles generics in the opposite way. It allows the type parameters to be annotated but disallows annotations on the type arguments when instantiating a parametric class. The difference between the two approaches is illustrated by the following example:

```
List<Date> lst1 = new List<Date>();
lst1.add(new Date());
List<Date> lst2 = lst1;
lst2.get(0).setHours(1);
```

Here Javari's inference tool (Javarifier) infers that reference lst2 is of type readonly List<mutable Date>. ReImInfer annotates lst2 as mutable List<Date>. There are advantages and disadvantages in both approaches. Javari permits the user to specify the mutability of a generic data structure as the same or different from its contents (whose type is specified by the type parameter). ReImInfer is simpler: the mutability of the data structure, lst2 in this example, is the same as the mutability of the data stored in it. (Javarifier does not have an option to make it prefer this solution.) Again, the primary motivation for the decision about ReIm's design is the application we had in mind: purity inference. For purity, the data structure must be considered along with its data. For example, if lst2 was a parameter in m, and m used setHours() on one of its elements, m must be reported as impure.

The mutability of a reference as needed for purity inference, cannot be deduced from the mutabilities of the top-level reference and its type arguments. Consider the following example:

```
class A<T> {
  T id(T p) { return p; }
}
A<Date> x = new A<Date>();
Date d = x.id(new Date());
d.setHours(0);
```

Here Javarifier infers that x is of type readonly A<mutable Date>. If we take into account the mutable type argument, we will conclude that x is mutable. For our purposes, x is readonly, because the type argument is not part of the state of the object. ReIm and ReImInfer, which are designed with purity analysis in mind, annotate x as readonly.

Arrays are treated similarly to generics in Javari and its inference tool. In the following code b would be annotated as mutable Date readonly [].

```
Date[] a = new Date[1];
a[0] = new Date();
Date[] b = a;
b[0].setHours(2);
```

Again, Javari and Javarifier separate the array from its elements. ReIm and ReImInfer treat the elements of arrays as fields (an array is considered along with the data stored in it), so the array reference b would be inferred as mutable Date mutable [] due to the mutation of element 0.

Another important (but non-essential for our purpose) difference between Javari and ReImInfer is the type qualifier hierarchy.

5.2 Comparison with Javarifier

Our inference approach is comparable to Javarifier, the inference tool of Javari. Both tools use flow-insensitive and context-sensitive analysis and solve constraints generated during type-based analysis. There are three substantial differences between the tools.

The most significant difference is in the context-sensitive handling of methods. The main idea of Javarifier is to create two context copies for each method that returns a reference, one copy for the case when the left-hand-side of the call is mutable, and another copy for the case when the left-hand-side is readonly. As a result, Javarifier doubles the total number of method-local references, including local variables, return values, formal parameters and implicit parameters this. It also doubles the number of constraints. In contrast, our inference uses polyread and viewpoint adaptation, which efficiently captures and propagates dependences from parameters to return values in the callee, to the caller. For example, in m() { x = this.f; y = x.g; return y; }, the polyread of the return value is propagated to implicit parameter this; the dependence is transferred to the callers when viewpoint adaptation is applied at the call sites of m.

Second, Javarifier and ReImInfer have different constraint resolution approaches. Javarifier computes graph reachability over the constraint graph. Its duplication of nodes in its constraint graph correctly handles context sensitivity. In contrast, ReImInfer uses fixpoint iteration on the set-based solution and outputs the final typing based on the preference ranking over the qualifiers.

Third, Javarifier is based on Soot [28] while ReImInfer is based on the Checker Framework, which did not yet exist when Javarifier was developed. Javari's type-checker is completely separate code from Javarifier, and Javarifier also requires an additional utility to map the inference result back to the source code in order to do type checking. In total, Javari and Javarifier depend on three tools: Soot, the annotation utility, and CF. In contrast, ReImInfer and the type checker are seamlessly integrated in the CF. In addition, it is difficult to incorporate programmer-provided annotations in Javarifier — annotations have to be written in a separate annotation file. ReImInfer seamlessly integrate programmer-provided annotations from the source file: for example, if the programmer decides to annotate a variable as readonly, the inference initializes the set for this variable to {readonly}, instead of the default {readonly, polyread, mutable}. These differences contribute to the usability of ReImInfer.

We conjecture that viewpoint adaptation, the constraint resolution approach and the better infrastructure in CF, contribute to the better scalability of ReImInfer compared to Javarifier.

5.3 Purity

Sălcianu and Rinard present a Java Pointer and Purity Analysis tool (JPPA) for reference immutability inference and purity inference. Their analysis is built on top of a combined pointer and escape analysis. Their analysis not only infers the immutability, but also the safety for parameters, which means the abstract state referred by a safe parameter will not be exposed to externally visible heap inside the method. However, the pointer and escape analysis is more expensive. It relies on whole program analysis, which requires main, and analyzes only methods reachable from main. ReImInfer does not require the whole program and thus it can be applied to libraries. Plus, we also include a type checker for verifying the inference result, which is not available in JPPA.

JPure [20] is a modular purity system for Java. The way JPure infers method purity is not based on reference immutability inference, as our purity inference and JPPA did. Instead, it exploits two properties, *freshness* and *locality*, for purity analysis. Its modular analysis enables inferring method purity on libraries and gains efficient runtime performance.

Rountev's analysis is designed to work on incomplete programs using fragment analysis by creating artificial main routine [24]. However, its definition of pure method is more restricted in that it disallows a pure method to create and use a temporary object.

Clausen develops Cream, an optimizer for Java bytecode using an inter-procedural side-effect analysis [6]. It infers an instruction or a collection of instructions as pure, read-only, write-only or read/write, based on which it can infer purity for methods, loops and instructions. It is a whole-program analysis which relies the main method and also unused methods are not covered.

Other researchers also explore the dynamic notion of purity. Dallmeier et al. develop a tool, also called JPURE, to dynamically infer pure methods for Java [8]. Their analysis calculates the *set of modified objects* for each method invocation and determines impure methods by checking if they write non-local visible objects. Xu et al. use both static and dynamic approaches to analyze method purity in Java programs [30]. Their implementation supports different purity definitions that range from strong to weak. These dynamic approach depends on the runtime behavior of programs, which is totally different from our purity analysis.

5.4 Other Related Work

Artzi et al. present Pidasa for classifying parameter reference immutability [1, 2]. They combine dynamic analysis and static analysis in different stages, each of which refines the result from the previous stage. The resulting analysis is scalable and produces precise result. They also incorporate optional unsound heuristics for improving precision. In contrast, our analysis is entirely static and it also infers immutability types for fields and method return values. It is unclear how their analysis handles polymorphism of methods.

JQual [13] is a framework for inference of type qualifiers. JQual's immutability inference in field-sensitive and context-sensitive mode is similar to Javari's inference. However, it is not scalable in this mode according to the authors. And Artzi et al.'s evaluation confirms this [2]. In field-insensitive mode, JQual suffers from the problem that the method receiver has to be mutable when the method reads a mutable field, even if the method itself does not mutate any program state. Our analysis is scalable and may even have better scalability than Javarifier. Also, by introducing the polyread annotation and viewpoint adaptation, our analysis is able to correctly infer

that a method receiver is readonly or polyread, even if a field is returned from the method, and the returned value is mutated later.

Poral et al. [21] present an analysis that detects immutable static fields and also addresses sealing/encapsulation. Their analysis is context-insensitive and libraries are not analyzed. Liu and Milanova [17] describe field immutability in the context of UML. Their work incorporate limited context sensitivity, analyze large libraries and focuses on instance fields. This work is an improvement over [17]. Immutability inference not only includes instance fields, but also local variables, return values, formal parameters and this parameters. Also, this work provides a type checker to verify the correctness of the inference result.

Chin et al. [4] propose CLARITY for the inference of user-defined qualifiers for C programs based on user-defined rules, which can also be inferred given user-defined invariants. It infers several type qualifiers, including pos and neg for integers, nonnull for pointers, and tainted and untainted for strings. These type qualifiers are not context-sensitive. In contrast, our tool focuses on the type system for reference immutability and it is context-sensitive, as viewpoint adaptation is used in the type system to express context sensitivity.

Our type system uses and adapts the concept of viewpoint adaptation from Universe Types [7, 9, 10], which is a lightweight ownership type system that optionally enforces the *owner-as-modifier* encapsulation discipline. The readonly qualifier in our system is similar to the any qualifier in Universe Types (in earlier work on Universe Types, qualifier any is actually called readonly). Both readonly references and any references disallow mutations on their referents. However, the ownership structure in Universe Types can be used to give a more concrete interpretation of casts from a readonly type to a mutable type.

In addition, the purity results from this work can be used in the inference of Universe Types, as shown by our previous work [15]. The type inference algorithm presented in this paper, fits in the framework from [15]. One difference is that viewpoint adaptation in [15] is the traditional viewpoint adaptation from Universe types: it uses the same operation at field accesses and at method calls, and adapts only from the point of view of the receiver. In this paper, we use a more general notion of viewpoint adaptation. The precise relation between [15] and this work, will be formalized in future work.

6. Conclusion

We have presented ReIm and ReImInfer, a type system and a type inference analysis for reference immutability. In addition, we have applied reference immutability to method purity inference. We have shown that our approach is scalable and precise by implementing a prototype, evaluating it on 13 large Java programs and Java libraries, and comparing the results to the leading reference immutability inference tool, Javarifier, and to method purity inference tools, JPPA and JPure.

References

[1] S. Artzi, A. Kieżun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, pages 104–113, 2007.

[2] S. Artzi, A. Kieżun, J. Quinonez, and M. D. Ernst. Parameter reference immutability: formal definition, inference tool, and comparison. *Automated Software Engineering*, 16(1):145–192, Dec. 2009.

[3] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *FTfJP*, pages 11–19, 2004.

[4] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, pages 264–278, 2006.

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

[6] L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9:1031–1045, 1997.

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

[8] V. Dallmeier, C. Lindig, and A. Zeller. Dynamic purity analysis for java programs. `http://www.st.cs.uni-saarland.de/models/jpure/`, 2007.

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

[10] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, pages 28–53, 2007.

[11] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In *ICSE*, pages 681–690, 2011.

[12] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, Sept. 12, 2008.

[13] D. Greenfieldboyce and J. S. J. Foster. Type qualifier inference for java. In *OOPSLA*, pages 321–336, 2007.

[14] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *PLDI*, pages 311–320, 2000.

[15] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and Checking of Object Ownership. In *ECOOP*, 2012. to appear.

[16] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *CC*, pages 287–304, 2005.

[17] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-Based Object Access Control. In *ICSE*, pages 323–332, 2007.

[18] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[19] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.

[20] D. Pearce. JPure: a modular purity system for Java. In *CC*, pages 104–123, 2011.

[21] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic Detection of Immutable Fields in Java. In *CASCON*, pages 10–24, 2000.

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

[23] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22:162–186, 2000.

[24] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, 2004.

[25] A. Sălcianu and M. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI*, pages 199–215, 2005.

[26] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, pages 188–197, 2003.

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

[28] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *CASCON*, pages 13–, 1999.

[29] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.

[30] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *PASTE*, pages 75–82, 2007.

[31] J. Zhao, I. Rogers, and C. Kirkham. Pure method analysis within jikes rvm. In *ICOOOLPS*, 2008.

A. Appendix

A.1 Concrete Semantics

We formulate the concrete semantics of reference immutability as an instrumented small-step operational semantics. The semantics formalizes the runtime properties expressed by reference immutability. The syntax is shown in Figure 8. The operational semantics associates runtime mutabilities, Readonly or Mutable, with each variable. It associates a *local* mutability and a *global* mutability to each reference variable. It associates a single, global mutability to each field. Below, we formalize the semantics.

H	$::= [] \mid H[\mathsf{f} \mapsto q]$	*heap*
S	$::= \epsilon \mid \langle F\ C\ \mathsf{s}\rangle S$	*stack*
F	$::= [] \mid F[\mathsf{y} \mapsto q]$	*local mutability map*
M	$::= [] \mid M[\mathsf{y} \mapsto q]$	*global mutability map*
q	$::= \mathsf{Readonly} \mid \mathsf{Mutable}$	*runtime mutabiltiy*
C		*local dependence graph*
G		*global dependence graph*

Figure 8. Syntax for heap, stack, maps, runtime mutabilities and dependence graphs. f is a field, and y is a parameter or local variable. C and G are unlabeled directed graphs. The nodes are fields and reference variables and the edges denote dependences.

A configuration $S\ H\ G\ M$ consists of a stack S, a heap H, global dependence graph G and global mapping M. A stack is a sequence of frames $\langle F\ C\ \mathsf{s}\rangle$ where F is a mapping from local variables to their local runtime mutabilities, C is a local dependences graph, and s is a statement. There is one graph C per stack frame. The nodes in C are local variables and fields. The edges are directed and represent the dependence relations. For example, if there is a statement $\mathsf{y} = \mathsf{z}$, then there is an edge $\mathsf{z} \rightarrow \mathsf{y}$ in C, which denotes that the runtime mutability of z depends on y, and if y is mutated by some $\mathsf{y.f} = \mathsf{w}$, z must be Mutable as well. We use C to define the *local scope* of a variable x — the local scope of x is the transitive closure of x in C. Thus, if a variable y in the local scope of x is mutated, the local mutability of x becomes Mutable; conversely, if no variable in the local scope of x is mutated, x is Readonly in local scope. Intuitively, the local scope of x consists of all local variables which have obtained their value through x; mutating a variable in the scope of x amounts to "mutating x's object through x".

The heap H is a map from fields to runtime mutabilities. H is a "summary" heap which does not distinguish between fields of different objects. Mutability of fields can be handled more precisely by including heap objects, and allowing the same field f to have different mutabilities in different objects (i.e., field f can be readonly in one object and mutable in another). For brevity and clarity, we choose to reason about a simplified "summary" heap.

G is a global dependence graph. We use G to define the global scope of a variable. Intuitively, the global scope of x consists of all local and non-local variables which have obtained their value through x. Non-local variables obtain their value *after* the enclosing method of x returns. For example, if the return type of a method m is readonly, this means that all variables that obtain their value through m's return must remain readonly; these variables can span many different methods. Map M maps variables to global runtime mutabilities. A variable can be mutated in local scope, and in this case it is mutated in global scope as well; this is captured through qualifier mutable in our type system. A variable can be readonly in local scope, but mutated after the method return and thus be mutable in global scope; this is captured through qualifier polyread. Finally, a variable can be readonly in both local and global scope; these variables are typed readonly. Map M records global runtime mutabilities.

The concrete semantics is shown in Figure 9. We skip the rule for object creation, because it does not create interesting dependences. Rule (DASSIGN) creates an edge in C from y to

(DASSIGN)
$$\frac{C' = C \cup \mathsf{y} \to \mathsf{x}}{\langle F\ C\ \mathsf{x} = \mathsf{y}; \mathsf{s}\rangle S\ H\ G\ M\ \to\ \langle F\ C'\ \mathsf{s}\rangle S\ H\ G\ M}$$

(DREAD)
$$\frac{C' = C \cup \mathsf{y} \to \mathsf{x} \cup \mathsf{f} \to \mathsf{x}}{\langle F\ C\ \mathsf{x} = \mathsf{y.f}; \mathsf{s}\rangle S\ H\ G\ M\ \to\ \langle F\ C'\ \mathsf{s}\rangle S\ H\ G\ M}$$

(DWRITE)
$$\frac{\begin{array}{c} C' = C \cup \mathsf{y} \to \mathsf{f} \\ \overline{\mathsf{z}} = \{\mathsf{z} \mid \mathsf{z} \to^* \mathsf{x} \in C\} \quad F' = F[\overline{\mathsf{z} \mapsto \mathsf{Mutable}}] \\ \overline{\mathsf{g}} = \{\mathsf{g} \mid \mathsf{g} \to^* \mathsf{x} \in C \cup G\} \quad H' = H[\overline{\mathsf{g} \mapsto \mathsf{Mutable}}] \\ \overline{\mathsf{z'}} = \{\mathsf{z'} \mid \mathsf{z'} \to^* \mathsf{x} \in G\} \quad M' = M[\overline{\mathsf{z'} \mapsto \mathsf{Mutable}}] \end{array}}{\langle F\ C\ \mathsf{x.f} = \mathsf{y}; \mathsf{s}\rangle S\ H\ G\ M\ \to\ \langle F'\ C'\ \mathsf{s}\rangle S\ H'\ G\ M'}$$

(DCALL)
$$\frac{\begin{array}{c} mbody(\mathsf{m}) = \mathsf{this\ p}\ \overline{\mathsf{y'}}\ \mathsf{s'}; \mathsf{return\ ret} \\ F' = [\mathsf{this} \mapsto \mathsf{Readonly}][\mathsf{p} \mapsto \mathsf{Readonly}][\overline{\mathsf{y'} \mapsto \mathsf{Readonly}}] \\ C' = \{\} \\ S' = \langle F'\ C'\ \mathsf{s'}; \mathsf{return\ ret}\rangle\langle F\ C\ \mathsf{x} = \mathsf{y.m(z)}; \mathsf{s}\rangle S \end{array}}{\langle F\ C\ \mathsf{x} = \mathsf{y.m(z)}; \mathsf{s}\rangle S\ H\ G\ M\ \to\ S'\ H\ G\ M}$$

(DRETURN)
$$\frac{\begin{array}{c} mbody(\mathsf{m}) = \mathsf{this\ p}\ \overline{\mathsf{y'}}\ \mathsf{s'}; \mathsf{return\ ret} \\ \\ \textit{if } F'(\mathsf{this}) = \mathsf{Mutable} \textit{ then} \\ \overline{\mathsf{w}} = \{\mathsf{w} \mid \mathsf{w} \to^* \mathsf{y} \in C\} \quad F'' = F[\overline{\mathsf{w} \mapsto \mathsf{Mutable}}] \\ \overline{\mathsf{g}} = \{\mathsf{g} \mid \mathsf{g} \to^* \mathsf{y} \in C \cup G\} \quad H' = H[\overline{\mathsf{g} \mapsto \mathsf{Mutable}}] \\ \overline{\mathsf{w'}} = \{\mathsf{w'} \mid \mathsf{w'} \to^* \mathsf{y} \in G\} \quad M' = M[\overline{\mathsf{w'} \mapsto \mathsf{Mutable}}] \\ \textit{else} \\ F'' = F \quad H' = H \quad M' = M \\ \\ \textit{if } F'(\mathsf{p}) = \mathsf{Mutable} \textit{ then} \\ \overline{\mathsf{w}} = \{\mathsf{w} \mid \mathsf{w} \to^* \mathsf{z} \in C\} \quad F''' = F''[\overline{\mathsf{w} \mapsto \mathsf{Mutable}}] \\ \overline{\mathsf{g}} = \{\mathsf{g} \mid \mathsf{g} \to^* \mathsf{z} \in C \cup G\} \quad H''' = H''[\overline{\mathsf{g} \mapsto \mathsf{Mutable}}] \\ \overline{\mathsf{w'}} = \{\mathsf{w'} \mid \mathsf{w'} \to^* \mathsf{z} \in G\} \quad M'' = M'[\overline{\mathsf{w'} \mapsto \mathsf{Mutable}}] \\ \textit{else} \\ F''' = F'' \quad H'' = H' \quad M'' = M' \\ \\ \textit{if } \mathsf{this} \to^* \mathsf{ret} \in C' \textit{ then } C'' = C \cup \mathsf{y} \to \mathsf{x} \textit{ else } C'' = C \\ \textit{if } \mathsf{p} \to^* \mathsf{ret} \in C' \textit{ then } C''' = C'' \cup \mathsf{z} \to \mathsf{x} \textit{ else } C''' = C'' \\ G' = G \cup C' \cup \mathsf{ret} \to \mathsf{x} \\ S' = \langle F'''\ C'''\ \mathsf{s}\rangle S \end{array}}{\langle F'\ C'\ \mathsf{s'}; \mathsf{return\ ret}\rangle\langle F\ C\ \mathsf{x} = \mathsf{y.m(z)}; \mathsf{s}\rangle S\ H\ G\ M\ \to\ S'\ H''\ G'\ M''}$$

Figure 9. Concrete semantics.

x; if x is mutated, the mutability must be propagated to y. Rule (DREAD) adds two dependence edges to C, $\mathsf{f} \to \mathsf{x}$ and $\mathsf{y} \to \mathsf{x}$ because a mutation of x will affect the mutability of both f and y.

Rule (DWRITE) is more interesting. It first adds an edge to C to reflect the dependence from y to f. Then it finds all local variables z (including x), such that x is reachable from z in C. In other words, x is in the local scope of each z. The rule also changes the mutabilities of all fields g in C or G, when there is a path from g to x in C or G. Finally, the rule changes the global mutabilities of all z (including x), such that there is a path from z to x in the global dependence graph G.

Rule (DCALL) retrieves the body of the target method m. The body consists of implicit parameter this, formal parameter p, local variables $\overline{\mathsf{y'}}$, statement s, and return statement return ret. The rule creates a new frame F' where every local variable including this and p, is mapped to Readonly. Dependence graph C' is empty.

Rule (DRETURN) is the most interesting. If this of the callee is mutable, then y as well as every local variable w such that y is in the local scope of w, must be mutable. Also, every field g such that y is in the global or local scope of g, must be mutable. Finally, every variable w' such that y is in the global scope of w' must be made mutable. If p is mutable, we propagate its mutability in analogous way. The next 2 lines propagate dependences from the callee to the caller: if ret is in the local scope of this, then we add an edge to C from y to x, and similarly, if ret is in the scope of p, we add an edge from z to x. The last line adds local dependence graph C' to G and "connects" C' and C in G with edge $\mathsf{ret} \to \mathsf{x}$.

A.2 Well-formedness

The well-formedness rules are shown in Figure 10. A configuration is well-formed, which is written $S\ H\ G\ M$ is WF, if the stack, the heap and $G\ M$ are well-formed. A stack is well-formed if all of its frames are well-formed. A frame is well-formed if for each variable x in the domain of F, $\Gamma(\mathsf{x}) = \mathsf{readonly}$ or polyread implies that $F(\mathsf{x}) = \mathsf{Readonly}$. In other words, if a variable x is readonly or polyread, then no variable in the local scope of x can be mutated at runtime. Another requirement for the well-formedness of a frame is that for every edge $\mathsf{x} \to \mathsf{y} \in C$ we have $\Gamma(\mathsf{x}) <: \Gamma(\mathsf{y})$. The heap H is well-formed when for every readonly field f $H(\mathsf{f}) = \mathsf{Readonly}$. $G\ M$ is well-formed when for every variable x, $\Gamma(\mathsf{x}) = \mathsf{readonly}$ implies $M(\mathsf{x}) = \mathsf{Readonly}$.

A.3 Type Soundness

We prove type soundness by showing preservation and progress. Preservation means that reduction of a well-formed configuration produces a well-formed configuration.

Theorem A.1. *Preservation. If $S\ H\ G\ M$ is WF and $S\ H\ G\ M \to H'\ S'\ G'\ M'$, then $H'\ S'\ G'\ M'$ is WF.*

The proof is by structural induction on the derivation; it enumerates the different kinds of steps in the semantics, and shows that after each step, a well-formed configuration stays well-formed.

$$\frac{H \text{ is WF} \quad S \text{ is WF} \quad G\ M \text{ is WF} \vdash CT}{S\ H\ G\ M \text{ is WF}}\ \text{(WF-CONFIGURATION)}$$

$$\frac{\begin{array}{c} vars(M) = \Gamma \\ \forall \mathsf{x} \in dom(M), \\ \Gamma(\mathsf{x}) = \mathsf{readonly} \Rightarrow M(\mathsf{x}) = \mathsf{Readonly} \end{array}}{G\ M \text{ is WF}}\ \text{(WF-GRAPH)}$$

$$\frac{}{[] \text{ is WF}}\ \text{(WF-HEAP-EMPTY)}$$

$$\frac{H \text{ in WF} \quad typeof(\mathsf{f}) = \mathsf{readonly} \ \Rightarrow\ q = \mathsf{Readonly}}{H[\mathsf{f} \mapsto q] \text{ is WF}}\ \text{(WF-HEAP)}$$

$$\frac{}{[] \text{ is WF}}\ \text{(WF-STACK-EMPTY)}$$

$$\frac{S \text{ is WF} \quad \langle F\ C\ \mathsf{s}\rangle \text{ is WF}}{\langle F\ C\ \mathsf{s}\rangle S \text{ is WF}}\ \text{(WF-STACK)}$$

$$\frac{\begin{array}{c} vars(F) = \Gamma \quad \Gamma \vdash \mathsf{s} \\ \forall \mathsf{x} \rightarrow \mathsf{y} \in C, \Gamma(\mathsf{x}) <: \Gamma(\mathsf{y}) \\ \forall \mathsf{x} \in dom(F), \\ \Gamma(\mathsf{x}) = \mathsf{readonly}, \mathsf{polyread} \Rightarrow F(\mathsf{x}) = \mathsf{Readonly} \end{array}}{\langle F\ C\ \mathsf{s}\rangle \text{ is WF}}\ \text{(WF-FRAME)}$$

Figure 10. Well-formedness rules.

Proof. (Sketch) We show the most interesting cases, (DASSIGN), (DWRITE) and (DRETURN).

(DASSIGN). This step changes only C of the current frame. It adds an edge $\mathsf{y} \rightarrow \mathsf{x}$ to C. By $\vdash CT$ we have that $\mathsf{x} = \mathsf{y}$ is well-typed, and therefore, $\Gamma(\mathsf{y}) <: \Gamma(\mathsf{x})$. Thus, the frame with C' remains well-formed.

(DWRITE). This step changes F and C in the current frame, as well as H and M. We have to show that C' remains well-formed, and $G\ M'$ and H' remain well-formed. We must show that for every $\mathsf{z} \rightarrow^* \mathsf{x} \in C$ (here z changes to Mutable), we have $\Gamma(\mathsf{z}) = \mathsf{mutable}$. Since $\mathsf{x.f} = \mathsf{y}$ is well-typed, we have $\Gamma(\mathsf{x}) = \mathsf{mutable}$. By well-formedness of C's frame, we have $\Gamma(\mathsf{z}) <: \Gamma(\mathsf{x})$ and therefore $\Gamma(\mathsf{z}) = \mathsf{mutable}$. We must show that for every w in $\mathsf{w} \rightarrow^* \mathsf{x} \in G$, $\Gamma(\mathsf{w}) \neq \mathsf{readonly}$. Suppose that $\Gamma(\mathsf{w})$ is readonly. By construction of G, we have that the path $\mathsf{w} \rightarrow^* \mathsf{x}$ is comprised of two kinds of edges, local edges $A \rightarrow B$ that belong to some C, and return edges $\mathsf{ret} \rightarrow C$. By induction on the length of the path $\mathsf{w} \rightarrow^* \mathsf{x}$ it easily follows that if $\Gamma(\mathsf{z})$ is readonly, then $\Gamma(\mathsf{x})$ must be readonly. This contradicts the fact that $\Gamma(\mathsf{x})$ is mutable (recall that x comes from $\mathsf{x.f{=}y}$ and the only way to have that statement well-typed is to type x as mutable). Analogous argument holds for fields f, and the well-formedness of H' follows.

(DRETURN). Again, we must prove that the new frame on top of the stack as WF, that $G'\ M''$ is WF and that H'' is WF. Well-formedness of $G'\ M''$ and H'' is analogous to (DWRITE). We show well-formedness of C'''. We must show that the new dependence edge added to C, namely $\mathsf{y} \rightarrow \mathsf{x}$, obeys $\Gamma(\mathsf{y}) <: \Gamma(\mathsf{x})$. By well-formedness of C' we have $\Gamma(\mathsf{this}) <: \Gamma(\mathsf{ret})$. $\Gamma(\mathsf{ret})$ can be readonly or polyread. If it is readonly, then the only way $\mathsf{x} = \mathsf{y.m(z)}$ would type check is if x is readonly; thus, $\Gamma(\mathsf{y}) <: \Gamma(\mathsf{x})$ holds for any $\Gamma(\mathsf{y})$. Otherwise, i.e., if $\Gamma(\mathsf{ret})$ is polyread, $\Gamma(\mathsf{this}) <: \Gamma(\mathsf{ret})$ gives us that $\Gamma(\mathsf{this})$ is either mutable or polyread. If $\Gamma(\mathsf{this})$ is mutable then $\Gamma(\mathsf{y})$ must be mutable, and $\Gamma(\mathsf{y}) <: \Gamma(\mathsf{x})$ holds for any value of $\Gamma(\mathsf{x})$. If $\Gamma(\mathsf{this})$ is polyread, then we have $\Gamma(\mathsf{y}) <: \Gamma(\mathsf{x}) \rhd_m \mathsf{polyread} = \Gamma(\mathsf{x})$ which is what we want. $\Gamma(\mathsf{z}) <: \Gamma(\mathsf{x})$ is shown analogously. □

Theorem A.2. *Progress. If $S\ H\ G\ M$ is WF, then either the problem ends or $S\ H\ G\ M \rightarrow H'\ S'\ G'\ M'$.*

Proof. (Sketch) The prove is by structural induction on s when $S = \langle F\ C\ \mathsf{s}\rangle S'$. It is immediate by applications of the corresponding rule for each s.

□