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

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

Werner Dietl Michael D. Ernst
University of Washington
Seattle, WA, USA
{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 ReIm, a type system for reference immutability, and ReImInfer, a corresponding type inference analysis. 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 348kLOC show that our approach achieves both scalability and precision.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Experimentation, Languages, Theory

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(2);     // compile−time error, rd is immutable
```

The type 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 which returns elements that have signed a particular class. In JDK 1.1, it is implemented approximately as follows:

```
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 of mutable objects is expressed, following Java 8 syntax [13], as Object readonly [].) As a result, mutations of the array through the returned reference will be disallowed:

```
Object readonly [] getSigners() {
  return signers;
}
```


OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

```
...
Object readonly [] signers = getSigners();
signers[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 mutability of parameters and return values; it helps prevent and detect errors caused by unwanted object mutations; and it facilitates reasoning about and proving other properties such as object immutability and method purity.

This paper presents a context-sensitive type system for reference immutability, ReIm, and an efficient inference analysis, ReImInfer. We implemented our inference system for Java and performed case studies of applications and libraries of up to 348kLOC.

ReIm is related to Javari [33], 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, in order to support code patterns such as caches and lazily-initialized fields. Another difference is that ReIm encodes context sensitivity using the concept of *viewpoint adaptation* from Universe Types [9, 10], while Javari uses templatizing. These design decisions result in a more compact type system, particularly suitable for reasoning about method purity.

Our inference 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 [26], 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, ReImInfer relies entirely on the Checker Framework [11, 23], 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, 19, 37], model checking [32], Universe Type inference [12, 18], and memoization of procedure calls [17]. 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. In fact, the work presented here was motivated by our need for purity information for Universe Type inference [12, 18]. We spent considerable time trying to use existing tools. Unfortunately, existing tools were imprecise, fragile, whole-program, and/or unscalable, as we detail in Section 4. Our solution was to build ReIm and ReImInfer and the purity extension. ReImInfer outperformed existing tools, and is fully integrated with the Checker Framework.

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 348kLOC, including widely used Java applications and libraries, comprising 766kLOC 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.

2. ReIm Reference Immutability Types

This section describes the immutability qualifiers (Section 2.1) and explains context sensitivity (Section 2.2). It proceeds to define the type system for reference immutability (Section 2.3) and the inference analysis (Section 2.4).

2.1 Immutability Qualifiers

The ReIm type system has three immutability qualifiers: mutable, readonly, and polyread. These qualifiers were introduced by Javari [33] (except that polyread was romaybe in Javari but became polyread in Javarifier [26]). They have essentially the same meaning in Javari and ReIm, except that readonly in Javari allows certain fields and generic type arguments to be excluded from the immutability guarantee,

while readonly in ReIm guarantees immutability of the entire structure. We detail the differences in Section 5.1.

- mutable: A mutable reference can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages.
- readonly: A readonly reference x cannot be used to mutate the referenced object nor anything it references. For example, all of the following are forbidden:
  - x.f = z
  - x.setField(z) where setField sets a field of its receiver
  - y = id(x); y.f = z where id is a function that returns its argument
  - x.f.g = z
  - y = x.f; y.g = z
- polyread: This qualifier expresses polymorphism over immutability. Essentially, polyread denotes a reference is immutable in the current context, but it may or may not be mutable in other contexts. The interpretation of polyread depends on the context and will be explained 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

ReIm expresses context sensitivity using a variant of viewpoint adaptation [9]. Consider the following code. For readability, code throughout this section makes the formal parameter this explicit.

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

In the above code, this of cellGetHours may be annotated as readonly, which is the top of the type hierarchy. Doing so is advantageous because then cellGetHours can be called on any argument.

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

A context-sensitive type is required for the return type of DateCell.getDate. The effective return type will depend on the calling context. An example of calling context is the type of the left-hand side of an assignment statement, when the call is the right-hand side. Another example is the type of a formal parameter, when the call is used as an actual argument.

The polymorphic qualifier polyread expresses context sensitivity. We annotate this, the return type of getDate, and field date as polyread:

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

Intuitively, viewpoint adaptation instantiates polyread to mutable in the context of cellSetHours, and to readonly in the context of cellGetHours. The call this.getDate on line 5 returns a mutable Date, and the call this.getDate on line 9 returns a readonly Date. As a result, the mutability of md propagates only to this of cellSetHours; it does not propagate to this of cellGetHours which remains readonly. ReIm handles polyread via viewpoint adaptation, and Javari/Javarifier handle polyread via templatizing methods. The two approaches appear to be semantically equivalent. Viewpoint adaptation however, is a more compact and scalable way of handling polymorphism than templatizing.

Conceptually, a method must type-check with each instance of polyread replaced by (adapted to) mutable, and with each instance of polyread replaced by readonly. Thus, a polyread reference x cannot be used to mutate the referenced object. A method may return x to the caller, in which case the caller might be able to mutate the object. Programmers should use polyread when the reference is readonly in the scope of the enclosing method, but may be modified in some caller contexts after the method's return.

The type of a polyread field f is adapted to the viewpoint of the receiver that accesses the field. If the receiver x is mutable, then x.f is mutable. If the receiver x is readonly, then x.f is readonly. If the receiver x is polyread, then x.f is polyread and cannot be used to modify the referenced object, as the access might be further instantiated with a readonly receiver. 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.

We forbid mutable as a qualifier for fields. ReIm gives a strong reference immutability guarantee, including the whole transitive state. A mutable field would not depend on the type of the receiver and would therefore violate this guarantee.

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 [35]. 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. In ReIm, $\triangleright$ adapts field accesses from the point of view of the receiver, but adapts method calls from the point of view of the *variable at the left-hand-side* of the call assignment. This concept is also known as the calling context or call-site context.

We define $\triangleright$ as:

$$\begin{array}{lcl} \_ \triangleright \mathsf{mutable} & = & \mathsf{mutable} \\ \_ \triangleright \mathsf{readonly} & = & \mathsf{readonly} \\ q \triangleright \mathsf{polyread} & = & q \end{array}$$

The underscore denotes a "don't care" value. Qualifiers mutable and readonly do not depend on the viewpoint. Qualifier polyread depends on the viewpoint and is substituted by that viewpoint.

For a field access, viewpoint adaptation $q \triangleright q_{\mathsf{f}}$ adapts the declared field qualifier q_{f} from the point of view of receiver qualifier q. In field access y.f where the field f is readonly, the type of y.f is readonly. In field access y.g where the field g is polyread, y.g takes the type of y. If y is readonly, then y.g must be readonly as well, in order to disallow modifications of y's object through y.g. If y is polyread then y.g is polyread as well, propagating the context-dependency.

For a method call $\mathsf{x} = \mathsf{y.m(z)}$, viewpoint adaptation $q_{\mathsf{x}} \triangleright q$ adapts q, the declared qualifier of a formal parameter/return of m, from the point of view of q_{x}, the qualifier at the left-hand-side x of the call assignment. 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 that ReIm allows 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, it can be interpreted as encoding context sensitivity at field-transmitted dependences *differently* from context sensitivity at call-transmitted dependences. And it can be viewed as allowing different *abstractions of context*. For example, adaptation from the point of view of the receiver amounts to object sensitivity [22]. 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 [27], and viewpoint adaptation is left for future work.

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

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, formal parameters, or parameter this. As in the code examples, this is explicit. For simplicity, we assume all names are unique.

2.3 Typing Rules

For brevity, we restrict our formal attention to a core calculus in the style of Vaziri et al. [35] 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 parameter this, and exactly one other formal 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.

In contrast to a formalization of standard Java, a type t has two orthogonal components: type qualifier q (which expresses reference immutability) and Java class type C. The immutability 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. 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 adaptation 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 parameter this, q_p is the type of the formal parameter, and q_{ret} is the type of the return. Rule (TCALL) requires $q_{\mathsf{x}} \triangleright q_{\mathsf{ret}} <: q_{\mathsf{x}}$. This constraint disallows the return value of m from being readonly when there is a call to m, $\mathsf{x} = \mathsf{y.m(z)}$, where left-hand-side x

(TNEW)

$$\frac{}{\Gamma \vdash \mathsf{x} = \mathsf{new\ mutable\ C}}$$

(TASSIGN)

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

(TWRITE)

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

(TREAD)

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

(TCALL)

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \qquad \Gamma(\mathsf{y}) = q_\mathsf{y} \qquad \Gamma(\mathsf{z}) = q_\mathsf{z} \qquad \mathit{typeof}(\mathsf{m}) = q_\mathsf{this}, q_p \rightarrow q_\mathsf{ret} \qquad q_\mathsf{y} <: q_\mathsf{x} \rhd q_\mathsf{this} \qquad q_\mathsf{z} <: q_\mathsf{x} \rhd q_p \qquad q_\mathsf{x} \rhd q_\mathsf{ret} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}$$

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.

is mutable. Only if the left-hand-sides of all call assignments to m are readonly, can the return type of m be readonly; otherwise, it is polyread. A programmer can 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.

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

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

which disallows q_y from being anything but mutable, as expected. The most interesting case arises when q_this is polyread. Recall that a polyread parameter this is readonly within the enclosing method, but there could be a dependence between this and ret such as

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

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

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

Recall that when there exists a context where the left-hand-side variable x is mutated, q_ret must be polyread. Therefore, constraint $q_\mathsf{this} <: q_\mathsf{ret}$ forces q_this to be polyread (let us assume that this is not mutated in the context of its enclosing method).

The role of viewpoint adaptation is to transfer the dependence between this and ret in m, into a dependence between actual receiver y and left-hand-side x in the call assignment. In the above example, there is a dependence between this and the return ret. Thus, we also have a dependence between y and x in the call $\mathsf{x = y.m()}$ — that is, a mutation of x makes y mutable as well. Function $\rhd$ does exactly that. Rule (TCALL) requires

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

When there is a dependence between this and ret, q_this is polyread, and the above constraint becomes

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

This is exactly the constraint we need. If x is mutated, y becomes mutable as well. In contrast, if x is readonly, y remains unconstrained.

Note that adapting from the viewpoint of the receiver, as is customary in ownership type systems, will result in

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

which does not impose any constraints on y when q_this is polyread.

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

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

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

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

It is instantiated to readonly in cellGetHours:

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

Thus, this of cellGetHours can be typed readonly.

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

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

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 parameters this, (2) field names, and (3) method returns. The values in the mapping are *sets* of type qualifiers. 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" and "variable" to refer to all kinds of keys: local variables, fields, and method returns.

S is initialized as follows. Programmer-annotated references, if any, are initialized to the singleton set that contains the programmer-provided type. Method returns are initialized $S(\mathsf{ret}) = \{\mathsf{readonly}, \mathsf{polyread}\}$ for each method m. Fields are initialized $S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{polyread}\}$. All other references are initialized to the maximal set of qualifiers, i.e., $S(\mathsf{x}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$. We denote the initial mapping by S_0.

There is a 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, f_s removes all inconsistent qualifier from the reference's set. Consider reference x. f_s removes each $q_\mathsf{x} \in S(\mathsf{x})$ 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. The same repeats for y, and then z. Note that the order in which references are examined does not affect the final result — one can see that f_s always removes the same set of qualifiers from, say, $S(\mathsf{x})$, regardless of whether x is examined first, second or last. This algorithm is similar to the one used by Tip et al. [14, 31].

For example, consider statement $\mathsf{x} = \mathsf{y.f}$ and corresponding rule (TREAD). Suppose that $S(\mathsf{x}) = \{\mathsf{polyread}\}$, $S(\mathsf{y}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$ and $S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{polyread}\}$ before the application of the function. The function removes readonly from $S(\mathsf{y})$ because there does not exist $q_\mathsf{f} \in S(\mathsf{f})$ and $q_\mathsf{x} \in S(\mathsf{x})$ that satisfies $\mathsf{readonly} \triangleright q_\mathsf{f} \ <: \ q_\mathsf{x}$. Similarly, the function removes readonly from $S(\mathsf{f})$ because there does not exist $q_\mathsf{y} \in S(\mathsf{y})$ and $q_\mathsf{x} \in S(\mathsf{x})$ that satisfies $q_\mathsf{y} \triangleright \mathsf{readonly} \ <: \ q_\mathsf{x}$. After the application of the function, S' is as follows: $S'(\mathsf{x}) = \{\mathsf{polyread}\}$, $S'(\mathsf{y}) = \{\mathsf{polyread}, \mathsf{mutable}\}$, and $S'(\mathsf{f}) = \{\mathsf{polyread}\}$.

```
class A {
  X f;                          S(f) = {polyread}
  X get(A this, Y y) {          S(this_get) = {polyread, mutable}
    ... = y.h;                  S(y_get) = {readonly, polyread, mutable}
    X x = this.getF();          S(x_get) = {polyread, mutable}
    return x;                   S(ret_get) = {polyread}
  }
  X getF(A this) {              S(this_getF) = {polyread, mutable}
    X x = this.f;               S(x_getF) = {polyread, mutable}
    return x;                   S(ret_getF) = {polyread}
  }
}
void setG() {
  A a = ...                     S(a_setG) = {mutable}
  Y y = ...                     S(y_setG) = {readonly, polyread, mutable}
  X x = a.get(y);               S(x_setG) = {mutable}
  x.g = null;
}
void getG() {
  A a = ...                     S(a_getG) = {readonly, polyread, mutable}
  Y y = ...                     S(y_getG) = {readonly, polyread, mutable}
  X x = a.get(y);               S(x_getG) = {readonly, polyread, mutable}
  ... = x.g;
}
```

Figure 3. Polymorphic methods. A.get(Y) has different mutabilities in the contexts of setG and getG. Also, A.getF(), which is called from A.get(Y), has different mutabilities in different calling contexts. The box beside each statement shows the set-based solution; the underlined qualifiers are the final qualifiers picked by ReImInfer.

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 qualifiers is derived as follows: for each reference x we pick the greatest element of $S(\mathsf{x})$ according to the subtyping relation, which we also call the "preference ranking" when we use it for this purpose. This ranking maximizes 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.

Consider the example in Figure 3. We use x_get to denote the reference x in method get. Initially, all references are initialized to the sets as described above. The analysis iter-

ates over all statements in class A and in methods setG and getG. In the first iteration, the analysis changes nothing until it processes x.g = null in setG. $S(\mathsf{x}_{\mathsf{setG}})$ is updated to {mutable}. In the second iteration, when the analysis processes x = a.get(y) in setG, $S(\mathsf{ret}_{\mathsf{get}})$ becomes {polyread}. In the third iteration, $S(\mathsf{x}_{\mathsf{get}})$ becomes {polyread, mutable} because $\mathsf{x}_{\mathsf{get}}$ has to be a subtype of $S(\mathsf{ret}_{\mathsf{get}})$. This in turn forces $S(\mathsf{ret}_{\mathsf{getF}})$ and subsequently $S(\mathsf{this}_{\mathsf{getF}})$ 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 greatest element in the preference ranking.

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.

The inferred solution is correct, precise, and maximal. The following propositions formalize its properties.

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 function, the rule type checks with the maximal assignment. Let $max(S(\mathsf{x}))$ return the maximal element of $S(\mathsf{x})$ according to the preference ranking (which is the same as the type hierarchy). We show (TCALL) x = y.m(z). The rest of the cases are straightforward.

- Let $max(S(\mathsf{x}))$ be readonly.
  If $max(S(\mathsf{this}))$ is readonly or polyread, $q_{\mathsf{y}} <: q_{\mathsf{x}} \triangleright q_{\mathsf{this}}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(\mathsf{this}))$ is mutable, the only possible max for y would be mutable (the others would have been removed by the function for (TCALL)). We do not discuss $q_{\mathsf{z}} <: q_{\mathsf{x}} \triangleright q_p$ because it is analogous to $q_{\mathsf{y}} <: q_{\mathsf{x}} \triangleright q_{\mathsf{this}}$
  $q_{\mathsf{x}} \triangleright q_{\mathsf{ret}} <: q_{\mathsf{x}}$ holds for any value of $max(S(\mathsf{ret}))$.
- Let $max(S(\mathsf{x}))$ be mutable.
  If $max(S(\mathsf{this}))$ is readonly, $q_{\mathsf{y}} <: q_{\mathsf{x}} \triangleright q_{\mathsf{this}}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(\mathsf{this}))$ is polyread, the only possible value for $max(S(\mathsf{y}))$ would be mutable. If $max(S(\mathsf{this}))$ is mutable, the only possible max for y would be mutable as well (the others would have been removed by the function for (TCALL)).
  If $max(S(\mathsf{ret}))$ is polyread, clearly $q_{\mathsf{x}} \triangleright q_{\mathsf{ret}} <: q_{\mathsf{x}}$ holds. $max(S(\mathsf{ret}))$ cannot be readonly, readonly would have been removed by the function.
- Let $max(S(\mathsf{x}))$ be polyread.
  If $max(S(\mathsf{this}))$ is readonly, $q_{\mathsf{y}} <: q_{\mathsf{x}} \triangleright q_{\mathsf{this}}$ holds for any value of $max(S(\mathsf{y}))$. If $max(S(q_{\mathsf{this}}))$ is polyread, the only possible values 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.
  If $max(S(\mathsf{ret}))$ is polyread, clearly $q_{\mathsf{x}} \triangleright q_{\mathsf{ret}} <: q_{\mathsf{x}}$ holds. $max(S(\mathsf{ret}))$ cannot be readonly, readonly would have been removed by the function. □

The next two propositions establish the precision of our inference algorithm. ReIm, like Ownership type systems [5, 9] allows many different typings for a given program. As mentioned earlier, the trivial typing for ReIm applies mutable to every reference; this clearly type checks, but is useless as we would like to prove as many references as readonly as possible. The main challenges for type inference is to define which typing is "best" (or most desirable/precise) and then infer that typing.

In previous work [18], we formalized the notion of best typing for ownership type systems, specifically Ownership types [5] and Universe Types [9], by using a heuristic ranking over typings. This formalization applies to ReIm as well as to other ownership-like type systems such as AJ [35] and EnerJ [29].

We say that T is a *valid typing* if T type checks with ReIm. We proceed to define an objective function o which ranks valid typings. o takes a valid typing T and returns a tuple of numbers. For ReIm, the objective function is the following:

$$o_{ReIm}(T) = (|T^{-1}(\mathsf{readonly})|, |T^{-1}(\mathsf{polyread})|, |T^{-1}(\mathsf{mutable})|)$$

The tuples are ordered lexicographically. We have $T_1 > T_2$ iff T_1 has more readonly references than T_2, or T_1 and T_2 have the same number of readonly references, but T_1 has more polyread references than T_2. The preference ranking over typings is based on the preference ranking over qualifiers: we prefer readonly over polyread and mutable, and polyread over mutable. This is a natural ranking for ReIm.

The following two propositions establish that the maximal typing, i.e., the typing that maps each x to $max(S(\mathsf{x}))$, maximizes the above objective function. In other words, the maximal typing computed by our inference procedure is the best typing. Note that Proposition 2.1 establishes that the maximal typing is valid. Recall that S_0 denotes the initial mapping. $T \in S_0$ denotes that $T(v) \in S_0(v)$ for every variable v.

Proposition 2.2. *Let S be the set-based solution. Let v be any variable in the program and let q be any qualifier in ReIm. If $q \notin S(v)$ then there does not exist a valid typing $T \in S_0$, such that $T(v) = q$.*

Proof. (Sketch) We say that q is a valid qualifier for v if there exists a valid typing T, where $T(v) = q$. Let v be the *first* variable that has a valid qualifier q removed from its set $S(v)$ and let f_s be the function that performs the removal. Since q is a valid qualifier there exist valid qualifiers $q_1, ..., q_k$ that make s type check. If $q_1 \in S(v_1)$ and $q_2 \in S(v_2)$, ..., and $q_k \in S(v_k)$, then by definition, f_s would not have had q removed from $S(v)$. Thus, one of $v_1, \ldots, v_k$ must have had a valid qualifier removed from its set *before* the application of

f_s. This contradicts the assumption that v is the first variable that has a valid qualifier removed. □

The second proposition states that if we map every variable v to $max(S(v))$ and the typing is valid, then this typing maximizes the objective function.

Proposition 2.3. *Let o_{ReIm} be the objective function over valid typings, and S be the set-based solution. The* maximal typing T *is the following: $T(v) = max(S(v))$ for every variable v. If T is a valid typing, then T is the unique maximal typing of the program under o_{ReIm}.*

Proof. (Sketch) We show that T is a maximal typing. Suppose that there exists a different valid typing $T' \geq T$. Let q' be the most-preferred qualifier such that $T'^{-1}(q') \neq T^{-1}(q')$. Since $T' \geq T$, there must exist a variable v such that $T'(v) = q'$, but $T(v) = q < q'$. In other words, T' types v with q', but T types v differently — and lesser in the preference ranking. Since $T(v) = max(S(v))$, it follows that $q' \notin S(v)$. By Proposition 2.2, if $q' \notin S(v)$ there does not exist a valid typing which maps v to q', which contradicts the assumption that T' is a valid typing. □

Additionally, the 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 Propositions 2.2 and 2.3, which state the precision argument, we perform detailed comparison with Javarifier, the state-of-the-art tool for inference of reference immutability.

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, 19, 37], model checking [32], Universe Type inference [12, 18], memoization of function calls [17], and so on.

We adopt the definition of purity given by Sălcianu and Rinard [30]: 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 to return a newly constructed object as a result. This is the semantics of the @Pure annotation in JML.

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,

```
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.len = 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 and example usage.

i.e., the objects referred by them may be mutated in add. When there is a method invocation lst.add(node), we know that the prestates referred to by the actual argument node and the receiver lst may be 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 m. Roughly, q_{m} is mutable when m accesses static state through some static field and then mutates this static state; q_{m} is polyread if m accesses static state but does not mutate this state directly, however, m may return this static state to the caller and the caller may mutate it; q_{m} is readonly otherwise. Static immutability types are computed using reference immutability. We introduce a function $statictypeof$ that retrieves the static immutability type of m:

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

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.

In contrast to instance fields, static fields are declared as either readonly or mutable. There is no receiver for static field accesses and therefore no substitution for polyread would occur.

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

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

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

$$\frac{\begin{array}{c}\text{(TCALL)}\\ \Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ typeof(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \\ q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd q_{\mathsf{this}} \qquad q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd q_p \\ q_{\mathsf{x}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}} \\ \underline{methodof(\mathsf{x} = \mathsf{y.m(z)}) = \mathsf{m}'} \quad \underline{statictypeof(\mathsf{m}) = q_{\mathsf{m}}} \\ \underline{statictypeof(\mathsf{m}') = q_{\mathsf{m}'}} \quad \underline{q_{\mathsf{m}'} <: q_{\mathsf{x}} \rhd q_{\mathsf{m}}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}$$

Figure 5. Extended typing rules for static fields (see Figure 2 for the base type system). Function $methodof(s)$ returns the enclosing method of statement s. Function $statictypeof(\mathsf{m})$ returns the static immutability type of m. Static immutability types can be readonly, polyread, or mutable . Rule (TCALL) includes the antecedents from the base type system and the new antecedents that handle the static immutability type of m, underlined.

$\mathsf{x} = \mathsf{sf}$ where x is inferred as mutable, 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 $\mathsf{x} = \mathsf{sf}$, 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 and the caller subsequently writes a field of the returned object, 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;
  ...
}
```

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.

Rule (TCALL) in Figure 5 captures two cases:

1. If the callee m mutates static fields, i.e. $statictypeof(\mathsf{m})$ = mutable, the $statictypeof(\mathsf{m}')$ of enclosing method m' has to be mutable as well, because $\_ \rhd \mathsf{mutable} = \mathsf{mutable}$.
2. If the callee m returns a static field which is mutated later, $statictypeof(\mathsf{m})$ would be polyread. If the enclosing method m' mutated the return value x, i.e. $q_{\mathsf{x}} = \mathsf{mutable}$, $statictypeof(\mathsf{m}')$ would be mutable because $\mathsf{mutable} \rhd \mathsf{polyread} = \mathsf{mutable}$. Otherwise, if x is polyread, $statictypeof(\mathsf{m}')$ is constrained to polyread or mutable. Finally, if x is readonly, $statictypeof(\mathsf{m}')$ is readonly, indicating that m' does not mutate static fields.

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 $\{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$ 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 polyread or 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. Note however that method m is impure, because there is a write to its parameter. 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 or polyread). 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 as compiler plug-ins in the Checker Framework (CF) [11, 23]. The purity inference is implemented on top of the CF as well. The implementation relies on the CF to generate type constraints for Java programs. Because it handles context sensitivity by using viewpoint adaptation, it generates less type constraints compared to Javarifier, and the constraints can be solved faster. The tool, called ReImInfer, is publicly available at `http://code.google.com/p/type-inference/`, including source.

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.

Benchmarks JOlden, tinySQL, htmlparser, and ejc are precisely the benchmarks used by Javarifier [33]. Javarifier's distribution includes regression tests, which greatly facilitates the comparison between Javarifier and ReImInfer. java.lang and java.util are included because they are representative libraries. The rest of the benchmarks come from our previous experimental work [18, 21].

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 1.3.0 on GNU/Linux 3.2.0.

The goal of our experiments is to demonstrate the scalability, precision, and robustness of ReImInfer.

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 [30]; 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 and/or written through this in exactly one instance method m, as if they were local variables. Precisely, this means that for these fields we allow qualifier mutable, 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: $q_{\mathsf{this}} <: q_{\mathsf{f}}$. 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. If f is readonly, no constraints are imposed on this. As an example, field 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.

[1] We added 392 empty methods in tinySQL in order to compile it with Java 1.6. The modified version is available online.

If current were treated as a field, it would be polyread, 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 state, 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 mutable, mutable $\rightarrow$ polyread. The mutable parameters assume worst-case behavior of the unknown callee — the unknown callee mutates its arguments. Clearly, readonly is the most general return type. However, this will require that every use of the return in the client is readonly, and many clients violate this restriction. mutable, mutable $\rightarrow$ polyread is safe because we can always assign the polyread return value to a readonly variable. And it also imposes a constraint on the callee: e.g., suppose the code for X id(X p) { return p; } was unavailable and we assumed typing mutable $\rightarrow$ polyread for id. When it becomes available, p will be polyread.

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 from the 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, $typeof(\mathsf{m}') <: typeof(\mathsf{m})$ must hold. For example, suppose that L contains code x.m() where $\mathsf{this}_{\mathsf{m}}$, 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 $typeof(\mathsf{m}') <: typeof(\mathsf{m})$ (Section 2.3) which entails $\mathsf{this}_{\mathsf{m}} <: \mathsf{this}_{\mathsf{m}'}$, ensures that $\mathsf{this}_{\mathsf{m}'}$ 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.

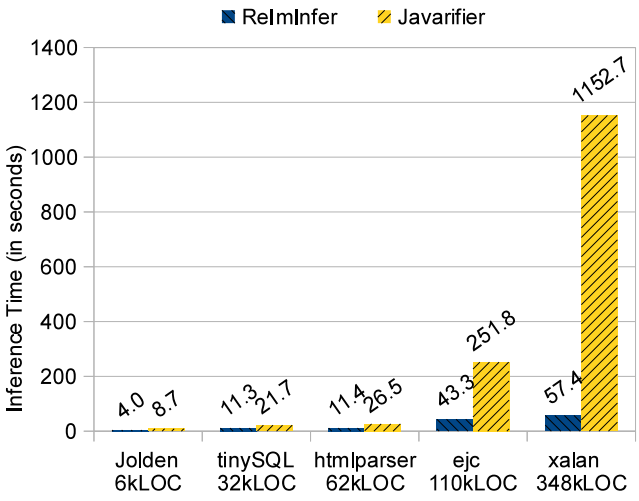

Figure 6. Runtime performance comparison. Note that the running time for type checking is excluded for both ReImInfer and Javarifier.

In all benchmarks, about 41% to 69% of references are reported as readonly, less than 16% are reported as polyread and 24% to 50% 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 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. The timings are the medians of three runs.

ReImInfer scales better than Javarifier. ReImInfer appears to scale approximately linearly. As the applications grow larger, the difference between ReImInfer and Javarifier becomes more significant. These results are consistent with the results reported by Quinonez et al. [26] 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

Benchmark	Code size			Annotatable References				Time (in seconds)		
	#Line	#Meth	#Pure	#Ref	#Readonly	#Polyread	#Mutable	Infer	Check	Total
JOlden	6223	326	175 (54%)	949	453 (48%)	149 (16%)	347 (37%)	4.0	1.6	5.7
tinySQL	31980	1597	965 (60%)	4247	2644 (62%)	418 (10%)	1185 (28%)	11.3	3.7	15.1
htmlparser	62627	1698	647 (38%)	4853	2711 (56%)	421 (9%)	1721 (35%)	11.4	5.3	16.9
ejc	110822	4734	1740 (37%)	15434	6161 (40%)	1803 (12%)	7470 (48%)	43.3	22.7	66.2
xalan	348229	10386	4019 (39%)	41186	25181 (61%)	3254 (8%)	12751 (31%)	57.4	23.3	81.1
javad	4207	140	60 (43%)	363	249 (69%)	19 (5%)	95 (26%)	2.2	0.9	3.2
SPECjbb	28333	529	200 (38%)	1537	830 (54%)	246 (16%)	461 (30%)	6.9	2.2	9.3
commons-pool	4755	275	94 (34%)	602	266 (44%)	37 (6%)	299 (50%)	2.7	1.0	3.8
jdbm	11610	446	139 (31%)	1161	470 (40%)	161 (14%)	530 (46%)	4.0	1.8	5.9
jdbf	15961	707	336 (48%)	2510	1669 (66%)	240 (10%)	601 (24%)	7.1	2.4	9.6
jtds	38064	1882	672 (36%)	5048	2805 (56%)	299 (6%)	1944 (39%)	12.1	5.0	17.2
java.lang	43282	1642	1128 (69%)	2970	2028 (68%)	187 (6%)	755 (25%)	8.6	3.0	12.1
java.util	59960	2724	1093 (40%)	6920	2852 (41%)	1005 (15%)	3063 (44%)	16.7	7.6	24.5

Table 1. Inference results for reference immutability. **#Line** shows the number of lines 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 variables 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, and method purity. **Check** is the running time for type checking. The last column **Total** shows the total running time, including reference immutability inference, purity inference and type checking.

```
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

not give identifiable names for local variables (it only shows local 0, local 1, and so on). In addition, polyread fields in Javarifier are called this-mutable. In the comparison, we view all such fields as polyread.

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 reported as readonly by Javarifier, while they are reported as mutable if this is mutated, by ReImInfer. 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 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 [30].

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 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 [30], 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.

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

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 [30] and JPure by Pearce [24]. We ran JPPA and JPure on the JOlden 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.

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 [30], 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 [30], 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. ReImInfer 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 17 differences out of 163 methods in total. 9 methods are not reachable according to JPPA. Of the remaining 8 differences, (a) 2 are due to limitations/bugs in JPPA and (b) 6 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 6 methods are reported as pure by JPPA but impure by ReImInfer; this is imprecision in ReImInfer. These methods are inferred as impure by ReImInfer because they are overridden by impure methods. This is an insurmountable imprecision for 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 [24]. 16 differences are caused by methods 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 [33] and its inference tool Javarifier [26], 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 ReIm with Javari, and Section 5.2 compares our inference approach, ReImInfer, with Javarifier. Section 5.3 discusses related work on purity inference, and Section 5.4 discusses other related work.

5.1 Comparison with Javari

There are two essential differences between ReIm and Javari [33]. First, Javari allows programmers to exclude fields from the abstract state by designating fields as assignable or mutable. Such a 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 or mutable fields and therefore it is less expressive but simpler. This decision is motivated by our intended application: purity inference. Including assignable and assignable for fields 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: a programmer can express designs such as "readonly list of readonly elements", "readonly list of mutable elements", "mutable list of readonly elements", and "mutable

list of mutable elements". ReIm does not support annotations on the type arguments when instantiating a parametric class, and can express only "readonly list of readonly elements" and "mutable list of mutable elements" (which it uses to approximate the two inexpressible designs). The difference between the two approaches is illustrated by the following example:

```
void m(List<Date> lst2) {
  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>. (Javarifier does not have an option to make it prefer the solution mutable List<mutable Date> over readonly List<mutable Date>.) Again, the primary motivation for the decision about ReIm's simpler design is the application we had in mind: purity inference. Purity is a single bit that summarizes whether any reachable datum may be modified, and finer-grained information is not of use when computing whether a method is pure.

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

```
void m(Date[] b) {
  b[0].setHours(2);
}
```

Again, Javari and Javarifier permit a programmer to give the array and its elements either the same or different mutability annotations. ReIm and ReImInfer enforce that the array and its elements have the same mutability annotation, so the array reference b would be inferred as mutable Date mutable [] due to the mutation of element 0.

One might imagine inferring method purity from Javarifier's output, as follows: a method is pure if all the mutabilites of its formal parameters and static variables, and their type arguments and array elements, are readonly. This approach is sound but can be unnecessarily conservative, in certain circumstances. A concrete example is when the type argument is not part of the state of the object but is mutated. Consider the following example:

```
class A<T> {
  T id(T p) { return p; }
}

void m(A<Date> x) {
  Date d = x.id(new Date());
  d.setHours(0);
}
```

Here Javarifier infers that x is of type readonly A<mutable Date>. Using the proposed approach, method m would be conservatively marked as non-pure. By contrast, ReImInfer annotates x as readonly, so m is inferred to be pure.

Another important (but non-essential for our purpose) difference between Javari and ReIm 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 assignment 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 [34] while ReImInfer is based on the Checker Framework (CF), 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 the Checker Framework. In contrast, ReImInfer and the type checker require only the Checker Framework and the annotation utility. These differences contribute to the usability of ReImInfer.

We conjecture that viewpoint adaptation, the constraint resolution approach, and the better infrastructure in the 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 [24] 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 [28]. 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 requires a 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 [36]. 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.

The IGJ [38] and OIGJ [39] type systems support both reference immutability (a la Javari and ReIm) and also object immutability. Concurrent work by Haack et al. [16] also supports object immutability.

JQual [15] 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.

Porat et al. [25] 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 [20] describe field immutability in the context of UML. Their work incorporates limited context sensitivity, analyzes large libraries and focuses on instance fields. This work is an improvement over [20]. 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.

ReIm 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 ReIm 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 [12, 18]. The type inference algorithm presented in this paper, fits in the framework from [18]. One difference is that viewpoint adaptation in [18] 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 [18] and this work, will be formalized in future work.

The inference algorithm (Section 2.4) of ReImInfer is similar to the algorithm used by Tip et al. [14, 31]. Both algorithms start with sets containing all possible answers and iteratively remove elements that are inconsistent with the typing rules. Then, they use a ranking over valid typings to select from the multiple options that remain.

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 purity inference tools, JPPA and JPure.

We envision several potential applications of ReIm and ReImInfer. We have already used ReImInfer to infer method purity which is needed, for example, for Universe Type inference [12, 18]. Other applications are flow-sensitive type inference, error detection in concurrent programs, and optimization of concurrent programs.

In future work, we plan to develop a framework for inference and checking of pluggable types, which will include ReIm, ownership types, Universe Types, as well as other type systems such as AJ [35], EnerJ [29], etc. In addition, we intend to study the interesting relationship between context sensitivity and CFL-reachability from data-flow analysis and viewpoint adaptation from ownership types.

Acknowledgments

This research was supported in part by NSF grants CCF-0642911 and CNS-0855252 and by DARPA contracts FA8750-12-2-0107 and FA8750-12-C-0174.

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] W. Dietl, M. D. Ernst, and P. Müller. Tunable static inference for Generic Universe Types. In *ECOOP*, pages 333–357, 2011.

[13] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, July 3, 2012.

[14] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, July 2005.

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

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

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

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

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

[20] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.

[21] A. Milanova and W. Huang. Static object race detection. In *APLAS*, pages 255–271, 2011.

[22] 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.

[23] 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.

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

[25] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, pages 10–24, 2000.

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

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

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

[29] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[30] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[31] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. D. Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3): 9:1–9:47, May 2011.

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

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

[34] 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.

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

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

[37] J. Zhao, I. Rogers, and C. Kirkham. Pure method analysis within Jikes RVM. In *ICOOOLPS*, 2008.

[38] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE*, pages 75–84, Sept. 2007.

[39] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic Java. In *OOPSLA*, pages 598–617, Oct. 2010.