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

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

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

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

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

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

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

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

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

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

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

A type system enforcing reference immutability has a number of benefits. It improves the expressiveness of interface design by specifying the immutability of parameters and return values; it helps prevent and detect errors caused by unwanted object mutations; it facilitates reasoning about and proving other properties such as object immutability.
and method purity; it supports enforcement of the owner-as-modifier discipline \cite{9,10}.

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

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

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

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

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

In summary, we make the following contributions:

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

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

2. ReIm Reference Immutability Types

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

2.1 Immutability Qualifiers

There are three immutability qualifiers in our type system:

- **mutable**: A mutable reference can be used to mutate the referenced object; this is the implicit and only option in standard object-oriented languages.
- **readonly**: A readonly reference $x$ cannot be used to mutate the referenced object nor anything it references. For example, all of the following are forbidden:
  - $x.f = z$
  - $x.set(z)$ where set sets a field of its receiver
  - $y = \text{id}(x); y.f = z$ where id is a function that returns its argument
  - $x.f.g = z$
  - $y = x.f; y.g = z$
- **polyread**: A polyread reference $x$ cannot be used to mutate the referenced object. A method may return $x$ to the caller, and the caller may mutate the object. Programmers should use polyread when the reference is readonly in the scope of the enclosing method, but depends on the context of the caller of the method. For example,
• \( x.f = 0 \), where \( x \) is polyread, is not allowed, but
• \( z = \text{id}(y) \); \( z.f = 0 \), where \( \text{id} \) is
  polyread \( X \) \( \text{id}(\text{polyread} \ X \ x) \) { return \( x; \)},
  is allowed when \( y \) and \( z \) are mutable.

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

The subtyping relation between the qualifiers is

\[
\text{mutable} <: \text{polyread} <: \text{readonly}
\]

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

2.2 Context Sensitivity

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

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

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

\[
\text{Date md} = \text{this.get\text{Date}}();
\]

The left-hand-side of the call is the mutable reference \( \text{md} \).
Therefore, field access expression `this.date` would become
mutable, which would force this of `getDate` to become
mutable as well (if `this.date` is of mutable, this means
that the current object was modified using this, which forces
this to become mutable). The mutability of this of `getDate`
will propagate, and force this of `m1` as well as this of `m2`
to become mutable.

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

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

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

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

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

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

Viewpoint adaptation \( q \triangleright_f q_f \) is applied at field accesses.
It adapts declared field qualifier \( q_f \) from the point of view of
receiver qualifier \( q \). We define \( \triangleright_f \) as:

\[
\cdot \triangleright_f \text{ readonly} = \text{readonly} \\
q \triangleright_f \text{ mutable} = q
\]

The underscore denotes a “don’t care” value. Consider field
access \( y.f \). If the type of receiver \( y \) is readonly and the
declared type of field \( f \) is mutable, then the type of \( y.f \)
is readonly \( \triangleright_f \text{ mutable} = \text{readonly} \). A field access \( y.f \) is
mutable if and only if both the receiver \( y \) and field \( f \) are
mutable. If the receiver or the field is readonly, \( y.f \) is readonly.
Finally, if the receiver is polyread and the field is mutable,
\( y.f \) is polyread.

Viewpoint adaptation \( q_x \triangleright_m q \) is applied at method calls
\( x = y.m(z) \). It adapts \( q \), the declared qualifier of a formal
parameter/return of \( m \), from the point of view of \( q_x \), the
can be viewed as allowing different viewpoint adaptation. We use viewpoint adaptation to encode variables or parameters. For simplicity, we assume all names are unique. Note that we distinguish the type qualifiers used for fields from those used for local variables/parameters.

A qualifier at the left-hand side \( x \). We define \( \triangleright_m \) as:

\[
\begin{align*}
\triangleright_m \text{ mutable} & = \text{ mutable} \\
\triangleright_m \text{ readonly} & = \text{ readonly} \\
q_x \triangleright_m \text{ polyread} & = q_x
\end{align*}
\]

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

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

### 2.3 Typing Rules

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

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

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

Rules \((\text{TNEW})\) and \((\text{TASSIGN})\) 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 \((\text{TWRITE})\) requires \( \Gamma(x) \) to be mutable because \( x \)'s field is updated in the statement. The adapt rules for field access are used in both \((\text{TWRITE})\) and \((\text{TREAD})\).

Rule \((\text{TCALL})\) demands a detailed explanation. Function \( \text{typeof} \) retrieves the type of \( m \). \( q_{\text{this}} \) is the type of implicit parameter \( \text{this} \), \( q_p \) is the type of the formal parameter, and \( q_{\text{ret}} \) is the type of the return. The rule requires \( q_f <: q_f \triangleright_m q_{\text{this}} \). When \( q_{\text{this}} \) is readonly or mutable, its adapted value is the...
same. Thus, when \( q_{\text{this}} \) is mutable due to a statement \( \text{this.f} = 0 \) for example,

\[
q_y <: q_x \triangleright_m q_{\text{this}} \quad \text{becomes} \quad q_y <: \quad \text{mutable}
\]

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

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

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

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

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

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

\[
q_{\text{md}} \triangleright_m q_{\text{this}} = \quad \text{mutable} \triangleright_m \text{polyread} = \quad \text{mutable}
\]

It is instantiated to readonly in `m2`:

\[
q_{\text{rd}} \triangleright_m q_{\text{this}} = \quad \text{readonly} \triangleright_m \text{polyread} = \quad \text{readonly}
\]

This allows this of `m2` to be typed readonly.

Method overriding is handled by the standard constraints for function subtyping. If \( m' \) overrides \( m \) we have

\[
\text{typeof}(m') <: \text{typeof}(m)
\]

and thus,

\[
q_{\text{this}_m} \cdot q_{\text{pm}} \to q_{\text{ret}_m} <: q_{\text{this}_m} \cdot q_{\text{pm}} \to q_{\text{ret}_m}
\]

This entails \( q_{\text{this}_m} <: q_{\text{this}_m} \cdot q_{\text{pm}} <: q_{\text{pm}} \) and \( q_{\text{ret}_m} <: q_{\text{ret}_m} \).

2.4 Type Inference

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

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

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

The inference analysis iterates over the statements in the program and refines the sets until either (1) a reference is assigned the empty set in which case the analysis terminates with an error, or (2) the iteration reaches a fixpoint.
Note that the result of fixpoint iteration is a mapping from references to sets. The actual mapping from references to types is derived as follows: for each reference \( x \) we pick the largest element of \( S(x) \) according to the preference ranking readonly > polyread > mutable, because we want to maximize the number of readonly references. Note that leaving all references as is also a valid typing but a useless one, as it expresses nothing about immutability. The following rather interesting propositions hold:

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

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

- Let \( \max(S(q_y)) \) be readonly. \( q_x \triangleright_m q_{ret} < q_y \), holds for any value of \( \max(S(ret)) \). If \( \max(S(q_{this})) \) is readonly or polyread, \( q_y \triangleright_q q_{this} \), holds for any value of \( \max(S(y)) \). If \( \max(S(q_{this})) \) is mutable, the only possible \( \max \) for \( y \) would be mutable (the others would have been removed by the transfer function).

- Let \( \max(S(q_y)) \) be mutable. If \( \max(S(ret)) \) is polyread, clearly \( q_x \triangleright_m q_{ret} < q_y \), holds. \( \max(S(ret)) \) cannot be readonly, readonly would have been removed by the transfer function.

- Let \( \max(S(q_y)) \) be polyread. If \( \max(S(ret)) \) is polyread, clearly \( q_x \triangleright_m q_{ret} < q_y \), holds. \( \max(S(ret)) \) cannot be readonly, readonly would have been removed by the transfer function.

If \( \max(S(q_{this})) \) is readonly or polyread, \( q_y \triangleleft q_x \triangleright_m q_{this} \), holds for any value of \( \max(S(y)) \). If \( \max(S(q_{this})) \) is polyread, the only possible value for \( \max(S(y)) \) would be mutable. If \( \max(S(q_{this})) \) is mutable, the only possible \( \max \) for \( y \) would be mutable as well (the others would have been removed by the transfer function).

\( (TCALL) \) \( x = y.m(z) \). The rest of the cases are straightforward.

- Let \( \max(S(q_y)) \) be polyread. If \( \max(S(ret)) \) is polyread, clearly \( q_x \triangleright_m q_{ret} < q_y \), holds. \( \max(S(ret)) \) cannot be readonly, readonly would have been removed by the transfer function.

If \( \max(S(q_{this})) \) is readonly, \( q_y \triangleleft q_x \triangleright_m q_{this} \), holds for any value of \( \max(S(y)) \). If \( \max(S(q_{this})) \) is polyread, the only possible value for \( \max(S(y)) \) would be polyread or mutable. If \( \max(S(q_{this})) \) is mutable, the only possible \( \max \) for \( y \) would be mutable.

\( \square \)

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

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

\( \square \)

These propositions are validated empirically as detailed in Section 4. To validate Proposition 2.1, we build an independent type checker in the Checker Framework and type check the inferred types. To validate Proposition 2.2, we perform detailed comparison with JavaInfer, the state-of-the-art tool for inference of reference 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 iterates
We introduce a static immutability type \( q_m \). A method is impure if it mutates (directly, or indirectly through callees), a static field, or objects reachable from a static field. We can also infer that method \( m \) is impure if it accesses static state through some parameters, and (2) it does not mutate prestates reachable through static fields. The definition allows a pure method to create and mutate local objects, as well as return a newly constructed object as a result.

For a method that does not access static fields, the prestates it can reach are the objects reachable from the actual arguments and the method receiver. Therefore, if any of the formal parameters of \( m \) or implicit parameter this, is inferred as mutable by reference immutability inference, \( m \) is impure. Otherwise, i.e., if none of the parameters is inferred as mutable, \( m \) is pure. Consider the implementation of List in the left column of Figure 4. For method add, reference immutability inference infers that both \( m1 \) and this are mutable, i.e. the objects referred by them are mutated in add. When there is a method invocation \( \text{lst.add(node)} \), we know that the prestates referred by the actual argument node and the receiver lst are mutated. As a result, we can infer that method add is impure. We can also infer that method reset is impure because implicit parameter this is inferred as mutable by reference immutability inference. Method size is inferred as pure because its implicit parameter this is inferred as readonly and it has no formal parameters.

However, the prestates can also come from static fields. A method is impure if it mutates (directly, or indirectly through callees), a static field, or objects reachable from a static field. We introduce a static immutability type \( q_m \) for each method \( m \), \( q_m \) can be readonly or mutable but not polyread. Roughly, \( q_m \) is mutable when \( m \) accesses static state through some static field, and then mutates this static state; \( q_m \) is readonly otherwise. Static immutability types are computed using reference immutability. We introduce a function \( \text{statictypeof} \) which retrieves the static immutability type of \( m \):

\[
\text{statictypeof}(m) = q_m
\]

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

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

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

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

**Figure 4.** A simple linked list
well, because it invokes method m2 does not account for all mutations of static state in m and local variable o System.out referred by mutable im or immutability type of readonly fixpoint. If S as reference immutability types. The analysis initializes every for modularity. ensures that m m even if In other words, if m m becomes mutable if the static immutability type of one of its callees is mutable (i.e., a static field may be mutated in a callee of m). This is expressed by rule (TCALL).

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

$q_m <: q_m'$

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 $S(m)$ to {readonly, mutable} and iterates over the statements in Figure 4 and the overriding constraints, until it reaches the fixpoint. If readonly remains in $S(m)$ at the end, the static immutability type of m is readonly; otherwise, it is mutable.

Consider the right column of Figure 3 $q_m$ becomes mutable because m1 assigns l to the static field sLst. $q_m$ 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. $q_m$ becomes mutable as well, because it invokes method m2 and $q_m$ is mutable.

The observer has likely noticed that $q_m$ 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:

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

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

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

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

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 (TREAD) or (TWRITE), and the mutation is captured by the static immutability type $q_m$.

4. Experiments

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

4.1 Benchmarks

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

Whole programs:
- Java Olden (JOlden) is a benchmark suite of 10 small programs.
- ejc-3.2.0 is the Java Compiler for the Eclipse IDE.
- javad is a Java class file disassembler.
- SPECjbb 2005 is SPEC’s benchmark for evaluating server side Java.
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 [25] as well as other related analyses such as JPPA [25]; these methods are intended to be observationally pure. Our analysis does not detect bugs due to unintended mutation in these methods.

ReImInfer treats private fields f that are read or written through this in exactly one method m, as if they were local variables. Precisely, this means that for these fields we allow qualifier polyread, and treat field reads x = this.f and writes this.f = x as if they were assignments x = f and f = x. One such field 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: qthis <= qt. Thus, when f is mutated in m, f and this are inferred as mutable. When f is readonly in the scope of m, but depends on the context of the caller, f is polyread and this is polyread or mutable. Otherwise, f is readonly. As an example, current and this of nextElement() in Figure 7 are both inferred polyread. The motivation behind this optimization is precisely the Enumeration class in Figure 7. The goal is to transfer the dependence from the element stored in the container, to the container itself, which is important for purity inference. If current was treated as a field, it would be mutable, and therefore, this of elements would be mutable, which entails that every container that creates an enumeration is mutable, even if its elements were not mutated. If current was excluded from abstract 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 -> polyread. The mutable parameters assume worst-case behavior of the unknown callee — the unknown callee mutates its arguments. The polyread return is an appropriate choice because readonly would have been too restrictive for the caller. User code U, which uses previously analyzed library L, is analyzed separately using the result of the analysis of L. In our case, when analyzing user code U, we use the annotated JDK available with Javafier in CF; the similarities between Javari and ReImInfer justify this use. Correctness of the composition is ensured by the check that the function subtyping constraints hold: for every m’ in U that overrides an m from L, typeof (m’) <= typeof (m) must hold. For example, suppose that L contains code x.m() where this_m, is inferred as readonly. The typing is correct even in the presence of callbacks. If x.m() results in a callback to m’ in U (m’ overrides m), constraint typeof (m’) <= typeof (m) which entails this_m <= this_m’, ensures that this_m’ is readonly as well.

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

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

4.2.1 Inference output

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

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

To summarize our findings, ReImInfer is more scalable than Javafier (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 Javafier on the first 5 benchmarks in Table 1. ReImInfer and Javafier analyze exactly the same set of classes (given

Libraries:

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

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

We added 392 empty methods in tinySQL in order to compile it with Java 1.6.
We exclude the following difference from the count: the
we compared our result with Javarifier on the first four
JOlden benchmark suite. We found 34 differences between
JOlden and so on). JOlden implements the
we examine only fields, return values, formal parameters and
JOlden (JOlden) benchmark suite. We found 34 differences between
Figure 6. Runtime Performance Comparison. Note that the
ence. Differences due to the annotated JDK are also excluded
Annotatable References include all references, including fields, local variables, return values, formal parameters and implicit parameters this. It does not include references of primitive type. #Ref is the total number of annotatable references, 
#Polyread and #Mutable are the number of references inferred as readonly, polyread and mutable, respectively. We also include the running time for the benchmarks. Infer is the running time of ReImInfer for inferring reference immutability. Purity is the time for inferring method purity based on the result of reference immutability. The last column Total shows the total running time, including source processing, reference immutability inference, purity inference and type checking.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code size</th>
<th>Annotatable References</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Line</td>
<td>#Meth</td>
<td>#Pure</td>
</tr>
<tr>
<td>JOlden</td>
<td>6223</td>
<td>326</td>
<td>175 (54%)</td>
</tr>
<tr>
<td>tinySQL</td>
<td>31980</td>
<td>1597</td>
<td>965 (60%)</td>
</tr>
<tr>
<td>htmlparser</td>
<td>6267</td>
<td>1698</td>
<td>642 (38%)</td>
</tr>
<tr>
<td>ejc</td>
<td>11082</td>
<td>4736</td>
<td>1701 (36%)</td>
</tr>
<tr>
<td>xalan</td>
<td>34829</td>
<td>10386</td>
<td>3942 (38%)</td>
</tr>
<tr>
<td>javad</td>
<td>4207</td>
<td>140</td>
<td>60 (43%)</td>
</tr>
<tr>
<td>SPECjbb</td>
<td>28333</td>
<td>529</td>
<td>195 (37%)</td>
</tr>
<tr>
<td>commons-pool</td>
<td>4755</td>
<td>275</td>
<td>94 (34%)</td>
</tr>
<tr>
<td>jdbm</td>
<td>11610</td>
<td>446</td>
<td>136 (30%)</td>
</tr>
<tr>
<td>jdbf</td>
<td>15961</td>
<td>707</td>
<td>304 (43%)</td>
</tr>
<tr>
<td>jids</td>
<td>38064</td>
<td>1882</td>
<td>671 (36%)</td>
</tr>
<tr>
<td>java.lang</td>
<td>43282</td>
<td>1642</td>
<td>1101 (67%)</td>
</tr>
<tr>
<td>java.util</td>
<td>39960</td>
<td>2727</td>
<td>1027 (38%)</td>
</tr>
</tbody>
</table>

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

Because Javarifier infers the return value of nextElement() becomes polyread. As a result, the local variable Object retval = current; current = current.next; return retval;

} return new Enumerate();
}

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

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 Javari’s handling of the annotated JDK.

4.3 Purity Inference

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

Our purity inference is modular. Reference immutability assumptions for unknown callees are exactly as before. Static immutability types, which we discussed in Section 3, are not available in Javari’s annotated JDK. We ran ReImInfer on the java.lang and java.util packages, and we assumed that other library methods have not mutated static fields. JPPA, a Java Pointer and Purity Analysis tool by Sălčianu and Rinard [25], makes the same assumption for unknown library methods, and our decision to use qm = 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 qm < qm’ in particular, if qm is inferred as readonly, then qm’ 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 static type on java.lang and java.util packages, and the vast majority of the violations are on the special-cased methods, equals, hashCode and toString. These violations are reported as warnings.

The results of purity inference by ReImInfer are shown in Table 1 column #Pure. To evaluate analysis precision, we compared with JPPA by Sălčianu and Rinard [25] and JPure by Pearce [20]. We ran JPPA and JPure on the JOlden
Table 2. Pure methods in Java Olden benchmarks

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

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

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

4.3.1 Comparison with JPPA

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

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

51 differences are due to limitations/bugs of JPPA. 38 differences are the constructors, which ReImInfer reports as pure but JPPA does not. According to [25], JPPA follows the JML convention and constructors that mutate only fields of the this object are pure. Thus, JPPA should have inferred them as pure. ReImInfer follows the same definition and reports these constructors as pure. There are 3 differences on methods that are inferred as pure by ReImInfer but impure by JPPA. These 3 methods that return newly-constructed objects, which are mutated later. According to the definition in [25], JPPA should have inferred them as pure. There is 1 difference on method loadTree in Jolden/BH. It is likely a bug in JPPA because the this parameter is passed to another object’s field which is mutated later, but JPPA reports loadTree as pure. ReImInfer detects 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 Javariifer. tinySQL is a library and there is no main method. htmlparser, which is a library as well, comes with a main, which exercises a portion of its functionality; JPPA threw an exception on htmlparser which we were unable to correct. JPPA completed on ejc. Due to the fact that it is a whole-program analysis, it analyzed 3790 reachable user methods; ReImInfer covered all 4734 user methods.

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

4.3.2 Comparison with JPure

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

29 differences are caused by different definitions of pure constructors. We follow the JML convention that a constructor is pure if it only mutates its own fields. JPure has different definition of a pure constructor and that leads to these differences. 2 differences are the nextElement method where ReImInfer considers the current field as a local variable as discussed above. There are 8 differences in toString methods, which are inferred as impure by JPure. Our examination shows that those methods are pure; it appears that they should be pure, but are inferred as impure due to imprecision in JPure, according to [20]. 16 differences are caused by meth-
ods that return fresh local references. JPure should have been able to identify them as @Fresh, but it did not. The remaining 5 differences are due to the static methods in java.lang.Math. JPure infers all methods that invoke the static methods in java.lang.Math as impure, while ReImInfer identifies that these methods satisfy $q_m$ is readonly by using the inference result from the java.lang package.

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

## 5. Related Work

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

### 5.1 Comparison with Javari

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

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

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

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

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

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

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

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

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

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

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

### 5.2 Comparison with Javarifier

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

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

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

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

### 5.3 Purity

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

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

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

Clausen develops Cream, an optimizer for Java bytecode using an inter-procedural side-effect analysis [9]. It infers an instruction or a collection of instructions as pure, rea-only, write-only or read/write, based on which it can infer purity for methods, loops and instructions. It is a whole-program analysis which relies the main method and also unused methods are not covered.

Other researchers also explore the dynamic notion of purity. Dallmeier et al. develop a tool, also called JPURE, to dynamically infer pure methods for Java [3]. Their analysis calculates the set of modified objects for each method invocation and determines impure methods by checking if they write non-local visible objects. Xu et al. use both static and dynamic approaches to analyze method purity in Java programs [30]. Their implementation supports different purity definitions that range from strong to weak. These dynamic approach depends on the runtime behavior of programs, which is totally different from our purity analysis.

### 5.4 Other Related Work

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

JQual [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.

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

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

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

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

6. Conclusion

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

References

We formulate the concrete semantics of reference immutability. The syntax is shown in Figure 8. The operational semantics associates runtime mutabilities, Readonly or Mutable, with each variable. It associates a local mutability and a global mutability to each reference variable. It associates a single, global mutability to each field. Below, we formalize the semantics.

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

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

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

The concrete semantics is shown in Figure 8. We skip the rule for object creation, because it does not create interesting dependences. Rule (\textsc{dassign}) creates an edge in \( C \) from \( y \) to...
\[
\begin{align*}
\text{(dassign)} & \quad (F \: C = y; s) \: S \: H \: G \: M \rightarrow (F \: C') \: s \: S \: H \: G \: M \\
\text{(dread)} & \quad C' = C \cup y \rightarrow x \\
\text{(fwrite)} & \quad C' = C \cup y \rightarrow f \\
\text{(dwrite)} & \quad C' = C \cup \{y \rightarrow f\} \\
\text{(dcall)} & \quad mbody(m) = \text{this } p \: y' \; s' \; \text{; return } \text{return(ret)} \\
\text{(dreturn)} & \quad mbody(m) = \text{this } p \: y' \; s' \; \text{; return } \text{return(ret)} \\
\end{align*}
\]

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

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

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

\begin{figure}[h!]
\centering
\begin{align*}
\text{A.2 Well-formedness}
\end{align*}

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

\begin{figure}[h!]
\centering
\begin{align*}
\text{A.3 Type Soundness}
\end{align*}

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

**Theorem A.1.** Preserved. If \( S \: H \: G \: M \) is WF and \( S \: H \: G \: M \rightarrow H' \: S' \: G' \: M' \), then \( H' \: S' \: G' \: M' \) is WF.

The proof is by structural induction on the derivition; it enumerates the different kinds of steps in the semantics, and shows that after each step, a well-formed configuration stays well-formed.
Proof. (Sketch) We show the most interesting cases, (DASSIGN), (DWRITE) and (DRETURN).

(DASSIGN). This step changes only C of the current frame. It adds an edge y → x to C. By ⊢ CT we have that x = y is well-typed, and therefore, Γ(y) ⊑ Γ(x). Thus, the frame with C' remains well-formed.

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

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

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

Proof. (Sketch) The prove is by structural induction on s when S = ⟨F C s⟩S'. It is immediate by applications of the corresponding rule for each s.