Definite Reference Mutability

Ana Milanova[1]
Dept. of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy NY, USA
milanova@cs.rpi.edu

Abstract

Reference immutability type systems such as Javari and ReIm ensure that a given reference cannot be used to mutate the referenced object. These systems are conservative in the sense that a mutable reference may be mutable due to approximation.

In this paper, we present ReM (for definite Re[ference] M[utability]). It separates mutable references into (1) definitely mutable, and (2) maybe mutable, i.e., references whose mutability is due to inherent approximation. In addition, we propose a CFL-reachability system for reference immutability, and prove that it is equivalent to ReIm/ReM, thus building a novel framework for reasoning about correctness of reference immutability type systems. We have implemented ReM and applied it on a large benchmark suite. Our results show that approximately 86.5% of all mutable references are definitely mutable.

2012 ACM Subject Classification Theory of computation → Program analysis, Software and its engineering → General programming languages

Keywords and phrases reference immutability, type inference, CFL-reachability, precision

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.25

Supplement Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.4.3.7`

Acknowledgements We thank the ECOOP 2018 PC and AEC for valuable suggestions, and the National Science Foundation for supporting our work through NSF grant 1319384.

1 Introduction

Reference immutability ensures that a *readonly* reference cannot be used to modify the state of the object, including its transitively reachable state. For example, in the code below

```
Date md = new Date();
readonly Date rd = md;
rd.setTime(1);
md.setTime(1);
```

the Date object cannot be modified through the readonly reference rd, however, the same object can be be modified through the mutable reference md.

Reference immutability has a wide variety of applications. It can enrich method specifications. It can help prevent errors due to unwanted aliasing and unwanted object mutation, as well as errors due to concurrency. It can enable compiler and runtime optimizations as well as reasoning about more complex properties such as method purity and object immutability. One application that has not received attention (to the best of our knowledge), is the impact

[1] This work was partially supported by NSF grant 1319384.

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 25; pp. 25:1–25:30
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of reference immutability on "flow systems". Flow systems track and prevent flow from *positive* references to *negative* ones:

```
a = b;
positive X x = ... ;
a.f = x;
negative X y = b.f;
```

Many interesting analyses fall into this category, most notably approximate computing systems (e.g., EnerJ [28]), which prevent flow of approximate values into precise ones, and taint systems, which prevent flow from sensitive sources to untrusted sinks (e.g., [29, 17]). Unfortunately, the natural subtyping negative <: positive is unsound in the presence of mutable references [3]. (In the above example, had we allowed for such subtyping, reference a could have been positive, b could have been negative, and the program would have type checked.) Therefore, flow systems disallow subtyping for reference types [29, 28, 12], forcing equality constraints at reference type assignments instead of the more precise subtyping constraints. Reference immutability can alleviate the imprecision arising from equality constraints – if the left-hand-side of the assignment is readonly, then subtyping is safe – allowing for more correct programs to type check. In summary, because of its many applications, reference immutability has been studied extensively [34, 39, 2, 40, 18, 13, 22], and it remains important to continue research in the area.

Javari [34] is the state-of-the art in reference immutability. ReIm [18] has similar core semantics but is less expressive and therefore simpler. In this paper we focus on ReIm because of its simplicity and clarity; we believe that our treatment extends to other reference immutability systems. Standard reference immutability systems, like Javari and ReIm capture what we call *definite immutability* – a readonly reference is truly immutable. However, a mutable reference may be truly mutable, or it may be mutable because of inherent approximation. ReIm (and Javari) approximate in the handling of structure-transmitted dependences [25] (i.e., flow through heap objects). For example, in the code below

```
x.f = y;   ...   w = z.f; w.g = ...
```

reference y is mutable. However, it is not necessarily mutable: if x and z refer to the same object o, then y is indeed mutable; if they refer to different objects, then it is not mutable (at least not because of the update to w). The system does not reason about aliasing, and errs on the safe side marking y mutable. ReIm (and Javari) handle call-transmitted dependences [25] precisely. In the code below, id is the standard "identity" function that returns its argument.

```
x = id(y);   // x is readonly
z = id(w);   // z is mutable
```

Reference y is readonly, and w is mutable. The system properly transmits mutability without mixing the two call sites.

The key contribution of our paper is reasoning about approximation. We propose a new type system ReM (for definite Re[ference] M[utability]). ReM captures definite immutability, and in addition it captures definite mutability – a mutable reference is now *definitely mutable*. We note that our use of "definitely mutable" is somewhat inaccurate. Of course, whether a given reference is ever mutated is undecidable for various reasons, e.g., it is undecidable whether a given statement is executed, or whether a given path is executed. We use it in the sense of *definitely mutable according to CFL-reachability*, which is a highly precise model of data dependence [25] and analyses are unlikely to improve upon it. ReM captures approximation explicitly by introducing the maybe qualifier. In the earlier example

$\mathsf{x.f = y;} \quad \ldots \quad \mathsf{w = z.f; w.g = \ldots}$

y is now maybe mutable. A key result is that empirically, approximation has limited impact – only about 13% of all ReIm-mutable references (about 6% of all references) are maybe mutable, leading to a conclusion that ReIm and ReM are precise, and therefore can be used to power client analyses.

Another contribution of our paper is the interpretation of reference immutability in terms of Context Free Language reachability, commonly referred to as CFL-reachability [27, 26, 25]. We propose a CFL-reachability system for inference of reference immutability, and prove that it is equivalent to the ReIm/ReM inference system, thus building a framework for reasoning about correctness, and proving ReIm and ReM correct. To the best of our knowledge, ReIm has not been proven correct, even though it has been used to power client analyses [17, 32]. We plan to extend our system for reasoning about approximation and correctness to flow systems [29, 28, 18, 17]. A CFL-reachability interpretation is beneficial for several reasons: (1) it defines the semantics of reference immutability type systems in terms of intuitive and well-known concepts, which may lead to wider applicability of reference immutability type systems in software engineering, and (2) it provides a framework for reasoning about approximation and correctness, not only for reference immutability type systems, but for the larger class of flow type systems as well.

This paper makes the following contributions.

- We present ReM, a novel type system for reference immutability. ReM captures explicitly definite mutability (in the CFL-reachability sense), and approximation.
- We interpret reference immutability in terms of CFL-reachability, and prove ReIm and ReM correct.
- We present an implementation and evaluation. We show that ReIm and ReM are precise – only 13% of mutable references (6% of all references) are maybe mutable. The implementation is publicly available online and has been evaluated and accepted by the ECOOP Artifact Evaluation committee.

The rest of the paper is organized as follows. Sect. 2 presents the mutability semantics based on CFL-reachability. Sect. 3 interprets ReIm in terms of the mutability semantics, and presents the novel system ReM. Sect. 4 establishes equivalence between the systems in Sects. 2 and 3. Sect. 5 presents the empirical evaluation, Sect. 6 discusses related work, and Sect. 7 concludes.

2 Mutability Semantics

2.1 Flow Graph

The mutability semantics builds a *flow graph* G that represents flow (data) dependences between variables. The nodes in the graph are program variables, e.g., x, y, this, and field access expressions, e.g., x.f, y.f, this.f. The edges capture flow from one variable/field access expression, to another. The goal is to capture deep reference (im)mutability with data dependence paths in G. For example, in

$\mathsf{x = y; z = x; z.f = w}$

cd	$::=$ class C extends D $\{\overline{fd}\ \overline{md}\}$	*class*
fd	$::= t$ f	*field*
md	$::= t$ m(t this, t x) $\{\ \overline{t\ \mathsf{y}}$ s; return y $\}$	*method*
s	$::=$ s; s $\mid$ x = new t $\mid$ x = y $\mid$ x.f = y $\mid$ y = x.f $\mid$ x = y.m(z)	*statement*
t	$::= q$ C	*qualified type*

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. Qualifiers q range over ReIm/ReM qualifiers (defined in Sect. 3).

y is mutable, because there is a path in G from y to z, which is the receiver of the update at field write z.f = w. Throughout the paper we refer to receivers at field writes as *updates*.

We restrict our core language to a "named form" in the style of Vaziri et al. [35, 10]. The language models Java with the syntax in Fig. 1, where the results of instantiations, field accesses, and method calls, 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.

An assignment statement contributes a *direct* (i.e., intraprocedural) edge as follows:

$$\mathsf{x = y} \quad \Rightarrow \quad \mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x}$$

It represents flow from variable y to variable x. Therefore, if x is an update, i.e., there is field write x.g = z, the direct edge propagates mutability to reference y.

A field write statement x.f = y contributes a direct edge from y to the field access node x.f, and an *approximate* edge from x.f to every $\mathsf{x'.f} \in G$, where x′.f is the right-hand-side of a field read y′ = x′.f. (Without loss of generality we may assume $\mathsf{x'} \neq \mathsf{x}$.)

$$\mathsf{x.f = y} \quad \Rightarrow \quad \mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f}$$

We elaborate upon approximate edges shortly. A field read statement y′ = x′.f contributes direct edges as follows:

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

Edge $\mathsf{x'} \xrightarrow{\mathsf{d}} \mathsf{x'.f}$ accounts for deep (im)mutability. It links x′ to y′, propagating mutability back to x′ when y′ is update.

Therefore, together a pair of field write x.f = y and field read y′ = x′.f contribute a triple

$$\mathsf{x.f = y,\ y' = x'.f} \quad \Rightarrow \quad \mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f} \xrightarrow{\mathsf{d}} \mathsf{y'}$$

creating a path from y to y′. It models flow through heap objects while completely avoiding heap objects. In terms of Reps' terminology [25], our mutability semantics, like ReIm, models structure (i.e., heap)-transmitted dependences approximately.

The approximate edge makes approximation explicit. The approximate path from y to y′ propagates mutability from y′ back to y, but "with an asterisk". This is *maybe* mutability – if x.f and x′.f are aliases because x and x′ point to the same object, then y is truly mutable, however, if they are not aliases, y is not mutable due to this path. ReIm (and other reference immutability systems) overapproximate, and mark y mutable. A key insight of our work is that the impact of approximate paths is quite muted. Generally, when there is an approximate

```
class DateCell {
  Date date;

  DateCell(DateCell this, Date p) {
  this.date = p;
  }
  Date getDate(DateCell this) {
  return this.date;
  }
  void cellSetHours(DateCell this) {
  Date md = this.getDate();
  md.setHours(1);
  }
  int cellGetHours(DateCell this) {
  Date rd = this.getDate();
  int hour = rd.getHours();
  return hour;
  }

  public static void main(String[] args) {
  Date d = new Date();
  DateCell dc = new DateCell(d);
  ...
  }
}
```

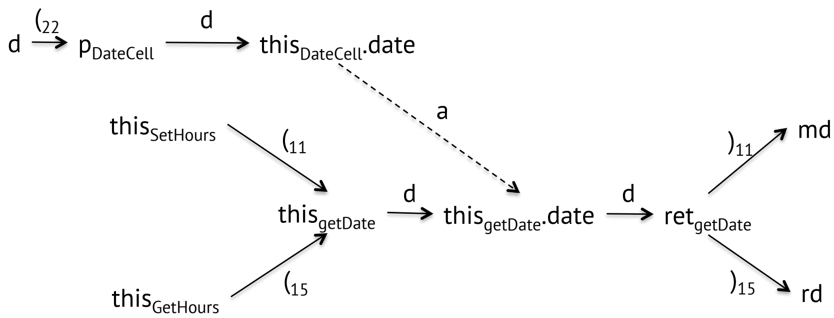


Figure 2 Running example. Code (adapted from Huang et al. [18]) and corresponding graph.

path from a reference y to an update, there is also a direct path from y to an update, and y would have become mutable regardless of the approximate path.

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

$$i: \mathsf{x} = \mathsf{y.m(z)} \quad \Rightarrow \quad \mathsf{y} \xrightarrow{(_i} \mathsf{this} \quad \mathsf{z} \xrightarrow{(_i} \mathsf{p}$$

Here this and p are the parameters of the compile-time target of the call. The standard CFL-reachability annotation $(_i$ marks call entry at call site i. A method return (method exit) creates a *return* edge from the return value to the left-hand-side of the call assignment:

$$i: \mathsf{x} = \mathsf{y.m(z)} \quad \Rightarrow \quad \mathsf{ret} \xrightarrow{)_i} \mathsf{x}$$

The standard CFL-reachability annotation $)_i$ marks a return at site i. In terms of Reps' terminology, the semantics models call-transmitted dependences precisely.

$$
\begin{array}{lcl}
E & ::= & R' \mid R \mid C \mid M \\
R' & ::= & R\,C \\
R & ::= &)_i \mid)_i\,M \mid)_i\,R \mid M\,R \\
C & ::= & (_i \mid (_i\,M \mid (_i\,C \mid M\,C \\
M & ::= & \mathsf{d} \mid (_i\,M\,)_i \mid M\,M
\end{array}
$$

Figure 3 A context-free grammar for exact paths, i.e., paths that account (solely) for call-transmitted dependences. M captures matched-parentheses strings, e.g., $(_i\ \mathsf{d}\ )_i$, C captures strings with one or more outstanding calls, e.g., $(_i\ \mathsf{d}\ (_j\ \mathsf{d}\ )_j$, and R and R' capture strings with one or more outstanding returns, e.g., $\mathsf{d}\ )_j$.

Since the goal is to capture dependences between variables, the semantics eschews objects and object creation. Fig. 2 shows an example including all kinds of statements and their corresponding edges.

2.2 Paths in Flow Graph

We classify paths in G into two categories: (1) *exact* paths, which do not contain approximate edges, and (2) *approximate* paths, which contain approximate edges. In our running example in Fig. 2, $\mathsf{this}_{\mathsf{getDate}} \rightsquigarrow \mathsf{rd}$ is an exact path, while $\mathsf{d} \rightsquigarrow \mathsf{md}$ is an approximate path. (We use squiggle arrows $\rightsquigarrow$ to denote multi-edge paths.) Not all paths in G are well-formed, and different well-formed paths have different meaning.

2.2.1 Exact Paths

Fig. 3 defines a context-free grammar that classifies exact paths into 3 categories. This grammar is standard in CFL-reachability theory. There is an *M-path* from node n to node u if and only if the edge annotations on the path form a string in the language described by M. M-paths are paths with matched parentheses. For example, path $\mathsf{this}_{\mathsf{GetHours}} \rightsquigarrow \mathsf{rd}$ is an M-path. However, $\mathsf{this}_{\mathsf{GetHours}} \rightsquigarrow \mathsf{md}$ is not a well-formed path because call edge $(_{15}$ and return edge $)_{11}$ do not match.

There is a *C-path* from n to u if and only if the edges from n to u form a string in the language described by C. More intuitively, these are paths with outstanding call edges. For example, $\mathsf{this}_{\mathsf{GetHours}} \rightsquigarrow \mathsf{ret}_{\mathsf{getDate}}$ is a C-path. With respect to reference immutability, if there is an M-path or a C-path from x (or from x.f) to an update, then x (or x.f) is definitely mutable, in the sense that analysis generally cannot improve from mutable. Because M-paths and C-paths have the same effect, from now on we refer to them as $M|C$-paths.

The third category is *R-paths*. There is an *R-path* from n to u if and only if the edges form a string in R or R'. That is, the path starts with outgoing return edges, and it may or may not descend into a call path before reaching u. For example, $\mathsf{this}_{\mathsf{getDate}} \rightsquigarrow \mathsf{md}$ is an R-path. With respect to reference immutability, if there is an R-path from x (or x.f) to an update, then x (or x.f) is *polymorphic*. It is mutable in some contexts of invocation of the enclosing method, and readonly in other. For example, $\mathsf{this}_{\mathsf{getDate}}$ and $\mathsf{ret}_{\mathsf{getDate}}$ are polymorphic. They are interpreted as mutable when getDate is called from cellSetHours, and they are interpreted as readonly when getDate is called from cellGetHours.

2.2.2 Approximate Paths

The following grammar rules capture approximate paths:

$$A \quad ::= \quad \mathsf{a} \mid \mathsf{a}\,E \mid \mathsf{a}\,A \mid E\,A$$

There is an A-*path* from n to u if and only if the edges form a string in A. Pictorially, an A-path consists of exact paths and approximate edges. For example,

$$n \overset{E}{\rightsquigarrow} \cdot \overset{\mathsf{a}}{\dashrightarrow} \cdot \overset{E}{\rightsquigarrow} \cdot \overset{\mathsf{a}}{\dashrightarrow} \cdot \overset{E}{\rightsquigarrow} u$$

is an A-path, and so is

$$n \overset{\mathsf{a}}{\dashrightarrow} \cdot \overset{E}{\rightsquigarrow} \cdot \overset{\mathsf{a}}{\dashrightarrow} u$$

The only mandatory component of the A-path is the one approximate edge.

A-paths fall into two categories, $(M|C)A$-paths, and RA-paths determined by the leading exact path:

(1) if the leading exact path is an $M|C$-path, then there is a $(M|C)A$-path. For example,

$$\mathsf{d} \xrightarrow{(_{20}} \mathsf{p}_{\mathsf{DateCell}} \xrightarrow{\mathsf{d}} \mathsf{this}_{\mathsf{DateCell}}.\mathsf{date} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{this}_{\mathsf{DateCell}}.\mathsf{date} \xrightarrow{\mathsf{d}} \mathsf{ret}_{\mathsf{getDate}} \xrightarrow{)_9} \mathsf{md}$$

is a $(M|C)A$-path.

(2) if the leading exact path is an R-path, then there is an RA-path. For the rest of the paper we use the term R-path to denote both the exact R-path and the RA-path as they have the same effect for our purposes.

Standard reference immutability type systems (e.g., ReIm), conservatively mark mutable every reference x, such that there is a $(M|C)A$-path from x to an update. As we mentioned earlier, an approximate path introduces uncertainty rather than definite mutability. The key observation of our work is the following. The majority of references x that exhibit an A-path from x to an update, also exhibit a "parallel" M-path or C-path to a (potentially different) update. Therefore, x is indeed definitely mutable and the A-path has no ill impact; an analysis that attempts to handle A-paths, i.e., structure-transmitted dependences, more precisely would not do better regarding x. Roughly speaking, our analysis separates the A-paths that do exhibit a "parallel" path to an update, from the A-paths that do not, thus separating references that are definitely mutable, from ones that are *maybe mutable*. If an analysis that treats structure-transmitted dependences more precisely is to realize precision improvement, the improvement is bounded by the number of maybe mutable references.

3 Type Systems

This section presents two reference immutability type systems, Huang et al.'s [18] ReIm, and our novel proposal ReM. ReIm captures *definite reference immutability*, that is, readonly references in ReIm are guaranteed immutable, however, mutable references are not necessarily mutable. ReM captures definite immutability and definite mutability – in ReM readonly references are still guaranteed immutable, and in addition, mutable references are guaranteed mutable (in the CFL-reachability sense).

The reader may wonder why one needs type-based reference immutability like ReIm and Javari, when one has a clear semantics expressed in terms of standard CFL-reachability. First, type-based reference immutability is studied extensively in the literature [34, 39, 40, 18, 13, 22]; its connection to CFL-reachability brings new insights. Second, type-based reference immutability allows programmers to specify immutability requirements with type qualifiers, e.g., readonly x, and take advantage of systems such as JSR 308 and the Checker Framework (`https://checkerframework.org/`) to check these immutability requirements; such requirements cannot be easily expressed or checked using CFL-reachability. Third, type systems promote modularity, while CFL-based systems are typically whole-program analyses. Yet another advantage comes when reasoning about complexity. While CFL-reachability is $O(N^3)$, ReIm/ReM inference is $O(N^2)$, where N is the program size.

Sect. 3.1 outlines ReIm, largely following Huang et al. [18]. We add a new interpretation in terms of our mutability semantics. Sect. 3.2 builds ReM upon the discussion in Sect. 3.1. Sect. 3.3 discusses type inference for ReIm and ReM.

3.1 ReIm

3.1.1 ReIm Qualifiers

The ReIm type system has three immutability qualifiers: mutable, readonly, and poly. We explain the qualifiers in terms of the mutability semantics defined in Sect. 2.

- mutable: A mutable reference x can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages. In terms of our mutability semantics, a mutable reference denotes an $M|C$-path, or a $(M|C)A$-path from x to an update.
- readonly: readonly captures "deep" immutability. A readonly reference x cannot be used to mutate the referenced object nor anything it references. All of the following are forbidden:
  - x.f = y
  - x.set(z) where set sets a field of its receiver x
  - z = id(x); z.f = w
  - y = x.f; y.g = z

  In terms of the mutability semantics, a readonly reference means that there does not exist either an exact or an approximate path to an update.
- poly: This qualifier expresses polymorphism over immutability. poly denotes that a reference is interpreted as mutable in some contexts, and it is interpreted as immutable in other contexts. The enclosing method *does not* mutate the reference, however, mutation to the reference or one of its components may happen after return. In terms of the mutability semantics, a poly reference denotes that there is an R-path from x to an update – the reference "flows" out of its enclosing method where it is mutated in some caller context.

The subtyping relation between the qualifiers is

$$\mathsf{mutable} <: \mathsf{poly} <: \mathsf{readonly}$$

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 poly or readonly one, but it is not allowed to assign a readonly reference to a poly or mutable one.

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

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

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

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \qquad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \qquad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \qquad typeof(\mathsf{m}) = q_{\mathsf{this}}, q_{\mathsf{p}} \to q_{\mathsf{ret}} \qquad q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd q_{\mathsf{this}} \qquad q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd q_{\mathsf{p}} \qquad q_{\mathsf{x}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ \text{(TCALL)}$$

Figure 4 Typing rules. Function *typeof* retrieves the qualifiers of fields and methods. Γ is a type environment that maps variables to their immutability qualifiers.

3.1.2 Typing Rules

ReIm is independent of the Java type system, which allows us to specify typing rules solely over type qualifiers q. The typing rules, following [18] are presented in Fig. 4. Rule (TASSIGN) is straightforward. It requires that the left-hand-side is a supertype of the right-hand-side. The system does not enforce object immutability and only mutable objects are created. The object creation rule becomes redundant and we omit it, just as we did in Sect. 2.

Rules (TREAD), (TWRITE) and (TCALL) make use of *viewpoint adaptation*, a concept from Universe Types [8, 9, 7]. Viewpoint adaptation of a type q' from the point of view of another type q, results in the adapted type q''. This is written as $q \rhd q' = q''$.

Below, we explain viewpoint adaptation in terms of the mutability semantics. At field accesses (TREAD) and (TWRITE) $\rhd$ adapts the field f from the viewpoint of (context of) the receiver. Viewpoint adaptation at field access handles structure-transmitted dependences, *approximately*. At method calls $\rhd$ adapts formal parameters and the return value from the point of view of the *variable at the left-hand-side* of the call assignment. This variable captures the calling context i. Viewpoint adaptation at calls handles call-transmitted dependences, *precisely*.

Notably, ReIm restricts fields to readonly or poly. Javari [34] does allow for mutable fields, increasing expressiveness and allowing Javari to express common idioms such as caching. However, mutable fields complicates the system. Declaring a field mutable in Javari excludes it from the state of the enclosing object, and adaptation of a mutable field requires special treatment, as discussed in [34, 18]. One can similarly allow mutable fields in ReIm/ReM. However, we are interested in type inference, and allowing mutable fields would create ambiguity: if a field access expression x.f is inferred mutable, do we infer that field f is mutable and is excluded from the state of a readonly x, or do we infer that f is just a "regular" field and a mutable x.f signals deep mutation of x and x must be mutable? Restricting fields to {readonly, poly} chooses the latter, as there is no way to know, without programmer annotations, which fields are caches and thus excluded from the object state. Javarifier [23], Javari's inference tool, makes the same choice. Javari is more expressive than ReIm, but its "inferable" semantics appears to be the same as ReIm's: Huang et al. [18] report essentially identical inference result for Javarifier and ReIm.

Following [18], we define $\triangleright$ as follows:

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

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

Let us take a closer look at rules (TWRITE) and (TREAD). For a pair of field write x.f = y and field read y′ = x′.f, the rules entail the following constraints:

$$q_{\mathsf{y}} <: q_{\mathsf{x}} \triangleright q_{\mathsf{f}} \qquad q_{\mathsf{x}'} \triangleright q_{\mathsf{f}} <: q_{\mathsf{y}'}$$

Suppose y′ is an update, i.e., there is statement y′.f = z, and $q_{\mathsf{y}'}$ is thus mutable. Therefore, q_{f} must be poly. First, recall that $q_{\mathsf{f}} \in \{\mathsf{readonly}, \mathsf{poly}\}$. Since readonly adapts to readonly, $q_{\mathsf{x}'} \triangleright q_{\mathsf{f}} <: \mathsf{mutable}$ does not type check, locking $q_{\mathsf{f}} = \mathsf{poly}$. (TWRITE) sets q_{x}, the type of the receiver to mutable. This serves two purposes in ReIm, (1) to account for the update of x, and (2) to account for the structure-transmitted dependence, i.e, the approximate path from y to the (eventual) update y′. Thus, $q_{\mathsf{x}} \triangleright q_{\mathsf{f}}$ evaluates to mutable, forcing q_{y} to be mutable as well. As mentioned earlier, ReIm handles approximate paths conservatively. If there is an $(M|C)A$-path from a reference y to an update, then ReIm's rules force y to be mutable, as is the case above, even though x and x′ may refer to different runtime objects.

Now, consider rule (TCALL). Function *typeof* retrieves the type of compile-time target 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 $q_{\mathsf{x}} \triangleright q_{\mathsf{ret}} <: q_{\mathsf{x}}$, which accounts for R-paths. The 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 call assignments to m are readonly, can the return type of m be readonly; otherwise, it is poly. A programmer can annotate the return type of m as mutable. However, this typing is pointless, as it unnecessarily forces local variables and parameters in m to become mutable when they may remain less restrictively poly. In Fig. 2, md = this.getDate(); entails constraint

$$q_{\mathsf{md}} \triangleright q_{\mathsf{ret}_{\mathsf{getDate}}} <: q_{\mathsf{md}} \quad \equiv \quad \mathsf{mutable} \triangleright q_{\mathsf{ret}_{\mathsf{getDate}}} <: \mathsf{mutable}$$

leading to $q_{\mathsf{ret}_{\mathsf{getDate}}} = \mathsf{poly}$. This accounts for the R-path from ret to the update through md. Continuing with the example, the field read in DateCell.getDate (line 8) entails constraint

$$q_{\mathsf{this}_{\mathsf{getDate}}} \triangleright q_{\mathsf{date}} <: q_{\mathsf{ret}_{\mathsf{getDate}}}$$

leading to $q_{\mathsf{this}_{\mathsf{getDate}}} = \mathsf{poly}$, which accounts for the R-path from $\mathsf{this}_{\mathsf{getDate}}$ to the update md.

Additionally, rule (TCALL) 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. This accounts for C- and CA-paths. The interesting case is when q_{this} is poly. A poly parameter this reflects a dependence between this and ret of m, such as the one in Fig. 2:

```
Date getDate(Date this) { ret = this.date; }
```

It allows the this object (or some part of it, in our example the date part of it), to be modified in caller context, after m's return. The type system entails that whenever there is intraprocedural dependence between this and ret, we have

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

Recall that when there exists a context where the left-hand-side variable x is mutated, q_{ret} must be poly. Therefore, constraint $q_{\mathsf{this}} <: q_{\mathsf{ret}}$ forces q_{this} to be poly (assuming that this is not mutated in the context of its enclosing method). Rule (TCALL) adds the 2 constraints "around" $q_{\mathsf{this}} <: q_{\mathsf{ret}}$ to capture call-transmitted dependences:

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

When m is called in a mutable context, i.e., q_{x} is mutable, q_{y} becomes mutable, as expected. Conversely, when m is called in a readonly context, i.e., q_{x} is readonly, $q_{\mathsf{x}} \rhd q_{\mathsf{this}}$ evaluates to readonly, leaving q_{y} unchanged. In terms of our mutability semantics, this behavior captures M- and MA-paths.

3.2 ReM

We now present the ReM type system, which builds upon ReIm.

3.2.1 ReM Qualifiers

The ReM type system adds to the set of ReIm qualifiers, and changes the meaning of some of the ReIm qualifiers. There are 5 qualifiers in ReM: ReIm's mutable, readonly and poly, and two new, maybe and polymaybe. Again, we interpret the qualifiers in terms of the mutability semantics defined in Sect. 2.

- mutable: A mutable reference x is now definitely mutable. It denotes that there is an $(M|C)$-path from x to an update.
- readonly: A readonly reference x has the same meaning as in ReIm, i.e., there is neither an exact nor an approximate path to an update.
- maybe: A maybe reference denotes that there is a $(M|C)A$-path to an update, but there is no R-path to an update.
- poly: A poly reference now denotes that there is an R-path to an update, but there is no $(M|C)A$-path.
- polymaybe: A polymaybe reference denotes that there is an R-path to update, *and* a $(M|C)A$-path to update.

The subtyping hierarchy is as follows:

$$\mathsf{mutable} \;<:\; \mathsf{polymaybe} \;<:\; \begin{array}{c}\mathsf{maybe}\\ \\ \mathsf{poly}\end{array} \;<:\; \mathsf{readonly}$$

3.2.2 Typing Rules

ReM rules extend ReIm. There are two extensions: (1) viewpoint adaptation must account for new qualifiers maybe and polymaybe, and (2) rule (TWRITE) must account for approximate paths.

Viewpoint adaptation rules from Sect. 3.1 remain in effect. We add two new rules:

$$\begin{array}{rcl} \_ \rhd \mathsf{maybe} & = & \mathsf{maybe} \\ q \rhd \mathsf{polymaybe} & = & (q \rhd \mathsf{poly}) \wedge \mathsf{maybe} \end{array}$$

Notation $\wedge$ stands for the standard meet operation: the result of $q_1 \wedge q_2$ is the greatest lower bound of q_1 and q_2 in the lattice of ReM types above.

Since field and return types are restricted to $\{\mathsf{readonly}, \mathsf{poly}\}$, adaptation of maybe or polymaybe happens *only when adapting parameters at method calls.*

Recall that a maybe parameter p denotes a $(M|C)A$-path from p. Thus, call $\mathsf{x} = \mathsf{y.m(z)}$ creates an $(M|C)A$ path from z (a CA-path to be precise). Rule (TCALL) requires

$$q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd q_{\mathsf{p}} \quad \equiv \quad q_{\mathsf{z}} <: \mathsf{maybe}$$

which accounts for the $(M|C)A$-path from z.

Now recall that a polymaybe parameter p denotes an R-path to update, and an $(M|C)A$-path to a (possibly different) update. These paths entail paths from z: one through ret, depending on the left-hand-side of the call assignment, and an $(M|C)A$ path. Rule (TCALL) applies viewpoint adaptation of q_{p}, essentially recording the more conservative choice at the caller. Consider (TCALL) constraint

$$q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd q_{\mathsf{p}}.$$

Suppose that x is readonly, and there is no path from the left-hand-side of the call assignment x to update (that is, the R-path from p is due to a different call site). Then there is no new path from z to update through $\mathsf{p} \rightsquigarrow \mathsf{ret} \xrightarrow{)_i} \mathsf{x}$. However, there is an $(M|C)A$-path from z to update. Viewpoint adaptation accounts for it:

$$q_{\mathsf{z}} <: (\mathsf{readonly} \rhd \mathsf{poly}) \;\wedge\; \mathsf{maybe} \quad \equiv \quad q_{\mathsf{z}} <: \mathsf{maybe}$$

Conversely, suppose x is mutable, and there is an $M|C$-path from x to update. Then the R-path leads to an $M|C$-path from z to update. Viewpoint adaptation accounts for this:

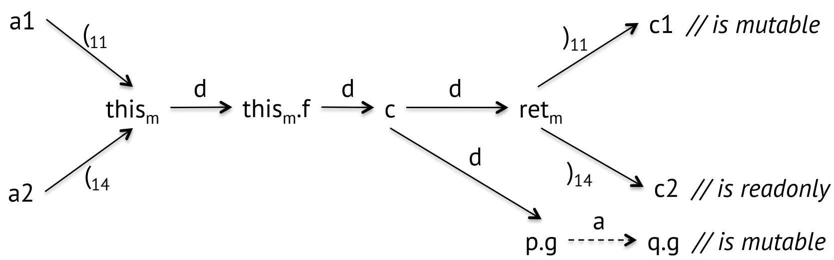
$$q_{\mathsf{z}} <: (\mathsf{mutable} \rhd \mathsf{poly}) \;\wedge\; \mathsf{maybe} \quad \equiv \quad q_{\mathsf{z}} <: \mathsf{mutable}$$

Consider the more detailed example:

```
class A {
 ...
 C m(A this, B p) {
 C c = this.f;
 p.g = c;
 return c;
 }

```

```
9   public static void main(String[] args) {
10  ...
11  C c1 = a1.m(b);
12  c1.setField();
13  ...
14  C c2 = a2.m(b);
15  int i = c2.getField();
16  ....
17  }
18  }
```



The R-path from $\mathsf{this_m}$ to c1, entails $q_{\mathsf{this}} <:$ poly, while the $(M|C)A$-path through p.g entails $q_{\mathsf{this_m}} <:$ maybe. Thus, $q_{\mathsf{this_m}} =$ polymaybe. At call 11 we have

$$q_{\mathsf{a1}} <: q_{\mathsf{c1}} \rhd q_{\mathsf{this_m}} \quad \equiv \quad q_{\mathsf{a1}} <: q_{\mathsf{c1}} \rhd \mathsf{polymaybe}.$$

Since q_{c1} is mutable, polymaybe adapts to mutable, setting q_{a1} to mutable. On the other hand, at call 14 we have

$$q_{\mathsf{a2}} <: q_{\mathsf{c2}} \rhd q_{\mathsf{this_m}} \quad \equiv \quad q_{\mathsf{a2}} <: q_{\mathsf{c2}} \rhd \mathsf{polymaybe} \quad \equiv \quad q_{\mathsf{a2}} <: \mathsf{maybe}.$$

Thus, since q_{c2} is readonly, the meet is maybe, and q_{a2} is precisely maybe.

We now change rule (TWRITE) to account for approximate paths:

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

q_{x} remains mutable to account for the direct update on x. However, instead of adapting by mutable context as in ReIm, we adapt by maybe. This reflects the approximate path, which ReIm conservatively made mutable. If q_{f} is readonly, then the maybe-mutability of x does not affect y. If q_{f} is poly, that reflects an update to some f, and $q_{\mathsf{y}} <: \mathsf{maybe} \rhd \mathsf{poly}$ propagates the "maybe" update to y.

3.3 Type Inference

Type inference for both ReIm and ReM proceeds as outlined in [16, 18] and earlier in [19, 33]. We present novel treatment in terms of dataflow frameworks, and include necessary extensions for ReM.

The inference operates on mappings from keys to values S. The keys in the mapping are (1) local variables and parameters, (2) fields, and (3) method returns. The values in the mapping are *sets* of type qualifiers. For instance, $S(\mathsf{x}) = \{\mathsf{poly}, \mathsf{mutable}\}$ in ReIm means the type of reference x can be poly or mutable.

S is initialized as follows. $S(\mathsf{ret}) = S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{poly}\}$ for all return values ret and fields f. The rest of the variables are initialized to the universal set of qualifiers U, which is $\{\mathsf{readonly}, \mathsf{poly}, \mathsf{mutable}\}$ in ReIm, and $\{\mathsf{readonly}, \mathsf{maybe}, \mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\}$ in ReM. For the rest of this paper we use U to refer to either ReIm or ReM. We denote the initial mapping by S_0.

The inference iterates over all statements s in the program and removes qualifiers inconsistent with the typing rule for s from S. More precisely, let s consist of variables $v_1, v_2, ...v_k$ and let s entail transfer function $c(s)$. Applying $c(s)$ removes each q_1 from $S(v_1)$ when there are no qualifiers $q_2 \in S(v_2)$, ... $q_k \in S(v_k)$, such that $q_1, q_2, ...q_k$ make s type check; then it removes all q_2 from $S(v_2)$, etc. For example, consider s: $\mathsf{y} = \mathsf{x.f}$ and corresponding rule (TREAD) triggering constraint $q_\mathsf{x} \rhd q_\mathsf{f} <: q_\mathsf{y}$. Let $S(\mathsf{x}), S(\mathsf{f})$ and $S(\mathsf{y})$ be as follows:

$S(\mathsf{x})$	$S(\mathsf{f})$	$S(\mathsf{y})$
$\{\mathsf{maybe}, \mathsf{polymaybe}, \mathsf{mutable}\}$	$\{\mathsf{readonly}, \mathsf{poly}\}$	$\{\mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\}$

$c(s)$ removes maybe from $S(\mathsf{x})$ because there does not exist $q_\mathsf{f} \in S(\mathsf{f})$ and $q_\mathsf{y} \in S(\mathsf{y})$ that satisfy $\mathsf{maybe} \rhd q_\mathsf{f} <: q_\mathsf{y}$. Similarly, it removes readonly from $S(\mathsf{f})$. After application of $c(s)$:

$S'(\mathsf{x})$	$S'(\mathsf{f})$	$S'(\mathsf{y})$
$\{\mathsf{polymaybe}, \mathsf{mutable}\}$	$\{\mathsf{poly}\}$	$\{\mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\}$

One can easily prove that, given S_0 as shown, $c(s)$ only need look at the left-hand-side of the constraint, i.e., the right-hand-side always remains unchanged.

The inference analysis iterates over the statements in the program and removes qualifiers from the sets until it reaches a fixpoint. The problem fits into the standard monotone dataflow framework [1, 20]. The lattice L_v for variables v is

$$\{\mathsf{mutable}\} \;>\; \{\mathsf{polymaybe}, \mathsf{mutable}\} \;>\; \begin{array}{c} \{\mathsf{maybe}, \mathsf{polymaybe}, \mathsf{mutable}\} \\ \\ \{\mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\} \end{array} \;>\; U$$

and the dataflow lattice L is the product lattice of all L_v lattices, which is standard. Initializing all variables to U corresponds to initializing with the 0 of the lattice. The function space is $L \to L$ and is monotone. This is a theorem that one can easily show by case-by-case analysis of each $c(s)$. Therefore, the result of fixpoint iteration is the maximal fixpoint solution. Call this solution S_{Fix}. Yet the fixpoint solution 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 *maximal* element of $S_{Fix}(\mathsf{x})$ according to the following ranking, which mirrors the subtyping lattice:

$$\mathsf{readonly} \;>\; \begin{array}{c} \mathsf{maybe} \\ \\ \mathsf{poly} \end{array} \;>\; \mathsf{polymaybe} \;>\; \mathsf{mutable}$$

Importantly, the maximal element exists because each $S_{Fix}(\mathsf{x})$ is an element of L_v. We denote this typing by $max(S_{Fix})$, and call it the *maximal typing*.

The following propositions state that (1) the maximal typing type checks, and (2) the maximal typing is the "best typing". (Note that setting all references to mutable also type checks, but makes up a useless typing.)

▶ **Proposition 1.** *ReIm's $max(S_{Fix})$ and ReM's $max(S_{Fix})$ always type check.*

Proof Sketch. The proof for ReIm is given in [18]. The proof for ReM proceeds by case-by-case analysis. The most difficult case arises at $q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd q_{\mathsf{p}}$. Let $S_{Fix}(\mathsf{p}) = \{\mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\}$ and $S_{Fix}(\mathsf{z}) = \{\mathsf{maybe}, \mathsf{polymaybe}, \mathsf{mutable}\}$. $S_{Fix}(\mathsf{x})$ must be either $\{\mathsf{maybe}, \mathsf{polymaybe}, \mathsf{mutable}\}$ or $\{\mathsf{readonly}, \mathsf{maybe}, \mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\}$. If it were any other set, then maybe would have been removed from $S(\mathsf{z})$ during fixpoint iteration. Combinations $q_{\mathsf{z}} = \mathsf{maybe}$, $q_{\mathsf{x}} = \mathsf{maybe}$, $q_{\mathsf{p}} = \mathsf{poly}$, and $q_{\mathsf{z}} = \mathsf{maybe}$, $q_{\mathsf{x}} = \mathsf{readonly}$, $q_{\mathsf{p}} = \mathsf{poly}$, maximal typings under the two cases, both typecheck.

A more general statement is true. For every S that satisfies the equations of the dataflow frameowork, typing $max(S)$ type checks. ◀

Previous work in [16] formalized the notion of "best typing" for ownership type systems, specifically Ownership types [5] and Universe Types [8], by using a heuristic ranking over typings. This formalization applies to ReIm/ReM, as well as other ownership-like type systems, e.g., AJ [35] and EnerJ [28]. Below we extend the treatment of [16] to ReM.

We say that T is a *valid typing* if T type checks. Objective function o ranks valid typings. o takes a valid typing T and returns a tuple of numbers. For ReIm, o is as follows:

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

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 poly references than T_2. The preference ranking over typings is based on ranking over qualifiers: naturally, we prefer readonly over poly and mutable, and poly over mutable.

ReM's objective function is the following:

$$o_{ReM}(T) = \quad (|T^{-1}(\mathsf{readonly})|, |T^{-1}(\mathsf{poly})| + |T^{-1}(\mathsf{maybe})|, |T^{-1}(\mathsf{polymaybe})|, |T^{-1}(\mathsf{mutable})|)$$

Again the tuples are ordered based on the natural ranking over qualifiers: readonly is the most preferred, followed by poly and maybe, which are equally preferred, and so on. The following proposition establishes that the maximal typing is the best typing.

▶ **Proposition 2.** *Let o be the objective function over valid typings (either o_{ReIm} over ReIm, or o_{ReM} over ReM). $o(max(S_{Fix})) > o(T)$ holds for every valid typing $T \neq max(S_{Fix})$.*

Proof Sketch. The fact that the maximal typing is the "best typing", follows from the properties of monotone dataflow frameworks. Let S be another solution of the dataflow framework. (It is easy to see that if T is valid typing, it must be contained into a solution of the dataflow framework.) Since $S > S_{Fix}$ by virtue of S_{Fix} being the maximal fixpoint solution, for every variable x $S(\mathsf{x}) \geq S_{Fix}(\mathsf{x})$, and there exist variables y such that $S(\mathsf{y}) > S_{Fix}(\mathsf{y})$. Thus, $o_{ReM}(max(S_{Fix})) > o_{ReM}(max(S))$. ◀

4 Equivalence

This section formally links the mutability semantics and the maximal typing. Specifically, we establish equivalence between CFL-reachability, as outlined in Sect. 2, and the maximal typing as outlined in Sect. 3.3 and previous work [16, 18, 19, 33].

Fig. 5 states the algorithms for CFL-reachability and type inference explicitly. Cfl initializes graph G to $\emptyset$, then iterates over the program statements adding paths to updates, until no more paths can be added. Types initializes S to the 0 of the lattice, then iterates over the statements, removing qualifiers from S until no more qualifiers can be removed. To

procedure CFL
  $G = \emptyset$
  Add $u \overset{M}{\rightsquigarrow} u$ to G for all updates u
  while G changes **do**
    for each s in Program **do**
      EDGE($e(s)$)
    end for
  end while
end procedure

procedure EDGE($n \xrightarrow{\mathtt{t}} n'$)
  for each $n' \overset{N}{\rightsquigarrow} u \in G$ **do**
    Add $n \overset{\mathtt{t} \oplus N}{\rightsquigarrow} u$ to G
  end for
end procedure

procedure TYPES
  $S(n) = U$
  $S(n) = \{\mathsf{mutable}\}$ for all updates n
  while S changes **do**
    for each s in Program **do**
      CONSTRAINT($c(s)$)
    end for
  end while
end procedure

procedure CONSTRAINT($l <: r$)
  Remove each q_l from $S(l)$
  if $\nexists\, q_r \in S(r)$ s.t. $q_l <: q_r$
end procedure

Figure 5 Algorithm CFL initializes G, then iterates over program statements s adding edges as specified in Sect. 2.1. Algorithm TYPES initializes S, then iterates over program statements s removing qualifies from S as specified in Sect. 3.3. The algorithms elide details to highlight the "parallel" structure of the two systems.

emphasize the parallel structure, Fig. 5 simplifies the presentation. Most notably, recall that according to Sect. 2 field read $\mathsf{y}' = \mathsf{x}'.\mathsf{f}$ accounts for two edges:

$$\mathsf{y}' = \mathsf{x}'.\mathsf{f} \quad \Rightarrow \quad \mathsf{x}' \xrightarrow{\mathtt{d}} \mathsf{x}'.\mathsf{f} \xrightarrow{\mathtt{d}} \mathsf{y}'$$

Even though Fig. 5 shows a single invocation of EDGE, in fact CFL processes two edges, first $\mathsf{x}'.\mathsf{f} \xrightarrow{\mathtt{d}} \mathsf{y}'$, followed by $\mathsf{x}' \xrightarrow{\mathtt{d}} \mathsf{x}'.\mathsf{f}$. Similarly, CFL processes multiple edges at field writes $\mathsf{x}.\mathsf{f} = \mathsf{y}$: for each field read $\mathsf{y}' = \mathsf{x}'.\mathsf{f}$, such that $\mathsf{x}'.\mathsf{f} \in G$, it processes $\mathsf{x}.\mathsf{f} \overset{\mathtt{a}}{\dashrightarrow} \mathsf{x}'.\mathsf{f}$, followed by $\mathsf{y} \xrightarrow{\mathtt{d}} \mathsf{x}.\mathsf{f}$.

Another detail elided from Fig. 5 is the meaning of $\mathtt{t}$, N and concatenation operator $\oplus$. $\mathtt{t}$ ranges over the terminals: $(_i,)_i, \mathtt{d}$, and $\mathtt{a}$. N ranges over the kinds of paths: $M|C$, $(M|C)A$, and R. Concatenation $\mathtt{t} \oplus N$ applies the grammar rules, and in all but one case, is straightforward:

$$\begin{array}{lcl} (_i \oplus M|C & = & M|C \\ (_i \oplus (M|C)A & = & (M|C)A \\ )_i \oplus \_ & = & R \\ \mathtt{d} \oplus N & = & N \\ \mathtt{a} \oplus \_ & = & (M|C)A \end{array}$$

$(_i \oplus R$ is the difficult case, because it applies rule $M ::= (_i\ M\ )_i$. EDGE applies concatenation $(_i \oplus R$ when processing call edge $\mathsf{z} \xrightarrow{(_i} \mathsf{p}$ and $\mathsf{p} \overset{R}{\rightsquigarrow} u \in G$. (Here p is the parameter of the compile-time target method of the call.) $\mathsf{p} \overset{R}{\rightsquigarrow} u \in G$ entails

$$\mathsf{p} \overset{M}{\rightsquigarrow} \mathsf{ret} \xrightarrow{)_j} \mathsf{x} \overset{N}{\rightsquigarrow} u$$

where ret is the return value of the target method, and x is some left-hand-side of a call assignment. Note that $\mathsf{p} \overset{M}{\rightsquigarrow} \mathsf{ret} \xrightarrow{)_j} \mathsf{x}$ are not explicitly in G, but $\mathsf{x} \overset{N}{\rightsquigarrow} u$ is in G. There are two cases. If there is no edge $)_j$ such that $j = i$, then $(_i \oplus R$ adds no new paths; the R-path from p is due to a different call site. Otherwise, that is, when $j = i$, concatenation adds $\mathsf{z} \overset{N}{\rightsquigarrow} u$ to G.

Let us illustrate CFL and EDGE, and TYPES and CONSTRAINT in parallel. Consider

$\mathsf{y} = \mathsf{x}; \mathsf{z} = \mathsf{y}; \mathsf{z.f} = \mathsf{w}$

Calling EDGE on $\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{z}$, which corresponds to statement $\mathsf{z} = \mathsf{y}$, leads to path $\mathsf{y} \overset{M}{\rightsquigarrow} \mathsf{z}$ in G. Subsequently calling EDGE on edge $\mathsf{x} \xrightarrow{\mathsf{d}} \mathsf{y}$ leads to concatenation of d and M and path $\mathsf{x} \overset{M}{\rightsquigarrow} \mathsf{z}$. There are two M-paths, x to z and y to z, as expected. Analogously, calling CONSTRAINT on $q_{\mathsf{y}} <: q_{\mathsf{z}}$, which corresponds to statement $\mathsf{z} = \mathsf{y}$, removes all qualifiers but $\mathsf{mutable}$ from $S(\mathsf{y})$. Subsequently calling CONSTRAINT on $q_{\mathsf{x}} <: q_{\mathsf{y}}$ removes all qualifiers but $\mathsf{mutable}$ from $S(\mathsf{x})$. $S(\mathsf{x}) = \{\mathsf{mutable}\}$, and $S(\mathsf{y}) = \{\mathsf{mutable}\}$ mirror the two M-paths that CFL finds.

The most interesting case arises (as it has been the case throughout the paper), when adding a call edge. Consider code

$\mathsf{y} = \mathsf{id}(\mathsf{x}); \mathsf{y} = \mathsf{z}; \mathsf{z.f} = \mathsf{w}$

and assume $\mathsf{y} \overset{M}{\rightsquigarrow} \mathsf{z}$ and $\mathsf{p} \overset{R}{\rightsquigarrow} \mathsf{z}$ are already in G. Concatenation breaks the R-path $\mathsf{p} \overset{R}{\rightsquigarrow} \mathsf{z}$ into $\mathsf{p} \overset{M}{\rightsquigarrow} \mathsf{ret} \xrightarrow{)_i} \mathsf{y} \overset{M}{\rightsquigarrow} \mathsf{z}$. Since $(_i$ and $)_i$ match, it adds path $\mathsf{x} \overset{M}{\rightsquigarrow} \mathsf{z}$. Analogously, for TYPES assume $S(\mathsf{p}) = \{\mathsf{poly}, \mathsf{polymaybe}, \mathsf{mutable}\}$ and $S(\mathsf{y}) = \{\mathsf{mutable}\}$. Calling CONSTRAINT on $q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{p}}$ removes all qualifiers but $\mathsf{mutable}$ from $S(\mathsf{x})$, which directly corresponds to the M-path from x to z that CFL finds.

To formally establish equivalence we use the bisumulation methodology for proving equivalence between two systems A and B [36, 6]. The methodology requires that we establish a relation that relates states in A to those in B. In our case, A is constructed by CFL-reachability inference (Algorithm CFL), and B is constructed by type inference (Algorithm TYPES). Our approach defines an explicit equivalence relation between the states in A, captured by G, and those in B, captured by S. Intuitively, assume algorithms CFL and TYPES run in "parallel". To show equivalence we must show that processing s in each system maintains equivalence.

We state two definitions that form the basis of equivalence. Informally, Def. 3 states that for every path from n to update u in G, n is correspondingly typed in S. In Def. 3 n stands for either a variable node x, or field access node $\mathsf{x.f}$.

▶ **Definition 3.** (Soundness) $G \Rightarrow S$ if and only if

1. $n \overset{M|C}{\rightsquigarrow} u \in G \quad \Rightarrow \quad max(S(n)) <: \mathsf{mutable}$
2. $n \overset{R}{\rightsquigarrow} u \in G \quad \Rightarrow \quad max(S(n)) <: \mathsf{poly}$
3. $n \overset{(M|C)A}{\rightsquigarrow} u \in G \quad \Rightarrow \quad max(S(n)) <: \mathsf{maybe}$

Def. 4 states that n's maximal type in S implies a corresponding path in G. For example, maximal typing $\mathsf{polymaybe}$ must imply that there are both an R-path and a $(M|C)A$-path in G, but there is no $M|C$-path.

▶ **Definition 4.** (Precision) $S \Rightarrow G$ if and only if

1. $max(S(\mathsf{x})) = \mathsf{mutable} \quad \Rightarrow \quad \exists\, \mathsf{x} \overset{M|C}{\rightsquigarrow} u \in G$
2. $max(S(\mathsf{x})) = \mathsf{polymaybe} \quad \Rightarrow \quad \exists\, \mathsf{x} \overset{R}{\rightsquigarrow} u \in G \land \exists\, \mathsf{x} \overset{(M|C)A}{\rightsquigarrow} u \in G \land \neg\exists\, \mathsf{x} \overset{M|C}{\rightsquigarrow} u \in G$
3. $max(S(\mathsf{x})) = \mathsf{poly} \quad \Rightarrow \quad \exists\, \mathsf{x} \overset{R}{\rightsquigarrow} u \in G \land \neg\exists\, \mathsf{x} \overset{(M|C)A}{\rightsquigarrow} u \in G \land \neg\exists\, \mathsf{x} \overset{M|C}{\rightsquigarrow} u \in G$
4. $max(S(\mathsf{x})) = \mathsf{maybe} \quad \Rightarrow \quad \neg\exists\, \mathsf{x} \overset{R}{\rightsquigarrow} u \in G \land \exists\, \mathsf{x} \overset{(M|C)A}{\rightsquigarrow} u \in G \land \neg\exists\, \mathsf{x} \overset{M|C}{\rightsquigarrow} u \in G$
5. $max(S(\mathsf{x})) = \mathsf{readonly} \quad \Rightarrow$ no path from x in G
6. $max(S(\mathsf{x.f})) = \mathsf{readonly} \quad \Rightarrow$ no path from $\mathsf{x.f}$ in G

▶ **Definition 5.** (Equivalence) $G \simeq S$ if and only if $G \Rightarrow S$ and $S \Rightarrow G$.

Let the following Hoare triple denote parallel execution of Edge and Constraint on statement s:

$$\{G, S\}\ \textsc{Edge}(e(s))\ ||\ \textsc{Constraint}(c(s))\ \ \{G', S'\}$$

Our key result is the following theorem:

▶ **Theorem 6.** *If* $G \simeq S$ *and* $\{G, S\}$ $\textsc{Edge}(e(s))$ $||$ $\textsc{Constraint}(c(s))$ $\{G', S'\}$ *then* $G' \simeq S'$.

Proof Sketch. As expected, the proof is by induction on the number of applications of

$$\textsc{Edge}(e(s))\ \ ||\ \ \textsc{Constraint}(c(s))$$

Clearly, the statement holds after initialization, lines 2-3 in Cfl and lines 2-3 in Types. The inductive step requires case-by-case analysis of each s.

To prove correctness, we must show that given $G \Rightarrow S$, after the execution of $\textsc{Edge}(e(s))$ and $\textsc{Constraint}(c(s))$, $G' \Rightarrow S'$ still holds. We outline the most difficult case, $\textsc{Edge}(\mathsf{z} \xrightarrow{(_i} \mathsf{p})$ $||$ $\textsc{Constraint}(q_\mathsf{z} <: q_\mathsf{x} \rhd q_\mathsf{p})$ (method call naturally brakes into three steps). Consider $\mathsf{z} \xrightarrow{(_i} \mathsf{p} \oplus \mathsf{p} \overset{R}{\rightsquigarrow} u$. Let x be the left-hand-side at call assignment i. If there does not exist a path $\mathsf{x} \overset{N}{\rightsquigarrow} u \in G$, then no new paths are added to G' and $G' \Rightarrow S'$ holds. If there exists $\mathsf{x} \overset{N}{\rightsquigarrow} u \in G$, then a new path $\mathsf{z} \overset{N}{\rightsquigarrow} u$ is added to G'. We must show that $S'(\mathsf{z})$ reflects N according to Def. 3 (e.g., if N is an $M|C$-path, then z is $\mathsf{mutable}$). By the inductive hypothesis $\mathsf{p} \overset{R}{\rightsquigarrow} u \in G \Rightarrow \mathit{max}(S(\mathsf{p})) <: \mathsf{poly}$. Similarly, the N-path entails appropriate $S(\mathsf{x})$: if N is an $M|C$ path, then $S(\mathsf{x}) = \{\mathsf{mutable}\}$, if N is an R path then $\mathit{max}(S(\mathsf{x})) <: \mathsf{poly}$, and if N is a $(M|C)A$-path, then $\mathit{max}(S(\mathsf{x})) <: \mathsf{maybe}$. Constraint $q_\mathsf{z} <: q_\mathsf{x} \rhd q_\mathsf{p}$ removes qualifiers from $S(\mathsf{z})$. For example, if N is $M|C$, then $S(\mathsf{x}) = \{\mathsf{mutable}\}$, and it is easy to see that $\mathit{max}(S(\mathsf{x})) \rhd \mathit{max}(S(\mathsf{p})) <: \mathsf{mutable}$. Thus $\mathit{max}(S'(\mathsf{z})) <: \mathsf{mutable}$, as needed. We enumerate all cases in Sect. A (Proofs).

In the other direction, we must show that if $S \Rightarrow G$ holds, after the execution of Edge and Constraint on s, $S' \Rightarrow G'$ still holds. Consider the analogous case, $\textsc{Edge}(\mathsf{z} \xrightarrow{(_i} \mathsf{p})$ $||$ $\textsc{Constraint}(q_\mathsf{z} <: q_\mathsf{x} \rhd q_\mathsf{p})$, and the following values of $S(\mathsf{x}), S(\mathsf{p})$ and $S(\mathsf{z})$ (only the maximal element shown):

$S(\mathsf{x})$	$S(\mathsf{p})$	$S(\mathsf{z})$
{ readonly, ...	{ maybe, ...	{ readonly, ...

Constraint $q_\mathsf{z} <: q_\mathsf{x} \rhd q_\mathsf{p}$ "lowers" $S(\mathsf{z})$ into $S'(\mathsf{z}) = \{\mathsf{maybe}, ...\}$. By the inductive hypothesis, $S(\mathsf{x})$ and $S(\mathsf{p})$ entail that there are no paths from x in G, no paths from z in G, and only a $(M|C)A$-path from p. Therefore, $\textsc{Edge}(\mathsf{z} \xrightarrow{(_i} \mathsf{p})$ adds an $(M|C)A$-path from z in G', and no other kind of path. Thus, $\mathit{max}(S'(\mathsf{z})) = \mathsf{maybe} \Rightarrow \mathsf{z} \overset{(M|C)A}{\rightsquigarrow} u$, as expected. We enumerate all cases in Sect. A (Proofs). ◀

For clarity, we omitted method overriding. It is handled in both the mutability semantics and type inference, and equivalence still holds. Concretely, if m' overrides m we add

$$q_{\mathsf{this}_\mathsf{m}} \xrightarrow{\mathsf{d}} q_{\mathsf{this}_{\mathsf{m}'}} \quad q_{\mathsf{p}_\mathsf{m}} \xrightarrow{\mathsf{d}} q_{\mathsf{p}_{\mathsf{m}'}} \quad q_{\mathsf{ret}_{\mathsf{m}'}} \xrightarrow{\mathsf{d}} q_{\mathsf{ret}_\mathsf{m}}$$

to G. Analogously, we require

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

which entails

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

which leads to the standard function subtyping constraints:

$$q_{\mathsf{this}_m} <: q_{\mathsf{this}_{m'}} \quad q_{\mathsf{p}_m} <: q_{\mathsf{p}_{m'}} \quad q_{\mathsf{ret}_{m'}} <: q_{\mathsf{ret}_m}$$

Our implementation handles function subtyping.

5 Empirical Results

We implemented ReM on top of ReIm. (ReIm is publicly available.) Soot is the underlying platform, and Jimple is the underlying intermediate representation. We evaluate ReM on DaCapo, plus the benchmarks used in Javarifier [23] and ReIm [18]. There are 13 whole programs, and 8 libraries:

- **DaCapo** suite DaCapo-2006-10MR.
- **JOlden** is a classical suite of 10 small whole programs (Javarifier and ReIm).
- **ejc-3.2.0** is the Java Compiler for the Eclipse IDE (Javarifier and ReIm).
- **javad** is a Java disassembler program (ReIm).
- **tinySQL-1.1** is a database engine (Javarifier and ReIm).
- **htmlparser-1.4** is an HTML parser library (Javarifier and ReIm).
- **commons-pool-1.2** is an object pooling library (ReIm).
- **jtds-1.0** is a JDBC driver (ReIm).
- **jdbm-1.0** is a transactional engine (ReIm).
- **jdbf-0.0.1** is an object-oriented mapping system (ReIm).
- **java.lang** and **java.util** are the packages from JDK 1.7.0_75.

All benchmarks are analyzed with JDK 1.7. On whole programs, our analysis relies on the standard Class Hierarchy Analysis (CHA)-based reachability in Soot, which pulls in all relevant packages according to CHA. ReIm/ReM analyzes all these packages. All experiments are done on a MacBook Pro 2.8 GHz Intel Core i7 and 16GB of RAM using default VM settings for everything, including maximal heap size.

Tab. 1 presents the results of running ReM inference on the benchmarks. On average, only 6.4% of all references are inferred as maybe or polymaybe. They make up only about 13.6% of all ReIm-mutable references while the remaining 86.4% are *definitely mutable*. To assess the impact of the intermediate representation, Soot's Jimple, which creates a significant number of temporary local variables, we computed statistics on parameter and returns (no local variables). The results show that 5.8% of all references are maybe or polymaybe, and approximately 84% of all ReIm-mutable parameters and returns are definitely mutable. These result is very similar, suggesting that the intermediate representation does not lead to an overestimation of the number of definitely mutable references. (In fact, our investigation suggests that it may lead to an underestimation, as we explain shortly.) Running times do not exceed 90 seconds, with most benchmarks completing in under 60 seconds on the commodity laptop described earlier.

In addition to the benchmarks from Tab. 1 we ran our analysis on Avrora, Batik and Sunflow from DaCapo-9.12-MR1-bach; these are whole-program benchmarks were added to

Table 1 Inference results for ReM. **Annotatable References** includes all variables of reference type, including locals, parameters, returns, and fields. It does not include variables of primitive type. Column **#Readonly** shows the number of references inferred as readonly, **#Poly** shows the number of variables inferred as poly, and **#Maybe/#polymaybe** shows the number of maybe and polymaybe references, respectively. Column **#Mutable** shows the number of mutable references, which are now definitely mutable. In parentheses is the percentage of definitely mutable references over of all potentially mutable ones: **#Mutable/(#Maybe+#Polymaybe+#Mutable)**.

Benchmark	Annotatable References				Time
	#Readonly	#Poly	#Maybe/ #Polymaybe	#Mutable	(sec.)
antlr	15751	1029	1198/2	12552 (91.3%)	64.3
bloat	12567	2299	3880/56	18844 (82.7%)	36.1
chart	31640	4973	7570/139	32063 (80.6%)	79.3
eclipse	16945	4117	2613/36	12807 (82.9%)	39.1
fop	47411	5563	5578/66	35564 (86.3%)	89.0
hsqldb	29907	4282	3673/31	26054 (87.6%)	66.8
luindex	5465	686	750/11	4604 (85.8%)	30.1
lusearch	6296	945	1019/16	5344 (83.8%)	30.2
pmd	37758	4605	5248/96	34096 (86.5%)	78.4
xalan	21254	3337	2853/63	19771 (87.1%)	75.4
jolden	1208	281	206/0	1069 (83.8%)	29.0
javad	796	45	15/0	417 (96.5%)	26.5
ejc	29062	8202	3612/174	24768 (86.7%)	49.3
tinySQL	6115	665	638/7	3585 (84.8%)	28.1
htmlparser	7787	1215	1049/12	4904 (82.2%)	29.2
commons-pool	847	25	222/0	640 (74.2%)	27.1
jdbm	1134	317	322/9	1584 (82.7%)	28.8
jdbf	4017	312	289/2	2379 (89.1%)	58.6
jtds	8628	825	587/25	5738 (90.4%)	32.5
java.lang	3336	350	249/1	4721 (95.0%)	29.5
java.util	3216	376	309/0	4758 (93.9%)	30.9
Average				86.4%	

DaCapo 2006 for the 2009 suite. Our analysis reports that on average 84.5% of ReIm-mutable references are definitely mutable, which is in line with Tab. 1.[2]

The results demonstrate that ReM and ReIm are *precise* and *scalable*. They can be used to power inference for approximate computing (e.g. EnerJ [28] and Rely [4]), taint analysis (e.g., DroidInfer [17]), and method purity [18], as well as other client analyses. Even if one designed a more complex system that handled structure-transmitted dependences more precisely, by employing a powerful alias analysis for example, improvement would be at most 5-6% of all references being promoted from mutable (12-13% of ReIm's mutable references). ReM/ReIm's complexity is $O(N^2)$, which leads to fast running times.

Finally, to better understand the results, we examined all 15 maybe references from javad, and 15 randomly selected maybe references from ejc. We looked to identify definite paths to mutation, or more precisely, we examined $\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f} \xrightarrow{\mathsf{d}} \mathsf{y'}$ and attempted to prove

[2] We omitted Tomcat, H2 and the Treadsoap benchmarks from DaCapo-9.12-MR1-bach as these are complex client-server programs and we were unable to set the analysis in time for the publication deadline. Recent work in this space [14] omits these program as well. Also due to timing, DaCapo 2009 was not included in Artifact Evaluation.

that there exist a runtime object o such that x points to o, x′ points to o, and the value of x.f indeed flows to x′.f. We immediately identified such definite paths in 16 out of 30 cases. The remaining 14 cases exhibited difficult data and control flow, and we could not identify definite paths. A typical case of obvious definite paths was the following. Consider this typical code for initializing an array field f:

```
f = new X[10];
for (int i=0; i<cnt; i++)
  f[i] = new X();
...
```

This code snippet is translated into the following Jimple:

```
r1 = newarray (X)[10];
this.f = r1;
...
r2 = this.f;
x = new X();
r2[i] = x;
```

Mutation of the array is captured by the approximate path, and the maybe typing of r1. This case leads to an overestimation of the number of maybe variables and the following simple optimization reduces the number of maybe/polymaybe references. Specifically, at each field write this.f = r1 we add constraint $q_{\mathsf{r1}} <: q_{\mathsf{m.this.f}}$ where m is the enclosing method and m.this.f is a dummy variable. Similarly, at each field read r2 = this.f we add constraint $q_{\mathsf{m.this.f}} <: q_{\mathsf{r2}}$. Thus, when r2 is mutable or poly/polymaybe, mutable or poly/polymaybe propagates through $q_{\mathsf{m.this.f}}$, and the analysis demotes r1 to mutable or polymaybe. Tab. 2 shows the results of this optimization – on average 87.3% of mutable references are now definitely mutable. As with DaCapo 2009 the optimization was added for the final version and was not part of Artifact Evaluation.

Another source of maybe mutability is containers. E.g., in

```
class Container {
  Data data;
  void set(Container this, Data p) {
  this.data = p;
  }
  Data get(Container this) {
  return this.data;
  }
}
```

parameter p of set is rightfully maybe mutable. p and the data object will be mutable in some clients of Container and readonly in others.

6 Related Work

The most closely related work is Huang et al.'s ReIm and ReImInfer [18]. Our work builds upon ReIm and ReImInfer but extends them in two directions. First, we build a theoretical framework that interprets ReIm and ReM in terms of CFL-reachability, and we prove them correct within this simple framework. To the best of our knowledge, ReIm has not been proven correct even though it has been used to power client analyses [17, 32]. Second, we propose ReM and definite mutability, which extends the expressive power of ReIm, and also,

Table 2 Results after optimization. 87.3% of ReIm-mutable references are definitely mutable.

Benchmark	Annotatable References #Readonly	#Poly	#Maybe/ #PolyOrMaybe	#Mutable	Time (sec.)
antlr	15751	1029	1093/6	12653 (92.0%)	128.2
bloat	12567	2299	3683/68	19029 (83.5%)	38.1
chart	31640	4973	7329/154	32289 (81.2%)	86.5
eclipse	16945	4118	2473/46	12937 (83.7%)	38.3
fop	47411	5563	5235/93	35880 (87.1%)	93.2
hsqldb	29907	4282	3303/37	26418 (88.8%)	86.8
luindex	5465	686	677/13	4675(87.1%)	31.6
lusearch	6296	945	932/17	5430 (85.1%)	35.9
pmd	37758	4605	4989/106	34345 (87.1%)	83.3
xalan	21254	3337	2634/71	19982 (88.1%)	79.8
jolden	1208	281	187/9	1080 (84.6%)	29.0
javad	796	45	3/0	429 (99.3%)	28.4
ejc	29062	8202	3146/194	25214 (88.3%)	51.7
tinySQL	6115	665	608/7	3615 (85.5%)	118.7
htmlparser	7787	1215	1023/12	4930(82.6%)	114.1
commons-pool	847	25	222/0	642 (74.5%)	25.6
jdbm	1134	317	293/9	1613(84.2%)	28.4
jdbf	4017	312	281/2	2387 (89.4%)	116.3
jtds	8628	825	552/25	5773(90.1%)	36.7
java.lang	3336	350	217/1	4753(95.6%)	34.1
java.util	3216	376	276/0	4791 (94.6%)	37.1
Average				87.3%	

establishes a bound on (im)precision. Our empirical results show that ReIm/ReM are highly precise and highly scalable.

Reference immutability has been an active area of research for many years. Tschantz et al. propose Javari [34] and Javari's inference tool Javarifier [23]. Javari is more expressive and more complex than ReIm, but the inferable features are essentially the same. (Huang et al. report that Javarifier and ReImInfer produce essentially the same result.) Zibin et al's IGJ [39] and OIGJ [40] are type systems that support reference immutability and object immutability. Haack and Poll [15] propose a type system for object immutability as well. Gordon et al. [13] propose a reference immutability system for safe parallelism. Potanin et al. [22] survey work on reference and object immutability, and method purity. As it is standard, these systems support definite immutability (like ReIm). They do not attempt to estimate precision (imprecision), or connect reference immutability and CFL-reachability.

Artzi et al. [2] propose a hybrid static and dynamic analysis for inference of parameter reference immutability. In contrast, our work focuses on static analysis.

Salcianu and Rinard's JPPA [31] and Pearce's JPure [21] infer method purity for Java. ReIm/ReM is more general, in the sense that it enables reasoning about method purity, as well as other client analyses (e.g., EnerJ [28] and DroidInfer [17]). The fact that ReIm/ReM is precise, suggests that client analyses would be precise as well.

CFL-reachability is a standard program analysis framework [25]. Rehof and Fahndrich [24] connect type-based flow analysis and CFL-reachability. This is similar to our interpretation of type-based reference immutability in terms of CFL-reachability. However, Rehof and Fahndrich do not discuss mutable references and it is unclear how they handle such references or structure-transmitted dependences. Fahndrich et al. [11] apply the theory of [24] to build a context-sensitive Steensgard-style points-to analysis for C, thus using equality constraints instead of subtyping constraints. (Equality constraints is the standard approach to the handling of mutable references [29, 28, 12], as we mention earlier.) Our work focuses specifically on reference immutability and reasoning about its precision. The result that ReIm/ReM is precise, indicates that they can be incorporated into flow analyses [29, 28, 12, 17].

Sridharan and Bodik [30] present refinement-based points-to analysis for Java using CFL-reachability. Xu et al. [37] improve the scalability of CFL-reachability-based points-to analysis. These works focus on points-to analysis and require heap abstraction. Therefore, they inherit known issues with reflection. Type-based reference immutability and the parallel CFL-reachability analysis avoid heap abstraction and thus, they completely avoid issues due to reflective object creation (x = Class.forName("className").newInstance()), for free. We still face issues with reflective method invocation (getMethod). However, reflective object creation is by far most common, and has been studied extensively in the points-to analysis community. Recent work by Zhang and Su [38] propose new approximation algorithms based on CFL-reachability that can handle both structure-transmitted and call transmitted dependences precisely. Our work focuses on type-based reference immutability, for which handling of structure-transmitted dependences approximately appears sufficient.

7 Conclusion

We presented ReM, a novel reference immutability type system. ReM separated potentially mutable references into definitely mutable, and maybe mutable, i.e., references that may be mutable due to inherent approximation. In addition, we proposed a CFL-reachability system for reference immutability, thus building a novel framework for reasoning about correctness of reference immutability type systems. We implemented ReM and showed that about 86.5% of all potentially mutable references were definitely mutable.

References

1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

2 Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 104–113, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321649.

3 Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 132–145, New York, NY, USA, 1997. ACM. doi:10.1145/263699.263714.

4 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 33–52, 2013.

5 David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM. doi:10.1145/286936.286947.

6 Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, pages 11–23, 1989. `doi:10.1007/3-540-52148-8_2`.

7 Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Formal methods for components and objects. In Frank S. Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, chapter Universe Types for Topology and Encapsulation, pages 72–112. Springer-Verlag, Berlin, Heidelberg, 2008. `doi:10.1007/978-3-540-92188-2_4`.

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

9 Werner Dietl and Peter Müller. Runtime universe type inference. In *IWACO*, pages 72–80, 2007. URL: `http://sct.ethz.ch/projects/student_docs/Frank_Lyner/Frank_Lyner_MA_paper.pdf`.

10 Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):1–48, apr 2012.

11 Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM. `doi:10.1145/349299.349332`.

12 Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. Efficiently refactoring java applications to use generic libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 71–96, Berlin, Heidelberg, 2005. Springer-Verlag. `doi:10.1007/11531142_4`.

13 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 21–40, New York, NY, USA, 2012. ACM. `doi:10.1145/2384616.2384619`.

14 Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis. *PACMPL*, 1(OOPSLA):102:1–102:28, 2017. `doi:10.1145/3133926`.

15 Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 520–545, Berlin, Heidelberg, 2009. Springer-Verlag. `doi:10.1007/978-3-642-03013-0_24`.

16 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 181–206, Berlin, Heidelberg, 2012. Springer-Verlag. `doi:10.1007/978-3-642-31057-7_9`.

17 Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 106–117, New York, NY, USA, 2015. ACM. `doi:10.1145/2771783.2771803`.

18 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Lan-*

guages and Applications, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384680.

19 Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing java classes. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 437–446, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/ICSE.2007.70.

20 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer-Verlag New York, Inc., 1999.

21 David J. Pearce. Jpure:: A modular purity system for java. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 104–123, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1987237.1987247.

22 Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 233–269. Springer-Verlag, apr 2013.

23 Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 616–641, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_26.

24 Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: From polymorphic subtyping to cfl-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 54–66, New York, NY, USA, 2001. ACM. doi:10.1145/360204.360208.

25 Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162—-186, 2000.

26 Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998. doi:10.1016/S0950-5849(98)00093-7.

27 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995. doi:10.1145/199448.199462.

28 Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 164–174, 2011.

29 Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1267612.1267628.

30 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM. doi:10.1145/1133981.1134027.

31 Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/978-3-540-30579-8_14.

32 Rishi Surendran and Vivek Sarkar. Automatic parallelization of pure method calls via conditional future synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Con-*

ference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pages 20–38, New York, NY, USA, 2016. ACM. doi:10.1145/2983990.2984035.
33 Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):1–47, 2011. doi:10.1145/1961204.1961205.
34 Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094828.
35 Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 304–328, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1883978.1884000.
36 Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL):56:1–56:29, dec 2017. doi:10.1145/3158144.
37 Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 98–122, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03013-0_6.
38 Qirun Zhang and Zhendong Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 344–358, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009848.
39 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie, un, and Michael D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM. doi:10.1145/1287624.1287637.
40 Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 598–617, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869509.

A Proofs

Our main theorem follows from the following two lemmas.

▶ **Lemma 7.** *If* $G \Rightarrow S$ *and* $\{G, S\}$ $\textsc{Edge}(e(s))$ || $\textsc{Constraint}(c(s))$ $\{G', S'\}$ *then* $G' \Rightarrow S'$.

Proof. The proof relies on the fact that viewpoint adaptation preserves subtyping. That is, for each x, x' and p, $\mathsf{x} <: \mathsf{x}' \Rightarrow \mathsf{x} \rhd \mathsf{p} <: \mathsf{x}' \rhd \mathsf{p}$. Also, for each x, p and p', $\mathsf{p} <: \mathsf{p}' \Rightarrow \mathsf{x} \rhd \mathsf{p} <: \mathsf{x} \rhd \mathsf{p}'$. Therefore, for each x, x', p, and p', $\mathsf{x} <: \mathsf{x}' \wedge \mathsf{p} <: \mathsf{p}' \Rightarrow \mathsf{x} \rhd \mathsf{p} <: \mathsf{x}' \rhd \mathsf{p}'$.

We proceed by induction on the number of applications of

$\textsc{Edge}(e(s))$ || $\textsc{Constraint}(c(s))$

and case by case analysis.

Consider the most difficult case, case 1: s is $\mathsf{x} = \mathsf{y.m(z)}$. Naturally, it breaks into 3 smaller cases, (1) $\mathsf{z} \xrightarrow{(_i} \mathsf{p}$, (2) $\mathsf{y} \xrightarrow{(_i} \mathsf{this}$ and (3) $\mathsf{ret} \xrightarrow{)_i} \mathsf{x}$. (2) is analogous to (1).

For (1), suppose Edge adds $\mathsf{z} \overset{(_i \oplus M|C}{\rightsquigarrow} u$ to G'. By the inductive hypothesis, $\mathsf{p} \overset{M|C}{\rightsquigarrow} u$ implies $max(S(\mathsf{p})) <: \mathsf{mutable}$. The corresponding constraint $q_\mathsf{z} <: q_\mathsf{x} \triangleright q_\mathsf{p}$ sets $max(S'(\mathsf{z})) = \mathsf{mutable}$. Similarly, $\mathsf{p} \overset{(M|C)A}{\rightsquigarrow} u \Rightarrow max(S(\mathsf{p})) <: \mathsf{maybe}$ and constraint $q_\mathsf{z} <: q_\mathsf{x} \triangleright q_\mathsf{p}$ leads to $q_\mathsf{z} <: q_\mathsf{x} \triangleright \mathsf{maybe} <: \mathsf{maybe}$ by the above theorem. Thus, $max(S'(\mathsf{z})) <: \mathsf{maybe}$, as needed. Now, suppose that (1) adds $\mathsf{z} \overset{(_i \oplus R}{\rightsquigarrow} u$ to G'. This entails $max(S(\mathsf{ret})) <: \mathsf{poly}$. If $\mathsf{x} \overset{M|C}{\rightsquigarrow} u \in G$ then $max(S(\mathsf{x})) <: \mathsf{mutable}$, leading to $max(S'(\mathsf{z})) <: \mathsf{mutable}$. If $\mathsf{x} \overset{(M|C)A}{\rightsquigarrow} u \in G$ then $max(S(\mathsf{x})) <: \mathsf{maybe}$, leading to $max(S'(\mathsf{z})) = \mathsf{maybe}$, as needed. If $\mathsf{x} \overset{R}{\rightsquigarrow} u \in G$ then $max(S(\mathsf{x})) <: \mathsf{poly}$; constraint $q_\mathsf{z} <: q_\mathsf{x} \triangleright q_\mathsf{p} \equiv q_\mathsf{z} <: \mathsf{poly} \triangleright \mathsf{poly}$ leads to $max(S'(\mathsf{z})) <: \mathsf{poly}$, as needed.

For (3), suppose Edge adds $\mathsf{ret} \overset{)_i \oplus N}{\rightsquigarrow} u$ to G'. This happens only if there is $\mathsf{x} \overset{N}{\rightsquigarrow} u \in G$. Therefore by the inductive hypothesis $max(S(\mathsf{x})) \neq \mathsf{readonly}$, and therefore, constraint $q_\mathsf{x} \triangleright \mathsf{ret} <: q_\mathsf{x}$ entails that $max(S'(\mathsf{ret})) <: \mathsf{poly}$, as needed.

Case 2: s is $\mathsf{x} = \mathsf{y}$ is straightforward.

Case 3: s is $\mathsf{x.f} = \mathsf{y}$. Suppose Edge processes approximate edge $\mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f}$ followed by direct edge $\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x.f}$. Edge adds to G' only if $\mathsf{x'.f}$ in G, which by the inductive hypothesis entails $S(\mathsf{f}) = \{\mathsf{poly}\}$. Thus, $max(S(\mathsf{x.f}))$ evaluates to $\mathsf{mutable}$, and the desired subtyping is preserved (even though this typing is not precise). Furthermore, field write constraint $q_\mathsf{y} <: q_\mathsf{x} \triangleright q_\mathsf{f}$ evaluates to $q_\mathsf{y} <: \mathsf{maybe} \triangleright \mathsf{poly}$, meaning that $max(S(\mathsf{y})) <: \mathsf{maybe}$ as needed to account for the $(M|C)A$-path from y in G'.

Finally, consider case 4: s is $\mathsf{y} = \mathsf{x.f}$. There is new path in G' only if there is a path in G from y. If there is a path from y, then $max(S(\mathsf{y})) \neq \mathsf{readonly}$. The constraint for field write $q_\mathsf{x} \triangleright q_\mathsf{f} <: q_\mathsf{y}$ entails $max(S(\mathsf{f})) = \mathsf{poly}$ and $max(S(\mathsf{x})) <: max(S(\mathsf{y}))$. Therefore, $max(S(\mathsf{x.f}))$ and $max(S(\mathsf{x}))$ reflect the new paths through y. ◀

▶ **Lemma 8.** *If* $S \Rightarrow G$ *and* $\{G, S\}$ Edge$(e(s))$ || Constraint$(c(s))$ $\{G', S'\}$ *then* $S' \Rightarrow G'$.

Proof. The proof is by induction on the number of applications of

Edge$(e(s))$ || Constraint$(c(s))$

We begin with the most difficult case, case 1: s is $\mathsf{x} = \mathsf{y.m(z)}$.

The tables below examine all cases for parameter constraint $q_\mathsf{z} <: q_\mathsf{x} \triangleright q_\mathsf{p}$. Constraint $q_\mathsf{y} <: q_\mathsf{x} \triangleright q_\mathsf{this}$ is completely analogous. For brevity, sets S show only the maximal element. For example, when

$S(\mathsf{x}) = \{\mathsf{readonly}, \ldots \quad S(\mathsf{p}) = \{\mathsf{maybe}, \ldots \quad S(\mathsf{z}) = \{\mathsf{readonly}, \ldots$

the constraint $q_\mathsf{z} <: q_\mathsf{x} \triangleright q_\mathsf{p}$ removes $\mathsf{readonly}$ and poly from $S(\mathsf{z})$ resulting in:

$S'(\mathsf{z}) = \{\mathsf{maybe}, \ldots$

To preserve precision, we must show that after execution of the parallel Edge operation, there exist only an $(M|C)A$-path from z to an update. The last column of the table enumerates the paths from z: added by Edge$(\mathsf{z} \xrightarrow{(_i} \mathsf{p})$ given $S(\mathsf{x})$ and $S(\mathsf{p})$, and existing ones in G. Continuing with the example, $(M|C)A[\mathsf{p}]$ + $\mathsf{none}[\mathsf{ret}]$ + $\mathsf{none}[\mathsf{z}]$ reads as follows: (1) an $(M|C)A$-path is added through p (the inductive hypothesis $S \Rightarrow G$ entails that the maybe typing of p implies an $(M|C)A$-path from p), and no paths are added through ret and x (again, the maybe typing implies that there is no path through ret and x), and (2) there are no existing paths from z (due to the readonly typing of z in S). Therefore, there is only an $(M|C)A$-path from z, and this case preserves precision.

The cases of $S(\mathsf{z}) = \{\mathsf{mutable}\}$ and $S(\mathsf{p}) = \{\mathsf{readonly}, ...\}$ are not shown because neither of these cases triggers change to S, and it is trivial to argue $S' \Rightarrow G'$.

The following table enumerates the cases for $S(\mathsf{x}) = \{\mathsf{readonly}, ...\}$:

S(x)	S(p)	S(z)	S'(z)	G, EDGE(z $\xrightarrow{(_i}$ p)
{ readonly,	{ maybe,			$(M\|C)A$[p]+none[ret]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] + R[z]
{ readonly,	{ poly,			none[p] + none[x]
		{ readonly,	NO CHANGE	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	NO CHANGE	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] + R[z]
{ readonly,	{ polymaybe,			$(M\|C)A$[p]+none[x]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] + R[z]

The table below enumerates the cases for $S(\mathsf{x}) = \{\mathsf{maybe}, ...\}$:

S(x)	S(p)	S(z)	S'(z)	G, EDGE(z $\xrightarrow{(_i}$ p)
{ maybe,	{ maybe,			$(M\|C)A$[p]+none[ret]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] + R[z]
{ maybe,	{ poly,			none[p]$+(M\|C)A$[x]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] + R[z]
{ maybe,	{ polymaybe,			$(M\|C)A$[p]$+(M\|C)A$[x]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] + R[z]

The table below enumerates the cases for $S(\mathsf{x}) = \{\mathsf{poly}, ...\}$:

$S(\mathsf{x})$	$S(\mathsf{p})$	$S(\mathsf{z})$	$S'(\mathsf{z})$	G, Edge($\mathsf{z} \xrightarrow{(_i} \mathsf{p}$)
{ poly,	{ maybe,			$(M\|C)A$[p]+none[ret]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] $+ R$[z]
{ poly,	{ poly,			none[p]$+R$[x]
		{ readonly,	{ poly,	+none[z]
		{ maybe,	{ polymaybe	$+(M\|C)A$[z]
		{ poly,	NO CHANGE	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] $+ R$[z]
{ poly,	{ polymaybe,			$(M\|C)A$[p]$+R$[x]
		{ readonly,	{ polymaybe,	+none[z]
		{ maybe,	{ polymaybe,	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] $+ R$[z]

The table below enumerates the cases for $S(\mathsf{x}) = \{\mathsf{polymaybe}, ...\}$:

$S(\mathsf{x})$	$S(\mathsf{p})$	$S(\mathsf{z})$	$S'(\mathsf{z})$	G, Edge($\mathsf{z} \xrightarrow{(_i} \mathsf{p}$)
{ polymaybe,	{ maybe,			$(M\|C)A$[p]+none[ret]
		{ readonly,	{ maybe,	+none[z]
		{ maybe,	NO CHANGE	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] $+ R$[z]
{ polymaybe,	{ poly,			none[p]$+R$[x]$+(M\|C)A$[x]
		{ readonly,	{ polymaybe,	+none[z]
		{ maybe,	{ polymaybe,	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] $+ R$[z]
{ polymaybe,	{ polymaybe,			$(M\|C)A$[p]$+(M\|C)A$[x]$+R$[x]
		{ readonly,	{ polymaybe,	+none[z]
		{ maybe,	{ polymaybe,	$+(M\|C)A$[z]
		{ poly,	{ polymaybe,	$+R$[z]
		{ polymaybe,	NO CHANGE	$+(M\|C)A$[z] $+ R$[z]

Consider constraint $q_{\mathsf{x}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}$. If $S(\mathsf{x})$ is $\{\mathsf{readonly}, ...\}$ then there is no change to S and no change to G, and the statement holds. If $S(\mathsf{ret})$ is $\{\mathsf{poly}\}$, then, there is no change to S and Edge "adds" a path already in G, resulting in no change in G as well. Let $S(\mathsf{x})$ be any other value but $\{\mathsf{readonly}, ...\}$ and let $S(\mathsf{ret})$ be $\{\mathsf{readonly}, ...\}$. $S'(\mathsf{ret})$ becomes $\{\mathsf{poly}\}$ implying an R-path. Since $S(\mathsf{x})$ is of any other value but $\{\mathsf{readonly}, ...\}$, this means that a path from x to update, $\mathsf{x} \overset{N}{\rightsquigarrow} u$ does exist in G, and Edge($\mathsf{ret} \xrightarrow{)_i} \mathsf{x}$) results in R path from ret in G'. Therefore, the theorem holds.

Consider case 2, s is $\mathsf{x} = \mathsf{y}$. We enumerate all possibilities analogously. Again we omit the cases when $S(\mathsf{x}) = \{\mathsf{mutable}, ...\}$ as well as the case when $S(\mathsf{y}) = \{\mathsf{readonly}, ...\}$, as they are trivial.

S(x)	S(y)	S'(y)	G, Edge(y $\xrightarrow{\mathsf{d}}$ x)
{ maybe,	{ readonly,	{ maybe,	$(M\|C)A$[x]+none[y]
	{ maybe,	NO CHANGE	$(M\|C)A$[x]+$(M\|C)A$[y]
	{ poly,	{ polymaybe,	$(M\|C)A$[x]+R[y]
	{ polymaybe,	NO CHANGE	$(M\|C)A$[x]+$(M\|C)A$[y]+R[y]
{ poly,	{ readonly,	{ poly,	R[x]+none[y]
	{ maybe,	{ polymaybe,	R[x]+$(M\|C)A$[y]
	{ poly,	NO CHANGE	R[x]+R[y]
	{ polymaybe,	NO CHANGE	R[x]+$(M\|C)A$[y]+R[y]
{ polymaybe,	{ readonly,	{ polymaybe,	$(M\|C)A$[x]+R[x]+none[y]
	{ maybe,	{ polymaybe,	$(M\|C)A$[x]+R[x]+$(M\|C)A$[y]
	{ poly,	{ polymaybe,	$(M\|C)A$[x]+R[x]+R[y]
	{ polymaybe,	NO CHANGE	$(M\|C)A$[x]+R[x]+$(M\|C)A$[y]+R[y]

Now consider case 3, s is x.f = y, and corresponding constraints q_{y} <: maybe $\triangleright$ q_{f}. If f is readonly, then maybe $\triangleright$ q_{f} is readonly, and there is no change in S. By the inductive hypothesis, x′.f being readonly implies that there does not exist a read x′.f such that there is a path from x′.f to update in G. Thus, no path is added to G' through x.f thus preserving the paths from y and the theorem. If f is poly, then y becomes maybe, or lower in S', thus properly accounting for the $(M|C)A$-path from y through x.f that appears in G'.

Finally, consider case 4. s is y′ = x′.f. If f is readonly, then there does not exist a path from x′.f in G. If y′ is not readonly, then there exists a path form y′. Thus, f becomes poly in S', and x′ <: y′ in S'. In G', there are new paths from x′.f and x reflecting S'. ◀