Definite Reference Mutability

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

Abstract
Reference immutability type systems such as Java 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

```java
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.
of reference immutability on “flow systems”. Flow systems track and prevent flow from positive references to negative ones:

```java
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

```java
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.

```java
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 Reference 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

\[ x.f = y; \quad \ldots \quad w = z.f; w.g = \ldots \]

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., \( x, y, \text{this} \), and field access expressions, e.g., \( x.f, y.f, \text{this.f} \). 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

\[ x = y; z = x; z.f = w \]
We elaborate upon approximate edges shortly. A field read statement
\[ x.g = z \]
that the impact of approximate paths is quite muted. Generally, when there is an approximate
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

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:
\[ x = y \Rightarrow y \xrightarrow{d} x \]
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 \( x'.f \in G \), where \( x'.f \) is the right-hand-side of a
field read \( y' = x'.f \). (Without loss of generality we may assume \( x' \neq x \).)
\[ x.f = y \Rightarrow y \xrightarrow{d} x.f \xrightarrow{a} x'.f \]
We elaborate upon approximate edges shortly. A field read statement \( y' = x'.f \) contributes
direct edges as follows:
\[ y' = x'.f \Rightarrow x' \xrightarrow{d} x'.f \xrightarrow{d} y' \]
Edge \( x' \xrightarrow{d} x'.f \) 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
\[ x.f = y, y' = x'.f \Rightarrow y \xrightarrow{d} x.f \xrightarrow{a} x'.f \xrightarrow{d} y' \]
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);
    ...
}

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{figure2}
\caption{Running example. Code (adapted from Huang et al. [18]) and corresponding graph.}
\end{figure}

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 \textit{call} edges from actual arguments to formal parameters:

\[ i: x = y.m(z) \Rightarrow y \xrightarrow{i} \text{this} \xrightarrow{i} z \xrightarrow{\ell} p \]

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

\[ i: x = y.m(z) \Rightarrow \text{ret} \xrightarrow{\ell} x \]

The standard CFL-reachability annotation \( \ell \), marks a return at site \( i \). In terms of Reps’ terminology, the semantics models call-transmitted dependences precisely.
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., $(d)$, $C$ captures strings with one or more outstanding calls, e.g., $(l,d)$, and $R$ and $R'$ capture strings with one or more outstanding returns, e.g., $d$.

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, $\text{this} \cdot \text{getDate} \rightarrow rd$ is an exact path, while $d \rightarrow md$ is an approximate path. (We use squiggle arrows $\rightarrow$ 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 $\text{this} \cdot \text{GetHours} \rightarrow rd$ is an $M$-path. However, $\text{this} \cdot \text{GetHours} \rightarrow md$ 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, $\text{this} \cdot \text{GetHours} \rightarrow \text{ret} \cdot \text{get} \cdot \text{Date}$ 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, $\text{this} \cdot \text{get} \cdot \text{Date} \rightarrow \text{md}$ 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, $\text{this} \cdot \text{get} \cdot \text{Date}$ and $\text{ret} \cdot \text{get} \cdot \text{Date}$ are polymorphic. They are interpreted as mutable when $\text{get} \cdot \text{Date}$ is called from $\text{cell} \cdot \text{Set} \cdot \text{Hours}$, and they are interpreted as readonly when $\text{get} \cdot \text{Date}$ is called from $\text{cell} \cdot \text{Get} \cdot \text{Hours}$.
2.2.2 Approximate Paths

The following grammar rules capture approximate paths:

\[
A \ ::= \ a | aE | aA | EA
\]

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,

\[
n \xrightarrow{E} \xrightarrow{a} \xrightarrow{E} \xrightarrow{a} \xrightarrow{E} u
\]

is an A-path, and so is

\[
n \xrightarrow{a} \xrightarrow{E} \cdots \xrightarrow{a} u
\]

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,

\[
d \xrightarrow{jm} p_{DateCell} \xrightarrow{d} \text{this}_{DateCell}.date \xrightarrow{a} \text{this}_{DateCell}.date \xrightarrow{d} \text{retget}_{Date} \xrightarrow{j} md
\]

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 an \((M|C)A\)-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,
Definite Reference Mutability

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 \( x \)
  - \( z = \text{id}(x); z.f = w \)
  - \( y = x.f; y.g = z \)

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

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

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

Below, we explain viewpoint adaptation in terms of the mutability semantics. At field accesses (TREAD) and (TWRITE) \( \triangleright \) 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 \( \triangleright \) 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
Definite Reference Mutability

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 \texttt{x.f} is inferred mutable, do we infer that field \texttt{f} is mutable and is excluded from the state of a readonly \texttt{x}, or do we infer that \texttt{f} is just a “regular” field and a mutable \texttt{x.f} signals deep mutation of \texttt{x} and \texttt{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:

\[
\begin{align*}
_\triangleright \text{mutable} &= \text{mutable} \\
_\triangleright \text{readonly} &= \text{readonly} \\
q \triangleright \text{poly} &= q
\end{align*}
\]

The underscore denotes a “don’t care” value. Qualifiers \text{mutable} and \text{readonly} do not depend on the viewpoint. Qualifier \text{poly} depends on the viewpoint (context), and is substituted by that viewpoint (context).

Let us take a closer look at rules (\text{twrite}) and (\text{tread}). For a pair of field write \texttt{x.f = y} and field read \texttt{y' = x'.f}, the rules entail the following constraints:

\[
q_y \ll q_x \triangleright q_t \quad q_y \triangleright q_t \ll q_y'
\]

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

Now, consider rule (\text{tcall}). Function \text{typeof} retrieves the type of compile-time target \texttt{m}. \(q_{\text{this}}\) is the type of parameter \texttt{this}, \(q_p\) is the type of the formal parameter, and \(q_{\text{return}}\) is the type of the return. Rule (\text{tcall}) requires \(q_x \triangleright q_{\text{return}} \ll q_x\), which accounts for \text{R}-paths. The constraint disallows the return value of \texttt{m} from being \text{readonly} when there is a call to \texttt{m} when \texttt{x = y.m(z)}, where left-hand-side \texttt{x} is \text{mutable}. Only if the left-hand-sides of all call assignments to \texttt{m} are \text{readonly}, can the return type of \texttt{m} be \text{readonly}; otherwise, it is \text{poly}. A programmer can annotate the return type of \texttt{m} as \text{mutable}. However, this typing is pointless, as it unnecessarily forces local variables and parameters in \texttt{m} to become \text{mutable} when they may remain less restrictively \text{poly}. In Fig. 2, \(md = \text{this}\.\text{getDate}();\) entails constraint

\[
q_{\text{md}} \triangleright q_{\text{return}\.\text{getDate}} \ll q_{\text{md}} \equiv \text{mutable} \triangleright q_{\text{return}\.\text{getDate}} \ll \text{mutable}
\]

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

\[
q_{\text{this}\.\text{get}\.\text{Date}} \triangleright q_{\text{date}} \ll q_{\text{return}\.\text{getDate}}
\]

leading to \(q_{\text{this}\.\text{get}\.\text{Date}} = \text{poly}\), which accounts for the \text{R}-path from \text{this}\.\text{get}\.\text{Date} to the update \text{md}. 


Additionally, rule (TCALL) requires \( q_y <: q_x > q_{\text{this}} \). When \( q_{\text{this}} \) is readonly or mutable, its adapted value is the same. Thus, when \( q_{\text{this}} \) is mutable (e.g., due to \( \text{this.f} = 0 \) in \( m \)),

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

which disallows \( q_y \) from being anything but mutable, as expected. This accounts for \( \text{C-} \) and \( \text{CA-} \) paths. The interesting case is when \( q_{\text{this}} \) is poly. A poly parameter \( \text{this} \) reflects a dependence between \( \text{this} \) and \( \text{ret} \) of \( m \), such as the one in Fig. 2:

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

It allows the \( \text{this} \) object (or some part of it, in our example the \( \text{date} \) part of it), to be modified in caller context, after \( m \)'s return. The type system entails that whenever there is intraprocedural dependence between \( \text{this} \) and \( \text{ret} \) of \( m \), such as the one in Fig. 2:

\[
q_{\text{this}} <: q_{\text{ret}}.
\]

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

\[
q_y <: q_x > q_{\text{this}} \quad q_{\text{this}} <: q_{\text{ret}} \quad q_x > q_{\text{ret}} <: q_x
\]

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, \( q_x > q_{\text{this}} \) 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:

\[
\text{maybe} \quad \text{poly} \\
\text{mutable} <: \text{polymaybe} <: \text{readonly}
\]
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:

\[
\begin{align*}
_\triangleright maybe & = maybe \\
q \triangleright polymaybe & = (q \triangleright poly) \land maybe
\end{align*}
\]

Notation $\land$ stands for the standard meet operation: the result of $q_1 \land q_2$ 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 \{readonly, 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 $x = y.m(z)$ creates an $(M|C)A$ path from $z$ (a $CA$-path to be precise). Rule (TCALL) requires

\[ q_z <: q_x \triangleright q_p \equiv q_z <: maybe \]

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

\[ q_z <: (readonly \triangleright poly) \land maybe \equiv q_z <: maybe \]

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:

\[ q_z <: (mutable \triangleright poly) \land maybe \equiv q_z <: mutable \]

Consider the more detailed example:

```java
class A {
  ...
  C m(A this, B p) {
    C c = this.f;
    p.g = c;
    return c;
  }
}
```
The $R$-path from $this_m$ to $c1$, entails $q_{this_m} <: poly$, while the $(M|C)A$-path through $p.g$ entails $q_{this_m} <: maybe$. Thus, $q_{this_m} = polymaybe$. At call 11 we have

$q_{a1} <: q_{c1} \triangleright q_{this_m} \equiv q_{a1} <: q_{c1} \triangleright polymaybe$.

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

$q_{a2} <: q_{c2} \triangleright q_{this_m} \equiv q_{a2} <: q_{c2} \triangleright polymaybe \equiv q_{a2} <: maybe$.

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

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

\[
\begin{align*}
\Gamma(x) &= q_x, & q_x &= \text{mutable} \\
\text{typeof}(f) &= q_f, & q_f &= maybe \\
\Gamma(y) &= q_y \\
\Gamma \vdash x.f = y
\end{align*}
\]

$q_x$ 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 $q_y$ is readonly, then the maybe-mutability of $x$ does not affect $y$. If $q_y$ is poly, that reflects an update to some $f$, and $q_y <: maybe \triangleright poly$ 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, $S(x) = \{\text{poly, mutable}\}$ in ReIm means the type of reference $x$ can be poly or mutable.

$S$ is initialized as follows. $S(\text{ret}) = S(f) = \{\text{readonly, poly}\}$ for all return values ret and fields f. The rest of the variables are initialized to the universal set of qualifiers $U$, which is \{\text{readonly, poly, mutable}\} in ReIm, and \{\text{readonly, maybe, poly, polymaybe, 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 $v_1, v_2, \ldots, v_k$ and let $s$ entail transfer function $c(s)$. Applying $c(s)$ removes each $q_1$ from $S(v_1)$ when there are no qualifiers $q_2 \in S(v_2), \ldots, q_k \in S(v_k)$, such that $q_1, q_2, \ldots, q_k$ make $s$ type check; then it removes all $q_2$ from $S(v_2)$, etc. For example, consider $s$: $y = x.f$ and corresponding rule (TREAD) triggering constraint $q_s \triangleright q_f <: q_r$. Let $S(x), S(f)$ and $S(y)$ be as follows:

\[
\begin{array}{|c|c|c|}
\hline
S(x) & S(f) & S(y) \\
\hline
\{\text{maybe, polymaybe, mutable}\} & \{\text{readonly, poly}\} & \{\text{poly, polymaybe, mutable}\} \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|c|}
\hline
S'(x) & S'(f) & S'(y) \\
\hline
\{\text{polymaybe, mutable}\} & \{\text{poly}\} & \{\text{poly, polymaybe, mutable}\} \\
\hline
\end{array}
\]

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

\[
\{\text{poly, polymaybe, mutable}\} > \{\text{poly, polymaybe, mutable}\} > \{\text{mutable}\} > \{\text{mutable}\} > \{\text{poly, polymaybe, mutable}\} > U
\]

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 $L \rightarrow L$ 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 $S_{\text{fix}}$. 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 $S_{\text{fix}}(x)$ according to the following ranking, which mirrors the subtyping lattice:

\[
\text{maybe} > \text{poly} > \text{polymaybe} > \text{mutable}
\]

Importantly, the maximal element exists because each $S_{\text{fix}}(x)$ is an element of $L_v$. We denote this typing by $\text{max}(S_{\text{fix}})$, 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(SFix) and ReM’s max(SFix) 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 \( q_x \prec q_y \prec q_p \). Let \( S_{\text{Fix}}(p) = \{\text{poly, polymaybe, mutable}\} \) and \( S_{\text{Fix}}(z) = \{\text{maybe, polymaybe, mutable}\} \). \( S_{\text{Fix}}(x) \) must be either \{maybe, polymaybe, mutable\} or \{readonly, maybe, poly, polymaybe, mutable\}. If it were any other set, then maybe would have been removed from \( S(z) \) during fixpoint iteration. Combinations \( q_x = \text{maybe}, q_y = \text{maybe}, q_p = \text{poly}, \) and \( q_z = \text{maybe}, q_x = \text{readonly}, q_p = \text{poly} \), maximal typings under the two cases, both typecheck.

A more general statement is true. For every \( S \) that satisfies the equations of the dataflow framework, 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:

\[
o_{\text{ReIm}}(T) = (|T^{-1}(\text{readonly})|, |T^{-1}(\text{poly})|, |T^{-1}(\text{mutable})|)
\]

The tuples are ordered lexicographically. We have \( T_1 > T_2 \) iff \( T_1 \) has more \text{readonly} references than \( T_2 \), or \( T_1 \) and \( T_2 \) have the same number of \text{readonly} references, but \( T_1 \) has more \text{poly} references than \( T_2 \). The preference ranking over typings is based on ranking over qualifiers: naturally, we prefer \text{readonly} over \text{poly} and \text{mutable}, and \text{poly} over \text{mutable}.

ReM’s objective function is the following:

\[
o_{\text{ReM}}(T) = (|T^{-1}(\text{readonly})|, |T^{-1}(\text{poly})| + |T^{-1}(\text{maybe})|, |T^{-1}(\text{polymaybe})|, |T^{-1}(\text{mutable})|)
\]

Again the tuples are ordered based on the natural ranking over qualifiers: \text{readonly} is the most preferred, followed by \text{poly} and \text{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 \( o_{\text{ReIm}} \) over ReIm, or \( o_{\text{ReM}} \) over ReM). \( o(\text{max}(S_{\text{Fix}})) > o(T) \) holds for every valid typing \( T \neq \text{max}(S_{\text{Fix}}) \).

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 \( S > S_{\text{Fix}} \) by virtue of \( S_{\text{Fix}} \) being the maximal fixpoint solution, for every variable \( x \) \( S(x) \geq S_{\text{Fix}}(x) \), and there exist variables \( y \) such that \( S(y) > S_{\text{Fix}}(y) \). Thus, \( o_{\text{ReM}}(\text{max}(S_{\text{Fix}})) > o_{\text{ReM}}(\text{max}(S)) \).

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
Definite Reference Mutability

1: procedure CFL
2: \( G \leftarrow \emptyset \)
3: Add \( u \ef f u \) to \( G \) for all updates \( u \)
4: while \( G \) changes do
5: for each \( s \) in Program do
6: \( \text{EDGE}(e(s)) \)
7: end for
8: end while
9: end procedure

1: procedure Types
2: \( S(n) = U \)
3: \( S(n) = \{ \text{mutable} \} \) for all updates \( n \)
4: while \( S \) changes do
5: for each \( s \) in Program do
6: \( \text{Constraint}(c(s)) \)
7: end for
8: end while
9: end procedure

\[ \begin{align*}
\text{Figure 5} & \quad \text{Algorithm CFL initializes } G, \text{ then iterates over program statements } s \text{ adding edges as specified in Sect. 2.1. Algorithm TYPES initializes } S, \text{ then iterates over program statements } s \text{ removing qualifies from } S \text{ as specified in Sect. 3.3. The algorithms elide details to highlight the "parallel" structure of the two systems.}
\end{align*} \]

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

\[ y' = x'.f \quad \Rightarrow \quad x' \xrightarrow{d} x'.f \xrightarrow{d} y' \]

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

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

\[
\begin{align*}
(i \oplus M|C) &= M|C \\
(i \oplus (M|C)A) &= (M|C)A \\
(i \oplus \_) &= R \\
d \oplus N &= N \\
a \oplus \_ &= (M|C)A \\
\end{align*}
\]

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

\[ p \xrightarrow{M} \xrightarrow{\ell} x \xrightarrow{N} u \]

where \( \text{ret} \) is the return value of the target method, and \( x \) is some left-hand-side of a call assignment. Note that \( p \xrightarrow{M} \text{ret} \xrightarrow{\ell} x \) are not explicitly in \( G \), but \( x \xrightarrow{N} u \) 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 \( z \xrightarrow{N} u \) to \( G \).
Let us illustrate CFl and EDGE, and Types and Constraint in parallel. Consider

\[ y = x; z = y; z.f = w \]

Calling EDGE on \( y \xrightarrow{d} z \), which corresponds to statement \( z = y \), leads to path \( y \xrightarrow{M} z \) in \( G \). Subsequently calling EDGE on edge \( x \xrightarrow{d} y \) leads to concatenation of \( d \) and \( M \) and path \( x \xrightarrow{M} z \). There are two \( M \)-paths, \( x \) to \( z \) and \( y \) to \( z \), as expected. Analogously, calling Constraint on \( q_x < \colon q_y \), which corresponds to statement \( z = y \), removes all qualifiers but \( \text{mutable} \) from \( S(y) \). Subsequently calling Constraint on \( q_x < \colon q_y \) removes all qualifiers but \( \text{mutable} \) from \( S(x) \). \( S(x) = \{\text{mutable}\} \), and \( S(y) = \{\text{mutable}\} \) 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

\[ y = \text{id}(x); y = z; z.f = w \]

and assume \( y \xrightarrow{M} z \) and \( p \xrightarrow{R} z \) are already in \( G \). Concatenation breaks the \( R \)-path \( p \xrightarrow{R} z \) into \( p \xrightarrow{M} \text{ret} \xrightarrow{R} y \xrightarrow{M} z \). Since \( (, \) and \( ) \), match, it adds path \( x \xrightarrow{M} z \). Analogously, for Types assume \( S(p) = \{\text{poly}, \text{polymaybe}, \text{mutable}\} \) and \( S(y) = \{\text{mutable}\} \). Calling Constraint on \( q_x < \colon q_y \) \( \Rightarrow q_y \) removes all qualifiers but \( \text{mutable} \) from \( S(x) \), which directly corresponds to the \( M \)-path from \( x \) to \( z \) that CFl finds.

To formally establish equivalence we use the bisimulation 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 \( x.f \).

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

1. \( n \xrightarrow{M|C} u \in G \Rightarrow \max(S(n)) \subseteq \text{mutable} \)
2. \( n \xrightarrow{R} u \in G \Rightarrow \max(S(n)) \subseteq \text{poly} \)
3. \( n \xrightarrow{(M|C)A} u \in G \Rightarrow \max(S(n)) \subseteq \text{maybe} \)

Def. 4 states that \( n \)'s maximal type in \( S \) implies a corresponding path in \( G \). For example, maximal typing \( \text{polymaybe} \) 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. \( \max(S(x)) = \text{mutable} \Rightarrow \exists x \xrightarrow{M|C} u \in G \)
2. \( \max(S(x)) = \text{polymaybe} \Rightarrow \exists x \xrightarrow{R} u \in G \land \exists x \xrightarrow{(M|C)A} u \in G \land \neg \exists x \xrightarrow{M|C} u \in G \)
3. \( \max(S(x)) = \text{poly} \Rightarrow \exists x \xrightarrow{R} u \in G \land \neg \exists x \xrightarrow{(M|C)A} u \in G \land \neg \exists x \xrightarrow{M|C} u \in G \)
4. \( \max(S(x)) = \text{maybe} \Rightarrow \neg \exists x \xrightarrow{R} u \in G \land \exists x \xrightarrow{(M|C)A} u \in G \land \neg \exists x \xrightarrow{M|C} u \in G \)
5. \( \max(S(x)) = \text{readonly} \Rightarrow \) no path from \( x \) in \( G \)
6. \( \max(S(x.f)) = \text{readonly} \Rightarrow \) no path from \( x.f \) 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 \( s \):

\[
\{G, S\} \quad \text{Edge}(e(s)) \parallel \text{Constraint}(c(s)) \quad \{G', S'\}
\]

Our key result is the following theorem:

Theorem 6. If \( G \simeq S \) and \( \{G, S\} \text{Edge}(e(s)) \parallel \text{Constraint}(c(s)) \{G', S'\} \) then \( G' \simeq S' \).

Proof Sketch. As expected, the proof is by induction on the number of applications of \( \text{Edge}(e(s)) \parallel \text{Constraint}(c(s)) \). Clearly, the statement holds after initialization, lines 2-3 in \( \text{Cfl} \) and lines 2-3 in \( \text{Types} \). 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 \( \text{Edge}(e(s)) \) and \( \text{Constraint}(c(s)) \), \( G' \Rightarrow S' \) still holds. We outline the most difficult case, \( \text{Edge}(z \xrightarrow{i} p) \parallel \text{Constraint}(q_x <: q_y) \) (method call naturally brakes into three steps). Consider \( z \xrightarrow{i} p \oplus p \xrightarrow{R} u \). Let \( x \) be the left-hand-side at call assignment \( i \). If there does not exist a path \( x \xrightarrow{N} u \in G \), then no new paths are added to \( G' \) and \( G' \Rightarrow S' \) holds. If there exists \( x \xrightarrow{N} u \in G \), then a new path \( z \xrightarrow{N} u \) is added to \( G' \). We must show that \( S'(z) \) reflects \( N \) according to Def. 3 (e.g., if \( N \) is an \( M[C] \)-path, then \( z \) is \( \text{mutable} \)). By the inductive hypothesis \( p \xrightarrow{R} u \in G \Rightarrow \text{max}(S(p)) <: \text{poly} \). Similarly, the \( N \)-path entails appropriate \( S(x) \): if \( N \) is an \( M[C] \) path, then \( S(x) = \{\text{mutable}\} \), if \( N \) is an \( R \) path then \( \text{max}(S(x)) <: \text{poly} \), and if \( N \) is a \( (M|C)A \)-path, then \( \text{max}(S(x)) <: \text{maybe} \). Constraint \( q_x <: q_y \) removes qualifiers from \( S(z) \). For example, if \( N \) is \( M[C] \), then \( S(x) = \{\text{mutable}\} \), and it is easy to see that \( \text{max}(S(x)) >: \text{max}(S(p)) <: \text{mutable} \). Thus \( \text{max}(S'(z)) :<: \text{mutable} \), 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, \( \text{Edge}(z \xrightarrow{i} p) \parallel \text{Constraint}(q_x <: q_y) \), and the following values of \( S(x), S(p) \) and \( S(z) \) (only the maximal element shown):

<table>
<thead>
<tr>
<th>( S(x) )</th>
<th>( S(p) )</th>
<th>( S(z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ readonly, ... }</td>
<td>{ maybe, ... }</td>
<td>{ readonly, ... }</td>
</tr>
</tbody>
</table>

Constraint \( q_x <: q_y \) “lowers” \( S(z) \) into \( S'(z) = \{\text{maybe}, ...\} \). By the inductive hypothesis, \( S(x) \) and \( S(p) \) 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, \( \text{Edge}(z \xrightarrow{i} p) \) adds an \( (M|C)A \)-path from \( z \in G' \), and no other kind of path. Thus, \( \text{max}(S'(z)) = \text{maybe} \Rightarrow z \xrightarrow{(M|C)A} u \), 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

\[
q_{\text{this}_{m}} \xrightarrow{d} q_{\text{this}_{m'}}, \quad q_{\text{am}} \xrightarrow{d} q_{\text{pm}'}, \quad q_{\text{ret}_{m'}} \xrightarrow{d} q_{\text{ret}_{m}}
\]

to \( G \). Analogously, we require

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

\[(q_{\text{this}_m'}, q_{\text{p}_m} \rightarrow q_{\text{ret}_m'}) <:\ (q_{\text{this}_m}, q_{\text{p}_m} \rightarrow q_{\text{ret}_m})\]

which leads to the standard function subtyping constraints:

\[q_{\text{this}_m} <:\ q_{\text{this}_m'}, \quad q_{\text{p}_m} <:\ q_{\text{p}_m'}, \quad q_{\text{ret}_m} <:\ q_{\text{ret}_m}\]

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
Definite Reference Mutability

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).

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

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 $y \xrightarrow{d} x. f \xrightarrow{a} x'. f \xrightarrow{d} y'$ 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 $f$:

```java
1    f = new X[10];
2    for (int i=0; i<cnt; i++)
3        f[i] = new X();
4    ...
```

This code snippet is translated into the following Jimple:

```java
1    r1 = newarray (X)[10];
2    this.f = r1;
3    ...
4    r2 = this.f;
5    x = new X();
6    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 $q_{r1} <: q_{m.this.f}$ 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 $q_{m.this.f} <: q_{r2}$. Thus, when $r2$ is `mutable` or `poly/polymaybe`, `mutable` or `poly/polymaybe` propagates through $q_{m.this.f}$, 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

```java
1 class Container {
2    Data data;
3    void set(Container this, Data p) {
4        this.data = p;
5    }
6    Data get(Container this) {
7        return this.data;
8    }
9 }
```

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Readonly</th>
<th>#Poly</th>
<th>#Maybe/ #PolyOrMaybe</th>
<th>#Mutable</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>15751</td>
<td>1029</td>
<td>1083/6</td>
<td>12653</td>
<td>92.0%</td>
</tr>
<tr>
<td>biost</td>
<td>12567</td>
<td>2299</td>
<td>3683/68</td>
<td>19029</td>
<td>83.5%</td>
</tr>
<tr>
<td>chart</td>
<td>31640</td>
<td>4973</td>
<td>7329/154</td>
<td>32389</td>
<td>81.2%</td>
</tr>
<tr>
<td>eclipse</td>
<td>10945</td>
<td>4118</td>
<td>2473/46</td>
<td>12937</td>
<td>83.7%</td>
</tr>
<tr>
<td>fop</td>
<td>47411</td>
<td>5563</td>
<td>5235/93</td>
<td>35880</td>
<td>87.1%</td>
</tr>
<tr>
<td>hsqldb</td>
<td>29907</td>
<td>4282</td>
<td>3303/77</td>
<td>26418</td>
<td>88.8%</td>
</tr>
<tr>
<td>luindex</td>
<td>5465</td>
<td>686</td>
<td>677/13</td>
<td>4675</td>
<td>87.1%</td>
</tr>
<tr>
<td>lusearch</td>
<td>6296</td>
<td>945</td>
<td>932/17</td>
<td>5430</td>
<td>85.1%</td>
</tr>
<tr>
<td>jedis</td>
<td>37758</td>
<td>4605</td>
<td>4989/106</td>
<td>34345</td>
<td>87.1%</td>
</tr>
<tr>
<td>xalan</td>
<td>21254</td>
<td>3337</td>
<td>2634/71</td>
<td>19982</td>
<td>88.1%</td>
</tr>
<tr>
<td>jolden</td>
<td>1208</td>
<td>281</td>
<td>187/9</td>
<td>1040</td>
<td>84.6%</td>
</tr>
<tr>
<td>javad</td>
<td>796</td>
<td>45</td>
<td>5/0</td>
<td>429</td>
<td>99.3%</td>
</tr>
<tr>
<td>ejc</td>
<td>29062</td>
<td>8202</td>
<td>3150/194</td>
<td>25214</td>
<td>88.3%</td>
</tr>
<tr>
<td>tinySQL</td>
<td>6115</td>
<td>665</td>
<td>608/7</td>
<td>3615</td>
<td>85.5%</td>
</tr>
<tr>
<td>htmlparser</td>
<td>7787</td>
<td>1215</td>
<td>1023/12</td>
<td>4980</td>
<td>82.0%</td>
</tr>
<tr>
<td>commons-pool</td>
<td>847</td>
<td>25</td>
<td>222/0</td>
<td>642</td>
<td>74.5%</td>
</tr>
<tr>
<td>jdk</td>
<td>1134</td>
<td>317</td>
<td>293/9</td>
<td>1613</td>
<td>84.2%</td>
</tr>
<tr>
<td>jsib</td>
<td>4017</td>
<td>312</td>
<td>281/2</td>
<td>2387</td>
<td>89.4%</td>
</tr>
<tr>
<td>jsds</td>
<td>8628</td>
<td>825</td>
<td>552/25</td>
<td>5773</td>
<td>90.0%</td>
</tr>
<tr>
<td>javalang</td>
<td>3336</td>
<td>350</td>
<td>217/1</td>
<td>4753</td>
<td>95.6%</td>
</tr>
<tr>
<td>javutil</td>
<td>3216</td>
<td>376</td>
<td>276/0</td>
<td>4791</td>
<td>94.6%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>87.3%</td>
</tr>
</tbody>
</table>

Definite Reference Mutability

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 Steensgaard-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 ($x = \text{Class.forName("className")}.\text{newInstance()}$), for free. We still face issues with reflective method invocation ($\text{getMethod}$). 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


A Proofs

Our main theorem follows from the following two lemmas.

Lemma 7. If \( G \Rightarrow S \) and \( \{G, S\} \) \text{Edge}(e(s)) \| \text{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 \( p \), \( x <: x' \Rightarrow x:p <: x':p \). Also, for each \( x, p \) and \( p' \), \( x:p \Rightarrow x:p' \). Therefore, for each \( x, x', p, \) and \( p' \), \( x <: x' \land p <: p' \Rightarrow x:p <:< x':p' \).

We proceed by induction on the number of applications of \text{Edge}(e(s)) \| \text{Constraint}(c(s))

and case by case analysis.

Consider the most difficult case, case 1: \( s = x = y.m(z) \). Naturally, it breaks into 3 smaller cases, (1) \( z \xrightarrow{t} p \), (2) \( y \xrightarrow{t} \text{this} \) and (3) \( \text{ret} \xrightarrow{t} x \). (2) is analogous to (1).
For (1), suppose \( \text{EDGE} \) adds \( z \xrightarrow{(\oplus M)C} u \) to \( G' \). By the inductive hypothesis, \( p^M \xrightarrow{C} u \) implies \( \max(S(p)) <: \text{mutable} \). The corresponding constraint \( q_2 <: q_x > q_p \) sets \( \max(S'(z)) = \text{mutable} \). Similarly, \( p \xrightarrow{(M)A} u \Rightarrow \max(S(p)) <: \text{maybe} \) and constraint \( q_2 <: q_x > q_p \) leads to \( q_2 <: q_x > \text{maybe} <: \text{maybe} \) by the above theorem. Thus, \( \max(S'(z)) <: \text{maybe} \), as needed.

Now, suppose that (1) adds \( z \xrightarrow{(\oplus R)C} u \) to \( G' \). This entails \( \max(S(\text{ret})) <: \text{poly} \). If \( x \xrightarrow{(M)C} u \in G \) then \( \max(S(x)) <: \text{mutable} \), leading to \( \max(S'(z)) <: \text{mutable} \). If \( x \xrightarrow{(M)A} u \in G \) then \( \max(S(x)) <: \text{maybe} \), leading to \( \max(S'(z)) = \text{maybe} \), as needed. If \( x \xrightarrow{R} u \in G \) then \( \max(S(x)) <: \text{poly} \); constraint \( q_2 <: q_x > q_p \equiv q_2 <: \text{poly} \Rightarrow \text{poly} \) leads to \( \max(S'(z)) <: \text{poly} \), as needed.

For (3), suppose \( \text{EDGE} \) adds \( \text{ret} \xrightarrow{(\oplus N)C} u \) to \( G' \). This happens only if there is \( x \xrightarrow{N} u \in G \). Therefore by the inductive hypothesis \( \max(S(x)) \neq \text{readonly} \), and therefore, constraint \( q_x > \text{ret} <: q_x \) entails that \( \max(S'(\text{ret})) <: \text{poly} \), as needed.

Case 2: \( s \) is \( x = y \). Suppose \( \text{EDGE} \) processes approximate edge \( x.f \xrightarrow{a} x'.f \) followed by direct edge \( y \xrightarrow{a} x.f \). \( \text{EDGE} \) adds to \( G' \) only if \( x'.f \) in \( G \), which by the inductive hypothesis entails \( S(f) = \{ \text{poly} \} \). Thus, \( \max(S(x.f)) \) evaluates to \( \text{mutable} \), and the desired subtyping is preserved (even though this typing is not precise). Furthermore, field write constraint \( q_y <: q_x > q_t \) evaluates to \( q_y <: \text{maybe} > \text{poly} \), meaning that \( \max(S(y)) <: \text{maybe} \) as needed to account for the \( (M)A \)-path from \( y \) in \( G' \).

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

**Lemma 8.** If \( S \Rightarrow G \) and \( \{G, S\} \text{ EDGE}(e(s)) \| \text{CONSTRAINT}(c(s)) \{G', S'\} \) then \( S' \Rightarrow G' \).

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

\[ \text{EDGE}(e(s)) \| \text{CONSTRAINT}(c(s)) \]

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

The tables below examine all cases for parameter constraint \( q_2 <: q_x > q_p \). Constraint \( q_2 <: q_x > q_p \) is completely analogous. For brevity, sets \( S \) show only the maximal element. For example, when

\[ S(x) = \{ \text{readonly} \} \quad S(p) = \{ \text{maybe} \} \quad S(z) = \{ \text{readonly} \} \]

the constraint \( q_2 <: q_x > q_p \) removes \text{readonly} and \text{poly} from \( S(z) \) resulting in:

\[ S'(z) = \{ \text{maybe} \} \]

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

<table>
<thead>
<tr>
<th>( S(x) )</th>
<th>( S(p) )</th>
<th>( S(z) )</th>
<th>( S'(z) )</th>
<th>( G, \text{ EDGE}(z \xrightarrow{\ell} p) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ readonly, }</td>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ (M(C)).A[p] + \text{none}[\text{ret}] }</td>
</tr>
<tr>
<td>{ readonly, }</td>
<td>{ maybe, }</td>
<td>{ poly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
</tr>
<tr>
<td>{ readonly, }</td>
<td>{ maybe, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ readonly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ readonly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ readonly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ readonly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
</tbody>
</table>

The table below enumerates the cases for \( S(x) = \{\text{maybe,} \ldots\} \):

<table>
<thead>
<tr>
<th>( S(x) )</th>
<th>( S(p) )</th>
<th>( S(z) )</th>
<th>( S'(z) )</th>
<th>( G, \text{ EDGE}(z \xrightarrow{\ell} p) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ (M(C)).A[p] + \text{none}[\text{ret}] }</td>
</tr>
<tr>
<td>{ maybe, }</td>
<td>{ poly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ maybe, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
</tbody>
</table>

The table below enumerates the cases for \( S(x) = \{\text{maybe,} \ldots\} \):

<table>
<thead>
<tr>
<th>( S(x) )</th>
<th>( S(p) )</th>
<th>( S(z) )</th>
<th>( S'(z) )</th>
<th>( G, \text{ EDGE}(z \xrightarrow{\ell} p) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ maybe, }</td>
<td>{ (M(C)).A[p] + \text{none}[\text{ret}] }</td>
</tr>
<tr>
<td>{ maybe, }</td>
<td>{ poly, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ maybe, }</td>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
<tr>
<td>{ polymaybe, }</td>
<td>{ NO CHANGE }</td>
<td>{ NO CHANGE }</td>
<td>{ NOCHANGE }</td>
<td>{ +\text{none}[z] }</td>
</tr>
</tbody>
</table>
The table below enumerates the cases for $S(x) = \{\text{poly}, \ldots\}$:

<table>
<thead>
<tr>
<th>$S(x)$</th>
<th>$S(p)$</th>
<th>$S(z)$</th>
<th>$S'(z)$</th>
<th>$G, \text{ EDGE}(z \xrightarrow{i} p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{poly, }</td>
<td>{maybe, }</td>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{(M(C))A[p] + \text{none}[ret]}</td>
</tr>
<tr>
<td>{readonly, } &amp; {maybe, } &amp; {poly, } &amp; {poly, }</td>
<td>{(M(C))A[z] + R[z]}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{(M(C))A[z] + R[z]}</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$S(x)$</th>
<th>$S(p)$</th>
<th>$S(z)$</th>
<th>$S'(z)$</th>
<th>$G, \text{ EDGE}(z \xrightarrow{i} p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{polymaybe, }</td>
<td>{maybe, }</td>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{(M(C))A[p] + \text{none}[ret]}</td>
</tr>
<tr>
<td>{readonly, } &amp; {maybe, } &amp; {poly, } &amp; {poly, }</td>
<td>{(M(C))A[z] + R[z]}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{polymaybe, }</td>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{poly, }</td>
<td>{(M(C))A[z] + R[z]}</td>
</tr>
</tbody>
</table>

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

Consider case 2, $s$ is $x = y$. We enumerate all possibilities analogously. Again we omit the cases when $S(x) = \{\text{mutable}, \ldots\}$ as well as the case when $S(y) = \{\text{readonly}, \ldots\}$, as they are trivial.
Now consider case 3, \( s = x.f = y \), and corresponding constraints \( \forall y < \text{maybe} \Rightarrow q_y \). If \( f \) is \textit{readonly}, then \textit{maybe} \( q_y \) is \textit{readonly}, and there is no change in \( S \). By the inductive hypothesis, \( x.f \) being \textit{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 \textit{poly}, then \( y \) becomes \textit{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. \( s = y' = x'.f \). If \( f \) is \textit{readonly}, then there does not exist a path from \( x'.f \) in \( G \). If \( y' \) is not \textit{readonly}, then there exists a path form \( y' \). Thus, \( f \) becomes \textit{poly} in \( S' \), and \( x' < y' \) in \( S' \). In \( G' \), there are new paths from \( x'.f \) and \( x \) reflecting \( S' \).