Dataflow and Type-based Formulations for Reference Immutability

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

Abstract

Reference immutability enforces the property that a reference cannot be used to mutate the referenced object. There are several type-based formulations for reference immutability in the literature. However, we are not aware of a dataflow formulation.

In this paper, we present a dataflow formulation for reference immutability using CFL-reachability, as well as a type-based formulation using viewpoint adaptation, a key concept in ownership types. We observe analogies between the dataflow formulation and the type-based formulation.

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

General Terms Languages, Theory

1. Introduction

Reference immutability enforces the property that the state of an object, including its transitively reachable state, *cannot be mutated through an immutable reference*. Reference immutability is different from object immutability in that the former enforces constraints on references while the latter focuses on the object instance. For instance, in the following code, we cannot mutate the Date object by using the immutable reference rd, but we can mutate the same Date object through the mutable reference md:

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

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

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


FOOL '12 October 22, 2012, Tucson, AZ, USA.

This implementation is not safe because a malicious client can obtain and manipulate the signers array by invoking the getSigners method. A solution is to use reference immutability and annotate the return value of getSigners as readonly. (A readonly array of mutable objects is expressed, following Java 8 syntax [5], as Object readonly [].) As a result, mutations on the array after return will be disallowed:

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

Reasoning about reference immutability has a number of benefits. It improves the expressiveness of interface design by specifying the immutability of parameters and return values; it helps prevent and it helps detect bugs or errors caused by unwanted object mutation; it facilitates reasoning about and proving other properties such as object immutability, method purity and object ownership. The problem has received abundant attention in the literature [1, 7, 12].

Reasoning about reference immutability entails partitioning the references in the program into three categories:

- mutable: A mutable reference can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages.
- readonly: A readonly reference x cannot be used to mutate the referenced object nor anything it references. For example, all of the following are forbidden:
  - x.f = z
  - x.setField(z) where setField sets a field of its receiver
  - y = id(x); y.f = z where id is a function that returns its argument
  - x.f.g = z
  - y = x.f; y.g = z
- polyread: A polyread reference x cannot be used to mutate the object in the scope of the enclosing method m; the object may be mutated after m returns. For example,
  - x.f = y is not allowed, but
  - z = id(y); z.f = 0, where id is polyread X id(polyread X x) { return x; }, and z and y are mutable, is allowed.

  polyread is useful because it allows for context sensitivity. Without polyread, the mutation of z in z = id(y); z.f = 0 would force the formal parameter and return value of id to be mutable. Therefore, if id is called elsewhere, e.g., z1 = id(y1),

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

Figure 1. Syntax. C and D are class names, f is a field name, m is a method name, and x, y, and z are names of local variables, formal parameters, or parameter this. As in the code examples, this is explicit. For simplicity, we assume all names are unique.

where z1 is readonly, the mutable formal parameter will cause y1 to be mutable, even though it is readonly.

Locals, parameters, and return variables in both instance and static methods can be polyread. Fields, either instance or static cannot be polyread. This essentially means that our reference immutability is context-sensitive in the call-transmitted dependences, but is approximate in the structure-transmitted (i.e., object-field-transmitted) dependences. This is necessitated by Reps' undecidability result [11] which states that context-sensitive structure-transmitted data-dependence analysis is undecidable.

Reference immutability can be formulated in several ways. A dataflow formulation focuses on inference of mutable, readonly and polyread references. The goal is to prove as many references as readonly as possible. A type-based formulation provides a type system for reference immutability. It focuses on enforcement of immutability. Programmers specify readonly annotations on certain references and the system either proves the desired immutability or issues an error.

There are several type-based formulations of reference immutability in the literature, most notably Javari by Tschantz and Ernst [12], and more recently ReIm by Huang et al. [7]. However, we are not aware of a dataflow formulation.

In this paper, we present a dataflow formulation of reference immutability using CFL-reachability. We argue the analogy between the dataflow and type-based formulations. In addition, we argue the analogy between context sensitivity in dataflow analysis and *viewpoint adaptation*, a key concept in ownership types [2–4]. The contributions of this paper are:

- A novel dataflow formulation of reference immutability using CFL-reachability.
- An observation on the analogy between context sensitivity in dataflow analysis and viewpoint adaptation.

The rest of the paper is organized as follows. Section 2 formalizes the program syntax. Section 3 describes the dataflow formulation of reference immutability. Section 4 describes the type-based formulation, argues the analogy with the dataflow one and describes the relationship between context sensitivity and viewpoint adaptation. Section 5 concludes the paper and outlines directions for future work.

2. Syntax

We restrict our formal attention to a core calculus in the style of Vaziri et al. [13] whose syntax appears in Figure 1. The language models Java with a syntax in a "named form", where the results of field accesses, method calls, and instantiations are immediately stored in a variable. Without loss of generality, we assume that methods have parameter this, and exactly one other formal parameter. Features not strictly necessary are omitted from the formalism, but they are handled correctly in the implementation. We write $\overline{t\ \mathsf{y}}$ for a sequence of local variable declarations.

For the purposes of the type-based formulation, a type t has two orthogonal components: type qualifier q (which expresses reference immutability) and Java class type C. The immutability type system is *orthogonal* to (i.e., independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone. In the dataflow formulation a type t has only one component, the Java class type C.

3. Dataflow Formulation

We can formulate the problem as a CFL-reachability computation [11] over a directed graph G. The nodes in G are the *references* in the program, and the edges show the dependences between these references. The term *reference* denotes (1) fields, (2) local variables and formal parameters, (3) method return values, and (4) objects, denoted by allocation sites.

The goal of the analysis is to infer mutable, polyread and readonly references maximizing the number of readonly references and minimizing the number of mutable ones. Section 3.1 describes the construction of dependence graph G, Section 3.2 describes the CFL-reachability computation. Section 3.3 and Section 3.4 elaborate on the handling of context sensitivity.

3.1 Dependence Graph

The edges in dependence graph G are constructed according to the rules in Figure 2. Initially, the graph is empty. Each rule takes as input current graph G and adds edges to it to produce graph G'. Note that there is no need to iterate: each rule adds a constant set of edges, regardless of input G. Rule (ASSIGN) adds an edge from y, the right-hand-side of the assignment to x, the left-hand-side of the assignment. The edge expresses a dependence of y on x, i.e., that the mutability of x affects y. If x is mutated in statement $\mathsf{x.f} = \mathsf{z}$, i.e. x is mutable, then y is mutable as well, because x obtained its value through the assignment $\mathsf{x} = \mathsf{y}$. Rule (READ) creates dependences from reference y to left-hand-side x, and from field f to x. The intuition is that a mutation of x affects y, because x refers to parts of the y's object, and x obtained its value through y. The mutation of x affects f as well. Rule (WRITE) adds an edge from y to f.

Rule (CALL) demands a more detailed explanation. Auxiliary function $target(i)$ retrieves the compile-time target m at call site i, namely $\mathsf{this_m}, p \rightarrow \mathsf{ret_m}$. $\mathsf{this_m}$ is m's implicit parameter this, p is m's formal parameter, and $\mathsf{ret_m}$ is m's return variable. For simplicity, we assume no inheritance, that is, there is a single target at each call; again, the general case can be handled easily [1].

Rule (CALL) creates labeled *procedure entry edges* from actual receiver y to parameter $\mathsf{this_m}$, and from actual argument z to formal parameter p. As it is customary in CFL-reachability, the label is an open parenthesis suffixed with the unique label at the call: $(_i$. The rule creates labeled *procedure exit edges* from return variable $\mathsf{ret_m}$ to the left-hand-side x at the call assignment. The label is a close parenthesis suffixed with the label at the call: $)_i$. The rule transmits dependences at method calls. The role of the labels is to transmit mutations only along valid paths and avoid polluting references as mutable when said references are readonly.

Consider the code example below. Recall that our syntax makes the receiver this an explicit parameter.

[1] We note that in Java, when there is no inheritance, method overloading does not lead to multiple targets. In Java, the compile-time target at the call is still decided at compile time, even in the presence of overloading. The run-time target is dispatched at run-time based on the type of the receiver. Under our simplifying assumption, there will be a single run-time receiver type, and therefore a single target.

$$
\begin{array}{lllll}
\text{(NEW)} & j : \mathsf{x} = \mathsf{new\ C}() & \Rightarrow & G' = G \cup j \to \mathsf{x} \\
\text{(ASSIGN)} & \mathsf{x} = \mathsf{y} & \Rightarrow & G' = G \cup \mathsf{y} \to \mathsf{x} \\
\text{(READ)} & \mathsf{x} = \mathsf{y.f} & \Rightarrow & G' = G \cup \mathsf{y} \to \mathsf{x} \cup \mathsf{f} \to \mathsf{x} \\
\text{(WRITE)} & \mathsf{x.f} = \mathsf{y} & \Rightarrow & G' = G \cup \mathsf{y} \to \mathsf{f} \\
\text{(CALL)} & i : \mathsf{x} = \mathsf{y.m(z)} & \Rightarrow & \mathit{let}\ \mathsf{this_m}, p \to \mathsf{ret_m} = \mathit{target}(i)\ \mathit{in} \\
 & & & G' = G \cup \mathsf{y} \xrightarrow{(_i} \mathsf{this_m} \cup \mathsf{z} \xrightarrow{(_i} p \cup \mathsf{ret_m} \xrightarrow{)_i} \mathsf{x}
\end{array}
$$

Figure 2. Rules for construction of G. Rules are defined over the "named form" syntax in Figure 1.

```
class DateCell {
  Date date;
  Date getDate(Date this) { return this.date; }
  void m1(Date this) {
    1: Date md = this.getDate();
    2: md.setHours(1); // md is mutated
  }
  void m2(Date this) {
    3: Date rd = this.getDate();
    4: int hour = rd.getHours();
  }
}
```

The dependence graph for this example is as follows:

Consider statement return this.date; as an example. It is treated as an assignment ret = this.date, and results in two edges: $\mathsf{this_{getDate}} \to \mathsf{ret_{getDate}}$ and $\mathsf{date} \to \mathsf{ret_{getDate}}$ (we denote reference variables by their name suffixed with the name of their enclosing method). Also, the call at 1 results in labeled procedure entry edge $\mathsf{this_{m1}} \xrightarrow{(_1} \mathsf{this_{getDate}}$ and labeled procedure exit edge $\mathsf{ret_{getDate}} \xrightarrow{)_1} \mathsf{md}$. The labeled edges are dashed in the graph.

We use the graph to propagate direct mutations, backwards, towards affected references. In the above example, only $\mathsf{this_{setHours}}$ is mutated directly (the mutation is not shown in the code). $\mathsf{this_{setHours}}$ is shown in red in the graph.

The mutation of $\mathsf{this_{setHours}}$ makes md mutable. The mutation of md is transmitted via call

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

back to $\mathsf{this_{m1}}$. However, it should not be transmitted to $\mathsf{this_{m2}}$, because the path from $\mathsf{this_{m2}}$ to md is not a valid path as we shall explain shortly in Section 3.2.

As a more involved example, consider the code in Figure 3 and its corresponding dependence graph in Figure 4. For the rest of this paper, we will use this code and its graph as a running example.

3.2 Reachability Computation

This section describes the CFL-reachability computation.

A path $\mathsf{x} \to^* \mathsf{y} \in G$ is a *same-level path* from x to y if all procedure exits *match* the corresponding procedure entries. More formally, path $\mathsf{x} \to^* \mathsf{y}$ is a same-level path the labels on its edges form a well-formed string in the language defined by the following context-free grammar:

$$SLP \to (_i\ SLP\ )_i \mid SLP\ SLP \mid \epsilon$$

For example, path $\mathsf{this_{get}} \xrightarrow{(_1} \mathsf{this_{getX}} \to \mathsf{x_{getX}} \to \mathsf{ret_{getX}} \xrightarrow{)_1} \mathsf{x_{get}}$ is a same-level path from $\mathsf{this_{get}}$ to $\mathsf{x_{get}}$.

```
class A {
  X f;
  X get(Y y) {
    ... = y.h;
    1: X x = this.getX();
    return x;
  }
  X getX() {
    X x = this.f;
    return x;
  }
}
```

```
void m1() {
  A a = ...
  Y y = ...
  2: X x = a.get(y);
  x.g = null;
}
```

```
void m2() {
  A a = ...
  Y y = ...
  3: X x = a.get(y);
  ... = x.g;
}
```

Figure 3. Example program

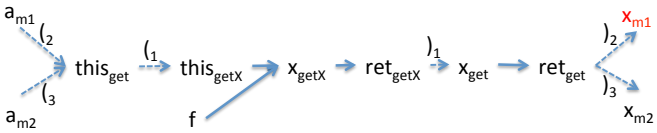

Figure 4. Dependence graph for example program. Labeled edges are dashed. Directly mutated variable x in m1 is shown in red.

A path $\mathsf{x} \to^* \mathsf{y}$ in G is a *call path* from x to y if all the procedure exits *match* the procedure entries but it is possible that some procedures are entered and not yet exited. The following grammar defines same-level paths:

$$CP \to (_i\ SLP \mid CP\ CP \mid SLP$$

For example, path $\mathsf{a_{m1}} \xrightarrow{(_2} \mathsf{this_{get}} \xrightarrow{(_1} \mathsf{this_{getX}} \to \mathsf{x_{getX}} \to \mathsf{ret_{getX}} \xrightarrow{)_1} \mathsf{x_{get}}$ is a call path. Note that a same-level path is also a call path.

Similarly, a path $\mathsf{x} \to^* \mathsf{z}$ in G is a *return path* from x to z if procedure exits *match* the procedure entries but there are at least some procedure exits whose corresponding entries are not on the path. The grammar that describes return paths is as follows:

$$RP \to SLP\ )_i\ SLP \mid RP\ RP$$

For example, path $\mathsf{this_{get}} \xrightarrow{(_1} \mathsf{this_{getX}} \to \mathsf{x_{getX}} \to \mathsf{ret_{getX}} \xrightarrow{)_1} \mathsf{x_{get}} \to \mathsf{ret_{get}} \xrightarrow{)_2} \mathsf{x_{m1}}$ is a return path from $\mathsf{this_{get}}$ to $\mathsf{x_{m1}}$. Note that a same-level path is not a return path.

We require that nodes on same-level, call and return paths *cannot be fields*. Dependences transmitted through fields are special and will be explained shortly.

The computation of reference immutability is as follows:

- Reference x in $\mathsf{x.f} = \mathsf{y}$ is marked mutable. Clearly, if x is the receiver in a field write $\mathsf{x.f} = \mathsf{y}$, then x must be mutable.
- Reference x is marked mutable if there exists a call path $\mathsf{x} \to^* \mathsf{y}$ in G such that y is marked mutable. In our running example (Figure 3 and Figure 4), $\mathsf{a_{m1}}$ is mutable because there is a call path from $\mathsf{a_{m1}}$ to $\mathsf{x_{m1}}$ and $\mathsf{x_{m1}}$ is mutable. Note that in this case, the call path is a same-level path. Reference $\mathsf{a_{m2}}$ is not mutable however, because there is no call path to the mutable

x_{m1}; there is a path of course, but it is not a valid call path because procedure entries and exits do not match.

- Reference x is marked polyread if there exist a return path $x \rightarrow^* z$ in G such that z is mutable. A reference can be marked as both mutable and polyread. A reference variable is marked as polyread when a mutation is reached after the return of the variable's enclosing method. For example, x_{getX} is polyread because x_{m1}, which is mutated, is reachable on a return path. Intuitively, x_{getX} is polymorphic. It is not mutated in its enclosing method getX. However, the object it refers to is mutated in one of the contexts of invocation of getX, after the return of getX; the object is not mutated in the other context.
- Field f is marked mutable if there is an edge $f \rightarrow y$ in G such that y is marked mutable or polyread. A field f is mutable if it is assigned to a mutable or polyread reference. As mentioned in the introduction, a field cannot be polyread. We elaborate on this restriction shortly.
- Reference x is marked mutable if there is an edge $x \rightarrow f$ in G such that field f is marked mutable. This rule marks a reference as mutable, if it is assigned to a mutable field.

The computation applies the above rules repeatedly, until it reaches a fixpoint — that is, no more references are marked mutable or polyread. At the end, references marked as mutable are inferred as mutable. References marked as polyread but not mutable, are inferred as polyread. The remaining references are inferred as readonly.

The final result in our running example is the following:
mutable: a_{m1}, x_{m1} and f.
polyread: $this_{get}$, ret_{get}, x_{get}, $this_{getX}$, ret_{getX}, x_{getX}.
readonly: a_{m2} and x_{m2}.

3.3 Call-transmitted Dependences

At this point, readers have noticed that our analysis is context-sensitive in the call-transmitted dependences. Clearly, it follows only valid call and return paths (i.e., paths with matching procedure entry and procedure exit edges). As mentioned earlier, the analysis does not propagate the mutation at x_{m1} back to a_{m2} because the path from a_{m2} to x_{m1} is not a valid call path. As a result, a_{m2} can be proven readonly.

3.4 Structure-transmitted Dependences

Structure-transmitted dependences are dependences that arise due to flow through object fields. Readers have likely noticed that our analysis is *approximate* in the structure-transmitted dependences. It merges the mutability of fields across all objects (recall that fields are either mutable or readonly but not polyread). In other words, the analysis handles imprecisely the case when there are two different objects of the same class, where one object has a mutable f field, but the other object has a readonly f field. Consider the example:

```
x = new C();    [i]
x.f = new D();
y = x.f;
y.g = 0;
...
x2 = new C();   [j]
x2.f = y2;
```

The mutation of y will be propagated through f to y2 and y2 will be inferred as mutable even though it is not mutated. This is because the analysis cannot distinguish that the x in $y = x.f$ and the x2 in $x2.f = y2$ refer to two distinct objects, i and j, and therefore the mutation of y cannot affect y2 at runtime.

The approximation is necessitated by Reps' undecidability result, which states that context-sensitive structure-transmitted data-dependence analysis is undecidable [11]. Thus, analysis designers must approximate in at least one of the two dimensions, at calls or at fields, and there is a wide variety of ways to approximate. In the analysis above, we use a straightforward approximation where every object (i.e., structure) is abstracted by its class. A more precise approximation is to abstract an object by its allocation site, an even more precise one is to use a combination of the object's allocation site and the allocation site of its creating object, and so on.

As an example, if objects are distinguished by their allocation sites, there we will use i to abstract the first Cobject and j to abstract the second. When constructing graph G we will create an edge $i.f \rightarrow y$ at field read $y = x.f$ and an edge $y2 \rightarrow j.f$ at field write $x2.f = y2$. Thus, the write of y will not propagate to y2.

4. Type-based Formulation

The type-based formulation of reference immutability uses the same type qualifiers with the same meaning. As with the dataflow formulation polyread cannot be applied to fields. The subtyping relation between the qualifiers is

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

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

In previous work we presented a type system for reference immutability called Relm [7]. The type system presented in this paper, which we call Relm′, differs slightly from Relm. Relm′ better illustrates the analogy between the dataflow formulation and the type based formulation. We will elaborate on the differences shortly. Relm′ is not a contribution over Relm.

4.1 Viewpoint Adaptation

Viewpoint adaptation is a concept from Universe types [3, 4], which applies to other ownership and ownership-like type systems as well [2, 13]. For example, the type of x.f is not just the declared type of field f — it is the type of f adapted from the point of view of x. For example, in Universe types, rep x denotes that the current this object is the owner of the object o_x referenced by x. If field f has type peer, this means that the object o_x and the object o_f referenced by field f have the same owner. Thus, the type of x.f, or the type of f adapted from the point of view of x, is rep — the object o_f's owner is the current this object as well.

Ownership type systems make use of a *single* viewpoint adaptation operation. This viewpoint adaptation operation is performed at both field accesses and method calls. It is written $q \triangleright q'$, which denotes that type q' is adapted from the point of view of type q to the viewpoint of the current object this. Viewpoint adaptation adapts the type of a field, formal parameter, or return type, from the viewpoint of the *receiver* at the corresponding field access or method call to the viewpoint of the current object this. In other words, the context of adaptation at both field access and method call, is the receiver object.

One key point of this paper is to illustrate and explore the interesting relationship between context sensitivity in dataflow analysis and viewpoint adaptation. We argue that the role of viewpoint adaptation is to transmit dependences at structures (by adapting fields), and at calls (by adapting formal parameters and return types).

In this spirit, we propose a generalization of traditional viewpoint adaptation. First, we allow for two different viewpoint adaptation operations, one applied at fields, and the other one applied at calls. Effectively, this separates the handling of dependences at fields, from the handling of dependences at calls. Second, we allow for

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}()}\ (\textsc{tnew})$$

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

$$\frac{\begin{array}{c}\Gamma(\mathsf{x}) = \mathsf{mutable} \quad typeof(\mathsf{f}) = q_{\mathsf{f}} \\ \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: \mathsf{mutable} \rhd_f q_{\mathsf{f}}\end{array}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\textsc{twrite})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad typeof(\mathsf{f}) = q_{\mathsf{f}} \\ \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{y}} \rhd_f q_{\mathsf{f}} <: q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad typeof(\mathsf{m}) = q_{\mathsf{this_m}}, q \to q_{\mathsf{ret_m}} \\ \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_m q_{\mathsf{this_m}} \quad q_{\mathsf{z}} <: q_{\mathsf{x}} \rhd_m q \\ q_{\mathsf{x}} \rhd_m q_{\mathsf{ret_m}} <: q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ (\textsc{tcall})$$

Figure 5. Typing rules. Function *typeof* retrieves the immutability types of fields and methods. Γ is a type environment that maps references to immutability qualifiers.

adaptation from different viewpoints, not only from the viewpoint of the receiver. This allows for different kinds of context sensitivity.

We now return to reference immutability and explain the viewpoint adaptation that it needs.

Viewpoint adaptation operation $q \rhd_f q_{\mathsf{f}}$ is applied at field accesses. It adapts field type q_{f} from the point of view of receiver type q. We define $\rhd_f$ for field access:

$$\begin{array}{rcl} \_ \rhd_f \mathsf{readonly} & = & \mathsf{readonly} \\ q \rhd_f \mathsf{mutable} & = & q \end{array}$$

The underscore denotes a "don't care" value. Consider field access y.f. If the type of receiver y is readonly and the type of field f is mutable, then the type of y.f is readonly $\rhd_f$ mutable = readonly. A field access y.f is mutable if and only if both the receiver y and field f are mutable. If the receiver or the field is readonly, y.f is readonly. It is important to note that the adapted type at y.f is the least upper bound of the types of y and f.

Viewpoint adaptation operation $q_{\mathsf{x}} \rhd_m q$ is applied at method calls $\mathsf{x} = \mathsf{y.m(z)}$. It adapts q, the type of a formal parameter/return value of m, from the point of view of q_{x}, the *context* of the call. $\rhd_m$ is defined as follows:

$$\begin{array}{rcl} \_ \rhd_m \mathsf{mutable} & = & \mathsf{mutable} \\ \_ \rhd_m \mathsf{readonly} & = & \mathsf{readonly} \\ q \rhd_m \mathsf{polyread} & = & q \end{array}$$

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

4.2 Typing Rules

The typing rules are presented in Figure 5. Rule (TASSIGN) is straightforward. They require that the left-hand-side is a supertype of the right-hand-side. Observe the analogy with the dataflow formulation: rule (ASSIGN) creates edge $\mathsf{y} \to \mathsf{x}$ in G. More generally, we conjecture that we have $\mathsf{y} <: \mathsf{x}$, including transitive subtyping, if and only if there is a same-level path from y to x in G.

Rule (TWRITE) requires $\Gamma(\mathsf{x})$ to be mutable because x's field is updated in the statement. The viewpoint adaptation operation for field access is used in both (TWRITE) and (TREAD). Intuitively, $\rhd_f$ combined with rules (TWRITE) and (TREAD) handles structure-transmitted dependences, in the same fashion, as the edges through fields f in dependence graph G do. Consider a field write $\mathsf{x.f} = \mathsf{y}$ and a field read $\mathsf{z} = \mathsf{w.f}$. Rule (TWRITE) enforces $q_{\mathsf{y}} <: q_{\mathsf{f}}$ and (TREAD) enforces $q_{\mathsf{f}} <: q_{\mathsf{z}}$. Thus, a mutation on z will force f to be mutable and this in turn will force y to be mutable. This is analogous to the dataflow formulation in Section 3. $\mathsf{x.f} = \mathsf{y}$ results in edge $\mathsf{y} \to \mathsf{f}$ in G and $\mathsf{z} = \mathsf{w.f}$ results in edge $\mathsf{f} \to \mathsf{z}$. A mutation on z forces f to be mutable, and the mutable f forces y to be mutable as well. The handling of structure-transmitted dependences in the type-based formulation is analogous to the handling in the dataflow formulation.

Rule (TCALL) handles calls and demands detailed explanation. This rule, along with $\rhd_m$ handles call-transmitted dependences. Function *typeof* retrieves the type of m. q_{this} is the type of implicit parameter this, q is the type of the formal parameter, and q_{ret} is the type of the return value. Rule (TCALL) requires $q_{\mathsf{x}} \rhd_{\mathsf{m}} q_{\mathsf{ret}} <: q_{\mathsf{x}}$. This constraint disallows the return value of m from being readonly when there is a call to m, $\mathsf{x} = \mathsf{y.m(z)}$, where left-hand-side x of the assignment is mutable. Only if the left-hand-sides of all call assignments to m are readonly, can the return type of m be readonly; otherwise, it is polyread. A programmer can annotate the return type of m as mutable. However, this typing is pointless, because it unnecessarily forces local variables and parameters in m to become mutable when they can be polyread.

In addition, the rule requires $q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd_{\mathsf{m}} q_{\mathsf{this}}$. When q_{this} is readonly or mutable, its adapted value is the same according to the adaptation rules of $\rhd_m$. Thus, when q_{this} is mutable (due to this.f = 0 in m, for example),

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

which disallows q_{y} from being anything but mutable, as expected. In Section 3 this is handled by call paths. In the case described above, there is a call path $\mathsf{y} \xrightarrow{(_i}$ this which forces y to be mutable.

The most interesting case arises when q_{this} is polyread. A polyread parameter this is readonly within the enclosing method, but there could be a dependence between this and ret such as

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

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

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

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

The role of viewpoint adaptation is to transfer the dependence between this and ret in m, into a dependence between actual receiver y and left-hand-side x in the call assignment.

In the above example, there is a dependence between this and the return ret. Thus, we also have a dependence between y and x in the call $\mathsf{x} = \mathsf{y.m()}$ — that is, a mutation of x makes y mutable as well. Function $\rhd_m$ does exactly that. Rule (TCALL) requires

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

When there is a dependence between this and ret, q_{this} is polyread, and the above constraint becomes

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

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

Note the analogy with the analysis in Section 3. In the example

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

there is a same-level path between this and ret. Thus, call x = y.m() generates a same-level path from y to x (the entry and exit parentheses will balance out) and a mutation of x propagates to y along this path. Again, as with structure-transmitted dependences, the handling of call-transmitted dependences in the type-based formulation is analogous to the handling in the dataflow formulation. Viewpoint adaptation helps achieve the desired behavior.

The typed DateCell class from Section 3 is as follows.

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

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

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

It is instantiated to readonly in m2:

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

Thus, $\mathsf{this}_{\mathsf{m2}}$ can be typed readonly.

We conclude this section with a brief discussion. Allowing for adaptation from *different viewpoints*, not only from the point of view of the receiver, enables different kinds of context sensitivity. For example, adapting from the viewpoint of the receiver, as it is customary in ownership type systems, can be interpreted as *object sensitivity* [9]. Adapting from the viewpoint of the context of invocation, as it is necessary for reference immutability, can be interpreted as *call-site context sensitivity*.

Differentiation of viewpoint adaptation at fields from viewpoint adaptation at methods allows us to implement the handling of structure-transmitted dependences *differently* from the handling of the call-transmitted dependences. For reference immutability, we handled transmission through fields approximately by merging flow through a field across all objects. We handled transmission through calls precisely, by matching calls and returns. We envision further generalization, where one can implement other, more interesting approximations.

The difference between Relm′ and Relm is that Relm allows only readonly and polyread fields, where a readonly field has the same semantics as a readonly field in Relm′, and a polyread field has exactly the same semantics as a mutable field in Relm′. Relm uses a single viewpoint adaptation operation applied at both field accesses and method calls:

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

instead of the two different operations in Relm′. Relm treats context as in Relm′, the context of adaptation at field access is the receiver, and at method calls is the left-hand-side of the call assignment.

We chose to use separate the viewpoint adaptation operations in Relm′ in order to emphasize the analogy with dataflow analysis (even though refenrece immutability can be formulated using a single operation as in Relm). Separate operations differentiate the handling of dependences at field access from dependences at method calls. Thus, we used two separate operations: $\rhd_{\mathsf{f}}$ is used to handle transmission of dependences at field access, and $\rhd_{\mathsf{m}}$ is used to handle transmission at calls. These operations can be instantiated in different ways in order to accommodate different approximations in data transmission. In the future, we plan to investigate different kinds of approximations and their handling using viewpoint adaptation.

4.3 Type Inference

In previous work [6, 7] we propose an inference algorithm, which infers the "best" (i.e., most desirable) typing for reference immutability. Roughly, this is the typing with a maximal number of readonly references and a minimal number of mutable references. We conjecture that when there are no programmer-provided annotations, this "best" typing is equivalent to the inference result obtained by the dataflow formulation in Section 3.

The analogy between the dataflow formulation and the type-based formulation is interesting because we can use dataflow (CFL-reachability) machinery to study and solve the type inference problem, or conversely, we can use type inference to solve the dataflow problem. The complexity bound of CFL-reachability in the general case is $O(|N \cup T|^3 n^3)$, where N is the set of nonterminals in the CFG grammar, T is the set of terminals in the grammar and n is the number of nodes in G, or roughly the number of reference variables in the program. For the special case of context-free language that we consider here, known as *Dyck language*, the bound can be improved to $O(kn^3)$ [8] where k is the number of different kinds of parentheses, or roughly, the number of call sites in the program. Thus, the complexity of the CFL-reachability formulation for reference immutability is $O(S^4)$ where S is the size of the program. Interestingly, the complexity of type inference for the type-based formulation is $O(S^2)$. Our hope is that other dataflow analyses can be found analogous to type inference, and therefore, type inference machinery can solve these problems.

5. Conclusions and Future Work

We have described a dataflow formulation of reference immutability using CFL-reachability, and an analogy between this dataflow formulation and a type-based formulation. We believe that the analogy between context sensitivity in dataflow analysis, and viewpoint adaptation in ownership types is a promising direction of future research.

In future work, we will continue to study the relationship between context sensitivity and viewpoint adaptation and more generally, the relationship between context-sensitive dataflow analysis (e.g., slicing, points-to analysis) and context-sensitive type systems such as Relm, Universe Types, Ownership types, and others. We conjecture that problems in dataflow analysis (e.g., points-to analysis) can be formulated as type-based analysis and solved using type inference. We plan to explore this direction in the future.

Acknowledgments

We thank the anonymous FOOL reviewers for their valuable comments on this paper.

References

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

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

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

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

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

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

[7] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm and ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, 2012.

[8] J. Kodumal and A. Aiken. The set constraint/cfl reachability connection in practice. In *PLDI*, pages 207–218, 2004.

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

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

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

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

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