FlowCFL: Generalized Type-Based Reachability Analysis

Graph Reduction and Equivalence of CFL-Based and Type-Based Reachability

ANA MILANOVA, Rensselaer Polytechnic Institute, USA

Reachability analysis is a fundamental program analysis with a wide variety of applications. We present FlowCFL, a type-based reachability analysis that accounts for mutable heap data. The underlying semantics of FlowCFL is Context-Free-Language (CFL)-reachability.

We make three contributions. First, we define a dynamic semantics that captures the notion of flow commonly used in reachability analysis. Second, we establish correctness of CFL-reachability over graphs with *inverse* edges (inverse edges are necessary for the handling of mutable heap data). Our approach combines CFL-reachability with *reference immutability* to avoid the addition of certain inverse edges, which results in graph reduction and precision improvement. The key contribution of our work is the formal account of correctness, which extends to the case when inverse edges are removed. Third, we present a type-based reachability analysis and establish equivalence between a certain CFL-reachability analysis and the type-based analysis, thus proving correctness of the type-based analysis.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; • **Software and its engineering** → **Software notations and tools**.

Additional Key Words and Phrases: CFL-reachability, reference immutability, type-based analysis

ACM Reference Format:
Ana Milanova. 2020. FlowCFL: Generalized Type-Based Reachability Analysis: Graph Reduction and Equivalence of CFL-Based and Type-Based Reachability. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 178 (November 2020), 29 pages. https://doi.org/10.1145/3428246

1 INTRODUCTION

Reachability analysis detects flow from *sources* to *sinks*. It is a fundamental program analysis technique with a wide variety of applications. One prominent application is taint analysis for Android, which detects flow from sensitive sources, such as phone and location data, to untrusted sinks, such as the Internet [Arzt et al. 2014; Ernst et al. 2014; Huang et al. 2015].

In this paper, we study FlowCFL, a type-based reachability analysis. FlowCFL supports two basic type qualifiers, pos (positive) and neg (negative). It permits flow from neg variables to pos ones, but forbids flow from pos variables to neg ones. Typically, the programmer specifies a set of pos variables (i.e., sources) and a set of neg variables (i.e., sinks) and FlowCFL decides whether there is flow from a pos variable to a neg one. FlowCFL is defined in terms of standard typing rules, however, the underlying semantics of flow that these rules entail is the classical Context-Free-Language(CFL)-reachability [Reps 1998, 2000; Reps et al. 1995].

Standard CFL-reachability analysis has two phases. First, it constructs a graph that represents flow of values from one variable to another; edges are annotated with *call* and *return* annotations to

Author's address: Ana Milanova, Rensselaer Polytechnic Institute, 110 8th Street, Troy, New York, 12180, USA, milanova@cs.rpi.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART178
https://doi.org/10.1145/3428246

model call-transmitted dependences and with field *write* and field *read* annotations to model heap-transmitted dependences. Next, the analysis searches for paths with properly matched call/return and write/read annotations. CFL-reachability analysis is a highly precise flow analysis. It has a long history [Reps 2000; Reps et al. 1995] and it is still actively studied and actively used in program analysis; [Chatterjee et al. 2018; Lu and Xue 2019; Späth et al. 2019; Xu et al. 2009; Zhang and Su 2017] are recent works among many other works. An important concept in CFL-reachability analysis is the concept of the *inverse edge* [Sridharan and Bodík 2006; Sridharan et al. 2005], which is necessary for the handling of mutable heap data. At assignments, e.g., at x = y, the standard analysis adds the expected *forward edge* from y to x that represents flow from y to x, however, it also adds an *inverse edge* from x to y thus constructing a *bidirectional CFL-reachability graph* G_{BI}. The concept of the inverse edge, which becomes more involved once we consider call/return and write/read annotated edges and paths, has not been formalized. The question *"Does CFL-reachability over G_{BI} capture all program flows?"* has not been answered formally. We consider an answer to this question.

As both CFL-reachability analysis and type-based reachability analyses are widely used techniques, it is important to clarify our contribution. First, we formalize the *notion of flow* in terms of a *novel dynamic semantics* and we use the semantics to construct a correctness argument that answers the above question. The semantics accounts for inverse edges, which present the key challenge to the formalization.

Additionally, we consider a graph that avoids adding certain inverse edges based on knowledge of reference immutability. We call this graph G_{RI}. Given an *immutable reference* x, there is no need to add inverse paths that originate at x. There is substantial precision improvement for reachability over G_{RI} compared to reachability over G_{BI} as demonstrated in earlier work [Milanova and Huang 2013; Zhang and Su 2017] and confirmed by experiments we run for this paper. Our formal treatment extends to this case. We answer the following question as well: *"Does CFL-reachability over G_{RI} capture all program flows?"*. We note that the idea of using reference immutability to improve reachability analysis dates back to prior work [Huang et al. 2015; Milanova and Huang 2013; Milanova et al. 2014]. However, prior treatment is ad-hoc and lacks a formal account. A key novelty of this work is that we *establish correctness* of reachability over G_{RI}. Therefore, future CFL-reachability-based analyses can use G_{RI} instead of the standard G_{BI}; the graph reduction that G_{RI} entails may lead to precision improvement in the analysis.

Returning to FlowCFL, the type-based analysis uses the pos, neg, and poly type qualifiers and a set of typing rules to model reachability. Although not as widespread as CFL-reachability, type-based reachability analysis has been used in existing works, e.g., [Dong et al. 2016; Huang et al. 2015; Sampson et al. 2011; Shankar et al. 2001]. Our motivation for the study of FlowCFL and its connection to CFL-reachability stems from our previous work on DroidInfer, a type-based taint analysis for Android [Huang et al. 2015]. DroidInfer reports hundreds of leaks in real-world Android apps. By virtue of being a type-based analysis, it reports leaks as type errors, but it remains non-trivial to connect a type error to a source-sink path. In DroidInfer, we draw on intuition of a connection between type-based analysis and CFL-reachability and explain type errors in terms of CFL-reachability paths. However, the connection is not formalized and it is not well understood. FlowCFL, a generalization of DroidInfer, models reachability problems in different domains, including taint analysis, approximate computing, and secure computation. In this work, we establish equivalence between a certain CFL-reachability analysis over G_{RI} and a certain type-based reachability analysis, specifically DroidInfer's analysis, thus establishing correctness of the type-based analysis. We believe that the formal account can advance application of type-based reachability in different problem domains and help explain type errors in terms of CFL-reachability paths.

In summary, this paper makes the following contributions:

- We present a dynamic semantics that formalizes the notion of *flow* commonly used in CFL-reachability and type-based reachability.
- We prove that CFL-reachability over G_{BI} captures all run-time flows. Our treatment extends to reachability over G_{RI}, which avoids adding certain edges based on knowledge of reference immutability. We present experiments that show substantial precision improvement in taint analysis for Android over G_{RI} compared to analysis over G_{BI}. Our experiments are in line with earlier work that has shown the importance of reducing the number of inverse edges [Milanova and Huang 2013; Zhang and Su 2017].
- We establish equivalence between a type-based reachability analysis and a CFL-reachability analysis, thus proving correctness of the type-based analysis.

The rest of the paper is organized as follows. Sect. 2 presents an overview of FlowCFL and briefly discusses applications of FlowCFL. Sect. 3 presents the dynamic semantics of flows. Sect. 4 presents CFL-reachability, G_{BI}, reference immutability, and the construction of G_{RI}. Sect. 5 details the correctness argument. Sect. 6 presents the type-based analysis, and Sect. 7 establishes equivalence between the type-based and CFL-based analyses. Sect. 8 discusses related work and Sect. 9 concludes.

2 OVERVIEW AND APPLICATIONS

2.1 Overview of FlowCFL

In a typical setting, reachability analysis reasons about flow from *sources* to *sinks*. FlowCFL assigns type *qualifier* to variables and fields. There are two basic qualifiers: pos, which denotes sources, and neg, which denotes sinks. We have

$$\mathsf{neg} <: \mathsf{pos}$$

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2. (q is also a subtype of itself $q <: q$.) Therefore, it is allowed to assign a neg variable to a pos one, i.e., a neg variable can flow to a pos one:

```
neg String n = ...;
pos String p = n;
```

However, it is not allowed to assign a pos variable to a neg one, i.e., a pos variable cannot flow to a neg one:

```
pos String p = ...;
neg String n = p; // error!
```

Note that this is the natural subtyping. Such subtyping is unsafe in the presence of mutable references [Bank et al. 1997; Sampson et al. 2011] and systems use *equality*, which is akin to the inverse edges in CFL-reachability. FlowCFL leverages reference immutability (e.g., ReIm [Huang et al. 2012b], Javari [Tschantz and Ernst 2005]) to allow for safe but limited subtyping.

The programmer specifies the sources and/or sinks and FlowCFL *infers* qualifiers for the rest of the variables. Roughly, if a source flows to a variable x, then x is pos; if a variable y flows to a sink, then y is neg. If inference fails, i.e., reports *error(s)*, then there may be a leak from a source to a sink. Otherwise, it is guaranteed that there is no flow from a source to a sink.

FlowCFL is context-sensitive (i.e., polymorphic) as illustrated by the following example. We elaborate on context sensitivity in Sects. 4-6.

```
1  poly String id(poly String p) {
2      return p;
3  }
4  pos String source = ...;
5  pos String x = id(source);

7  neg String y = ...;
8  neg String sink = id(y);
```

In the above example, the identity function id is context-sensitive. id is interpreted as pos in line 5 and it is interpreted as neg in line 8. FlowCFL precisely propagates source to x but not to sink; it propagates sink back to y but not to source. A context-insensitive system rejects the program as it merges flow through id and imprecisely decides that there is flow from source to sink.

From a practical point of view, FlowCFL supports two different settings of the problem and the user can configure the system to one of these settings. In the *negative setting* FlowCFL expects that the user provide a set of sinks, i.e., negative annotations and it propagates those sinks backwards, i.e., against the direction of the flow. Unaffected variables remain positive. The more precise the analysis, the fewer variables become neg and a larger number of variables remain pos. In the *positive setting*, FlowCFL expects that the user provide a set of sources, i.e., positive annotations and it propagates those sources forward. In either setting, the user can annotate both sources and sinks, though sources are optional in the negative setting and sinks are optional in the positive setting. The well-known taint analysis problem, which entails annotations on both sources and sinks, can be cast in either of the settings. FlowCFL can run in either the negative or positive setting; if it detects a conflict, i.e., a variable is annotated pos but is inferred neg (or in the positive setting, it is annotated neg but is inferred pos), it reports an error.

FlowCFL has a fixed semantics of flow (i.e., propagation) that corresponds to a certain approximation of CFL-reachability. Sect. 6 and Sect. 7 define the precise semantics and correspondence to CFL-reachability. In future work we plan to extend configurability to support different approximations of CFL-reachability.

2.2 Applications

Many analyses can be cast as instances of FlowCFL. The user casts a system/analysis into FlowCFL by mapping the "native" positive and negative qualifiers to FlowCFL's pos and neg respectively. They set the setting, annotate sources and/or sinks, and FlowCFL does the inference automatically. Below we describe two analyses, one from the domain of taint analysis, and another from the domain of approximate computing.

2.2.1 Taint Analysis for Android. DroidInfer [Huang et al. 2015] is the prototypical instance of FlowCFL. The user annotates a set of sensitive sources (e.g., phone data, location data) and a set of untrusted sinks (e.g., the Internet, Sms texts). As mentioned earlier, taint analysis can be done in either setting, and we pick the negative setting.

Fig. 1 illustrates the analysis. getSimSerialNumber in line 5 in FieldSensitivity2 retrieves sensitive telephony information and its return value is a source. The parameter of sendTextMessage in line 9 is a sink. There is flow from source sim to sink sg through the Data container and FlowCFL reports an error. We note that there are no annotations in app code, all user annotations appear in the Android SDK; we have annotated sim and sg purely for illustration purpose.

```
1  public class Data {
2    String secret;
3    void set(String p) {
4      this.secret = p;
5    }
6    String get() {
7      return this.secret;
8    } l
9  }
```

```
1  public class FieldSensitivity2 extends Activity {
2    protected void onCreate(Bundle b) {
3      Data dt = new Data(); the
4      TelephonyManager tm = (TelephonyManager)
             getSystemService("phone");
5      pos String sim = tm.getSimSerialNumber();
6      dt.set(sim);
7      SmsManager sms = SmsManager.getDefault();
8      neg String sg = dt.get();
9      sms.sendTextMessage("+123",null,sg,null,null);
10   }
11 }
```

Fig. 1. FieldSensitivity2 is rephrased from DroidBench [Arzt et al. 2014; Fritz et al. 2013].

```
1  class Newton {
2    static float tolerance = 0.00001;
3    static int maxsteps = 40;
4    static float F(float x) { ... }
5    static float dF(float x) { ... }

7    static float newton(urel float xs) {
8      float x, xprim;
9      float t1, t2;
10     int count = 0;
11     x = xs;
12     xprim = xs + 2*tolerance;
13     while ((x − xprim >= tolerance) || (x − xprim <= −tolerance)) {
14       xprim = x;
15       t1 = F(x);
16       t2 = dF(x);
17       x = x − t1 / t2;
18       if (count++ > maxsteps) break;
19     }
20     if (!((x − xprim <= tolerance) && (x − xprim >= −tolerance))) {
21       x = INFTY;
22     }
23     return x;
24   }
25 }
```

Fig. 2. Newton's method from [Carbin et al. 2013a].

In summary, FlowCFL decides that there is flow from positive sim, the device SIM serial number (SSN), to negative sg, the body of the text message. The analysis determines that sim is neg, which clashes with the designation of sim as source and sim's pos type.

2.2.2 Annotations for Rely. Rely [Carbin et al. 2013a] is a system that reasons about execution on unreliable hardware. Programmers explicitly annotate all unreliable variables (using the urel annotation), as well as all operations on unreliable variables (e.g., unreliable + becomes +.). Unannotated variables and operations are considered reliable. Rely verifies a bound on the reliability of computation with respect to the reliability of its input. The annotation system of Rely can be cast as an instance of FlowCFL in the positive setting. (Of course, the proof system of Rely is not an instance of FlowCFL, the purpose of FlowCFL here is to minimize the annotation burden on the programmer.) The urel (unreliable) Rely annotation maps to pos, and the default reliable annotation maps to neg. Programmers annotate unreliable inputs with pos and FlowCFL infers types for the rest of the program, thus minimizing the unreliable partition.

Fig. 2 illustrates inference for Rely. Input xs in line 7 is annotated unreliable (urel corresponds to pos in FlowCFL). FlowCFL fills in the remaining annotations. It infers that F is poly float F(poly float x) and so is dF. Variables x, xprim, t1 and t2 are inferred urel (as explicitly annotated in [Carbin et al. 2013a]). All operations, except for line 19, are unreliable (as in [Carbin et al. 2013a]). FlowCFL infers types for the programs in [Carbin et al. 2013a,b] exactly as annotated in [Carbin et al. 2013a,b].

2.2.3 Other Instantiations. In our technical report [Milanova 2020] we cast EnerJ [Sampson et al. 2011], a classical system from the domain of approximate computing, and JCrypt [Dong et al. 2016], a system for secure computation, as instances of FlowCFL. The variety of applications motivates our study of FlowCFL and its connection to CFL-reachability.

2.3 Overview of CFL-Reachability

FlowCFL is a type-based analysis but its underlying semantics is the classical CFL-reachability analysis [Reps 1998, 2000; Reps et al. 1995]. CFL-reachability analysis proceeds in two phases. First, it constructs a flow graph representation of the program. Second, it reasons about reachability over the graph. Throughout the paper, we will work with the example in Fig. 3, which is a rephrase of the FieldSensitivity example in Fig. 1. CFL-reachability constructs the graph shown below the code. Annotations $(_i$ and $)_i$ model call-transmitted dependences. For example, edge $\mathsf{a} \xrightarrow{(_6} \mathsf{p}$ represents that at call site 6 in main a flows to parameter p of set and edge $\mathsf{ret} \xrightarrow{)_7} \mathsf{b}$ represents that at call site 7 ret of get flows to b. Annotations w_f and r_f model heap-transmitted dependences. $\mathsf{p} \xrightarrow{w_\mathsf{f}} \mathsf{this}_\mathsf{set}$ models flow of p into field f of $\mathsf{this}_\mathsf{set}$, and $\mathsf{this}_\mathsf{get} \xrightarrow{r_\mathsf{f}} \mathsf{ret}$ models read of field f from $\mathsf{this}_\mathsf{get}$ into ret.

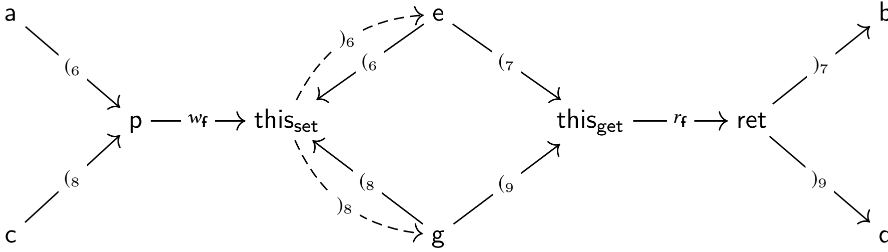
The principal problem is to decide whether there is flow from one node to another and CFL-reachability analysis makes use of the call/return and write/read annotations to make the decisions. In our example, there is a path from e to b because the call and return annotations, $(_7$ and $)_7$ respectively, match. However, there is no path from e to d because call annotation $(_7$ and return annotation $)_9$ do not match; they denote two distinct calls. Analogously, field write and field read annotations have to match; there is a path from p to ret because w_f and r_f denote a write and a read of the same field f. The analysis decides that there is flow from a to b and from c to d, however there is no flow from a to d or from c to b.

There are two notable points. The first point concerns inverse edges. In the example, there is a forward edge from e to $\mathsf{this}_\mathsf{set}$ (solid: $\longrightarrow$) and there is an inverse edge from $\mathsf{this}_\mathsf{set}$ to e (dashed: $\dashrightarrow$) that reverses the direction of flow and the annotation; call annotation $(_6$ becomes return annotation $)_6$. The forward edge is a natural addition to the graph representing flow of receiver e to $\mathsf{this}_\mathsf{set}$. The inverse edge, however, is unnatural but it is necessary to discover the path from a to e and then to b as the *mutation*, i.e., update of $\mathsf{this}_\mathsf{set}$ reverses flow. Standard CFL-reachability in the presence of mutable heap data (e.g., [Zhang and Su 2017], [Lu and Xue 2019]) adds an inverse edge for every forward edge in order to account for aliasing; in our example, every solid edge

```
1  public class A {
2     B f;
3     void set(A this, B p) {
4         this.f = p;
5     }
6     B get(A this) {
7         ret = this.f;
8         return ret;
9     }
10 }
```

```
1  public class C {
2     public static void main(...) {
3         A e = new A();
4         A g = new A();
5         ...
6         e.set(a);
7         b = e.get();
8         g.set(c);
9         d = g.get();
10    }
11 }
```

Fig. 3. Running example. a in line 6 in main flows to b in line 7; c in line 8 flows to d in line 9. We make parameter this explicit.



would have had a corresponding dashed edge in the standard G_{BI}. Our analysis takes into account reference immutability information and adds only those inverse edges that are necessary to decide flow correctly; the graph shown is the G_{RI}. One key problem we address is to show that G_{BI}, and more interestingly G_{RI}, indeed captures all run-time flows. We define a dynamic semantics that formalizes run-time flows, in our example, the meaning of "p in context of invocation of set in line 6 flows to ret in context of invocation of get in line 7" (Sect. 3). We proceed to define the construction of G_{BI} and G_{RI} (Sect. 4) and argue soundness, i.e., that G_{RI} does represent all run-time flows (Sect. 5). The second point concerns paths with interleaved call/return and write/read annotations. For example, the path from a to b involves matching call/return annotations, $(_6$ and $)_6$, as well as $(_7$ and $)_7$, and separately, matching write/read annotations w_f and r_f. Exact reasoning over such paths is undecidable [Reps 2000]. We present a certain approximate reachability analysis over G_{RI} and show that type-based FlowCFL is equivalent to that analysis (Sect. 7). We interpret the seemingly different type-based FlowCFL in terms of CFL-reachability (Sect. 6).

3 DYNAMIC SEMANTICS

In this section we formalize the notion of flow in terms of a dynamic semantics. We restrict our core language to a "named form" in the style of Vaziri et al. [2010] and Dolby et al. [2012]. The language models Java with the syntax in Fig. 4, 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.

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

Fig. 4. 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. The syntax separates object creation from initialization, i.e., x = new t() becomes x = new t; x.init(). (We can exclude poly from the syntax, but we include it because it can be used to annotate polymorphic libraries.)

3.1 Stack Contexts and Chains

Stack contexts describe stack configurations at a point of program execution; as expected, they help formalize run-time local variables, such as “p in context of invocation of set at line 6“. $\langle \mathsf{main}, f_1, f_2 ... f_n\rangle$ is the stack made up of main, followed by frame identifier f_1 corresponding to some callee m_1 in main, followed by f_2 for some callee m_2 in m_1, etc, with frame f_n at the top of the stack. Each f_i has a unique identifier—if, say, m_2 is called again from the same call site in m_1, that would entail a new frame and a new frame identifier at runtime. We use $A, B, C, \ldots$ to denote stack contexts. Local variables, naturally, are characterized by their stack context: we write x^A to denote local variable x in context A.

In our running example in Fig. 3, we have stack contexts $\langle \mathsf{main}, f_1\rangle$ and $\langle \mathsf{main}, f_2\rangle$ where f_1 corresponds to the frame of set invoked in line 6, and f_2 corresponds to the frame of set invoked in line 8. Variables in set are characterized by their stack context: $\mathsf{p}^{\langle \mathsf{main}, f_1\rangle}$, $\mathsf{this}^{\langle \mathsf{main}, f_1\rangle}$, $\mathsf{p}^{\langle \mathsf{main}, f_2\rangle}$, etc.

The notion of the *chain* is essential in our treatment. Informally, there is a chain from x^A to y^B, denoted by $(\mathsf{x}^A, \mathsf{y}^B)$, if x^A flows to y^B. In Fig. 3 there is a chain from $\mathsf{p}^{\langle \mathsf{main}, f_1\rangle}$ to $\mathsf{ret}^{\langle \mathsf{main}, f_3\rangle}$ where f_3 is the frame that corresponds to the invocation of get in line 7. Similarly, there are chains $(\mathsf{a}^{\langle \mathsf{main}\rangle}, \mathsf{ret}^{\langle \mathsf{main}, f_3\rangle})$, $(\mathsf{a}^{\langle \mathsf{main}\rangle}, \mathsf{b}^{\langle \mathsf{main}\rangle})$ among others. Chains are represented in G_{RI} by appropriately annotated paths. E.g., chain $(\mathsf{a}^{\langle \mathsf{main}\rangle}, \mathsf{ret}^{\langle \mathsf{main}, f_3\rangle})$ is represented by path

$$\mathsf{a} \xrightarrow{(_6} \mathsf{p} \xrightarrow{w_\mathsf{f}} \mathsf{this_{set}} \xrightarrow{)_6} \mathsf{e} \xrightarrow{(_7} \mathsf{this_{get}} \xrightarrow{r_\mathsf{f}} \mathsf{ret}$$

The unmatched call annotation $(_7$ represents that a flows into frame f_3, as f_3 maps to call site 7 in the abstract. The string with unmatched call $(_7$ captures in the abstract the “difference” between contexts $\langle \mathsf{main}\rangle$ and $\langle \mathsf{main}, f_3\rangle$.

The semantics is a standard small-step dynamic semantics extended with the treatment of chains. We write $[\![s]\!](A, \mathbb{C}, \mathbb{S}, \mathbb{H}) = \mathbb{C}', \mathbb{S}', \mathbb{H}'$ to model execution of statement s in context A and its effect on chains $\mathbb{C}$, stack $\mathbb{S}$, and heap $\mathbb{H}$. Here A is the list of invocations and $\mathbb{S}$ is a map from variables, where each variable is decorated with a list of invocations A, to their values. $\mathbb{C}$ is a map from variables y^B to *sets of sources* of chains that end at y^B. In addition, $\mathbb{C}$ includes a map from object fields $o.\mathsf{f}$ to sets of sources of chains that end at $o.\mathsf{f}$. We say $(\mathsf{x}^A, \mathsf{y}^B) \in \mathbb{C}$ iff $\mathsf{y}^B \in Dom(\mathbb{C})$ and $\mathsf{x}^A \in \mathbb{C}(\mathsf{y}^B)$. $\mathbb{S}$ and $\mathbb{H}$ are the standard maps from variables to objects o (map $\mathbb{S}$), and from object/field tuples $o.\mathsf{f}$ to objects o' (map $\mathbb{H}$). Below we discuss the semantics of individual statements.

An assignment x = y in context A records chains that end at x^A. There is a new chain $(v, \mathsf{x}^A) \in \mathbb{C}'$ for every chain $(v, \mathsf{y}^A) \in \mathbb{C}$, and there is a chain $(\mathsf{y}^A, \mathsf{x}^A) \in \mathbb{C}'$ that accounts for the flow from y to x. The transition on the stack is standard: x^A points to the object that y^A points to and the heap

remains the same:

$$\text{ASSIGN} \quad [\![\mathsf{x} = \mathsf{y}]\!](A, \mathbb{C}, \mathbb{S}, \mathbb{H}) = \mathbb{C}[\mathsf{x}^A \leftarrow \mathbb{C}(\mathsf{y}^A) \cup \{\mathsf{y}^A\}], \mathbb{S}[\mathsf{x}^A \leftarrow \mathbb{S}(\mathsf{y}^A)], \mathbb{H}$$

Field write x.f = y records chains $(v, o.\mathsf{f})$ and field read $\mathsf{y}' = \mathsf{x}'.\mathsf{f}$ references those chains to record chains $(v, (\mathsf{y}')^A)$:

$$\begin{array}{llcl} \text{WRITE} & [\![\mathsf{x.f} = \mathsf{y}]\!](A, \mathbb{C}, \mathbb{S}, \mathbb{H}) & = & \mathbb{C}[o.\mathsf{f} \leftarrow \mathbb{C}(\mathsf{y}^A) \cup \{\mathsf{y}^A\}], \mathbb{S}, \mathbb{H}[o.\mathsf{f} \leftarrow o'] \\ & & & \quad \text{where } o = \mathbb{S}(\mathsf{x}^A) \text{ and } o' = \mathbb{S}(\mathsf{y}^A) \\ \text{READ} & [\![\mathsf{y}' = \mathsf{x}'.\mathsf{f}]\!](A, \mathbb{C}, \mathbb{S}, \mathbb{H}) & = & \mathbb{C}[(\mathsf{y}')^A \leftarrow \mathbb{C}(o.\mathsf{f})], \mathbb{S}[(\mathsf{y}')^A \leftarrow o'], \mathbb{H} \\ & & & \quad \text{where } o = \mathbb{S}((\mathsf{x}')^A) \text{ and } o' = \mathbb{H}(o.\mathsf{f}) \end{array}$$

Note that we do not record o.f as a chain source in READ. It is an invariant of $\mathbb{C}$ that the values in map $\mathbb{C}$ are sets that contain only variables, e.g., x^A. We do record o.f as chain target in WRITE because it serves as an intermediary in the chain from y in x.f = y to y′ in y′ = x′.f. The semantics of chains elides heap objects, just as the static flow graph G_{RI} does (recall the graph in Sect. 2.3). The goal is to establish a connection between the concrete domain of chains and the abstract domain of annotated paths in the flow graph.

Allocation x = new o creates the trivial chain $(\mathsf{x}^A, \mathsf{x}^A)$:

$$\begin{array}{llcl} \text{ALLOC} & [\![\mathsf{x} = \mathsf{new}\ o]\!](A, \mathbb{C}, \mathbb{S}, \mathbb{H}) & = & \mathbb{C}[\mathsf{x}^A \leftarrow \{\mathsf{x}^A\}], \mathbb{S}[\mathsf{x}^A \leftarrow o], \mathbb{H}[o.\mathsf{f} \leftarrow \mathsf{null}] \\ & & & \quad \text{where } o \text{ is a fresh object} \end{array}$$

A call entails a fresh frame identifier f, which is appended to context A to form the new context $A \oplus f$. Calls record chains that reflect the standard flow from actuals to formals:

$$\begin{array}{llcl} \text{CALL} & [\![\mathsf{x} = \mathsf{y.m(z)}]\!](A, \mathbb{C}, \mathbb{S}, \mathbb{H}) & = & \mathbb{C}[\mathsf{this}^{A\oplus f} \leftarrow \mathbb{C}(\mathsf{y}^A) \cup \{\mathsf{y}^A\}][\mathsf{p}^{A\oplus f} \leftarrow \mathbb{C}(\mathsf{z}^A) \cup \{\mathsf{z}^A\}], \\ & & & \mathbb{S}[\mathsf{this}^{A\oplus f} \leftarrow \mathbb{S}(\mathsf{y}^A)][\mathsf{p}^{A\oplus f} \leftarrow \mathbb{S}(\mathsf{z}^A)], \mathbb{H} \\ & & & \quad \text{where } f \text{ is a fresh frame identifier} \end{array}$$

A return from context $A \oplus f$ creates chains with target x^A; these chains are due to the standard flow from $\mathsf{ret}^{A\oplus f}$ to the left-hand side of the return assignment x^A:

$$\text{RET} \quad [\![\mathsf{x} = \mathsf{y.m(z)}]\!](A \oplus f, \mathbb{C}, \mathbb{S}, \mathbb{H}) \;=\; \mathbb{C}[\mathsf{x}^A \leftarrow \mathbb{C}(\mathsf{ret}^{A\oplus f}) \cup \{\mathsf{ret}^{A\oplus f}\}], \mathbb{S}[\mathsf{x}^A \leftarrow \mathbb{S}(\mathsf{ret}^{A\oplus f})], \mathbb{H}$$

The following lemma allows us to express the points-to relation entailed by $\mathbb{S}$ and $\mathbb{H}$ in terms of the reachability relation entailed by $\mathbb{C}$. If a variable x^A points to some object o, then there is a chain $(\mathsf{w}^B, \mathsf{x}^A)$ in $\mathbb{C}$ where w^B is the local variable at the left-hand side of the allocation site of o. If we have x.f = y in context A followed by y′ = x′.f in context A', where x and x′ point to the same object o, then there is flow from y to y′. The flow is expressed via paths that represent chains $(\mathsf{w}^B, \mathsf{x}^A)$ and $(\mathsf{w}^B, (\mathsf{x}')^{A'})$. The key idea is that in the abstract, we have a path from w to x and an *inverse path* from x to w. The inverse path $\mathsf{x} \rightsquigarrow \mathsf{w}$ combines with the path $\mathsf{w} \rightsquigarrow \mathsf{x}'$ which leads to a path $\mathsf{x} \rightsquigarrow \mathsf{x}'$ and subsequently $\mathsf{y} \rightsquigarrow \mathsf{y}'$. Recall Fig. 3 and the accompanying graph:

$$\mathsf{this_{set}} \overset{)_6}{\dashrightarrow} \mathsf{e} \text{ and } \mathsf{e} \overset{(_7}{\longrightarrow} \mathsf{this_{get}} \text{ give rise to } \mathsf{p} \overset{w_\mathsf{f}}{\longrightarrow} \mathsf{this_{set}} \overset{)_6}{\dashrightarrow} \mathsf{e} \overset{(_7}{\longrightarrow} \mathsf{this_{get}} \overset{r_\mathsf{f}}{\longrightarrow} \mathsf{ret}$$

which abstracts the chain from $\mathsf{p}^{\langle \mathsf{main}, f_1 \rangle}$ to $\mathsf{ret}^{\langle \mathsf{main}, f_3 \rangle}$. We elaborate later in the paper.

LEMMA 3.1. *For every state $\mathbb{C}, \mathbb{S}, \mathbb{H}$ and every object $o \in \mathbb{H}$*

- $\mathbb{S}(\mathsf{x}^A) = o \quad \Rightarrow \quad (\mathsf{w}^B, \mathsf{x}^A) \in \mathbb{C}$ *and*
- $\mathbb{H}(o'.\mathsf{f}) = o \quad \Rightarrow \quad (\mathsf{w}^B, o'.\mathsf{f}) \in \mathbb{C}$

where w = *new* C *in context B is the creation site of o.*

Our technical report [Milanova 2020] presents a proof sketch.

3.2 Operations on Contexts

Next we define several useful operations on stack contexts. $A - B$ is defined when B is a prefix of A; it removes B from A. $A \leq B$ is true if and only if A is a prefix of B. ΔAB denotes the "difference" between context A and context B. ΔAB is defined as the tuple $(A - D, B - D)$, where D is the longest common prefix of A and B. Such a prefix clearly exists, in the worst case it is main.

Returning to Fig. 3, consider the flow from p in context $\langle \mathsf{main}, f_2 \rangle$ of set, to ret in context $\langle \mathsf{main}, f_3 \rangle$ of get (recall that f_3 is the frame invoked in line 7). We have:

$$\Delta \langle \mathsf{main}, f_2 \rangle \langle \mathsf{main}, f_3 \rangle = (\langle f_2 \rangle, \langle f_3 \rangle)$$

Informally, the first term in the tuple is the sequence of *returns* and the second term is the sequence of *calls* that happen when state transitions from A to B. In the example, state transitions from $\langle \mathsf{main}, f_2 \rangle$ to $\langle \mathsf{main}, f_3 \rangle$ by first returning from f_2 into main, then calling into f_3 from main.

In Sect. 5 we define an abstraction function over stack contexts and differences ΔAB. It helps establish that for every chain from x^A to y^B, G_{BI} and G_{RI} contain appropriately annotated paths from x to y. As stated earlier, a key difficulty arises in the reasoning about inverse edges.

4 CFL-REACHABILITY

Sect. 4.1 describes the construction of G_{BI}. Sect. 4.2 argues that there is inherent imprecision in G_{BI}. Sect. 4.3 discusses reference immutability and Sect. 4.4 describes the construction of G_{RI} based on knowledge of reference immutability.

4.1 Bidirectional Flow Graph G_{BI}

As it is customary for CFL-reachability, we build a static *flow graph* that represents data dependences between variables. The nodes in the graph are (context-insensitive) program variables, e.g., x, y, this. The edges represent flow from one variable to another and paths represent dynamic chains as defined in Sect. 3. The standard approach in the presence of mutable data is to build a *bidirectional* flow graph (as in [Chatterjee et al. 2018; Sampson et al. 2011; Shankar et al. 2001; Späth et al. 2019; Sridharan and Bodík 2006; Xu et al. 2009; Zhang and Su 2017] among other works) where *inverse edges* handle updates safely. We call this graph G_{BI}. Below we describe the semantics of G_{BI} construction. Solid arrows $\longrightarrow$ denote forward edges, and dashed arrows $\dashrightarrow$ denote inverse edges.

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

$$\text{ASSIGN} \quad [\![\mathsf{x} = \mathsf{y}]\!](G_{BI}) = G_{BI} \cup \{\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x}\} \cup \{\mathsf{x} \overset{\mathsf{d}}{\dashrightarrow} \mathsf{y}\}$$

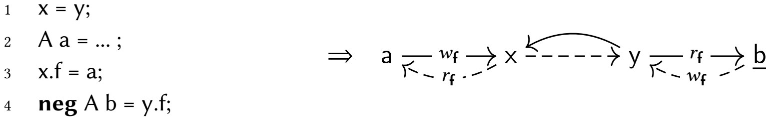
A field write statement x.f = y contributes a forward edge from y to x annotated with w_f and an inverse edge from x to y annotated with r_f:

$$\text{WRITE} \quad [\![\mathsf{x.f} = \mathsf{y}]\!](G_{BI}) = G_{BI} \cup \{\mathsf{y} \xrightarrow{w_\mathsf{f}} \mathsf{x}\} \cup \{\mathsf{x} \overset{r_\mathsf{f}}{\dashrightarrow} \mathsf{y}\}$$

The meaning of the forward edge is that y flows (is written) into field f of x. The corresponding inverse edge reverses the direction of the flow and the field annotation, denoting that field f of x is read into y. Similarly, a field read statement $\mathsf{y}' = \mathsf{x}'.\mathsf{f}$ contributes

$$\text{READ} \quad [\![\mathsf{y}' = \mathsf{x}'.\mathsf{f}]\!](G_{BI}) = G_{BI} \cup \{\mathsf{x}' \xrightarrow{r_\mathsf{f}} \mathsf{y}'\} \cup \{\mathsf{y}' \overset{w_\mathsf{f}}{\dashrightarrow} \mathsf{x}'\}$$

The following example illustrates once again the need for inverse edges. From now on, we will underline sinks in the graphs to improve readability.



Inverse edge $x \overset{d}{\dashrightarrow} y$ is necessary to establish the path from a to b.

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

$$\text{CALL} \quad [\![i: \mathsf{x = y.m(z)}]\!](G_{BI}) = G_{BI} \cup \{\mathsf{y} \overset{(_i}{\rightarrow} \mathsf{this}\} \cup \{\mathsf{z} \overset{(_i}{\rightarrow} \mathsf{p}\} \cup \{\mathsf{this} \overset{)_i}{\dashrightarrow} \mathsf{y}\} \cup \{\mathsf{p} \overset{)_i}{\dashrightarrow} \mathsf{z}\}$$

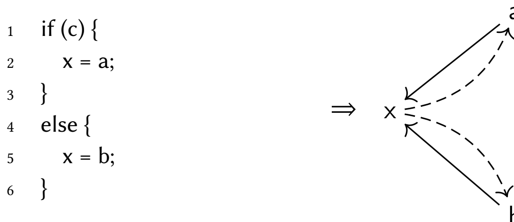
The standard CFL-reachability annotation $(_i$ marks call entry at call site i. A method return (exit) creates a *return* edge from the return value to the left-hand side of the call assignment, plus the inverse *call* edge:

$$\text{RET} \quad [\![i: \mathsf{x = y.m(z)}]\!](G_{BI}) = G_{BI} \cup \{\mathsf{ret} \overset{)_i}{\rightarrow} \mathsf{x}\} \cup \{\mathsf{x} \overset{(_i}{\dashrightarrow} \mathsf{ret}\}$$

The CFL-reachability problem is to decide whether there is a path from x to y in G_{BI} with properly matched call/ret annotations, and properly matched write/read annotations. Note the arbitrary interleaving of (,) and w, r annotations. Due to recursion in both call-transmitted and heap-transmitted dependences, the CFL-reachability problem is undecidable [Reps 2000] and analyses have to adopt approximations. One approximation essentially replaces all $(_i$ and $)_j$ with d annotations and leaves all w, r annotations, thus treating call-transmitted dependences context-insensitively and heap-transmitted dependences fully precisely. This approach is known as CIFS (context-insensitive, field-sensitive) CFL-reachability [Zhang and Su 2017]. Another approximation replaces all w_{f} and r_{g} annotations with d thus treating call-transmitted dependences fully precisely and heap-transmitted dependences approximately. This approach is known as CSFI (context-sensitive, field-insensitive) CFL-reachability [Zhang and Su 2017]. Consider the CR and PG context-free grammars in Fig. 5. CSFI amounts to PG-reachability, i.e., if the path from x to y is in the language L(PG), then y is PG-reachable from x. Analogously, CIFS amounts to CR-reachability, i.e., if the path from x to y is in L(CR), then y is CR-reachable from x. There are many other approximations that have been proposed in the literature. For example, an interesting new approximation by Spath et al. [Späth et al. 2019] shows that (essentially) tracking whether y is independently PG- and CR-reachable from x largely retains the precision of tracking properly matched parentheses and w, r annotations.

4.2 Imprecision in G_{BI}

Bidirectionality of G_{BI} causes imprecision. Consider the example:



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

(a) Call-Return (CR) CFG

$$
\begin{array}{lcl}
G & ::= & r_{\mathsf{f}} \mid r_{\mathsf{f}}\, B \mid r_{\mathsf{f}}\, G \mid B\, G \\
P & ::= & w_{\mathsf{f}} \mid w_{\mathsf{f}}\, B \mid w_{\mathsf{f}}\, P \mid B\, P \\
B & ::= & \mathsf{d} \mid w_{\mathsf{f}}\, B\, r_{\mathsf{f}} \mid \mathsf{d}\, B \mid w_{\mathsf{f}}\, B\, w_{\mathsf{f}}\, B
\end{array}
$$

(b) Put-Get (PG) CFG

Fig. 5. The CR grammar in (a) defines well-formed call-transmitted paths. R generates strings with outstanding (R)eturn edges, e.g., $(_1\ \mathsf{d}\ )_1\ )_2$. C generates strings with outstanding (C)all edges, e.g., $(_1$, and M generates same-level paths, i.e., paths with matching call and return edges, e.g., $(_1\ \mathsf{d}\ )_1$. The PG grammar in (b) defines heap-transmitted paths. P generates strings with outstanding [G]etfield (i.e., field read) edges, e.g., $w_{\mathsf{f}}\ \mathsf{d}\ r_{\mathsf{f}}\ r_g$. P generates strings with outstanding [P]utfield (i.e., field write) edges, e.g., $w_{\mathsf{f}}\ \mathsf{d}$. Finally, B generetes strings with matching field write and field read edges, e.g., $w_{\mathsf{f}}\ \mathsf{d}\ r_{\mathsf{f}}$.

The two forward edges have corresponding inverse edges, creating paths from x to b and from a to b. If b is negative, then both x and a spuriously become negative. The imprecision propagates, "polluting" variables throughout the program.

Zhang and Su [2017] report that linear conjunctive reachability over G_{BI}, a novel highly-precise CFL-reachability technique, achieves only modest improvement compared to CSFI over G_{BI}; they conjecture that this is due to the bidirectionality of G_{BI}. In contrast, the technique achieves substantial precision improvement over G_{RI} [Zhang and Su 2017]. (Recall that G_{RI} is the subgraph of G_{BI} that avoids certain inverse edges; we describe reference immutability and the construction of G_{RI} in Sect. 4.3 and Sect. 4.4.) Similarly, Milanova and Huang [2013] report substantial negative impact of bidirectionality on taint analysis of Java web applications—a taint analysis that removes infeasible edges based on reference immutability information infers 20% to 79% fewer negative variables compared to a taint analysis on the bidirectional graph. (Recall that the goal of taint analysis is to propagate negative sinks to as few program variables as possible.)

We conducted experiments using the implementation and benchmarks of DroidInfer [Huang et al. 2015] that are made publicly available with the artifact of DroidInfer.[1] We ran the analysis on 77 Android apps from the artifact, excluding apps that crashed and apps that contained 0 sources or 0 sinks. We ran the taint analysis over G_{BI} and over G_{RI}. *Analysis over G_{RI} reduced the number of reported errors by 41% on average per app compared to G_{BI}. An error essentially corresponds to a* (source, sink) *pair, or in other words, to a report of flow from* source *to* sink*; the analysis proved 120% more apps safe compared to analysis over G_{BI}*. These results confirm that removing infeasible inverse edges benefits analysis precision.

Inverse edges capture bidirectionality of aliasing. Suppose reference x flows to x′. Then for all fields f, x.f and x′.f are aliases. If we *write* a value into x′.f, we should be able to *read* that value out of x.f and the inverse path from x′ to x enables that. However, if x′ is an immutable reference, then there is no need for the inverse path from x′ to x.

4.3 Reference Immutability

A key goal of this work is to formalize the notion of the inverse edge and understand the role of reference immutability in removing infeasible inverse edges. Reference immutability [Huang et al. 2012b; Milanova 2018; Tschantz and Ernst 2005] ensures that a *readonly* (also called immutable) reference cannot be used to mutate the state of the object it refers to, including its transitive state.

[1] The artifact and code are publicly available at https://github.com/proganalysis/type-inference.

For example, x is not readonly in y = x; y.f = z; because it is used in a way that leads to a mutation of the object it references; similarly x is not readonly in y.f = x; z = y; w = z.f; w.g = 10;.

Reference immutability semantics is typically described in terms of a type system, most notably Javari [Tschantz and Ernst 2005] and ReIm [Huang et al. 2012b]. Recent work has shown that it can be described in terms of CFL-reachability as well [Milanova 2018]. In this paper, we largely follow the CFL-reachability interpretation of ReIm given in [Milanova 2018].

In the style of Sect. 4.1, we analyze each program statement and build a *reference immutability graph* G then decide immutability/mutability of references based on reachability over G. Informally, x is mutable if it reaches a variable that is updated, such as y and w above. An assignment statement contributes the following forward edge to G. There are no inverse edges in reference immutability:

$$\text{ASSIGN} \quad [\![\mathsf{x} = \mathsf{y}]\!](G) = G \cup \{\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x}\}$$

Similarly to Sect. 4.1 a method call creates call and return forward edges as expected:

$$\text{CALL/RET} \quad [\![i : \mathsf{x} = \mathsf{y.m(z)}]\!](G) = G \cup \{\mathsf{y} \xrightarrow{(_i} \mathsf{this}\} \cup \{\mathsf{z} \xrightarrow{(_i} \mathsf{p}\} \cup \{\mathsf{ret} \xrightarrow{)_i} \mathsf{x}\}$$

A difference with Sect. 4.1 arises in the handling of heap-transmitted dependences. Together, a pair of field write x.f = y and field read y′ = x′.f contribute the following edges

$$\text{WRITE/READ} \quad [\![\mathsf{x.f} = \mathsf{y},\ \mathsf{y'} = \mathsf{x'.f}]\!](G) = G \cup \{\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f} \xrightarrow{\mathsf{d}} \mathsf{y'}\} \cup \{\mathsf{x'} \xrightarrow{\mathsf{d}} \mathsf{x'.f}\}$$

The first set of edges creates a path from y to y′. Here $\mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f}$ is an *approximate edge*. In terms of Reps' terminology [Reps 2000], the reference immutability semantics models heap-transmitted dependences approximately. The approximation comes from the fact that regardless of whether x.f and x′.f are aliases, the semantics propagates y′ back to y. If there is an update of y′, e.g., y′.g = 5, then y′ is mutable by definition, and y will be determined mutable via path $\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f} \xrightarrow{\mathsf{d}} \mathsf{y'}$. This approximate handling of aliasing, which forgoes "going back" to the allocation site, makes inverse edges unnecessary here. Note that, one can improve this graph by avoiding edges $\mathsf{x.f} \overset{\mathsf{a}}{\dashrightarrow} \mathsf{x'.f}$ that are ruled out by an imprecise alias analysis such as for example reachability over G_{BI}. A field read contributes an additional edge $\mathsf{x'} \xrightarrow{\mathsf{d}} \mathsf{x'.f}$ needed to account for the mutation to transitive state. If there is a mutation of y′, this last edge propagates the mutation back to x′.

An *update* is a node y such that there is a write statement of the form y.f = z. Essentially, we are interested if there is an "appropriately annotated" path in G from a reference x to an update node. For example, consider the code and its corresponding graph G. id is the standard identity function:

$$i : \mathsf{x} = \mathsf{id(y)}; \mathsf{z} = \mathsf{x}; \mathsf{w} = \mathsf{z.f}; \mathsf{w.g} = 10; \quad \Rightarrow \quad \mathsf{y} \xrightarrow{(_i} \mathsf{p} \xrightarrow{\mathsf{d}} \mathsf{ret} \xrightarrow{)_i} \mathsf{x} \xrightarrow{\mathsf{d}} \mathsf{z} \xrightarrow{\mathsf{d}} \mathsf{z.f} \xrightarrow{\mathsf{d}} \mathsf{w}$$

There is a path from y to the update w with properly matched parenthesis which means that y is a mutable reference; the object o that y refers to is modified through y because passing y to id leads to the mutation o.f.g = 10.

A *call-transmitted* path contains no approximate edges and it is well-formed in CR, i.e., its annotations form a string in L(CR) (recall Fig. 5(a)). A *heap-transmitted* path is made up of call-transmitted paths interleaved with approximate edges. Let U denote the set of all call-transmitted and heap-transmitted paths to updates in G. We break paths in U into 2 categories: (1) *M/C-paths* and (2) *R-paths*. A path in U is an M/C-path if and only if the annotations on the *leading*, i.e., first, call-transmitted path form a string in the language described by M (i.e., calls and returns balance out), or they form a string in the language described by C (i.e., outstanding calls). For example, $\mathsf{e} \xrightarrow{(_6} \mathsf{this_{set}}$ is a C-path. A path in U is an R-path if and only if the edge annotations on the leading

call-transmitted path form a string in R, e.g., $\mathsf{ret} \xrightarrow{)_7} \mathsf{b}$ is an R-path. We consider that there is a trivial path from each node to itself, so an update node is mutable.

(1) Examples of M/C-paths, following the graph in Sect. 2.3, include (assuming b is an update):

$$\mathsf{e} \xrightarrow{(_7} \mathsf{this_{get}} \xrightarrow{\mathsf{d}} \mathsf{this_{get}.f} \xrightarrow{\mathsf{d}} \mathsf{ret} \xrightarrow{)_7} \underline{\mathsf{b}} \text{ (leading call-transmitted path is an } M\text{-path)}$$

and

$$\mathsf{a} \xrightarrow{(_6} \mathsf{p} \xrightarrow{\mathsf{d}} \mathsf{this_{set}.f} \dashrightarrow^{\mathsf{a}} \mathsf{this_{get}.f} \xrightarrow{\mathsf{d}} \mathsf{ret} \xrightarrow{)_7} \underline{\mathsf{b}} \text{ (leading call-transmitted path is a } C\text{-path)}$$

(2) examples of R-paths include (again assuming b is an update):

$$\mathsf{this_{get}} \xrightarrow{\mathsf{d}} \mathsf{this_{get}.f} \xrightarrow{\mathsf{d}} \mathsf{ret} \xrightarrow{)_7} \underline{\mathsf{b}}$$

Reference immutability classifies variables as follows [Huang et al. 2012b; Milanova 2018; Tschantz and Ernst 2005]:

x is mutable	if there is an M/C-path from x to an update
x is poly	if there is no M/C-path but there is an R-path from x to an update
x is readonly	if there is neither M/C-path nor R-path from x to an update

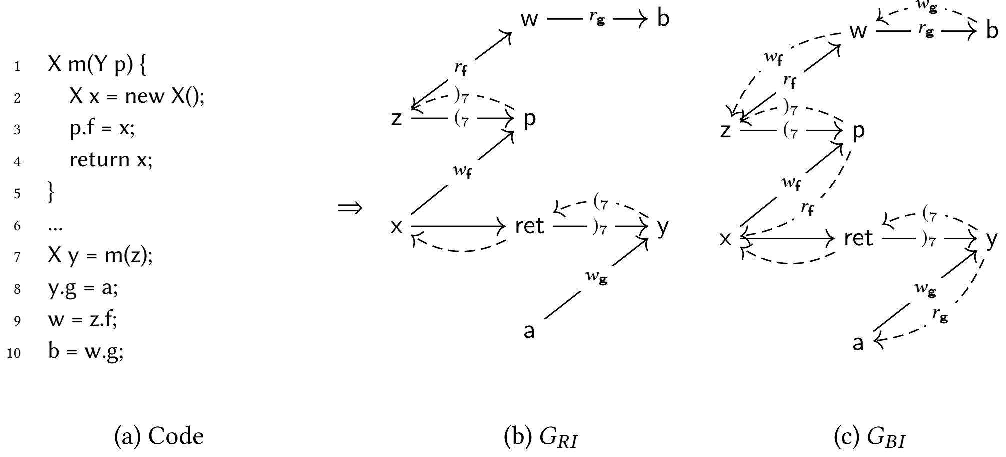
In the above examples e and a are both mutable and $\mathsf{this_{get}}$ and ret are poly, due to the R-paths to the update. If we removed the assumption that b is an update, then e, a, $\mathsf{this_{get}}$ and ret above would be readonly. Intuitively, an M/C-path from x unequivocally makes x mutable—mutation is immediate or within the immediate call. An R path does not necessarily make x mutable; it is mutable in the context of the R-path, but it may be readonly in the context of other paths. For example, mutable b = e.get() makes $\mathsf{this_{get}}$ mutable in the context of this path (precisely the R-path above), however, readonly d = g.get() leaves $\mathsf{this_{get}}$ readonly in the context of this different path: $\mathsf{this_{get}} \xrightarrow{\mathsf{d}} \mathsf{this_{get}.f} \xrightarrow{\mathsf{d}} \mathsf{ret} \xrightarrow{)_9} \mathsf{d}$.

Lastly, we summarize the *adaptation operation* [Dietl et al. 2011; Huang et al. 2012b] which plays a role in the theorem that proves soundness of CFL-reachability over G_{RI}. Viewpoint adaptation, written as $\mathsf{x} \rhd \mathsf{p}$, interprets the immutability type of p in the context of x. We simplify the adaptation notation to work on variables rather than immutability qualifiers. $\mathsf{x} \rhd \mathsf{p}$ refers to the immutability types of x and p, not to the variables themselves.

$$\begin{array}{lcllcl} \mathsf{x} & \rhd & \mathsf{readonly} & = & \mathsf{readonly} \\ \mathsf{x} & \rhd & \mathsf{mutable} & = & \mathsf{mutable} \\ \mathsf{x} & \rhd & \mathsf{poly} & = & \mathsf{x} \end{array}$$

A readonly or mutable p remains as is, regardless of the context x. A poly p takes the type of x, which makes adaptation interesting. In reference immutability, left-hand sides x of call assignments serve as contexts of adaptation. This is because, intuitively, there are multiple paths from a poly variable p in m, one through each one of the left-hand sides of calls to m; the mutability status of each path is determined by the left-hand side of the call. If i : x = m(z) is such that x is mutable, then p is mutable at i. Returning to the examples above, at mutable b = e.get() we have $\mathsf{b} \rhd \mathsf{this_{get}} = \mathsf{mutable}$ and $\mathsf{b} \rhd \mathsf{ret} = \mathsf{mutable}$ reflecting that $\mathsf{this_{get}}$ and ret are mutable at the call to get at 7 because the left-hand side, i.e., the context b is mutable. In contrast, at readonly d = g.get() we have $\mathsf{d} \rhd \mathsf{this_{get}} = \mathsf{readonly}$ and $\mathsf{d} \rhd \mathsf{ret} = \mathsf{readonly}$. [2]

[2] Standard viewpoint adaptation from Universe types [Dietl et al. 2011] uses the receiver as context of adaptation, e.g., in x = y.m(z), the context of adaptation is y. Reference immutability uses the left-hand side x, which reflects its specific semantics; [Huang et al. 2012b] elaborates on this.



(a) Code (b) G_{RI} (c) G_{BI}

Fig. 6. An example contrasting G_{RI} and G_{BI}. G_{RI} retains the path from a to b but avoids multiple infeasible paths from G_{BI}, e.g., the path fro b to a.

We generalize adaptation to a sequence of contexts, e.g., $\mathsf{x}_i \rhd \mathsf{x}_j \rhd \mathsf{x}_k \rhd$ ret adapts ret in a larger context. Suppose ret is poly, and left-hand-sides x_k at call k and x_j at call j are poly, however, x_i at the outermost call site i is readonly; ret in context $\mathsf{x}_i \rhd \mathsf{x}_j \rhd \mathsf{x}_k$ is readonly.

```
1  A id0(A p0) { ret0 = p0; }

3  A id1(A p1) { ret1 = id0(p1); }

5  A id2(A p2) { ret2 = id1(p2); }
6  ...
7  mutable b = id2(a);
8  b.f = z;
9  readonly d = id2(c);
```

In the above example, ret0, p0, ret1, p1, ret2, and p2 are all poly due to the R-paths to update b. In context 7 ret0 is interpreted as b $\rhd$ ret2 $\rhd$ ret1 $\rhd$ ret0 = mutable $\rhd$ poly $\rhd$ poly $\rhd$ poly = mutable. b $\rhd$ ret2 $\rhd$ ret1 abstracts the stack context of id0 that line 7 initiates.

4.4 Flow Graph G_{RI}

We use reference immutability to build a new flow graph G_{RI} without certain infeasible inverse edges. In summary, explicit and implicit assignments forgo inverse edges if the left-hand-side of the assignment is readonly as illustrated by the rule for assignment statement.

$$\text{ASSIGN} \quad [\![\mathsf{x} = \mathsf{y}]\!](G_{RI}) = \begin{cases} G_{RI} \cup \{\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x}\} & \text{if x is readonly} \\ G_{RI} \cup \{\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x}\} \cup \{\mathsf{x} \overset{\mathsf{d}}{\dashrightarrow} \mathsf{y}\} & \text{otherwise} \end{cases}$$

The rules for WRITE, READ, CALL and RET are analogous. CALL i: x = y.m(z) adds inverse edge this $\overset{)_i}{\dashrightarrow}$ y when x $\rhd$ this $\neq$ readonly, and it adds p $\overset{)_i}{\dashrightarrow}$ z when x $\rhd$ p $\neq$ readonly. RET adds x $\overset{(_i}{\dashrightarrow}$ ret when x $\rhd$ ret $\neq$ readonly.

Consider the example in Fig. 6. p and y are updates and therefore mutable, and z is mutable as well due to the C-path to update p: z $\xrightarrow{(_7}$ p. x and ret are poly due to the R-path x $\xrightarrow{d}$ ret $\xrightarrow{)_7}$ y. However, w, a, b, as well as fields f and g are readonly, as there is neither M/C-path nor R-path to an update. The rules above entail only 3 inverse edges: (1) p $\overset{)_7}{\dashrightarrow}$ z, (2) y $\overset{(_7}{\dashrightarrow}$ ret, and (3) ret $\overset{d}{\dashrightarrow}$ x, precisely the edges needed to capture the path from a to b.

Our key result is that all chains as defined in Sect. 3 are represented by paths in G_{RI} with properly matched w/r and call/ret annotations.

5 SOUNDNESS OF CFL-REACHABILITY OVER G_{RI}

To prove soundness of reachability over G_{RI}, we first define the abstraction function $\alpha(A)$ over stack contexts A:

$$\alpha(\langle \mathsf{main}, f_1, f_2 ... f_n \rangle) = \langle i_1, i_2 ... i_n \rangle$$

where i_1, i_2, ... i_n are the static call sites that triggered frames f_1, f_2, ... f_n. $\alpha(A)$ extends to partial contexts A as follows: $\alpha(\langle f_2 ... f_n \rangle) = \langle i_2 ... i_n \rangle$.

$$\alpha(B) \triangleright \mathsf{y} = \alpha(\langle \mathsf{main}, f_1, f_2 ... f_n \rangle) \triangleright \mathsf{y} = \mathsf{x}_{i_1} \triangleright \mathsf{x}_{i_2} \triangleright ... \triangleright \mathsf{x}_{i_n} \triangleright \mathsf{y}$$

where $\mathsf{x}_{i_1}, \mathsf{x}_{i_2}, ... \mathsf{x}_{i_n}$ are the left-hand sides of call assignments i_1, i_2, ... i_n.

We define the abstract difference, $\alpha(\Delta AB)$, as the tuple $(\alpha(A - D), \alpha(B - D))$ where D is the longest common prefix of A and B. We will denote these tuples as $(ret, call)$ as $\alpha(A - D)$ abstracts a certain *return sequence* and $\alpha(B - D)$ abstracts a certain *call sequence* as we discussed in Sect. 3.2. We define the concatenation operation over abstract differences as follows:

$$(ret_1, call_1) \oplus (ret_2, call_2) = \begin{cases} (ret_1, (call_1 - ret_2) + call_2) & \text{if } ret_2 \text{ is a suffix of } call_1 \\ (ret_1 + (ret_2 - call_1), call_2) & \text{if } call_1 \text{ is a suffix of } ret_2 \\ undefined & \text{otherwise} \end{cases}$$

In the above, we overload *minus* − to subtract a suffix, not just a prefix of a string and + is just standard string concatenation. Consider $\alpha(\Delta AB) = (\alpha(A - D), \alpha(B - D)) = (ret_1, call_1)$ and $\alpha(\Delta BC) = (\alpha(B - E), \alpha(C - E)) = (ret_2, call_2)$. If we think of ret strings as strings of closing parentheses, i.e., $ret = \langle i_1, i_2 ... i_n \rangle =)_{i_n} ...)_{i_2}\ )_{i_1}$, and of $call$ strings as strings of opening parentheses, i.e., $call = \langle j_1, j_2 ... j_m \rangle = \ (_{j_1}\ (_{j_2} ... (_{j_m}$, then concatenation cancels out call annotations $(_k$ and return annotations $)_k$. For the remainder of this section we will interpret $\alpha(\Delta AB)$ to mean both the tuple $(ret, call)$ as defined above and the string $)_{i_n} ...)_{i_2}\ )_{i_1} (_{j_1}\ (_{j_2} ... (_{j_m}$. In the abstract domain, graph G_{RI}, we will be looking at paths from x to y; a chain $(\mathsf{x}^A, \mathsf{y}^B)$ will map into a path from x to y in G_{RI} such that the string of unmatched call and return annotations on that path is exactly $\alpha(\Delta AB)$. If $call_1 = \alpha(B - D)$ is longer than $ret_2 = \alpha(B - E)$, then $\alpha(B - D)$ cancels out $\alpha(B - E)$ and the outstanding call annotations are prepended onto $\alpha(C - E)$. Analogously, if $\alpha(B - E)$ is longer than $\alpha(B - D)$, then $\alpha(B - E)$ cancels out $\alpha(B - D)$, and the outstanding return annotations are appended to $ret_1 = \alpha(A - D)$. If neither $call_1$ cancels out ret_2, nor ret_2 cancels $call_1$, then concatenation is undefined because the strings represent distinct runtime contexts.

As an example, return to Fig. 3. $\mathsf{p}^{\langle \mathsf{main}, f_1 \rangle}$ flows to $\mathsf{ret}^{\langle \mathsf{main}, f_3 \rangle}$, and we are interested in the abstract difference $\alpha(\Delta \langle \mathsf{main}, f_1 \rangle \langle \mathsf{main}, f_3 \rangle)$ which is $(\langle 6 \rangle, \langle 7 \rangle)$ or just the string $)_6 (_7$. Similarly, since $\mathsf{ret}^{\langle \mathsf{main}, f_3 \rangle}$ flows to $\mathsf{b}^{\langle \mathsf{main} \rangle}$ we are interested in $\alpha(\Delta \langle \mathsf{main}, f_3 \rangle \langle \mathsf{main} \rangle)$ which is just $)_7$, with an empty call sequence. $)_6 (_7 \oplus)_7$ equals $)_6$.

The concatenation lemma below states that if we have the strings $\alpha(\Delta AB)$ and $\alpha(\Delta BC)$ their concatenation as defined above produces $\alpha(\Delta AC)$, precisely the abstraction of ΔAC.

LEMMA 5.1. $\alpha(\Delta AB) \oplus \alpha(\Delta BC) = \alpha(\Delta AC)$

Our technical report [Milanova 2020] presents the proof of the lemma.

Our main theorem, Theorem 5.2, shows that every chain $(\mathsf{x}^A, \mathsf{y}^B)$ in $\mathbb{C}$ is represented by an appropriately annotated path from x to y in G_{RI}. We write $\mathsf{x} \overset{\alpha(\Delta AB)}{\rightsquigarrow} \mathsf{y}$ to denote the existence of a path from x to y in G_{RI} with a string $s \in$ L(PG) $\cap$ L(CR)[3], where the PG (i.e., w and r) component of s is balanced, and the CR (i.e., (and)) component of s contains exactly the $\alpha(A - D)$ string of unbalanced returns, followed by the $\alpha(B - D)$ string of unbalanced calls. (D is the longest common prefix as expected.) As an example, consider $\mathsf{x} \xrightarrow{w_\mathsf{f}} \mathsf{p} \xrightarrow{)_7} \mathsf{z} \xrightarrow{r_\mathsf{f}} \mathsf{w}$ in Fig. 6(b); the field component is balanced while the call/ret component reflects that x flows from the context of the call at line 7 back into main. It is easy to see that if there is a path $\mathsf{x} \overset{\alpha(\Delta AB)}{\rightsquigarrow} \mathsf{y}$ and a path $\mathsf{y} \overset{\alpha(\Delta BC)}{\rightsquigarrow} \mathsf{z}$, then there is a path $\mathsf{x} \overset{\alpha(\Delta AB) \oplus \alpha(\Delta BC)}{\rightsquigarrow} \mathsf{z}$. By the concatenation lemma, this is exactly $\mathsf{x} \overset{\alpha(\Delta AC)}{\rightsquigarrow} \mathsf{z}$.

Theorem 5.2. *Let $\mathbb{C}, \mathbb{S}, \mathbb{H}$ be a program state and let $(\mathsf{x}^A, \mathsf{y}^B) \in \mathbb{C}$. The following statements are true.*

- *There is a path $\mathsf{x} \overset{\alpha(\Delta AB)}{\rightsquigarrow} \mathsf{y}$ in G_{RI}.*
- *If $\alpha(B) \triangleright \mathsf{y}$ is* mutable *according to reference immutability (ReIm), then there is an inverse path $\mathsf{y} \overset{\alpha(\Delta BA)}{\rightsquigarrow} \mathsf{x}$ in G_{RI}.*

A corollary is that if y is an update, i.e., we have a write y.f = z, then there is an inverse path $\mathsf{y} \overset{\alpha(\Delta BA)}{\rightsquigarrow} \mathsf{x}$ in G_{RI}. By definition, ReIm types an update as mutable and therefore $\alpha(B) \triangleright \mathsf{y}$ is mutable; the inverse path follows from the second clause of the theorem. An inverse path is not necessarily made up of inverse edges, it is just the "inverse" of $\mathsf{x} \overset{\alpha(\Delta AB)}{\rightsquigarrow} \mathsf{y}$.

Our technical report [Milanova 2020] presents the proof of the theorem.

Of course, even though we have shown that each chain has, roughly speaking, a corresponding path p in the intersection of L(PG) and L(CR), computing the exact set of such paths p is undecidable. We define an approximate reachability analysis over G_{RI}, CFL, where each path p in the intersection has a corresponding representative path in CFL. In the following section we define type-based FlowCFL and interpret it in terms of CFL-reachability. In Sect. 7 we define the CFL algorithm and establish equivalence of FlowCFL and CFL thus proving FlowCFL correct.

On a final note, since every chain is represented by a path in G_{RI}, it follows that every chain is represented by a path in G_{BI} as well, since G_{RI} is a subgraph of G_{BI}.

6 TYPE-BASED ANALYSIS

This section presents the type-based FlowCFL outlining parallels with CFL-reachability as described in Sect. 4. The reader may wonder why one needs a type system, when one has a clear semantics in terms of standard CFL-reachability. First, type systems and type-based taint analysis have already been used in the literature [Huang et al. 2014, 2015; Sampson et al. 2011; Shankar et al. 2001], in some cases without correctness proofs. CFL-reachability brings insight into type-based reachability/taint analysis, in particular, it can help explain type errors. Type-based reachability analysis reports potential flows as type errors, and CFL-reachability can explain type errors by mapping them to corresponding CFL-reachability source-sink paths. In addition, CFL-reachability presents a novel framework for reasoning about correctness of type-based analysis. Second, a type system allows programmers to specify requirements with type qualifiers, e.g., pos x, and take advantage of systems such as the Checker Framework (`https://checkerframework.org/`) to statically check these requirements; such requirements cannot be easily expressed or checked using

[3] As it is standard, for the purposes of the intersection, we extend the alphabets of grammars PG and CR to include the symbols of the other grammar.

CFL-reachability. Third, type systems are modular, while CFL-reachability systems are typically whole-program analyses. A significant advantage of a type-based interpretation is that it allows for modular reasoning. We can infer type annotations for libraries (e.g., as in [Huang et al. 2012b]), then type check a user program against annotated libraries while handling callbacks via standard function subtyping.

Sect. 6.1 describes the type qualifiers in FlowCFL and Sect. 6.2 describes the typing rules. Sect. 7 establishes equivalence of a certain CFL-reachability analysis and the type-based analysis.

6.1 Type Qualifiers

FlowCFL makes use of the pos and neg type qualifiers that we introduced in Sect. 2.1:

- pos — a pos variable x is a source or x is such that a source flows to x. A pos x or any of its components cannot flow to a sink. For example

  $$\mathsf{y = x.f}$$

  where y is a sink, is not allowed. Similarly,

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

  where z is a sink and id is the identity function, is not allowed.
- neg — a neg variable x is a sink, or x flows to a sink.
- poly — a poly variable expresses polymorphism. In some contexts, poly is interpreted as pos and in other contexts, it is interpreted as neg.

The subtyping hierarchy with poly becomes

$$\mathsf{neg} <: \mathsf{poly} <: \mathsf{pos}$$

It is allowed to assign a poly variable into a pos one, but not the other way around; similarly, it is allowed to assign a neg variable into a poly one, but not the other way around. Subtyping in FlowCFL models flow of values. The poly value is interpreted as either neg or pos, depending on the stack context. It would be safe to assign a neg value into a poly variable (which becomes either neg or pos) without causing flow from pos to neg. However, it would not be safe to assign a pos value into a poly variable because it may become neg, causing flow from a pos to a neg variable.

We define the adaptation operation, analogously to the operation in Sect. 4.3.

$$\begin{array}{cccccc} _ & \rhd & \mathsf{pos} & = & \mathsf{pos} \\ _ & \rhd & \mathsf{neg} & = & \mathsf{neg} \\ q & \rhd & \mathsf{poly} & = & q \end{array}$$

Again, a pos or neg variable remains pos or neg. As in Sect. 4.3 a poly variable takes the value of the *adapter* (i.e., context of adaptation): if the adapter is pos, then poly is interpreted as pos, and if the adapter is neg then poly is interpreted as neg.

To avoid clutter we have used the same notation for the adaptation operator, $\rhd$, as in Sect. 4.3. From now on, we will use $\rhd$ to refer to the above definition (FlowCFL), and we will use $\rhd_{RI}$ to refer to the adaptation operator of reference immutability in the rare occasions it comes into play.

Adaptation adapts fields, formal parameters, and return values according to the context at the field access and method call. The type of a poly field f takes the value of the receiver at the field access. The type of a poly parameter or return is interpreted by adapters at call site i. We elaborate on this shortly.

$$\frac{\Gamma \vdash \mathsf{y} <: \mathsf{x} \in C \qquad \Gamma \vdash \mathsf{x} \text{ is not readonly} \Rightarrow \mathsf{x} <: \mathsf{y} \in C \qquad \Gamma \vdash C \text{ holds}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\textsc{tassign})$$

$$\frac{\Gamma \vdash \mathsf{y} <: \mathsf{x} \rhd \mathsf{f} \in C \qquad \Gamma \vdash \mathsf{x.f} \text{ is not readonly} \Rightarrow \mathsf{x} \rhd \mathsf{f} <: \mathsf{y} \in C \qquad \Gamma \vdash C \text{ holds}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\textsc{twrite})$$

$$\frac{\Gamma \vdash \mathsf{y} \rhd \mathsf{f} <: \mathsf{x} \in C \qquad \Gamma \vdash \mathsf{x} \text{ is not readonly} \Rightarrow \mathsf{x} <: \mathsf{y} \rhd \mathsf{f} \in C \qquad \Gamma \vdash C \text{ holds}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

$$\frac{\begin{array}{c} \textit{typeof}(\mathsf{m}) = \mathsf{this}, \mathsf{p} \rightarrow \mathsf{ret} \\ \Gamma \vdash \mathsf{y} <: q^i_{\mathsf{this}} \rhd \mathsf{this} \in C \qquad \Gamma \vdash \mathsf{x} \rhd_{RI} \mathsf{this} \text{ is not readonly} \Rightarrow q^i_{\mathsf{this}} \rhd \mathsf{this} <: \mathsf{y} \in C \\ \Gamma \vdash \mathsf{z} <: q^i_{\mathsf{p}} \rhd \mathsf{p} \in C \qquad \Gamma \vdash \mathsf{x} \rhd_{RI} \mathsf{p} \text{ is not readonly} \Rightarrow q^i_{\mathsf{p}} \rhd \mathsf{p} <: \mathsf{z} \in C \\ \Gamma \vdash q^i_{\mathsf{ret}} \rhd \mathsf{ret} <: \mathsf{x} \in C \qquad \Gamma \vdash \ \text{ is not readonly} \Rightarrow \mathsf{x} <: q^i_{\mathsf{ret}} \rhd \mathsf{ret} \in C \\ \Gamma \vdash C \text{ holds} \end{array}}{\Gamma \vdash i : \mathsf{x} = \mathsf{y.m(z)}}\ (\textsc{tcall})$$

Fig. 7. Typing rules associated to program statements. $\Gamma \vdash A$ means $C, \sigma \vdash A$ as $\Gamma = \langle C, \sigma \rangle$. "$C$ holds" requires (1) that C is closed under the rules from Fig. 8 and (2) that all constraints in C type check.

6.2 Typing Rules

The typing rules for program statements appear in Fig. 7. The rules are defined in terms of a type environment Γ, which is standard. $\Gamma = \langle C, \sigma \rangle$, where C is a set of subtyping constraints and σ is a map from program variables to type qualifiers: $\sigma : V \longrightarrow \{\mathsf{pos}, \mathsf{poly}, \mathsf{neg}\}$. The premise of each rule in Fig. 7 consists of two parts: one part adds constraints to C, and the other part, "C holds", enforces those constraints. Concretely, "C holds" requires (1) that C is closed under the rules in Fig. 8 and (2) that assignment of qualifiers to variables is such that all subtyping constraints in C hold.

Rule (tassign) adds constraint $\mathsf{y} <: \mathsf{x}$ to C, which forbids assignment of a pos or poly reference to a neg one as well as assignment of a pos reference to a poly one. Again, we abuse notation by eliding qualifiers. Strictly, the above constraint should have been written as $q_{\mathsf{y}} <: q_{\mathsf{x}}$ where $q_{\mathsf{y}} = \Gamma(\mathsf{y})$ and $q_{\mathsf{x}} = \Gamma(\mathsf{x})$; rules are more compact while still clear. If x is not readonly according to reference immutability, (tassign) adds the *inverse* constraint $\mathsf{x} <: \mathsf{y}$ as well. In other words, the expected subtyping constraint turns into an equality constraint. This is a well-known issue, typically referred to as the problem of covariant arrays [Bank et al. 1997], which stipulates that subtyping is unsafe in the presence of mutable references. The standard solution, adopted by the majority of systems, e.g., EnerJ [Sampson et al. 2011], is to impose equality constraints for *all references.* This is akin to the bidirectional flow graph in Sect. 4. Our solution combines the "bidirectional" system with reference immutability to achieve limited subtyping and better precision.

One immediately notices the parallel with CFL-reachability. $\mathsf{y} <: \mathsf{x}$ corresponds to forward edge $\mathsf{y} \xrightarrow{\mathsf{d}} \mathsf{x}$ in G_{RI} and the inverse constraint $\mathsf{x} <: \mathsf{y}$ corresponds to the inverse edge $\mathsf{x} \xrightarrow{\mathsf{d}} \mathsf{y}$. The same reasoning applies to all other explicit and implicit assignments: if the left-hand-side is readonly then the rule enforces a subtyping constraint, otherwise it adds an inverse constraint, thus ensuing an equality just as in Sect. 4.

Rules (TWRITE), (TREAD) and (TCALL) use adaptation. At field accesses y.f, field f is interpreted in the context of receiver y. If f is pos (or neg in the positive setting), then its adapted value remains pos (or neg). If f is poly, then the adapted value assumes the type of y. Notably, FlowCFL restricts the type of f to {pos, poly} in the negative setting, and to {poly, neg} in the positive setting. In our discussion going forward, we assume the negative setting (as described in Sect. 2.1), however all reasoning applies to the symmetric positive setting as well. We can allow neg fields in FlowCFL. However, we are interested in type inference, and allowing neg fields would create ambiguity: if x.f flows to neg, (1) do we infer that field f is neg, and is "special", i.e., it is excluded from the state of a potentially pos x, or (2) do we infer that f is just a "regular" field, and a negative x.f entails a neg x? Restricting fields to {pos, poly} chooses the latter, as there is no way to know, without programmer annotations, which fields are "special". Inference tools such as Javarifier [Quinonez et al. 2008] and ReImInfer [Huang et al. 2012b] make the same choice. The restriction states that a positive reference x cannot have negative components.

Rules (TWRITE) and (TREAD) handle heap-transmitted dependences. Consider

$$\mathsf{x.f = a;} \quad \mathsf{y = x;} \quad \textbf{neg}\ \mathsf{b = y.f}$$

Rules (TWRITE), (ASSIGN), and (TREAD) create constraints

$$\mathsf{a} <: \mathsf{x} \rhd \mathsf{f} \quad \mathsf{x} <: \mathsf{y} \quad \mathsf{y} \rhd \mathsf{f} <: \mathsf{b}$$

Since b = neg, f is poly and we have

$$\mathsf{a} <: \mathsf{x} \quad \mathsf{x} <: \mathsf{y} \quad \mathsf{y} <: \mathsf{b}$$

which forces a = neg, as needed. Notice again the parallel with CFL-reachability. Constraints

$$\mathsf{a} <: \mathsf{x} \rhd \mathsf{f} \quad \mathsf{x} <: \mathsf{y} \quad \mathsf{y} \rhd \mathsf{f} <: \mathsf{b}$$

correspond to the path in G_{RI}

$$\mathsf{a} \xrightarrow{w_\mathsf{f}} \mathsf{x} \xrightarrow{\mathsf{d}} \mathsf{y} \xrightarrow{r_\mathsf{f}} \underline{\mathsf{b}}$$

and the "linear" constraints

$$\mathsf{a} <: \mathsf{x} \quad \mathsf{x} <: \mathsf{y} \quad \mathsf{y} <: \mathsf{b}$$

correspond to dropping the w_f and r_f annotations, thus achieving a CSFI approximation. FlowCFL is a more precise variant of CSFI as we argue in Sect. 7.

Rule (TCALL) accounts for call-transmitted dependences and is the most involved. Unlike previous systems, e.g., EnerJ and DroidInfer, FlowCFL allows for *distinct* adapters. Every parameter/return has a distinct associated adapter q^i_{this}, q^i_{p} and q^i_{ret} instead of a single per-call-site adapter q^i. This is necessary to achieve the CFL-reachability semantics. We elaborate on this shortly.

Before we delve into (TCALL), we consider the rules in Fig. 8. The rules explicitly collect transitive intraprocedural constraints into C; they account for constraints that correspond to call/ret balanced paths (i.e., M-paths). (ERASE-LEFT) and (ERASE-RIGHT) "linearize" constraints—e.g., when f is poly, $\mathsf{y} \rhd \mathsf{f} <: \mathsf{b}$ becomes $\mathsf{y} <: \mathsf{b}$. This corresponds to dropping field w, r annotations, thus achieving a variant of CSFI. Notably, a constraint is linearized only if the corresponding field is poly, which happens only when the field is on a path to a sink.

Rule (TRANS-CALL) in Fig. 8 transfers constraints from the callee to the caller. If there is flow from a parameter p to return ret, captured by subtyping constraint $\mathsf{p} <: \mathsf{ret} \in C$, then there is flow from actual argument z to the left-hand-side of the call assignment x.

Consider the example below, which is similar to Fig. 3:

$$\frac{\Gamma \vdash x \rhd f <: y \in C \qquad \Gamma \vdash \sigma(f) = \text{poly}}{\Gamma \vdash x <: y \in C} \;(\text{ERASE-LEFT})$$

$$\frac{\Gamma \vdash y <: x \rhd f \in C \qquad \Gamma \vdash \sigma(f) = \text{poly}}{\Gamma \vdash y <: x \in C} \;(\text{ERASE-RIGHT})$$

$$\frac{\Gamma \vdash x <: y \in C \qquad \Gamma \vdash y <: z \in C}{\Gamma \vdash x <: z \in C} \;(\text{TRANS-LOCAL})$$

$$\frac{\Gamma \vdash z <: q^i_{\mathsf{p}} \rhd \mathsf{p} \in C \qquad \Gamma \vdash q^i_{\mathsf{ret}} \rhd \mathsf{ret} <: x \in C \qquad \Gamma \vdash \mathsf{p} <: \mathsf{ret} \in C}{\Gamma \vdash z <: x \in C} \;(\text{TRANS-CALL})$$

Fig. 8. Constraint propagation. p and ret in (TRANS-CALL) can be any of this, p, or ret. Recall that Γ includes the set of subtype constraints C in addition to the standard map σ from variables to type qualifiers.

```
1  class A {
2    poly B f;
3    void set(poly A this, poly B p) {
4      this.f = p;
5    }
6    poly B get(poly A this) {
7      ret = this.f;
8      return ret;
9    }
10 }
```

```
1  main() {
2    neg A e = new A;   [o]
3    B a = new B;
4    e.set(a); // q^4_this = q^4_p = neg
5    neg A g = e;
6    neg B b = g.get(); // q^6_this = q^6_ret = neg
7  }
```

Class A is polymorphic and main uses A in a negative context. As b is a sink, it follows that a flows to a sink and the types should properly reflect the flow. Line 4 in A.set results in constraint p <: this ⊳ f. Since f is poly, (ERASE-RIGHT) in Fig. 8 produces p <: this. Call site 4 in main entails

$$a <: q^4_{\mathsf{p}} \rhd \mathsf{p} \quad e <: q^4_{\mathsf{this}} \rhd \mathsf{this} \quad q^4_{\mathsf{this}} \rhd \mathsf{this} <: e$$

The last constraint is the inverse of the previous one due to the mutation of this. Rule (TRANS-CALL) in Fig. 8 combines constraints

$$a <: q^4_{\mathsf{p}} \rhd \mathsf{p} \quad \mathsf{p} <: \mathsf{this} \quad q^4_{\mathsf{this}} \rhd \mathsf{this} <: e$$

to get a <: e. Analogously, constraint this ⊳ f <: ret in get and call site 6 in main yield g <: b. Constraints a <: e, e <: g (due to g = e) and g <: b capture the flow from a to b.

6.3 FlowCFL⁻

The reader may wonder, why not use a simpler rule for calls? A natural question arises, why not use the following simpler (TCALL)?

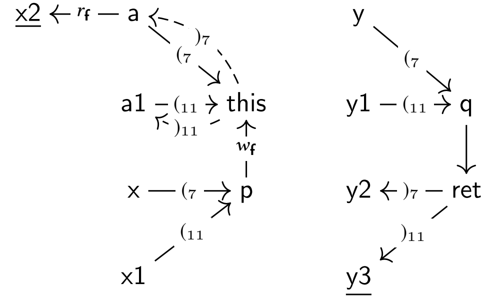
$$\frac{\begin{array}{c}\text{(TCALL)}\\ \mathit{typeof}(\mathsf{m}) = \mathsf{this}, \mathsf{p} \longrightarrow \mathsf{ret} \\ \Gamma \vdash \mathsf{y} <: q^i \rhd \mathsf{this} \qquad \Gamma \vdash \mathsf{x} \rhd_{RI} \text{ this is not readonly} \Rightarrow q^i \rhd \mathsf{this} <: \mathsf{y} \\ \Gamma \vdash \mathsf{z} <: q^i \rhd \mathsf{p} \qquad \Gamma \vdash \mathsf{x} \rhd_{RI} \text{ p is not readonly} \Rightarrow q^i \rhd \mathsf{p} <: \mathsf{z} \\ \Gamma \vdash q^i \rhd \mathsf{ret} <: \mathsf{x} \qquad \Gamma \vdash \mathsf{x} \text{ is not readonly} \Rightarrow \mathsf{x} <: q^i \rhd \mathsf{ret} \end{array}}{\Gamma \vdash i : \mathsf{x} = \mathsf{y.m(z)}}$$

The rule makes use of a single viewpoint adapter q^i rendering C and the rules in Fig. 8 unnecessary! Replacing (TCALL) in Fig. 7 with the above (TCALL) yields a new type system, which we call FlowCFL$^-$ (FlowCFL minus). An advantage of FlowCFL$^-$ is its simplicity; it is also sound, however, it rejects programs that CFL-reachability over G_{RI} handles precisely.

The following (somewhat contrived) example illustrates the imprecision of FlowCFL$^-$ and the need for multiple adapters:

```
1   poly Y m(poly A this, poly X p, poly Y q) {
2     this.f = p;
3     ret = q;
4   }
5   ...
6   A a, X x, Y y;
7   Y y2 = a.m(x,y);
8   neg X x2 = a.f;

10  A a1, X x1, Y y1;
11  neg Y y3 = a1.m(x1,y1);
```



There are two disjoint paths through m: (1) p $\rightsquigarrow$ this, and (2) q $\rightsquigarrow$ ret. At call 7 the first path appears in negative context (as subpath of the path from x to neg x2), while the second path appears in positive context (as subpath of the path from y to y2). At call 11 the opposite happens: the first path appears in positive context and the second one in negative context. CFL-reachability precisely propagates the negative qualifier neg x2 back to x and neg y3 back to y1. FlowCFL propagates the negative qualifiers in exactly the same way. It discovers paths x $\rightsquigarrow$ x2 and y1 $\rightsquigarrow$ y3 via C: it adds x <: x2 and y1 <: y3 to C based on p <: this and q <: ret respectively; it adds no spurious constraints (i.e., paths).

In contrast, with a single adapter q^i (e.g., as in DroidInfer and EnerJ) the above precise typing is impossible. This is because the role of the adapter is twofold: (1) to interpret the poly parameter/return in the corresponding context, and (2) to propagate paths from callee to caller. Given sink neg X x2 in line 8, field f is poly (due to the flow of f to sink x2). This forces this and p of m to poly, as shown in the typing of m in lines 1-4. Sink neg Y y2 in line 11 forces ret and q to poly as well. Due to $q^7 \rhd$ this <: a and $q^{11} \rhd$ ret <: y3 respectively, we have q^7 = neg and q^{11} = neg. Thus, y <: $q^7 \rhd$ q and a1 <: $q^{11} \rhd$ this unnecessarily force y and a1 to neg.

Multiple adapters differentiate between flow paths. This is because the purpose of the adapters is only to interpret the poly parameter/return in the corresponding context; propagation of paths from calee to caller is done via C. In our example we have a1 <: $q^{11}_{\mathsf{this}} \rhd$ this, x1 <: $q^{11}_{\mathsf{p}} \rhd$ p, y1 <: $q^{11}_{\mathsf{q}} \rhd$ q, and $q^{11}_{\mathsf{ret}} \rhd$ ret <: y3. $q^{11}_{\mathsf{this}} = q^{11}_{\mathsf{p}}$ = pos, and $q^{11}_{\mathsf{ret}} = q^{11}_{\mathsf{q}}$ = neg. The qualifiers are flipped at call site 7: $q^7_{\mathsf{this}} = q^7_{\mathsf{p}}$ = neg, and $q^7_{\mathsf{ret}} = q^7_{\mathsf{q}}$ = pos.

7 EQUIVALENCE OF CFL-REACHABILITY AND TYPE-BASED ANALYSES

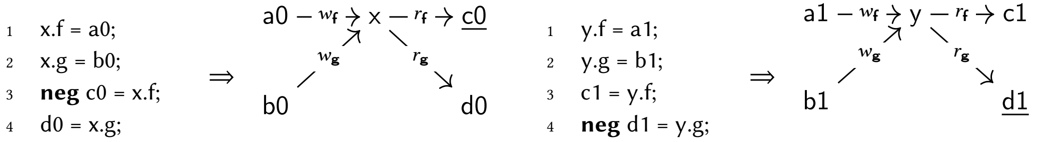
Recall that CFL-reachability over both w, r and call/ret annotations is undecidable. Typical approximations are CSFI (context-sensitive, field-insensitive) and CIFS, and variants in-between. FlowCFL implements a variant of CSFI, which we call CSFI$^+$. DroidInfer implements and evaluates the CSFI$^+$ variant [Huang et al. 2015]. DroidInfer's extensive empirical results justify and motivate CSFI$^+$ and FlowCFL, and allow us to focus on the theoretical results. FlowCFL at this stage is a generalization of DroidInfer as it supports a fixed semantics of flow, namely CSFI$^+$. In the future, we plan to add customizability to support different approximation of CFL-reachability, e.g., CIFS or CIFI.

As mentioned earlier, CSFI replaces all field annotations with d and performs CR-reachability. E.g., in

$$\mathsf{x.f = a;}\ \ \textbf{neg}\ \mathsf{b = x.g;} \quad \Rightarrow \quad \mathsf{a} \xrightarrow{\mathsf{d}} \mathsf{x} \xrightarrow{\mathsf{d}} \underline{\mathsf{b}}$$

CSFI replaces w_{f} and r_{g} with d and spuriously propagates neg b back to a. Another way to look at CSFI is as if we replaced production $B \longrightarrow w_{\mathsf{f}}\ B\ r_{\mathsf{f}}$ in the PG grammar in Fig. 5(b) with $B \longrightarrow w_{\mathsf{f}}\ B\ r_{\mathsf{g}}$.

CSFI$^+$ does match certain distinct field annotations w_{f} and r_{g}, but not all; in contrast, CSFI matches all field annotations w_{f} and r_{g}. CSFI$^+$ matches w_{f} and r_{g} only if fields f and g both flow to sinks. As an example of potential imprecision in CSFI$^+$ consider the two snippets of the same program:



Since both f and g flow to sinks, CSFI$^+$ matches the distinct field annotations. It propagates negative c0 to both a0 and b0. Similarly, it propagates negative d1 to both a1 and b1.

The problem is to find a set of paths that includes *all* properly matched w/r and call/ret paths to sinks in G_{RI}. Without loss of generality we assume that sinks are primitive types, i.e., the PG-component of every path from v to a sink is either a G-path or a B-path. CSFI$^+$ back-propagates sinks maintaining a set F of fields that flow to sinks. The difference between precise propagation, CSFI$^+$, and CSFI lies in production $B \longrightarrow w_{\mathsf{f}}\ B\ r_{\mathsf{g}}$. Precise propagation infers a B-path when f = g (as in Fig. 5(b)), CSFI$^+$ infers a B-path only when $\{\mathsf{f}, \mathsf{g}\} \in F$, and CSFI infers a B-path in all cases.

Fig. 9 presents two equivalent implementations of CSFI$^+$. Both algorithms implement FlowCFL in the negative setting—they start from a set of sinks and back-propagate those sinks via CSFI$^+$-reachability. Algorithm Cfl collects all paths from variables to sinks in P, as well as all balanced subpaths of these paths (M-paths). One can easily show (by induction on the length of the path) that Cfl computes in P all properly matched paths in G_{RI}. Algorithm Types makes use of the type system in Sect. 6. It computes a map S from program variables to sets of qualifiers. S is initialized as follows: $S(u) = \{\mathsf{neg}\}$ for each sink u, $S(\mathsf{x}) = \{\mathsf{pos}, \mathsf{poly}, \mathsf{neg}\}$ for each variable x, and $S(\mathsf{f}) = \{\mathsf{pos}, \mathsf{poly}\}$ for each field f. Types iterates through program statements; it infers new "linear" constraints and removes infeasible qualifiers from variable sets until it reaches a fixed point. Function Solve takes a constraint, e.g., $\mathsf{x} <: \mathsf{y}$ and updates $S(\mathsf{x})$. E.g., if $S(\mathsf{y}) = \{\mathsf{neg}\}$ and $S(\mathsf{x}) = \{\mathsf{pos}, \mathsf{poly}, \mathsf{neg}\}$, Solve removes pos and poly from $S(\mathsf{x})$ because neither is a subtype of neg. As another example, consider constraint $\mathsf{x} \rhd \mathsf{f} <: \mathsf{y}$ where $S(\mathsf{y}) = \{\mathsf{poly}, \mathsf{neg}\}$, $S(\mathsf{x}) = \{\mathsf{pos}, \mathsf{poly}, \mathsf{neg}\}$, and $S(\mathsf{f}) = \{\mathsf{pos}, \mathsf{poly}\}$. Solve removes pos from both $S(\mathsf{x})$ and $S(\mathsf{f})$ because the constraint cannot be satisfied if either x or f is pos. Such fixpoint iteration has been used in previous work [Huang et al. 2012a; Kiezun et al. 2007; Tip et al. 2011].

```
1:  procedure Cfl
2:    P = ∅, F = ∅
3:    Add n ~M~> n to P for all sinks n
4:    while P or F changes do
5:      for each s in Program do
6:        Edge(forward(s))
7:        Edge(inverse(s))
8:      end for
9:    end while
10: end procedure
```

```
1:  procedure Edge(x -t-> y) // x -t-> y ∈ G_RI
2:    if y ∈ {ret, this, p} then Add y ~M~> y to P
3:    for each y ~N~> n ∈ P do
4:      case t, N of
5:        d, _   ->  Add x ~N~> n, x ~M~> y to P
6:        r_f, _  ->  Add x ~N~> n, x ~M~> y to P, Add f to F
7:        w_f, _  ->  if f ∈ F then Add x ~N~> n, x ~M~> y to P
8:        )_i, _  ->  Add x ~R~> n to P
9:        (_i, M/C  ->  Add x ~C~> n to P
10:       (_i, R  ->  for each y ~M~> ret -)_i-> z ~N'~> n ∈ P do
11:                       Add x ~N'~> n, x ~M~> z to P
12:                   end for
13:   end for
14:   for each x ~M~> y ∈ P do
15:     for each y ~M~> ret ∈ P do
16:       Add x ~M~> ret to P
17:     end for
18:   end for
19: end procedure
```

```
1:  procedure Types
2:    S(n) = {neg} for all sinks n
3:    S(n) = {neg, poly, pos}
4:    while S or C changes do
5:      for each s in Program do
6:        Constraint(forward(s))
7:        Constraint(inverse(s))
8:      end for
9:    end while
10: end procedure
```

```
1:  procedure Constraint(c)
2:    Solve(c)
3:    case c of
4:      x <: y  ->  Add x <: y to C
5:      x ▷ poly <: y  ->  Add x <: y to C
6:      x <: y ▷ poly  ->  Add x <: y to C
7:      q^i_ret ▷ ret <: x  ->  -
8:      x <: q^i_p ▷ p  ->
9:        for each p <: ret ∈ C,
10:                q^i_ret ▷ ret <: z ∈ C do
11:         Add x <: z to C
12:         Solve(x <: z)
13:       end for
14:   for each x <: y ∈ C do
15:     for each y <: ret ∈ C do
16:       Add x <: ret to C
17:       Solve(x <: ret)
18:     end for
19:   end for
20: end procedure
```

Fig. 9. Algorithms Cfl and Types assume a set of user-defined sinks. Cfl computes P, which collects all paths $x \overset{N}{\rightsquigarrow} n$ from variables to sinks. It iterates over program statements s processing the edges $x \xrightarrow{t} y \in G_{RI}$ and adding paths to P by concatenating the "terminal" annotation t and the "nonterminal" annotation N according to the rules of the CR context-free grammar in Fig. 5. Types initializes S, then iterates over program statements s removing qualifies from S and collecting constraints in C. The algorithms elide details to highlight the "parallel" structure of the two systems.

Types assigns sets of qualifiers to variables. To assign a final typing to a variable/field, we pick the *maximal qualifier* according to preference ranking pos > poly > neg. One can see through case by case analysis that the *maximal typing* type checks with the rules in Fig. 7 and Fig. 8. Qualifiers q^i_{this}, q^i_p, q^i_{ret} can take any value that satisfies the maximal typing.

We argue correctness of our type-based analysis by establishing equivalence between Cfl and Types. Def. 7.1 states that if there is an M/C-path from x to a sink n, then the maximal typing of x in S is neg. If there is an R-path, the maximal typing is poly or neg.

Definition 7.1. (Soundness) $P \Rightarrow S$ if and only if

$$\begin{array}{lll} 1. & \mathsf{x} \overset{M/C}{\rightsquigarrow} n \in P & \Rightarrow \quad max(S(\mathsf{x})) <: \mathsf{neg} \\ 2. & \mathsf{x} \overset{R}{\rightsquigarrow} n \in P & \Rightarrow \quad max(S(\mathsf{x})) <: \mathsf{poly} \end{array}$$

Def. 7.2 states that x's maximal type in S implies a corresponding path in P. For example, maximal typing poly means that there is a R-path but there is no M/C-path.

Definition 7.2. (Precision) $S \Rightarrow P$ if and only if

$$\begin{array}{llll} 1. & max(S(\mathsf{x})) = \mathsf{neg} & \Rightarrow & \exists\, \mathsf{x} \overset{M/C}{\rightsquigarrow} n \in P \\ 2. & max(S(\mathsf{x})) = \mathsf{poly} & \Rightarrow & \exists\, \mathsf{x} \overset{R}{\rightsquigarrow} n \in P \ \wedge\ \nexists\, \mathsf{x} \overset{M/C}{\rightsquigarrow} n \in P \\ 3. & max(S(\mathsf{x})) = \mathsf{pos} & \Rightarrow & \text{no path from x to any } n \text{ in } P \end{array}$$

Definition 7.3. (Equivalence) $P \simeq S$ if and only if $P \Rightarrow S$ and $S \Rightarrow P$.

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

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

The equivalence result comes from the following theorem:

Theorem 7.4. *If* $P \simeq S$ *and* $\{P, S\}$ $\textsc{Edge}(e(s)) \ || \ \textsc{Constraint}(c(s))$ $\{P', S'\}$ *then* $P' \simeq S'$.

Our technical report [Milanova 2020] gives a proof sketch.

8 RELATED WORK

CFL-reachability dates decades back [Reps 2000; Reps et al. 1995], yet it remains highly relevant. Zhang and Su [2017], Späth et al. [2019], and Chatterjee et al. [2018], among other works, present novel CFL-reachability approximations and algorithms with application to data dependence. Xu et al. [2009], and Lu and Xue [2019], again among other works, present novel CFL-reachability-based points-to analyses. In all works, the concept of the inverse edge, introduced by Sridharan et al. [2005], factors in. Our work presents a formal treatment of the inverse edges and paths and a correctness argument for CFL-reachability over graphs with inverse edges. Recent work by Li et al. [2020] presents a graph simplification algorithm for CFL-reachability that removes certain edges that do not contribute to paths to sinks. This work nicely complements our work, as it can be applied on any CFL-reachability graph, including G_{BI} and G_{RI}; Li et al., demonstrate their technique using DroidInfer's graphs [Huang et al. 2015], which are G_{RI} graphs. (We use DroidInfer's graphs in our experiments as well.) We have focused on understanding the dynamic semantics of flows, establishing soundness of the removal of certain inverse edges, and drawing a connection between CFL-reachability and type-based flow analysis.

The idea that reference immutability can be used to improve type-based flow analysis dates back to Milanova and Huang [2013], to the best of our knowledge. Milanova et al. [2014], and Huang et al. [2015] explore a connection of DroidInfer to CFL-reachability and suggest, in an ad-hoc way, the construction of G_{RI} graphs from the type constraints of DroidInfer. However, a proof that G_{RI} is correct has not been accomplished before. The key novelty of this paper is the general treatment and proofs, which became possible because (1) we built a suitable dynamic semantics (Sect. 3), and (2) a recent result cast ReIm in terms of CFL-reachability and built a "recipe" for showing equivalence ([Milanova 2018]).

Zhang and Su [2017] and recently Li et al. [2020] develop novel CFL-reachability techniques. They validate the techniques by applying them on input graphs constructed from the type constraints of DroidInfer; these are G_{RI} graphs. Interestingly, Zhang and Su's Linear Conjunctive Reachability

works better when applied on the G_{RI} graphs than on G_{BI} graphs (constructed by a different tool in a different problem domain rather than taint analysis) [ref. [Zhang and Su 2017], p. 11, Section 6.3]. Li et al. [2020] validate on DroidInfer's G_{RI} graphs. We view their works as additional motivation for the study of $G_{RI} - G_{RI}$ not only directly improves precision of CFL-reachability analysis, but it appears to enable more advanced and powerful techniques.

Type-based analysis has a long history as well [Palsberg 2001] and our analysis falls into this line of work. Classical work on type-based taint (information flow) analysis includes work by Shankar et al. [2001], Volpano et al. [1996], and Myers [1999]. The pos/neg qualifiers are standard and date back to at least Volpano et al. [1996]. A distinguishing feature of FlowCFL is the handling of mutable heap data, which is not handled in the classical system by Volpano et al. [1996]. On the other hand, the classical type systems handle implicit flows and side channels, which we do not.

Rehof and Fähndrich [2001] connect type-based flow analysis and CFL-reachability. However, Rehof and Fahndrich do not discuss mutable references and it is unclear how their analysis and interpretation, targeting a pure functional language, can handle mutable data and heap-transmitted dependences. On the other hand, Rehof and Fahndrich handle higher-order functions while we do not. An important direction of future work is extending our approach with handling of higher-order functions which will enable application of the FlowCFL framework to the analysis of dynamic languages. Fähndrich et al. [2000] apply the theory of Rehof and Fähndrich [2001] 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 mutable references [Fuhrer et al. 2005; Sampson et al. 2011; Shankar et al. 2001].

9 CONCLUSION AND FUTURE WORK

We presented FlowCFL, a framework for type-based reachability analysis. We presented (1) a novel dynamic semantics, (2) correctness arguments for CFL-reachability over graphs with inverse edges, and (3) equivalence between a CFL-reachability analysis and a type-based reachability analysis.

There are several directions of future work. First, we believe that G_{RI} can be useful in CFL-reachability analysis. One can implement CFL-reachability analysis over G_{RI}, instead of what is now the standard, over G_{BI}. We conjecture that the graph reduction G_{RI} entails, will improve analysis precision and enable advanced CFL-reachability techniques such as those described in [Zhang and Su 2017] and [Li et al. 2020]. We will implement and experiment with a variety of analyses towards this conjecture.

In other directions, we are particularly interested to develop principled approaches that explain type errors in terms of CFL-reachability paths. We will begin with investigation of FlowCFL, however, we believe that CFL-reachability can be used with other, more widely-known type systems as well, for example, Hindley-Milner. Finally, as mentioned earlier, we will extend customizability of FlowCFL to support other variants of CFL-reachability approximation, e.g., context-insensitive but field-sensitive analysis, i.e., CIFS. We conjecture that the graph reduction G_{RI} entails will impact precision of other CFL-reachability approximations.

ACKNOWLEDGEMENTS

We thank the OOPSLA 2020 and the ECOOP 2020 reviewers whose detailed comments and suggestions helped improve this paper significantly. The author is supported by NSF grant 1814898.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis

for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

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

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013a. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, New York, NY, USA, 33–52. https://doi.org/10.1145/2509136.2509546

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013b. Verifying quantitative reliability for programs that execute on unreliable hardware (Appendix). http://groups.csail.mit.edu/pac/rely/rely13appendix.pdf.

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck reachability for data-dependence and alias analysis. *PACMPL* 2, POPL (2018), 30:1–30:30. https://doi.org/10.1145/3158118

Werner Dietl, Michael D. Ernst, and Peter Müller. 2011. Tunable Static Inference for Generic Universe Types. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings (Lecture Notes in Computer Science)*, Mira Mezini (Ed.), Vol. 6813. Springer, Berlin, Heidelberg, 333–357. https://doi.org/10.1007/978-3-642-22655-7_16

Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. 2012. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems* 34, 1 (April 2012), 1–48. https://doi.org/10.1145/2160910.2160913

Yao Dong, Ana Milanova, and Julian Dolby. 2016. JCrypt: Towards Computation over Encrypted Data. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. ACM, New York, NY, USA, 8:1–8:12. https://doi.org/10.1145/2972206.2972209

Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward XueJun Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, New York, NY, USA, 1092–1104. https://doi.org/10.1145/2660267.2660343

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. 2000. Scalable Context-sensitive Flow Analysis Using Instantiation Constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 253–263. https://doi.org/10.1145/349299.349332

Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2013. Highly precise taint analysis for Android applications. EC SPRIDE Technical Report TUD-CS-2013-0113. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf.

Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. 2005. Efficiently Refactoring Java Applications to Use Generic Libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 71–96. https://doi.org/10.1007/11531142_4

Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. 2012a. Inference and Checking of Object Ownership. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 181–206. https://doi.org/10.1007/978-3-642-31057-7_9

Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-Based Taint Analysis for Java Web Applications. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Stefania Gnesi and Arend Rensink (Eds.), Vol. 8411. Springer, Berlin, Heidelberg, 140–154. https://doi.org/10.1007/978-3-642-54804-8_10

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012b. Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 879–896. https://doi.org/10.1145/2384616.2384680

Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. 2007. Refactoring for Parameterizing Java Classes. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 437–446. https://doi.org/10.1109/ICSE.2007.70

Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, New York, NY, USA, 780–793. https://doi.org/10.1145/3385412.3386021

Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *PACMPL* 3, OOPSLA (2019), 148:1–148:29. https://doi.org/10.1145/3360574

Ana Milanova. 2018. Definite Reference Mutability. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs)*, Todd D. Millstein (Ed.), Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:30. https://doi.org/10.4230/LIPIcs.ECOOP.2018.25

Ana Milanova. 2020. FlowCFL: A Framework for Type-based Reachability Analysis in the Presence of Mutable Data. (2020). https://arxiv.org/abs/2005.06496

Ana Milanova and Wei Huang. 2013. Composing polymorphic information flow systems with reference immutability. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*. ACM, New York, NY, USA, 5:1–5:7. https://doi.org/10.1145/2489804.2489809

Ana Milanova, Wei Huang, and Yao Dong. 2014. CFL-reachability and context-sensitive integrity types. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, Joanna Kolodziej and Bruce R. Childers (Eds.). ACM, New York, NY, USA, 99–109. https://doi.org/10.1145/2647508.2647522

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. ACM, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Jens Palsberg. 2001. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, John Field and Gregor Snelting (Eds.). ACM, New York, NY, USA, 20–27. https://doi.org/10.1145/379605.379635

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

Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 54–66. https://doi.org/10.1145/360204.360208

Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

Thomas W. Reps. 2000. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/1993498.1993518

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1267612.1267628

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *PACMPL* 3, POPL (2019), 48:1–48:29. https://doi.org/10.1145/3290361

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems* 33, 3 (April 2011), 1–47. https://doi.org/10.1145/1961204.1961205

Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 211–230. https://doi.org/10.1145/1094811.1094828

Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. 2010. A Type System for Data-centric Synchronization. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 304–328. http://dl.acm.org/citation.cfm?id=1883978.1884000

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3009837.3009848