Data Flow Analysis in the Presence of Correlated Calls

by

Marianna Rapoport

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis presents a technique to improve the precision of data-flow analyses on object-oriented programs in the presence of *correlated calls*. We say that two method calls are correlated if they are polymorphic (have multiple targets) and are invoked on the same object. Correlated calls are problematic because they can make existing data-flow analyses consider certain infeasible data-flow paths as valid. This leads to loss in precision of the analysis solution.

We show how infeasible paths can be eliminated for *Inter-procedural Finite Distributive Subset* (IFDS) problems, a large class of data-flow analysis problems. We show how the precision of IFDS problems can be improved in the presence of correlated calls, by using the *Inter-procedural Distributive Environment* (IDE) algorithm to eliminate infeasible paths. Using IDE, we eliminate the infeasible paths and obtain a more precise result for the original IFDS problem.

Our analysis is implemented in Scala, using the WALA framework for static program analysis on Java bytecode.

Acknowledgements

I want to thank all who have supported me in writing this thesis:

Ondřej Lhoták for his patience and the generosity with which he invested so much time in advising me; for the meticulous approach to the project and correction of my work. Every time I left Ondřej's office I had learned something new.

Frank Tip for the idea of correlated calls, steering the project in the right direction (while keeping the details in scope), and encouraging me to start a PhD.

Prabhakar Ragde who accepted me to Waterloo, for the invaluable freedom and support in all my undertakings, and also for the comments and suggestions to improve this thesis, together with Werner Dietl.

Cosmin Rădoi and Julian Dolby, for their responsiveness and help with the WALA framework.

Karim Ali, Magnus Madsen, and Nomair Naeem, for the discussions that helped me understand program analysis.

Oleg Sobolev for the large effort he put into teaching me how to program; for making me care about program quality and good design, and also for introducing me to functional programming. Thanks to Oleg I am now here.

Josh Hoyt and Eric Conlon, for making me a better programmer.

Sergei Kovalev and Yuri Golovin, for supporting me in moving to Canada.

David Dufour, Cecylia Bocovich, Stephen Kiazyk, Oliver Trujillo, Laura Cang, Valerie Sugarman, Dean Shaft, Hella Hoffmann, Jack Thomas, Liuda Mesropyan, and Kirill Zorin, for making the last two years a fun time.

Lena Khomyakova and Alexey Khomyakov, for their selfless readiness to help.

Olga Zorin and Nika Haghtalab, for their support and friendship.

The loving Lisa Evteeva, Kirill Chernyshov, and Lena Tardova, who are behind the ocean but always on my side.

My family, for the best in me.

Abel, for being my closest and dearest friend.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Static program analysis aims to discover properties of computer programs without running them. Static analysis has applications in compiler optimization, development of programming tools, and computer security, among others. As an example, we might want to analyze a program to know which variables are constants. We could then write a compiler optimization that ensures that the values of those variables are computed only once. Alternatively, we could use the information about constant variables in an integrated development environment; for instance, to notify the user when an if-expression executes only one of its branches because its test condition has a constant value.

There are demonstrable limits on what information we can obtain about a program without running it. Rice's theorem states that verifying any non-trivial property of a program is an undecidable problem [19]. However, it is sometimes possible to design an algorithm that *over-* or *underapproximates* the solution that we are seeking.

Data-flow analysis is an area of program analysis whose goal is to compute approximations of certain information (for example, which variables must be constants) for each program point.

Other examples of data-flow analyses are *reaching definitions* (finding out up to which instruction a given assignment of a variable must be valid) and *available expressions* (retrieving the expressions in the program that do not need to be recomputed at a given program point).

Another example of a data-flow analysis is *taint analysis* [25]. Taint analysis discovers if "secret" values, like passwords or other confidential user information, can leak to an external observer. Methods that generate secret values, e.g. those that read user input,

are called *sources*. Methods that can leak information, e.g. those that write data to a file or send data through a network, are called *sinks*. The goal of taint analysis is to find out whether data can propagate from sources to sinks.

An important property of a data-flow analysis is *precision*. Precision reflects how closely a data-flow-analysis result over- or underapproximates the information we are interested in. In the case of taint analysis, let T be the number of sinks that the analysis considers to leak secret information, and R the real number of potential information leaks. The smaller the difference between T and R, the greater the precision of the taint analysis.

Data-flow analyses operate on *control-flow graphs* that model the order in which the instructions of a program are executed. A data-flow-analysis problem defines *flow functions* that represent how data is propagated along the edges of the control-flow graph. The *confluence operator* specifies how the data that has been computed along different paths should be merged when the paths join.

Since a control-flow graph is an overapproximation of the possible flows of control in concrete executions of a program, the graph may contain *infeasible* paths that cannot occur at runtime.

One way to improve the precision of a data-flow analysis is to detect and eliminate infeasible paths.

The goal of this thesis is to improve the precision of solutions to problems that can be solved by the *Inter-procedural Finite Distributive Subset* (IFDS) algorithm [18]. The IFDS algorithm is a general data-flow algorithm that can compute solutions to various data-flow problems, like reaching definitions, available expressions, and taint analysis.

We improve the precision of IFDS problem solutions by eliminating infeasible paths that occur in object-oriented programs in the presence of *correlated method calls* — polymorphic calls that are invoked on the same object [24].

1.1 Correlated Calls

Consider a call site $r.m()$ in an object-oriented programming language, where the variable r is the *receiver* variable of the call site and m is the name of the invoked method[1]. In the rest of the thesis, we use the general term *receiver* to mean a receiver variable. At runtime, the actual method that will be invoked by the call site depends on the runtime type of the

[1] We assume an internal representation of the program in which for each call site $e_r.m()$, the expression e_r has been evaluated to the variable r.

object referenced by r. If the call site $r.m()$ can be associated with more than one method at compile time, we will say that the call site is *polymorphic*.

For example, in Listing 1.1, it is not possible to infer statically whether the runtime type of the variable `a` in line 17 is `A` or `B`. The call `a.foo()` can be dispatched to either `A.foo` or `B.foo`, and `a.bar(v)` can be dispatched to either `A.bar` or `B.bar`. A concrete execution path for the main method might therefore go through `A.foo` and `A.bar`, or through `B.foo` and `B.bar`. However, there cannot be an execution path through `A.foo` and `B.bar` or through `B.foo` and `A.bar`.

```
1  class A {
2    String foo {
3      return secret();
4    }

5    void bar(String s) {}
6  }

7  class B extends A {
8    String foo {
9      return "not secret";
10   }

11   void bar(String s) {
12     System.out.println(s);
13   }
14 }

15 class Main {
16   public static void main(String[] args) {
17     A a = args == null ? new A() : new B(); // a has runtime type A or B
18     String v = a.foo();
19     a.bar(v);
20   }
21 }
```

Listing 1.1: Example program containing correlated calls

We call the invocations to methods `foo` and `bar` *correlated*. More generally, correlated calls occur when more than one polymorphic call is invoked on the same receiver variable.

Suppose we wanted to perform a taint analysis on the program in Listing 1.1. Most dataflow-analysis algorithms, including IFDS, would conservatively assume that the call

`a.bar` could be dispatched to both `A.bar` and `B.bar`, independently of what `a.foo` had been dispatched to in the previous line.

As a result, such an analysis would consider a path through `A.foo` and `B.bar` feasible. This means that the variable `v` would be considered secret. We would conclude that a secret value is passed to `B.bar` and printed to the user. In other words, we would consider the program to leak secret information, which it does not do in any concrete execution.

Our technique for improving the precision of an IFDS result is based on transforming the original IFDS problem into a more expressive *Inter-procedural Distributive Environment* (IDE) problem. IDE problems can be solved with the IDE algorithm which is a generalization of IFDS [21]. The IDE algorithm can, for instance, solve certain versions of the constant propagation problem that IFDS cannot.

To improve the precision of IFDS results, given an IFDS problem P, we convert it into an IDE problem Q that accounts for correlated calls. We then use the IDE algorithm to obtain a solution to Q. Finally, we convert the IDE result into a IFDS result. In the presence of correlated calls, the obtained IFDS result can be more precise than the solution that the IFDS algorithm would compute for P.

1.2 IFDS and IDE

The IFDS framework is a precise and efficient algorithm for data-flow analysis. IFDS was developed in 1995 by T. Reps, S. Horwitz, and M. Sagiv at the University of Wisconsin and has been used to solve a variety of data-flow analysis problems [4, 14, 11, 25]. The IFDS analysis is a version of the classic *functional approach* to data-flow analysis proposed by M. Sharir and A. Pnueli [22].

Given a data-flow problem that satisfies the restrictions of IFDS, the algorithm provides a *context-sensitive* solution in polynomial time. In other data-flow algorithms not based on the functional approach, the result of the analysis at the entry of a procedure "merges" the incoming data obtained from all callers of the procedure. As a consequence, there is one global data-flow result computed at the end of the procedure. Context-sensitivity, however, allows an analysis to compute the data-flow result for a given procedure as a *function* of the data-flow value at the start of the procedure. In other words, the analysis result for a procedure varies depending on where the procedure was called from. This significantly improves the precision of a data-flow analysis, which is why context-sensitivity is an important advantage of IFDS over classic data-flow algorithms.

Compared to IFDS, most data-flow analyses are either general but do not run in polynomial time [9, 22] or handle a very specific set of problems [10].

The IFDS algorithm is applicable to problems which can be expressed with data-flow functions that satisfy certain restrictions. *Inter-procedural* flow functions specify how data flows from the invocation of a procedure to its start, and from the procedure's end back to its call site. *Distributive* flow functions are those that distribute over the confluence operator. In the context of IFDS, the confluence operator is called meet, and it can be either union or intersection. The data-flow facts on which the analysis operates must be a *finite* set D. Each flow function operates on a *subset* of D (for example, the set of variables in the program) which makes the domain of the flow functions the power set of D. We describe the IFDS restrictions in detail in Section 2.3.1.

The IDE framework is an expressive extension to IFDS that was created by the same authors in 1996. The problems that IDE can solve include, but are not limited to, IFDS problems. Just as the IFDS algorithm, the IDE algorithm is suitable for data-flow analyses that can be encoded with inter-procedural, distributive flow functions. However, in IDE, the domain of the flow functions is not restricted to sets D of data-flow facts. The IDE domain of a flow function consists of *environments* that map data-flow facts from the set D to lattice elements.

As an example, in a constant propagation problem, an IDE environment would map each variable to the (possibly) constant value that it is bound to. To illustrate the distinction between IFDS and IDE we could say that IFDS can find out which variables in a program are constants, whereas IDE can additionally retrieve the values of the constant variables.

1.3 Thesis Outline

The goal of the correlated-calls analysis presented in this work is to modify the output of an IFDS analysis to account for correlated calls. Specifically, the correlated-calls analysis improves the precision of IFDS problem results by eliminating infeasible execution paths caused by correlated calls. This is done by converting the input-IFDS problem to an IDE problem that detects infeasible paths, and converting the IDE result back to a more precise IFDS result.

The contributions of this thesis are:

- A transformation from IFDS to IDE problems that considers correlated calls.

- An implementation in Scala of the correlated-calls transformation and the IDE algorithm which is based on the WALA framework for static analysis on Java bytecode [6].

We prove that the solution to an IDE problem that considers correlated calls is more precise than the solution to the original IFDS problem. We also show that the correlated-calls analysis is sound, i.e. that it never considers concrete execution paths as infeasible.

Finally, we evaluate the effectiveness of the correlated-calls analysis using an implementation of taint analysis as the source IFDS problem.

The remainder of this thesis is organized as follows. In the next chapter, we describe the IFDS and IDE analyses in detail. In Chapter 3 we present the correlated-calls analysis as a transformation of IFDS problems into a special kind of IDE problem. Chapter 4 describes an efficient representation of the data structures that are required to define a correlated-calls IDE transformation. In Chapter 5 we address some implementation aspects of the correlated-calls analysis and present an evaluation of its results. Chapter 6 contains concluding remarks.

Chapter 2

Background

The purpose of the correlated-calls analysis is to solve IFDS problems more precisely than using the standard IFDS algorithm by ruling out some infeasible paths. The correlated-calls analysis works by transforming an IFDS problem to an IDE problem, solving the IDE problem, and transforming the IDE result to a solution to the original IFDS problem. This chapter describes the general ideas underlying IFDS and IDE.

2.1 Related Work

IFDS is a version of the functional approach to data-flow analysis developed by M. Sharir and A. Pnueli [22]. Their algorithm is based on computing *summary functions* that return the data-flow value at the end of a procedure, given the data-flow value at the start of the procedure. IFDS problems form a more restricted set of data-flow problems: unlike in the functional approach, IFDS flow functions have to be distributive, and the set of data-flow facts D has to be finite. However, the IFDS algorithm is more general than Sharir's and Pnueli's algorithm in that it can handle programs containing local variables and parameters in recursive methods.

IFDS has been used to encode a variety of data-flow problems. More complex examples of applications include typestate analysis (determining which operations can be performed on an object at a given program point) [14] or shape analysis (detecting errors and validating properties of programs at compile time) [11].

IFDS is implemented for two popular static-analysis frameworks, the T.J. Watson Libraries for Analysis (WALA) [6] and Soot [26].

WALA is a framework for static analysis on Java bytecode developed by the IBM T.J. Watson Research Center. In the implementation of our work, we use WALA to build and traverse the supergraph (a special kind of control-flow graph) of a Java program[2].

Soot is a framework for program analysis and optimization on Java bytecode, developed by the Sable Research Group at McGill University. Unlike WALA, Soot also has an implementation of the IDE algorithm. The IFDS and IDE implementations for Soot are part of the Heros project [3].

Whereas one advantage of Soot's IFDS implementation (and other static analysis tools) is ease of use and extensibility, WALA's primary focus is efficiency. For example, WALA uses bit-vectors to represent some of the analysis data types, like local variables and parameters. Another difference is that WALA's intermediate representation of a program uses static single assignment (SSA) form [5]. SSA form is a representation of the program in which each variable has only one definition (assignment). SSA can make dataflow analysis simpler and more efficient [1].

Work on improving the IFDS algorithm includes Practical Extensions by N. Naeem and O. Lhoták [15]. Their paper presents four extensions to the IFDS algorithm. Two of the extensions improve the efficiency of the IFDS analysis for certain classes of IFDS problems. Another extension widens the class of problems applicable for the IFDS analysis. However, those extensions do not affect the precision of IFDS problems. Our analysis, in contrast, does not improve the efficiency or generality of IFDS, but it allows us to solve IFDS problems more precisely.

The fourth extension is targeted towards programs that are represented in SSA form. Executing the IFDS analysis on such programs results in loss of precision in the presence of control-flow constructs (e.g. conditionals and loops), compared to programs in non-SSA form. The extension makes the IFDS analysis on programs in SSA form as precise as on programs that are not represented in SSA form. In contrast, the correlated-calls analysis is applicable to programs in both SSA and non-SSA forms. Even if applied to a program in SSA form, our analysis and the extension improve the precision of IFDS in unrelated situations: the first analysis handles correlated calls, and the latter handles control-flow constructs. Thus, an IFDS analysis could benefit from both precision improvements independently.

Another work on improving the efficiency of the IFDS algorithm is E. Bodden et al.'s framework for the analysis of software products lines [4]. Their paper uses transformations from IFDS to IDE problems, a technique we also employ. Finally, J. Rodriguez and O. Lhoták

[2]However, we do not use WALA's IFDS implementation, as explained in Chapter 5.

implemented a concurrent version of the IFDS algorithm using actors [20]. However, neither of those works is concerned with improving the precision of IFDS results.

The correlated-calls analysis improves the precision of a data-flow analysis by eliminating a special type of infeasible paths. This is similar to the idea of context-sensitive analysis: just as a context-sensitive analysis eliminates infeasible paths from the end of a procedure to the call sites that do not match the given procedure call, the correlated-calls analysis eliminates infeasible paths caused by correlated method calls.

The idea of using correlated calls to remove infeasible paths in data-flow analyses of object-oriented programs was introduced by F. Tip [24]. The possibility of using IDE to achieve this is mentioned, but not elaborated upon. Our work presents a concrete solution to the problem and an implementation of that solution.

The idea of eliminating infeasible paths caused by correlated calls is similar to M. Sridharan et al.'s work on improving the precision of pointer analysis for JavaScript programs [23]. For each pointer, a pointer analysis determines the possible set of objects (the *points-to* set) that the pointer can reference at a given program point. In JavaScript, it is challenging to compute the points-to set of fields because in general, field names can be derived from arbitrary expressions and bound at runtime. As a result, an imprecise data-flow analysis will include infeasible paths between values of the form `o[p]` (access of a property `p` of object `o`), where at compile time, `p` can be bound to different values. The idea of the paper is to track all dynamic property accesses (reads and writes) on an object `o` with property name `p`. The code snippets containing the references `o[p]` are then extracted into a separate function f. The analysis is then run so that for each possible value of `p`, f is analyzed separately; therefore, for a given property name, all correlated objects with that name are analyzed together.

The differences between this method of tracking correlated calls and our analysis are the following.

- *Type of target data-flow analysis* whose precision is to be improved. Our analysis improves the precision of IFDS data-flow analyses, whereas the JavaScript analysis improves the precision of pointer analysis.
- *Target language.* Our analysis is for object-oriented languages where polymorphic methods, and not property names (which are known at compile time), cause infeasible paths.
- *Different handling of correlated calls.* Extracting code that contains correlated calls into separate methods would not prevent infeasible paths. Instead, our analysis uses IDE flow functions to detect and eliminate infeasible paths caused by correlated calls.

2.2 Terminology and Notation

We will start by introducing several concepts used by the IFDS and IDE analyses.

A *control-flow graph* is a directed graph in which nodes correspond to instructions and edges represent transfer of control between the instructions during an execution of the program. A control-flow graph has a unique start node, $\mathsf{start}_{\texttt{main}}$, which is the node corresponding to the program entrypoint.

An *intra-procedural* path is a path in a control-flow graph whose nodes are in the same procedure. By contrast, an *inter-procedural path* is one that contains nodes from different procedures.

A *control-flow supergraph* is a control-flow graph in which each procedure p is augmented with an additional *start node* start_p and *end node* end_p, and for each call c_q to a procedure q, there is a *call node* call_{c_q} and subsequent *return node* return_{c_q}.

A control-flow supergraph allows us to model the control flow in inter-procedural paths. The flow from the caller to the callee is represented using an edge $(\mathsf{call}_{c_q}, \mathsf{start}_q)$. The control flow from the callee back to the caller goes through an edge $(\mathsf{end}_q, \mathsf{return}_{c_q})$.

Example 2.2.1. Consider the program in Listing 2.2. The supergraph corresponding to that program is shown in Figure 2.1.

A *flow-sensitive* data-flow analysis is one that takes the order of program instructions into account.

Let each call node in a program be labeled with a distinct opening parenthesis and the corresponding return node with the matching closing parenthesis. For a given path p, let s be the string that is obtained by concatenating the labels of the nodes in p. Then p is *valid* if s belongs to the language of substrings of balanced parentheses. The set of all inter-procedurally valid paths from the start node to a node n is denoted as $\mathsf{VP}(n)$. The set $\mathsf{VP}(n)$ is a conservative approximation of all concrete execution paths from the start node to n.

A *context-sensitive* data-flow analysis is an analysis that considers only inter-procedurally valid paths.

Example 2.2.2. In the supergraph in Figure 2.1, let us assign $\{, \}$ parentheses to $\mathsf{call}_{\texttt{A.f}}$ and $\mathsf{return}_{\texttt{A.f}}$, and $\langle, \rangle$ parentheses to $\mathsf{call}_{\texttt{f}}$ and $\mathsf{return}_{\texttt{f}}$. Then the string corresponding to the path

$$p_1 = \Big[\mathsf{call}_{\texttt{A.f}},\ \mathsf{start}_{\texttt{f}},\ \texttt{if (s == null)},\ \texttt{return s},\ \mathsf{end}_{\texttt{f}},\ \mathsf{return}_{\texttt{A.f}}\Big]$$

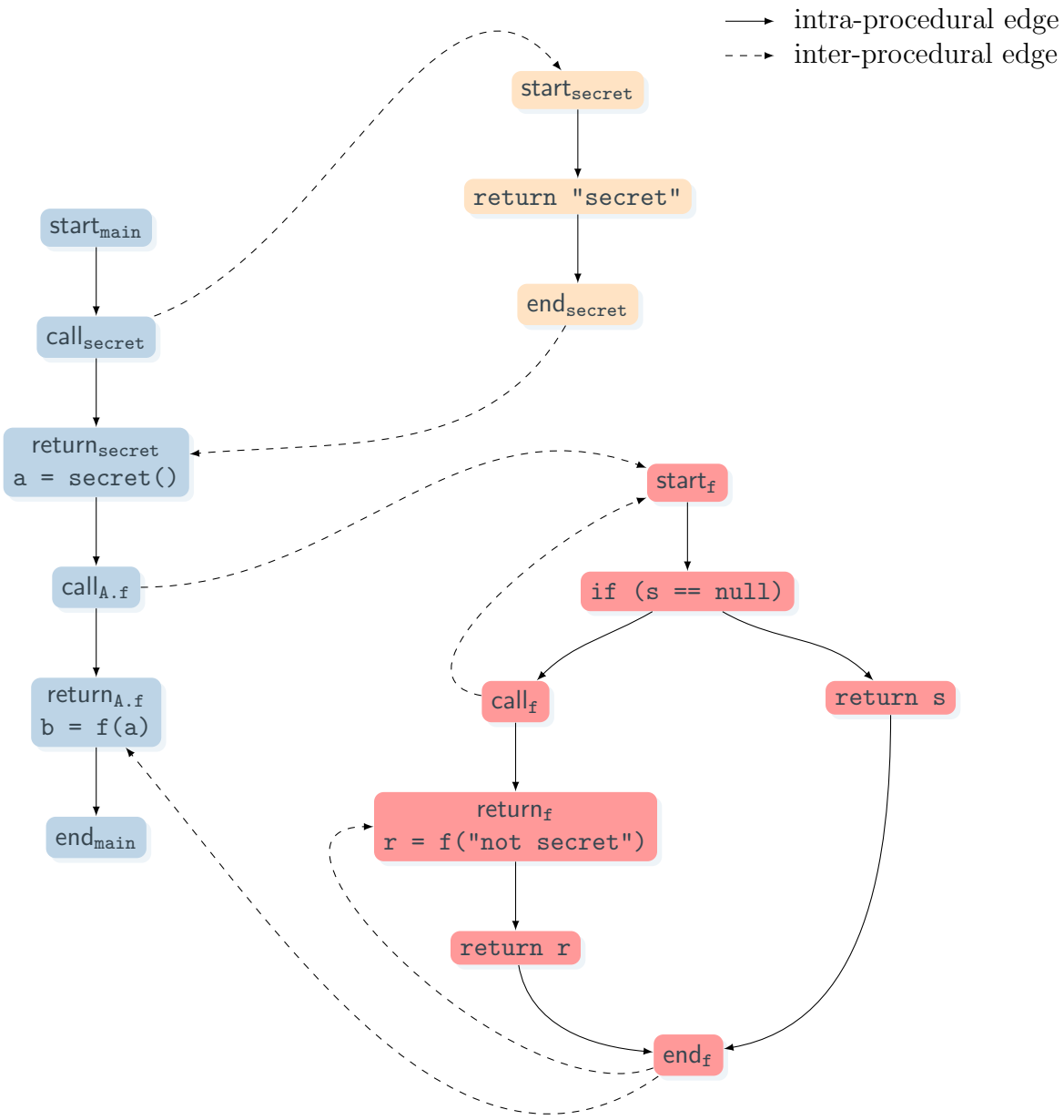


Figure 2.1: An example supergraph for Listing 2.2

```
1  class Main {
2    public static void main(String[] args) {
3      String a = secret();
4      String b = A.f(a);
5    }

6    static String secret() {
7      return "secret";
8    }
9  }

10 class A {
11   static String f(String s) {
12     if (s == null) {
13      String r = f("not secret");
14      return r;
15     }
16     return s;
17   }
18 }
```

Listing 2.2: An example Java program

is $\{\}$, which indicates that p_1 is valid. Every prefix of p_1 is also a valid path.

However, the graph also contains an inter-procedurally invalid path

$$p_2 = \left[\mathsf{call}_{\mathtt{f}}, \mathsf{start}_{\mathtt{f}}, \mathtt{if\ (s\ ==\ null)}, \mathtt{return\ s}, \mathsf{end}_{\mathtt{f}}, \mathsf{return}_{\mathtt{A.f}}\right]$$

with corresponding string $\langle\{\}$.

A *lattice* is a partially ordered set in which each subset has a least upper bound and a greatest lower bound.

A *meet semilattice* $L = (S, \sqcap)$ is defined by a set S and a meet operation $\sqcap$ that is associative, commutative, and idempotent. The meet operation induces a partial order $(S, \sqsubseteq)$ where every subset contains a greatest lower bound: For all $x, y \in S$, $x \sqsubseteq y$ if $x \sqcap y = x$. The greatest lower bound, or top element, of the semilattice is denoted as $\top$. If k is the length of the longest chains of elements in the semilattice, then the *height* of the semilattice is $k - 1$.

Finally, we introduce the notion of *distributivity*. Given a set D, a function $f : 2^D \to 2^D$ is distributive if $\forall x_1, x_2 \in 2^D$,

$$f(x_1 \cup x_2) = f(x_1) \cup f(x_2).$$

In this thesis, we will denote a map from a set of keys K to values from set V as

$$\{(k, v) \mid k \in K, v \in V\}.$$

For an arbitrary map m, $m(x)$ is the value to which x is mapped in m. We denote by $m[x \to y]$ a map identical to m, except that the element x is mapped to y. To avoid excessive parentheses, we write $(m[x_1 \to y_1])\,[x_2 \to y_2]$ as $m[x_1 \to y_1][x_2 \to y_2]$.

We will denote the identity function $\lambda x\,.x$ by id. We will use a typed version of this function in various contexts, where the type of x will vary with the context.

2.3 IFDS

The purpose of the IFDS framework is to solve a special subset of inter-procedural, flow-sensitive, context-sensitive data-flow-analysis problems. The main idea of IFDS is to encode the data-flow analysis problem into a graph-reachability problem.

2.3.1 Data-Flow Problems Suitable for IFDS

In this section we describe the data-flow problems that can be solved by an IFDS analysis. We will start with an intuitive definition and later on formalize the notion of an IFDS-suitable problem.

Informally, an IFDS analysis can only solve decision problems. An IFDS analysis answers questions of the following kind: "is property X true at program point Y?". For example, a taint-analysis problem asks, for each variable v in the program, "is v secret at a given program point?". An available-expressions problem asks, for each expression e, "does e have to be recomputed at a given program point?".

Formally, a data-flow analysis problem is suitable for an IFDS analysis if it can be encoded as an IFDS problem

$$(G^*, D, F, M_F, \sqcap),$$

where $G^* = (N^*, E^*)$ is the supergraph of the input program with nodes N^* and edges E^*, D is a finite set of *data-flow facts*, F is a set of distributive dataflow functions of type $2^D \to 2^D$, $M_F : E^* \to F$ is a function that maps supergraph edges to dataflow functions, and M_F is extended to paths by composition[3]. The *meet operator* $\sqcap$ is either union or intersection.

Without loss of generality, we will take meet to denote union. It can be shown that any problem where meet is defined as intersection can be reformulated into an equivalent one where meet is defined as union [18].

2.3.2 Overview of the IFDS Algorithm

Formally, given an IFDS problem, for each node $n \in N^*$ the IFDS algorithm computes the *meet-over-all-valid-paths* solution

$$\mathsf{MVP}_F(n) = \bigsqcap_{q \in \mathsf{VP}(n)} M_F(q)(\varnothing).$$

To compute the meet-over-all-valid-paths solution, each node in the control-flow supergraph is paired with a *fact* $d \in D \cup \{\mathbf{0}\}$, $\mathbf{0} \notin D$, yielding the nodes $N^\#$ of the *exploded supergraph* $G^\# = (N^\#, E^\#)$. Roughly, for each node in the program, a fact denotes a binary property whose value (true or false) we want to find out. The start node of the exploded supergraph is the node $(\mathsf{start}_{\texttt{main}}, \mathbf{0})$.

The flow functions F define the edges of the exploded supergraph. Using the flow functions, the IFDS algorithm computes the inter-procedurally *realizable* paths from the start to the rest of the exploded graph's nodes. A realizable path is a valid path in the exploded supergraph that starts with the entry node $\mathsf{start}_{\texttt{main}}$.

If there is a realizable path from the node $(\mathsf{start}_{\texttt{main}}, \mathbf{0})$ to a given node (n, d), $d \neq \mathbf{0}$, then the fact d is considered to hold at node n. A path to a node $(n, \mathbf{0})$ means that in the control-flow supergraph, there is a path from $\mathsf{start}_{\texttt{main}}$ to n.

In this way, the IFDS algorithm reduces the input data-flow problem to a graph-reachability problem.

[3]Let A be a set and $f : E^* \to (A \to A)$ a function from supergraph edges to functions on A. We say that f is extended to paths by composition to denote that for a path q consisting of the edges $e_1, \ldots, e_k$, $f(q) = f(e_k) \circ \ldots \circ f(e_1) \circ \mathsf{id}$.

Example 2.3.1. In a taint analysis, D is the set of variables in the program. If a fact $d \in D$ is reachable at a given node, then the variable is considered secret at that node. Otherwise, it is considered not secret. The question "is d secret at node n?" becomes "is there a realizable path from $(\mathsf{start}_{\texttt{main}}, \mathbf{0})$ to (n, d)?".

Example 2.3.2. In an available-expressions analysis, D is the set of all expressions in the program. If an expression $d \in D$ is reachable at a certain node, it means that it does not need to be recomputed at that node.

Example 2.3.3. The exploded supergraph for Listing 2.2 is shown in Figure 2.2. We can see that there is a realizable path from the start node of the exploded graph to the variable `b` at the node $\mathsf{return}_{\texttt{A.f}}$ in the `main` method. This means that at that node, `b` is considered secret.

The flow functions $F \subseteq 2^D \to 2^D$ allow us to establish the edges in the exploded supergraph.

Given a control-flow-graph edge $e = (n_1, n_2) \in E^*$ and a distributive dataflow function $f = M(e)$, the *representation relation* $R_f : (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$ of f is defined as

$$R_f = \{(\mathbf{0}, \mathbf{0})\} \cup \{(\mathbf{0}, d_j) \mid d_j \in f(\varnothing)\} \cup \{(d_i, d_j) \mid d_j \in f(\{d_i\}), d_j \notin f(\varnothing)\}.$$

Each pair $(d_i, d_j) \in R_f$ corresponds to an edge $((n_1, d_i), (n_2, d_j))$ in the exploded supergraph.

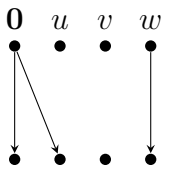
Note that R_f constructs pairs of dataflow facts so that

- there is always an edge $(\mathbf{0}, \mathbf{0})$ corresponding to the control-flow-graph edge;
- if there is an edge $(\mathbf{0}, d_j)$, then there is no other edge leading to d_j; in particular, there is never an edge $(d_i, \mathbf{0})$ where $d_i \neq \mathbf{0}$.

Example 2.3.4. The representation relation R_f for a set of data-flow facts $D = \{u, v, w\}$ and dataflow function $f = \lambda S \,.\, S \setminus \{v\} \cup \{u\}$ looks as follows:

$$R_f = \{(\mathbf{0}, \mathbf{0}), (\mathbf{0}, u), (w, w)\}.$$

The corresponding exploded-graph edges are shown below.



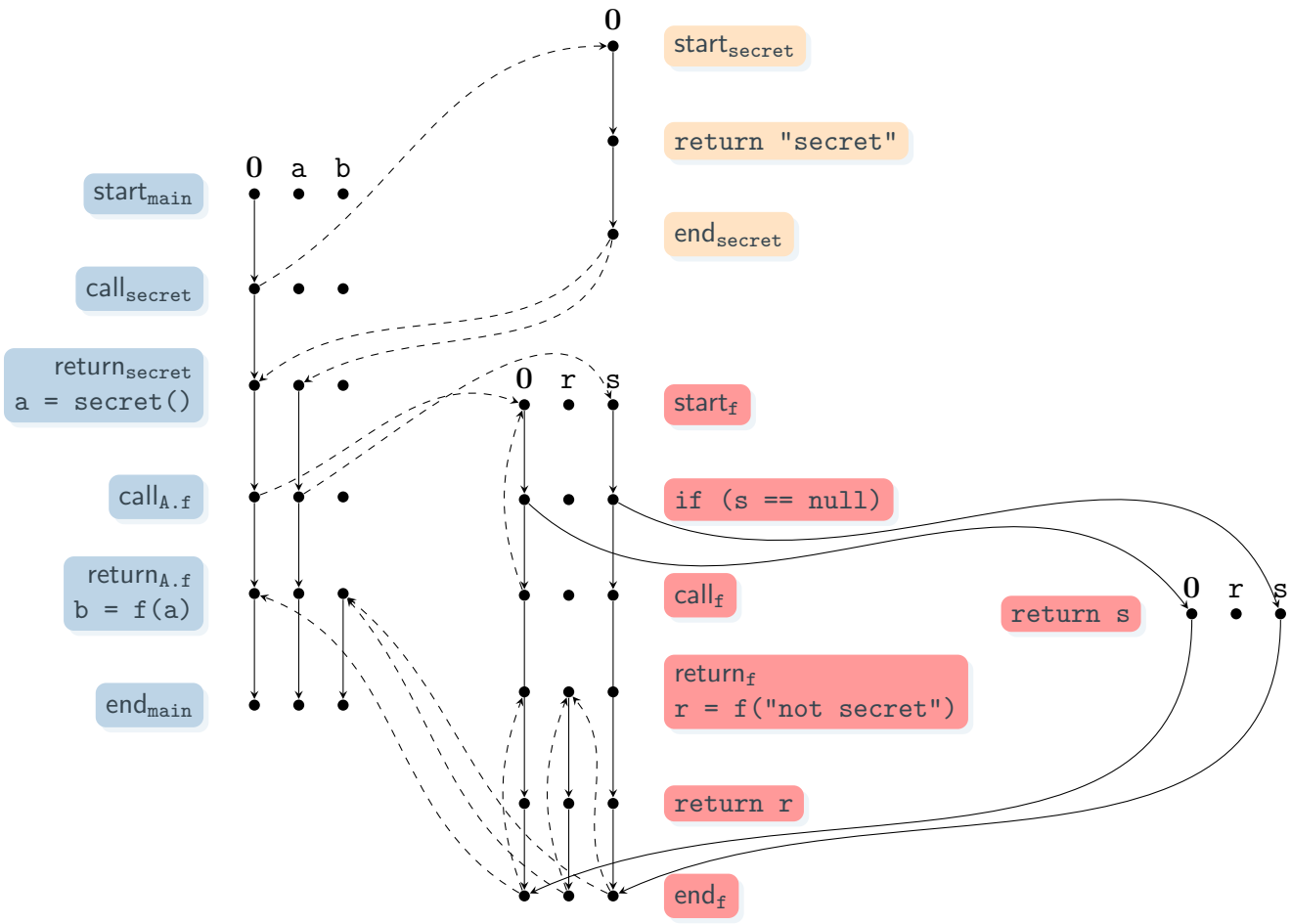


Figure 2.2: The exploded supergraph corresponding to a taint analysis for the input program in Listing 2.2

The representation relation lets us decompose a flow function into functions that operate on each fact individually. This is possible due to distributivity: we can apply the flow function on each single fact and take the union of the results, rather than applying the function to the union of the facts.

The representation relation allows us to compactly represent the composition and meet operations which are required for the IFDS algorithm.

For two representation relations R_{f_1}, R_{f_2}, the composition and meet operations are defined as follows:

$$R_{f_1} \circ R_{f_2} = \{(d_1, d_3) \mid \exists d_2 : \ (d_1, d_2) \in R_{f_1}, (d_2, d_3) \in R_{f_2}\}.$$

and

$$R_{f_1} \sqcap R_{f_2} = R_{f_1} \cup R_{f_2}.$$

The representation relation distributes over composition and meet:

$$R_{f_1} \circ R_{f_2} = R_{f_1 \circ f_2},$$

and

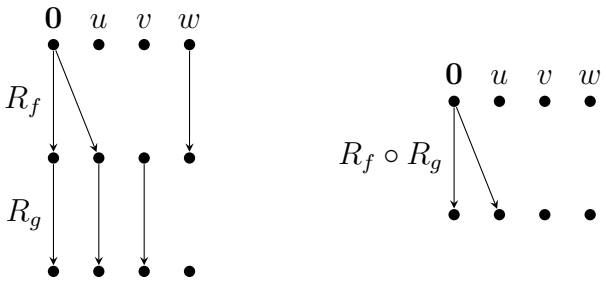
$$R_{f_1} \sqcap R_{f_2} = R_{f_1 \sqcap f_2},$$

On the exploded graph, the composition of two functions is represented by the paths that are formed when the exploded-graph edges are combined.

Example 2.3.5. If $g = \lambda S \,.\, S \setminus \{w\}$ and f is defined as in example 2.3.4, then

$$R_f \circ R_g = \{(\mathbf{0}, \mathbf{0}), (\mathbf{0}, u)\},$$

as illustrated by the corresponding exploded graph edges:



To convert a representation relation R_f back into the original flow function f, we can use the *interpretation* function $[\![R_f]\!]$:

$$f = [\![R_f]\!] = \lambda D_1 \,.\, (\{d_2 \mid \exists d_1 \in D_1 : (d_1, d_2) \in R_f\} \cup \{d_2 \mid (\mathbf{0}, d_2) \in R_f\}) \setminus \{\mathbf{0}\}.$$

We presented an overview of the IFDS analysis. IFDS problems are transformed into IDE problems by the correlated-calls analysis. The IDE analysis is described in the next section.

2.4 IDE

There exists an entire class of data-flow problems that cannot be formulated as IFDS problems. Informally, the problems cannot be formulated as decision problems. For instance, a constant-propagation problem asks, for each variable v in the program, "if v is a constant at a given program point, what is v's value?". The questions asked by constant propagation are of the form "if property X *(v being a constant)* is true at program point Y, what is the value of some property Z *(the value of the constant)* corresponding to X?". It turns out that problems with such questions can often be solved by the IDE algorithm.

Instead of just telling us whether a fact holds or not, the IDE analysis can provide us with additional information about facts.

Just as in the IFDS analysis, the IDE algorithm reduces a data-flow problem to a graph-reachability problem. Additionally, for each program point, the algorithm computes an *environment* $\mathsf{Env}(D, L)$, where data-flow facts are mapped to values of a lattice L.

For example, using the IDE analysis, we can encode a restricted version of a constant-propagation analysis[4]. The data-flow facts correspond to program variables, and the lattice incorporates all possible values for constants. If a fact d in the exploded supergraph is reachable at node n, and $\mathsf{Env}(d) \notin \{\bot, \top\}$, it means that the variable associated with d is a constant. Furthermore, the value of the constant can be inferred from the environment for the corresponding node and is equal to $\mathsf{Env}(d)$.

Formally, an IDE problem is defined as a four-tuple

$$(G^*, D, L, M_{\mathsf{Env}}),$$

where G^* is a control-flow supergraph, D is a set of data-flow facts, and L is a meet semilattice with finite height. Finally, $M_{\mathsf{Env}} : E^* \to (\mathsf{Env}(D, L) \to \mathsf{Env}(D, L))$ is a function

[4]In the general case, constant propagation cannot be encoded with distributive flow functions and is therefore not suitable for an IDE analysis [13].

from the edges of the control-flow supergraph to distributive *environment transformers*. M_{Env} is extended to paths by composition.

Given an IDE problem, for each node $n \in N^*$ and fact $d \in D$, the IDE algorithm computes the meet-over-all-valid-paths solution

$$\mathsf{MVP}_{\mathsf{Env}}(n, d) = \bigsqcap_{q \in \mathsf{VP}(n)} M_{\mathsf{Env}}(q)(\Omega)(d), \tag{2.1}$$

where M_{Env} is extended to paths by composition and

$$\Omega = \lambda d \,.\, \top$$

is the top element in the environment lattice $\mathsf{Env}(D, L)$.

The IDE analysis is a generalization of the IFDS analysis: every IFDS problem can be converted into an equivalent IDE problem [18]. The equivalent problem can be solved by the IDE algorithm, and the result converted into an IFDS result. In an IFDS-equivalent IDE problem, the graph G^* and the set D of data-flow facts remain the same. The L lattice is a two-point lattice: if a fact is mapped to the top (bottom) element, then it is reachable (unreachable). The conversion between IFDS and IDE problems is discussed in detail in Section 3.2.

2.4.1 Environment Transformers

For each node in the control-flow graph, the result of an IDE analysis computes an environment $\mathsf{Env}(D, L)$, which is a map from data-flow facts to lattice elements.

Instead of flow functions that show how to propagate facts, the IDE framework uses distributive environment transformers to propagate environments. For each edge (n_1, n_2) in the control-flow supergraph, an environment transformer indicates how the environment at node n_1 is modified at node n_2.

From Section 2.3.2 we know that flow functions can be represented with exploded-graph edges. To represent environment transformers, we will construct *labeled* exploded-graph edges, where each edge is associated with a distributive *micro function*[5] $f : L \to L$. A micro function shows how to change a lattice element for a given node and fact.

If an IDE problem is equivalent to an IFDS problem, the edges of the exploded supergraph are the same for both problems. In the IDE problem, the edges of the exploded supergraph are labeled with identity micro functions.

[5] See Sagiv et al. [21] for a formal definition of the representation relation for environment transformers.

We extend the meet operator to work on micro functions by defining

$$(f_1 \sqcap f_2)(l) = f_1(l) \sqcap f_2(l)$$

for all $l \in L$.

In IDE problems, the auxiliary fact analogous to **0** in IFDS is denoted as Λ.

Example 2.4.1. One version of the constant propagation analysis that can be encoded with IDE is *linear constant propagation*. A linear constant propagation analysis can detect constants of the form $a \cdot x + b$, where a and b are integers and x is a variable. In particular, a variable can only be considered constant if it depends on at most one other constant variable: even if y and z are variables that are considered constant, the variable $x = y + z$ will be considered not constant. If we encoded the analysis in a way to handle non-linear constant assignments, we would have to use non-distributive flow functions, which would violate the requirements of the IDE algorithm.

For linear constant propagation, the L lattice consists of the set of integers $\mathbb{Z}$, a top element denoting "not a constant", and a bottom element denoting an unknown value. The meet of two lattice elements is defined as follows: for any lattice element $l \in L$,

$$\top \sqcap l = \top \qquad \text{and} \qquad \bot \sqcap l = l.$$

For two lattice elements l_1, $l_2 \in \mathbb{Z}$,

$$l_1 \sqcap l_2 = \top.$$

We define the addition and multiplication operations on lattice elements $l \in L$ and integers $c \in \mathbb{Z}$ as follows:

$$l + c = \begin{cases} \bot & \text{if } l = \bot; \\ \top & \text{if } l = \top; \\ l + c & \text{otherwise.} \end{cases} \qquad c \cdot l = \begin{cases} \bot & \text{if } l = \bot; \\ \top & \text{if } l = \top; \\ c \cdot l & \text{otherwise.} \end{cases}$$

Let the function M that maps supergraph edges to environment transformers be defined in the following way:

$$M = \lambda((n_1, n_2)) \,. \begin{cases} \lambda \mathsf{env} \,.\, \mathsf{env}[x \to a \cdot \mathsf{env}(y) + c] & \text{if } n_1 \text{ contains an assignment} \\ & x = a \cdot y + c, \text{ where } y \text{ is a variable} \\ & \text{and } a, c \text{ are constants;} \\ \mathsf{id} & \text{otherwise.} \end{cases}$$

Here, we denote with $\mathsf{env}[x \to a]$ an environment env in which the key x is mapped to a, and all other keys $y \neq x$ are mapped to their old values $\mathsf{env}(y)$. When M is applied to an edge whose source node contains an assignment for a variable x, M returns an environment transformer that updates the argument environment with a new value for x.

Consider the following program:

```
1 int u = 1;
2 int v = u + 2;
3 int w = u + v;
4 u = 5;
```

For the edges e_1, e_2, e_3, and e_4 that start at the first, second, third, and fourth instruction, M creates the following environment transformers:

$$\begin{aligned} M(e_1) &= \lambda\mathsf{env}\,.\,\mathsf{env}[\mathtt{u} \to 1] \\ M(e_2) &= \lambda\mathsf{env}\,.\,\mathsf{env}[\mathtt{v} \to \mathsf{env}(\mathtt{u}) + 2] \\ M(e_3) &= \lambda\mathsf{env}\,.\,\mathsf{env}[\mathtt{w} \to \top] \\ M(e_4) &= \lambda\mathsf{env}\,.\,\mathsf{env}[\mathtt{u} \to 5]. \end{aligned}$$

The corresponding labeled exploded supergraph is shown in Figure 2.3.

The result of the analysis yields a map from nodes to environments. Each environment maps variables to elements of the constant-propagation lattice. The environment at the last node will look as follows:

$$\{(\mathtt{u},\, 5),\ (\mathtt{v},\, 3),\ (\mathtt{w},\, \top)\}.$$

In this way, each edge in the exploded graph is labeled with a micro function. The mapping from exploded-graph edges to the corresponding micro functions is stored in *edge functions*, denoted as EdgeFn: $E^{\#} \to (L \to L)$.

2.4.2 Overview of the IDE Algorithm

Given a labeled exploded supergraph, the IDE algorithm computes the environments for all nodes in the control-flow graph.

The algorithm first computes the lattice elements $l_{n,d}$ that correspond to each reachable node $(n,\, d)$ in the exploded supergraph. The union of the exploded nodes $(n,\, d)$ for a

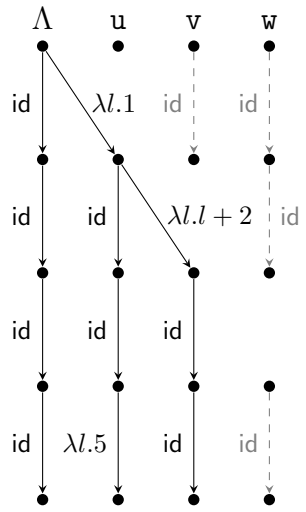


Figure 2.3: A labeled exploded supergraph for a constant-propagation analysis described in Example 2.4.1. The dashed edges are edges not reachable from the entry node.

given control-flow node n, mapped to the corresponding lattice elements $l_{n,d}$, form the environment Env_n for that node:

$$\mathsf{Env}_n = \left\{(d,\, l_{n,d}) \mid (n,\, d) \in N^{\#}\right\}.$$

The overall idea behind computing the lattice elements $l_{n,d}$ is the following. For each inter-procedurally realizable path

$$p = [(\mathsf{start}_{\mathtt{main}},\, \Lambda),\, (n_1,\, d_1),\, \ldots,\, (n_k,\, d_k)]$$

that starts with the entrypoint of the exploded supergraph, we compute the function f_p that corresponds to p. The micro function consists of the composition of all individual micro functions with which the edges of p are labeled:

$$f_p = \mathsf{EdgeFn}((n_{k-1},\, d_{k-1}),\, (n_k,\, d_k)) \circ \ldots \circ \mathsf{EdgeFn}(\mathsf{start}_{\mathtt{main}},\, \Lambda),\, (n_1,\, d_1)).$$

Let the lattice element that $(n,\, d)$ is mapped to according to path p be denoted as $l^p_{n,d}$. As shown in Sagiv et al. [21], the lattice element can be obtained by applying f_p to the bottom element:

$$l^p_{n,d} = f_p(\bot).$$

Let Q be the set of paths that start at the entry point and end at the given node $(n,\, d)$.

The lattice element $l_{n,d}$ is the meet of the lattice elements corresponding to all the paths in Q:

$$l_{n,d} = \prod_{q \in Q} l^q_{n,d}.$$

This is a general outline of the IDE analysis. We use the IDE framework to improve the precision of IFDS problems in the presence of correlated calls. The next chapter describes how this is done.

Chapter 3

Correlated Calls Analysis

The correlated-calls analysis is presented as a transformation from an arbitrary IFDS problem to a corresponding IDE problem.

After solving the generated IDE problem, its result can be converted to an IFDS result. If the input program contains correlated calls, the converted IFDS result can be more precise than the original IFDS result.

In this chapter, we first discuss what is necessary to define IFDS and IDE problems. Next we describe how to convert any IFDS problem into an equivalent IDE problem, and, given a solution to the generated IDE problem, how to obtain the result of the original IFDS problem. We then show how to transform an IFDS problem into an IDE problem using the correlated-calls transformation, and how to convert the solution to the latter IDE problem into a more precise IFDS result.

3.1 Defining IFDS and IDE Problems

In Chapter 2, we defined what IFDS and IDE problems are, their applications, and their constraints. In this section, we describe how to create instances of IFDS and IDE problems.

3.1.1 Defining an IFDS Problem

Recall that an IFDS problem instance is defined as a five-tuple

$$(G^*,\ D,\ F,\ M_F, \sqcap),$$

where $G^* = (N^*,\ E^*)$ is the control-flow supergraph of the program, D is the set of dataflow facts, $F \subseteq 2^D \to 2^D$ is a set of distributive dataflow functions, and the function

$$M_F :\ E^* \to (2^D \to 2^D)$$

maps the supergraph edges to dataflow functions, and is extended to paths by composition.

In practice, an IFDS problem can be defined by providing an exploded supergraph $G^\# = (N^\#,\ E^\#)$. Each node of $G^\#$ is a pair $(n,\ d)$, where $n \in N^*$ is a node in the control-flow supergraph and $d \in (D \cup \{\mathbf{0}\}),\ \mathbf{0} \notin D$, where $\mathbf{0}$ is an auxiliary fact that is necessary for the IFDS algorithm.

The meaning of an edge in the exploded supergraph is the following. Let $(n_1,\ d_1)$ and $(n_2,\ d_2)$ be two nodes in the exploded supergraph $G^\#$. Furthermore, assume that if fact d_1 at node n_1 holds, then the fact d_2 at node n_2 also holds. Then there is an edge $(n_1,\ d_1),\ (n_2,\ d_2) \in E^\#$.

3.1.2 Defining an IDE Problem

An IDE problem instance is a four-tuple

$$(G^*,\ D,\ L,\ M_{\mathsf{Env}}),$$

where G^* and D are defined in the same way as for IFDS. L is a finite-height lattice that represents the values to which dataflow facts are mapped in an IDE problem. An environment $Env(D, L)$ maps dataflow facts to lattice elements. Finally, the map

$$M_{\mathsf{Env}} :\ E^* \to (\mathsf{Env}(D, L) \to \mathsf{Env}(D, L))$$

is a function from the control-flow-supergraph edges to environment transformers, extended to paths by composition.

An IDE problem can be defined with a labeled exploded supergraph[6], in which an edge function

$$\mathsf{EdgeFn} :\ E^\# \to (L \to L)$$

pairs edges with *micro functions*, and is extended to paths by composition.

The set of micro functions of an IDE problem is a subset of $L \to L$ that is closed under function meet and composition.

[6] The exploded supergraph in an IDE problem is defined in the same way as in an IFDS problem. The only difference is that the $\mathbf{0}$ fact is denoted as Λ [18, 21].

The meaning of an edge in the labeled exploded supergraph is the following. Let $e = ((n_1, d_1), (n_2, d_2)) \in E^{\#}$ be an edge in the exploded supergraph with label $f = \mathsf{EdgeFn}(e)$. Then

- if at node n_1 the fact d_1 was mapped to a lattice element l_1 by an environment $Env(D, L)$, then the fact d_2 at node n_2 should be mapped to $f(l_1)$.

As shown in Sagiv et al. [21], the relationship between environment transformers and edge functions can be described with the following equations. For individual edges $(n_1, n_2) \in E^*$,

$$\begin{aligned} &M_{\mathsf{Env}}((n_1, n_2))(\mathsf{env})(d) \\ &\quad = \mathsf{EdgeFn}((n_1, \Lambda), (n_2, d))(\top) \sqcap \bigsqcap_{d' \in D} \mathsf{EdgeFn}((n_1, d'), (n_2, d))(\mathsf{env}(d')), \end{aligned} \tag{3.1}$$

where env is an environment $\mathsf{Env}(D, L)$. Informally, for a given control-flow-supergraph edge e and data-flow fact d, the M_{Env} function captures the meet of the edge function applied to all possible exploded-graph edges along e.

For paths p that start with the entry point $\mathsf{start}_{\texttt{main}}$,

$$M_{\mathsf{Env}}(p)(\Omega)(d) = \bigsqcap_{r \in \mathsf{RP}(p, d)} \mathsf{EdgeFn}(r)(\top), \tag{3.2}$$

where $n \in N^*$, $d \in D$, $p \in \mathsf{VP}(n)$, and RP is the set of all inter-procedurally realizable paths.

To summarize, an IDE problem can be defined by a labeled exploded supergraph

$$(G^{\#}, L, \mathsf{EdgeFn}),$$

where each edge of the exploded supergraph corresponds to a micro function.

3.2 Transformations Between IFDS and IDE

The correlated-call analysis transforms an existing IFDS problem into a special kind of IDE problem. We described what is necessary to define IFDS and IDE problems independently.

Let $P = (G^{\#})$ be an IFDS problem and $Q = (G^{\#}, \mathsf{EdgeFn})$ an IDE problem obtained by a conversion from P.

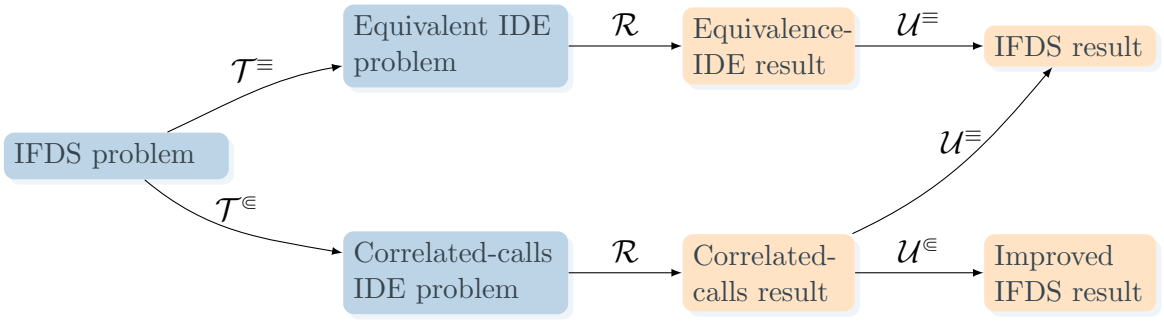


Figure 3.1: Transformations between IFDS and IDE problems and their results

We will look at two kinds of transformations

$$\mathcal{T} : (G^{\#}) \to (G^{\#}, \mathsf{EdgeFn})$$

from IFDS to IDE problems:

- an equivalence transformation $\mathcal{T}^{\equiv}$ (pronounced as "t-equiv"), in which we show how to transform IFDS problems into equivalent IDE problems;
- a correlated-call transformation $\mathcal{T}^{\Subset}$ (pronounced as "t-c-c"), where we show how to convert IFDS problems into a special form of IDE problems that help eliminate infeasible paths.

In each case we also show how to convert the result of the generated IDE problem to a result of the original IFDS problem.

An overview of the transformations is shown in Figure 3.1.

3.2.1 Equivalence Transformation

We start with an equivalence transformation $\mathcal{T}^{\equiv}$ to present a simple IFDS-to-IDE conversion that does not change the result of the original IFDS problem. We will compare the correlated-calls transformation with the equivalence transformation, and use the latter to show that the correlated-calls analysis results in a precision improvement of the original IFDS problem result.

3.2.1.1 Converting IFDS problems to IDE problems

Since IDE is a generalization of IFDS, any IFDS problem can be converted into an equivalent IDE problem [21]. For an equivalence transformation $\mathcal{T}^{\equiv}$, the generated lattice $L^{\equiv}$ consists of two elements, bottom and top:

$$L^{\equiv} = \{\bot,\ \top\},$$

where $\bot$ means "reachable", and $\top$ means "not reachable".

All micro functions are identity functions.

Given an exploded supergraph $G^{\#}$ provided by an IFDS problem, we want to create an edge function $\mathsf{EdgeFn}^{\equiv}$ that maps $G^{\#}$'s edges $E^{\#}$ to micro functions $L^{\equiv} \to L^{\equiv}$.

The edge functions $\mathsf{EdgeFn}^{\equiv}$ are defined as

$$\mathsf{EdgeFn}^{\equiv} = \begin{cases} \lambda e\,.\,\lambda m\,.\,\bot & \text{if } d_1(e) = \Lambda \text{ and } d_2(e) \neq \Lambda; \\ \lambda e\,.\,\mathsf{id} & \text{otherwise,} \end{cases}$$

where $d_1(e)$ is the source fact of an edge e and $d_2(e)$ is its target fact. At a "diagonal" edge from a Λ-fact to a non-Λ-fact d, the micro function is a constant function that returns $\bot$, which makes it a bottom element in the $L \to L$ lattice. Since the initial lattice element passed to the micro function at the start node is the top element (see (3.2)), the bottom function at the diagonal edge swaps the top element to bottom to make the fact d reachable.

The resulting equivalence transformation looks as follows:

$$\mathcal{T}^{\equiv}((G^{\#})) = (G^{\#},\ L^{\equiv},\ \mathsf{EdgeFn}^{\equiv}).$$

Thus, in $\mathcal{T}^{\equiv}$, all non-diagonal edges in the original IFDS problem are mapped to identity functions.

3.2.1.2 Converting IDE Results to IFDS Results

The output of an IFDS analysis states whether a node is reachable in the exploded supergraph. This means that for an IFDS problem P, the IFDS-analysis result $\mathcal{R}_{\text{IFDS}}(P) : N^* \to 2^D$ is a map from nodes of the control-flow supergraph to sets of facts:

$$\mathcal{R}_{\text{IFDS}}(P) = \{(n,\ \mathsf{MVP}_F(n)) \mid n \in N^*\}.$$

Example 3.2.1. The solution to the taint-analysis IFDS problem $\mathcal{P}$ in Listing 2.2 whose exploded supergraph is presented in Figure 2.2 looks as follows:

$$\begin{aligned}\mathcal{R}_{\text{IFDS}}(\mathcal{P}) = \{&(\texttt{return}_{\texttt{secret}}, \{\texttt{a}\}), && (\texttt{start}_{\texttt{f}}, \{\texttt{s}\}), && (\texttt{return}_{\texttt{f}}, \{\texttt{r}, \texttt{s}\}),\\ &(\texttt{call}_{\texttt{A.f}}, \{\texttt{a}\}), && (\texttt{if(s==null)}, \{\texttt{s}\}), && (\texttt{return r}, \{\texttt{r}, \texttt{s}\}),\\ &(\texttt{return}_{\texttt{A.f}}, \{\texttt{a}, \texttt{b}\}), && (\texttt{call}_{\texttt{f}}, \{\texttt{s}\}), && (\texttt{end}_{\texttt{f}}, \{\texttt{r}, \texttt{s}\})\}.\\ &(\texttt{end}_{\texttt{main}}, \{\texttt{a}, \texttt{b}\}), && (\texttt{return s}, \{\texttt{s}\}),\end{aligned}$$

All other nodes of the control-flow supergraph are mapped to the empty set.

The IDE analysis associates a lattice element with each node in the exploded supergraph. For an IDE problem Q, the result $\mathcal{R}(Q) : N^{\#} \to L$ maps nodes of the exploded supergraph to lattice elements (see (2.1)):

$$\mathcal{R}(Q) = \{((n, d), \mathsf{MVP}_{\mathsf{Env}}(n, d)) \mid n \in N^*, d \in D\}. \tag{3.3}$$

In other words, for each fact $d \in D$ at a given node $n \in N^*$, $\mathcal{R}(Q)(n, d)$ returns a lattice element. If a fact $d \in D$ is unreachable, $\mathcal{R}(Q)(n, d) = \top$.

In the case of an equivalence transformation from IFDS to IDE, if a node in the IFDS result is reachable, it will be also reachable in the IDE result, and it will be mapped to the bottom lattice element. For an exploded node in the IDE result, being mapped to the top element means being not reachable.

The domain of an equivalence-IDE result

$$\mathcal{R}_{\equiv} = \mathcal{R}(\mathcal{T}^{\equiv}(P))$$

consists of pairs of control-flow-supergraph nodes and data-flow facts. The range of the result is the set of lattice elements. To transform an IDE result to an IFDS result, we need to map each control-flow-supergraph node to the set of facts with which it is paired, provided that the pair is mapped to the bottom lattice element.

Example 3.2.2. Converting the IFDS problem $\mathcal{P}$ from Example 3.2.1 into an equivalent IDE problem and solving it will yield the following result:

$$\begin{aligned}\mathcal{R}(\mathcal{T}^{\equiv}(\mathcal{P})) = \{&((\texttt{return}_{\texttt{secret}}, \texttt{a}), \bot),\\ &((\texttt{call}_{\texttt{A.f}}, \texttt{a}), \bot),\\ &((\texttt{return}_{\texttt{A.f}}, \texttt{a}), \bot),\\ &((\texttt{return}_{\texttt{A.f}}, \texttt{b}), \bot),\\ &\ldots\}.\end{aligned}$$

Suppose that for a pair (n, d), where $n \in N^*$ and $d \in D$, there is no corresponding result in $\mathcal{R}_{\text{IFDS}}(\mathcal{P})$ (see Example 3.2.1). Then (n, d) appears in $\mathcal{R}(\mathcal{T}^{\equiv}(\mathcal{P}))$ as $((n, d), \top)$.

Let ρ be the result of an equivalence-IDE analysis for an IFDS problem P:

$$\rho = \mathcal{R}(\mathcal{T}^{\equiv}(P)).$$

For a node $n \in N^*$, let $D_n^{\equiv}(\rho)$ be a set of data-flow facts such that

$$D_n^{\equiv}(\rho) = \{d \mid d \in D \wedge \rho(n, d) \neq \top\}.$$

Then the transformation function $\mathcal{U}^{\equiv} : (N^{\#} \to L) \to (N^* \to 2^D)$ from an IDE result to an IFDS result looks as follows:

$$\mathcal{U}^{\equiv}(\rho) = \{(n, D_n^{\equiv}(\rho)) \mid n \in N^*\}.$$

Obviously, if applied to the result of an equivalence-IDE problem, $\mathcal{U}^{\equiv}$ returns a result equivalent to the original IFDS problem result. In other words, for any IFDS problem P with supergraph N^*, and any node $n \in N^*$,

$$\mathcal{U}^{\equiv}(\mathcal{R}(\mathcal{T}^{\equiv}(P)))(n) = \mathcal{R}_{\text{IFDS}}(P)(n).$$

Example 3.2.3. Converting the result in Example 3.2.2 with the equivalence-transformation from an IDE result to an IFDS result $\mathcal{U}^{\equiv}$ will yield the same result as in Example 3.2.1.

3.2.2 Correlated-Call Transformation

To improve the precision of an IFDS problem, we can convert it to a special type of IDE problem, and use lattice elements to provide us with additional information about a node. When converting the IDE result to an IFDS result, lattice elements will tell us whether to make the corresponding exploded nodes reachable. This is the idea of the correlated-calls analysis.

3.2.2.1 Lattice Elements

Just like in the equivalence transformation $\mathcal{T}^{\equiv}$, the exploded supergraph for $\mathcal{T}^{\Subset}$ is the same as in the original IFDS problem. The elements of the correlated-calls lattice $L^{\Subset}$ are functions that map receivers to sets of types:

$$L^{\Subset} = \left\{ m : \ R \to 2^T \right\},$$

where R is the set of receivers and T is the set of all types in the program. The type power set 2^T is also a lattice with a bottom element

$$\bot_T = T$$

and top element

$$\top_T = \varnothing.$$

The top element of the function lattice

$$\top_{\Subset} = \lambda r.\top_T$$

is a function that maps any receiver to the empty set[7]. The bottom element

$$\bot_{\Subset} = \lambda r.\bot_T$$

maps any receiver to all types in the program.

To understand the meaning of lattice elements in a correlated-call analysis, suppose that an IFDS problem has been converted to an IDE problem using the correlated-calls transformation. Assume also that s is the entrypoint of the program, n is a node in the exploded supergraph, and that in the IDE result, n is mapped to a lattice element $l \in L^{\Subset}$. Then the purpose of l is to provide information about the set of types of the objects that may be referenced by each receiver at runtime at a path from s to n. If a receiver is mapped to the empty set $\top_T$, it means that for the given program point, the receiver cannot reference an object of any type. In other words, the corresponding data-flow fact is considered not reachable.

3.2.2.2 Micro Functions

Unlike in the equivalence transformation, the micro functions returned by the edge function $\mathsf{EdgeFn}^{\Subset}$ are not always identity functions.

Let $e = (n_1, n_2) \in E^{\#}$ be an edge in the exploded supergraph. $\mathsf{EdgeFn}^{\Subset}(e)$ returns a micro function $f \subset L^{\Subset} \to L^{\Subset}$. Given a micro function (a map from receivers to sets of types) $m \in L^{\Subset}$, $f(m)$ returns a new map from receivers to sets of types. In other words, f shows how to update the map from receivers to sets of types when we encounter program point n_1.

Let f_1 and f_2 be two micro functions such that $f_1 = \lambda m \,.\, \lambda r \,.\, t_1(r)$ and $f_2 = \lambda m \,.\, \lambda r \,.\, t_2(r)$. We define the meet operation on micro-functions as follows:

$$\lambda m \,.\, \lambda r \,.\, t_1(r) \sqcap \lambda m \,.\, \lambda r \,.\, t_2(r) = \lambda m \,.\, \lambda r \,.\, t_1(r) \cup t_2(r). \tag{3.4}$$

The composition of micro functions is defined as ordinary function composition.

3.2.2.3 Edge Functions

Let $\mathcal{F}$ be the set of methods in a program with a signature $s_{\mathcal{F}}$.

Definition 3.2.4. *Let $r.c()$ be a call site on a receiver $r \in R$ with runtime type $t \in T$. Let $s_{\mathcal{F}}$ be the method signature corresponding to the call $c()$. For $s_{\mathcal{F}}$ and t, a lookup function returns the method implementation $f \in \mathcal{F}$ to which the call $r.c()$ is dispatched:*

$$\mathit{lookup}(s_{\mathcal{F}},\, t) = f.$$

Definition 3.2.5. *For a method signature $s_{\mathcal{F}}$ and a method implementation $f \in \mathcal{F}$, the static-type function τ returns the set of types for which the lookup function yields f:*

$$\tau(s_{\mathcal{F}},\, f) = \{\, t \mid \mathit{lookup}(s_{\mathcal{F}},\, t) = f \}.$$

In other words, τ computes the set of types for which calls to methods with signatures $s_{\mathcal{F}}$ are dispatched to f.

If there is a supergraph path from a method call with signature $s_{\mathcal{F}}$ to the start of f, then the set $\tau(s_{\mathcal{F}},\, f)$ is always non-empty.

Definition 3.2.6. *A call site is called monomorphic if it can be dispatched to only one method. If a call site can be dispatched to more than one method it is called polymorphic.*

Let $r.c()$ be a call on a receiver $r \in R$ with a method signature $s_{\mathcal{F}}$ to a function $f \in \mathcal{F}$. If the call site is monomorphic, then $\tau(s_{\mathcal{F}},\, f)$ contains all types $T' \subseteq T$ that are compatible with the static type of r. If the call site is polymorphic, then $\tau(s_{\mathcal{F}},\, f) \subset T'$, since some types $t \in T'$ cause dispatch to a method other than f.

Definition 3.2.7. *For an edge e, let $n_1(e)$ and $n_2(e)$ be the source and target nodes of e, and $d_1(e)$ and $d_2(e)$ be its source and target facts. A correlated-call edge function for the set $S \subseteq R$ is defined as follows:*

$$\mathit{EdgeFn}_S^{\Subset} = \lambda e \,.\, \begin{cases} \mathit{id} & \text{if } d_1(e) = d_2(e) = \Lambda, \\ \lambda m \,.\, \varepsilon_S(e)(\bot_{\Subset}) & \text{if } d_1(e) = \Lambda \text{ and } d_2(e) \neq \Lambda, \\ \lambda m \,.\, \varepsilon_S(e)(m) & \text{otherwise,} \end{cases} \tag{3.5}$$

where $\varepsilon_S : E \to (L \to L)$ *is a function defined as*

$$\varepsilon_S = \lambda e\,.\begin{cases} \lambda m\,.\,m[r \to m(r) \cap \tau(s_{\mathcal{F}},\, f)], & \text{if } e \text{ is a call-start edge. } r.c() \text{ is the call site at } n_1(e),\ f \text{ is the called procedure with signature } s_{\mathcal{F}}, \text{ and } r \in S; \\ \lambda m\,.\,m[r \to m(r) \cap \tau(s_{\mathcal{F}},\, f)] \\ \qquad [v_1 \to \bot_T] \\ \qquad \ldots \\ \qquad [v_k \to \bot_T], & \text{if } e \text{ is an end-return edge. } v_1, \ldots, v_k \in S \text{ are the local variables in the callee method, } r.c() \text{ is the call corresponding to the return node at } n_2(e),\ f \text{ is the called method with signature } s_{\mathcal{F}}, \text{ and } r \in S; \\ \lambda m\,.\,m\,[r \to \bot_T]\,, & \text{if } n_1(e) \text{ contains an assignment for } r \in S; \\ \mathit{id} & \text{otherwise.} \end{cases} \tag{3.6}$$

We define both $\mathsf{EdgeFn}_S^{\Subset}$ *and* ε_S *to be extended to paths by composition.*

In the above definition, the purpose of the set S is to limit the set of considered receivers. We will use S in Section 3.2.2.5.

The micro functions returned by a correlated-calls edge function can be described as follows. Along Λ-edges, the micro functions are identity functions. All other functions can be described with ε_S. On "diagonal" edges from Λ facts to non-Λ facts, ε_S creates edge-specific mappings for a set of receivers, and maps all the other receivers to the set of all types $\bot_T$. On all other edges, ε_S modifies the mappings for a set of receivers and leaves the mappings for the other receivers unchanged.

Example 3.2.8. Consider the program Listing 1.1. The exploded supergraph for that program is shown in Figure 3.2.

Returning a secret value in method `A.foo` creates a "diagonal" edge from the Λ-fact to the secret fact ψ. The diagonal edge is labeled with the micro function $\lambda m\,.\,\bot_{\Subset}$. Thus, at the end node of the method, every receiver is mapped to the set of all types $\bot_T$.

On the end-return edge from `A.foo` to `main`, we need to restrict the set of types for the receiver `a` by labeling the end-return edge from the fact ψ to the fact `v` with the micro function $\lambda m\,.\,m[\mathtt{a} \to m(\mathtt{a}) \cap \{\mathtt{A}\}]$.

Similarly, on the call-start edge from method `main` to method `B.bar`, from fact `v` to `s`, we restrict the type of the receiver `a` to the set $\{\mathtt{B}\}$ with the micro function $\lambda m\,.\,m[\mathtt{a} \to m(\mathtt{a}) \cap \{\mathtt{B}\}]$.

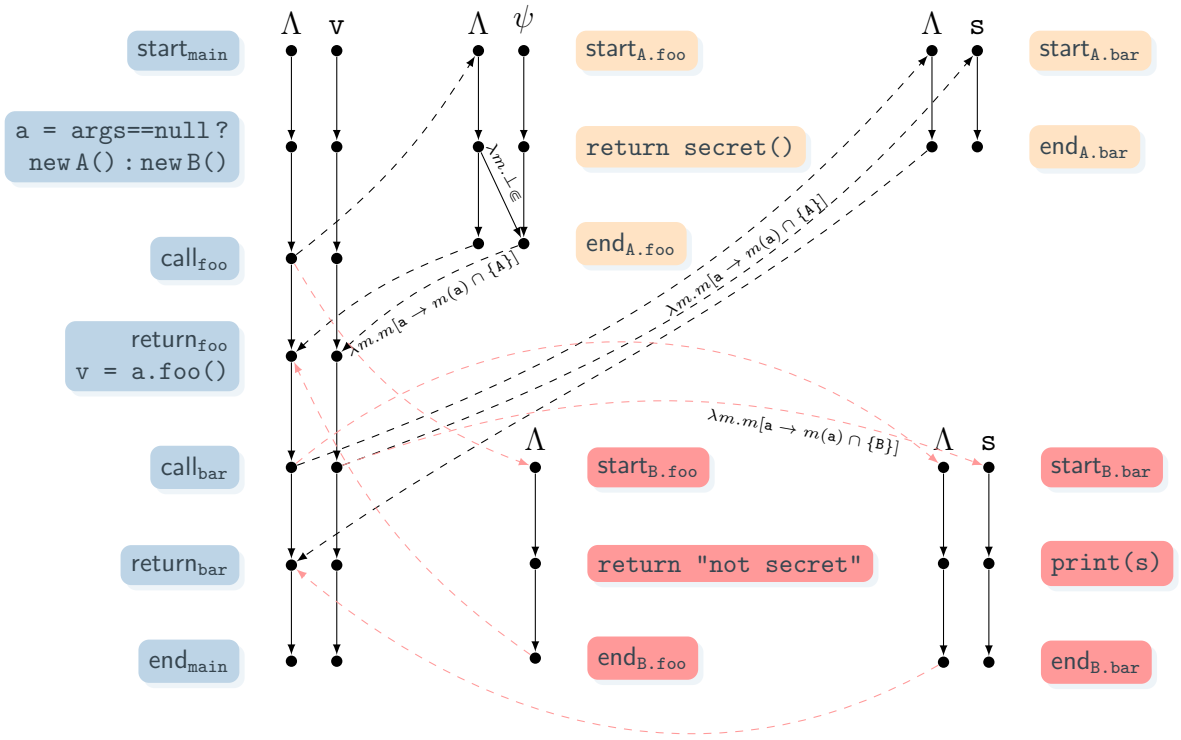


Figure 3.2: An example program demonstrating correlated-call edge functions on the Λ-node path for Listing 1.1. All non-labeled edges are implicitly labeled with identity functions id. The variable corresponding to an initial secret value is denoted as ψ.

After we have shown the definitions for the meet and composition operations, we will show in Example 4.2.8 how the correlated-calls analysis uses the presented micro functions to detect infeasible paths.

Definition 3.2.9. *For an IFDS problem $P = (G^{\#})$ and a set S, the correlated-calls transformation $\mathcal{T}_S^{\in}$ is defined as*

$$\mathcal{T}_S^{\in}((G^{\#})) = \left(G^{\#}, L_S^{\in}, \mathit{EdgeFn}_S^{\in}\right),$$

where $L_S^{\in} : S \to 2^T$.

Then, for an edge e, the correlated-call micro functions can be defined as $\mathsf{EdgeFn}_R^{\in}$ and a correlated-calls transformation is defined as $\mathcal{T}_R^{\in}$.

3.2.2.4 Converting IDE Results to IFDS Results

Let P be an IFDS problem. Let $E : N \times D$ be the domain of the IDE result $\mathcal{R}(Q)$. To convert $\mathcal{R}(\mathcal{T}_R^{\Subset}(P))$ to an IFDS result, we need to map the control-flow-supergraph nodes $n \in N^*$ to the corresponding facts $d \in D$. Unlike in $\mathcal{U}^{\equiv}$, we will only map each n to the facts d for which $\mathcal{R}(\mathcal{T}_R^{\Subset}(P))(n,\, d)$ does not contain an empty mapping for any receiver.

For a node $n \in N^*$ and a correlated-calls IDE problem result $\rho = \mathcal{R}(\mathcal{T}_S^{\Subset}(P))$, let $D_n^{\Subset}(\rho)$ be a set of data-flow facts defined as

$$D_n^{\Subset}(\rho) = \{d \mid d \in \mathsf{MVP}_F(n) \wedge \forall r \in R :\ \rho(n,\ d)(r) \neq \top_T\}\,. \tag{3.7}$$

Then, for a set $S \subseteq R$, the correlated-calls-conversion function from a correlated-calls IDE result ρ to an IFDS result looks as follows:

$$\mathcal{U}^{\Subset}\,(\rho) = \{(n,\, D_n^{\Subset}(\rho) \mid n \in N^*\}\,. \tag{3.8}$$

In the following lemma we show that the result of an IDE problem obtained through a correlated-calls transformation is a subset of the original IFDS result.

Lemma 3.2.10 (Precision)**.** *For an IFDS problem P and all $n \in N^*$,*

$$\mathcal{U}^{\Subset}\left(\mathcal{R}(\mathcal{T}_R^{\Subset}(P))\right)(n) \subseteq \mathcal{R}_{IFDS}(P)(n). \tag{3.9}$$

Proof. The transformation $\mathcal{U}^{\Subset}$ is the same as $\mathcal{U}^{\equiv}$, except that it can remove data-flow facts from the result:

$$\begin{aligned}
\mathcal{U}^{\Subset}\left(\mathcal{R}(\mathcal{T}_R^{\Subset}(P))\right)(n) &= \{(n',\, D_n'^{\Subset}(\mathcal{R}(\mathcal{T}_R^{\Subset}(P)))) \mid n \in N^*\}(n) \\
&= D_n^{\Subset}(\mathcal{R}(\mathcal{T}_R^{\Subset}(P))) \\
&\subseteq \mathsf{MVP}_F(n) \\
&= \mathcal{R}_{\text{IFDS}}(P)(n).
\end{aligned}$$

□

We will next show, in Lemma 3.2.13, that our analysis is sound, i.e. that the result of an IDE problem obtained through a correlated-calls transformation removes only facts that occur on infeasible paths. To prove the Soundness Lemma, we first introduce Lemmas 3.2.11 and 3.2.12.

We will denote the top element in the environment lattice as

$$\Omega = \lambda d\,.\,\top_{\Subset}.$$

For the purpose of the proofs, we will rewrite Equation (3.5) that defines an edge function as follows:

$$\mathsf{EdgeFn}_S^{\Subset} = \lambda e \,.\, \begin{cases} \mathsf{id} & \text{if } d_1 = d_2 = \Lambda, \\ \lambda m \,.\, \varepsilon(e)(\delta(m)) & \text{otherwise,} \end{cases} \tag{3.10}$$

where $S \subseteq R$, d_1 and d_2 are the source and target facts, and for a map $m \in L_U^{\Subset}$, $\delta(m)$ is either m or $\bot_{\Subset}$:

$$\delta(m) = \begin{cases} \bot_{\Subset} & \text{if } d_1 = \Lambda \\ m & \text{otherwise.} \end{cases} \tag{3.11}$$

Additionally, for a path $p = [\mathsf{start}_{\mathtt{main}}, \ldots]$ and a fact $d \in D$, we will denote the lattice element that is mapped to d according to the flow functions of path p as follows:

$$\xi(p,\, d) = M_{\mathsf{Env}}(p)(\Omega)(d).$$

The following Lemma shows that the lattice elements (receiver-to-types maps) of a correlated-calls IDE analysis correctly overapproximate the possible types of a receiver in a program execution.

Lemma 3.2.11. *Let $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n]$ be some concrete execution trace of the program, and let $r \in R$ be a receiver. If after the execution trace p, at node n, r points to an object of runtime type t, and $d \in D$ is a fact such that $d \in M_F(p)(\varnothing)$, then*

$$t \in \xi(p,\, d)(r).$$

Proof. By induction on the length of the trace.

Basis: $p = [\mathsf{start}_{\mathtt{main}}]$. Then there is no instruction at which a receiver r could be instantiated, and the Lemma is trivially true.

Induction hypothesis: Let $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1}]$, and let τ be the set of types to which $\xi(p,\, d_{k-1})$ maps r:

$$\tau = \xi(p,\, d_{k-1})(r).$$

Assume that for a concrete execution path $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1}]$, at node $(n_{k-1},\, d_{k-1})$, the Lemma holds, i.e. $t \in \tau$.

Induction step: Let $p' = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1},\, n_k]$ and $t' \in T$ be the type to which r is mapped at n_k.

For each i, let e_i be the edge $((n_{i-1},\, d_{i-1}),\, (n_i,\, d_i))$. Note that $e_1 = ((\mathsf{start}_{\mathtt{main}},\, \Lambda),\, (n_1,\, d_1))$.

Observe that

$$\begin{aligned}\xi(p',\, d) &= M_{\mathsf{Env}}(p')(\Omega)(d)\\ &= \left(M_{\mathsf{Env}}(e_k) \circ M_{\mathsf{Env}}(e_{k-1}) \circ \ldots \circ M_{\mathsf{Env}}(e_1)\right)(\Omega)(d)\\ &= M_{\mathsf{Env}}(e_k)\left(M_{\mathsf{Env}}(p)(\Omega)\right)(d).\end{aligned}$$

According to (3.1),

$$\begin{aligned}&M_{\mathsf{Env}}(e_k)\left(M_{\mathsf{Env}}(p)(\Omega)\right)(d)(r)\\ &= \left(\mathsf{EdgeFn}_R^{\Subset}((n_{k-1},\, \Lambda),\, (n_k,\, d))(\top_{\Subset}) \sqcap \bigsqcap_{d' \in D} \mathsf{EdgeFn}_R^{\Subset}((n_{k-1},\, d'),\, (n_k,\, d))(M_{\mathsf{Env}}(p)(\Omega)(d'))\right)(r)\\ &\supseteq \bigsqcap_{d' \in D} \mathsf{EdgeFn}_R^{\Subset}((n_{k-1},\, d'),\, (n_k,\, d))(M_{\mathsf{Env}}(p)(\Omega)(d'))(r)\\ &\supseteq \mathsf{EdgeFn}_R^{\Subset}((n_{k-1},\, d_{k-1}),\, (n_k,\, d))(\xi(p,\, d_{k-1}))(r).\end{aligned}$$

Therefore,

$$\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) \subseteq \xi(p',\, d)(r). \tag{3.12}$$

We will now show that

$$t' \in \mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r),$$

which, due to (3.12), means that the Lemma holds.

According to (3.10), there are two cases in which $\mathsf{EdgeFn}_R^{\Subset}(e_k)$ could fall.

If $d_{k-1} = d_k = \Lambda$, then $d_k \notin M_F(p)(\varnothing)$, since it does not belong to the set D, and the Lemma trivially holds.

Otherwise,

$$\mathsf{EdgeFn}_R^{\Subset}(e_k) = \lambda m\, .\varepsilon(e_k)(\delta(m)).$$

It follows that

$$\begin{aligned}\mathsf{EdgeFn}_R^{\Subset}(e_k)(\xi(p,\, d_{k-1}))(r) &= (\lambda m\, .\varepsilon(e_k)(\delta(m)))(\xi(p,\, d_{k-1}))(r)\\ &= \varepsilon(e_k)(\delta(\xi(p,\, d_{k-1})))(r).\end{aligned} \tag{3.13}$$

Let us denote the lattice element $\delta(\xi(p,\, d_{k-1}))$ with Δ:

$$\Delta = \delta(\xi(p,\, d_{k-1})).$$

Note that since Δ, according to (3.11), can be either $\bot_{\Subset}$ or $\xi(p,\, d_{k-1})$, it always maps r to a set containing t:

$$t \in \Delta(r). \tag{3.14}$$

Note also that unless the instruction at n_{k-1} contains an assignment for r, r is mapped to the same object of type t as at node n_{k-1}, and $t = t'$. Therefore, for the non-assignment instructions, it is sufficient to prove that $t \in \Delta(r)$.

Depending on the instructions at the nodes n_{k-1} and n_k, there are four cases:

1. The instruction at n_{k-1} is an assignment for a receiver $r' \in R$. Since $\varepsilon_R(e_k) = \lambda m \,.\, m[r' \to \bot_T]$,

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) &= (\lambda m \,.\, m[r' \to \bot_T])(\Delta)(r)\\ &= \Delta[r' \to \bot_T](r).\end{aligned}$$

   In the resulting map, r' is mapped to $\bot_T$. Then

   (a) if $r = r'$, then $\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) = \bot_T$, which contains t'.

   (b) If $r \neq r'$, then r has not been reassigned a value, and still maps to the same object of type t. The receiver r is mapped to $\Delta(r)$, which, according to (3.14), contains t. Since $t = t'$, $\Delta(r)$ contains t'.

2. e_k is a call-start edge with signature $s_{\mathcal{F}}$, and $f \in \mathcal{F}$ is the called procedure. Then

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) &= (\lambda m \,.\, m[r' \to m(r') \cap \tau(s_{\mathcal{F}},\, f)])(\Delta)(r)\\ &= \Delta[r' \to \Delta(r') \cap \tau(s_{\mathcal{F}},\, f)],\end{aligned}$$

   where r' is the receiver of the call.

   - If $r' = r$, then $\Delta(r') = \Delta(r)$ which contains t. Since $t \in \tau(s_{\mathcal{F}},\, f)$, it follows that $t \in \Delta(r) \cap \tau(s_{\mathcal{F}},\, f)$, and $t \in \mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r)$.
   - If $r' \neq r$, see (1b).

3. e_k is an end-return edge, $r_1, \ldots, r_k \in R$ are the local variables in the callee method, r' is the receiver of the call site corresponding to the return node n_k, and $f \in \mathcal{F}$ is the called method with signature $s_{\mathcal{F}}$. Then

$$\varepsilon_R(e_k) = \lambda m \,.\, m[r' \to m(r') \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T] \ldots [r_k \to \bot_T].$$

   If $r \in \{r_1, \ldots, r_k\}$, see Case 1. Otherwise, the case is analogous to Case 2.

4. The node contains any other instruction. Then

$$\mathsf{EdgeFn}^{\Subset}_{R}(e_k)(\xi(p,\, d_{k-1}))(r) = \mathsf{id}(\Delta)(r) = \Delta(r),$$

which contains t according to (3.14). □

We will now show that on a node of a concrete execution path, the correlated-calls analysis does not map receivers to $\top_T$. In other words, the analysis never considers nodes of a concrete execution path unreachable.

Lemma 3.2.12. *Let $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n]$ be a concrete execution path, $r \in R$ a receiver, and $d \in D$ a data-flow fact. Then if $d \in M_F(p)(\varnothing)$,*

$$\xi(p,\, d)(r) \neq \top_T.$$

Proof. By induction on the length of the execution trace.

Basis: Let $p = [\mathsf{start}_{\mathtt{main}}]$. Since the only realizable path corresponding to p is $[(\mathsf{start}_{\mathtt{main}},\, \Lambda)]$, there is no fact $d \in D$ such that $d \in M_F(p)(\varnothing)$, and the claim follows immediately.

Induction hypothesis: Let $p = [\mathsf{start}_{\mathtt{main}}, \ldots, n_{k-1}]$. Let τ be the set of types to which r is mapped by $\xi(p,\, d_{k-1})$:

$$\tau = \xi(p,\, d_{k-1})(r).$$

Assume the Lemma holds for that for a concrete execution path $p = [\mathsf{start}_{\mathtt{main}},\, n_1, \ldots, n_{k-1}]$, i.e. $\tau \neq \top_T$ for an arbitrary $r \in R$ and $d_{k-1} \in D$.

Induction step: Let $p' = [\mathsf{start}_{\mathtt{main}},\, n_1, \ldots, n_{k-1},\, n_k]$ be a concrete execution path.

Let $e_k = ((n_{k-1},\, d_{k-1}),\, (n_k,\, d))$. As shown in (3.12),

$$\xi(p',\, d)(r) \supseteq \mathsf{EdgeFn}^{\Subset}_{R}(e_k)(\xi(p,\, d_{k-1}))(r).$$

From Definition 3.2.7, we can see that unless e_k is a call-start edge or an end-return edge, the result follows from the induction hypothesis. More formally, if e_k is not a call-start or end-return edge, then for all $m \in L^{\Subset}_R$,

$$\mathsf{EdgeFn}^{\Subset}_{R}(e_k)(m) \sqsubseteq m.$$

The edge function corresponding to the call-start and end-return edges is the only place in which the set of types that a receiver maps to can be reduced.

Assume that e_k is a end-return edge with a call on the receiver $r' \in R$ with a signature $s_{\mathcal{F}}$ to a function $f \in \mathcal{F}$.

$$\begin{aligned}
&\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) \\
&\qquad = (\lambda m\,.\, m[r' \to m(r) \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T] \ldots [r_l \to \bot_T])\,(\xi(p,\, d_{k-1}))(r) \\
&\qquad = (\xi(p,\, d_{k-1})[r' \to \tau \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T] \ldots [r_l \to \bot_T])\,(r),
\end{aligned}$$

where $r_1, \ldots, r_l \in R$ are the local variables in the called method.

If $r \in \{r_1, \ldots, r_l\}$, then $\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) = \bot_T \ni t$[8].

Otherwise, if $r = r'$, then $\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) = \tau \cap \tau(s_{\mathcal{F}},\, f)$.

According to Lemma 3.2.11 and by the induction hypothesis, the runtime type t of r must be contained in $\xi(p,\, d_{k-1})(r) = \tau$. At the same time, by definition, t is part of $\tau(s_{\mathcal{F}},\, f)$. Therefore, $t \in \tau \cap \tau(s_{\mathcal{F}},\, f) \subseteq \mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r)$, which means that $\mathsf{EdgeFn}^{\Subset}_R(e_k)(\xi(p,\, d_{k-1}))(r) \neq \top_T$.

The same reasoning applies to the case where e_k is a call-start edge. □

Finally, we will prove the soundness of the correlated-calls analysis: we will show that our analysis only considers a path infeasible if it cannot occur in a concrete execution of a program.

Lemma 3.2.13 (Soundness)**.** *Let $p = [\mathit{start}_{\mathit{main}}, \ldots, n]$ be a concrete execution path, and let $d \in D$. If $d \in M_F(p)(\varnothing)$, then*

$$d \in \mathcal{U}^{\Subset}\left(\mathcal{R}(\mathcal{T}^{\Subset}_R(P))\right)(n).$$

Proof. Let $\rho = \mathcal{R}(\mathcal{T}^{\Subset}_R(P))$. Then

$$\begin{aligned}
\mathcal{U}^{\Subset}(\rho)(n) &= D^{\Subset}_n(\rho) \\
&= \{d' \mid d' \in \mathsf{MVP}_F(n) \wedge \forall r \in R :\ \rho(n,\ d')(r) \neq \top_T\}.
\end{aligned}$$

Since $\mathsf{MVP}_F(n) = \bigsqcap_{q \in \mathsf{VP}(n)} M_F(q)(\varnothing)$, and $p \in \mathsf{VP}(n)$, it follows that

$$\begin{aligned}
d &\in M_F(p)(\varnothing) \\
&\subseteq \mathsf{MVP}_F(n).
\end{aligned}$$

[8] In the case of a recursive call, it is possible that both $r \in \{r_1, \ldots, r_l\}$ and $r = r'$. In that case, the set to which r will be mapped would be still "overwritten" by $\bot_T$.

At the same time, for all receivers $r \in R$,

$$\rho(n,\, d)(r) = \left(\prod_{q \in \mathsf{VP}(n)} \xi(q,\, d) \right)(r) = \prod_{q \in \mathsf{VP}(n)} \xi(q,\, d)(r).$$

According to Lemma 3.2.12, $\xi(p,\, d)(r) \neq \top_T$. Since $p \in \mathsf{VP}(n)$,

$$\xi(p,\, d)(r) \subseteq \prod_{q \in \mathsf{VP}(n)} \xi(q,\, d)(r).$$

From $\prod_{q \in \mathsf{VP}(n)} \xi(q,\, d)(r) = \rho(n,\, d)(r)$ it follows that $\xi(p,\, d)(r) \subseteq \rho(n,\, d)(r)$. Therefore, $\rho(n,\, d)(r) \neq \top_T$, and $d \in D_n^{\Subset}(\rho) = \mathcal{U}^{\Subset}(\rho)(n)$. □

3.2.2.5 Correlated-Call Receivers

We will now show that in a correlated-calls transformation, it is enough to consider only some of the receivers of set R.

Definition 3.2.14. *Let c_1 and c_2 be two call sites on a receiver $r \in R$. If both call sites are polymorphic, then we say that r is a correlated-call receiver.*

In other words, a correlated-call receiver is a receiver that has at least two polymorphic call invocations. We will denote the set of correlated-call receivers as $R^{\Subset}$.

We will describe a "reduced" correlated-calls transformation in which we only consider receivers from $R^{\Subset}$ and ignore other receivers of R. We will show that IDE problems obtained through ordinary and reduced correlated-calls transformations yield the same results.

The following Lemma shows that the types to which a given receiver is mapped in the result of the algorithm is not affected by other receivers and the types to which they are mapped.

Lemma 3.2.15. *Let P be an IFDS problem. Let N^* be the supergraph for P, D the set of data-flow facts, $n \in N^*$ a node, and $p = [\mathsf{start}_{main}, \ldots, n]$ a path in the supergraph. Let $d \in D \cup \{\Lambda\}$. Then for any realizable path $p' \in \mathsf{RP}(p,\, d)$, set $S \subseteq R$, and receiver $r \in S$,*

$$\mathsf{EdgeFn}_S^{\Subset}(p')(\top_{\Subset})(r) = \mathsf{EdgeFn}_{\{r\}}^{\Subset}(p')(\top_{\Subset})(r).$$

Proof. By induction on the length of p.

Basis: $p' = [(\mathsf{start}_{\mathtt{main}}, \Lambda)]$. Then $\mathsf{EdgeFn}^{\Subset}_{S}(p') = \mathsf{id} = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')$, and the Lemma follows directly.

Induction hypothesis: Suppose that for a path $q = [(\mathsf{start}_{\mathtt{main}}, \Lambda), \ldots, (n_{k-1}, d_{k-1})]$, where $q \in \mathsf{RP}(n, d)$, the Lemma holds, i.e. both edge functions map r to the same set of types τ:

$$\begin{aligned}\tau &= \mathsf{EdgeFn}^{\Subset}_{S}(q)(\top_{\Subset})(r) \\ &= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q)(\top_{\Subset})(r).\end{aligned}$$

Induction step: Let $q' = [(\mathsf{start}_{\mathtt{main}}, \Lambda), \ldots, (n_{k-1}, d_{k-1}), (n_k, d_k)]$ and the edge $e_k = ((n_{k-1}, d_{k-1}), (n_k, d_k))$.

Observe that for any set $U \subseteq R$ such that $r \in U$,

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \mathsf{EdgeFn}^{\Subset}_{U}(e_k)(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r). \tag{3.15}$$

We can see from (3.10) that there are two cases.

If $d_{k-1} = d_k = \Lambda$, $\mathsf{EdgeFn}^{\Subset}_{S}(e_k) = \mathsf{id} = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_k)$, and, due to (3.15),

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_{S}(q')(\top_{\Subset})(r) &= \tau \\ &= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r).\end{aligned}$$

Otherwise, there are four sub-cases.

1. e_k is a call-start edge, $r'.c()$ is the call site at n_{k-1} with signature $s_{\mathcal{F}}$, $f \in \mathcal{F}$ is the called procedure, and $r' \in U$. Then

   $$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)[r' \to \delta(m)(r) \cap \tau(s_{\mathcal{F}}, f)].$$

   There are two sub-cases.

   (a) If $r = r'$, then, according to (3.15), the resulting set of types

   $$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r) \cap \tau(s_{\mathcal{F}}, f).$$

   If $d_{k-1} = \Lambda$, then $\delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r) = \bot_{\Subset}(r) = \bot_{T}$. If $d_{k-1} \neq \Lambda$, then $\delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r) = \mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset})(r) = \tau$. The set $\tau(s_{\mathcal{F}}, f)$ is the same for either case.

   Therefore, the value of $\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r)$ has the same result regardless of U, which means that $\mathsf{EdgeFn}^{\Subset}_{S}(q')(\top_{\Subset})(r) = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r)$, and the Lemma holds.

(b) If $r \neq r'$, then

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \delta(\mathsf{EdgeFn}^{\Subset}_{U}(q)(\top_{\Subset}))(r),$$

which, as we have seen in Case (1a), does not depend on U, and the Lemma holds.

2. e_k is an end-return edge, $r_1, \ldots, r_l \in U$ are the local variables in the callee method, $r'.c()$ is the call corresponding to the return node at n_k, $f \in \mathcal{F}$ is the called method with signature $s_{\mathcal{F}}$, and $r' \in U$. Then

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)[r' \to \delta(m)(r) \cap \tau(s_{\mathcal{F}},\, f)][r_1 \to \bot_T] \ldots [r_l \to \bot_T].$$

There are three sub-cases.

(a) If $r \in \{r_1, \ldots, r_l\}$, then regardless of the value of U,

$$\mathsf{EdgeFn}^{\Subset}_{U}(q')(\top_{\Subset})(r) = \bot_T,$$

and the Lemma holds.

(b) Otherwise, if $r = r'$, the case is analogous to Case (1a).

(c) If $r \notin \{r', r_1, \ldots, r_l\}$, then see Case (1b).

3. n_{k-1} contains an assignment for $r' \in U$. Then

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m)[r' \to \bot_T].$$

If $r = r'$, see Case (2a). If $r \neq r'$, see Case (1b).

4. Otherwise,

$$\mathsf{EdgeFn}^{\Subset}_{U}(e_k) = \lambda m\,.\,\delta(m),$$

and the case is analogous to Case (1b). □

The following Lemma shows that the correlated-calls analysis computes the results for each receiver independently, or separately. To compute the set of types to which a receiver r is mapped at each exploded-graph node, we can exclude all other receivers in the program from the analysis (recall from (3.5) that the set of receivers that are considered in the analysis is specified by the set S in a correlated-calls transformation $\mathcal{T}^{\Subset}_{S}$). Therefore, for a given receiver r, the results of a $\mathcal{T}^{\Subset}_{S}$- and a $\mathcal{T}^{\Subset}_{\{r\}}$-analysis are the same.

Lemma 3.2.16. *Let P be an IFDS problem. Let N^* be the supergraph for P, D the set of data-flow facts, and $S \subseteq R$ a set of receivers. Then for any $n \in N^*$, $d \in D$, and receiver $r \in S$,*

$$\mathcal{R}\left(\mathcal{T}_S^{\Subset}(P)\right)(n,\, d)(r) = \mathcal{R}(\mathcal{T}_{\{r\}}^{\Subset}(P))(n,\, d)(r).$$

Proof. According to (3.3), (2.1), and (3.2),

$$\begin{aligned}
\mathcal{R}\left(\mathcal{T}_S^{\Subset}(P)\right)(n,\, d)(r) &= \mathsf{MVP}_{\mathsf{Env}}(n,\, d)(r) \\
&= \left(\prod_{q \in \mathsf{VP}(n)} M_{\mathsf{Env}}(q)(\Omega)(d)\right)(r) \\
&= \left(\prod_{q \in \mathsf{VP}(n)} \prod_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}_S^{\Subset}(q')(\top_{\Subset})\right)(r) \\
&= \bigcup_{q \in \mathsf{VP}(n)} \bigcup_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}_S^{\Subset}(q')(\top_{\Subset})(r). \qquad (3.16)
\end{aligned}$$

Then from Lemma 3.2.15,

$$\begin{aligned}
\mathcal{R}\left(\mathcal{T}_S^{\Subset}(P)\right)(n,\, d)(r) &= \bigcup_{q \in \mathsf{VP}(n)} \bigcup_{q' \in \mathsf{RP}(q,\, d)} \mathsf{EdgeFn}_{\{r\}}^{\Subset}(q')(\top_{\Subset})(r) \\
&= \mathcal{R}\left(\mathcal{T}_{\{r\}}^{\Subset}(P)\right)(n,\, d)(r). \qquad \square
\end{aligned}$$

The next lemma shows that the set of types to which a receiver is mapped in a correlated-calls lattice element can be represented as an intersection of static-type function applications $\tau(s_{\mathcal{F}_i},\, f_i)$.

Lemma 3.2.17. *For an IFDS problem P, a node $n \in N^*$, and fact $d \in D$, let $p \in \mathsf{RP}(n,\, d)$ be a realizable path and $r \in R$ a receiver. Then there exists a non-negative number γ of calls on the receiver r with signatures $s_{\mathcal{F}_\gamma}$ to the functions $f_\gamma \in \mathcal{F}_\gamma$, for which*

$$\mathsf{EdgeFn}_{\{r\}}^{\Subset}(p)(\top_{\Subset})(r) = \bigcap_{\gamma \geq 0} \tau(s_{\mathcal{F}_\gamma},\, f_\gamma).$$

Proof. Let p have the following form[9]:

$$p = [(\mathsf{start}_{\mathtt{main}},\, \Lambda),\, (n_1,\, \Lambda),\, \ldots,\, (n_k,\, \Lambda), (n_{k+1},\, d_{k+1}),\, \ldots,\, (n_{k+l},\, d_{k+l})],$$

[9]It can be shown from the definition of a pointwise representation in Sagiv et al. [21] that in a realizable path, there is never an edge from a fact of the set D to a Λ fact. Therefore, we can represent p as a sequence of nodes that has a prefix of Λ-fact nodes, after which all nodes are non-Λ facts.

where $l \geq 1$ and the facts for all nodes up to n_k are equal to Λ and $d_{k+i} \in D$ for $0 < i \leq l$. As previously, for all i, we will denote the edge (n_i, n_{i+1}) by e_i.

From (3.5) we can infer that

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p) = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l}) \circ \ldots \circ \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+2}) \circ (\lambda m \,.\, \beta) \circ \mathsf{id} \circ \ldots \circ \mathsf{id},$$

where

$$\beta = \begin{cases} \bot_{\Subset}[r \to \tau(s_{\mathcal{F}},\, f)] & \text{if } (n_k,\, n_{k+1}) \text{ is a call-start or end-return edge, and the call} \\ & \text{site } r.c() \text{ with signature } s_{\mathcal{F}} \text{ to the function } f \in \mathcal{F} \text{ corresponds} \\ & \text{to the call-start or end-return edge,} \\ \bot_{\Subset} & \text{otherwise}[10]. \end{cases}$$

Therefore,

$$\begin{aligned} \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset}) &= \left(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l}) \circ \ldots \circ \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+2})\right)((\lambda m \,.\, \beta)(\top_{\Subset})) \\ &= \left(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l}) \circ \ldots \circ \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+2}) \circ \mathsf{id}\right)(\beta). \end{aligned} \tag{3.17}$$

We can now prove the lemma by induction on l.

Basis: If $l = 1$, then $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset}) = \mathsf{id}(\beta) = \beta$. There are two cases.

If $\beta = \bot_{\Subset}$, then

$$\begin{aligned} \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) &= \beta(r) \\ &= \bot_T, \end{aligned}$$

and $\gamma = 0$.

If $\beta = \bot_{\Subset}[r \to \tau(s_{\mathcal{F}},\, f)]$, then

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \tau(s_{\mathcal{F}},\, f),$$

and $\gamma = 1$.

[10] Since $d_k = \Lambda$ and $d_{k+1} \neq \Lambda$, the micro function for the edge e_{k+1} is equal to $\lambda m \,.\, \varepsilon_{\{r\}}(e_{k+1})(\bot_{\Subset})$. From the definition of ε_S (3.6) we can see that the only case where $\varepsilon_{\{r\}}(e_{k+1})(m)$ would not be equal to $\bot_{\Subset}$ is when e_{k+1} is call-start or end-return edge.

Induction hypothesis: Assume that for a path $p = [(\mathsf{start}_{\mathtt{main}}, \Lambda), \ldots, (n_{k+l}, d_{k+l})]$, the Lemma holds for $\gamma = N$, where $N \geq 0$.

Induction step: Let $p' = [(\mathsf{start}_{\mathtt{main}}, \Lambda), \ldots, (n_{k+l}, d_{k+l}), (n_{k+l+1}, d_{k+l+1})]$.

Recall that

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')(\top_{\Subset})(r) = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1}) \left(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})\right)(r).$$

From (3.6) we can see that unless e_{k+l+1} is a call-start or end-return edge corresponding to a call on the receiver r, then $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})(r)$ must be equal to either $\bot_T$ or $m(r)$, where $m = \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})$.

If $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})(r) = \bot_T$, then the Lemma holds for $\gamma = 0$.

Otherwise,

$$\begin{aligned}\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1})(\top_{\Subset})(r) &= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) \\ &= \bigcap_N \tau(s_{\mathcal{F}_N}, f_N),\end{aligned}$$

and therefore $\gamma = N$.

Suppose that e_{k+l+1} is a call-start edge with a call on the receiver r with signature $s_{\mathcal{G}}$ to a function $g \in \mathcal{G}$. Then, according to (3.6),

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(e_{k+l+1}) = \lambda m \,.\, m[r \to m(r) \cap \tau(s_{\mathcal{G}}, g)].$$

Therefore,

$$\begin{aligned}&\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')(\top_{\Subset})(r) \\ &\quad= \lambda m \,.\, m[r \to m(r) \cap \tau(s_{\mathcal{G}}, g)] \left(\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})\right)(r) \\ &\quad= \mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) \cap \tau(s_{\mathcal{G}}, g) \\ &\quad= \left(\bigcap_N \tau(s_{\mathcal{F}_N}, f_N)\right) \cap \tau(s_{\mathcal{G}}, g),\end{aligned}$$

and the Lemma holds for $\gamma = N + 1$.

The case where e_{k+l+1} is an end-return edge is analogous to the previous case. □

We now show that a receiver will be only mapped to $\top_{\Subset}$ if it is the receiver of a correlated call.

Lemma 3.2.18. *For an IFDS problem P, let $n \in N^*$ be a node, and $d \in D$ a dataflow fact such that there exists a realizable path $p \in \mathit{RP}(n, d)$. Let T be the set of all types in the program. If there exists a receiver $r \in R$ such that*

$$\mathit{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \top_T,$$

then $r \in R^{\Subset}$.

Proof. According to Lemma 3.2.17,

$$\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p)(\top_{\Subset})(r) = \bigcap_{\gamma \geq 0} \tau(s_{\mathcal{F}_\gamma}, f_\gamma).$$

Let $\tau_i = \tau(s_{\mathcal{F}_i}, f_i)$. For a given k, let $r.m_k()$ be the call site corresponding to τ_k, and T' the set of types compatible with the static type of r. Recall from Section 3.2.2.3 that

- $\tau_k \neq \top_T$;
- if $\tau_k = T'$ then the corresponding call site is monomorphic;
- if $\tau_k \subset T'$ then the call site is polymorphic.

From the conditions of the Lemma,

$$\bigcap_{\gamma \geq 0} \tau_\gamma = \top_T.$$

If all $\tau_k = T'$, then $\bigcap_{\gamma \geq 0} \tau_\gamma$ is also equal to T'. Since $T' \neq \top_T$, this is a contradiction.

If exactly one $\tau_k \subset T'$ and the rest are equal to T', then $\bigcap_{\gamma \geq 0} \tau_\gamma$ is equal to τ_k, which cannot be $\top_T$ either.

Therefore, there are at least two sets, τ_i and τ_j, which are strict subsets of T'. Since both τ_i and τ_j are non-empty and their intersection equals $\top_T$, τ_i and τ_j must be disjoint. If τ_i and τ_j are disjoint, they must correspond to different call sites.

In other words, there are at least two calls on the same receiver for which the static-type function is a strict subset of the set of types compatible with a given receiver r. It follows that both calls have to be polymorphic. Therefore, $r \in R^{\Subset}$. □

We will now show that if a receiver ever gets mapped to top, then it is a correlated-calls receiver.

Lemma 3.2.19. *For an IFDS problem P, let $n \in N^*$ be a node, and $d \in D$ a dataflow fact such there exists a realizable path $p \in \mathsf{RP}(n, d)$. Then, if there exists a receiver $r \in R$, such that*

$$\mathcal{R}\left(\mathcal{T}^{\Subset}_{\{r\}}(P)\right)(n, d)(r) = \top_T,$$

then $r \in R^{\Subset}$.

Proof. As shown in (3.16),

$$\mathcal{R}\left(\mathcal{T}^{\Subset}_{\{r\}}(P)\right)(n, d)(r) = \bigcup_{q \in \mathsf{VP}(n)} \bigcup_{q' \in \mathsf{RP}(q, d)} \mathsf{EdgeFn}^{\Subset}_{\{r\}}(q')(\top_{\Subset})(r).$$

Since the latter is equal to $\top_T$, it follows that for each realizable path p' to node n, $\mathsf{EdgeFn}^{\Subset}_{\{r\}}(p')(\top)(r) = \top_T$. According to Lemma 3.2.19, this is only possible if $r \in R^{\Subset}$. □

Finally, we show that if a correlated calls analysis considers only correlated-call receivers, no precision is lost. A correlated-calls analysis that considers all receivers computes the same result as an analysis that considers only correlated-call receivers.

Lemma 3.2.20. *Let P be an IFDS problem. Then*

$$\mathcal{U}^{\Subset}\left(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R^{\Subset}}(P)\right)\right) = \mathcal{U}^{\Subset}\left(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R}(P)\right)\right).$$

Proof. From (3.8) we know that

$$\mathcal{U}^{\Subset}\left(\mathcal{R}\left(\mathcal{T}^{\Subset}_{R}(P)\right)\right) = \left\{(n, D^{\Subset}_n(\mathcal{R}(\mathcal{T}^{\Subset}_{R}(P)))) \mid n \in N^*\right\}.$$

According to (3.7) and Lemma 3.2.16, for a given $n \in N^*$,

$$\begin{aligned}
&D^{\Subset}_n(\mathcal{R}(\mathcal{T}^{\Subset}_{R}(P))) \\
&\quad = \left\{d \mid d \in \mathsf{MVP}_F(n) \wedge \forall r \in R : \left\{(r, \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n, d)(r)) \mid r \in R\right\}(r) \neq \top_T\right\} \\
&\quad = \left\{d \mid d \in \mathsf{MVP}_F(n) \wedge \forall \boldsymbol{r} \in \boldsymbol{R} : \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n, d)(r) \neq \top_T\right\}.
\end{aligned}$$

Since, according to Lemma 3.2.19, $\mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n, d)(r)$ can only be equal to $\top_T$ when $r \in R^{\Subset}$, we can conclude that

$$\begin{aligned}
&D^{\Subset}_n(\mathcal{R}(\mathcal{T}^{\Subset}_{R}(P))) \\
&\quad = \left\{d \mid d \in \mathsf{MVP}_F(n) \wedge \forall \boldsymbol{r} \in \boldsymbol{R}^{\Subset} : \mathcal{R}(\mathcal{T}^{\Subset}_{\{r\}}(P))(n, d)(r) \neq \top_T\right\} \\
&\quad = D^{\Subset}_n(\mathcal{R}(\mathcal{T}^{\Subset}_{R^{\Subset}}(P))).
\end{aligned}$$

Therefore,

$$\begin{aligned}\mathcal{U}^{\Subset}(\mathcal{R}\,(\mathcal{T}_R^{\Subset}(P))) &= \{(n,\, D_n^{\Subset}(\mathcal{R}(\mathcal{T}_{R^{\Subset}}^{\Subset}(P)))) \mid n \in N^*\} \\ &= \mathcal{U}^{\Subset}(\mathcal{R}\,(\mathcal{T}_{R^{\Subset}}^{\Subset}(P))).\end{aligned}$$ □

To summarize, Lemma 3.2.13 shows that the result $\mathcal{R}_{\Subset}$ of a correlated-calls analysis is sound since it overapproximates the data flow of all possible concrete execution paths. We have also shown in Lemma 3.2.10 that the correlated-calls analysis improves the precision of the original IFDS result $\mathcal{R}_{\text{IFDS}}$, because the correlated-calls result $\mathcal{R}_{\Subset}$ underapproximates an equivalence-IDE result $\mathcal{R}_{\equiv} = \mathcal{R}_{\text{IFDS}}$. Finally, we showed that a correlated-call transformation to IDE that considers only correlated-call receivers $R^{\Subset}$ achieves the same result $\mathcal{R}_{\Subset}$ that is obtained when considering all receivers R.

This is the general idea of the correlated-calls analysis. The analysis involves a transformation from IFDS to IDE problems. To implement an IDE problem, it is necessary to define a representation of lattice elements and micro functions. An efficient representation of those data structures for the correlated-calls analysis is presented in the next chapter.

Chapter 4

Correlated Calls Representations

In order to define a correlated-calls transformation, we need to represent lattice elements $L^{\Subset}_{R^{\Subset}} : R^{\Subset} \to 2^T$ of the target IDE problem, which are functions from receivers to sets of types, and micro functions $L^{\Subset}_{R^{\Subset}} \to L^{\Subset}_{R^{\Subset}}$.

As defined in Sagiv et al. [21], a representation of micro functions is efficient if the following conditions hold:

1. There is a representation for the identity and top functions.
2. The representation is closed under the meet and composition operations.
3. The micro functions form a finite-height lattice.
4. The apply, meet, composition, and equality-check operations can be computed in constant time.
5. There is a constant bound on the storage space for a micro function representation.

We will distinguish the representation of a concept from its denotation. For a concept c, we will write $[\![c]\!]$ for its denotation and just c for its representation. For example, if we want to represent a constant function g with the constant value v that it returns, we will write for g's representation, $g = v$, and for g's denotation, $[\![g]\!] = \lambda x \,.\, v$.

4.1 Lattice Elements

Elements of the $L^{\Subset}_{R^{\Subset}}$ lattice can be represented with a map from receivers to sets of types. The bottom element maps each receiver to the set T of all types:

$$\bot_{\Subset} = \{(r, T) \mid r \in R^{\Subset}\}$$

and the top element maps each receiver to the empty set:

$$\top_{\Subset} = \{(r, \varnothing) \mid r \in R^{\Subset}\}\,.$$

4.2 Micro Functions

In the context of the correlated-calls transformation to an IDE problem, a lattice element is a map from receivers to sets of types. Thus, a micro function transforms, or updates, an existing receiver-to-types map with new information about the types of a receiver.

We will represent micro functions with *update maps* which we describe in the next section.

4.2.1 Update Maps

To represent micro functions that transform maps from receivers to sets of types, we use *update maps*. To define update maps, we first introduce the notions of *update functions* and *normalization*.

Let f be a micro function, $r \in R^{\Subset}$ a correlated-call receiver, and T the set of types in a program.

Definition 4.2.1. *A non-normalized update function* $\mathsf{update}^*_{f,r}$ *is a pair of sets*

$$\mathsf{update}^*_{f,r} = \langle I_{f,r}, U_{f,r} \rangle\,,$$

where $I_{f,r} \subseteq T$ *is called the intersection set and* $U_{f,r} \subseteq T$ *the union set of the update function.*

Definition 4.2.2. *Let* $\langle I, U \rangle$ *be a pair of sets. The normalization function* $\mathcal{N}$ *is defined as*

$$\mathcal{N}(\langle I, U \rangle) = \langle I \cup U, U \rangle\,.$$

Definition 4.2.3. *An update function* $\mathsf{update}_{f,r}$ *is a normalized pair of sets*

$$\mathsf{update}_{f,r} = \mathcal{N}(\mathsf{update}^*_{f,r}) = \mathcal{N}(\langle I_{f,r},\, U_{f,r}\rangle).$$

Definition 4.2.4. *The update map of* f *is a map from receivers to update functions:*

$$\mathsf{update}_f = \{(r,\, \mathsf{update}_{f,r}) \,|\, r \in R^{\Subset}\}.$$

Thus, each micro function f is represented with an update map:

$$f = [\![\mathsf{update}_f]\!]\,.$$

4.2.2 Denotation of Update Maps

Intuitively, the meaning of a micro function is the update that it performs on a receiver-to-types map. To represent a micro function, it is enough to specify how the set of types for a given receiver has to be transformed. This is what the update map does.

For a micro function f, an update map takes a receiver and returns an update function:

$$[\![\mathsf{update}_f]\!] = \lambda r\,.\; [\![\mathsf{update}_{f,r}]\!]\,.$$

Given a micro function $f : (R^{\Subset} \to 2^T) \to (R^{\Subset} \to 2^T)$, an update map is defined so that

$$f(m) = \left\{ \left(r,\, [\![\mathsf{update}_f]\!]\,(r)(m(r)\right) \;\middle|\; r \in R^{\Subset} \right\},$$

where the update map $[\![\mathsf{update}_f]\!]$ has type $R^{\Subset} \to (2^T \to 2^T)$, and the update function $[\![\mathsf{update}_{f,r}]\!] = [\![\mathsf{update}_f]\!]\,(r)$ has type $2^T \to 2^T$.

For any receiver-to-types map m, an update function specifies two things:

- which elements of $m(r)$ should be preserved, and
- which new elements should be added to $m(r)$.

This can be achieved by maintaining the intersection $I_{f,r}$ and union set $U_{f,r}$, where

$$[\![\mathsf{update}_{f,r}]\!]\,(m(r)) = (m(r) \cap I_{f,r}) \cup U_{f,r}.$$

However, as we will see in Section 4.2.3, we need to be able to check update functions for equality, which is difficult to do with non-normalized update functions.

Example 4.2.5. For a non-empty set of types T, consider two update functions

$$u_1 = \langle T, T \rangle$$

and

$$u_2 = \langle \varnothing, T \rangle .$$

The denotations of the functions look as follows:

$$[\![u_1]\!] = \lambda t \,.\, (t \cap T) \cup T$$

and

$$[\![u_2]\!] = \lambda t \,.\, (t \cap \varnothing) \cup T.$$

We can see that both $[\![u_1]\!]$ and $[\![u_2]\!]$ are equal to the function $\lambda t \,.\, T$.

Therefore, the same function can have more than one non-normalized representation. This means that to check two functions for equality, it is not enough to compare their non-normalized representations.

This is why Definition 4.2.3 requires update functions to be normalized. Normalization makes the union set of the update function a subset of the intersection set. As we show later, normalization guarantees that each update function has a unique representation.

4.2.3 Equality of Micro Functions

We will now show that micro functions can be checked for equality if their representations use normalized update functions.

First, let us show that normalization does not change the behaviour of an update function. This means that the normalized and non-normalized versions of an update function always denote the same function.

Lemma 4.2.6. $[\![\mathcal{N}(\mathit{update}^*_{f,r})]\!] = [\![\mathit{update}^*_{f,r}.]\!]$

Proof. Let $\mathsf{update}^*_{f,r} = \langle I, U\rangle$. For any $\tau \in T$,

$$\begin{aligned}
[\![\mathcal{N}(\mathsf{update}^*_{f,r})]\!]\,(\tau) &= [\![\mathcal{N}(\langle I,\ U\rangle)]\!]\,(\tau) \\
&= [\![\langle I \cup U, U\rangle]\!]\,(\tau) \\
&= \tau \cap (I \cup U) \cup U \\
&= (\tau \cap I) \cup (\tau \cap U) \cup U \\
&= \tau \cap I \cup U \\
&= [\![\langle I, U\rangle]\!]\,(\tau) \\
&= [\![\mathsf{update}^*_{f,r}]\!]\,(\tau).
\end{aligned}$$

Thus, $[\![\mathcal{N}(\mathsf{update}^*_{f,r})]\!] = [\![\mathsf{update}^*_{f,r}.]\!]$. □

Let us show that two update functions are equal if and only if their normalized representations are equal.

It is obvious that two functions represented with the same pairs of sets are equal. Let us prove that two different pairs of sets represent different update functions.

Lemma 4.2.7. *Let $\langle I, U\rangle$ and $\langle I', U'\rangle$ be two normalized update functions such that $\langle I, U\rangle \neq \langle I', U'\rangle$. Then $[\![\langle I, U\rangle]\!] \neq [\![\langle I', U'\rangle]\!]$.*

Proof. Let us show that there always exists a set $\tau \subseteq T$ such that $[\![\langle I, U\rangle]\!]\,(\tau) \neq [\![\langle I', U'\rangle]\!]\,(\tau)$. There are two cases:

1. $U \neq U'$. Then for the empty set $\tau = \varnothing$,

   $$[\![\langle I, U\rangle]\!]\,(\tau) = [\![\langle I, U\rangle]\!]\,(\varnothing) = (\varnothing \cap I) \cup U = U,$$

   whereas

   $$[\![\langle I', U'\rangle]\!]\,(\tau) = [\![\langle I', U'\rangle]\!]\,(\varnothing) = (\varnothing \cap I') \cup U' = U'.$$

   Hence, $[\![\langle I, U\rangle]\!] \neq [\![\langle I', U'\rangle]\!]$.

2. $I \neq I'$. Then for the set of all types $\tau = T$,

   $$[\![\langle I, U\rangle]\!]\,(\tau) = [\![\langle I, U\rangle]\!]\,(T) = (T \cap I) \cup U = I \cup U.$$

Since $\langle I,\, U\rangle$ is normalized, $U \subseteq I$, and

$$I \cup U = I.$$

At the same time,

$$[\![\langle I',\, U'\rangle]\!]\,(\tau) = [\![\langle I',\, U'\rangle]\!]\,(T) = (T \cap I') \cup U' = I' \cup U' = I'.$$

Since $I \neq I'$, it follows that $[\![\langle I,\, U\rangle]\!] \neq [\![\langle I',\, U'\rangle]\!]$. □

We have shown that transfer functions can be represented using update maps.

4.2.4 Operations on Update Maps

Let us now define the apply, compose, meet, and equals functions on micro function representations.

For an update map f and a receiver $r \in R^{\Subset}$, let the update function $f(r) = \langle I_{f,r},\, U_{f,r}\rangle$.

Then the operations on the update maps f_1 and f_2 are defined as follows:

$$\mathsf{apply}_{f_1} = \lambda m\,.\,\{(r,\, (m(r) \cap I_{f,r}) \cup U_{f,r}) \mid r \in R^{\Subset}\},$$

$$f_1 \circ f_2 = \{(r,\, \mathcal{N}(\langle I_{f_1,r} \cap I_{f_2,r},\ (I_{f_1,r} \cap U_{f_2,r}) \cup U_{f_1,r}\rangle)) \mid r \in R^{\Subset}\},$$

$$f_1 \sqcap f_2 = \{(r,\, \langle I_{f_1,r} \cup I_{f_2,r},\ U_{f_1,r} \cup U_{f_2,r}\rangle) \mid r \in R^{\Subset}\},$$

$$\mathsf{equals}(f_1,\, f_2) = \begin{cases} \mathsf{true} & \text{if } f_1 \text{ and } f_2 \text{ are structurally equal,} \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

The denotation of the operations on update maps can be explained in the following way.

The apply function of a micro function f maps over all receivers. For each receiver $r \in R^{\Subset}$, $\mathsf{update}_f(r)$ transforms the argument receiver-to-types map m. It returns a new map in which r is mapped to a new set of types, $\mathsf{update}_{f,r}(m(r))$:

$$\begin{aligned} [\![\mathsf{apply}_f]\!] &= [\![\lambda m\,.\,\{(r,\, (m(r) \cap I_{f,r}) \cup U_{f,r}) \mid r \in R^{\Subset}\}]\!] \\ &= \lambda m\,.\,\left\{\left(r,\, [\![\mathsf{update}_{f,r}]\!]\,(m(r))\right) \mid r \in R^{\Subset}\right\}. \end{aligned}$$

Note that in the beginning of the algorithm, m maps each receiver to $\bot_T$ (all types).

Composing two micro functions means to compose their update-map denotations:

$$\begin{aligned}
[\![f_1 \circ f_2]\!] &= [\![\{(r,\, \mathcal{N}(\langle I_{f_1,r} \cap I_{f_2,r},\ (I_{f_1,r} \cap U_{f_2,r}) \cup U_{f_1,r}\rangle)) \mid r \in R^{\Subset}\}]\!] \\
&= {}^{11}\lambda m\,.\, \{(r,\, [\![\langle I_{f_1,r} \cap I_{f_2,r},\ (I_{f_1,r} \cap U_{f_2,r}) \cup U_{f_1,r}\rangle]\!]\,(m(r))) \mid r \in R^{\Subset}\} \\
&= \lambda m\,.\, \{(r,\, (I_{f_1,r} \cap m(r) \cap I_{f_2,r}) \cup (I_{f_1,r} \cap U_{f_2,r}) \cup U_{f_1,r}) \mid r \in R^{\Subset}\} \\
&= \lambda m\,.\, \{(r,\, (((m(r) \cap I_{f_2,r}) \cup U_{f_2,r}) \cap I_{f_1,r}) \cup U_{f_1,r}) \mid r \in R^{\Subset}\} \\
&= (\lambda m\,.\, \{(r,\, (m(r) \cap I_{f_1,r}) \cup U_{f_1,r}) \mid r \in R^{\Subset}\}) \\
&\quad \circ ((\lambda m\,.\, \{(r,\, (m(r) \cap I_{f_2,r}) \cup U_{f_2,r}) \mid r \in R^{\Subset}\}) \\
&= [\![\{(r,\, \langle I_{f_1,r},\, I_{f_2,r}\rangle)\}]\!] \circ [\![\{(r,\, \langle I_{f_2,r},\, U_{f_2,r}\rangle)\}]\!] \\
&= [\![f_1]\!] \circ [\![f_2]\!]\,. \qquad (4.1)
\end{aligned}$$

Finally, the meet operation on two micro functions is the union of their update maps:

$$\begin{aligned}
[\![f_1 \sqcap f_2]\!] &= [\![\{(r,\, \langle I_{f_1,r} \cup I_{f_2,r},\ U_{f_1,r} \cup U_{f_2,r}\rangle) \mid r \in R^{\Subset}\}]\!] \\
&= \lambda m\,.\, \{(r,\, m(r) \cap (I_{f_1,r} \cup I_{f_2,r}) \cup U_{f_1,r} \cup U_{f_2,r}) \mid r \in R^{\Subset}\} \\
&= \lambda m\,.\, \{(r,\, ((m(r) \cap I_{f_1,r}) \cup U_{f_1,r}) \cup ((m(r) \cap I_{f_2,r}) \cup U_{f_2,r})) \mid r \in R^{\Subset}\} \\
&= \lambda m\,.\, \{(r,\, (m(r) \cap I_{f_1,r}) \cup U_{f_1,r}) \mid r \in R^{\Subset}\} \\
&\quad \sqcap \lambda m\,.\, \{(r,\, (m(r) \cap I_{f_2,r}) \cup U_{f_2,r}) \mid r \in R^{\Subset}\} \\
&= [\![f_1]\!] \sqcap [\![f_2]\!]\,. \qquad (4.2)
\end{aligned}$$

We can now show how the correlated-calls definitions of the meet and composition operations on micro functions allow us to detect infeasible paths in a program.

Example 4.2.8. The edges of the exploded supergraph in Figure 3.2 correspond to the edges of an IFDS taint analysis. We can see that there is a path from the node ($\mathsf{start}_{\mathtt{main}}$, Λ) to (`print(s)` , s). This means that the IFDS taint analysis considers s to be a secret value that is leaked at the print statement.

The correlated-calls analysis, on the other hand, detects that the path to (`print(s)` , s) is infeasible: at the print node, the lattice element corresponding to the fact s contains a mapping $\mathtt{a} \to \top_T$.

The lattice element for the print statement is evaluated as follows:

$$\begin{aligned}
&((\lambda m\,.\, m[\mathtt{a} \to m(\mathtt{a}) \cap \{\mathtt{B}\}]) \circ \mathsf{id} \circ (\lambda m\,.\, m[\mathtt{a} \to m(\mathtt{a}) \cap \{\mathtt{A}\}]) \circ (\lambda m\,.\, \bot_{\Subset}) \circ \mathsf{id} \circ \ldots \circ \mathsf{id})\,(\top_{\Subset}) \\
=&(\bot_{\Subset}[\mathtt{a} \to m(\mathtt{a}) \cap (\{\mathtt{A}\} \cap \{\mathtt{B}\})]) \\
=&(\bot_{\Subset}[\mathtt{a} \to \top_T]).
\end{aligned}$$

[11] See Lemma 4.2.6.

Therefore, the path to the print statement will be considered infeasible, and the analysis does not claim that the program leaks a secret value.

4.2.5 Efficiency

In this section, we will show that our representation of micro functions is efficient according to the definition of efficiency discussed in Section 4.

Lemma 4.2.9. *The correlated-call representation of a micro function is efficient.*

Proof.

1. The identity function is represented as

$$[\![\mathsf{id}]\!] = \{(r, \langle \bot_T, \top_T \rangle) \mid r \in R^{\Subset}\};$$

   the top function is represented as

$$[\![\lambda m \,.\, \top_{\Subset}]\!] = \{(r, \langle \top_T, \top_T \rangle) \mid r \in R^{\Subset}\}.$$

2. Equations (4.1) and (4.2) show that the representation of micro functions is closed under composition and meet.

3. To show that our representation for micro functions forms a lattice with finite height, let us first show that $L^{\Subset}_{R^{\Subset}} : R^{\Subset} \to 2^T$ forms a lattice. Since T is a finite set, $(2^T, \subseteq)$ is a finite-height lattice. $R^{\Subset}$ is a finite set. Hence, the mapping

$$R^{\Subset} \mapsto 2^T = \{(r, t) \mid r \in R^{\Subset}, t \in 2^T\} = L^{\Subset}_{R^{\Subset}}$$

   also forms a finite-height lattice [16].

   Furthermore, $L^{\Subset}_{R^{\Subset}}$ is a finite set. Every element of $L^{\Subset}_{R^{\Subset}}$ can be applied to $|R^{\Subset}|$ receivers, where each receiver is mapped to a set of types. There are $|R^{\Subset}| \cdot 2^{|T|}$ different possibilities to form those mappings, so

$$|L^{\Subset}_{R^{\Subset}}| = |R^{\Subset}| \cdot 2^{|T|}.$$

   Therefore, $L^{\Subset}_{R^{\Subset}} \mapsto L^{\Subset}_{R^{\Subset}}$ also forms a finite-height lattice.

4. All operations can be computed in $O(R^{\Subset} \times T)$ time. Note that the $R^{\Subset}$ and T sets are an input to the correlated-calls analysis, and the time it takes to compute the meet or composition of micro functions is independent of the representation of the specific operand micro functions.

5. The space bound is $O(R^{\Subset} \times T)$.

□

Final Remarks. A straightforward solution to representing micro functions would be to use the function constructs that are provided by many programming languages. The efficiency requirement prohibits us from doing so. For most programming languages, equality for functions is either defined as reference equality (as in Scala), or is not defined at all (as in Haskell). Even if we were to define our own definition of equality for functions, we would have to iterate over the whole domain of the functions and compare the results of the function applications, which would be inefficient. Additionally, the equality check would be non-terminating if the domain of the functions were infinite, and undecidable if the language for defining the functions were Turing-complete.

Second, a composition f of two functions f_1 and f_2 would have to store both f_1 and f_2. For instance, if $f_1 = \lambda x \,.\, x + 1$ and $f_2 = \lambda x \,.\, x + 2$, then $f = f_2 \circ f_1$ would be represented as

$$f = \lambda x \,.\, (\lambda y \,.\, y + 2)((\lambda z \,.\, z + 1)\, x)$$

instead of

$$f = \lambda x \,.\, x + 3.$$

Having a compact representation for function composition is especially important for the first phase of the IDE algorithm, in the computation of jump functions [21]. The same argument applies to computing function meets.

4.2.6 Edge Function Representation

We will now show the representations for the correlated-call micro functions $\mathsf{EdgeFn}^{\Subset}_{R^{\Subset}}(e)$, described in Definition 3.2.7. Let $\mathsf{identity} = \langle \bot_T, \top_T \rangle$ represent the identity function id and $\mathsf{bottom} = \langle \bot_T, \bot_T \rangle$ represent the function $\lambda t \,.\, \bot_T$.

On the call-start edge,

$$m\,[r \to (m(r) \cap \tau(s_{\mathcal{F}},\, f))] = [\![\{(r,\, \langle \tau(s_{\mathcal{F}},\, f),\, \top_T \rangle)\} \cup \{(r',\, \mathsf{identity}) \,|\, r' \in R^{\Subset},\, r' \neq r\}]\!]\,.$$

On the end-return edge,

$$\begin{aligned}&\lambda m\,.\,m\,[v_1 \to \bot_T] \ldots [v_k \to \bot_T][r \to (m(r) \cap \tau(s_{\mathcal{F}},\, f))]\\ &\qquad = [\![\{(r,\, \langle \tau(s_{\mathcal{F}},\, f),\, \top_T \rangle)\} \cup \{(r',\, w(r')) \,|\, r \in R^{\Subset},\, r' \neq r\}]\!]\,,\end{aligned}$$

where

$$w(r) = \begin{cases} \mathsf{bottom} & \text{if } r \text{ is a local variable in the exiting method,} \\ \mathsf{identity} & \text{otherwise.} \end{cases}$$

For assignments in the source node of e,

$$\lambda m.m\,[r \to \bot_T] = [\![\{(r,\, \mathsf{bottom})\} \cup \{(r',\, \mathsf{identity}) \,|\, r' \in R^{\Subset},\, r' \neq r\}]\!]\,.$$

In the default case,

$$\mathsf{id} = [\![\{(r,\, \mathsf{identity}),\, r \in R^{\Subset}\}]\!]\,.$$

We have shown how IDE problems that account for correlated calls can be represented in an efficient way. In the next chapter, we address the implementation and present an evaluation of the correlated-calls analysis.

Chapter 5

Evaluation

This chapter discusses implementation aspects of the correlated-calls analysis and presents experimental results.

5.1 Implementation of the Analysis

The correlated-calls analysis was implemented in the Scala programming language [17]. We chose Java as the target language for client programs of the analysis. To retrieve information about an input program, such as its control-flow supergraph or the set of receivers and their types, we used the WALA framework for static analysis on Java bytecode [6].

Since WALA currently only contains an implementation of IFDS, we implemented IDE from scratch. Instead of using WALA's IFDS implementation, to run an IFDS problem, we converted it to an IDE problem and used our own IDE solver.

5.1.1 IFDS

As described in Section 3.1.1, an IFDS problem is defined in terms of an exploded supergraph. The control-flow supergraph of an input program can be retrieved using WALA. Hence, our implementation of an IFDS problem should be able to convert a control-flow supergraph into an exploded supergraph.

We represent an IFDS problem with a trait, or protocol, that contains declarations of four *flow functions*. Each function has type

$$F : (N \times D \times N) \to 2^D$$

and defines a set of edges on the exploded graph. Given an edge (n_1, n_2) of the control-flow supergraph and the fact d_1 that corresponds to the source node n_1, $F(n_1, d_1, n_2)$ returns the set of all facts $d_2 \in D_2$ such that $((n_1, d_1), (n_2, d_2)) \in E^{\#}$[12]. The four functions are:

- call-start, for inter-procedural edges from a call node to the start node of the target method;
- call-return, for intra-procedural edges from a call node to its return node;
- end-return, for inter-procedural edges from the end node of a method to the return node of the callee;
- default, for all other intra-procedural edges.

5.1.1.1 Taint Analysis

Using this representation of an IFDS problem, we implemented an IFDS problem instance for taint analysis. We used it as a sample IFDS problem on which to evaluate the correlated-calls-IDE construction.

Let N^* be the control-flow supergraph of a program and D the set of the program variables. Let $\mathsf{encl}(n)$ be a function that returns the enclosing method of a node $n \in N^*$. Finally, let the function $r_m : D \to 2^D$ be defined as follows:

$$r_m(d) = \begin{cases} \varnothing & \text{if } d \text{ is a local variable in method } m, \\ \{d\} & \text{otherwise.} \end{cases}$$

When defining the flow functions for a taint analysis, we will use r_m to avoid the propagation of local variables, as shown below.

For a fact $d_1 \in D \cup \{\mathbf{0}\}$ and two nodes $n_1, n_2 \in N^*$, the simplified[13] version of flow functions for a taint-analysis looks as follows.

If n_1 is a call node that calls method m, and n_2 is m's start node,

$$\mathsf{call\text{-}start}(n_1, d_1, n_2) = \begin{cases} r_{\mathsf{encl}(n_1)}(d_1) \cup \{v\} & \text{if } a \text{ is the } i\text{th argument of the call, } d_1 = a, \\ & \text{and } v \text{ is the } i\text{th parameter of } m; \\ r_{\mathsf{encl}(n_1)}(d_1) & \text{otherwise.} \end{cases}$$

[12] In each invocation of a flow function, the fact d_1 is provided by the IDE algorithm.

[13] For simplicity, the shown flow functions do not account for different Java-specific features such as arrays, fields, operations on strings, etc.

If n_1 is a call node with corresponding return node n_2,

$$\textsf{call-return}(n_1,\, d_1,\, n_2) = \begin{cases} \{d_1\} & \text{if } d_1 \text{ is a local variable in } \textsf{encl}(n_1), \\ \varnothing & \text{otherwise.} \end{cases}$$

If c is a call node calling method m, n_1 is m's end node, and n_2 is c's return node,

$$\textsf{end-return}(n_1,\, d_1,\, n_2) = \begin{cases} r_{\textsf{encl}(n_1)}(d_1) \cup \{x\} & \text{if } n_1 \text{ is a return statement returning } v \\ & n_2 \text{ is an assignment with left-hand side } x, \\ & \text{and } d_1 = v; \\ r_{\textsf{encl}(n_1)}(d_1) & \text{otherwise.} \end{cases}$$

Otherwise,

$$\textsf{default}(n_1,\, d_1,\, n_2) = \{d_1\}.$$

Example 5.1.1. Consider the supergraph in Figure 2.2. The call-to-start flow function from method `main` to `f` looks as follows:

$$\begin{aligned} \textsf{call-start}(\textsf{call}_{\texttt{A.f}},\, \texttt{a},\, \textsf{start}_{\texttt{f}}) &= r_{\texttt{main}}(\texttt{a}) \cup \{\texttt{s}\} \\ &= \{\texttt{s}\}. \end{aligned}$$

We can see that correspondingly, the exploded supergraph contains an edge from $(\textsf{call}_{\texttt{A.f}},\, \texttt{a})$ to $(\textsf{start}_{\texttt{f}},\, \texttt{s})$.

5.1.2 IDE

The correlated-calls analysis was implemented as an IDE problem instance.

We defined an IDE problem in the same way as an IFDS problem, except that the IDE flow functions are of type

$$(N \times D \times N) \to 2^{D \times (L \to L)}.$$

With the new flow functions, we can implement a labeled exploded supergraph, since the new flow functions return a set of facts that are paired with micro functions.

For example, if Q is an IDE problem, then the call-to-start flow function for Q is defined as follows:

$$\textsf{call-start}^Q(n_1,\, d_1,\, n_2) = \left\{(d_2,\, f) \,\middle|\, d_2 \in D,\, f \in L^Q \to L^Q :\, \textsf{EdgeFn}^Q((n_1,\, d_1),\, (n_2,\, d_2)) = f\right\}.$$

The other flow functions are defined analogously.

5.2 Testing

In this section we assess the correctness and effectiveness of the correlated-calls analysis.

5.2.1 Conversion from IFDS to IDE

We implemented the equivalence transformation $\mathcal{T}^{\equiv}$ and the correlated-calls transformation $\mathcal{T}^{\Subset}_{R^{\Subset}}$ from IFDS to IDE described in Section 3.2.1. To run an IFDS problem, we converted it to an IDE problem using $\mathcal{T}^{\equiv}$ and $\mathcal{T}^{\Subset}_{R^{\Subset}}$ and used our IDE analysis algorithm to run the latter.

Given an IFDS problem described with IFDS flow functions, an equivalence transformation creates an IDE problem described with the following IDE flow functions:

$$\begin{aligned}
\mathsf{call\text{-}start}^{\equiv}(n_1,\, d_1,\, n_2) &= \{(d_2,\, \epsilon(d_1,\, d_2)) \mid d_2 \in \mathsf{call\text{-}start}(n_1,\, d_1,\, n_2)\} \\
\mathsf{call\text{-}return}^{\equiv}(n_1,\, d_1,\, n_2) &= \{(d_2,\, \epsilon(d_1,\, d_2)) \mid d_2 \in \mathsf{call\text{-}return}(n_1,\, d_1,\, n_2)\} \\
\mathsf{end\text{-}return}^{\equiv}(n_1,\, d_1,\, n_2) &= \{(d_2,\, \epsilon(d_1,\, d_2)) \mid d_2 \in \mathsf{end\text{-}return}(n_1,\, d_1,\, n_2)\} \\
\mathsf{default}^{\equiv}(n_1,\, d_1,\, n_2) &= \{(d_2,\, \epsilon(d_1,\, d_2)) \mid d_2 \in \mathsf{default}(n_1,\, d_1,\, n_2)\},
\end{aligned}$$

where ϵ is the bottom function on an edge from a Λ-fact to a non-Λ-fact, and the identity function otherwise:

$$\epsilon(d_1,\, d_2) = \begin{cases} \lambda l\,.\,\bot & \text{if } d_1 = \Lambda \text{ and } d_2 \neq \Lambda; \\ \mathsf{id} & \text{otherwise.} \end{cases}$$

We also implemented a correlated-call transformation from IFDS into IDE problems that consider correlated calls. This transformation is described in Section 3.2.2. The flow functions can be easily inferred from Section 4.2.6.

5.2.2 Regression Testing

We used regression tests to assess the correctness of the implemented analyses. Each test involves running a certain analysis on one input Java program.

5.2.2.1 IDE-Implementation Correctness

To test the correctness of the IDE algorithm implementation, we implemented a copy-constant-propagation IDE problem [21]. In a copy-constant propagation analysis, a variable is considered constant if it is assigned a constant literal or another variable that is also a constant. For example, in a program

```
1   int a = 1;
2   int b = a;
3   int c = a + b;
4   int d = a + 2;
```

`a` and `b` are considered constant, but `c` and `d` are not (although `d` would be considered constant in linear-constant propagation).

We tested the propagation of constants on different intra- and inter-procedural data-flow paths, in parameter passing, and in conditional branches. Each regression test contained assertions of the form "at the end of method m, variable with name x should be (not) constant".

We also tested the implementation of the IDE algorithm on an IDE problem generated by conversion from an IFDS problem.

To do that, we implemented an IFDS instance for taint analysis.

Recall from Section 2.3 that taint analysis aims to discover variables that are secret at a given program point called a sink.

We used assertions of the form "at program statement n, variable x should be (not) secret" by defining the sink of a secret value through special `isSecret` and `notSecret` methods. Those methods asserted that the parameter passed to them is secret and not secret, respectively. To define a source secret value we created a static `secret()` method that returned a string.

Example 5.2.1. Listing 5.3 illustrates the use of the `isSecret` and `notSecret` assertions.

```
1   public static void main(String[] args) {
2     String n = "not secret";
3     notSecret(n);  // assert that n is not secret
4     String s1 = f(n);
5     isSecret(s1);  // in the next statement, f is invoked with a secret value
6                    // hence, the argument of f will always be considered secret
7                    // and f will always return a secret value
8     String s2 = f(secret());
9   }

10  static String f(String str) {
11    isSecret(str); // the function is once invoked with a secret value
12                   // hence, assert that str is secret
13    return str;
14  }

15  public static String secret() {  // the secret source
16    return "secret";
17  }
```

Listing 5.3: Example usage of `isSecret` and `notSecret` assertions in regression tests

We tested data flow through

- method calls and returns;
- conditional branches and loops, including nested constructions, the ternary operator, and `switch` statements;
- arrays and fields[14];
- static and instance class members;
- classes and interfaces that involve inheritance, overriding, and overloading;
- recursion;

[14] In Java, arrays are allocated on the heap, and array elements can be aliases of each other. Hence, if any array element gets assigned a secret value, we considered all elements of any `String` or `Object` array in the program secret. For the same reason, if a field `f` of an object of class `A` is assigned a secret value, then we considered the field `f` of any object of class `A` secret.

- library calls[15];
- string concatenation and usage of the `StringBuffer` and `StringBuilder` classes[16];
- generics, type conversions through castings, and exception handling.

Our taint analysis implementation becomes unsound in the presence of static initializers. If a static field is initialized to a secret value, our analysis will not detect it as such.

A static initializer is invoked only once, before the instance creation of a class or the access of a static member of that class. Static initializers are invoked lazily by the Java Virtual Machine [12]. This makes finding out at which program point a static initializer is invoked undecidable [8]. To account for static initializers in the analysis would require modifying WALA's control-flow supergraph (which does not have edges to static initializers) or using a data-flow analysis for static initialization. Since the primary purpose of the taint-analysis implementation was to test the correlated-call analysis, we did not include a static-initializer analysis in this work.

5.2.2.2 Correlated-Calls-Analysis Correctness

We tested the implementation of the correlated-calls analysis by converting the taint analysis into an IDE problem with an implementation of $\mathcal{T}^{\in}_{R^{\in}}$.

Since none of the test cases in the previous section contained correlated calls, we used the same tests with the same assertions to ensure that the correlated-calls analysis produces the same results as an IFDS-equivalent analysis in the absence of correlated calls.

We then added test cases that contained correlated calls. We added a new assertion method, `notSecretCC`. For the IFDS-equivalent analysis, the method asserted that the argument passed to it was secret, and for the correlated-calls analysis, it asserted that the argument was not secret.

Separately, we used unit tests to check the implementation correctness of micro functions. We wrote assertions for the results of the equality, meet, and composition operations on all possible combinations of the identity, top, bottom, and constant functions.

[15] We created a specification for library functions that allowed us to indicate under which conditions a library function returned a secret value. This let us avoid the expensive analysis of library functions.

[16] Using mutation, objects of these classes can be converted into wrappers around secret strings. This is why we added a special handling for `StringBuffer` and `StringBuilder` objects. For instance, if a field had the `StringBuilder` type, it was considered secret.

5.2.3 Benchmark Testing

To assess the benefit of the correlated-calls analysis, we counted the frequencies of correlated-call occurrences in the Dacapo benchmarks [2]. We then ran the normal- and correlated-call-taint analysis on the Dacapo benchmarks to see what improvement we would get from the correlated-calls analysis.

5.2.3.1 Occurrences of Correlated Calls

Our goal was to obtain an upper bound on the number of redundant IFDS-result nodes that could be potentially removed by our analysis. We counted the number of correlated calls that occurred in programs of the Dacapo benchmarks, as shown in Table 5.1.

In the table, the number of all call sites in a program is denoted as C. Polymorphic call sites are denoted as C_P, and correlated call sites as $C^{\Subset}$. The first four columns indicate the overall number of various call sites and correlated-call receivers in a program. The last three columns indicate the ratio of polymorphic to all call sites, the ratio of correlated to polymorphic call sites, and the ratio of correlated call sites to correlated-call receivers.

Benchmark	$\lvert C\rvert$	$\lvert C_P\rvert$	$\lvert C^{\Subset}\rvert$	$\lvert R^{\Subset}\rvert$	$\frac{\lvert C_P\rvert}{\lvert C\rvert}$	$\frac{\lvert C^{\Subset}\rvert}{\lvert C_P\rvert}$	$\frac{\lvert C^{\Subset}\rvert}{\lvert R^{\Subset}\rvert}$
antlr	7,610	428	299	70	**6%**	**70%**	4
bloat	18,157	933	429	119	**5%**	**46%**	4
chart	18,101	466	195	61	**3%**	**42%**	3
eclipse	3,222	100	35	10	**3%**	**35%**	4
fop	4,831	129	40	12	**3%**	**31%**	3
hsqldb	3,573	81	35	10	**2%**	**43%**	4
jython	12,149	487	129	54	**4%**	**26%**	2
luindex	7,190	188	79	29	**3%**	**42%**	3
lusearch	9,043	350	126	47	**4%**	**36%**	3
pmd	10,972	219	68	23	**2%**	**31%**	3
xalan	3,889	110	35	10	**3%**	**32%**	4
Geom. mean	**7,572**	**240**	**91**	**29**	**3%**	**38%**	**3**

Table 5.1: Frequencies of correlated-call occurrences in the Dacapo benchmarks

We can see that on average, 3% of all call sites C are polymorphic call sites C_P. Out of

those call sites, 38% are correlated call sites $C^{\Subset}$. We also see that for one correlated-call receiver, there are on average three correlated calls.

5.2.3.2 Experiments

We ran the analysis on the Dacapo benchmarks to test if the taint analysis would benefit from the improved, correlated-calls based, analysis. We defined any user input string to be considered a secret source and compared the overall number of results in the original and correlated-call taint analyses. If the number of secret values in the original result were larger than in the correlated-call result, we would see a practical benefit from our analysis.

However, even when we considered each program point as a sink, the "improved" analysis revealed the same number of secret values as the original taint analysis.

A correlated call that could affect a taint-analysis result could most likely occur in the following scenario:

- there is a receiver with at least two polymorphic calls;
- at least one of the calls c_1 returns a string — this would mean that the method potentially returns a secret value;
- at least one of the calls c_2 takes a string parameter — this would mean that a secret value could potentially be propagated to the method as an argument.

Then, if the correlated call occurred on an invocation $c_2(c_1())$, there might be a possibility of benefiting from the correlated-calls analysis. Given the relatively rare occurrence of correlated calls, this situation is not likely to appear often. This is illustrated in Table 5.2 which shows how often correlated calls would invoke methods that either take a string as a parameter *or* return a string. The set of receivers on which there are invocations of such methods is denoted as ${R^{\Subset}}_S$. A situation where one correlated call returned a string, *and* another correlated call on the same receiver took a string parameter, appeared in only one case in the `antlr` benchmark. However, the strings invoked were not designated as secret.

This explains why, specifically for a taint analysis as the client analysis, and specifically for the Dacapo benchmarks, the correlated call analysis did not make a difference.

Benchmark	$\lvert R^{\Subset}{}_S \rvert$	$\lvert R^{\Subset} \rvert$	$\frac{\lvert R^{\Subset}{}_S \rvert}{\lvert R^{\Subset} \rvert}$
antlr	43	70	62%
bloat	0	119	0%
chart	1	61	2%
eclipse	0	10	0%
fop	0	12	0%
hsqldb	0	10	0%
jython	6	54	23%
luindex	0	29	0%
lusearch	2	47	6%
pmd	1	23	3%
xalan	0	10	0%
Geom. mean	**3**	**29**	**9**

Table 5.2: Frequency of correlated-call receivers for which at least one of the correlated calls takes a string as a parameter or returns a string

5.3 Future Work

In this section we point out the limitations of the correlated-calls analysis and suggest improvements to the analysis for future work.

One limitation of the analysis is that it only works for IFDS problems like taint analysis, reachable definitions, or available expressions. The correlated-call analysis is not applicable to IDE problems like copy- or linear-constant propagation. Therefore, a possible direction for future work is to create a correlated-calls analysis that transforms an original IDE problem into one that considers correlated calls (with a modified lattice and edge function definition), and then transforms the correlated-calls result into a more precise result of the original IDE problem.

Another constraint of the algorithm is that it only accounts for intra-procedurally-correlated receivers, or receivers on which correlated calls occur within one method. For example, in Listing 5.4, `a` is a correlated-call receiver, since there are two polymorphic method invocations on `a`. However, the first one, `a.setString()`, is inside method `main`, and the second one, `a.printString()`, is inside method `propagate`. Therefore, we do not treat `a` as a correlated-call receiver, and the analysis would not improve the original IFDS result for this program.

```
1   class A {
2     String string;
3     public static void main(String[] args) {
4       A a = args == null ? new A() : new B();
5       a.setString();
6       propagate(a);
7     }
8     static void propagate(A a) {
9       a.printString();
10    }
11    void setString() {
12      string = secret();
13    }
14    void printString() {
15      System.out.println("not secret");
16    }
17  }
18  class B extends A {
19    void setString() {
20      string = "not secret";
21    }
22    void printString() {
23      System.out.println(a);
24    }
25  }
```

Listing 5.4: Inter-procedurally-correlated calls

Finally, correlated calls can occur on multiple receivers and other scenarios discussed in [24] that are not handled in this work.

Chapter 6

Conclusions

We presented a technique to improve the precision of solutions to IFDS problems in the presence of correlated calls. Correlated calls occur when there are multiple polymorphic method invocations on the same receiver. Such method calls cause a data-flow analysis to consider infeasible paths, which makes the data-flow analysis less precise.

Our method of eliminating infeasible paths caused by correlated calls works by transforming an existing IFDS problem into a specialized IDE problem. In this way, we are able to track the classes to which method invocations get dispatched. After solving the specialized IDE problem, we convert its result into an IFDS result that is potentially more precise than the solution to the original IFDS problem. The increase in precision can occur for programs that contain correlated calls. Specifically, if, on a certain data-flow path, there are two polymorphic method invocations on the same receiver that dispatch to incompatible classes, the IDE analysis will consider the path infeasible.

We proved that the correlated-calls analysis is sound and that it improves the precision of IFDS results.

Our Scala implementation of the correlated-calls analysis includes

- an implementation of the IDE analysis, which is based on the WALA static program analysis framework;
- a taint-analysis implementation as an IFDS problem instance;
- a transformer of IFDS problems to equivalent IDE problems, and a second transformer that accounts for correlated calls.

We tested the correlated-calls analysis on our taint analysis implementation by comparing the number of secret values that were leaked when using an IFDS taint analysis and a taint analysis that accounts for correlated calls. We used the Dacapo benchmarks as input programs. Although the benchmarks contained a number of correlated calls, we were not able to improve the precision of the taint analysis, because the correlated calls did not occur on paths of secret information leaks.

We are hopeful that other analyses can benefit from the extra information provided by the correlated-calls analysis, and plan to test this hypothesis in the future.

References

[1] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, (4):17–20, 1998.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006*, pages 169–190, 2006.

[3] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, June 14, 2012*, pages 3–8, 2012.

[4] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT - statically analyzing software product lines in minutes instead of years. In *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014*, pages 81–82, 2014.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, (4):451–490, 1991.

[6] Stephen Fink and Julian Dolby. WALA — the TJ Watson libraries for analysis. http://wala.sourceforge.net, 2012.

[7] Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm. Invariance of approximate semantics with respect to program transformations. In *GI — 11. Jahrestagung in*

Verbindung mit Third Conference of the European Co-operation in Informatics (ECI), 20.-23. Oktober 1981, Proceedings, pages 1–10, 1981.

[8] Laurent Hubert and David Pichardie. Soundly handling static fields: Issues, semantics and analysis. *Electr. Notes Theor. Comput. Sci.*, (5):15–30, 2009.

[9] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Compiler Construction, 4th International Conference on Compiler Construction, CC'92, October 5-7, 1992, Proceedings*, pages 125–140, 1992.

[10] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, (3):268–299, 1996.

[11] Jörg Kreiker, Thomas W. Reps, Noam Rinetzky, Mooly Sagiv, Reinhard Wilhelm, and Eran Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics — Essays in Memory of Harald Ganzinger*, pages 414–445, 2013.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. 1997.

[13] Markus Müller-Olm and Oliver Rüthing. On the complexity of constant propagation. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, April 2-6, 2001, Proceedings*, pages 190–205, 2001.

[14] Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008*, pages 347–366, 2008.

[15] Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *Compiler Construction, 19th International Conference, CC 2010, March 20-28, 2010. Proceedings*, pages 124–144, 2010.

[16] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. 2005.

[17] Martin Odersky. Essentials of Scala. In *Langages et Modèles à Objets, LMO 2009, 25-27 mars 2009*, page 2, 2009.

[18] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 23-25, 1995*, pages 49–61, 1995.

[19] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.

[20] Jonathan David Rodriguez. A concurrent IFDS dataflow analysis algorithm using actors. Master's thesis, 2010.

[21] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, May 22-26, 1995, Proceedings*, pages 651–665, 1995.

[22] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[23] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, June 11-16, 2012. Proceedings*, pages 435–458, 2012.

[24] Frank Tip. Infeasible paths in object-oriented programs. *Science of Computer Programming*. To appear, 2014.

[25] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, June 15-21, 2009*, pages 87–97, 2009.

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot — a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999*, page 13, 1999.