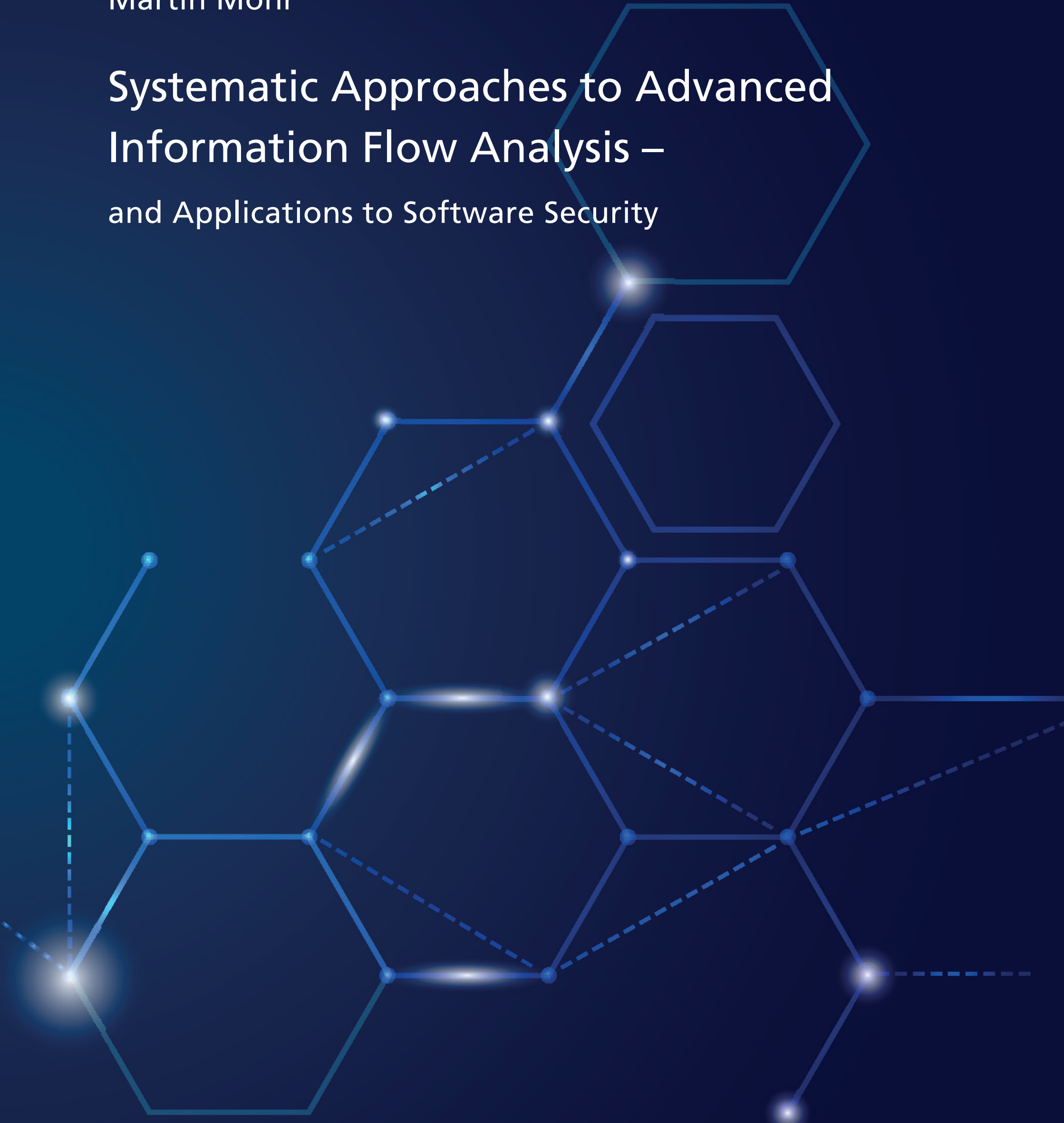
Martin Mohr

Systematic Approaches to Advanced Information Flow Analysis – and Applications to Software Security



KIT Scientific Publishing

Martin Mohr

Systematic Approaches to Advanced Information Flow Analysis – and Applications to Software Security

Systematic Approaches to Advanced Information Flow Analysis – and Applications to Software Security

by

Martin Mohr

KIT Scientific Publishing

Karlsruher Institut für Technologie
Institut für Programmstrukturen und Datenorganisation (IPD)

Systematic Approaches to Advanced Information Flow Analysis – and Applications to Software Security

Zur Erlangung des akademischen Grades eines Doktor-Ingenieurs von der KIT-Fakultät für Informatik des Karlsruher Instituts für Technologie (KIT) genehmigte Dissertation

von Martin Mohr

Tag der mündlichen Prüfung: 6. Mai 2022
Erster Gutachter: Prof. Dr.-Ing. Gregor Snelting
Zweiter Gutachter: Prof. Dr.-Ing. Christian Hammer

Impressum

KIT Scientific Publishing

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe

KIT Scientific Publishing is a registered trademark
of Karlsruhe Institute of Technology.
Reprint using the book cover is not allowed.

www.ksp.kit.edu

Print on Demand 2023 – Gedruckt auf FSC-zertifiziertem Papier

ISBN 978-3-7315-1275-2
DOI 10.5445/KSP/1000155035

Contents

With a little help from my friends...
THE BEATLES

Danksagung

Zunächst einmal möchte ich mich bei meinem Doktorvater und Erstgutachter Prof. Gregor Snelting dafür bedanken, dass er mir so viele Freiheiten bei meiner Forschung und der Ausarbeitung meiner Dissertation gelassen hat, aber auch für seine Unterstützung, sein Vertrauen und seine Geduld. Ich danke auch Prof. Christian Hammer für seine Bereitschaft, als Zweitgutachter dieser Arbeit tätig zu sein.
Ich danke der Deutschen Forschungsgemeinschaft, die diese Arbeit im Rahmen des Projektes „Zuverlässig sichere Software-Systeme" (SPP 1496, Sn11/12-1/2/3) teilweise finanziert hat. Ebenso möchte ich Prof. Heiko Mantel und den Mitarbeiterinnen und Mitarbeitern seines Lehrstuhls dafür danken, dass sie das Projekt so professionell organisiert und sich dafür eingesetzt haben, die Zusammenarbeit der verschiedenen, deutschlandweit verteilten Forschungsgruppen zu fördern. Ferner danke ich den Kolleginnen und Kollegen, mit denen ich im Rahmen des Projektes zusammenarbeiten durfte. Insbesondere danke ich Prof. Markus Müller-Olm und seinen Mitarbeitern Sebastian Kentner, Benedikt Nordhoff und Alexander Wenner für die gute Zusammenarbeit in unserem Teilprojekt „Information Flow Control for Mobile Components" (Sn11/12-1/2/3).
Danken möchte ich auch meinen ehemaligen Kollegen am Lehrstuhl von Prof. Snelting, namentlich meinen Programmanalytiker-Kollegen Jürgen Graf, Martin Hecker und Simon Bischof, sowie Joachim Breitner, Matthias Braun, Sebastian Buchwald, Andreas Fried, Sebastian Graf, Andreas Lochbihler, Denis Lohner, Manuel Mohr, Sebastian Ullrich, Maximilian Wagner und Andreas Zwinkau. Die insgesamt sieben Jahre, die ich mit ihnen zusammen arbeiten durfte, werden mir immer in guter Erinnerung bleiben. Ich erinnere mich gern an die vielen interessanten Fachdiskussionen, die

gute und professionelle Zusammenarbeit, aber auch an unsere sonstigen gemeinsamen Aktivitäten wie Film- oder Spieleabende, Kinogänge, Programmier-, Doktorhut- und Musikprojekte. Allen eben genannten bin ich außerdem dafür dankbar, dass sie mit mir ihr umfangreiches Fachwissen über Programmiersprachen, Compilerbau, Theorembeweiser und Programmanalyse geteilt haben, sowie für ihre Unvoreingenommenheit, ihre positive Problemlösermentalität, ihren Humor und ihre Offenheit. Ich danke meinen Kollegen insbesondere auch dafür, dass sie mir Feedback und Korrekturvorschläge zu Teilen meiner Dissertation gegeben haben. Ich danke auch Brigitte Sehan-Hill und Andreas Ladanyi, die durch ihre Arbeit im Hintergrund dafür gesorgt haben, dass wir wissenschaftlichen Mitarbeiter uns auf Forschung und Lehre konzentrieren konnten. Auch möchte ich an dieser Stelle Martin Armbruster, Tobias Blaschke, Stephan Gocht und Maik Wiesner dankend erwähnen, die uns im Rahmen von Abschlussarbeiten und Hilfstätigkeiten bei der Weiterentwicklung von Joana unterstützt haben.

Dankbar bin ich auch meinen Eltern Dorothea Mohr-Andrich und Harald Mohr. Sie haben mich immer gefördert und unterstützt, mir Vertrauen geschenkt, mich bestärkt und die von mir eingeschlagenen Wege nie in Frage gestellt. Ohne ihre liebevolle Unterstützung hätte ich wohl nie eine Promotion in Informatik angestrebt. Meinen Geschwistern Katrin Wünsch, Stephan Mohr und Anneliese Mohr danke ich für ihren Rückhalt und ihre bedingungslose Unterstützung.
Besonderer Dank gilt meiner langjährigen Partnerin und Ehefrau Claudia Lienau-Mohr für ihre Liebe, ihr Verständnis und dafür, dass sie mit mir alle Höhen und Tiefen überstanden und mich immer wieder aufgebaut hat. Ohne ihren Einsatz und die Opfer, die sie bringen musste, hätte ich diese Arbeit nicht beenden können. Auch Claudias Eltern möchte ich dafür danken, dass sie mir durch ihren Einsatz die Zeit für die Beendigung meiner Dissertation geschenkt haben und dabei nie an mir gezweifelt haben.

Ich danke Dorothea Jansen, die mich fachlich und moralisch unterstützt hat. Trotz gut gefüllten Alltags fand sie nicht nur die Zeit, Teile meiner Arbeit Korrektur zu lesen und mir wertvolles Feedback zu geben, sondern war und ist auch immer eine Ansprechperson und eine sehr gute Freundin für mich.

Weiterer Dank gilt Johannes Bechberger, Jan Betzing, Susanne Eckhardt, Melanie Heßmer, Stephan Mohr, Friederike Morfeld, Joachim Morfeld, Sebastian Schneider, Martin Wilde, Daniel Wünsch und Katrin Wünsch, die mir ebenfalls Feedback und Korrekturvorschläge gegeben haben.

Münster, Oktober 2021 *Martin Mohr*

With a little help from my friends. . .
The Beatles

Acknowledgments

First of all, I would like to thank my advisor Prof. Gregor Snelting for giving me so much freedom in my research and in the preparation of my thesis, but also for his support, trust, and patience. I also thank Prof. Christian Hammer for his willingness to contribute the second review of this thesis. I thank the German Research Foundation, who partially funded this work within the scope of the project "Reliably Secure Software Systems" (RS^3, SPP 1496). Particularly, I would like to thank Prof. Heiko Mantel and the staff of his chair for organizing the project in such a professional way and for their efforts to promote cooperation between the various research groups across Germany. I would also like to thank the colleagues with whom I have collaborated within RS^3. In particular, I thank Prof. Markus Müller-Olm and the members of his research staff Sebastian Kentner, Benedikt Nordhoff and Alexander Wenner for the good collaboration in our sub-project "Information Flow Control for Mobile Components" (Sn11/12-1/2/3).

I thank my former colleagues at the chair of Prof. Snelting, namely the other program analysts Jürgen Graf, Martin Hecker, and Simon Bischof, as well as Joachim Breitner, Matthias Braun, Sebastian Buchwald, Andreas Fried, Sebastian Graf, Andreas Lochbihler, Denis Lohner, Manuel Mohr, Sebastian Ullrich, Maximilian Wagner and Andreas Zwinkau. I will always have fond memories of the seven years that I worked with them – particularly of the many interesting discussions, the good and professional cooperation, but also of our other joint activities such as movie and game nights, programming and music projects, as well as the numerous "doctoral hat construction sprints". I am also grateful to them for sharing their extensive knowledge of programming languages, compiler construction, theorem provers and program analysis with me as well as for their impartiality,

positive problem-solving mentality, humor, and open-mindedness. I particularly thank those colleagues who where willing to provide feedback and suggest corrections to parts of my dissertation.
I also thank Brigitte Sehan-Hill and Andreas Ladanyi, whose organizational and technical work in the background made sure that the research staff members could concentrate on research and teaching, as well as Martin Armbruster, Tobias Blaschke, Stephan Gocht, and Maik Wiesner, who helped us in the context of their theses and other support activities in the further development of Joana.

I am grateful to my parents Dorothea Mohr-Andrich and Harald Mohr. They have always encouraged and supported me, gave me confidence, encouraged me and never questioned the paths I took. Without their loving support, I would probably never have pursued a doctorate in computer science. I thank my siblings Katrin Wünsch, Stephan Mohr and Anneliese Mohr for their backing and unconditional support.
Special thanks go to my longtime partner and wife Claudia Lienau-Mohr for her love, understanding and for the fact that she overcame all the ups and downs with me and built me up again and again. Without her commitment and the sacrifices she had to make, I would not have been able to finish this work. I would also like to thank Claudia's parents for giving me the time to finish my dissertation and for never doubting me.

I thank Dorothea Jansen, who supported me professionally and morally. Despite a fully packed everyday life, she not only found the time to proofread parts of my work and give me valuable feedback, but was and still is always a contact person and a very good friend for me.
Further thanks go to Johannes Bechberger, Jan Betzing, Susanne Eckhardt, Melanie Heßmer, Stephan Mohr, Friederike Morfeld, Joachim Morfeld, Sebastian Schneider, Martin Wilde, Daniel Wünsch and Katrin Wünsch, who provided me with feedback and suggestions for corrections.

Münster, October 2021 *Martin Mohr*

Zusammenfassung

Bei der *statischen Programmanalyse* geht es darum, Eigenschaften von Programmen abzuleiten, ohne sie auszuführen. Zwei wichtige statische Programmanalysetechniken sind zum einen die *Datenflussanalyse* auf *Kontrollflussgraphen* und zum anderen *Slicing* auf *Programmabhängigkeitsgraphen* (PAG). In dieser Arbeit berichte ich über Anwendungen von Slicing und Programmabhängigkeitsgraphen in der Softwaresicherheit. Außerdem schlage ich ein Analyse-Rahmenwerk vor, welches Datenflussanalyse auf Kontrollflussgraphen und Slicing auf Programmabhängigkeitsgraphen verallgemeinert. Mit einem solchen Rahmenwerk lassen sich neue PAG-basierte Analysen systematisch ableiten, die über Slicing hinausgehen.

Eine wichtige Anwendung von PAG-basiertem Slicing liegt in der Softwaresicherheit, genauer in der *Informationsflusskontrolle*. Bei dieser geht es darum sicherzustellen, dass ein gegebenes Programm keine vertraulichen Informationen über öffentliche Kanäle preisgibt bzw. öffentliche Eingaben keine kritischen Berechnungen beeinflussen können. Der Lehrstuhl Programmierparadigmen am KIT entwickelt seit einiger Zeit Joana, ein Werkzeug zur Informationsflusskontrolle für Java, das unter anderem auf Programmabhängigkeitsgraphen und Slicing basiert. Von 2011 bis 2017 war der Lehrstuhl am Schwerpunktprogramm 1496 „Zuverlässig sichere Softwaresysteme" (engl. Reliably Secure Software Systems, RS^3) der Deutschen Forschungsgemeinschaft (DFG) beteiligt.
Im ersten Teil dieser Arbeit gebe ich einen Überblick über Beiträge des Lehrstuhls zu RS^3, an denen ich beteiligt war. Diese Beiträge umfassen zum einen die Erweiterung eines mit PAG-basierten Techniken überprüfbaren Informationsflusskontrollkriteriums für nebenläufige Programme, und

zum anderen eine Reihe von Anwendungen von Joana in der Softwaresicherheit.

Im zweiten Teil meiner Doktorarbeit schlage ich vor, Datenflussanalysen auf Kontrollflussgraphen und Slicing auf Programmabhängigkeitsgraphen zu einer gemeinsamen, verallgemeinerten Analysetechnik zu vereinheitlichen. Eine solche Vereinheitlichung ermöglicht beispielsweise neue Analysen auf PAGs, die über bestehende PAG-basierte Ansätze hinausgehen. Darüber hinaus können für die Instanzen der allgemeinen Analysetechnik bestimmte formale Garantien gegeben werden, welche die Korrektheitsargumente, wie sie u.a. für bestehende PAG-basierte Analysen gegeben wurden, vereinfachen.

Zunächst stelle ich ein allgemeines Graphmodell sowie ein allgemeines Analyse-Rahmenwerk vor und zeige, dass sich sowohl Datenflussanalyse auf Kontrollflussgraphen als auch Slicing auf Programmabhängigkeitsgraphen darin ausdrücken lassen.

Anschließend zeige ich, dass sich Instanzen des allgemeinen Analyse-Rahmenwerkes durch Ungleichungssysteme beschreiben lassen. Hierbei greife ich auf klassische Ansätze zurück und passe diese geeignet an.

Ich gebe außerdem Algorithmen zur Lösung der zuvor aufgestellten Ungleichungssysteme an. Diese kombinieren klassische Lösungsalgorithmen für Datenflussanalysen mit einer Erreichbarkeitsanalyse.

Schließlich beschreibe ich eine Instanziierung der allgemeinen Analysetechnik für Programmabhängigkeitsgraphen. Ich stelle eine Implementierung in Joana vor und evaluiere diese anhand von realen Programmabhängigkeitsgraphen und einiger Beispielanalysen.

Die Hauptthesen meiner Arbeit lauten wie folgt:

1. PAG-basierte Informationsflusskontrolle ist nützlich, praktisch anwendbar und relevant.

2. Datenflussanalyse kann systematisch auf Programmabhängigkeitsgraphen angewendet werden.

3. Datenflussanalyse auf Programmabhängigkeitsgraphen ist praktisch durchführbar.

Abstract

Static program analysis is concerned with deriving properties of computer programs without executing them. Two important static program analysis techniques are *data-flow analysis on control-flow graphs* and *slicing on program dependence graphs (PDGs)*. In this thesis, I report on applications of slicing and program dependence graphs to software security. Moreover, I propose a framework that generalizes both data-flow analysis on control-flow graphs and slicing on program dependence graphs. Such a framework enables to systematically derive data-flow-like analyses on program dependence graphs that go beyond slicing.

One important application of PDG-based slicing lies in the field of software security, more specifically in *information flow control*. The goal of information flow control is to verify that a given program does not leak confidential information to public channels, or, respectively, that public input cannot influence critical computations.
The programming paradigms group at KIT develops Joana, a PDG-based information flow control tool for Java that employs program dependence graphs and slicing. From 2011 to 2017, our group participated in the priority program "Reliably Secure Software Systems" (RS^3, SPP 1496) of the German Research Foundation (DFG).
In the first part of this dissertation I give an overview of the contributions of the programming paradigms group to RS^3 in which I participated. These contributions include, on the one hand, the extension of an information flow control criterion for concurrent programs that can be checked with PDG-based techniques, and, on the other hand, a number of applications of Joana in software security.

In the second part of my dissertation, I present a unification of data flow analysis on control flow graphs and slicing on program dependence graphs into a common, general analysis technique. Such a unification enables new analyses on PDGs that go beyond existing PDG-based approaches. In addition, for instances of the general analysis technique, certain formal guarantees can be given for instances of the general analysis technique, which simplify correctness proofs such as those given for existing PDG-based analyses.

First, I introduce a general graph model as well as a general analysis framework and show that data-flow analysis on control-flow graphs as well as slicing on program dependence graphs can be expressed in this framework.

Then, I show that instances of the general analysis framework can be described by monotone constraint systems. For this, I resort to traditional approaches and adapt them appropriately.

I also present algorithms for solving the constraint systems set up earlier. These algorithms combine traditional solution approaches for data flow analyses with a reachability analysis.

Finally, I describe an instantiation of the general analysis technique for program dependence graphs: I present an implementation in Joana and evaluate it using real programs and a selection of example analyses.

In summary, the main theses of my dissertation are:

1. PDG-based information flow control is useful, practically applicable and relevant.

2. Data-flow analysis can be systematically applied to program dependence graphs.

3. Data-flow analysis on PDGs can be practically conducted.

Dear Sir or Madam, will you read my book?
It took me years to write, will you take a look?
The Beatles

1 Introduction

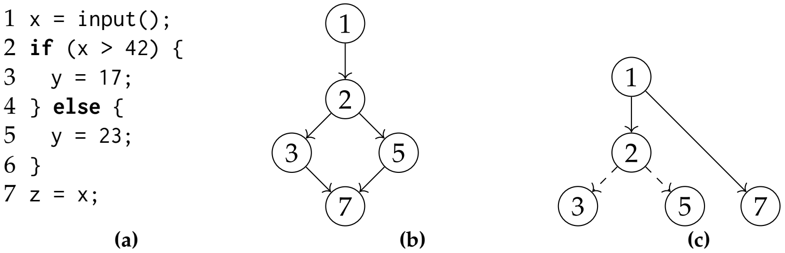
Program analysis is a branch of computer science that is concerned with the derivation of a given computer program's properties. *Static program analysis* [130, 76], or static analysis for short, is a sub-branch of program analysis that considers techniques to derive properties of a given program without executing it.

This thesis is located within the field of static program analysis. More specifically, it considers *automatic* static program analysis techniques, i.e. static analyses that can execute without human interaction.

Two notable static analysis techniques, which are important to this thesis, are *data-flow analysis* [101] and *program slicing* [165, 166, 58]. The basic idea of data-flow analysis is to analyze how a given program transforms data along its executions. Slicing was originally developed to aid programmers in debugging. Its goal is, given a program p, to extract a sub-program of p that behaves equivalently to p with respect to a given observation. Such a sub-program is also called a *slice*.

Both data-flow analysis and program slicing can be conducted on a graph representation of the given program. Data-flow analysis typically uses the program's *control-flow graph* [12], whereas an established approach to program slicing employs the *program dependence graph* (PDG) [58, 93, 137]. These two graph representations concentrate on different aspects of a program.

The control-flow graph focuses on the actual executions of the program and how control is transferred between its statements. Its nodes can be thought of as the program's statements, while an edge between a statement s_1 and another statement s_2 means that s_2 may be executed directly after s_1. A program dependence graph on the other hand materializes the *dependencies* between the program's variables and statements. There are



```
1 x = input();
2 if (x > 42) {
3   y = 17;
4 } else {
5   y = 23;
6 }
7 z = x;
```

(a) **(b)** **(c)**

Figure 1.1: A small code snippet **(a)** with its control-flow graph **(b)** and its program dependence graph **(c)** – node labels correspond to line numbers; in Figure 1.1c, data and control dependencies are represented by solid and dashed edges, respectively.

two major kinds of dependencies. A *data dependency* arises between two statements s_1 and s_2 if s_1 writes a value that s_2 reads. A *control dependency* describes that s_1 controls whether s_2 is executed or not. Figure 1.1 shows an example for control-flow graphs and program dependence graphs.

As has been shown by Ferrante et al. [58], PDGs can be used for program slicing: Giving a node n in a PDG G, called *slicing criterion*, the (PDG-based) backwards slice of n consists of all nodes from which n is reachable in G. Analogously, the forward slice of n consists of all nodes which are reachable from n.

Both control-flow graphs and program dependence graphs can also be used to represent programs with multiple procedures. These variants and the analyses that process them are called *interprocedural* [154, 93].

1.1 Applications to Software Security

The first main part of this thesis is concerned with applications of static analysis techniques such as PDGs and slicing to software security. The goal of static analysis in software security (which I will call *static security analysis* in the following) is usually to analyze a given program or system with respect to some desirable security property.

In software security, there are three desirable classes of properties of a given system. *Confidentiality* means that no sensitive information is disclosed

```
1 int h = inputPIN(); // HIGH
2 print(h); // LOW
3 if (h > 4711) {
4   print("OK"); // LOW
5 } else {
6   print("FAIL") // LOW
7 }
8 print("A") // LOW
```

Figure 1.2: Simple examples for illegal information flows

to a public or untrusted channel, *integrity* describes the property that no untrusted input can influence critical computations and, finally, *availability* states that data is always accessible if necessary [17]. This thesis focuses on confidentiality and integrity, which both can be generically formulated as an *information-flow property*. Basically, such a property demands that *high input cannot influence low output*. In the following, I briefly explain this in more detail. Information-flow properties assume that a program contains *(information) sources* and *(information) sinks*. Sources are typically points in the program where data is imported from the outside, whereas sinks are points in the program where it emits output, e.g. where it exports data to the outside. An information flow between a source and a sink manifests itself if a change in the data that a source imports can lead to a change in the output that a sink generates. A typical information-flow property demands that the given program does not contain any illegal information flows. In order to distinguish between legal and illegal information flows, sources and sinks are usually labeled with some form of sensitivity level, in the simplest case *high* and *low*. An information flow is called *illegal* if it starts in a high source and ends in a low sink.

Simple examples for legal and illegal information flows are shown in Figure 1.2. On the one hand, this code snippet contains illegal information flows from the high source in line 1 to the low sinks in lines 2, 4, and 6, respectively. On the other hand, no information flows from line 1 to line 8. The class of static security analyses that are concerned with verifying information-flow properties is also called *information flow control* [53].

As Snelting et al. [157] noted, slicing can be used to perform static information flow control. The basic idea is as follows: A (backwards) slice of a program with respect to a public sink contains all parts from which information may flow to the given sink. Therefore, if such a slice does not contain any secret source, there is no information flow between any secret source and the public sink.

Moreover, as mentioned before, program dependence graphs turn out to be an appropriate program representation to perform slicing. In particular, they reduce the task of computing a slice to a form of graph reachability. A slice can be obtained by traversing the graph and collecting all nodes that are connected to a given node via a chain of edges.

Extending upon these ideas, the programming paradigms group[1] at KIT developed JOANA [104], an information flow control tool for JAVA. First, given a JAVA application, JOANA builds a program dependence graph. Then, the user can annotate sources and sinks on this graph and use JOANA to perform various slicing-based static information flow control checks.

From 2011 to 2017, the programming paradigms group participated in the priority program "Reliably Secure Software Systems" (RS^3) of the German Research Foundation (DFG). The main thesis of RS^3 was that classical mechanism-based approaches to software security like authentication and access control need to be complemented with property-oriented approaches such as information flow control [4].

In chapter 4 of this thesis, I will give an overview of the achievements of the programming paradigms group within the RS^3 project. Our group extended the theoretical foundations of JOANA, participated in a number of collaborations, and applied JOANA to various scenarios within the field of software security.

1.2 Systematic Approaches to Advanced Information Flow Analysis

The second part of my thesis is concerned with a generalization of both data-flow analysis on control-flow graphs and slicing-based techniques on program dependence graphs. One motivation for this is that a generalized analysis framework enables data-flow-like analyses on program dependence graphs and therefore can extend the toolkit of PDG-based tools like JOANA by a family of powerful analyses. Moreover, such a generalization

[1] `https://pp.ipd.kit.edu`

is also theoretically interesting because it clarifies the relation between the two techniques, enables the re-use of formal guarantees and simplifies the development of new slicing-based techniques.

While they have different purposes and have developed independently, data-flow analysis on control-flow graphs and slicing on program dependence graphs share a fairly large amount of similarities. Both operate on a graph that represents the program to be analyzed and obtain their result by some form of propagation along an appropriate set of paths on the given graph. Moreover, both techniques face the same challenge for programs with multiple procedures. To analyze such programs properly, it is crucial to only consider paths where procedure calls return to the sites that they were actually called from. Data-flow analyses are usually conducted in order to derive properties of the set of a given program's executions by propagating pieces of data along abstract representations of these executions. Traditionally, they are expressed using a generic framework, for which general formal guarantees can be given [101, 154, 106]. Moreover, they can be systematically derived from program semantics [45]. In this sense, data-flow analysis can be thought of as executing the program using an abstraction of the real program semantics that concentrates only on those aspects of interest. Thus, data-flow analysis can represent and process fairly complex data.

Slicing on program dependence graphs is a form of reachability analysis. While it appears to be expressible as a very simple data-flow-like analysis – the information that slicing propagates is "this node is reachable"[2] – slicing can in fact *not* be cast as an instance of ordinary data-flow frameworks. This is because slicing requires a richer setting than such frameworks.

In the following, I briefly discuss the setting for data-flow analysis and then contrast it with the one for slicing.

Firstly, interprocedural data-flow analyses track the flow of data beginning in a *main* procedure from which every node is assumed to be reachable. Secondly, because data-flow analyses are supposed to consider abstractions of actual program executions, they naturally only follow *descending* control-flow paths that begin in entry of the *main* procedure. A control-flow path π is called *descending*, if π only contains returns from procedures for which π also contains the corresponding call.

[2] This will be detailed in chapter 3.

In summary, interprocedural data-flow analyses on control-flow graphs traverse descending paths that begin in the entry of the *main* procedure and thus reach every node in the graph.

In contrast, PDG-based slicing operates with different assumptions.

While data-flow analysis on control-flow graphs always starts propagation with a fixed node for which it is known per se that every node is reachable from it, PDG-based slicing starts from *an arbitrary node of interest*. Naturally, it cannot be assumed that this node is reachable in the PDG from the entry of *main*. Moreover, the node for which the slice is to be computed can be situated in some arbitrary part of the PDG. Hence, in order to compute a complete slice for this node, it is necessary to not only consider descending paths, but also paths that contain unmatched returns.

To sum up, the difference between the two techniques can be described best as follows: While data-flow analysis assumes that every node is reachable and computes a value for it, the task of PDG-based slicing is *to compute the very reachability information that data-flow analysis assumes.*

Hence, a generalization of both data-flow analysis on control-flow graphs and PDG-based slicing has to account for the two aspects that the two techniques lay their respective focus on: On the one hand, it needs to determine which nodes are reachable (like slicing does) and on the other hand, it needs to compute some result for every reachable node (like data-flow analysis does).

The second part of my thesis develops this idea. I propose a generalization of both data-flow analysis and slicing. Firstly, my generalization comprises a general graph model that covers both control-flow graphs and program dependence graphs. Secondly, it provides a framework for generalized data-flow analysis on this graph model. With this framework, both data-flow analyses and slicing-based analyses are expressible. My generalized data-flow framework combines the strengths of both classical data-flow analysis and PDG-based slicing. Like data-flow analysis, it allows to express data with complex structure. Like PDG-based slicing, it takes the node from which propagation is supposed to start as additional input and also follows paths with unmatched returns. I will show that classical approaches to interprocedural data-flow analysis are transferable to my generalized framework. Moreover, I present general algorithms that compute solutions to generalized data-flow analysis problems. These algorithms combine the classical solution algorithms for data-flow analyses [102, 130] with reachability analysis and a well-known approach to interprocedural slicing [93, 137].

1.3 Main Theses and Contributions

The main theses of this dissertation are:

Main Thesis 1: PDG-based information flow control is useful, practically applicable and relevant. I report on several applications of JOANA within the field of security analysis. These applications were the result of my collaborations with other research groups within the RS^3 project.

Main Thesis 2: Data-flow analysis can be systematically applied to program dependence graphs.

- I propose a general graph model and – building upon this model – a general data-flow analysis framework. In this framework, both classic data-flow analyses on interprocedural control-flow graphs and slicing-based analyses on interprocedural program dependence graphs can be expressed.
- I present two approaches with which instances of the proposed framework can be solved. These approaches are based on the approaches that were proposed by Sharir and Pnueli [154] for classic data-flow analysis problems and use monotone constraint systems to describe solutions.
- I describe generic solution algorithms for the presented approaches. These algorithms combine the classic worklist-based algorithm for solving monotone constraint systems and hence classic data-flow analysis problems with a reachability analysis.
- I give formal characterizations of the results of my algorithms and prove correctness results.
- I demonstrate that my solution algorithms can be refined in such a way that they can be reduced to the well-known algorithms proposed for context-sensitive slicing (for an appropriate framework instance).

Main Thesis 3: Data-flow analysis on PDGs can be practically conducted. Within the scope of this work, I have implemented my algorithms in JOANA and have evaluated them on real program dependence graphs. Thus, I demonstrate that the presented approaches not only enjoy pleasant theoretical properties but are also practically feasible and useful.

1.4 Organization of This Thesis

The organization of this thesis is depicted in Figure 1.3.

Chapter 2 compiles the basic notions and theories that are relevant throughout this thesis. In particular, it gives an introduction to basic order and fixed-point theory and, based upon that, a presentation of monotone constraint systems and the classical worklist-based algorithm to solve such systems. Monotone constraint systems form the theoretic foundation of data-flow analysis as it is presented and used in this thesis. The solving algorithm serves as the basis for algorithms that are developed in later chapters.

After the general foundations were laid in chapter 2, chapter 3 prepares for the two main topics of this thesis. Its first part, which ranges from 3.1 to 3.3, gives an introduction to several concepts and techniques employed in static program analysis. Specifically, it introduces both data-flow analysis on interprocedural control-flow graphs and context-sensitive slicing on interprocedural program dependence graphs. Moreover, it introduces information flow control and discusses its relation to slicing. Finally, section 3.4 prepares the reader for chapter 4. It gives an overview of JOANA, the information flow control tool developed by the programming paradigms group. In particular, it explains various data-flow analysis techniques that JOANA employs in order to properly compute program dependence graphs for object-oriented languages such as JAVA.

After chapter 3 has provided the necessary program analysis background, chapter 4 reports on the contributions of the programming paradigms group to RS^3. These contributions consist of (a) advancements of PDG-based checks for concurrent non-interference and (b) applications of JOANA to various scenarios and collaborations.

The chapters following chapter 4 lay out the second main topic of my thesis: The development of a general interprocedural framework that subsumes

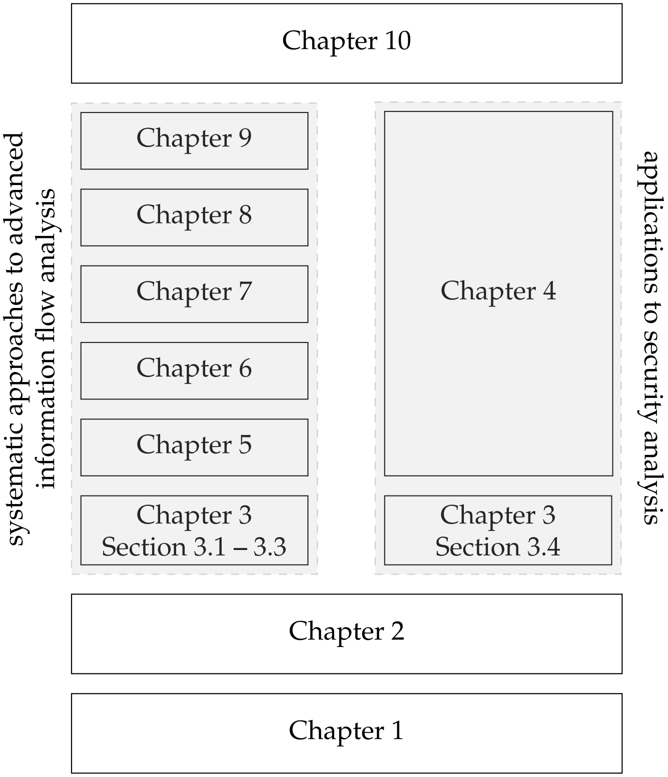


Figure 1.3: Visualization of the organization of this thesis – including the relation to its title

both data-flow analysis on control-flow graphs and context-sensitive slicing on program dependence graphs.

In chapter 5, which directly builds on the first part of chapter 3, I derive a graph model that subsumes both control-flow graphs and program dependence graphs and, based on this model, a general data-flow framework. In addition, I discuss a variety of example analyses that can be expressed within this framework.

Chapter 6 is concerned with the characterization of solutions to the problems posed by instances of the data-flow framework in chapter 5. To accomplish this task, I employ monotone constraint systems. I demonstrate that both the functional approach and the call-string approach can be applied to my general data-flow framework and that – under appropriate assumptions – both solve the problem with maximal precision. Because the unrestricted call-string approach in general does not allow a practical solution algorithm, I also consider a technique that enables practical algorithms and maintains correctness (while sacrificing precision). In particular, this technique generalizes the well-known technique that was proposed for interprocedural data-flow analysis on control-flow graphs.

In chapter 7, I describe algorithms for solving the constraint systems given in chapter 6. These algorithms combine a general worklist-based approach with a reachability analysis. For the functional approach, the algorithms that I obtain look like generalized versions of the well-known algorithms for context-sensitive slicing.

In chapter 8, I present an implementation of the algorithms derived in chapter 7, describe an evaluation of the implementation and discuss the evaluation results.

Chapter 9 gives a critical discussion of the work developed in the preceding chapters and relates it to the existing literature.

Finally, in chapter 10, I recapitulate the contents of this thesis and give an outlook on possible future work.

Usage of Personal Pronouns At this point, I want to briefly make clear the convention that I apply concerning the usage of personal pronouns in this dissertation.

Generally, it is similar to earlier dissertations [39]: I will mainly use the first person singular. In proofs, I use the plural form in order to invite the reader to conduct them with me. An exception is chapter 4, where I report about research that was conducted by multiple persons, including me. This is why I use the plural form “we” there. Generally, I choose to deviate from these rules whenever I think that it is necessary.

My foundations were made of clay.
Eric Clapton

2 Foundations

2.1 Sets and Relations

I assume in the following that the reader is familiar with basic set theory. This section is supposed to clarify the notations and conventions that I apply in this thesis.

For a given set A, I write $2^A \overset{def}{=} \{B \mid B \subseteq A\}$ for the *power set* of A.

For two sets A and B, $A \times B \overset{def}{=} \{(a,b) \mid a \in A \wedge b \in B\}$ is the *cartesian product* of A and B. Elements $(a,b) \in A \times B$ are also called *pairs*. For the purposes of this thesis, I consider cartesian products associative, that is I identify $A \times (B \times C)$ with $(A \times B) \times C$. Hence, the parentheses can generally be omitted.
The cartesian product can also be considered for $n \in \mathbb{N}$ sets $A_1, \dots, A_n$:

$$A_1 \times \cdots \times A_n = \prod_{i=1}^{n} A_i \overset{def}{=} \{(a_1, \dots, a_n) \mid \forall 1 \leq i \leq n.\ a_i \in A_i\}$$

Elements $(a_1, \dots, a_n)$ of $\prod_{i=1}^{n} A_i$ are called *n-tuples*. A special case is $n = 0$: $\prod_{i=1}^{n} A_i$ is defined to be the set that consists only of the *empty tuple* $\{()\}$.
In the following, I want to consider the special case of relations between two sets that I also call *binary relations*.
A binary relation $R \subseteq A \times B$ is called

1. *left-unique* if

$$\forall x \in A. \forall x' \in A. \forall y \in B. (x,y) \in R \wedge (x',y) \in R \implies x = x'$$

2. *right-unique* if

$$\forall x \in A. \forall y \in B. \forall y' \in B. (x, y) \in R \land (x, y') \in R \implies y = y'$$

3. *left-total* if

$$\forall x \in A. \exists y \in B. (x, y) \in R$$

4. *right-total* if

$$\forall y \in B. \exists x \in A. (x, y) \in R$$

The *domain* of a binary relation R is

$$dom(R) \overset{def}{=} \{a \in A \mid \exists b \in B.\ (a, b) \in R\}$$

The *image* of a binary relation R is

$$im(R) \overset{def}{=} \{b \in B \mid \exists a \in A.\ (a, b) \in R\}$$

If R is right-unique and $a \in dom(R)$, then I write $R(a)$ for the one and only $b \in B$ such that $(a, b) \in R$. Analogously, if R is left-unique and $b \in dom(R)$, then I write $R^{-1}(b)$ for the one and only $a \in A$ such that $(a, b) \in R$.
If $R \subseteq A \times B$ is right-unique, then R is also called *partial function from A to B*. For the set of partial functions from A to B, I use the notation $A \to_p B$. For $f \in A \to_p B$, I also write $f : A \to_p B$. If R is additionally left-total, then R is also called *function from A to B*. By $A \to B$, I mean the set of functions from A to B and for $f \in A \to B$, I also write $f : A \to B$.
A function R is called

1. injective if R is left-unique,
2. surjective if R is right-total, and
3. bijective if R is both injective and surjective.

Every binary relation $R \subseteq A \times B$ can be assigned a function $f_R : A \to 2^B$, defined by

$$f_R(a) \overset{def}{=} \{b \in B \mid (a, b) \in R\}.$$

Occasionally, I will use R and f_R interchangeably, that is I will consider relations $R \subseteq A \times B$ as functions $A \to 2^B$.

2.2 Complete Lattices, Fixed Point Theory Theory and Monotone Constraint Systems

In this section, I recall the basic notions that data-flow analysis builds upon and show generic algorithms that can be used to perform data-flow analyses. Particularly, I define monotone constraint systems and show how to solve them.
In subsection 2.2.1, I recall and clarify the basic notions and compile important results from the literature. In subsection 2.2.2, I introduce monotone constraint systems and characterize their solutions. In subsection 2.2.3, I present algorithms for solving monotone constraint systems.

2.2.1 Partial Orders and Fixed-Points

Partial Orders. A *partial order* is a tuple $(L, \leq)$ which consists of a set L and a relation $\leq \subseteq L \times L$ with the following properties:

(reflexivity) $\forall x \in L.\, x \leq x$

(anti-symmetry) $\forall x, y \in L.\, x \leq y \wedge y \leq x \implies x = y$

(transitivity) $\forall x, y, z \in L.\, x \leq y \wedge y \leq y \implies x \leq z$

Let $(L, \leq)$ be a partial order and $A \subseteq L$. Then x is called *minimal in A* if $\forall y \in A.x \not\geq y$. Moreover, $x \in A$ is called *least element of A* if $\forall y \in A.\, x \leq y$. Note that least elements do not need to exist but are unique if they do, while minimal elements neither need to exist nor need to be unique. However, if A has a least element, then this element is necessarily minimal. Plus, if A is finite and non-empty, it always has minimal elements.

Bounds. An element $u \in L$ is called *upper bound* of $A \subseteq L$ if

$$\forall a \in A.\, a \leq u.$$

Upper(A) denotes the set of upper bounds of A. $u \in L$ is called *least upper bound* of A if u is an upper bound of A and if

$$\forall u' \in Upper(A).\, u \leq u'.$$

I also write $\bigsqcup A$ for the least upper bound of A. $l \in L$ is called *lower bound* of A if

$$\forall a \in A.\, l \leq a.$$

$Lower(A)$, or $\bigsqcap A$, respectively, denotes the set of lower bounds of A. $l \in L$ is called *greatest lower bound* of A if l is a lower bound of A and if

$$\forall l' \in Lower(A).\, l' \leq l.$$

In the following, I compile some basic facts about least upper bounds and greatest lower bounds. Since they do not need to exist, I apply the usual convention for equations about partially defined objects: If I state an equation of the form $x = y$, I mean that either both x and y exist and coincide or that neither exists.
Least upper bounds and greatest lower bounds are dual to each other in the following sense:

$$\forall A \subseteq L.\, \bigsqcup A = \bigsqcap Upper(A) \tag{2.1}$$

$$\forall A \subseteq L.\, \bigsqcap A = \bigsqcup Lower(A) \tag{2.2}$$

Assume that $A = \{a, b\}$ and that $\bigsqcup A$ exists. Then we define

$$a_1 \sqcup a_2 \overset{def}{=} \bigsqcup \{a_1, a_2\} \tag{2.3}$$

$$a_1 \sqcap a_2 \overset{def}{=} \bigsqcap \{a_1, a_2\} \tag{2.4}$$

This defines partial binary operations $\sqcup, \sqcap : L \times L \rightarrow_p L$. Easy calculations show that both $\sqcup$ and $\sqcap$ are associative and commutative, that is

$$\forall a, b \in L.\, a \sqcup b = b \sqcup a \wedge a \sqcap b = b \sqcap a \tag{2.5}$$

$$\forall a, b, c \in L.\, a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \wedge a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c. \tag{2.6}$$

These properties of $\sqcup$ and $\sqcap$ justify to extend both operations to arbitrary finite sets:

$$a_1 \sqcup \cdots \sqcup a_n \overset{def}{=} \bigsqcup \{a_1, \ldots, a_n\} \tag{2.7}$$

$$a_1 \sqcap \cdots \sqcap a_n \overset{def}{=} \bigsqcap \{a_1, \ldots, a_n\} \tag{2.8}$$

Monotone Functions. For two partial orders $(L, \leq)$ and $(L, \leq')$, I call a function $f : L \to L'$ *monotone*, if

$$\forall l_1, l_2 \in L.\, l_1 \leq l_2 \implies f(l_1) \leq' f(l_2) \tag{2.9}$$

Given a partial order $(L, \leq)$ and a monotone function $f : L \to L$, we can consider elements of L that behave specially with respect to f. I call $x \in L$

1. *reductive* if $f(x) \leq x$
2. *extensive* if $x \leq f(x)$
3. *fixed-point* if x is both reductive and extensive.

The sets of reductive elements, extensive elements and fixed-points are denoted by $Ext(f)$, $Red(f)$ and $Fix(f)$, respectively.
If $Fix(f)$ has a least element, i.e. if there is $x \in Fix(f)$ with the property $\forall y \in Fix(f).\, x \leq y$, then I call x *least fixed-point of f* and write it as $lfp(f)$.

Complete Lattices. A *complete lattice* is a partial order $(L, \leq)$ in which every subset has a least upper bound. In particular, it has a least element $\bot \overset{def}{=} \bigsqcup \emptyset = \bigsqcap L$ and a greatest element $\top \overset{def}{=} \bigsqcup L = \bigsqcap \emptyset$.

Theorem 2.1 (Knaster-Tarski, cf.[161, Theorem 1])**.** *Let $f : L \to L$ be a monotone function on a complete lattice $(L, \leq)$. Then $Fix(f)$ is not empty and $(Fix(f), \leq)$ is a complete lattice. In particular, we have*

$$\bigsqcup Fix(f) = \bigsqcup Ext(f)$$

and

$$\bigsqcap Fix(f) = \bigsqcap Red(f)$$

Theorem 2.1 reduces the problem of finding the least reductive point to the problem of finding the least fixed point. Unfortunately, it is not constructive, in the sense that it gives no recipe of how to find the least fixed point. However, with some modifications, a constructive result can be given in the form of such a recipe. Before I present this result, I introduce some auxiliary definitions.

Chains and Chain conditions. A *chain* in $(L, \leq)$ is a subset $C \subseteq L$ such that

$$\forall x, y \in C.\, x \leq y \vee y \leq x.$$

With $Chains(L)$ I denote the set of chains in L. A sequence $(l_i)_{i \in \mathbb{N}}$ is called *ascending chain* if

$$\forall i, j \in \mathbb{N}.\, i \leq j \implies l_i \leq l_j$$

I denote the ascending chains of L with $Asc(L)$. A sequence $(l_i)_{i \in \mathbb{N}}$ is called *descending chain* if

$$\forall i, j \in \mathbb{N}.\, i \leq j \implies l_i \geq l_j$$

I denote the descending chains of L with $Desc(L)$.
It is important to consider the different natures of chains on the one side and ascending and descending chains on the other side. Any ascending or descending chain can be assigned a chain as follows: Every sequence can be considered a function $l : \mathbb{N} \to L$. Hence, we can consider the image $im(l)$ of l and observe that $im(l)$ is a chain if l is an ascending or descending chain.
Conversely, for a given chain C, any monotone function

$$l : (\mathbb{N}, \leq_{\mathbb{N}}) \to L$$

with $C = im(L)$ can justify to regard C as ascending chain and any monotone function $l : (\mathbb{N}, \geq_{\mathbb{N}}) \to L$ with $C = im(L)$ can justify to regard C as descending chain.
However, there are chains which cannot be considered as ascending or descending chains. Take for example the set $\mathbb{Z}$ of integers with its natural ordering and consider the set $2\mathbb{Z}$ of even integers. It is easy to see that $2\mathbb{Z}$ forms a chain in $\mathbb{Z}$ but there is no monotone function $l : \mathbb{N} \to \mathbb{Z}$ with $im(l) = 2\mathbb{Z}$ (neither for $\leq_{\mathbb{N}}$ nor for $\geq_{\mathbb{N}}$).
A partial order satisfies the *ascending chain condition*, if every ascending chain eventually stabilizes:

$$\text{(ACC)} \qquad \forall (l_i)_{i \in \mathbb{N}} \in Asc(L) \exists n_0 \in \mathbb{N}.\, \forall n \geq n_0.\, l_n = l_{n_0}$$

A partial order satisfies the *descending chain condition*, if every descending chain eventually stabilizes:

$$\text{(DCC)} \qquad \forall (l_i)_{i \in \mathbb{N}} \in Desc(L) \exists n_0 \in \mathbb{N}.\, \forall n \geq n_0.\, l_n = l_{n_0}$$

If C is a finite chain, then the cardinality $|C|$ is also called the *height of* C. L is said to have *finite height* if there is $n \in \mathbb{N}$ such that every chain has a height of at most n. The smallest such n (if it exists) is also called the *height of* L and is written as $height(L)$.
L is said to have *no infinite chains*, if all chains $C \subseteq L$ are finite.
There is a subtle difference between partial orders of finite height and partial orders with no infinite chains. Figure 2.1b illustrates the difference.

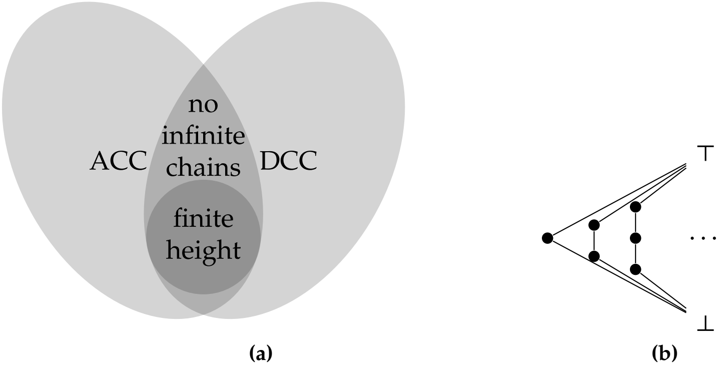


Figure 2.1: **(a)**: Relationship between different chain conditions – **(b)**: proof sketch for why the inclusion between “finite height” and “no infinite chains” is proper

As I elaborated on above, ascending and descending chains are special cases of chains. However, with regard to the respective chain conditions, there is a strong connection.

Theorem 2.2 ([50, p. 2.40]). *A partial order has no infinite chains if and only if it satisfies both the ascending and the descending chain condition.*

Chain-Complete Partial Orders. A *chain-complete partial order* (CCPO) is a partial order in which every chain in L has a least upper bound. In particular, a CCPO has a bottom element $\bot = \bigsqcup \emptyset$, since $\emptyset$ is a chain.
Let $(L, \leq)$ and $(L', \leq')$ be two CCPOs. Then $f : L \to L'$ is called *continuous*, if f is monotone and for every chain C in L we have $f(\bigsqcup C) = \bigsqcup f(C)$.

Theorem 2.3 (Fixed-point theorem of Kleene). *If $(L, \leq)$ is a CCPO and $f : L \to L$ is continuous, then f has a least fixed-point that is characterized as follows:*

$$lfp(f) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$$

Proof. See, e.g., [28, Theorem 6.3.2]. □

Theorem 2.3 not only gives a constructive characterization of the least fixed-point, it also enables a powerful proof technique that I will make use of later. This proof technique is formalized in the following lemma (cf.[28, Theorem 6.3.5]).

Lemma 2.4 (fixed-point induction principle). *Consider a continuous function $f : L \to L$ on a CCPO L. Furthermore, let $A \subseteq L$ be a subset of L which has the following closure properties:*

1. $\bot \in A$

2. $\forall B \in Chains(L).\, B \subseteq A \implies \bigsqcup B \in A$

3. $\forall a \in L.\, a \in A \implies f(a) \in A$

Then $lfp(f) \in A$.

Proof. Due to the assumptions about L and f, we can apply Theorem 2.3 and obtain

$$lfp(f) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$$

With the help of properties 1 and 3 we can show by complete induction on $i \in \mathbb{N}$:

$$\forall i \in \mathbb{N}.\, f^i(\bot) \in A$$

Finally, by property 2, we get

$$lfp(f) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot) \in A$$

□

Every complete lattice is a CCPO. However, the fixed-point theorem of Kleene is not applicable under the assumptions of Theorem 2.1. For general complete lattices, not all monotone functions are continuous. However, if we restrict the lattice, monotonicity and continuity coincide and Kleene's fixed-point theorem becomes applicable.

Lemma 2.5. *If* $(L, \leq)$ *is a CCPO which satisfies* (ACC)*, then every monotone function* $f : L \to L$ *is continuous.*

Proof. Let $f : L \to L$ be monotone and assume that $C \subseteq L$ is a chain. We must show that

$$f(\bigsqcup C) = \bigsqcup f(C)$$

Due to the monotonicity of f, it is clear that $\bigsqcup f(C) \leq f(\bigsqcup C)$, so it suffices to show

$$f(\bigsqcup C) \leq \bigsqcup f(C)$$

This is easy to see if C is finite, so we assume that C is infinite. First we observe that $\bigsqcup C \in C$, otherwise we could construct an ascending sequence $(c_i)_{i \in \mathbb{N}}$ with $\forall i \in \mathbb{N}.\ c_i \in C$ and $\forall i, j \in \mathbb{N}.\ i < j \implies c_i < c_j$ in contradiction to L satisfying (ACC).
But $\bigsqcup C \in C$ implies $f(\bigsqcup C) \in f(C)$ and this implies $f(\bigsqcup C) \leq \bigsqcup f(C)$. □

Corollary 2.6. *If* $(L, \leq)$ *is a CCPO which satisfies* (ACC) *and* $f : L \to L$ *is monotone, then the least fixed-point of* f *is characterized as follows:*

$$lfp(f) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot) \tag{2.10}$$

In particular, there is an $n \in \mathbb{N}$ *such that* $lfp(f)$ *coincides with* $f^n(\bot)$*:*

$$lfp(f) = f^n(\bot) \tag{2.11}$$

Proof. By Lemma 2.5, any monotone function on L is continuous. Hence, the first claim follows by Theorem 2.3.
Next, we show (2.11). For this, consider the set

$$K_f \overset{def}{=} \{f^i(\bot) \mid i \in \mathbb{N}\}$$

If K_f is finite, we choose n as the greatest number such that $f^n(\bot) \in K_f$. This is always possible because K_f is not empty. For this choice of n, (2.10) implies (2.11).
Now consider the case that K_f infinite. We take a look at the observation made in the proof of Lemma 2.5: It said that $\bigsqcup C \in C$ for every chain $C \subseteq L$, provided that L satisfies (ACC). Plus, an easy inductive argument shows that K_f is indeed a chain. Hence, we may conclude that

$$\bigsqcup K_f \in K_f.$$

In other words, there must be $n \in \mathbb{N}$ such that

$$\bigsqcup_{i \in \mathbb{N}} f^i(\bot) = f^n(\bot),$$

as desired. □

Corollary 2.7. *Let L be a CCPO which satisfies* (ACC) *and $f : L \to L$ be a monotone function on L. Furthermore, let $A \subseteq L$ be a subset of L which has the following closure properties:*

1. $\bot \in A$
2. $\forall B \in Chains(L).\, B \subseteq A \implies \bigsqcup B \in A$
3. $\forall a \in L.\, a \in A \implies f(a) \in A$

Then $lfp(f) \in A$.

Proof. This follows from Lemma 2.5 and Lemma 2.4. □

The ascending chain condition also comes in handy when we want to show that a partial order is a complete lattice. With (ACC) it is enough to show the existence of a bottom element and that every finite subset has a least upper bound. This is formalized by Lemma 2.8.

Lemma 2.8. *Let $(L, \leq)$ be a partial order that satisfies* (ACC)*. Then the following statements are equivalent:*

1. *L is a complete lattice.*
2. *L has a least element $\bot$ and for every $x, y \in L$, $x \sqcup y$ exists.*

Proof. This follows from [50, p. 2.41]. □

Some Constructions I want to conclude this section with the compilation of some examples of complete lattices, which will play a role later in this thesis.
For any set A, the power set 2^A with respect to set inclusion forms a complete lattice $(2^A, \subseteq)$, called the *power set lattice of* A. If A is finite, then the power set lattice of A has finite height. If A is infinite, then neither (ACC) nor (DCC) are satisfied.
If $(L_1, \leq_1)$ and $(L_2, \leq_2)$ are two partial orders, then the cartesian product $(L_1 \times L_2, \leq)$ forms a partial order where $\leq$ is the relation

$$(x, y) \leq (x', y') \overset{def}{\Longleftrightarrow} x \leq_1 x' \wedge y \leq_2 y'$$

Moreover,

- $(L_1 \times L_2, \leq)$ is a complete lattice if both L_1 and L_2 are.
- $(L_1 \times L_2, \leq)$ satisfies (ACC) if and only if both L_1 and L_2 do.

This can be generalized to an arbitrary finite number of complete lattices.
If $(L, \leq)$ is a partial order and A is any set, then the set of total functions $A \rightarrow L$ is a partial order with

$$f \leq g \overset{def}{\Longleftrightarrow} \forall x \in A.\ f(x) \leq g(x)$$

If L is a complete lattice, then $A \rightarrow L$ is, too. Moreover, if L satisfies (ACC) and A is finite, then $A \rightarrow L$ satisfies (ACC). Conversely, if $A \rightarrow L$ satisfies (ACC), then either L contains only one element or A is finite and L satisfies (ACC).
If $(L_1, \leq_1)$ and $(L_2, \leq_2)$ are partial orders, then the space $L_1 \rightarrow_{mon} L_2$ of monotone functions is a partial order via the relation defined above. Plus, if L_2 is a complete lattice, then $L_1 \rightarrow_{mon} L_2$ is also a complete lattice. That is, for any set $A \subseteq L_1 \rightarrow_{mon} L_2$ of monotone functions, the least upper bound defined by

$$(\bigsqcup A)(x) = \bigsqcup \{f(x) | f \in A\}$$

is also monotone.

2.2.2 Monotone Constraint Systems

In this section, I introduce *monotone constraint systems*. These systems are fundamental for program analysis and in particular for data-flow analysis, since they are expressive enough to describe abstractly how a program propagates values.

2.2.2.1 Syntax

I start with a set X of variables and a set $\mathcal{F}$ of function symbols. Every function symbol has an *arity* $a(f) \in \mathbb{N}$. Particularly, I allow function symbols with $a(f) = 0$, which I also call *constants*.
The set $Expr(X, \mathcal{F})$ of expressions over X and $\mathcal{F}$ is defined inductively as follows:

1. $X \subseteq Expr(X, \mathcal{F})$

2. If $f \in \mathcal{F}$ with $a(f) = n$ and $t_1, \ldots, t_n \in Expr(X, \mathcal{F})$, then $f(t_1, \ldots, t_n) \in Expr(X, \mathcal{F})$.

A *monotone constraint* (or constraint for short) has the form $x \geq t$ where x is a variable and $t \in Expr(X, \mathcal{F})$ is some expression over X and $\mathcal{F}$.
A *(monotone) constraint system* is a set C of constraints.
With $FV(t)$ I denote the variables occurring in $t \in Expr(X, \mathcal{F})$. I refer to $FV(t)$ as the set of *free variables in t*.
Let C be a monotone constraint system and $c = x \geq t \in C$. Then I define $lhs(c) \overset{def}{=} x$ and $rhs(c) \overset{def}{=} t$ and refer to $lhs(c)$ and $rhs(c)$ as the left-hand side and right-hand side of c, respectively.
With $Vars(C)$, I denote the set of left-hand sides of constraints in C, i.e.

$$Vars(C) = \{lhs(c) \mid c \in C\} \tag{2.12}$$

For a constraint $c \in C$ with $x = lhs(c)$, I also say that c *defines* x and refer to c as *defining constraint* for x.

Definition 2.9. *Let C be a constraint system.*

1. I define $\rightsquigarrow \subseteq C \times C$ by $x \geq t \rightsquigarrow x' \geq t'$ if $x \in FV(t')$. With $\rightsquigarrow^\star$ I denote the reflexive-transitive closure of $\rightsquigarrow$ and with $\rightsquigarrow^+$ the transitive closure.

2. *For sets $C_1, C_2 \subseteq C$ of constraints I write $C_1 \rightsquigarrow C_2$ if*

$$\exists c_1 \in C_1.\, \exists c_2 \in C_2.\, c_1 \rightsquigarrow c_2$$

$C_1 \rightsquigarrow^{\star} C_2$ and $C_1 \rightsquigarrow^{+} C_2$ have the respective meaning. Instead of $\{c\} \rightsquigarrow C_2$, I also write $c \rightsquigarrow C_2$ (analogously for $C_2 = \{c\}$). For $C_0 \subseteq C$ I denote with $C_0^{\star}$ the set $\{c \in C : C_0 \rightsquigarrow^{\star} c\}$ (C_0^{+} is defined analogously).

3. *For $x \in X$ I denote with $Def(x)$ the set of constraints with x on the left-hand side.*

4. *I overload $\rightsquigarrow$ to $\rightsquigarrow \subseteq X \times X$ as follows*

$$x \rightsquigarrow x' \iff Def(x) \rightsquigarrow Def(x').$$

$\rightsquigarrow^{\star} \subseteq X \times X$ and $\rightsquigarrow^{+} \subseteq X \times X$ have the respective meaning for sets of variables.

2.2.2.2 Semantics

In the following, I define what it means for a constraint system to be satisfied. In a nutshell, I assign every occurring expression a monotone function on an appropriately chosen partially ordered set L. This allows to evaluate the left-hand side and the right-hand side of a constraint and to determine whether the left-hand side is greater than or equal to the right-hand side, with respect to the partial order on L.
Let $(L, \leq)$ be a partially ordered set.
A *variable assignment* is a function $\psi : X \to L$. Furthermore, I assign every function symbol $f \in \mathcal{F}$ with $a(f) = n$ a monotone *interpretation* $\alpha(f) : L^n \to_{mon} L$. Now I inductively define the interpretation $[\![t]\!] : (X \to L) \to L$ of expressions:

- for $x \in X : [\![x]\!](\psi) = \psi(x)$
- $[\![f(t_1, \ldots, t_n)]\!](\psi) = \alpha(f)([\![t_1]\!](\psi), \ldots, [\![t_n]\!](\psi))$

By straight-forward induction I can show Lemma 2.10, which states basic but important properties of $[\![\cdot]\!]$ and FV.

Lemma 2.10. *Let $t \in Expr(X, \mathcal{F})$ be an expression. Then the following statements hold:*

1. *$[\![t]\!] : (X \to L) \to L$ is monotone.*
2. *If ψ and ψ' are two variable assignments with*

$$\forall x \in FV(t).\ \psi(x) \leq \psi'(x),$$

then

$$[\![t]\!](\psi) \leq [\![t]\!](\psi').$$

In particular: $\forall x \in FV(t).\ \psi(x) = \psi'(x)$ implies $[\![t]\!](\psi) = [\![t]\!](\psi')$.

A variable assignment $\psi : X \to L$ *satisfies a constraint* $x \geq t$ if $\psi(x) \geq [\![t]\!](\psi)$. ψ *satisfies a constraint system* if it satisfies all $c \in C$. I this case, I call ψ a *solution* of C. ψ is called *least solution* of C if $\psi \leq \psi'$ for all solutions ψ' of C. Given a constraint system C I define the *corresponding functional*

$$F_C : (X \to L) \to (X \to L)$$

by

$$F_C(\psi)(x) = \bigsqcup \{[\![t]\!](\psi) | x \geq t \in C\}$$

For the moment, I implicitly require that F_C is well-defined, i.e. that all least upper bounds on the right-hand side exist.
An easy calculation shows that F_C is monotone. Furthermore, there is a strong connection between the solutions of C and the reductive points of F_C:

Lemma 2.11. *ψ is a solution of C if and only if $F_C(\psi) \leq \psi$.*

Proof. " $\Longleftarrow$ ": Assume $F_C(\psi) \leq \psi$ and let $x \geq t \in C$. Then, by definition of F_C,

$$F_C(\psi)(x) \geq [\![t]\!](\psi)$$

Since $F_C(\psi) \leq \psi$, this implies

$$\psi(x) \geq [\![t]\!](\psi)$$

Hence, ψ is a solution of C.
" $\Longrightarrow$ ": If we have $\psi(x) \geq [\![t]\!](\psi)$ for every $x \geq t \in C$, then

$$\forall x \in X. \psi(x) \geq \bigsqcup \{[\![t]\!](\psi) | x \geq t \in C\} = F_C(\psi)(x)$$

which is equivalent to $\psi \geq F_C(\psi)$. □

The connection between the solutions of C and the reductive points of F_C makes the theory developed in section 2.2 available. The least solution exists and coincides with the least fixed-point of F_C if

- L is a CCPO and F_C is well-defined and continuous, or
- L is a complete lattice.

If L is a CCPO and F_C is well-defined and continuous (which is for example the case if L is a complete lattice satisfying the ascending chain condition), then the least solution is characterized by

$$lfp(F_C) = \bigsqcup_{i \in \mathbb{N}} F_C^i(\bot).$$

2.2.3 Solving Monotone Constraint Systems

Now we know how monotone constraint systems look like and what they mean. In the following, I show algorithms that can solve them under some common conditions.

Let C be a monotone constraint system over a complete lattice L that satisfies the ascending chain condition. Then Kleene's fixed-point theorem (Corollary 2.7) suggests a simple algorithm to compute the least solution of C, which is presented in Algorithm 1.

Algorithm 1 is indeed very simple. From previous considerations, it can easily be seen that Algorithm 1 must terminate and that upon termination, the value of $\mathcal{A}$ is indeed the least fixed-point of $\mathcal{F}_C$ and therefore the least solution of C. On the other hand, it is very inefficient: $F_C(\mathcal{A})$ is computed by computing $F_C(\mathcal{A}(x))$ for all $x \in X$ and $F_C(\mathcal{A})(x)$ is computed by evaluating $[\![t]\!](\mathcal{A})$ for all t such that $x \geq t \in C$ and computing their supremum. In effect, all constraints are evaluated and this is re-iterated for all constraints, even if only one value changed.

More efficient algorithms can be obtained by carefully tracking the constraints that need to be updated. For every constraint $x \geq t$, according to Lemma 2.10, $[\![t]\!]$ only depends on $FV(t)$, i.e. it can only change if the value of at least one $y \in FV(t)$ has changed. Conversely, this means: If we re-evaluate $\mathcal{A}(x)$ and it changes, then afterwards we only have to consider those constraints where x occurs on the right-hand side, i.e. the constraints

Algorithm 1: Simple algorithm to compute the least solution of a monotone constraint system

```
Input: a finite monotone constraint system $C$
Result: the least solution of $C$
$\mathcal{A} \leftarrow \bot$
$changed \leftarrow true$
while $changed$ do
    $changed \leftarrow false$
    $\mathcal{A}_{old} \leftarrow \mathcal{A}$
    $\mathcal{A} \leftarrow F_C(\mathcal{A})$
    if $\mathcal{A} \neq \mathcal{A}_{old}$ then
        $changed \leftarrow true$
return $\mathcal{A}$
```

of the form $y \geq t$ such that $x \in FV(t)$, or, in the spirit of Definition 2.9, which are related to $x \geq t$ via C's $\rightsquigarrow$-relation.

We now can give an improved version of Algorithm 1, which is shown in Algorithm 2. This algorithm does not apply F_C globally but considers each constraint individually.

Like Algorithm 1, Algorithm 2 maintains a function $\mathcal{A} : X \rightarrow L$ which is initialized to $\bot$ for all x and then updated incrementally. Additionally, it maintains a list of constraints which have to be considered later – the so-called *worklist*[3]. This list initially contains all constraints, which ensures that every constraint is considered at least once. In the iteration phase, the worklist is processed as follows: First, a constraint $x \geq t$ is removed. Then, the algorithm checks whether the current value of $\mathcal{A}$ satisfies this constraint. If this is the case, the constraint can be discarded and $\mathcal{A}$ is left unchanged. If this is not the case, then $\mathcal{A}$ is updated. After the update, $\mathcal{A}$ satisfies $x \geq t$. Lastly, all constraints which may be influenced by x, i.e. all $x' \geq t'$ such that $x \rightsquigarrow x'$, are inserted into the worklist. This ensures that every constraint that may have been violated by the recent update of $\mathcal{A}$ will be considered again later.

[3] I use the term work*list* here although the algorithms treat W like a set. The reason is that I do not want to deviate from the literature too much.

Algorithm 2: Worklist algorithm for computing the least solution of a monotone constraint system (adapted from [130, Table 6.1])

```
Input: a finite monotone constraint system $C$
Result: the least solution of $C$
$W \leftarrow \emptyset$
foreach $x \geq t \in C$ do
    $W \leftarrow W \cup \{x \geq t\}$
    $\mathcal{A}(x) \leftarrow \bot$
while $W \neq \emptyset$ do
    $x \geq t \leftarrow remove(W)$
    $new \leftarrow eval(t, \mathcal{A})$
    if $\mathcal{A}(x) \not\geq new$ then
        $\mathcal{A}(x) \leftarrow \mathcal{A}(x) \sqcup new$
        foreach $x' \geq t'$ such that $x \geq t \rightsquigarrow x' \geq t'$ do
            $W \leftarrow W \cup \{x' \geq t'\}$
return $\mathcal{A}$
```

By considering each constraint at least once and ensuring that a constraint is considered again if the value of an influencing variable has changed, it can be shown that Algorithm 2 indeed computes the least solution of C, as stated by Theorem 2.12.

Theorem 2.12 (cf. Lemma 6.4 in [130])**.** *If C is finite and L satisfies the ascending chain condition, then Algorithm 2 computes the least solution of C.*

Note that Algorithm 2 does not specify how elements are inserted into the worklist and how they are removed. This means that the algorithm is correct no matter in which order the constraints are processed, so that it can be further improved by optimizing the evaluation order.
Instead of maintaining constraints in a worklist, one can also maintain variables instead of constraint. This leads to Algorithm 3. Here, the worklist contains all the variables whose value needs to be updated.

Theorem 2.13. *If C is finite and L satisfies the ascending chain condition, then Algorithm 3 computes the least solution of C.*

Proof. We prove the claim as in the proof of [130, Lemma 6.4] in three steps:

Algorithm 3: A variable-oriented variant of Algorithm 2

```
    Input: a finite monotone constraint system $C$
    Result: the least solution of $C$
 1  foreach $x \geq t \in C$ do
 2      $\mathcal{A}(x) \leftarrow \bot$
 3      $W \leftarrow W \cup \{x\}$
 4  while $W \neq \emptyset$ do
 5      $x \leftarrow remove(W)$
 6      $old \leftarrow \mathcal{A}(x)$
 7      forall $x \geq t \in C$ do
 8          $\mathcal{A}(x) \leftarrow \mathcal{A}(x) \sqcup [\![t]\!](\mathcal{A})$
 9          if $\mathcal{A}(x) \neq old$ then
10              foreach $x' \geq t' \in C$ such that $x \in FV(t')$ do
11                  $W \leftarrow W \cup \{x'\}$
12  return $\mathcal{A}$
```

1. Algorithm 3 terminates.

2. If $\mathcal{A}_i$ denotes the value of $\mathcal{A}$ after the i-th iteration, then

$$\forall i \in \mathbb{N}.\, \mathcal{A}_i \leq lfp(F_C)$$

3. Upon termination, we have

$$\mathcal{A} \geq lfp(F_C)$$

For the first two steps, we refer to the proof of [130, Lemma 6.4]. For the third step, we prove that the loop in lines 4–11 maintains the following invariant:

$$\textbf{(Inv)}\quad \forall x \geq t \in C.\, x \notin W \implies \mathcal{A}(x) \geq [\![t]\!](\mathcal{A})$$

First we note that this invariant proves our claim: Upon termination, the worklist is empty. The invariant **(Inv)** implies then that every constraint is satisfied.

Next we show that **(Inv)** holds before the first loop iteration. But this is clear since after the initialization loop in lines 1–3 has finished, every variable occurring on the left-hand side of a constraint in C is contained in W. Hence, the premise of **(Inv)** is false, so **(Inv)** holds before the first loop iteration.

Next we show that **(Inv)** is preserved by each iteration of the loop in lines 4–11. So, consider the i-th loop iteration. Let $\mathcal{A}_{Old}$ and W_{Old} be the values of $\mathcal{A}$ and W at the beginning and $\mathcal{A}_{New}$ and W_{New} be the values of $\mathcal{A}$ and W at the end of the i-th iteration, respectively. We assume that **(Inv)** holds at the beginning of the i-th iteration and show that it still holds at the end. So, consider any constraint $x \geq t \in C$ with $x \notin W_{New}$.

We distinguish two cases:

1. $x \notin W_{Old}$: Then $x \geq t$ was satisfied at the beginning of the i-th loop iteration and x cannot have been removed in this iteration, which means that $\mathcal{A}(x)$ is not touched. Moreover, no variable from $FV(t)$ can have been touched in this iteration: Otherwise, line 11 would have been executed and $x \in W_{New}$. Hence, $x \geq t$ is still satisfied at the end of the iteration.

2. $x \in W_{Old}$: Since $x \notin W_{New}$, x must be the variable which is processed in the i-th iteration. Then, at some point, line 8 is executed for $x \geq t$, so that $x \geq t$ is satisfied afterwards. Now we make three observations that together allow us to conclude that $x \geq t$ is still satisfied at the end of the i-th iteration: First, we see that line 8 is the only place in the loop where $\mathcal{A}(x)$ is modified. Secondly, $\mathcal{A}$ is only modified for variable x and no other variable. Finally, we make the observation that no variable from $FV(t)$ can have been touched in this iteration: If that were the case, then this would necessarily entail $x \in FV(t)$. But then line 11 would be executed for $x \geq t$ and we would have $x \in W_{New}$ (which we have not).

From these three observations, it follows that the i-th iteration only changes $Analysis(x)$ and leaves $[\![t]\!](Analysis)$ unchanged. Together with the fact that line 8 changes $Analysis(x)$ upwards, we can conclude that $x \geq t$ is still satisfied at the end of the i-th iteration.

This concludes the proof that **(Inv)** is preserved by the loop in lines 4–11. □

2.3 Inductive Definitions

In later chapters, I will use inductive definitions at various places. Furthermore, I will use induction principles derived from the respective definition. In this section, I make these notions precise by applying the theoretic foundations compiled in section 2.2. A general version of the following definitions and results can be found in the literature [7].

To improve presentation, I abbreviate $(x_1, \ldots, x_n) \in B^n$ as $\underline{\mathbf{x}} \in B^n$, for a given set B and $n \in \mathbb{N}$. Moreover, for $\underline{\mathbf{x}} \in B^n$, I write $\underline{\mathbf{x}}_i$ to denote the i-th component of $\underline{\mathbf{x}}$.

Definition 2.14. *Let X be an arbitrary set. An* operator *(on X) is a partial function $f : X^n \to X$ for some $n \in \mathbb{N}$, which is also called the* arity *of f and which is written as $ar(f)$.*

Definition 2.15. *For a set $\mathcal{F}$ of operators (on a set X), I say that $A \subseteq X$ is* closed under *$\mathcal{F}$ if*

$$\forall f \in \mathcal{F}.\ \forall \underline{\mathbf{x}} \in X^{ar(f)}.\ \underline{\mathbf{x}} \in A^{ar(f)} \cap dom(f) \implies f(\underline{\mathbf{x}}) \in A \tag{2.13}$$

Usually, I will specify $\mathcal{F}$ by giving a list of closure properties of the form (2.13).

Mostly for layout reasons, I express (2.13) in the following way:

$$\frac{\underline{\mathbf{x}}_1 \in A \quad \ldots \quad \underline{\mathbf{x}}_{ar(f)} \in A \quad \underline{\mathbf{x}} \in dom(f)}{f(\underline{\mathbf{x}}) \in A} \tag{2.14}$$

Occasionally, I will omit the "$\in A$"'s if they are clear from the context. I will also omit other assertions that restrict elements to membership in a given set if these assertions do not restrict the elements more than other assertions. I will also omit quantifiers and assume that all free variables are universally quantified.

Using (2.13) or (2.14) not only specifies an operator set on X but also a canonical subset of X. I introduce this set in the following.

Definition 2.16. *Let $\mathcal{F}$ be a set of operators on the set X. I say that $A \subseteq X$ is inductively defined by $\mathcal{F}$ if*

$$A \textit{ is closed under } \mathcal{F} \tag{2.15}$$

$$\forall B \subseteq X.\ B \textit{ is closed under } \mathcal{F} \implies A \subseteq B \tag{2.16}$$

Remark 2.17. *For every set X and every set $\mathcal{F}$ of operators on X, there is exactly one subset $A \subseteq X$ that is inductively defined by $\mathcal{F}$.*

Proof. Define A by

$$A \overset{def}{=} \bigcap \{B \subseteq X \mid B \text{ is closed under } \mathcal{F}\} \tag{2.17}$$

Then it can be easily seen that A satisfies the two conditions in Definition 2.16. Now let $A_1 \subseteq X$ and $A_2 \subseteq X$ be two subsets of X that are inductively defined by $\mathcal{F}$. Then by (2.15), both A_1 and A_2 are closed under $\mathcal{F}$. Hence, by (2.16), we have $A_1 \subseteq A_2$ and $A_2 \subseteq A_1$. Due to anti-symmetry of $\subseteq$, it follows that $A_1 = A_2$. □

Because of (2.16), one can also say that the set that is inductively defined by $\mathcal{F}$ is *the least subset that is closed under* $\mathcal{F}$
For an inductively defined set $A \subseteq X$, the following proof principle can be applied.

Theorem 2.18. *Let $A \subseteq X$ be inductively defined by $\mathcal{F}$ and let $P : X \to \{true, false\}$ be a statement about elements of X. Suppose that we can show*

$$\forall f \in \mathcal{F}.\ \forall \underline{\mathbf{x}} \in X^{ar(f)}.\ \underline{\mathbf{x}} \in dom(f) \wedge \bigwedge_{i=1}^{ar(f)} P(\underline{\mathbf{x}}_i) \implies P(f(\underline{\mathbf{x}})). \tag{2.18}$$

Then $\forall a \in A.\ P(a)$.

Proof. Define $C_f, C_{\mathcal{F}} : 2^X \to 2^X$ by

$$C_f(B) \overset{def}{=} \{f(\underline{\mathbf{x}}) \mid \underline{\mathbf{x}} \in B^{ar(f)} \cap dom(f)\} \tag{2.19}$$

$$C_{\mathcal{F}}(B) \overset{def}{=} \bigcup_{f \in \mathcal{F}} C_f(B) \tag{2.20}$$

Then it is easy to see that

1. $C_{\mathcal{F}}$ is a monotone function on the complete lattice $(2^X, \subseteq)$.

2. The subsets of X that are closed under $\mathcal{F}$ are exactly the reductive points of $C_{\mathcal{F}}$.

From Theorem 2.1, we know that

$$lfp(C_{\mathcal{F}}) = \bigcap Red(C_{\mathcal{F}})$$

Hence, $lfp(C_{\mathcal{F}})$ is the least subset of X that is closed under $\mathcal{F}$.
Another basic observation is that for all $\mathcal{A} \subseteq 2^X$ we have

$$C_{\mathcal{F}}(\bigcup \mathcal{A}) = \bigcup_{A \in \mathcal{A}} C_{\mathcal{F}}(A)$$

In particular, $C_{\mathcal{F}}$ is a continuous function on the CCPO 2^X. Hence, we can use Lemma 2.4 to give a proof for Theorem 2.18. Abbreviate $lfp(C_{\mathcal{F}})$ as A and define

$$\mathcal{P} \overset{def}{=} \{B \subseteq A \mid \forall b \in B.\, P(b).\}$$

Then we need to show $A \in \mathcal{P}$. By Lemma 2.4, it suffices to show

(2.21) $\emptyset \in \mathcal{P}$

(2.22) $\forall \mathcal{B} \subseteq 2^X.\, \mathcal{B} \subseteq \mathcal{P} \implies \bigcup \mathcal{B} \in \mathcal{P}$

(2.23) $\forall B \subseteq X.\, B \in \mathcal{P} \implies C_{\mathcal{F}}(B) \in \mathcal{P}$

The first two properties are easy to see. Now consider (2.23). Let $B \in \mathcal{P}$. We need to show $C_{\mathcal{F}}(B) \in \mathcal{P}$, that is

$$\bigcup_{f \in \mathcal{F}} \{f(\underline{\mathbf{x}}) \mid \underline{\mathbf{x}} \in B^{ar(f)} \cap dom(f)\} \in \mathcal{P}$$

By (2.22), it suffices to show that

$$\{f(\underline{\mathbf{x}}) \mid \underline{\mathbf{x}} \in B^{ar(f)} \cap dom(f)\} \in \mathcal{P}$$

for all $f \in \mathcal{F}$. For this, it is sufficient to show

$$\forall f \in \mathcal{F}.\, \forall \underline{\mathbf{x}} \in B^{ar(f)} \cap dom(f).\, P(f(\underline{\mathbf{x}}))$$

So let $f \in \mathcal{F}$ and $\underline{\mathbf{x}} \in B^{ar(f)} \cap dom(f)$. From $B \in \mathcal{P}$ we can derive

$$\bigwedge_{i=1}^{ar(f)} P(x_i)$$

and by (2.18),

$$P(f(\underline{\mathbf{x}})) \in B,$$

follows, as desired. □

In this thesis, I will at several points consider a set A that is defined in some non-inductive way. Then, I will specify a set $\mathcal{F}$ of operators and propose that A is inductively defined by $\mathcal{F}$. In order to show this, by Remark 2.17, I only need to show that A is the least set that is closed under $\mathcal{F}$. The following theorem formalizes this argument.

Theorem 2.19. *Let X be a set, $\mathcal{F}$ a set of operators on X and $A \subseteq X$. Furthermore, let $X_0 \subseteq X$ be the least subset of X that is closed under $\mathcal{F}$. In order to show that A is inductively defined by $\mathcal{F}$, it suffices to show the following two statements.*

(2.24) *A is closed under $\mathcal{F}$.*

(2.25) *$A \subseteq X_0$, where X_0 is the least subset of X that is closed under $\mathcal{F}$.*

Proof. (2.24) is exactly the same as (2.15). (2.25) is a reformulation of (2.16) with regards to the representation (2.17). Hence, (2.24) and (2.25) indeed imply together that A is inductively defined by $\mathcal{F}$ according to Definition 2.16. □

2.4 Symbol Sequences

Let E be a finite set. I will refer to E as *alphabet* and to the elements of E as *letters*, or *symbols*, respectively.

$$E^n \overset{def}{=} \prod_{i=1}^{n} E$$

is the set of sequences with items in E and length n. In particular, E^0 contains exactly one element ϵ that I also call *the empty sequence*.

$$E^\star \overset{def}{=} \bigcup_{i \geq 0} E^i$$

is the set of all sequences with items in E.

I will use some abbreviating notations: I will use an interval notation for ranges of integers. For instance $[i, j] \subseteq \mathbb{N}$ is meant to be the set of non-negative integers which are $\geq i$ and $\leq j$, the notation for open and half open intervals $]i, j[$, $[i, j[$ and $]i, j]$ have respective meaning. For a sequence $\pi \in E^\star$ of symbols from E, π^i and $|\pi|$ denote the i-th item in π and the length of π, respectively. Moreover, I define $\pi^{[i,j]}$, $\pi^{[i,j[}$, $\pi^{]i,j]}$ and $\pi^{]i,j[}$ as the sub-sequence of π that is obtained by taking only the items of the respective intervals. So, for example, if $\pi = \pi^0 \cdot \ldots \cdot \pi^{n-1}$, then $\pi^{[i,j]} = \pi_i \cdot \ldots \cdot \pi_j$. $\pi^{<i}$ is short for $\pi^{[0..i[}$, $\pi^{>i}$ is short for $\pi^{]i,|\pi|-1]}$, $\pi^{\leq i}$ and $\pi^{\geq i}$ are defined analogously. Generally, I consider π' a sub-sequence of π if and only if $\pi' = \pi^I$ for some interval $I \subseteq range(\pi)$. Moreover, although I will mostly treat π^I as a sequence of its own right, I consider it to also implicitly contain the interval I and the sequence π from which it was extracted. This avoids ambiguities for cases in which there are multiple occurrences of a sub-sequence in a sequence. For instance, for $\pi = abab$, the sub-sequence ab could be represented as both $\pi^{[0,1]}$ and $\pi^{[2,3]}$. By also incorporating the interval, I specify which occurrence of ab I mean.
Both π and I will be clear from the context, unless stated otherwise.
For a sub-sequence $\pi' = \pi^I$ of π, I denote with $range_\pi(\pi') := I$ the range of indices which need to be selected from π to obtain π'. If I omit the index, then I mean the full range of π, so $range(\pi) = \{0, ...|\pi| - 1\}$.
Lastly, $Prefixes(\pi) := \{\pi^{<i} \mid i \in range(\pi)\}$ is the set of all prefixes of π.

Sequences can be concatenated. For $\pi \in E^m$, $\pi' \in E^n$, $\pi'' \overset{def}{=} \pi \cdot \pi'$ is the sequence of length $m + n$ with $\pi''^{<|\pi|} = \pi$ and $\pi''^{\geq|\pi|} = \pi'$. This defines a binary operation on $E^\star$, i.e. a function

$$\cdot : E^\star \times E^\star \to E^\star.$$

This operation is associative, that is $(\pi_1 \cdot \pi_2) \cdot \pi_3 = \pi_1 \cdot (\pi_2 \cdot \pi_3)$ and has ϵ as neutral element, that is we have $\pi \cdot \epsilon = \epsilon \cdot \pi = \pi$. Moreover, for every $\pi \in E^\star$, if we split $range(\pi)$ into adjacent intervals $I_1, \ldots, I_k$ that only have endpoints in common, then we can write

$$\pi = \pi^{I_1} \cdot \ldots \cdot \pi^{I_k}.$$

Specifically, by considering each symbol in a sequence π as a sequence of its own right, then we can write

$$\pi = \pi^0 \cdot \ldots \cdot \pi^{|\pi|-1}$$

2.5 Directed Graphs

The following definitions are standard and can be found in any text book about graph theory [44, 55].

Definition 2.20. *A* directed graph *is a pair $G = (N, E)$ where N is a set of* nodes *and $E \subseteq N \times N$ is a binary relation over N whose elements are called* edges*.*

Definition 2.21 (edge labels)**.** *An edge-labeled directed graph is a tuple $G = (N, E, L, l)$ such that (N, E) is a directed graph and*

$$l : E \to L$$

is a function from edges to a finite, non-empty set L of labels*. I assume that L always contains the empty label τ.*

For $e \in E$, I will write $n \xrightarrow{e} n'$ if $e = (n, n')$ or, for labeled edges, if $\exists l.(n, l, n') \in E$. Given a (labeled) edge $n \xrightarrow{e} n'$, $src(e) = n$ and $tgt(e) = n'$ denote *the source* and *target* of e, respectively.
For a given sequence $\pi \in E^\star$, $\pi \neq \epsilon$, I define *start* and *end* as follows:

$$start(\pi) = src(\pi^0) \tag{2.26}$$

$$end(\pi) = tgt(\pi^{|\pi|-1}) \tag{2.27}$$

A *path* is an edge sequence $\pi \in E^\star$ with the property

$$\forall 1 \leq i \leq |\pi| - 1.\, src(\pi^i) = tgt(\pi^{i-1}).$$

We can characterize the paths in G as follows.

The set $Paths_G \subseteq N \times N \times E^\star$ is inductively defined by the following rules:

$$(\textsc{path-empty})\ \frac{-}{(s,s,\epsilon) \in Paths_G}$$

$$(\textsc{path-extend})\ \frac{(s,t,\pi) \in Paths_G \qquad t \xrightarrow{e} t'}{(s,t',\pi \cdot e) \in Paths_G}$$

Instead of $(s,t,\pi) \in Paths_G$, I also write $\pi \in Paths_G(s,t)$ and refer to $Paths_G$ as function $N \times N \to 2^{E^\star}$.

It is easy to see that $Paths_G$ relates the pairs $(s,t) \in N \times N$ to the paths that start in s and end in t, as formalized in Lemma 2.22. Occasionally, I will consider $Paths_G$ not only as a relation but also as a subset of $E^\star$ by identifying it with $\bigcup_{s,t \in N}\{\pi \in E^\star \mid (s,t,\pi) \in Paths_G\}$. It will be clear from context, which of the two I mean.

Lemma 2.22. *For all $\pi \in E^\star$ and all $s,t \in N$ the following two statements are equivalent:*

(1) $(s,t,\pi) \in Paths_G$

(2) *π is a path and* $(s = t \wedge \pi = \epsilon \vee start(\pi) = s \wedge end(\pi) = t)$

Proof. " $\Longrightarrow$ " can be seen by induction on $(s,t,\pi) \in Paths_G$, " $\Longleftarrow$ " can be shown by induction on the length of sequences in $E^\star$. □

Lastly, I state three elementary properties of paths that I will make use of later.

Lemma 2.23. *If $\pi \in Paths_G(s,t)$ and $\pi' \in Paths_G(t,t')$, then $\pi \cdot \pi' \in Paths_G(s,t')$.*

Proof. Fix $s,t \in N$ and $\pi \in Paths_G(s,t)$. Then we can show

$$\forall \pi' \in E^\star.\ \forall t' \in N.\ \pi' \in Paths_G(t,t') \implies \pi \cdot \pi' \in Paths_G(s,t')$$

by induction on the length of π'. □

Lemma 2.24. *For all $\pi, \pi' \in E^\star$, the following statement holds:*
If $\pi \cdot \pi' \in Paths_G(s,t)$, then there is $t' \in N$ such that $\pi \in Paths_G(s,t')$ and $\pi' \in Paths_G(t',t)$.

Proof. Fix $\pi' \in E^\star$ and $t \in N$. Then we can show

$$\begin{aligned}&\forall \pi \in E^\star.\, \forall s \in N.\, \pi \cdot \pi' \in \mathit{Paths}_G(s,t)\\ &\qquad\qquad\qquad \implies \exists t' \in N. \pi \in \mathit{Paths}_G(s,t') \wedge \pi' \in \mathit{Paths}_G(t',t)\end{aligned}$$

by induction on the length of π. □

Remark 2.25. *If π is a path and $i, j \in \mathit{range}(\pi)$, then $\pi^{[i,j]}$ is a path.*

Proof. This is clear by definition. □

My independence seems to vanish in the haze.
THE BEATLES

3 Program Dependence Graphs for Object-Oriented Programs

This chapter introduces the reader to common terminology and patterns of thought used in static program analysis. Moreover, it describes several techniques that enable JOANA to analyze security properties of JAVA programs. This prepares the reader for chapter 4, which shows several applications of the information flow control tool JOANA to software security. This chapter plays another important role: It describes two fundamental and widely used static program analysis techniques, namely *data-flow analysis on control-flow graphs* and *slicing on program dependence graphs*. These two techniques, which were developed more or less independently, face similar issues that have been solved with analogous approaches. In later chapters, I will present a common generalization of data-flow analysis and slicing that combines the strengths of both techniques.

The following sections are structured as follows. In section 3.1, I give an overview of basic aspects of static program analysis and introduce several central concepts that play important roles throughout this chapter. After that, section 3.2 and section 3.3 present data-flow analysis on control-flow graphs and slicing on program dependence graphs, respectively.

Finally, in section 3.4 I describe JOANA and show how it can be used to verify non-interference for JAVA programs. In this last section, I put an emphasis on the explanation of several techniques that are particularly important for the analysis of the programming language features of JAVA and also play a role in chapter 4.

3.1 Principles of Static Program Analysis

Program Analysis is concerned with techniques that allow to derive information about a given program and its properties. Roughly, program analysis techniques can be grouped into dynamic techniques and static techniques. Dynamic techniques generate information while executing the program [75], while static techniques aim to analyze a program without executing it [130, 76].
Another criterion that can be used to classify program analyses is whether they are automatic or not. As the name suggests, an automatic program analysis is usually another program that takes a given program as input, performs some algorithm on it and outputs its analysis result. In contrast, non-automatic techniques are either fully manual or *interactive*, that is they generally employ automatic techniques but query the user if they cannot complete their task in an automatic fashion.
In this thesis, I focus on automatic and static program analysis techniques such as *data-flow analysis* [101] and *static program slicing* [165, 166, 58]. In the following, I assume that all analyses are static and automatic, unless explicitly stated otherwise. Furthermore, I assume that the programs under analysis are written in a Turing-complete language.
Two important concepts in program analysis are *soundness* and *precision*[4]. Before I explain these two concepts, I first introduce some formalisms: Let ϕ be a property of programs, that is a given program $\mathcal{P}$ either can satisfy ϕ, written $\mathcal{P} \models \phi$, or not, written $\mathcal{P} \not\models \phi$. The property ϕ can for example make a statement about $\mathcal{P}$'s semantics, i.e. about how $\mathcal{P}$ transforms states or can state that $\mathcal{P}$ is secure in some sense.
Now let $\mathcal{A}$ be a program analysis whose goal is to analyze programs with respect to ϕ. Formally, we can imagine $\mathcal{A}$ as a function that takes a program $\mathcal{P}$ as input and outputs either "$\mathcal{P}$ satisfies ϕ" or "$\mathcal{P}$ does not satisfy ϕ". We write the former as $\mathcal{P} \vdash_{\mathcal{A}} \phi$ and the latter as $\mathcal{P} \nvdash_{\mathcal{A}} \phi$.
Note that the result of $\mathcal{A}$ a priori has nothing to do with whether $\mathcal{P}$ actually satisfies ϕ or not. It is merely a consequence of the formal reasoning performed by $\mathcal{A}$.
The notions of soundness and precision establish a connection between $\models$ and $\vdash_{\mathcal{A}}$:

[4] In the area of formal systems, precision is also called *completeness*.

1. $\mathcal{A}$ is said to be *sound* with respect to ϕ if

$$\mathcal{P} \vdash_{\mathcal{A}} \phi \implies \mathcal{P} \models \phi.$$

That is, if $\mathcal{A}$ concludes that $\mathcal{P}$ satisfies ϕ, then this is indeed the case.

2. $\mathcal{A}$ is said to be *complete* with respect to ϕ if

$$\mathcal{P} \models \phi \implies \mathcal{P} \vdash_{\mathcal{A}} \phi.$$

That is, if $\mathcal{P}$ has property ϕ, then $\mathcal{A}$ is able to derive that.

If $\mathcal{A}$ is sound, then it can give the guarantee that a program actually satisfies a property. If it is additionally complete, then it is actually able to *decide* ϕ. Unfortunately, there can be no static automatic analysis for a non-trivial property of programs written in a Turing-complete programming language that is both sound and complete [142]. Hence, analysis designers must make compromises, i.e. analyses usually are usually not sound or not complete (or neither of the two).

In the context of static program analysis, completeness is commonly also referred to as *precision*. Throughout this thesis, I use both terms interchangeably.

It is common to focus on soundness and sacrifice precision, especially in security analyses where one aims to give strong guarantees for programs. However, if an analysis is not precise, then it may raise *false alarms*, e.g. report that a program is insecure although it is not. This can undermine the credibility of an analysis.

Hence, another goal of program analysts is to minimize false alarms, i.e. make their analyses *as precise as possible*. Note that, although precision in theory is a binary property that can be either true or false, in this thesis I use it as a property that has multiple degrees, so that it can also be used comparatively or even quantified. Given two analyses $\mathcal{A}_1$ and $\mathcal{A}_2$, one can say that $\mathcal{A}_1$ is at least as precise as $\mathcal{A}_2$ if

$$\{\mathcal{P} \mid \mathcal{P} \models \phi \land \mathcal{P} \nvdash_{\mathcal{A}_1} \phi\} \subseteq \{\mathcal{P} \mid \mathcal{P} \models \phi \land \mathcal{P} \nvdash_{\mathcal{A}_2} \phi\}.$$

This defines a partial order on the set of analyses that can be used to compare different analyses.

There are different trade-offs that can be made with respect to analysis precision. In the following, I give an overview of a selection of them

```
int z = input();
x = 2;
if (z * z + z + 1 == 0) {
  x = 3;
}
```

(a)

```
int z = input();
x = 2;
if (z * z - 9 == 0) {
   x = 3;
}
```

(b)

Figure 3.1: Example for the impact of value-sensitivity

that play some role in this thesis. For illustration, I will use the example property *upon program termination, variable x always has the value 2* that I write as γ_x.

One trade-off static analyses can make is how precisely they handle values, for example numbers and algebraic identities.

Consider the two programs in Figure 3.1. Since the quadratic function $z^2 + z + 1$ does not have any integral zeros, it is relatively easy to see that `x` is always 2 upon termination of the program in Figure 3.1a, hence γ_x holds for it. However, γ_x is not satisfied by the program in Figure 3.1b, since x is set to 3 if $z = 3$. A program analysis that is sound with respect to γ_x but does not reason about algebraic identities will fail to verify γ_x for the program on the left. There are analysis techniques that can deal with such challenges[128], but the techniques that I consider in this thesis usually do not reason about values beyond the use of limited constant propagation [126].

Another aspect of analysis precision is *flow-sensitivity*, a property that refers to the ability of an analysis to take the order of statements into account. As an example, it is clearly the case that the program $\mathcal{P}_1$ in Figure 3.2a satisfies γ_x, while program $\mathcal{P}_2$ in Figure 3.2b does not. Now consider an analysis $\mathcal{A}$ that is sound with respect to γ_x. Then it must be the case that $\mathcal{P}_2 \nvdash_{\mathcal{A}} \gamma_x$, because $\mathcal{P}_2 \not\models \gamma_x$. Now, if $\mathcal{A}$ is flow-insensitive, then it usually yields the same result for $\mathcal{P}_1$, as they only differ in the order of statements. Hence, a flow-insensitive analysis is usually not able to verify γ_x for $\mathcal{P}_1$. In contrast, it may be the case that a flow-sensitive analysis may consider $\mathcal{P}_1$ and $\mathcal{P}_2$ as different programs.

Context-sensitive program analyses consider not only the program statements, but also take their *execution context* into account. Examples for the execution context of a statement include the site from which a procedure

```
x = 3
x = 2
```

(a)

```
x = 2
x = 3
```

(b)

```
void main() {
  y = f(3);
  x = f(2);
}
void f(int a) {
  return a;
}
```

(c)

```
void main() {
  x = f(2);
  x = f(3);
}
void f(int a) {
  return a;
}
```

(d)

Figure 3.2: Illustration of different sensitivities: Programs a and b cannot be distinguished by a flow-insensitive analysis, programs c and d cannot be distinguished by a context-insensitive analysis

was called (in programs with multiple procedures), or the object on which a method is called (in object-oriented programs).

For example, it clearly can be seen that the program in Figure 3.2c satisfies γ_x, while the program in Figure 3.2d does not. However, a context-insensitive program analysis that is sound with respect to γ_x cannot verify Figure 3.2c: It has to "merge" the calls in lines 2 and 3 and deem it possible that `f` may also return 3, otherwise it would not be able to reject the program in Figure 3.2d.

3.2 Data-Flow Analysis on Control-Flow Graphs

In this section, I introduce *control-flow graphs*, a classical data structure of static analysis and *data-flow analysis*, which is an important, generic operation on control-flow graphs that forms the essence of many static program analyses.

This section is structured as follows: First, I introduce control-flow graphs in subsection 3.2.1. After that, I explain how data-flow analysis works in subsection 3.2.2.

The two subsections are structured analogously: First, they consider programs without procedures and then show how the respective formalism can be extended to the interprocedural case.

```
 1  read(n)
 2  if (n <= 1) {
 3    print(n)
 4  } else {
 5    a = 0
 6    b = 1
 7    c = a + b
 8    i = 0
 9    while (i < n) {
10      a = b
11      b = c
12      c = a + b
13      i = i + 1
14    }
15    print(c)
16  }
```

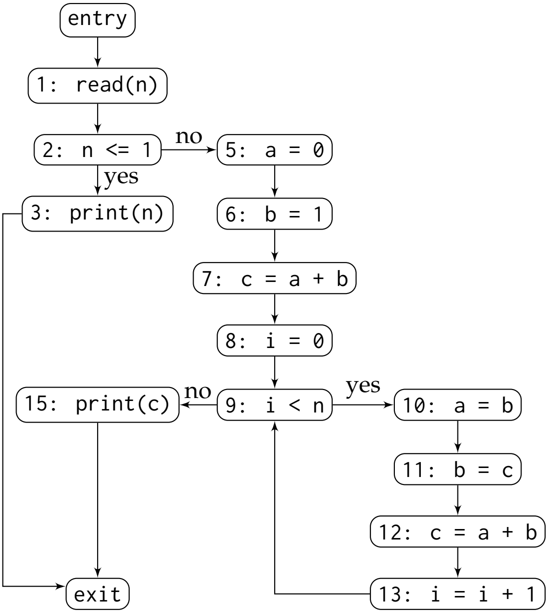


Figure 3.3: An example program and its control-flow graph

3.2.1 Control-Flow Graphs

Control-flow graphs are a classical program representation on which many program analyses operate. A control-flow graph of a given program $\mathcal{P}$ is a directed graph that represents the possible control-flow between $\mathcal{P}$'s statements and predicates.

3.2.1.1 Intraprocedural Control-Flow Graphs

An example of a simple program and its control-flow graph can be seen in Figure 3.3. Its nodes represent the program's statements and predicates. Directed edges connect nodes with those nodes which may immediately follow them in an execution.

There are two kinds of control-flow. The control-flow from line 1 to line 2 is unconditional: After the read operation has been executed, control is always transferred to the following **`if`** statement.

The control-flow from line 2 to line 5, however, is conditional. Line 5 is only executed if the predicate in line 2 is evaluated to `true`. If it is evaluated to `false`, line 3 is executed instead.

It is also a common assumption that control-flow graphs have a unique entry node from which all nodes in the control-flow graph are reachable and a unique exit node which can be reached by all nodes.

Next, I introduce control-flow graphs more formally.

Definition 3.1 (control-flow graph, [65, based on Definition 2.1])**.** *Given a procedure (or procedure-less program) $\mathcal{P}$, a control-flow graph (CFG) is an edge-labeled directed graph with two distinguished nodes $s, e \in N$, written $G = (N, E, s, e, L, l)$, with the following properties:*

- *N is the set of nodes and each statement or predicate in p is represented by a node $n \in N$,*
- *E is the set of edges representing the control flow between nodes,*
- *s, also called* start *or* entry *node, has no incoming edges,*
- *e, also called* exit node*, has no outgoing edges,*
- *L is a set of* labels *and $l : E \rightarrow L$ is a function that maps each edge to a label.*

The labels are used to model conditional flows. As they do not play an important role in this thesis, I will mostly ignore them in the following.

For intraprocedural graphs that are considered in isolation, a classical assumption is that for all $n \in N$ there is a path which starts at s and ends at n.

In the literature there is some variation in the concrete representation of control-flow graphs. Some authors [12, 106, 130] use a *node-oriented* notation, in which the nodes of the control-flow graph represent the statements, while others [46, 153, 127] use an *edge-oriented* notation, in which the edges are annotated with the statements. Also, there are some differences among the node-oriented notations: some of them use a different node for each statement, while others use nodes for each *basic block*, which are linear chains of statements.

I use a node-oriented notation of control-flow graphs in this thesis. Moreover, I will only consider basic blocks with one statement, unless I deviate from this convention explicitly.

3.2.1.2 Interprocedural Control-Flow Graphs

For programs with multiple procedures, control-flow graphs have to be adapted in order to model the procedure calls properly. Again, there are several notions in the literature, which differ slightly. In most of them, interprocedural control-flow graphs are families of intraprocedural control-flow graphs for each procedure. They mainly differ in the exact way in which they model calls and whether the procedural control-flow graphs are connected or not.
My definition follows notations used by De Sutter et al. [51] and Hammer [86].

Definition 3.2. *Let Proc be a finite set of procedures with main* $\in P$. *An interprocedural control-flow graph (ICFG) is a quadruple*

$$G = ((G_p)_{p \in Proc}, E_{call}, E_{ret}, \Phi)$$

- *For every* $p \in Proc$, $G_p = (N_p, E_p, s_p, e_p)$ *is an intraprocedural control-flow graph. For different procedures* $p, p' \in Proc$, *the corresponding control-flow graphs* G_p *and* $G_{p'}$ *are disjoint:* $N_p \cap N_{p'} = \emptyset$ *and* $E_p \cap E_{p'} = \emptyset$. *If* $e \in E_p$ *for some* $p \in Proc$, *e is called* intraprocedural *edge. The set of intraprocedural edges is denoted by* E_{intra}.

- $E_{call}, E_{ret} \subseteq \bigcup_{p,p'} N_p \times N_{p'}$ – *The elements of* E_{call} *are called* call edges *and the element of* E_{ret} *are called* return edges. *There is a bijective function* $\Phi : E_{call} \to E_{ret}$ *which maps call edges to their* corresponding *return edges (and vice versa). This function is also called* correspondence function.

- *If* $n \to n' \in E_{call}$, *then there are* $p, q \in Proc$ *such that* $n \in N_p \setminus \{e_p\}$ *and* $n' = s_q$. *For* $e_{call} \in E_{call}$, *I call* $src(e)$ *a* call node.

- *If* $n \to n' \in E_{ret}$, *then there are* $p, q \in Proc$ *such that* $n = e_p$ *and* $n' \in N_q \setminus \{e_q\}$. *For* $e_{ret} \in E_{ret}$, *I call* $tgt(e)$ *a* return node.

- *Call nodes have, apart from call edges, no further outgoing edges.*

- *Return nodes have, apart from return edges, no further incoming edges.*

Figure 3.4 shows how procedure calls can be modeled: Each call is represented by two nodes, one for the call itself and one for the point just after the call to which the called procedure returns. I call this point the

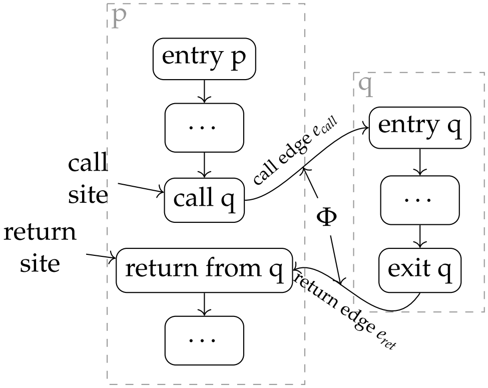


Figure 3.4: Additional control-flow structure for procedure calls

return site. A *call edge* e_{call} connects the call node with the entry node of the callee and a *return edge* e_{ret} connects the exit node of the callee to the return site. These two edges correspond to each other, i.e. $\Phi(e_{call}) = e_{ret}$.

3.2.2 Data-Flow Analysis

In this subsection, I introduce *data-flow analysis*, a classical static program analysis technique. My presentation roughly follows textbook literature [152, 130] and classic articles [102, 82, 101, 46] on the topic.

Roughly, data-flow analysis gathers information about the possible executions of a program. This information can then be used in subsequent analyses and program transformations.

Before I introduce data-flow analysis in the following, I want to consider the paths of control-flow graphs more closely.

A usual assumption about control-flow graphs, which I also make in this thesis, is that they are sound. This means that, given the control-flow graph G of a program $\mathcal{P}$, every execution of $\mathcal{P}$ is represented by a path in G. Note however, that this representation is not always exact. For example, a common simplifying assumption is that all outgoing edges of a given node might be taken by some program execution, regardless of the path that an execution had taken before.

An example of what this means can be seen in Figure 3.5.

Assuming that statement S does not change the outcome of b it is clear that no program execution can take the path 1 → 2 → 6 → 7. Since 1 → 2 is only traversed if b is evaluated to **true** at 1 and S does not change the outcome of b, any execution that traverses 1 → 2 must traverse 6 → 9 after that. Analyses that take into account the histories of paths are also called *path-sensitive*. Several publications on path-sensitive analysis can be found in the literature [89, 54, 35, 49]. In this thesis, I will only consider path-*in*sensitive analyses.

3.2.2.1 Intraprocedural Data-Flow Analysis

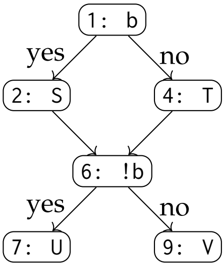
In the following, I concentrate on intraprocedural data-flow analyses that only consider a single procedure. The notions that I introduce however are also useful for interprocedural data-flow analysis, which I will describe in subsubsection 3.2.2.2.

A *data-flow framework* is a pair (L, F) that specifies the general structure of the data-flow analysis to be performed. The set L describes how information look like, while F is a set of functions on L which transform this information.

A data-flow analysis associates each node n in a given control-flow graph with some property of the control-flow paths from the graphs start node to n.

```
 1 if (b) {
 2   S
 3 } else {
 4   T
 5 }
 6 if (!b) {
 7   U
 8 } else {
 9   V
10 }
```

(a)



(b)

Figure 3.5: A code snippet and its control-flow graph

The elements of L are used to represent properties like "a is definitely 42", "the value of b is unknown" or "variable c has the value assigned to it in line 5". Such properties can be partially ordered with respect to the amount of information they provide. For example, if x represents the property "a is definitely 42" and y represents the property "a has an unknown value", then x can be considered to provide more information than y, which is formally expressed by $x \leq y$. Because we want to propagate information along the paths of a given control-flow graph, we need a way to combine values coming from different paths. The value resulting from a combination of $x, y \in L$ should provide at least as much information as the combined values but can provide no more information, in order to be safe. So a good candidate for such a combination is the *least upper bound* or *joins* of x and y, In order to have a well-defined structure it is customary to require L to have least upper bounds for arbitrary subsets $A \subseteq L$. In other words, L is assumed to be a *complete lattice* (see page 15).

The transfer functions are abstractions of the program's statements' effect on properties[5]. They are assumed to be *information-preserving*, that is, if x provides more information than y, then the same should hold for the transformed values. Formally, this means that transfer functions are *monotone*. Additionally, it is customary to require that F enjoys some closure properties: Usually, one assumes that F contains the identity function and is closed under composition and arbitrary joins. Since the set of monotone functions $L \rightarrow L$ has all these properties, F can in theory be assumed to contain all monotone functions. In practice however, this set is usually "too large" in the sense that it contains more functions than actually needed for the given data-flow analysis. Hence, one aims to find more precise descriptions of F that allow for effective or even efficient representations.

Classically, a data-flow framework is thought to be independent of the control-flow graphs they work on. To actually perform a data-flow analysis on a given control-flow graph, a data-flow framework needs to be *instantiated*.

Hence, a *data-flow instance* is a quintuple $(L, F, G, \rho, \mathit{init})$ that adds to a data-flow framework (L, F) a control-flow graph $G = (N, E, s, e)$ connects the two using (a) an *initial information init* $\in L$ that represents the properties

[5] The area of *abstract interpretation* is concerned with the systematic derivation of transfer functions from program semantics [46].

which hold at G's entry node s and (b) a function $\rho : E \rightarrow F$. This function associates each edge[6] e with a transfer function $\rho(e)$ that describes the effect of the statement $src(e)$ on the properties in L and $init \in L$. Instead of $\rho(e)$, I also write f_e.

By induction, transfer functions can be extended to the paths of G:

$$f_\epsilon = id$$
$$f_{\pi \cdot e} = f_e \circ f_\pi$$

Due to the closure properties of F, all f_π are elements of F.

The functions f_π describe how properties are transformed along control-flow paths. We are interested in the properties which hold at each node, no matter which path was taken. For this purpose, we take the least upper bound of all f_π and apply this function the initial information *init*. The result of this operation is also called the *merge-over-all paths* solution

$$MOP(n) = \bigsqcup_{\pi \in Paths_G(n)} f_\pi(init), \tag{3.1}$$

where $Paths_G(n)$ is the set of paths in G from s to n.

The function *MOP* is in some sense the ideal solution of the given data-flow analysis problem, so the goal of data-flow analysis is to compute *MOP*. Note that *MOP* cannot be computed directly using (3.1), since (3.1) potentially merges over infinitely many paths. Also, there are data-flow instances for which *MOP* is not computable at all [101]. However, there are sufficiently interesting and useful data-flow frameworks for which it is possible to compute a *safe over-approximation* of the corresponding *MOP*

[6] It may seem odd that in the formalism I use in this thesis, statements are represented by nodes but, in contrast, transfer functions are associated with edges. This "hybrid" variant of data-flow analyses, however, can also be found in the literature [82, 136], just like the "pure" variants that either associate both statements and transfer functions with nodes [102, 100, 10, 106] or edges [46, 153], respectively. All three variants appear to be equivalent. My decision is mostly for pragmatic reasons: Although I prefer to associate transfer functions with edges, I want to describe control-flow graphs and program dependence graphs uniformly and acknowledge that in previous work [58, 93], program dependence graphs are derived from control-flow graphs in which the nodes represent statements. Hence I inherit node-oriented control-flow graphs from these earlier presentations.

solution. A safe over-approximation is a function $A : N \rightarrow L$ that has the property

$$\forall n \in N.\, A(n) \geq MOP(n). \tag{3.2}$$

This property is equivalent to

$$\forall n \in N.\, \forall \pi \in Paths_G(n).\, A(n) \geq f_\pi(init). \tag{3.3}$$

Property (3.3) says that for every $n \in N$ and every $\pi \in Paths_G(n)$, $A(n)$ can provide no more information than $f_\pi(init)$. This is safe in the sense that no piece of information coming from a path ending in n is left out of $A(n)$. A program optimization (or any other program transformation) that solely relies on the information provided by A never changes a program's behavior, provided that the transfer functions make sure that the program semantics is abstracted faithfully.

For instance, suppose that a compiler aims to identify variables that always have the same values in order to substitute read accesses by the constant value. A possible data-flow analysis for this would then propagate along a control-flow path π whether the value of variable x stays the same on π (and the value itself, if applicable). Then $A(n)$ says something about whether x always has the same value on *any* control-flow path ending in n and provides this value, if applicable. But then $A(n)$ has to integrate the information of *all* paths ending in n. Otherwise, $A(n)$ could express that x's value is always the same up until n but ignores some of the paths where the value of x is indeed different, which means that $A(n)$ makes an unsafely wrong statement about the program under analysis. Clearly, the optimization step that substitutes accesses to x with the value computed by A would result in a program that differs in behavior from the original program.

A trivial safe over-approximation of MOP is the function which returns $\top$ for every n. This is of course safe because $\top$ provides no information at all. For our example, $A(n) = \top$ would mean that variable x may assume different values during program execution, even if this is not the case. Clearly, this may also be a wrong statement about the program under analysis but this time it can be considered safe because $A(n)$ provides no information that can be exploited by the subsequent optimization step.

This leads to a notion of *precision* specialized to data-flow analysis: For two safe over-approximations A and B, A is at least as precise as B if $\forall n \in N.\, A(n) \leq B(n)$. In other words, for every $n \in N$, A needs to provide at least as much information as B.

In general, one aims to obtain a safe over-approximation for *MOP* which provides as much information as possible. One way of obtaining such a solution is to solve the following system of *monotone constraints*:

Constraint System 3.1.

$$\begin{aligned} A(s) &\geq init \\ m \xrightarrow{e} n \implies A(n) &\geq f_e(A(m)) \end{aligned}$$

The idea of this system is to build up *MOP* "edge by edge".

By grouping together constraints with the same left-hand side, we can transform this constraint system to a system with one constraint per node:

$$A(n) \geq F_n((A(m))_{m \in N}).$$

Each $F_n : L^{|N|} \to L$ is monotone.

The whole constraint system can be described by one constraint

$$A \geq F(A) \tag{3.4}$$

where

$$F : (N \to L) \to (N \to L) \tag{3.5}$$

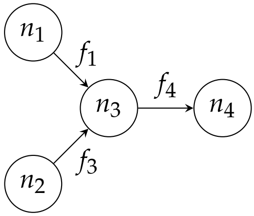
$$F = (F_n)_{n \in N} \tag{3.6}$$

$$F_n(A) = \bigsqcup_{m \xrightarrow{e} n} f_e(A(m)) \tag{3.7}$$

Since L is a complete lattice, $N \to L$ is, too. Moreover, $F : (N \to L) \to (N \to L)$ is monotone.

In order to solve the constraint system, we need to find a function $A : N \to L$ that satisfies (3.4).

The theory of complete lattices tells us that this is always possible: Theorem 2.1 implies that (3.4) has a unique least solution, that is a solution A_0



$$\begin{aligned} MOP(n4) &= \underline{f_3(f_1(init))} \sqcup \underline{f_3(f_2(init))} \\ MFP(n4) &= f_3(\underline{f_1(init) \sqcup f_3(f_2(init))}) \end{aligned}$$

Figure 3.6: Illustration of the effect of non-distributivity on the difference between *MOP* and *MFP*, based on Constraint System 3.1

such that $A_0 \leq A$ for every solution A of (3.4). So, A_0 is the most precise solution of (3.4). In the context of data-flow analysis, A_0 is also referred to as $\underline{M}\mathit{inimal}\ \underline{F}\mathit{ixpoint}$, or *MFP* for short.

Furthermore, it can be shown that A_0 is a safe over-approximation of *MOP*:

$$A_0 \geq MOP.$$

Moreover, Theorem 2.13 states that A_0 can be computed using Algorithm 3, provided that L satisfies the ascending chain condition (see page 16).

Note however, that *MFP* in general does not coincide with *MOP*. This is only the case if all the transfer functions enjoy a property that is called *distributivity*: A function $f : L \to L$ is distributive, if

$$\forall A \subseteq L.\ f(\bigsqcup A) = \bigsqcup f(A)$$

Whereas *MOP* first applies all transfer functions along the different paths, *MFP* applies joins at each node, for results along the incoming edges. In non-distributive instances, it is impossible to "pull joins out of" functions, which prevents *MFP* from coinciding with *MOP*. This is illustrated in Figure 3.6.

As an example of intraprocedural data-flow analysis, I want to discuss *reaching definitions*. This is a standard data-flow analysis applied in compilers and can also be used to compute data dependencies (see paragraph 3.3.2.1.1).

$$\begin{aligned}
A(entry) &\supseteq \emptyset \\
A(1) &\supseteq A(entry) \\
A(2) &\supseteq (A(1) - \{1\}) \cup \{1\} \\
A(3) &\supseteq A(2) \\
A(5) &\supseteq A(2) \\
A(6) &\supseteq (A(5) - \{5, 10\}) \cup \{5\} \\
A(7) &\supseteq (A(6) - \{6, 11\}) \cup \{6\} \\
A(8) &\supseteq (A(7) - \{7, 12\}) \cup \{7\} \\
A(9) &\supseteq (A(8) - \{8, 13\}) \cup \{8\} \\
A(9) &\supseteq (A(13) - \{8, 13\}) \cup \{13\} \\
A(10) &\supseteq A(9) \\
A(11) &\supseteq (A(10) - \{5, 10\}) \cup \{10\} \\
A(12) &\supseteq (A(11) - \{6, 11\}) \cup \{11\} \\
A(13) &\supseteq (A(12) - \{7, 12\}) \cup \{12\} \\
A(15) &\supseteq A(9) \\
A(exit) &\supseteq A(3) \\
A(exit) &\supseteq A(15)
\end{aligned}$$

node	value
entry	$\emptyset$
1	$\{1\}$
2	$\{1\}$
3	$\{1\}$
5	$\{1\}$
6	$\{1, 5\}$
7	$\{1, 5, 6\}$
8	$\{1, 5, 6, 7\}$
9	$\{1, 5, 6, 7, 8, 10, 11, 12, 13\}$
10	$\{1, 5, 6, 7, 8, 10, 11, 12, 13\}$
11	$\{1, 6, 7, 8, 10, 11, 12, 13\}$
12	$\{1, 7, 8, 10, 11, 12, 13\}$
13	$\{1, 8, 10, 11, 12, 13\}$
15	$\{1, 5, 6, 7, 8, 10, 11, 12, 13\}$
exit	$\{1, 5, 6, 7, 8, 10, 11, 12, 13\}$

Figure 3.7: The constraint system and its least solution for the reaching definition analysis applied to the example from Figure 3.3

Given a control-flow graph $G = (N, E, s, e)$, a *definition* of a variable x is a node $n \in N$ which represents an assignment statement $x := e$. Let $Def(x) \subseteq N$ be the definitions of variable x and assume for simplicity that each statement can define at most one variable. A definition *n may reach* a node n' if there is a path from n to n' on which there is (apart from n) no further definition of the variable defined by n.

For $n \in N$, the *reaching definitions of n* are the definitions which may reach n.

The data-flow framework for reaching definitions consists of $(2^D, F)$ where $D \subseteq N$ is the set of definitions in N, partially ordered by $\subseteq$. F consists of functions of the form $\lambda A.\ (A - K) \cup G$ where $K, G \subseteq D$ are sets[7] of definitions. It can easily be verified that F contains the identity function and is closed under composition and joins.

Let $m \overset{e}{\rightarrow} n$ be an edge in G. Then m's effect on the reaching definitions can be described as follows: If m is not a definition, then every definition which reaches m also reaches n. Therefore, all definitions reaching m are propagated to n. If m is a definition of variable x, then every definition of some variable $x' \neq x$ is propagated to n, but since m (re-)defines x, all previously reaching definitions of x are deleted and replaced by m. Formally, the edge transfer functions f_e are defined by

$$f_e = \begin{cases} \lambda A.\ A & \text{if } m \text{ does not define any variable} \\ \lambda A.\ (A - Def(x)) \cup \{m\} & \text{if } m \text{ is a definition of } x \end{cases} .$$

Since initially no variable is defined, the initial information *init* is $\emptyset$.

For the example program from Figure 3.3, the monotone constraint system resulting from the corresponding data-flow instance and its least solution is shown in Figure 3.7.

3.2.2.2 Interprocedural Data-Flow Analysis

Like intraprocedural data-flow analysis operates on intraprocedural control-flow graphs, interprocedural data-flow analysis operates on interprocedural control-flow graphs.

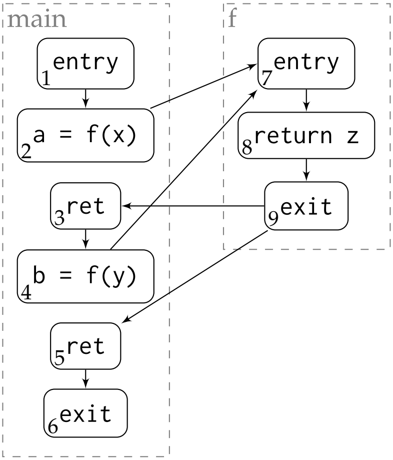
3.2.2.2.1 Context Problem Interprocedural control-flow graphs introduce a source of imprecision, which I already discussed in section 3.1. It is caused by a main benefit of having procedures in the first place, namely by the fact that procedures can be called from multiple call sites.

As an example, consider the program in Figure 3.8a. It contains a function `f` that is called from two call sites. Its control-flow graph is shown in Figure 3.8b. The graph contains for example the path $\pi_1 : 1 \rightarrow 2 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 3$, which corresponds to the normal execution of the given

[7] Data-flow analyses where the transfer functions can be expressed in this way are also called gen/kill or bit vector analyses.

```
void main(int x, int y) {
  a = f(x);
  b = f(y);
}
int f(int z) {
  return z;
}
```

(a)



(b)

Figure 3.8: A small program with its interprocedural control-flow graph with $\Phi(2 \to 7) = 9 \to 3$ and $\Phi(4 \to 7) = 9 \to 5$

program. However, another path is $\pi_2 : 1 \to 2 \to 7 \to 8 \to 9 \to 5$. This path does not correspond to any execution because it does not respect the semantics of procedure calls: When a procedure is called, the site from which the call is performed is pushed to the *call stack*. Once the procedure is finished, the call is popped off the call stack and execution continues from that call site. The path π_2 obviously does not respect this semantics: No execution that enters `f` through $2 \to 7$ leaves it through $9 \to 5$.

Relating calls and returns to one another is the main task of the correspondence function Φ. In the example, Φ is given by $\Phi(2 \to 7) = 9 \to 3$ and $\Phi(4 \to 7) = 9 \to 5$.

This leads to the notion of *interprocedurally valid paths*. Intuitively, an interprocedurally valid path is a path that respects the semantics of procedure calls. Validness can be defined with the help of Φ: We say that a path π is valid if $\Phi(e_{call}) = e_{ret}$ for all pairs (e_{call}, e_{ret}) on path such that e_{ret} is the return that finishes the procedure call performed by e_{call}.

Consider the path π_1 in the example. The return edge $9 \to 3$ finishes the call that is started by $2 \to 7$ and the return edge $9 \to 5$ finishes the

call that is started by $4 \to 7$. According to the definition of Φ, we have $\Phi(2 \to 7) = 9 \to 3$ and $\Phi(4 \to 7) = 9 \to 5$, hence π_1 is valid.
By contrast, π_2 is not valid, since $9 \to 5$ finishes the call that is started by $2 \to 7$ but we have $\Phi(2 \to 7) \neq 9 \to 5$.
In chapter 5, I will consider valid paths in a more general context. In particular, I will make precise what I mean by "e_{ret} finishes the call that is started by e_{call}".

3.2.2.2.2 Two Approaches for tackling the Context Problem

From the previous considerations, it is clear that if we perform data-flow analysis on interprocedural control-flow graphs like we do on intraprocedural control-flow graphs, the result is overly imprecise: Even if *MFP* coincides with *MOP*, the result is still imprecise since *MOP* merges over too many paths. To increase precision, we can consider a version of *MOP* that ignores paths that are definitely invalid. To define that, we let $VP(n)$ be the set of valid paths that start in s_{main} and end in n. Now let $(L, F, G, \rho, init)$ be a data-flow instance.
Then we can define the *merge-over-all-valid-paths* $MOVP$ as

$$MOVP(n) \overset{def}{=} \bigsqcup_{\pi \in VP(n)} f_\pi(init) \tag{3.8}$$

Sharir and Pnueli [154] presented two approaches to compute *MOVP* and showed that these two approaches compute under certain assumptions the same solution. In the following two sub-paragraphs, I give a short summary on both of these approaches. I will also consider both approaches in chapter 6 and chapter 7 in a more general setting and in more detail.

Call-String Approach The idea of the call-string approach is to extend the constraint system that describes *MOP* by an additional stack component. Every time a call is encountered, this call is pushed onto the stack and each time a return is encountered, it can be checked whether it corresponds to the call at the top of the stack. Constraints are only generated for corresponding call-return pairs.
More formally, the constraint system describes a function

$$A : N \times S \to L$$

where $S = E^{\star}_{call}$ is the set of all call stacks. The intraprocedural constraints naturally extend the constraint system given for intraprocedural data-flow analysis. Constraints for call and return edges not only apply the transfer functions but also manipulate the call stack according to procedure calling semantics, using the usual stack operations *push*, *pop* and *top*, with the properties

$$\begin{aligned} push(e,\sigma) &= e \cdot \sigma \\ pop(e \cdot \sigma) &= \sigma \\ top(e \cdot \sigma) &= e \end{aligned}$$

The empty stack is denoted by ϵ. The full constraint system C_{Stack} looks as follows:

Constraint System 3.2.

$$\begin{aligned} & & A(s,\epsilon) &\geq init \\ m \xrightarrow{e} n \wedge e \in E_{intra} &\implies & A(n,\sigma) &\geq f_e(A(m,\sigma)) \\ m \xrightarrow{e} n \wedge e \in E_{call} &\implies & A(n, push(e,\sigma)) &\geq f_e(A(m,\sigma)) \\ \begin{array}{r} m \xrightarrow{e} n \wedge e \in E_{ret} \wedge \sigma \neq \epsilon \\ \wedge \Phi(top(\sigma)) = e \end{array} &\implies & A(n, pop(\sigma)) &\geq f_e(A(m,\sigma)) \end{aligned}$$

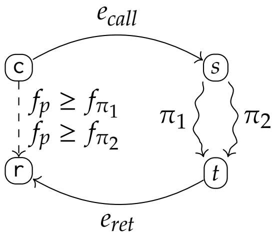
Note the additional precondition for the return constraints: No constraint is generated if $\Phi(top(\sigma)) \neq e$. This ensures that the resulting function A does not mix up calling contexts.

Like in the intraprocedural case, the above defined constraint system C_{Stack} has a least solution A_0.

To ensure comparability with $MOVP$, we define

$$\tilde{A_0}(n) \overset{def}{=} \bigsqcup_{\sigma \in S} A_0(n,\sigma).$$

Then it turns out that $\tilde{A_0} \geq MOVP$ and that $\tilde{A_0} = MOVP$ under some additional assumptions. However, there is one hitch: Unlike the intraprocedural constraint system, Constraint System 3.2 cannot be computed by the usual method, particularly if the program contains recursive calls.



(a)

$$\begin{aligned}
&(1) \quad A(r) \geq f_{e_{ret}}(f_p(f_{e_{call}}(A(c))))\\
&(2) \quad A(s) \geq f_{e_{call}}(A(c))
\end{aligned}$$

(b)

Figure 3.9: Sketch of the idea of the functional approach

The reason simply is that Constraint System 3.2 is not guaranteed to be finite.

The usual solution is to restrict the depth of the stacks. So instead of computing a function in $N \times E^{\star}_{call} \to L$, we compute a function in $N \times E^{\leq k}_{call} \to L$ and an adjusted *push* function

$$push_k(e, \sigma) = (e \cdot \sigma)^{\leq k}$$

that discards the lowermost item on the stack σ if σ already has k elements. By using $push_k$ instead of *push*, Constraint System 3.2 is always finite, its least solution $A_0^{(k)}$ can be computed using the usual method and still has the property $\tilde{A}_0 \geq MOVP$. However, note that discarding parts of the stack does not yield a fully context-sensitive analysis.

Functional Approach The *functional approach* uses the following idea, which is illustrated in Figure 3.9: Suppose that we have for each procedure p a transfer function f_p that faithfully describes a complete traversal of a procedure p's control-flow graph. Then we can obtain a context-sensitive constraint system as follows: For normal intra-procedural edges, we use the usual constraints. For each call site c of p we can use f_p to describe the effect of a call of f at c with a constraint like (1) in Figure 3.9b.

Additionally, a constraint like (2) ensures that the data-flow information is propagated from each call site to the entry of p.

Now, because of constraint (1), no constraint of the form

$$A(r) \geq f_{e_{ret}}(A(t))$$

is needed to propagate data-flow information back to the return site r. Such a constraint would introduce the context problem because the data-flow information at t subsumes all paths to t. This includes in particular the paths which come from call sites other than c. Information along these paths is not supposed to be propagated to r. Constraint (1) avoids this problem by propagating the information at c through the whole procedure p.

It remains to solve the sub-problem of providing the f_p. The idea is to describe them by a monotone constraint system, just like the final solution of the overall data-flow analysis. Note however that the f_p do not represent single data-flow information but describe how data-flow information is transformed. This means that the f_p do not live in L but in F.

As I already mentioned above, any f_p should faithfully describe a complete traversal of p. More formally, this means that the f_p incorporate the effects of traversing a certain class of paths from p's entry to its p's exit, the so-called *same-level paths*. A same-level path is a path that leaves every called procedure at the right call site and, additionally, ends in the same procedure in which it started. Hence a same-level path from p's entry to p's exit can be considered a complete traversal of p, as it may also occur in a real execution.

The sets $SL(s_p, t)$ of same-level paths from s_p to $t \in N_p$ are defined inductively for procedure entries s_p and nodes $t \in N_p$ of the same procedure. In order to avoid that call sites are mixed up, only corresponding call and return edges can be appended.

$$\epsilon \in SL(s_p, s_p) \tag{3.9}$$

$$\pi \in SL(s_p, t) \wedge t \overset{e_{intra}}{\to} e_{p'} \implies \pi \cdot e_{intra} \in SL(s_p, t') \tag{3.10}$$

$$\pi \in SL(s_p, t) \wedge t \overset{e_{call}}{\to} s_{p'} \wedge \pi' \in SL(s_{p'}, e_{p'}) \wedge e_{p'} \overset{\Phi(e_{call})}{\to} t'' \implies \pi \cdot e_{call} \cdot \pi' \cdot \Phi(e_{call}) \in SL(s_p, t'') \tag{3.11}$$

Using the same-level paths, we can now specify what we expect of the f_p: The f_p shall incorporate the effects of traversing any same-level path:

$$f_p \geq \bigsqcup_{\pi \in SL(s_p, e_p)} f_\pi$$

The constraint system for the f_p can defined along the same-level paths as follows:

Constraint System 3.3.

$$X(s_p, s_p) \geq id \tag{3.12}$$

$$t \xrightarrow{e} t' \wedge e \in E_{intra} \implies X(s_p, t') \geq f_e \circ X(s_p, t) \tag{3.13}$$

$$\begin{array}{l} t \overset{e_{call}}{\rightarrow} s_{p'} \\ \wedge e_{call} \in E_{call} \\ \wedge e_{p'} \overset{\Phi(e_{call})}{\rightarrow} t' \end{array} \implies X(s_p, t') \geq f_{\Phi(e_{call})} \circ X(s'_p, e_{p'}) \circ f_{e_{call}} \circ X(s_p, t) \tag{3.14}$$

Both Constraint System 3.3 and the one sketched in Figure 3.9b are finite, even in the presence of recursion. This means that, if the complete lattice $N \times N \rightarrow F$ satisfies the ascending chain condition, it can be solved by Algorithm 3. Note however that this additional condition restricts the practical applicability of the functional approach in comparison to the restricted call-string approach: It can only be applied to data-flow frameworks in which not only the complete lattice L but also the function space F satisfies the ascending chain condition. However, if the functional approach is applicable, it obtains a fully context-sensitive solution.

3.3 Slicing on Program Dependence Graphs

In this section, I introduce *slicing*, another important static program analysis technique, and *program dependence graphs*, a data-structure that reduces slicing to graph reachability.

This section is organized as follows: First, I explain the general idea of slicing in subsection 3.3.1. In subsection 3.3.2, I introduce program dependence graphs, first for the intraprocedural case and subsequently for the interprocedural case. Finally, in subsection 3.3.3 I show how program slicing can be performed on program dependence graphs. Specifically, I consider an approach to obtain context-sensitive slices on interprocedural program dependence graphs.

3.3.1 Slicing

Program slicing was introduced by Weiser [165] as a technique for focusing on specific parts of a program. For a given program $\mathcal{P}$, a slice is defined with respect to a *slicing criterion* which consists of a program location l and a variable x. Given such a slicing criterion $c = (x, l)$, a *valid* slice with respect to c is any sub-program $\mathcal{P}'$ of $\mathcal{P}$ which produces the same behaviour with respect to c as $\mathcal{P}$. This means that if $\mathcal{P}$ and $\mathcal{P}'$ are started in the same state and both terminate, then $\mathcal{P}$ and $\mathcal{P}'$ cannot be distinguished by just looking at the values of x at each respective execution of location l. It is desirable to have an automatic procedure which can always find slices of the smallest possible size. Due to decidability reasons, such a procedure cannot exist, but Weiser [165] describes a procedure to obtain a valid program slice that is fairly small. The idea is roughly to traverse the program's control-flow graph backwards from the given slicing criterion (x, l) and include in the slice every statement which may have an influence on the value of x in l. Essentially, Weiser's procedure transitively follows the data and control dependencies ending in x backwards.

3.3.2 Program Dependence Graphs

Program Dependence Graphs [58] (PDGs) are another program representation used in program analysis. Roughly, PDGs model the dependencies between the statements and expressions of a program.
Ferrante, Ottenstein and Warren [58] introduced the program dependence graph as a program representation which makes control dependencies and data dependencies explicit. On this representation, program slicing can be expressed as a graph traversal. A slice thus obtained is indeed valid [140].

3.3.2.1 Intraprocedural Program Dependence Graphs

For intraprocedural programs, there are two main kinds of dependencies: *data dependencies* and *control dependencies*. In the following, I will briefly explain data and control dependencies. After that, I will show how Program Dependence Graphs are extended for programs with multiple procedures.

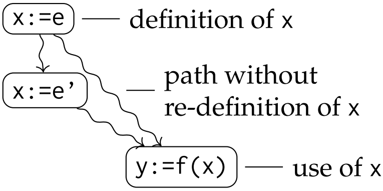


Figure 3.10: Visualization of data dependencies

3.3.2.1.1 Data Dependencies

Data dependencies capture the flow of data inside a program. A statement *defines* a variable if it writes a value to it and *uses* a variable if it reads the current value of that variable and then uses this value, for example to define other variables or to evaluate a branching condition.

For a statement (or CFG node, respectively) s I denote with $def(s)$ the set of variables defined by s and with $use(s)$ the set of variables used by s.

Basically, a statement or expression s_2 is data-dependent on statement s_1 if there is a variable x such that (1) s_1 defines x, (2) s_2 uses x and (3) there is a control-flow path between s_1 and s_2 that does not define x. Such a situation is illustrated in Figure 3.10. Definition 3.3 gives a formal definition.

Definition 3.3 (data dependencies)**.** *Let $G = (N, E, s, e)$ be a control-flow graph. $n \in N$ is data-dependent on $m \in N$, written $m \to_{dd} n$, if there is $x \in def(m) \cap use(n)$ and there is a path $\pi \in Paths_G(m, n)$ such that $\forall 1 \leq i < |\pi| - 1.\ x \notin def(\pi^i)$.*

Figure 3.11 shows the data dependency graph of the program from Figure 3.3.

For example, there is a data dependency from line 5 to line 7 because line 5 defines the variable a, line 7 uses a and a is not overwritten between line 5 and line 7. In contrast, line 5 and line 12 are not connected by a data dependency: Though line 12 uses a to define c, it definitely does not use the value of a from line 5, since a is overwritten in line 10.

Data dependencies can be computed using the reaching definitions analysis described earlier.

3.3.2.1.2 Control Dependencies

The other kind of dependencies in a program dependence graph are *control dependencies*. The intuition behind control dependencies is illustrated in Figure 3.12. Control dependencies capture that a node $m \in N$ is the “latest” node that “decides” whether (or how often) a node n is executed or not. This means that if program execution traverses m, then it can either proceed to a branch from which m can be bypassed or to a branch from which the execution of n is inevitable. Throughout this thesis, I assume the classical definition by Ferrante et al. [58], which I present here for reference. Ferrante et al. also propose an efficient algorithm for computing control dependency graphs that constructs the post-dominator tree using a fast algorithm presented by Lengauer and Tarjan [119].

Definition 3.4 ([58], Definitions 2 and 3)**.** *Let $G = (N, E, s, e)$ be a control-flow graph and $m, n \in N$.*

1. Node n post-dominates m if

$$\forall \pi \in Paths_G(m, e).\ n \in nodes(\pi)$$

2. Node n is control-dependent on m, written $m \rightarrow_{cd} n$, if

a) there is a path $\pi \in Paths_G$ from m to n such that n post-dominates every node in π (except for m and n)

b) n does not post-dominate m.

Figure 3.13 shows the control dependence graph of the example from Figure 3.3.

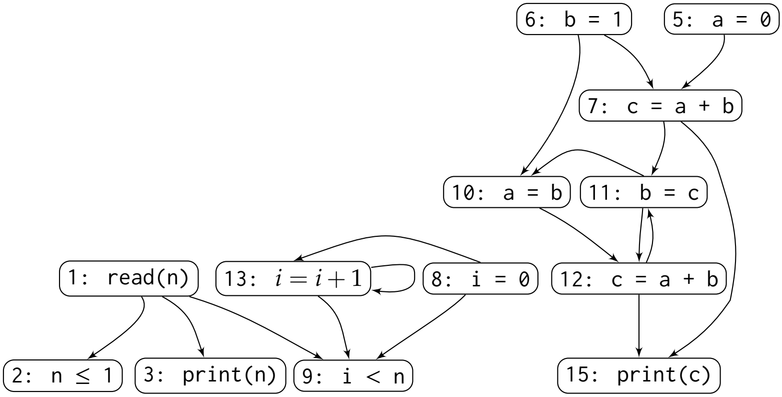


Figure 3.11: Data dependency graph for the example from Figure 3.3

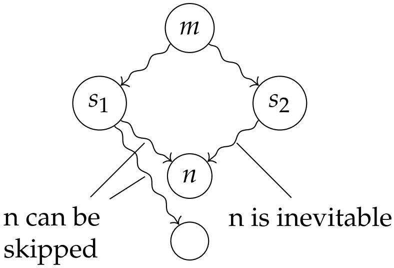


Figure 3.12: Illustration of control-dependencies

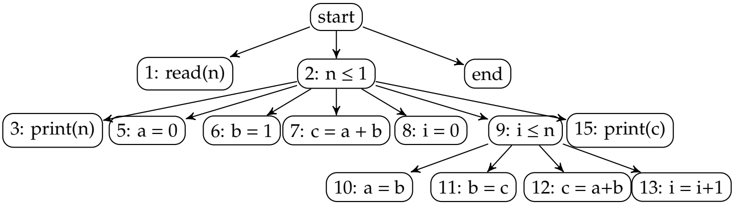


Figure 3.13: Control dependence graph of the program in Figure 3.3 – note that for well-formedness reasons an additional synthetic control-flow edge between the entry node and the exit node is assumed

3.3.2.2 Interprocedural Program Dependence Graphs

In the following, I describe briefly how Program Dependence Graphs look like for programs with multiple procedures. The standard approach has been described by Horwitz et al. [93].

In this approach, interprocedural PDGs are constructed from intraprocedural PDGs in a similar way as interprocedural control-flow graphs are constructed from intraprocedural control-flow graphs. An interprocedural PDG consists of a PDG for every procedure. These *procedure dependence graphs* are enriched with additional nodes and edges which model the call itself and the passing of parameters from caller to callee and the passing of return values from callee to caller. This modelling assumes call by value. First of all, a *call dependence* connects the call node at the call site and the entry method of the callee. This is a special control dependence

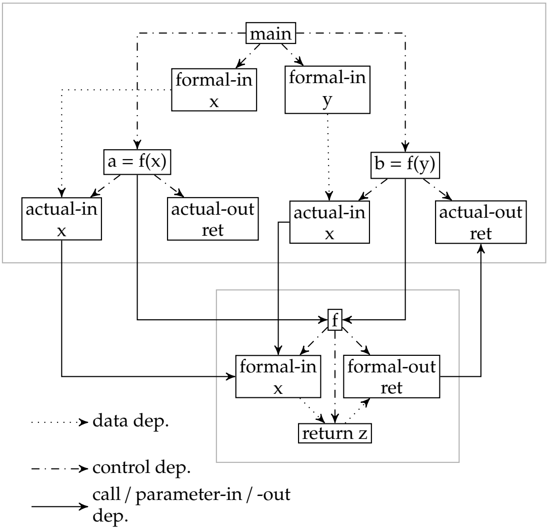


Figure 3.14: The interprocedural PDG of the example in Figure 3.8a

that captures the intuition that the call node “decides” whether the callee is called or not.

Moreover, for every parameter of a procedure *p*, there is a *formal-in* parameter node, and for its return value, there is a *formal-out* node. At each call site of *p*, there is an *actual-in* node for each of *p*’s parameters and an *actual-out* node for *p*’s return value. Formal and actual parameter nodes are connected via *parameter-in* and *parameter-out* edges, which model passing of parameters from caller to callee and of the return values from callee to caller. A procedure call can be thought of to be preceded by a series of assignment statements which assign the actual parameter values to special variables which only the callee has access to. After the procedure has finished, it copies its return value to a special variable which only the caller has access to. In this sense, parameter-in and parameter-out edges can be seen as special interprocedural data dependencies. Parameter passing is then modeled using a *parameter-in* edge which connects the actual-in parameter node at the call site with the formal-in node in the callee. Additional *parameter-out* edges model how data flows from a formal-out parameter node of a procedure to its counterpart in the caller. Consider for example the interprocedural PDG in Figure 3.14: Procedure `f` has one formal-in parameter node for its parameter z and one formal-out parameter node for its return value. Both are connected to their counterparts at each of the two call sites of f.

In the following, I consider call dependencies and parameter-in edges as *call edges* and refer to all call edges as E_{call}. Analogously, I consider parameter-out edges as return edges and use E_{ret} to refer to the set of parameter-out edges.

3.3.3 PDG-based Slicing

As I already mentioned in the beginning, program dependence graphs make explicit the dependencies that are traversed implicitly during Weiser's slicing procedure [165]. Hence, PDGs reduce the slicing problem to mere graph traversal.

3.3.3.1 PDG-based Intraprocedural Slicing

Algorithm 4: A simple intraprocedural backward slicer – upon termination, we have $W = BS_{intra}(s)$

```
Input: a PDG $G = (N, E)$ with intraprocedural edges $E_{intra}$, slicing criterion $s \in N$
Result: intraprocedural slice $BS_{intra}(s)$
$W \leftarrow \{s\}$
while $W \neq \emptyset$ do
    $n \leftarrow remove(W)$
    foreach $m \rightarrow n \in E_{intra}$ do
        $W \leftarrow W \cup \{m\}$
return $W$
```

For single procedures, PDG-based slicing works as follows: Given a PDG $G = (N, E)$ and some node n, the *backwards-slice* is the set $BS(n) \subseteq N$ of all nodes that reach n in the PDG:

$$BS(n) \overset{def}{=} \{s \in N \mid s \rightarrow_G^* n\}$$

Analogously, the *forward-slice* is the set $FS(n) \subseteq N$ of all nodes that is reachable by n in the PDG:

$$FS(n) \overset{def}{=} \{s \in N \mid n \to_G^* s\}$$

BS and FS are related by the property

$$s \in BS(n) \iff n \in FS(s)$$

and hence are dual to each other. Algorithm 4 shows a simple algorithm that computes an intraprocedural backward slice of $s \in N$. It is very easy to modify Algorithm 4 such that it computes an intraprocedural forward slice.

3.3.3.2 PDG-based Interprocedural Slicing

Interprocedural slicing operates on an interprocedural program dependence graph, just like intraprocedural slicing operates on an intraprocedural program dependence graph.

3.3.3.2.1 Context Problem Similar to data-flow analysis, interprocedural slicing faces the problem that not all paths in an interprocedural program dependence graph represent a valid chain of dependencies. In effect, if BS and FS are not adapted to the interprocedural case, they yield context-insensitive slices, which are usually too big.
For example, Figure 3.15 shows a simple backwards slice of the return value of the second call of `f` from Figure 3.14. This slice also contains the actual parameter of the first call of `f`.
However, the only path from the actual parameter of the first call of `f` to the return value of the second call of `f` is interprocedurally invalid – just like the path that enters `f` through the first call site and leaves it through the second call site.

3.3.3.2.2 Context-Sensitive Interprocedural Slicing The context problem for interprocedural slicing can be tackled in similar ways as the context problem for interprocedural data-flow analysis. Agrawal and Guo [9] consider a call-string-based approach that supports call-strings of unlimited length. However, Krinke [110] observes that this approach is indeed incorrect. He proposes a fixed version and also considers call-strings with limited depths.

Horwitz, Reps and Binkley [92, 93] propose an approach to interprocedural slicing that resembles the functional approach for interprocedural data-flow analysis and thus avoids the context problem. First, in a pre-processing step the program dependence graph is extended with additional *summary edges*. A summary edge is added between an actual-in node and an actual-out node of the same call site if a corresponding formal-in/formal-out pair is connected by a chain of intraprocedural edges or summary edges. The resulting graph is called *System Dependence Graph*.

In comparison to the original presentation [92, 93], the runtime performance of the pre-processing step can be improved by introducing additional book-keeping and trading space for speed, as shown by Reps, Horwitz, Mooly and Rosay [137]. My presentation in this thesis concentrates on the improved version.

The actual slicing can then be performed on the System Dependence Graph using an algorithm that operates in two phases. Each phase basically consists of a simple graph reachability approach that skips either parameter-in or parameter-out edges. Both phases use summary edges to traverse call sites. This way, the approach avoids traversing call and return

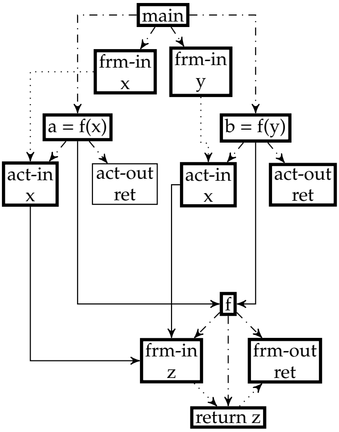


Figure 3.15: Context-insensitive backwards slice of the actual-out ret parameter of the second call of `f`

edges individually – the root cause of the context problem, as I mentioned earlier.

Similar to the functional approach to interprocedural data-flow analysis described in section 3.2.2.2.2, summary edges describe a complete traversal of the called procedure and the System Dependence Graph is nothing more than the integration of these additional helper transfer functions into the Program Dependence Graph.

More generally, the PDG edges themselves also can be viewed as transfer functions. The information they propagate is mere reachability. In this sense, the two-phase slicer can be understood as the solution algorithm for a data-flow analysis instance with very simple transfer functions. I will discuss the similarities between and differences of data-flow analysis and slicing in more detail in subsection 3.3.4.

In the following, I briefly describe the summary edge algorithm and the two-phase slicer.

Summary Edges The summary edge computation algorithm proposed by Reps et al. [92, 93] is shown in Algorithm 5. The rough idea is to compute all node pairs for which there is a *same-level path* in the given program dependence graph. To define same-level paths for program dependence graphs, some modifications are necessary: For one, we need to use parameter-in edges instead of call edges and parameter-out edges instead of return edges. Moreover, the correspondence function needs to be a correspondence *relation*, since in the presence of multiple parameters, there may be more than one parameter-out edge that corresponds to a given parameter-in edge. In chapter 5, I will present a definition that applies to both control-flow graphs and program dependence graphs.

The algorithm maintains two sets of node pairs (v, w) where w is a formal-out node and v is an arbitrary node of the same procedure. If $(v, w) \in$ *PathEdge*, then there is a same-level path between v and w. If $(v, w) \in W$, then there is a same-level path between v and w and (v, w) has been discovered for the first time.

Initially, for every formal-out node w, the pair (w, w) is contained in both W and *PathEdge*. This is because the empty path is a same-level path.

In the main iteration, the algorithm removes a pair (v, w) from W and reacts to two relevant cases for (v, w):

Algorithm 5: Summary Edge Algorithm proposed by Reps et al.– compare [137, Figure 5]

```
Input: a PDG $G$
Result: summary egdes in $G$
$PathEdge \leftarrow \emptyset$
$SummaryEdge \leftarrow \emptyset$
$W \leftarrow \emptyset$
foreach $w \in FormalOuts(G)$ do
    $PathEdge \leftarrow PathEdge \cup \{(w,w)\}$
    $W \leftarrow W \cup \{(w,w)\}$
while $W \neq \emptyset$ do
    $(v,w) \leftarrow remove(W)$
    if $v$ is a formal-in node then
        foreach $x \overset{param-in}{\rightarrow} v,\ w \overset{param-out}{\rightarrow} y$ do
            if $x$ and $y$ belong to the same call-site then
                $SummaryEdge \leftarrow SummaryEdge \cup \{x \rightarrow y\}$
                foreach $a$ such that $(y,a) \in PathEdge$ do
                    if $x \rightarrow a \notin PathEdge$ then
                        $PathEdge \leftarrow PathEdge \cup \{(x,a)\}$
                        $W \leftarrow W \cup \{(x,a)\}$
    else
        foreach $x \rightarrow v \in E_{intra}$ do
            if $(x,w) \notin PathEdge$ then
                $PathEdge \leftarrow PathEdge \cup \{(x,w)\}$
                $W \leftarrow W \cup \{(x,w)\}$
        if $v$ is an actual-out node then
            foreach $(x,v) \in SummaryEdge$ do
                if $(x,w) \notin PathEdge$ then
                    $PathEdge \leftarrow PathEdge \cup \{(x,w)\}$
                    $W \leftarrow W \cup \{(x,w)\}$
return $SummaryEdge$
```

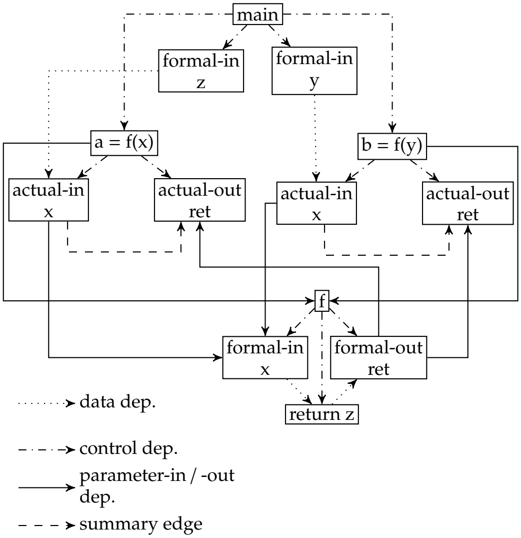


Figure 3.16: The interprocedural PDG of the example in Figure 3.8a with summary edges

1. If v is a formal-in node, then a same-level path has been completed. The algorithm then adds a summary edge between every corresponding actual-in/actual-out pair (x, y). In this context, this means that there is a parameter-in edge from x to v, a parameter-out edge from w to y and that x and y belong to the same call site. Now that there is an additional edge between x and y, it may be the case that some pair (y, a) can be extended to (x, a). Because of this, for all already discovered pairs (y, a), (x, a) is added to W if it has not been discovered before. This ensures that the call site "notices" that propagation can be continued.

2. If v is not a formal-in node, then every incoming intraprocedural edge $x \to v$ of v is processed; since there is a same-level path between v and w and there is an incoming edge $x \to v$, there is a same-level path between x and w. Hence, (x, w) is added to *PathEdge* and W if it was encountered for the first time. If v is an actual-out node, then the algorithm additionally processes the already discovered summary edges.

Applied to Figure 3.14, Algorithm 5 starts at the formal-out return node of `f` and traverses `f`'s PDG backwards until it arrives at `f`'s formal-in node for

z. Now, a complete same-level path has been discovered and Algorithm 5 inserts a summary edge between the actual-in node for x and the actual-out return node at each of the two call sites of f. This results in the system dependence graph shown in Figure 3.16.

The Two-Phase Slicer The idea of the two-phase slicer, which can be seen in Algorithm 6, bears some similarity to the idea of the functional approach: With summary edges, procedure calls can be skipped.

Algorithm 6: Backwards two-phase slicer proposed by Horwitz et al. [92]

```
Input: a PDG $G = (N, E)$; $E$ consists of the intraprocedural edges
       $E_{intra}$ and $E_{call}$, $E_{ret}$ as described on page 67; $E_{sum}$ is the set
       of summary edges as computed by Algorithm 5; slicing
       criterion $s$
Result: context-sensitive backwards slice of $s$
$S \leftarrow \emptyset$
$W_1 = \{s\}$ // phase 1: only ascend to callers
while $W_1 \neq \emptyset$ do
    $n \leftarrow remove(W_1)$
    $S \leftarrow S \cup \{n\}$
    foreach $m \xrightarrow{e} n \in E$ do
        if $e \in E_{intra} \cup E_{sum} \cup E_{call}$ then
            $W_1 \leftarrow W_1 \cup \{m\}$
        else
            // $e \in E_{ret}$ and $m$ is an exit or formal-out node
            $W_2 \leftarrow W_2 \cup \{m\}$
// phase 2: only descend to callees
while $W_2 \neq \emptyset$ do
    $n \leftarrow remove(W2)$
    $S \leftarrow S \cup \{n\}$
    foreach $m \rightarrow n \in E_{intra} \cup E_{sum} \cup E_{ret}$ do
        $W_2 \leftarrow W_2 \cup \{m\}$
return $S$
```

This way, simple graph reachability can be used to obtain a slice that respects calling contexts. Note, however, that unlike with data-flow analysis, we are not interested in paths from the entry of the `main` procedure but actually want to start at an arbitrary place in the given graph. But this means that it does not suffice to descend into called procedures. It is also necessary to ascend to calling procedures. For this purpose, the slicer proposed by Horwitz et al. consists of two phases: The first phase only ascends to calling procedures and the second phase only descends into called procedures.

Figure 3.17 shows how Algorithm 6 is applied to Figure 3.14. The slicing criterion is the actual-out return node of the second call of `f`. In Figure 3.17a, we see the state after the completion of the first phase of Algorithm 6: At this point, the slice contains all nodes with bold frame. Moreover, the actual-out return node of the second call of `f` is contained in $W2$ so that phase 2 will start with that node.

Phase 2 then descends into `f`. The result of this can be seen in Figure 3.17b. It can easily be seen that it is more precise than a context-insensitive slice, since unlike the slice in Figure 3.15, it does not contain the actual parameter of the first call of `f`.

3.3.4 Relation of PDG-based Slicing and Data-Flow Analysis

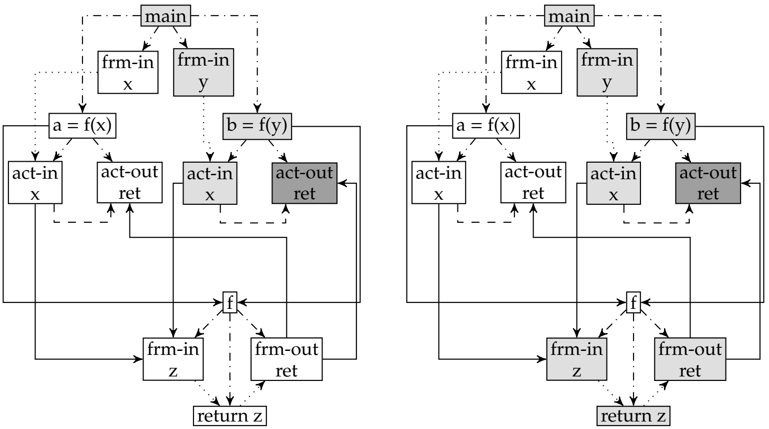
In the following, I investigate the relation between PDG-based slicing and data-flow analysis. My observations are

1. slicing can be cast as a very simple data-flow problem,

2. this data-flow problem can be solved with basically the same techniques, however, different assumptions have to be made, and

3. particularly, summary-based two-phase slicing can be seen as an application of a modified functional approach to interprocedural data-flow analysis

After having identified slicing as a special case of data-flow analysis, we can generalize it and obtain arbitrary data-flow analyses on program dependence graphs. These analyses can be solved for instance with the functional approach. This will be the subject of chapters 5, 6 and 7.



(a) after phase 1 **(b)** after phase 2

Figure 3.17: Applying the two-phase slicer to the example in Figure 3.14; the slicing criterion (with dark gray background) is the actual-out return node of the second call of `f`; nodes in slice after the respective phase have light gray background

In the following presentation, I fix a node $s \in N$ and consider the forward slice $FS(s)$ of s. Note that I knowingly ignore the fact that Algorithm 5 and Algorithm 6 traverse the graph *backwards*. In contrast, I only consider forward traversals and, in particular, I pretend that Algorithm 5 and Algorithm 6 traverse the graph forward. I showed these algorithms in their original version that was motivated by applications such as debugging, where backward slices are natural. However, in general the propagation direction does not matter and can easily be changed.

3.3.4.1 PDG-based Slicing as a Data-Flow Problem

As I already mentioned earlier, slicing can be described as a kind of reachability analysis. As such, it can easily be cast as a data-flow problem. Mainly, we will have to adapt the way in which the constraint systems are generated from the given graph.
Let $L = \{\bot, \top\}$ with $\bot < \top$ and let $F = \{\lambda x.\bot, \lambda x.x\}$ with the usual partial order. Then (L, F) is a data-flow framework.

An instance of this framework is given by the PDG G, the initial information $init = \top$ and $\rho(e) = id$.
For simplicity, I consider the special case *reachability from a given node*[8] s.

Intraprocedural case For $\pi \in Paths_G(s,t)$, we have $f_\pi = id$. Hence, with

$$F_{intra}(t) = \bigsqcup_{\pi \in Paths_G(s,t)} f_\pi(init),$$

we have $F_{intra}(t) = \top$ if and only if $t \in FS_{intra}(s)$ (where FS_{intra} is the intra-procedural forward slice of s). The function F_{intra} is similar to *MOP* (see (3.1) on page 50), but merges over a different path set. I will examine the difference later and ignore it for now.
F_{intra} can be described by a monotone constraint system that is very similar to the one used in intra-procedural data-flow analysis:

Constraint System 3.4.

$$\begin{aligned} &X(s) \geq init \\ &X(t) \geq X(t') \mathit{\ for\ } t' \xrightarrow{e} t. \end{aligned}$$

As can easily be seen, this constraint system actually precisely characterizes F_{intra}: Its least solution $\underline{X}$ can be used to extract the intra-procedural forward-slice FS_{intra} of s:

$$FS_{intra} = \{n \in N \mid \underline{X}(n) = \top\}.$$

Interprocedural case In the interprocedural case, we want to compute

$$F_{inter}(t) = \bigsqcup_{\pi \in VP(s,t)} f_\pi(init).$$

This is similar to *MOVP* (see (3.8) on page 57). However, there are two differences:

[8] In later chapters, I consider data-flow problems whose *MOP* functions take two parameters.

$$\begin{tikzcd}
 & & n_3 \arrow[r, squiggly, "\pi_3"] & n_4 \arrow[d, "\text{call}"] & \\
 & n_1 \arrow[r, squiggly, "\pi_2"] & n_2 \arrow[u, "\text{return}"'] & n_5 \arrow[r, squiggly, "\pi_4"] & n_6 \\
s \arrow[r, squiggly, "\pi_1"] & n_0 \arrow[u, "\text{return}"'] & & &
\end{tikzcd}$$

Figure 3.18: Illustration of a valid PDG path: The π_i are same-level paths. All n_i are reachable from s by a valid path.

1. F_{inter} does not merge over all valid paths starting in s_{main} and ending in t but over those that start in s (similar to F_{intra}).

2. F_{inter} refers to a different notion of *validness*, which I describe in the following.

For interprocedural control-flow graphs, two usual assumptions are

1. that every procedure p has an entry node s_p and each of p's node is same-level reachable from s_p, and

2. that there is a main procedure *main* and for each procedure p, s_p is reachable from s_{main} by a *descending* path, that is a path that consists of a series of same-level paths interspersed with call edges.

In interprocedural data-flow analysis, one usually is interested in proper executions that indeed start at s_{main} and proceed until a given program point is reached. This is why $MOVP(t)$ merges over all valid paths from s_{main} to t. For slicing, this is different: As we are interested in reachability from s, we only want to consider the valid paths that start in s and not necessarily in s_{main}. Moreover, since s may lie anywhere in the program, the notion of validness employed here can not only consider descending paths, like in interprocedural control-flow graphs, but also has to include paths that have an *ascending* prefix. Analogously to a descending path, an ascending path can be imagined as a series of same-level paths interrupted by return edges. A valid path is an ascending path followed by a descending path. An illustration is given in Figure 3.18.

Now we want to describe F_{inter} by a series of monotone constraint systems. We use the functional approach, since it is applicable to the reachability framework and we want to maximize precision. Recall that the idea of the functional approach is to avoid traversing call and return edges at the same time by first solving a helper system that describes the effect of completely traversing procedures from entry to exit.

Helper System for Same-Level Reachability The helper system is similar to the helper system for interprocedural data-flow analysis on page 61. However, for PDGs we have to make a few modifications: For one, procedures may have multiple entries, namely the actual procedure entry and the formal-in parameter nodes, and also multiple exits, namely the actual procedure exit and the formal-out parameter nodes. Correspondingly, call sites may have multiple call nodes (the actual call node and the actual-in parameter nodes) and multiple return nodes (the actual return node and the actual-out parameter nodes). In particular, the correspondence relation Φ in this case relates actual-ins and actual-outs that belong to the same call site and therefore in general is not a function, but rather an arbitrary relation.
With these modifications, the helper constraint system that describes same-level reachability looks as follows:

Constraint System 3.5.

$$X(n,n) \geq id \tag{3.15}$$

$$t \xrightarrow{e} t' \wedge e \in E_{intra} \implies X(n,t') \geq X(n,t) \tag{3.16}$$

$$(e_{call}, e_{ret}) \in \Phi \wedge m \xrightarrow{e_{call}} n_0 \wedge n_1 \xrightarrow{e_{ret}} t \implies X(n,t) \geq X(n_0,n_1) \circ X(n,m). \tag{3.17}$$

With uniqueness assumptions of entries, exits, calls and returns in place, Constraint System 3.5 reduces to Constraint System 3.3.
The least solution of this system is a function $\underline{X}_{SL} : N \times N \to F$ with $\underline{X}(s,t) = id$ if and only if t is same-level reachable from s.
In the following, I show how to use $\underline{X}_{SL}$ to actually compute F_{inter} and hence $FS(s)$. The approach exploits a fundamental property about valid paths:
Every valid path $\pi \in VP(s,t)$ is either ascending or there is $n \in N_{call}$ so that π can be split up into $\pi_1 \cdot \pi_2$ such that π_1 is an ascending path from s to n and π_2 is a non-empty descending path from n to t.

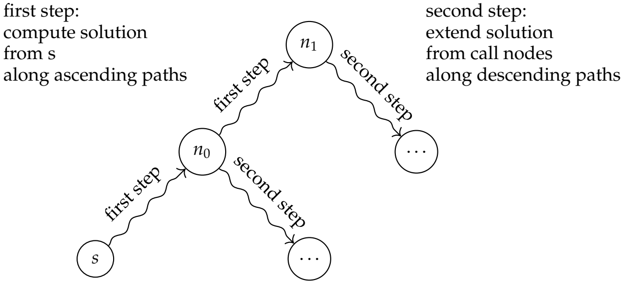


Figure 3.19: Illustration of how the computation of the reachability solution along valid paths works

I will also call the second kind of valid paths *non-ascending*.
As a consequence of the above mentioned property, F_{inter} can be obtained in two steps as illustrated by Figure 3.19. They can be described intuitively as follows:

1. The first step computes the solution along the ascending paths starting with s.

2. The second step starts at the nodes $n \in N_{call}$ that are reachable from s by an ascending path and extends the solution along descending paths.

Computing Reachability Along Ascending Paths This first step works like the intra-procedural case but additionally propagates the reachability information along return edges and uses $\underline{X}_{SL}$ for propagating from actual-ins to corresponding actual-outs.

Constraint System 3.6.

$$X_{ASC}(s) \geq init \tag{3.18}$$

$$e \in E_{intra} \cup E_{ret} \wedge t' \xrightarrow{e} t \implies X_{ASC}(t) \geq X_{ASC}(t') \tag{3.19}$$

$$(e_{call}, e_{ret}) \in \Phi \wedge m \overset{e_{call}}{\to} n_0 \wedge n_1 \overset{e_{ret}}{\to} t \implies X_{ASC}(t) \geq \underline{X}_{SL}(n_0, n_1) \circ X_{ASC}(m) \tag{3.20}$$

The least solution $\underline{X}_{ASC}$ has the property

$$\underline{X}_{ASC}(t) \neq \bot \iff s \text{ reaches } t \text{ using an ascending path} \tag{3.21}$$

Extending Reachability Along Descending Paths The second step starts at the call nodes that are reachable by ascending paths from s and extends the reachability solution along descending paths. Again, it works like the intra-procedural case but additionally propagates the reachability information along call edges and uses $\underline{X}_{SL}$ for propagating from actual-ins to corresponding actual-outs.

Constraint System 3.7.

$$\begin{aligned}
&(3.22) && a \in N_{call} \wedge a \xrightarrow{e} t \wedge e \in E_{call} \\
& && \implies X_{NASC}(t) \geq \underline{X}_{ASC}(a) \\
&(3.23) && e \in E_{intra} \cup E_{call} \wedge t' \xrightarrow{e} t \\
& && \implies X_{NASC}(t) \geq X_{NASC}(t') \\
&(3.24) && (e_{call}, e_{ret}) \in \Phi \wedge m \xrightarrow{e_{call}} n_0 \wedge n_1 \xrightarrow{e_{ret}} t \\
& && \implies X_{NASC}(t) \geq \underline{X}_{SL}(n_0, n_1) \circ X_{NASC}(m)
\end{aligned}$$

The least solution $\underline{X}_{NASC}$ has the property

$$\underline{X}_{NASC}(t) \neq \bot \iff s \text{ reaches } t \text{ using a non-ascending path} . \tag{3.25}$$

Putting the Steps Together With the fundamental property of valid paths and properties (3.21) and (3.25), we can characterize F_{inter} as $\underline{X}_{ASC} \sqcup \underline{X}_{NASC}$.

$$F_{inter} = \underline{X}_{ASC} \sqcup \underline{X}_{NASC} \tag{3.26}$$

3.3.4.2 Comparisons of the Algorithms

After having compared the specification side of data-flow analysis and slicing, I examine the algorithm side more closely.

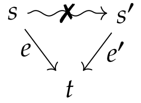
Roughly, one can say that each of the slicers described so far, namely Algorithm 4, Algorithm 5 and Algorithm 6, correspond to one or more of the constraint systems that I just showed, in the sense that they compute a representation of the least solution for significant parts of them.

In particular, the summary edges used by Algorithm 6 can be considered as procedural effect functions that are used to safely skip procedure calls, or, for PDGs, transitions from actual-in nodes to actual-out nodes. Conceptually, all pairs of actual-in and actual-out nodes at the same call site are connected by edges. The presence of a summary edge between such an actual-in node m and an actual-node out n encodes whether the transfer function for the edge between m and n is $\lambda x.x$ or $\lambda x.\bot$. In the former case, reachability is propagated, in the latter case propagation stops.

The constraint systems that I showed so far can also be solved with appropriate instantiations of Algorithm 3. In the following, I investigate the differences between the slicers and Algorithm 3.

First, when looking at the constraint systems, we see that they are *too large*: They contain a lot more constraints than the ones that are actually needed to compute their respective result. An example for this for the intraprocedural case can be seen in Figure 3.20: Suppose that the PDG contains a node s' that is not reachable from s but like s has an outgoing edge to t. Then both edges are incorporated in Constraint System 3.4 but only the edge from s to t is relevant. The other constraint systems have a similar problem.

The reason is that they cannot already incorporate reachability information as *their purpose is to characterize that very information*. An unmodified version of Algorithm 3 would solve the complete systems and in particular process large parts that are actually not relevant. The slicers however only explore the relevant parts of the respective constraint systems, namely the ones that are reachable from the initial constraints that do not have variables on the right-hand side.



$$X(t) \geq X(s)$$
$$X(t) \geq X(s')$$

Figure 3.20: Illustration for the way in which the constraint systems for slicing – for example Constraint System 3.4 – contain irrelevant constraints

A striking similarity between Algorithm 3 and the slicing algorithms is that they all use worklists to keep track of the items that have yet to be processed. However, they differ in what I want to call *workflow policy*. A workflow policy answers the following questions and thus determines how the algorithm operates on the worklist to finish its task:

1. Why is an item put on the worklist?
2. Which values are updated?
3. Which items are put on the worklist during processing?

Figure 3.21 illustrates the differences in the worklist policies of Algorithm 3 and the slicing algorithms.
As shown in Figure 3.21a, Algorithm 3 puts an item onto the worklist if one of its predecessors has changed in an earlier iteration. Then, when an item x is taken off the worklist and processed, its value is updated with the value of its predecessors. If the value has changed, all its successors are put on the worklist.
The slicers have a slightly different worklist policy, as illustrated in Figure 3.21b. There, an item is put on the worklist if its value has been changed in an earlier iteration but this has not been propagated yet. Then when an item x is taken off the worklist and processed, all its successors are updated with respect to the value of x. All successors whose value changes by this are put on the worklist.

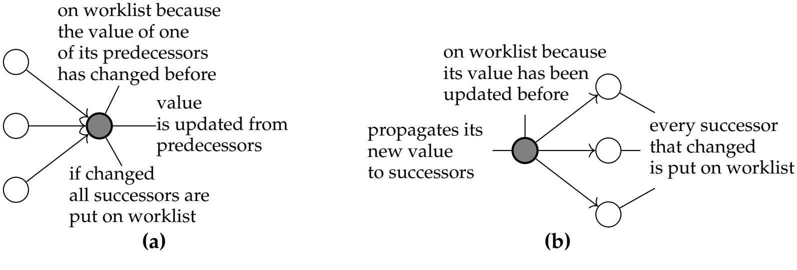


Figure 3.21: Illustration of the difference between (a) Algorithm 3 and (b) Algorithm 4 – the node highlighted in gray is assumed to have just been taken off the worklist and about to be processed

Another difference between Algorithm 3 and the slicers is that Algorithm 3 requires that all variables are processed at least once. For this, the algorithm initially puts all variables on the worklist. In contrast, the slicing algorithms initialize the worklist only with those variables that are defined by initial constraints.

In short, one can say that the slicers integrate a reachability analysis of the constraint system into their solution process. This may seem like a pointless observation: For one, reachability analysis is exactly what the slicers do. Secondly, data-flow analysis makes several reachability assumptions and therefore has no necessity to additionally perform reachability analysis. However, this observation is actually helpful for generalizing slicing to arbitrary data-flow analysis: Since slicing does not make the reachability assumptions of data-flow analysis, its generalization also does not make them. *Therefore, in order to perform data-flow analysis on program dependence graphs that includes slicing as a special case, we need solution algorithms that integrate slicing into its solution processing!* I will present such algorithms in chapter 7, along with the theory that explains why they work and what exactly they do. These algorithms can be used to perform data-flow analysis not only on program dependence graphs but also on control-flow graphs that do not satisfy reachability assumptions such as the ones mentioned on page 77.

3.4 JOANA: PDG-based Information Flow Control for JAVA

In this section, I describe the JOANA framework in more detail.

JOANA heavily uses the T.J. Watson Libraries for Analysis (WALA), a program analysis framework for JAVA bytecode [141].

Using WALA, JOANA applies a wide variety of program analysis techniques in order to construct program dependence graphs and to verify various non-interference-like properties of a given JAVA application. Apart from the techniques I have shown so far in section 3.2 and section 3.3, JOANA can also handle modern programming language features.

The goal of this section is to give the reader a rough understanding of how JOANA works so that they are equipped to follow the details of chapter 4.

For a more thorough description of the inner workings of Joana, I refer the reader to earlier publications [78, 65, 86].

In subsection 3.4.1, I describe the connection between information flow control and slicing that is exploited by Joana. After that, I go into some of the features of Java and describe some of the techniques that Joana applies in order to treat them appropriately. Specifically, subsection 3.4.2 considers dynamic dispatch, subsection 3.4.3 looks at the challenges posed by exceptions and, lastly subsection 3.4.4 shows how objects are represented and handled by Joana.

Note that I will not discuss concurrency in this chapter but in chapter 4, particularly in section 4.2.

3.4.1 Slicing and Information Flow Control

Slicing has a strong connection with non-interference and, hence, information flow. In the following, I give a cursory description of the intuition behind this connection. For my explanation, I use *batch-job termination-insensitive non-interference* (BTINI) [22] in a very simple setting.

Consider a deterministic program $\mathcal{P}$ which defines and uses only two variables h, l. Variable h is thought of to contain secret values that are not supposed to influence the variable l, which is assumed to be publicly accessible.

A suitable instantiation of BTINI then demands that

$$\forall \sigma, \sigma' \in \Sigma.\ \sigma(l) = \sigma'(l) \wedge \mathcal{P} \Downarrow \sigma \wedge \mathcal{P} \Downarrow \sigma' \implies [\![\mathcal{P}]\!](\sigma)(l) = [\![\mathcal{P}]\!](\sigma')(l),$$

where Σ is the set of all program states and $P \Downarrow \sigma$ means that $\mathcal{P}$ terminates for initial state σ.

Now assume that there is a valid slice $\mathcal{P}'$ of $\mathcal{P}$ with respect to the value of l at the end of $\mathcal{P}$ that does not contain any use of h. Then it can be shown that $\mathcal{P}$ is non-interferent. A rough and intuitive argument for this goes as follows: Fix a valid slice $\mathcal{P}'$ of $\mathcal{P}$ that does not contain any use of h and let $\sigma, \sigma' \in \Sigma$ with $\sigma(l) = \sigma'(l)$ and assume that $\mathcal{P} \Downarrow \sigma$ and $\mathcal{P} \Downarrow \sigma'$. Since $\mathcal{P}'$ is a valid slice of $\mathcal{P}$ with respect to the final value of l, we can conclude that $\mathcal{P}'$ also terminates on σ and that the final state $\mathcal{P}'(\sigma)$ coincides with $\mathcal{P}(\sigma)$ on l:

$$\mathcal{P}' \Downarrow \sigma \wedge \mathcal{P}'(\sigma)(l) = \mathcal{P}(\sigma)(l).$$

The same argument can be made for σ':

$$\mathcal{P}' \Downarrow \sigma' \wedge \mathcal{P}'(\sigma')(l) = \mathcal{P}(\sigma')(l).$$

Moreover, since (a) $\mathcal{P}'$ does not contain any use of h and therefore only uses l itself and (b) $\sigma(l) = \sigma'(l)$, it must be $\mathcal{P}'(\sigma)(l) = \mathcal{P}'(\sigma')(l)$. Together, it follows that

$$\mathcal{P}(\sigma)(l) = \mathcal{P}'(\sigma)(l) = \mathcal{P}'(\sigma')(l) = \mathcal{P}(\sigma')(l).$$

For PDG-based slicing, this basic idea can be exploited to give formal proofs for the general case. Horwitz et al. [91] show that program dependence graphs adequately capture program execution behaviour. Extending this work, Reps et al. [140] show the *Slicing Theorem* that states that PDG-based program slices are indeed valid, i.e. that a program and its slices exhibit the same execution behavior with respect to the slicing criterion.

Snelting, Robschink and Krinke apply the Slicing Theorem to argue that non-interference according to Goguen and Meseguer [71, 70] can be verified using PDGs and slicing [157].

Wasserrab [164] shows the correctness of PDG-based slicing and applies this result to show that slicing can be used to verify BTINI for sequential programs with multiple procedures.

For concurrent programs, simple properties such as BTINI are not sufficient anymore. I will consider non-interference properties for concurrent programs in section 4.2.

3.4.2 Dynamic Dispatch

In subsubsection 3.2.1.2, I discussed how interprocedural control-flow graphs are built by constructing the control-flow graphs of each procedure and then connecting them appropriately. This works fine for simple languages in which all call targets can be resolved *statically*, that is at compile-time.

Modern programming languages, however, usually support some form of *late binding*, i.e. that names are not resolved statically at compile-time but dynamically at run-time. In the context of procedures (which are also called *methods* in object-oriented languages), late binding is also known as *dynamic dispatch*: For a dynamically dispatched method, there may exist multiple implementations and in contrast to statically dispatched methods,

the actually executed implementation is not selected at compile-time but deferred to runtime.
In Java, the programmer can declare *classes* and two types of methods: *Static methods* are associated with the class itself and therefore independent of any instances of this class. They are dispatched statically, that is calls of them are resolved at compile-time.
The other type of methods are *instance methods*: They are associated with every instance of the class individually[9].
For instance methods, Java makes it possible to provide multiple implementations through its class inheritance mechanism which allows methods to be *overridden* – that is, a sub-class can re-declare a method declared in its superclass and provide another implementation for it.
For the call of an instance method, the bytecode only specifies the *static* call target, that is the call target which is derivable at compile-time from static type information. The method which is actually called is then resolved dynamically at run-time using the actual type of the receiver object.
An example for this can be seen in Listing 3.1: The method call in line 25 is dispatched dynamically. The static call target is `A::f`, however at runtime `B::f` is called since the runtime type of parameter `a` is `B`.
For a static analysis this means that method calls in general cannot be resolved uniquely at compile-time but must be approximated. In order for the static analysis to be sound, this has to be an *over*-approximation. The example Listing 3.1 also illustrates that it is crucial for a sound program analysis to capture all possible call targets of dynamically dispatched method calls. If it does not, it may miss important program behaviour: For example, Listing 3.1 contains an information leak: the secret value that is read in line 18 is printed to a public console in line 8. An information flow analysis that does not detect `B::f` as possible call target for the call in line 25 would miss this illegal information flow.
A static program analysis that aims for a safe over-approximation of the actual program behaviour must therefore in particular over-approximate the possible targets of every instance method call.
Hence, in the interprocedural control-flow graph multiple outgoing call edges from the same call target are allowed. With the definition of

[9] In Java there are also **`private`** methods which are only accessible from within the same class and therefore are also bound statically. But I ignore them here for simplicity.

```
class A {
  void f(int x) {
    //do nothing
  }
}
class B extends A {
  void f(int x) {
    print(x);
  }
}
class C extends A {
  void f(int x) {
    print(42);
  }
}
class Main {
  static void main() {
    int secret = readPIN();
    A a = new A();
    B b = new B();
    run(secret, b);
    run(secret, b);
  }
  static void run(int x, A a) {
    a.f(x);
  }
}
```

Listing 3.1: An example which shows why dynamic dispatch must be handled correctly

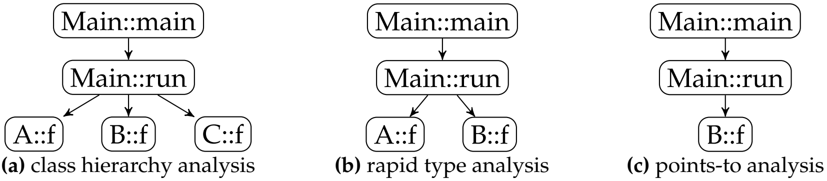
interprocedural control-flow graphs presented in Definition 3.2 on page 46, this is no problem: The correspondence relation Φ discussed earlier relates call *edges* with return *edges* and not just nodes and, thus, also incorporates the call target in the identification of a call.

There exist several program analysis techniques for the approximation of dynamically dispatched method calls that I will discuss briefly in the following paragraphs. Since Java allows that class loading is deferred to runtime, they all assume that all classes are available for static analysis.

All of these techniques construct a directed graph, the *call graph* which reflects the calling structure of a given program. Its nodes correspond to the program's procedures and there is an edge from p to p' if p may call p'. Call graphs can be obtained both dynamically and statically. In this work, I only consider static call graphs and because of that I will omit the qualifier *static* from now on.

The analysis techniques that I will describe mainly differ in their *precision*, that is, their ability to approximate the possible targets of a call as closely as possible.

Precise resolution of dynamic dispatch is an important and critical feature for any static analysis aimed at a language like Java and in particular for static information flow analysis tools like Joana.
For one, the precision of dynamic dispatch resolution directly affects the precision of a static analysis: Suppose that the object `b` in line 22 is an instance of `C` instead of `B`. Then the program in Listing 3.1 would be secure because the `secret` is never `print`ed. However, an analysis that fails to exclude `B::f` as possible call target in line 25 is also not able to rule out the execution of line 8.
Secondly, precise handling of dynamic dispatch is also important for scalability of a sound analysis: Consider a statement such as `p.equals(q)`. Then a sound analysis has to assume that `p` may be an instance of any available class and that the call `p.equals(q)` resolves to every available implementation of `equals` – unless it is able to incorporate additional information about `p`. If such information is not exploited, the resulting call graph may be substantially bigger than necessary, practically prohibiting any further analysis for scalability reasons.
Particularly, the applications of Joana that I present in chapter 4 (see e.g. section 4.3 or section 4.7) rely on precise handling of dynamic dispatch.



(a) class hierarchy analysis **(b)** rapid type analysis **(c)** points-to analysis

Figure 3.22: Call graphs for the program from Listing 3.1 resulting from the application of different analyses

3.4.2.1 Class Hierarchy Analysis

Perhaps the simplest non-trivial analysis for approximating dynamic dispatch is *class hierarchy analysis (CHA)* [52]. This analysis considers the inheritance relationships between classes and resolves a dynamically dispatched method call to the static call target `C::f` to all methods `D::f` such that `D` is a subclass of `C` that overrides method `f`.

Clearly, for class-based languages like JAVA, this rule is sound: If `D'` is an arbitrary class that is not a subclass of `C`, then no method `D::f` can be a valid call target for any call with static target `C::f`.

For the example in Listing 3.1, using CHA would resolve the call in line 25 to the possible call targets `A::f`, `B::f` and `C::f`. This results in the call graph depicted in Figure 3.22a. Note however that `C::f` is considered as a possible runtime call target even though `C` is never instantiated.

3.4.2.2 Rapid Type Analysis

Rapid Type Analysis (RTA) [23] is an improvement of CHA which additionally takes instantiation into account: If c is a call site with static call target `C::f` and `D` is a subclass of `C` that implements `f` and is instantiated somewhere in the program, then `D::f` is considered a possible call target for c.

Implementations of RTA usually require a main entry point and use an iterative approach to compute the classes which are instantiated and the methods reachable according to the above rule.

RTA is obviously still sound for languages like JAVA: `D::f` cannot be called at runtime if `D` is never instantiated. Moreover, RTA always delivers a result that is at least as precise as CHA: If CHA does not consider `D::f` as possible runtime call target, then neither does RTA.

For the example in Listing 3.1, RTA finds out that `C` is not instantiated. Therefore, it resolves the call in line 25 to the possible call targets `A::f` and `B::f`, resulting in the call graph in Figure 3.22b. This shows that RTA can be more precise than CHA.

However, RTA still cannot rule out the case that `A::f` is called since `A` is instantiated.

3.4.2.3 Points-To Analysis

Points-to analysis [16, 88, 158] is a general technique which aims to determine the possible runtime values of pointer variables. Among its numerous applications, points-to analysis can in particular be used to resolve static call targets [84].

Points-to analysis is concerned with the computation of *points-to graphs*, i.e. relations $PT \subseteq \mathcal{P} \times \mathcal{I}$ where $\mathcal{P}$ is a finite set of *abstract pointers* and $\mathcal{I}$ is a finite set of *abstract instances*.

For an abstract pointer p,

$$PT(p) \overset{def}{=} \{i \mid (p,i) \in PT\}$$

is also called the *points-to set of p*. The commonly used intuitive meaning for $o \in PT(p)$ is that *p may point to o at runtime* or, conversely, the intuitive meaning of $o \notin PT(p)$ is that p definitely does not point to o at runtime. This is specifically useful for call graph construction: in order to stay sound, one wants to rule out impossible call targets[10].

With points-to information available, static call targets can be resolved as follows: Let $c.f(o_1, \ldots, o_n)$ be a call site with static call target `C::f`. Then `D::f` may be an actual runtime call target if c may point to a D object. This is still sound: If c definitely never points to any D object, then `D::f` definitely is not a call target. Furthermore, points-to analysis always delivers a result which is at least as precise as the result of RTA: If D is not instantiated at all, then no reference can point to any D object.

Lastly, for the example in Listing 3.1, points-to analysis can find out that `a` in line 25 does not point to any instance of runtime type `A` or `C`. Hence, it can rule out `A::f` and `C::f` as runtime call targets for the call in line 25. This results in the call graph depicted in Figure 3.22c.

As I already mentioned, points-to analysis is a powerful analysis technique. Joana not only uses it to resolve dynamic dispatch but also for alias analysis. I will look at this more closely in subsection 3.4.4.

3.4.3 Exceptions

Exceptions are Java's mechanism and language construct for handling errors at runtime. If a program encounters an erroneous state or condition, it can *throw* an exception that can be *caught* at some other place to handle the error gracefully. In Java, there are two kinds of exceptions. *Explicit* exceptions have to be declared, thrown and caught explicitly. The other kind, *implicit* exceptions, are thrown by the Java runtime environment in

[10] Note that the notion *p may point to o at runtime* does not say anything about whether this means "sometime at runtime" or whether this statement is bound to a specific point of the program. Further note that it not necessarily means that p actually will point to o at some point at runtime. It only means that it is not the case that p definitely does not point to o at runtime (bound to some specific point or not).

certain situations for which there is no sensible reaction. This includes environmental problems like memory shortage or input/output errors, but also the failure of single instructions because of programming errors. For example, a field access of the form `a.x = y` may fail because `a` is **`null`**, or an array access `a[i] = o` may fail because `i` is out of the bounds of `a`.

An overview of Java's language constructs in connection with exceptions is given in Figure 3.23.

From the point of view of a static information flow analysis, exceptions are challenging: On the one hand, they have to be properly dealt with in order to capture every possible program behavior. In particular, bytecode instructions like field or array accesses may cause exceptions that are not apparent from the program's bytecode. Therefore, a static information flow analysis must model the behavior of these bytecode instructions carefully. On the other hand, exceptions have to be handled with sufficient precision in order to not introduce too much spurious control-flow which in turn may cause many false alarms [103]. Particularly, Joana's precision is heavily affected by its handling of exceptions. For example, the successful verification of the case studies described in section 4.3 would not have been possible without precise exception handling.

Like Joana's exception analysis, I focus in the following on implicit exceptions that are caused by programming errors.

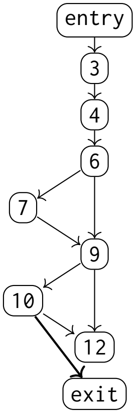
```
int readFile(File f)
throws IOException (1) {
  if (!f.exists()) {
    throw new IOException(); (2)
  }
  ...
}
int foo() {
  int[] arr = new int[5];
  ...
  return arr[6]; (3)
}
```

```
void bar(File f) {
  try { (4)
    int x = readFile(f);
  } catch (IOException e) {
    (5)
  }
  println(foo()); (6)
}
```

Figure 3.23: Overview of exceptions in Java: checked exceptions have to be declared (1), thrown (2) and handled (4)/(5); unchecked exceptions are thrown implicitly by the JVM and do not have to be handled (3)/(6)

```
 1  class A {int x;}
 2  void foo(String[] args) {
 3    int secret = inputPIN(); // HIGH access
 4    A benign = inputLOW();
 5    A a = null;
 6    if (benign != null) {
 7      a = new A();
 8    }
 9    if (secret < 9999) {
10      a.x = 42;
11    }
12    print(0); // LOW output
13  }
```

(a) source code



(b) control-flow graph

Figure 3.24: Example for an information leak through exceptions

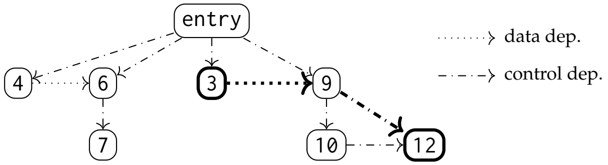


Figure 3.25: program dependence graph for the example in Figure 3.24 – the path from the high access in line 3 to 12 is highlighted in bold

Figure 3.24a shows an example for information flow that solely occurs because of implicit exceptions. The control-flow graph can be seen in Figure 3.24b and the program dependence graph in Figure 3.25.
With the assumption that the return values of `inputPIN` and `inputLOW` are not statically determinable, the program in Figure 3.24a has a control-flow graph like the one depicted in Figure 3.24b: In particular, this control-flow graph has a control-flow edge from line 10 to the exit node since the field access in line 10 may lead to a crash if `a` is **`null`**. If it fails, line 12 is not executed because the program crashes with a `NullPointerException`. This means that the value of `secret` influences whether line 12 is executed or not.

Consequently, the program dependence graph contains a control dependency from line 9 to line 12 that, together with the data dependency from line 3 to 9, constitutes a path from line 3 to line 12.

The example in Figure 3.24 shows that exceptions can induce a significant amount of additional control-flow and, thus, additional control dependencies. The control-flow from line 10 to the exit for example induces control-dependencies both from line 10 to 12 and from 9 to 12. If there were statements after line 12, then all these statements would also be control-dependent on both 9 and 10.

However, assume that `inputLOW` is never **`null`**. Then `a` cannot be null and therefore the program cannot crash. If static analysis cannot prove that `inputLOW` is never **`null`**, then it will report a false alarm. Therefore it can be beneficial to perform additional analyses that enable to safely rule out control-flow due to exceptions, e.g. by proving that certain pointer variables cannot be **`null`**.

JOANA performs an analysis that rules out impossible null pointer exceptions, both intraprocedurally and interprocedurally. Its capabilities are illustrated in Figure 3.26:

- Since it is flow-sensitive with respect to `a!=`**`null`**, it finds out that the field access in line 10 is safe.

- Moreover, it can detect that accesses such as the one in line 12 are safe because the pointer variable is guaranteed to be initialized.

- It also is able to follow references passed as parameters to methods. In particular, it tracks whether any of them may be **`null`**. Hence, the analysis can find out that line 17 is safe. However, it is still conservative enough to detect that accesses such as line 18 is not safe.

Details about JOANA's null pointer analysis can be found in the dissertation of Graf [78].

Apart from that, within the scope of the RS^3 project[11], we also integrated an analysis to rule out exceptions due to out-of-bounds array accesses. I will briefly go into that in section 4.3.

[11] I will introduce and explain RS^3 in chapter 4.

```
class A {int x;}
void foo() {
  int benign = inputLOW();
  A a = null;
  A b = new A();
  if (benign > 1742) {
    a = new A();
  }
  if (a != null) {
    a.x = 42;
  }
  b.x = 42;
  bar(b);
  baz(b);
  baz(null);
}
void bar(A a) {a.x = 42;}
void baz(A x) {a.x = 42;}
```

Figure 3.26: The capabilities of Joana's null pointer analysis

3.4.4 Objects

For static information flow analysis, objects pose a number of challenges:

- They usually have *fields* through which information can flow. An analysis that distinguishes between fields can be more precise than an analysis that does not. The ability to differentiate between two different fields of the same objects is also called *field-sensitivity*.
- In Java, objects can be *aliased* which means that the same location in memory can be referenced by different access paths. Information flow analysis has to take aliasing into account in order to ensure that all information flows can be detected.

Figure 3.27 shows two example that illustrate these challenges. In Figure 3.27a we see a small program that stores high and low data within the same object, but in different fields. Hence, the print-statement in line 8 does not reveal high data. Figure 3.27b shows why it is important to handle aliasing properly: The print statement in line 10 obviously leaks the secret that was just stored in `a1.x`. Since after line 6, `a1` and `a3` refer to the same object, line 14 is illegal as well. In contrast, line 12 does not leak secret information since `a1` and `a2` refer to different objects and therefore line 8 does not influence the value of `a2.x`.

Joana handles all the examples shown in Figure 3.27 properly by carefully incorporating objects into its PDG representation. The full details can be found in the dissertations of Hammer [86] and Graf [78].

3.4.4.1 Heap Dependencies

Central to this approach is the notion of *heap dependencies* that can be defined with the help of points-to analysis. I have already explained the basic intuition behind points-to analysis in subsection 3.4.2 and will go a bit deeper into it in subsubsection 3.4.4.3.

Heap dependencies can occur between statements that store to or read from the heap. A statement s_2 is called *heap-dependent* on a statement s_1 if s_2 may use a heap location that s_1 may have defined. For example, a

```
1    class A {int x; int y}
2    void foo() {
3      A a = new A();
4      a.y = 0;
5      int high = inputPIN();
6      a.x = high;
7      int low = a.y;
8      print(low); // OK
9    }
```

(a) fields

```
1    class A {int x;}
2    void bar() {
3      A a1 = new A();
4      A a2 = new A();
5      a2.x = 0;
6      A a3 = a1;
7      int high = inputPIN();
8      a1.x = high;
9      int out1 = a1.x;
10     print(out1); // ILLEGAL
11     int out2 = a2.x;
12     print(out2); // OK
13     int out3 = a3.x;
14     print(out3); // ILLEGAL
15   }
```

(b) aliasing

Figure 3.27: Two small example programs that illustrate aspects of objects that need to be handled properly by a static information flow analysis

statement $y = a.f$ is heap-dependent on a statement $b.g = z$, if the fields f and g are the same and if a and b may point to the same object.
For illustration, consider Figure 3.28 and Figure 3.29, respectively.
Figure 3.28 shows the data dependency graph of the code example in Figure 3.27a. The statement `low = a.y` is heap-dependent on `a.y = 0` since the former reads from the same heap location that `a.y = 0` has written to. Also note that there is no heap dependency between the statements from `a.x = high` to `low = a.y`. Although they both access the object that is pointed to by `a`, they refer to different primitive fields within `a`. In JAVA, if an object has two fields of primitive type[12] with different names, they are known to reside in different memory locations.
Another example can be seen in Figure 3.29, which shows the data dependency graph of the code example in Figure 3.27b.
This example shows three heap dependencies. Most notably, the statement `out3 = a3.x` is heap-dependent on `a1.x = high`. This is because the points-to sets of `a1` and `a3` coincide and therefore have a non-empty intersection. Hence, we conclude that `a3.x` in `out3 = a3.x` may refer to the same heap location as `a1.x` in `a1.x = high`.
Note that the data dependency graphs in Figure 3.28 and Figure 3.29 are simplified. In fact, JOANA represents field access not only by a single node but by multiple nodes. Mainly this is done to be able to distinguish between different information flow caused by field accesses. For example, the operation that reads an object's field's value from the heap is represented

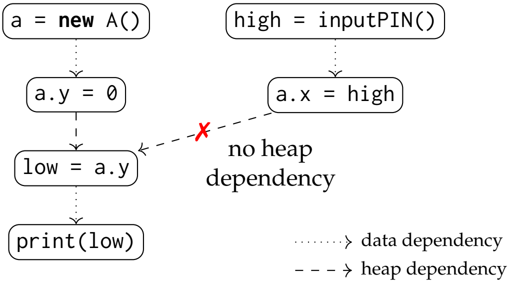


Figure 3.28: Data dependency graph of Figure 3.27a

[12] The term *primitive type* is used in JAVA for non-object types like **int**, **double**, **char** or **boolean**.

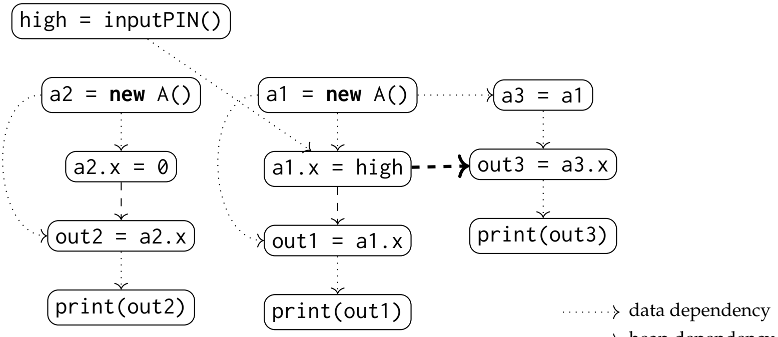


Figure 3.29: Data dependency graph of Figure 3.27b

by a structure as depicted in Figure 3.30. This structure consists of four nodes: One node for the actual instruction, two nodes for the base object and the field, respectively, and an additional artificial exit node. Such a representation makes it possible to distinguish between three kinds of information flow: For one, there are data dependencies from the base object and the field to the actual instruction (the instruction uses both the base object and the field to read its value). Secondly, a data-heap dependency to the field node represents information flow through the heap. And last but not least, the field access operation may fail because the base object is **null**.

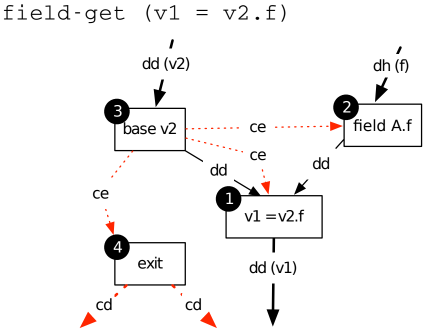
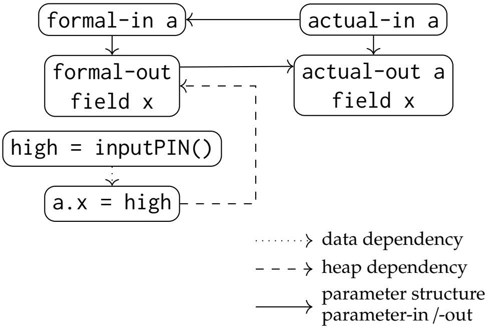


Figure 3.30: JOANA's PDG node structures corresponding to the operation that reads an object's field's value from the heap (taken from [78, Figure 2.31])

```
void foo() {
  ...
  bar(a);
  ...
}
void bar(A a) {
  int high = inputPIN();
  a.x = high;
}
```

(a) exemplified code snippet



(b) relevant section of the PDG of the code in a

Figure 3.31: How Joana incorporates objects into its parameter passing structures

Only the base object and the exit node are involved in this exceptional information flow.

3.4.4.2 Propagating Heap Access Across Procedures

In order to also model field accesses across procedural boundaries, Joana incorporates them in the structures that represent interprocedural parameter passing. For every method `m`, additional formal parameter nodes represent the field accesses performed by `m` or by any method called directly or indirectly by `m`.

For this, Joana performs an *interprocedural side-effect analysis*: For each method, all its field accesses are collected and summarized as formal parameter nodes: Each of these additional parameter nodes represents read or write access to a set of heap locations. Moreover, Joana relates parameter nodes by *parameter-structure edges*: Roughly, parameter node p is connected to a parameter node p', if p' represents a field of p. More technically, parameter node p is connected to a parameter node p' if the heap locations of p' contain a field that can be obtained by dereferencing an object represented by p.

Consider Figure 3.31 for a simple example: Method `bar` writes the field `x` of its parameter `a`. Hence, `bar` has a formal-out parameter node for the field `a.x`. This node is propagated to callers, so for each of `a`'s call sites, there is an actual-out parameter node corresponding to the formal-out parameter node for a.x. The formal-in node for bar's parameter a is connected to the formal-out node for a.x, since the latter represents accesses to the field x of a.

3.4.4.3 Points-to Analysis

For the remainder of this section, I take a closer look at points-to analysis, which is not only a tool for constructing precise call graphs but also a key ingredient for analyzing information flows across the heap.
Points-to analysis as used by JOANA is not a single, fully-determined analysis but rather a family of possible concrete points-to analyses where various aspects can be configured. Every choice has consequences with respect to the JOANA's precision and runtime performance. Hence, the choice of points-to analysis is important for practical applications of JOANA and deserve explanation. Particularly, I will illustrate that for demanding analysis clients such as information flow analysis, there is no perfect choice of points-to analysis.

Abstractions As I already mentioned in subsubsection 3.4.2.3, points-to analysis is concerned with the computation of *points-to graphs*, which are one-to-many relations between abstractions of pointer variables and abstraction of concrete object instances. These abstractions can be thought of as some kind of description generated from the static information available in the program's code.
Consider as an example Figure 3.32: Since n is a parameter, potentially infinite List objects are created in the loop. A common approach for points-to analyses is to represent all these objects by one abstract instance, described by something like "any instance of List that is instantiated in line 10". Furthermore, the potentially infinitely many incarnations of the local pointer variable pr are represented by one abstract pointer variable, described by something like "the pointer variable pr at any point in the method List::create".

Sensitivities Like all static analyses, points-to analyses are subject to several precision trade-offs. In the following, I look more closely at some of them.

```
class List {
  int d;
  List prev;
}
```

```
class Foo {
  List create(int n) {
    List cur = null;
    for (int i = 0; i < n; i++) {
      List pr = cur;
      cur = new List();
      cur.d = i;
      cur.prev = pr;
    }
    return cur;
  }
}
```

Figure 3.32: A code snippet that illustrates the abstractions in points-to analysis

Flow-sensitivity Recall the general descriptions in section 3.1: A flow-sensitive points-to analysis takes into account the order of statements, whereas a flow-insensitive points-to analysis does not. Flow-insensitive points-to analyses usually compute one global points-to graph for the whole program, whereas flow-sensitive points-to analyses result in a separate points-to graph for each statement.

Figure 3.33 shows an example which highlights the effect of flow-sensitivity in points-to analysis: We see two code snippets there which are identical up to statement order. Flow-insensitive points-to analysis computes the same points-to graphs for both of them, whereas flow-sensitive analysis results in different points-to graphs.

Joana employs flow-insensitive points-to analysis. Some of the resulting precision loss is recovered by using *Static single assignment form* (SSA) [47, 38, 42], an intermediate representation which is widely used in compilers and program analysis tools. Specifically, it is employed by WALA and Joana which is why I want to describe it briefly in the following. The key property of SSA form is that every variable is assigned to at most once. This simplifies some program analyses. For example, the reaching definitions analysis described before becomes simpler because definitions do not need to be deleted anymore.

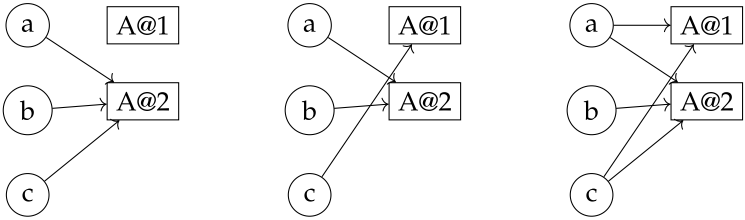
Every program can be transformed into SSA form. This transformation is usually performed on the given program's control-flow graph. The idea is to introduce a separate copy for each definition of a variable. At join

```
a = new A();
b = new A();
a = b;
c = a;
```

(a)

```
a = new A();
b = new A();
c = a;
a = b;
```

(b)



(c) **(d)** **(e)**

Figure 3.33: Two code snippets (upper part) and their points-to graphs (lower part) – c and d show the flow-sensitive points-to graphs at the end of the code in a and b, respectively. The points-graph in e results from flow-insensitive points-to analyses of both snippets.

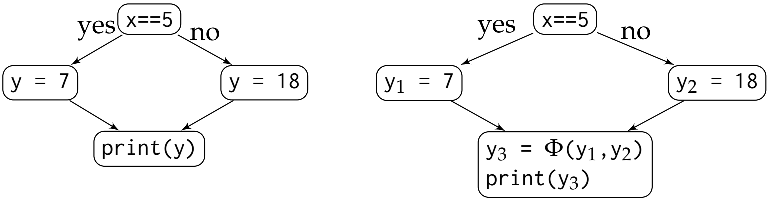


Figure 3.34: Example showing a simple program and its SSA form

points, where several control-flow paths meet, special statements called Φ functions have to be inserted. An easy example is given in Figure 3.34: The statement $y_3 = \Phi(y_1, y_2)$ means that y_3 is either y_1 or y_2, depending on which control-flow path was taken before. Figure 3.35 shows the control-flow graph of the running example in SSA form. At node 9, three Φ statements[13] have to be inserted. For simplicity, I allow all these statements to be contained in the same basic block and assume that they are executed at the beginning of each loop iteration, just before the loop predicate is evaluated.

Consider Figure 3.36 for an example of how SSA form can affect points-to precision. It shows the code snippets of Figure 3.33 in SSA form and their flow-insensitive points-to graphs. It can be seen that the points-to graphs

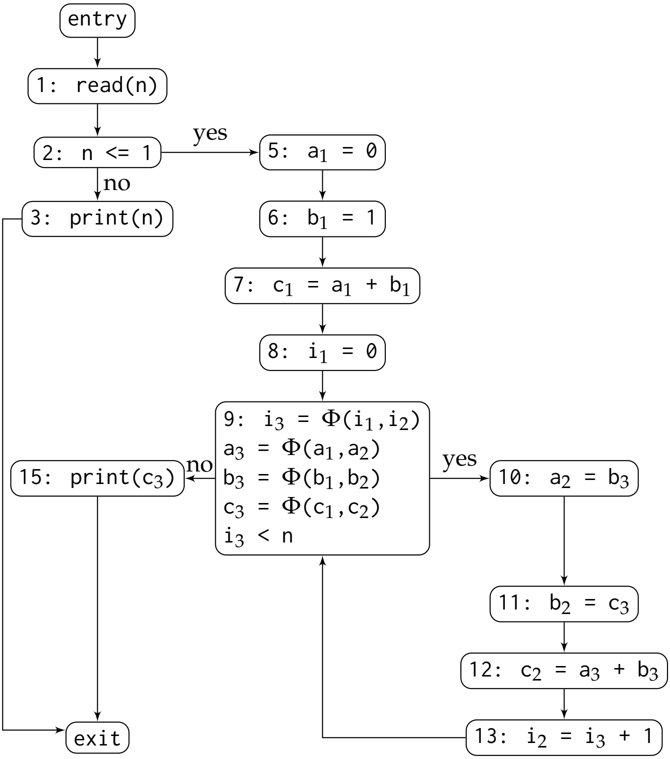


Figure 3.35: The control-flow graph from Figure 3.3 in SSA form

[13] I use the symbol Φ here to be consistent with the literature. This is not to be confused with the correspondence function for interprocedural control-flow graphs.

differ and are almost the same as the flow-sensitive graphs depicted in Figure 3.33c and Figure 3.33d, respectively. Due to SSA form, it is made explicit which definition of a is used in the last two lines. Since there are two local variables for a now (one for each definition), there are also two points-to sets.

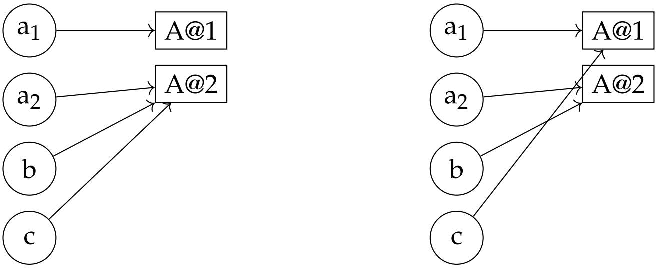
Equality-Based vs. Subset-Based Another precision trade-off for points-to analyses is whether they are *equality-based* [159] or *subset-based* [16]. Equality-based points-to analyses do not take into account the direction of assignments, whereas subset-based points-to analyses do. An example for this can be found in Figure 3.37. The two code snippets only differ in whether b is assigned to a or vice versa. Since subset-based points-to analysis is sensitive to this difference, it produces two different points-to graphs, whereas equality-based points-to analysis produces the

```
a1 = new A();
b = new A();
a2 = b;
c = a2;
```

(a)

```
a1 = new A();
b = new A();
c = a1;
a2 = b;
```

(b)



(c)

(d)

Figure 3.36: Effect of SSA form on flow-insensitive points-to analysis – The upper part shows the code snippets from Figure 3.33 in SSA form and the lower part the respective flow-insensitive points-to graph

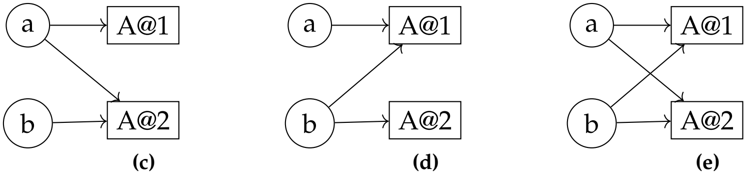
same points-to graph for both snippets. Joana uses subset-based points-to analyses.

```
a = new A();
b = new A();
a = b;
```

(a)

```
a = new A();
b = new A();
b = a;
```

(b)



(c) (d) (e)

Figure 3.37: Subset-based (c, d) vs. equality-based (e) points-to analysis

Context-Sensitivity Finally, I want to consider the effect of context-sensitivity on points-to analyses and their client analyses, specifically on the heap dependency graph construction performed by Joana.
As already explained generally in section 3.1, a context-sensitive analysis not only analyzes the individual statements of a program but also takes into account their *execution context*[14]. For points-to analyses, the execution context can include

- (a finite portion of) the runtime stack just before the execution (*k*-CFA [155])
- the instance on which the method was called which contains the statement (object-sensitivity [125])
- the instance on which the method was called which contains the statement combined with the combination of the values of object-valued parameters (*cartesian product algorithm*(CPA) [8])

[14] Note that context-sensitive points-to analysis is usually not *fully* context-sensitive. This can be compared to the call string approach subsubsection 3.2.2.2 with limited stack depth.

Usually, the context information is incorporated in the abstract descriptions of pointers. The kind of used context information can have a profound effect on client analyses, especially on Joana's PDG construction algorithm.

This is illustrated by the example programs in Figure 3.38. Both programs are secure, since the value they print is not affected by the secret input. Let us first consider Figure 3.38a in more detail. If Joana analyzes this program with context-insensitive analysis, it yields a heap dependency graph like depicted in Figure 3.39a. This is caused by the fact that (a) the points-to analysis performed by Joana is flow-insensitive, (b) after side-effect analysis is performed for each method, the same access summary is propagated to the callees without adapting it to the respective call site and

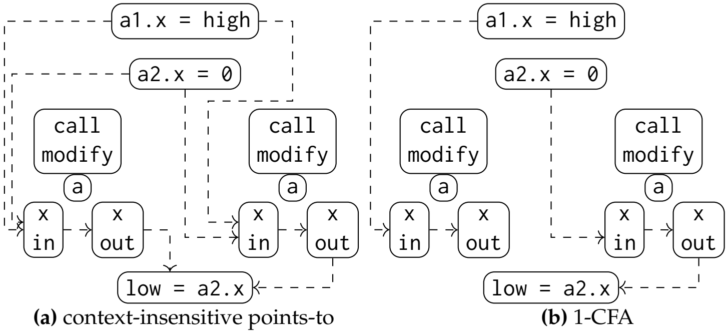
```java
class A {
  int x;



}
class CM1 {
  void foo(int high) {
    A a1 = new A(); // o1
    a1.x = high;
    A a2 = new A(); // o2
    a2.x = 0;
    modify(a1);
    modify(a2);
    int low = a2.x;
    println(low);
  }
  void modify(A a) {
    a.x++;
  }
}
```

(a)

```java
class A {
  int x;
  void modify() {
    a.x++;
  }
}
class CM2 {
  void foo(int high) {
    A a1 = new A(); // o1
    a1.x = high;
    A a2 = new A(); // o2
    a2.x = 0;
    A a = random()>0.5?a1:a2;
    a.modify();
    int low = a2.x;
    println(low);
}
```

(b)

Figure 3.38: Two example programs showing the effect of context-sensitivity in points-to analysis on information flow analysis



(a) context-insensitive points-to

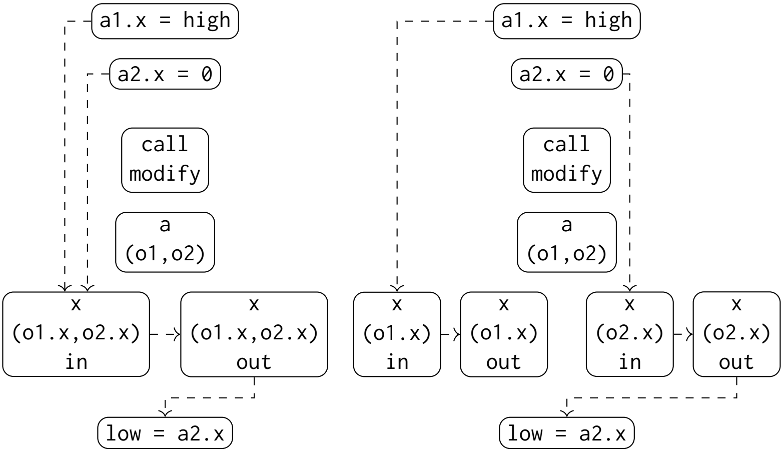
(b) 1-CFA

Figure 3.39: Relevant section of the heap dependency graph of Figure 3.38a with different pointer analyses

(c) with context-insensitive points-to analysis, the parameter `a` of method `modify` is described by the same abstract pointer, regardless of the call site. Since both `o1` and `o2` may be passed to `modify`, `a` may point to both of them. Hence, according to the points-to information, the field access in line 19 may access both `o1.x` and `o2.x`. This information is propagated to both call sites of `modify` and, based on this information, JOANA adds a heap dependency: one from line 10 to the actual-in node for `a.x` at the call in line 14 and one from the actual-out node for `a.x` at the call in line 13 to line 15. This constitutes a PDG path from line 10 to 15.

In contrast, such a mix-up does not occur when JOANA analyzes the example with 1-CFA. Here, a heap dependency graph such as the one depicted in Figure 3.39b is obtained. The reason is that 1-CFA uses two different abstract pointers for the parameter `a` of `modify`, one for each call site. This way, both calls can be treated as if they called different methods, which each operate exclusively on `o1` and `o2`, respectively. In effect, the access summaries become more precise, JOANA does not add the spurious heap dependencies, so that line 10 and line 15 are not connected via heap dependencies. Note that object-sensitive points-to analysis yields the same result as context-insensitive points-to because `modify` is only called on one object.

However, there are also examples where no *k*-CFA helps: Such an example is shown in Figure 3.38b. Again, this program is secure as it always prints `0`, regardless of the high value.



(a) context-insensitive points-to **(b)** object-sensitive points-to

Figure 3.40: Relevant section of the heap dependency graph of Figure 3.38b with different pointer analyses

As the return value of `random()` cannot be statically determined, the points-to set of `a` contains both `o1` and `o2`, with any points-to analysis. Hence both `o1` and `o2` may be the receiver object of the call in line 14. Both context-insensitive points-to and *k*-CFA merge `o1` and `o2` into the points-to set of the **`this`** pointer in `modify`. In particular *k*-CFA uses only one abstract pointer for **`this`**, since `modify` only has one call site. This leads to multiple spurious heap dependencies and a false alarm, as can be seen in Figure 3.40a.

In contrast, object-sensitive points-to analysis uses two different abstract pointers <`o1`,**`this`**> and <`o2`,**`this`**> for **`this`** and can distinguish between the call on `o1` and the call on `o2`. The points-to set of <`o1`,**`this`**> only contains `o1` and the points-to set of <`o2`,**`this`**> only contains <`o2`,**`this`**>.

Technically, Joana treats `o1::modify` and `o2::modify` as two different methods. Hence, it also computes two distinct access summaries that are both propagated to the only call site. In effect, Joana is able to keep the accesses on `o1` and `o2` separate. The heap dependency graph, which is shown in Figure 3.40b, does not contain the spurious heap dependencies of Figure 3.40a and hence also not the unnecessary heap dependency path.

```
 1 class A {
 2   B b;
 3   void init(int x) {
 4     B b = new B();
 5     // <o1::init, o3>,
 6     // <o2::init, o3>
 7     b.x = x;
 8     this.b = b;
 9   }
10 }
11 class B {
12   int x;
13 }
```

```
13 class C {
14   void foo(int high) {
15     A a1 = new A(); // o1
16     A a2 = new A(); // o2
17     A a = random()>0.5?a1:a2;
18     a.init(high);
19     B b = a2.b;
20     int low = b.x;
21     print(low);
22   }
23
24 }
```

Figure 3.41: Effects of a context-sensitive heap model on JOANA's PDG construction

Additional context information can not only improve the representation of abstract pointers, but also of abstract instances. A context-sensitive points-to analysis that incorporates the context information in its representation of abstract instances is said to employ a *context-sensitive heap model*. With a context-sensitive heap model, more instances can be distinguished, which again can be beneficial for client analyses like JOANA's PDG construction, especially when dealing with nested object structures.

Consider the example in Figure 3.41. We assume that JOANA uses object-sensitive points-to analysis with a context-sensitive heap model. Because of object-sensitive abstract pointers, the two possible calls of `A::init` in line 18 are treated as calls to two different methods, `o1::init` and `o2::init`. This additional context information is used to split up the creation site of `B` in `A::init`. In `o1::init`, the local variable `b` points to `<o1::init, o3>` and in `o2::init`, it points to `<o2::init, o3>`. Hence the write accesses to `b.x` in line 7 can be separated and JOANA is able to conclude that the heap location accessed in line 20 does not contain high information.

The examples I just presented not only show that context-sensitivity can have a positive effect on JOANA's precision but also that there is no perfect choice of points-to analysis. Indeed, Figure 3.38a is an example that profits from the use of 1-CFA, whereas with object-sensitive points-to, JOANA reports a false alarm. In contrast, JOANA can verify the security of Figure 3.38b only with object-sensitive points-to, whereas no *k*-CFA provides enough context information to eliminate false alarms. Figure 3.41 illustrates the same phenomenon for context-sensitive heap models: Joana does not report an illegal flow with object-sensitive points-to but no *k*-CFA is sufficient for that. With a slight variation of Figure 3.41, we yield an example analogous to Figure 3.38a whose security can be proven with 1-CFA, while object-sensitivity has no positive effect in comparison to context-insensitive points-to.

Performance Impact of Points-to Analysis The examples also give an idea of another important effect of the choice of points-to analysis, namely the effect on the runtime performance of Joana. In my explanation of the examples, I mentioned two important aspects of Joana's modelling of objects and handling of points-to analysis:

- Joana implements context-sensitivity in points-to analysis by treating a method with different contexts as two different methods. That is, if a method is analyzed in multiple points-to contexts, multiple copies of it occur in the call graph and in Joana's PDG.

- Joana performs an analysis of memory access for each method in each points-to context and propagates summaries of these field accesses from callee to caller, in order to construct the heap dependency graph. These summaries are not only propagated to the direct, but also to indirect callers. Hence, every procedure dependency graph contains information about every field access performed by one of its direct or indirect callers.

These two aspects can cause a substantial amount of PDG nodes representing field accesses, in particular at parameter nodes at call sites. For example, this may lead to a significant rise in the memory and runtime performance of summary edge computation.
Therefore, the choice of the points-to analysis and how Joana incorporates points-to information into its structures plays an important role for applications such as those presented in chapter 3. For example, the case study described in section 4.7 could only be verified with object-sensitive points-to, and the PDG construction times differed significantly between different choices of points-to analysis. I will discuss this in section 4.7.

Listen, do you want to know a secret?
THE BEATLES

4 Applications of JOANA to Software Security

In this chapter, I report on several applications of JOANA within the scope of the *Reliably Secure Software Systems* priority program (RS^3) that ran from 2010 to 2017 and was funded by the German Research Foundation (DFG). In particular, I will focus on the contributions of the programming paradigms group of Prof. Dr.-Ing Gregor Snelting at KIT.
The chapter is organized as follows. In section 4.1 I give a general introduction into the RS^3 project and its motivations. After that, I describe various collaborations with other groups of RS^3. These collaborations can be generally split into three categories:

- RS^3 consisted of several sub-projects – the programming paradigms group was part of the sub-project *Information Flow Control for mobile components*. I describe this project and its contributions in section 4.2.
- Within RS^3, three *reference scenarios* were developed. They served as real-world examples where the several research groups who participated in RS^3 [3] could apply and combine their results. Our group participated in two of the three reference scenarios:
  - In section 4.3, I describe the *Security in E-Voting* scenario [41] and how JOANA was combined with an interactive theorem prover to prove certain cryptographic properties of a prototypical electronic voting system.
  - In section 4.4, I describe the reference scenario *Software Security for Mobile Devices* [20] and particularly how JOANA was extended to show information flow properties of Android applications.

- In addition to such structural umbrellas like the reference scenarios, RS^3 also fostered other collaborations of the several research groups that were part of it. Our group participated in four such collaborations:
  - We took part in the development of the *RS^3 Information Flow Language* (RIFL), a language that makes it possible to specify security requirements in a tool-independent, language-independent and machine-readable way. I describe RIFL and Joana's support for it in section 4.5.
  - As an application of RIFL, several RS^3 research groups developed ifspec, a benchmark suite for information flow security. Together with two other tools, Joana and its Android variant Jodroid can be evaluated using ifspec. I go into the details of ifspec in section 4.6.
  - In section 4.7, I report on the SHRIFT approach, which shows how a static information flow control tool like Joana can be applied to improve the performance, precision and thus the usability of system-wide usage control.
  - Last but not least in section 4.8 I shortly report on a collaboration with the *Application-oriented Formal Verification* group at KIT that presents an approach to the modular security verification of component-based systems in which Joana was used to lower the burden for a first-order theorem prover.

4.1 General Description and Motivation of RS^3

In this subsection, I give an overview over the motivation, goals and structure of the RS^3 priority program. This overview is based on the descriptions that were given on the website [6] of RS^3 and on excerpts of the program's proposal, which also can be found on its website [4, 2].
The main thesis of RS^3 was that there is a need for complementing traditional approaches to IT security in order to give reliable security guarantees for complex software systems.

RS^3 In classical IT security approaches, mechanisms like authentication, cryptographic protocols are used to ensure that only *trusted* entities (e.g.

programs) may perform actions in a given system. Trust is usually provided by some form of certificate that uses cryptographic signatures to prove the given entity's identity and integrity. Additionally, access control ensures that a given entity may only perform allowed actions.

Together, these techniques create a zone that is to a great extent protected from unknown and potentially malicious code. However, even with such a zone it remains unclear what guarantees can be given with respect to security. Once an entity has entered the trusted zone, it may perform all allowed actions. For example, it may combine these actions to do harm or to disclose information which are not supposed to be disclosed. In consequence, trust-based and mechanism-oriented approaches cannot protect from malicious entities that are falsely trusted.

Therefore, such approaches need to be complemented by *property-oriented solutions*: Such solutions concentrate on (a) formalizing the security requirements of a given system in the form of properties and (b) providing methods for verifying that the given system enjoys these properties.

Focusing on properties and their verification offers a number of advantages: Firstly, security properties can be rigorously analyzed (e.g. for contradictions). Moreover, precise and well-defined security guarantees can be given if a system can be rigorously shown to satisfy a given property.

Therefore, the main goal of RS^3 was to develop concepts and techniques that enable trustworthy certification of system-wide, technical security requirements and which adequately respect the semantics of programs.

In order to achieve this goal, RS^3 was driven by three guiding themes:

1. the development of precisely defined security properties; such properties enable to formalize and hence reason about security, requirements for a given system, just like it is common for functional requirements.

2. the development of program analysis methods and tools for the verification of security properties; ideally, these techniques are sound, scalable and usable, and

3. the development of concepts for understanding and certifying security aspects in complex software systems.

4.2 The Sub-Project "Information Flow Control for Mobile Components"

Information Flow Control For Mobile Components (IFC4MC) was a sub-project within RS^3 that comprised the programming paradigms group at KIT and the Software Construction and Verification group at the university of Münster.
I focus on the KIT side of the projects. From our point of view, IFC4MC was concerned with three main topics:

1. information flow properties that are suitable for modern program structures like concurrent programs

2. enforcement of these properties using program dependence graphs

3. modularity of PDG-based program analyses

In the following, I give an overview of the achievements in the first two items. The first two items were first tackled by Giffhorn [65, 66]. The third item is considered in detail in the dissertation of Graf [78].

4.2.1 Information Flow Properties for Concurrent Programs

The security properties that we were mainly concerned with are *non-interference-like*. Generally, non-interference-like properties demand that high input does not influence low-observable program behavior. For sequential and deterministic batch-like programs, this essentially means that if the program is applied to two states that only differ in high variables, the result states also only differ in high variables.
However, this is insufficient for advanced programming language features like concurrency. In contrast to sequential programs, *concurrent*, or *multi-threaded* programs are composed of multiple *threads*, sub-programs that run simultaneously and may or may not be executed on multiple processors. A *scheduler* periodically distributes threads to the available processors. Consequently, if no particular scheduler is assumed, concurrent programs have to be considered as *non-deterministic* in the sense that a given input may lead to multiple possible program behaviors. This has to be accounted for when designing appropriate security properties.

Figure 4.1a and Figure 4.1b show examples that illustrate different types of possible information leaks in concurrent programs.

Figure 4.1a contains one simple explicit (A) and one simple implicit (B) leak. These two types of information flow also can occur in sequential programs. The example also contains a third leak (C) that can also be considered an explicit information flow but crosses thread borders: It may occur since statement (D) may be executed by thread `t1` before the `main` threads executes statement (C).

Another type of leak is shown in Figure 4.1b. This example neither contains an explicit nor an implicit information leak. However, there are two output statements which may be observed by a low attacker and whose execution order may reveal something about high data: Observe that `t2` is delayed by a loop whose execution time directly depends on the high input `pin`. Hence, if we assume a scheduler that after each step chooses the next active thread by fair dice roll, then the larger the `pin` is, the more likely it is that `t1` executes its output statement before `t2` does.

The notion that captures such considerations is *probabilistic non-interference*. The idea here is to consider the probability distribution of possible program behaviors. Probabilistic non-interference then demands that if two inputs

```
main:
  pin = input(HIGH)
  spawn t1
  output(LOW, 42 * pin + 17) (A)
  if (pin > 0) {
    output(LOW, pin) (B)
  }
  output(LOW, x) (C)
t1:
  x = pin (D)
```

(a)

```
main:
  spawn t1
  pin = input(HIGH)
  while (pin > 0) {
    pin--
  }
  spawn t2
t1: output(LOW, 0)
t2: output(LOW, 1)
```

(b)

Figure 4.1: Examples for different types of leaks that can occur in concurrent programs: a shows explicit information flows within (A) and across (C,D) thread borders and an implicit information flow (B); b shows a truly probabilistic leak: the larger the value of `pin` the more likely it is that `0` is output before `1`

only differ in high parts, then the probabilities of the resulting low-observable program behavior is the same.

Let Tr be the set of program behaviors, also called *traces*, and I the set of inputs. A trace can be thought of to be a sequence of operations that accurately describes what a program does and how the memory contents develop. Some operations are used for input and output. We assume that input and output operations (also called input events and output events, respectively) use different channels for high and low observers.

An input $i \in I$ leads to multiple possible traces $t \in Tr(i) \subseteq Tr$. Each of these traces has an occurrence probability[15] $P_i(t)$.

An attacker does not see the full trace but only its so-called *low-observable* part. This is modeled by a function $E_L : Tr \rightarrow Tr$ that strips off the parts of a trace that cannot be observed by a low attacker, like output events on high channels or high parts of the memory.

The attacker also only sees some part of the input, namely those input events that operate on the low channel. To model this, we overload E_L to $E_L : I \rightarrow I$ that strips off high parts of inputs.

Hence, if program $\mathcal{P}$ is run with input i and exhibits program behavior t, then a low observer only sees the low part $i_L = E_L(i)$ of that input and observes that $\mathcal{P}$ exhibits $t_L = E_L(t)$. From this, they can conclude that the input must have been some $i' \in E_L^{-1}(i_L)$ and that some $t' \in E_L^{-1}(t_L)$ must have been executed.

Now, probabilistic non-interference aims to ensure that the attacker cannot learn anything from that. The argument goes as follows: Assume that the attacker knows the probabilities $P_{i'}(E_L^{-1}(t_L))$ for all $i' \in E_L^{-1}(i_L)$ and also assume that these probabilities differ. In other words, there are $i_0, i_1 \in E_L^{-1}(i_L)$ with $P_{i_0}(E_L^{-1}(t_L)) > P_{i_1}(E_L^{-1}(t_L))$. Then the attacker could conclude that it is more likely that the full input is i_0 than i_1. In order to deprive the attacker of this possibility, probabilistic non-interference demands that $P_{i_0}(E_L^{-1}(t_L)) = P_{i_1}(E_L^{-1}(t_L))$ for all $i_0, i_1 \in E_L^{-1}(i_L)$.

The example in Figure 4.1b clearly violates probabilistic non-interference: The attacker may conclude from the low output `01` that the `pin` was probably large and from `10` that it was probably small.

[15] In general, P_i assigns probabilities to *sets* of traces. We assume a countable set of traces, hence all P_i are discrete probability distributions so that P_i is fully specified by specifying it for single traces.

```
main:
  spawn t1
  spawn t2
t1: output(LOW, 0)
t2: output(LOW, 1)
```

(a)

```
main:
  spawn t1
  spawn t2
  pin = input(HIGH)
  while (pin > 0) {
    pin--
  }
t1: output(LOW, 0)
t2: output(LOW, 1)
```

(b)

Figure 4.2: a: An example which passes Giffhorn's criterion but not LSOD – b: Giffhorn's criterion allows access to high input before the first low-observable non-determinism occurs

Probabilistic non-interference is in general hard to verify directly as it requires to know probability distributions of traces which is why Giffhorn also considered sufficient criteria for it. One such criterion is *low-security observational determinism* (LSOD) [169], which essentially demands that a program only has one low-observable low-behavior for a given low-part of the input. If LSOD holds, then the probability distributions become very simple and probabilistic non-interference can be easily verified.

Central to LSOD is the observation that non-determinism manifests itself in the form of *conflicts*. Two operations form a conflict if they may be executed in multiple orders. For instance, if two statements s_1 and s_2 are composed concurrently then the scheduler may decide to run either of them first so that both the execution orders s_1, s_2 and s_2, s_1 are possible. Apart from the absence of explicit and implicit leaks, LSOD also demands that there is *no* conflict of low-observable events. For instance, the example Figure 4.1b is clearly rejected by LSOD since it contains two conflicting low-output statements.

One particularly pleasant property of LSOD is that it is *scheduler-independent*. This means that no assumptions on the scheduler are necessary for probabilistic non-interference to hold. Therefore, a respective security certificate can be re-used when changing the environment.

In his dissertation [65], Giffhorn showed that LSOD can be checked using program dependence graphs.

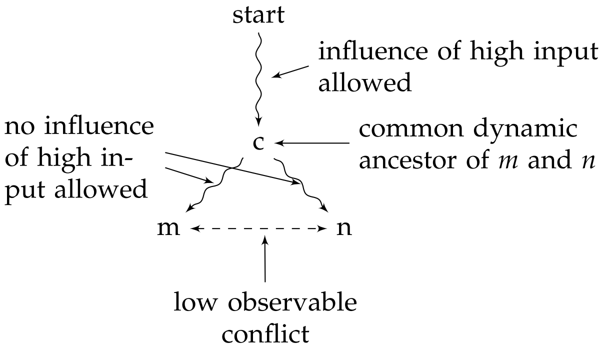

start
influence of high input allowed
no influence of high input allowed
c
common dynamic ancestor of m and n
m
n
low observable conflict

Figure 4.3: Idea of the RLSOD improvement

However, LSOD is also very restrictive: It essentially forbids any low-observable non-determinism, which is often the very motivation for designing a system in a concurrent way in the first place. So, even a program that does not access high data, like the one in Figure 4.2a, violates LSOD if it contains conflicts.

This is why Giffhorn also proposed a slight improvement on LSOD. His idea was to allow conflicts if they are not influenced by high data. One example of this can be seen in Figure 4.2b.

In later work [40, 31], we took this as a starting point for improvements on LSOD. The central idea here was to push the "influence sphere" of high events on conflicts even further into the direction of conflicts. We observed that, under some assumptions about the scheduler, more conflicts can be considered benign. Key to this observation is the notion of *common dynamic ancestors (cda)*: For two statements m, n, a statement c is called *common dynamic ancestor* of m and n if (a) c is a dominator for m and n, i.e. if every control-flow path to m or n must traverse c first and (b) if c is guaranteed to never execute concurrently to m and n. Hence, a common dynamic ancestor of m and n is any point in the program which is guaranteed to be executed before m and n.

Now the idea of RLSOD, which is sketched in Figure 4.3, is as follows: If c is a common dynamic ancestor of two conflicting statements m and n, any statement s that is guaranteed to be executed before c can only delay both m and n but cannot determine in which order m and n are executed. Hence s can safely be allowed to be influenced by high input. Consequently, it

```
main:
  pin = input(HIGH)
  while (pin > 0) {
    pin--
  }
spawn t1
spawn t2
t1: output(LOW, 0)
t2: output(LOW, 1)
```

Listing 4.1: An example which passes RLSOD but not Giffhorn’s criterion

suffices to check that no statement *s* that lies on some control-flow path between *c* and *m* or *c* and *n* may be influenced by high input.

As an example, consider the program in Listing 4.1: It contains a low-observable conflict after high data is accessed, hence it is rejected by Giffhorn’s criterion. However, this high access is guaranteed to execute before t1 is spawned – a point in the program that is a common dynamic ancestor of the two output statements. Such high access is allowed by RLSOD and hence the conflict can be considered benign. As we showed in Bischof et al. [31], this consideration works with every common dynamic ancestor. Indeed, the RLSOD check can be specified with respect to a function *cda* which assigns every two statements m, n a common dynamic ancestor. The closer $cda(m, n)$ is to m and n, the larger the portion of the program on which influence of high input is allowed can be and, hence, the more precise the check becomes.

An imprecise yet safe choice is to always use the start point of the program as *cda*. With this choice of the *cda* function, the RLSOD check becomes essentially Giffhorn’s criterion.

4.2.2 Modeling and Analyzing Concurrency in Program Dependence Graphs

Basically, PDGs for multi-threaded programs can be obtained by computing a PDG for each thread and connecting these sub-PDGs using *interference edges*, an additional kind of edge that captures inter-thread data dependencies.

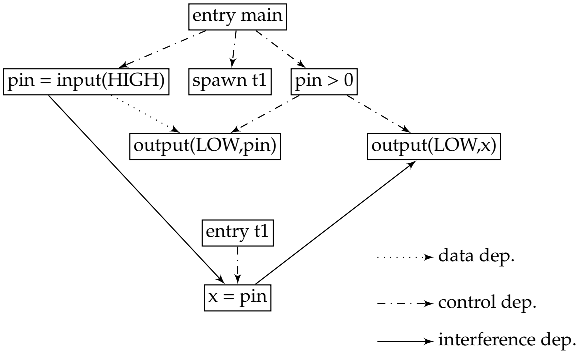


Figure 4.4: PDG of Figure 4.1a with interference edges

To illustrate this kind of dependency, consider again Figure 4.1a. The information leak in statement (C) occurs because statement (D) may be executed before statement (C) and therefore the high value `pin` is first written into the variable `x` and then output to a low channel. This is a special kind of data dependency that crosses thread borders and is called *interference dependency*. The PDG of Figure 4.1a can be found in Figure 4.4. As defined formally by Giffhorn [65, Definition 3.8], statement s_2 is interference-dependent on s_1, if s_2 may use a value which s_1 computes and s_1 and s_2 *may happen in parallel*. Two statements s_1 and s_2 *may happen in parallel* if it is both possible that s_1 is scheduled before s_2 and that s_2 is scheduled before s_1.

Due to decidability reasons, may-happen-in-parallel information must be approximated. To yield a sound analysis, this approximation is conservative in the sense that the statically computed relation *MHP* has the property:

$$s_1 \text{ and } s_2 \text{ may happen in parallel} \implies MHP(s_1, s_2)$$

That is, if $\neg MHP(s_1, s_2)$, then only one execution order of s_1 and s_2 is allowed. Conversely, however, it may be the case that $MHP(s_1, s_2)$ holds even though s_1 and s_2 can only occur in one particular execution order.

In his dissertation [65], Giffhorn describes a fairly sophisticated MHP analysis that takes into account (a) definite execution orders that can be inferred from the program's control-flow and (b) thread creation and joining.

This MHP analysis can be further improved by also taking *concurrency control mechanisms* like *locks* into account:
For example, in Java one can use *synchronization on objects' monitors*, a simple locking mechanism to achieve mutual exclusion, to ensure that for critical sections one thread at a time can be active. Threads that are about to enter such a critical section while another thread is active have to wait until the active thread is finished. This way, the possible interleavings can be restricted.
A MHP analysis that takes into account locking will consider less statements to run in unspecified order and hence be more precise.
One static analysis formalism which can model concurrency and in particular locks is *dynamic push-down networks* (DPNs) [37, 118]. Roughly, DPNs represent programs with multiple threads by a series of call stacks and are able to model dynamic thread creation, unbounded recursion and finite abstractions of thread-local and procedure-local state. Moreover, using tree automata techniques, DPNs can be used to check whether a multi-threaded program has lock-sensitive executions with given properties [63], like MHP information. Furthermore, by iterated analysis [131], DPNs can also be used to compute interference dependencies directly.
We combined DPNs and PDGs to remove interferences which in fact do not occur due to locking [80]. This is especially beneficial in situations where locking is actually used to impose definite execution orders.

4.3 Reference Scenario “Security in E-Voting”

In this section, I describe the reference scenario *Security in E-Voting*. This description is based on various RS^3 publications [5, 41, 115]. First, I give a short motivation. After that, I give an overview of the contributions of the reference scenario overall and the contributions of the programming paradigms group in particular.

4.3.1 Motivation

In recent years, more and more elections are conducted electronically. This includes national and municipal elections, as well as elections within associations, societies, and companies. There are two main categories of such systems: The first kind consists of electronic voting machines like

recording electronic voting systems and scanners that are usually installed in polling stations. The other kind is remote electronic voting systems that are used by voters to vote over the internet using their own devices like desktop computers or smart-phones.
Since elections are a critical part of democracies, it is crucial that they are held in a way that satisfies some basic properties. Two such properties are:

Privacy The system ensures that the voters' votes remain confidential.

Verifiability Voters have the possibility to check that their choices have been properly counted.

In traditional elections, these properties are usually sufficiently ensured by providing voting booths that are not observable from outside or by making the counting public.
E-voting systems also aim for such properties and ideally, the developers of E-Voting systems can be presumed to be benevolent. However, it is also true that E-Voting systems are complex hardware and software systems and as in all such systems programming errors can hardly be avoided. Verification techniques and procedures are therefore used to ensure that these systems enjoy particular security properties.
In this reference scenario, we consider the verification of privacy properties of JAVA programs that use cryptographic operations, where the final aim is to provide strong cryptographic guarantees on the code of a fully fledged remote e-voting system which is designed to provide confidentiality of the votes.

4.3.2 CVJ Framework

As a first step, Küsters, Truderung Graf and Scapin proposed the *CVJ framework* for the cryptographic analysis of JAVA programs that use cryptographic primitives [116, 114]. This framework enables existing tools that can check non-interference properties for JAVA programs, but a priori cannot deal with cryptography, to establish cryptographic indistinguishability properties at the code level.
The CVJ framework combines techniques from program analysis and *universal composability* [43, 134, 113], a well-established concept in cryptography. CVJ works in two steps. The idea of the first step, which is

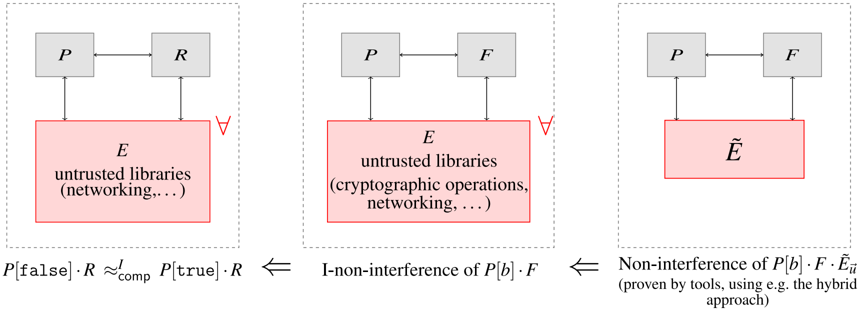

P
R
E
untrusted libraries
(networking,...)
∀
P
F
E
untrusted libraries
(cryptographic operations,
networking, ...)
∀
P
F
$\tilde{E}$
$P[\mathtt{false}] \cdot R \approx^{I}_{\mathrm{comp}} P[\mathtt{true}] \cdot R$
⇐
I-non-interference of $P[b] \cdot F$
⇐
Non-interference of $P[b] \cdot F \cdot \tilde{E}_{\vec{u}}$
(proven by tools, using e.g. the hybrid approach)

Figure 4.5: Visualization of the general approach employed to verify the security of the E-Voting system [115, Figure 1]

illustrated on the left side of Figure 4.5, is to check non-interference properties for the Java program to be analyzed where cryptographic operations such as encryption are performed within so-called *ideal functionalities*.

A given Java program $\mathcal{P}_{real}$ (the box on the left in Figure 4.5) that uses real cryptographic primitives is transformed into a Java program $\mathcal{P}_{ideal}$ (the box in the middle of Figure 4.5), where the real cryptographic primitives are replaced by idealized primitives. These idealized primitives provide guarantees in face of unbounded adversaries and can often be formulated without probabilistic operations. Therefore, they can be analyzed by tools that cannot deal with security notions specific to cryptography (probabilities, polynomially bounded adversaries). The results of the CVJ framework imply that if $\mathcal{P}_{ideal}$ is non-interferent, then the original Java program $\mathcal{P}_{real}$ (using actual cryptographic operations) enjoys strong cryptographic indistinguishability.

In addition to the reduction of a cryptographic verification task to an ordinary non-interference check, the CVJ framework also consists of a second step that is illustrated on the right side of Figure 4.5 and tackles the problem that the systems to be analyzed are often *open*: They interact, for example, with an untrusted (and unspecified) network. Analysis tools such as Joana however can only deal with *closed* Java programs, in this case the combination of the open system with one particular environment. Therefore, the CVJ framework also provides a proof technique that enables program analysis tools to verify non-interference properties of open systems. Such an open system is non-interferent if the combination of this system with any environment (which is closed) is non-interferent in

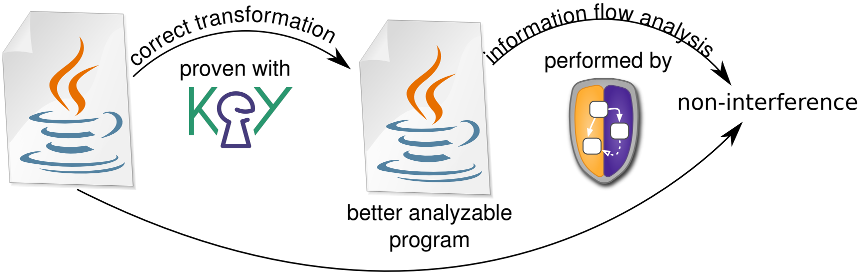


Figure 4.6: Visualization of the hybrid approach

the ordinary sense. According to the CVJ framework, it is not necessary to check ordinary non-interference for *all* environments, rather it suffices to establish non-interference for a carefully designed family of environments. These environments only differ in their inputs, so that they can be expressed as one parameterized environment (see box on right-hand side of Figure 4.5).
Graf [78] shows that this works in particular for PDG-based tools such as Joana.

4.3.3 The Hybrid Approach

In summary, the CVJ framework in principle enables a static analysis tool that can verify unrestricted non-interference to perform cryptographic analyses on programs using cryptographic primitives. However, an automatic tool like Joana cannot be sound and completely precise at the same time. Joana in particular may report *false alarms*, i.e. falsely reject a program which is actually non-interferent.
To remedy this, Küsters et al. propose the *hybrid approach* [115], which combines the strengths of automatic information-flow analysis tools with the strengths of interactive theorem provers. This was joint work of the groups of Küsters, Snelting and Beckert, to which I contributed the Joana part of the verification of the case study. In our work [115], we demonstrate the hybrid approach on a case study, a small prototypical e-voting system. In this case study, we combined Joana with KeY [11], a theorem-prover for Java.

Figure 4.6 illustrates how the hybrid approach works: The task is to verify non-interference of a given (Java-like) program and we assume that the security of this program as-is cannot be verified using a given automatic tool. The hybrid approach now works in two steps:

1. Additional code is added to the program which makes explicit to the automatic tool that the falsely reported illegal flow is in fact not present.

2. It is shown that the modifications of step 1 satisfy the requirements for a *conservative extension*. Apart from certain syntactical restrictions, this means that the essential behavior of the program has not been changed by the modification. This boils down to verifying a functional property of the given program, a task that can be performed using an interactive theorem prover.

For reference, I cite the definitions of extensions and conservativity here.

Definition 4.1 (Extension [115, Definition 1]). *Let $P = P[\vec{x}]$ be a deterministic and closed (Jinja+*[16]*) program. An* extension *of P is a program $P' = P'[\vec{x}]$ obtained from P in the following way. First, a new component M is added to P consisting of some number of classes with the following properties:*

(i) the methods and fields of the classes in M are static,

(ii) the arguments and the results of the methods of M are of primitive types,

(iii) the methods of M do not refer to classes defined in P (in particular, no methods and fields of P are used in M),

(iv) all potential exceptions are caught inside M,

(v) all methods of M always terminate.

Second, P is extended by adding statements of the following form in arbitrary places within methods of P:

[16] *Jinja* [105] is a java-like programming language which is equipped with a formal semantics to make it accessible for reasoning with a theorem prover. *Jinja*+ [116] extends Jinja by various features which are useful in the context of the CVJ framework.

(a) (output to M)

$$C.f(e_1,\dots,e_n) \tag{4.1}$$

where C is a class in M with a (static) method f and $e_1,\dots,e_n$ are expressions without side effects.

(b) (input from M)

$$r = C.f(e_1,\dots,e_n), \tag{4.2}$$

where C is a class in M, C.f is a (static) method with some (primitive) return type τ, $e_1,\dots,e_n$ are expressions as above, and r is an expression that evaluates without side effects to a reference of type τ. (Such an expression can, for example, be a variable or an expression of the form o.x, where o is an object with field x.)

Definition 4.2 (Conservative extension [115, Definition 2])**.** *An extension $P'[\vec{x}]$ of $P[\vec{x}]$ is called a* conservative extension *of $P[\vec{x}]$, if for all initial values $\vec{a}$ of high variables $\vec{x}$ the following is true in the run of $P'[\vec{a}]$: Whenever a statement of the form* (4.2) *is executed, it does not change the value of r. That is, the value of r right* before *the execution of the assignment coincides with the value returned by the method call $C.f(e_1,\dots,e_n)$. As such, statement* (4.2) *is redundant.*

Consider the example in Listing 4.2. We assume the security policy that the initial value of `secret` must not influence the final value of `pub` and that `bar` does not violate this policy. Then the program is secure: Although `secret` is added to the result of the method `foo` in line 12, it actually has no influence on it. This is because line 12 is only executed if `secret==0`, but has no effect in this case. However, a static information flow tool like Joana reports an illegal flow because it does not reason about values and assumes that the final value of `b` (and therefore the final value of `pub`) may be influenced by the initial value of `secret`. Now consider Listing 4.3. Here, the program from Listing 4.2 has been modified in such a way that Joana is able to verify non-interference: The value of `b` is saved (line 12) before it is possibly incremented (line 13) and overwritten afterwards (line line 14). Using a local killing definitions analysis [78], Joana is able to detect that `b` is indeed overwritten in line 14 and can correctly verify that the modified program is secure. According to the hybrid approach, it remains to be shown that Listing 4.3 is a conservative extension of Listing 4.2. It is easy to see that Listing 4.3 is an extension of Listing 4.2. It remains to be shown

```
class Example {
  static public int pub;
  static private int a;
  public static void main(int secret) {
    a = 42;
    bar(secret);
    int b = foo(secret);
    pub = b;
  }
  static int foo(int secret) {
    int b = a;
    if (secret==0) b+=secret;
    return b;
  }
  static void bar(int secret) {
   ...
  }
}
```

Listing 4.2: A secure program for which Joana reports a false alarm (adapted from [115], p. 309)

that this extension is indeed conservative. This boils down to proving that in line 14 the value returned by `M.get()` equals the value of `b` just before the execution of line 14. This can be done for example with an interactive theorem prover like KeY.

4.3.4 Case Study: E-Voting Machine

Within the scope of our work [115], we demonstrated the hybrid approach on a small prototypical e-voting machine, using Joana as automatic information flow control tool and KeY as a theorem prover for the subsequent proof of conservativity.

We extended the program conservatively and proved a non-interference property with Joana (and the CVJ framework). The size of the conservative extension was 934 lines of code (LoC). The non-interference property that Joana had to verify essentially says that the voters' choices have no

```
class Example {
  static public int pub;
  static private int a;
  public static void main(int secret) {
    a = 42;
    bar(secret);
    int b = foo(secret);
    pub = b;
  }
  static int foo(int secret) {
    int b = a;
    M.set(b);
    if (secret==0) b+=secret;
    b = M.get();
    return b;
  }
  static void bar(int secret) {
   ...
  }
}
class M {
       static int x;
       public static void set(int n) { x=n; }
       public static int get()    { return x; }
}
```

Listing 4.3: An extension of the program in Listing 4.2 which makes the absence of illegal information flow explicit

influence to the low output of the system[17]. Joana could verify non-interference of this program in about 18 seconds on a standard laptop (Core i5 2.5GHz, 8GB RAM). To conduct the analysis, we wrote a small driver program (about 60 LoC) which sets various configuration options of Joana, initiates the PDG construction, identifies and annotates the appropriate nodes in the PDG, and triggers the information flow analysis.

[17] For full details, I refer the interested reader to the original article [115].

Apart from the adaptions to obtain the conservative extension, the further small adaptions of the e-voting machine were necessary for Joana to verify the system. In the following, I want to elaborate a bit on these adaptions.

```
 1  for( int i=0; i<N; ++i ) {
 2  switch( actions[i] ) {
 3    case 0: // next voter votes
 4      if (voterNr<numberOfVoters) {
 5        int choice = secret ? choices0[voterNr]:choices1[voterNr];
 6        vm.collectBallot(choice);
 7        ++voterNr;
 8      }
 9      break;
10    [...]
11  }
```

Listing 4.4: A code snippet from the case study of [115] which needed to be adapted

Listing 4.4 shows a critical code snippet from the case study which needed to be changed. The code is responsible for selecting a series of votes according to a secret bit. It processes two arrays of votes and, depending on `secret`, one of these arrays is chosen to be the array of votes that is processed subsequently.

The problem for Joana is that it does not reason about values or array bounds. As a consequence, it must assume that `voterNr` may be out of the bounds of the arrays `choices0` and `choices1`. This means that both branches of the statement in line 5 may throw an `ArrayIndexOutOfBoundsException`. Hence, all the program's statements after line 5 are control-dependent on both branches of 5.

Furthermore, both branches are control-dependent on the secret bit. As a consequence, every statement which is executed after line 5, including public outputs, is dependent on `secret`, so that Joana is not able to prove any reasonable non-interference property.

However, the code snippet could be modified as shown in Listing 4.5. Here, both possible choices are loaded from the two arrays in lines 5 and 6 before the actual decision is made in line 7.

Since we assume a single-threaded environment, neither `choices0` nor `choices1` can change after lines 5/6, so that the code snippet in Listing 4.5

```
 1  for(int i=0; i<N; ++i ) {
 2  switch( actions[i] ) {
 3    case 0: // next voter votes
 4      if (voterNr<numberOfVoters) {
 5        int choice0 = choices0[voterNr];
 6        int choice1 = choices1[voterNr];
 7        int choice = secret ? choice0 : choice1;
 8        vm.collectBallot(choice);
 9        ++voterNr;
10      }
11      break;
12    [...]
13    }
14  }
```

Listing 4.5: A code snippet from Listing 4.4 with a small but critical modification

is equivalent to the one in Listing 4.4. Moreover, the fatal chain of dependencies described before is prevented: Lines 5 and 6 may still throw an `ArrayIndexOutOfBoundsException` but this is independent of `secret`. Plus, line 7 does not throw any exception.

Listing 4.6 shows another critical code snippet that needed to be adapted. The method is called only with valid values of `votersChoice`, i.e. with values between `0` and `numberOfCandidates - 1`, where the field `numberOfCandidates` coincides with `votersForCandidates.length`. Formally, an exception is thrown if `votersChoice` is outside the desired range, but this actually never happens for this program. Consequently, since `votersChoice` is within the bounds of `votesForCandidates`, the array access succeeds.

For Joana, there are two problems here. Firstly, whether the exception is thrown depends on the value `votersChoice`. This is a problem since

```
public int collectBallot(int votersChoice) throws InvalidVote {
if ( votersChoice < 0 || votersChoice >= numberOfCandidates ) {
    throw new InvalidVote();
  }
  votesForCandidates[votersChoice]++;
}
```

Listing 4.6: Another critical code snippet from the E-Voting Machine case study

`votersChoice` depends on the secret bit and JOANA does not know here that `votersChoice` is actually within bounds. Hence, as before, the program may possibly crash dependent on the secret bit, which precludes any sensible non-interference property. The other problem is very similar: JOANA assumes that the array access may fail and since `votersChoice` is considered secret, JOANA must assume that the program may crash depending on the secret value.
Our solutions to these problems were

1. remove lines 2 and 3; and

2. surround line 5 with a **`try`**`..`**`catch`**-block (with empty catch clause) which catches all `Throwables`, which effectively suppresses all possible exceptions which may occur there.

It remained to show that the method indeed never throws any exception if called with valid values of `votersChoice`. This was done during the KeY proof phase.
Apart from the two code snippets that I just discussed there were also others which prevented JOANA from showing the desired non-interference property because of possibly invalid array accesses. This motivated us to introduce a simple array analysis into the WALA framework[18]. This analysis was implemented by a student researcher under my supervision. It is based on the ABCD analysis by Bodík et al. [34] and can prove at least for simple cases that array accesses are valid and that a crash cannot happen[19]. It was integrated into WALA in 2016 [69] and JOANA can be configured to use it. Examples of what the analysis can and cannot recognize can be found on Github [67, 68].

4.3.5 Spec Slicing

Within the scope of the case study, we observed in the KeY proof part that there are situations in which a given KeY specification only covers a small part of the program or, in other words, significant parts of the given

[18] In section 3.4, I mentioned that JOANA makes heavy use of WALA for analyzing JAVA bytecode.

[19] Note that due to time constraints the analysis could not be applied to the E-Voting reference scenario.

program are irrelevant to the specification. KeY specifications tend to be very complex and detailed and their proof may require a considerable amount of manual interaction. This motivated us to develop a method to decrease the amount of proof work that needs to be performed.

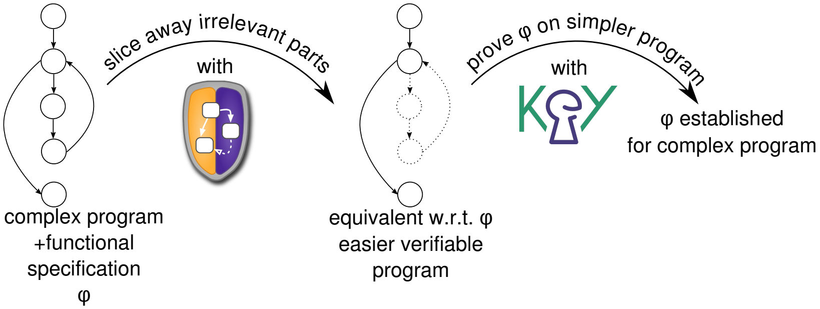


Figure 4.7: Visualization of spec slicing

We devised a general technique which we call *spec slicing* [41]. The idea behind spec slicing is illustrated in Figure 4.7: If parts of the program do not influence the final state with respect to the proof obligation, they can be safely removed and functional verification can be performed on the simpler program.

Verification of this simpler program can then be performed without any loss of precision but with possibly much less effort. The identification and removal of irrelevant program parts can be performed by an automatic tool like Joana. We already applied this technique to the E-Voting machine mentioned above: Parts of its implementation perform mere logging, which does not affect the voting result and therefore does not have any influence on the overall functional property which had to be verified. Using Joana, we gained a simpler but equivalent program without logging, which we verified using KeY to establish functional correctness for the whole program.

4.4 Reference Scenario "Software Security for Mobile Devices"

The programming paradigms group also participated in the RS^3 Reference Scenario "Software Security for Mobile Devices" (SSMD). The goal of this reference scenario was to develop an app store that offers apps with security guarantees. The reference scenario combined several security techniques developed in the scope of RS^3, including static analysis with PDGs and type systems, secure modeling and runtime enforcement. In the following, I describe the context of the reference scenario and its main result, the RS^3 certifying app store. After that, I focus on the contributions of the programming paradigms group. In the scope of this scenario, we developed JODROID, an extension of JOANA with support for the specialties of Android apps. I will describe the specifics of JODROID and will elaborate on the challenges which have already been addressed and also on the challenges which remain open until this point.

4.4.1 Motivation

In today's world, smartphones and other mobile devices are ubiquitous. They are used to store and process a wide variety of personal and sensitive data, including contacts, location, financial and health information and hence can be considered security-critical infrastructure.

The most commonly used mobile operating system is Android, with a market share of 86.6% at the end of 2019 [1] and currently over 2.8 million apps in its app store [18]. The Android ecosystem offers a number of security mechanisms, such as sandboxing of applications and a permission system restricting access to critical resources. Moreover, applications available on Google Play[20] are scanned to detect malicious behavior. At the same time, Android still has problems with security violations [162] These commonly take the form of the application revealing the user's sensitive information [133] or behaving in a way that is unexpected and harmful to the user [150].

[20] *Google Play* is Android's native app store.

This suggests that the security mechanisms employed in the Android ecosystem are not sufficient for enabling security-aware end users of Android devices to reliably enforce their personal security requirements. The RS^3 reference scenario "Software Security for Mobile Devices" aimed to offer a solution to these security problems by proposing an app store that provides user-definable security guarantees by integrating several of static and dynamic program analysis and security enforcement techniques.

4.4.2 The RS^3 Certifying App Store

The artifact of the SSMD scenario was the *RS^3 certifying app store*. Its architecture is shown in Figure 4.8. Basically, it follows a client-server architecture.

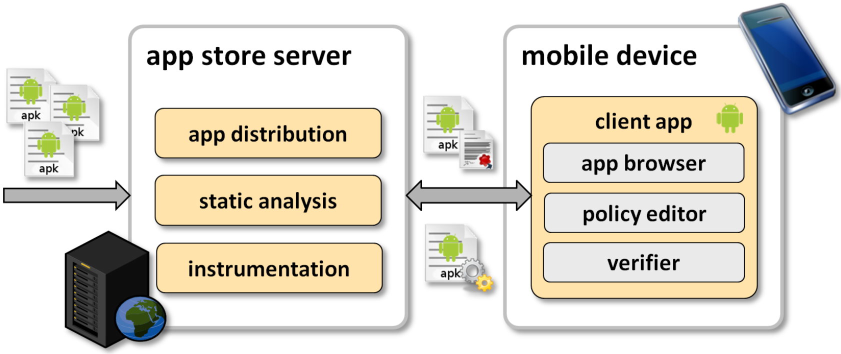
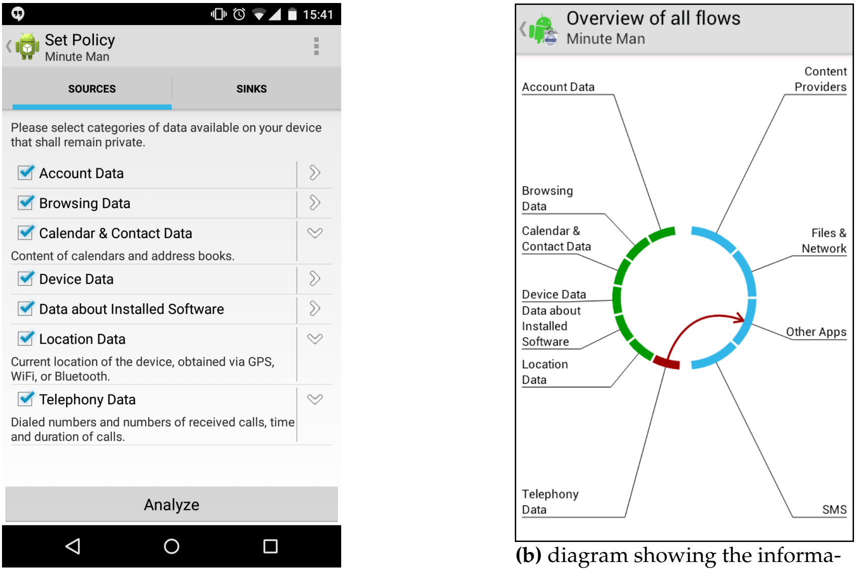


Figure 4.8: Architecture of the RS^3 certifying app store [20, Fig. 1]

The client consists of an app store app that the user can use to download the Android applications, configure information flow policies (see Figure 4.9a) and run and view the results of various information flow analyses (see Figure 4.9b).

The app store integrates the different approaches of the various groups which contributed to the reference scenario. These approaches range from static analyses like PDGs and slicing or type systems to dynamic analyses combined with runtime enforcement.

Joana, or more specifically its Android variant Jodroid, is integrated in the server component: The user can specify an information flow policy and send an analysis request to the server. The server then computes the app's PDG and checks whether the given flow policy can be verified by using a PDG-based security check. The internal format of the specification is largely similar to the *RS^3 information flow language* that I will consider in section 4.5. There, I will also explain how JOANA can be used to verify such policies.



15:41

Set Policy
Minute Man

SOURCES | SINKS

Please select categories of data available on your device that shall remain private.

- ☑ Account Data
- ☑ Browsing Data
- ☑ Calendar & Contact Data
  Content of calendars and address books.
- ☑ Device Data
- ☑ Data about Installed Software
- ☑ Location Data
  Current location of the device, obtained via GPS, WiFi, or Bluetooth.
- ☑ Telephony Data
  Dialed numbers and numbers of received calls, time and duration of calls.

Analyze

(a) client-side policy editor

(b) diagram showing the information flows of an app

Figure 4.9: Screenshots from the client-side app store app of the RS^3 certifying app store (compare [120, Figure 1(b), Figure 1(d)] and [20, Figure 2])

4.4.3 JODROID: JOANA for Android

In the following, I describe JODROID, an extension of JOANA to support Android apps. The following text is largely based on an earlier publication[123]. The initial version of JODROID has been implemented within the scope a diploma thesis under the supervision of our group [33].

The goal of extending JOANA to handle Android apps poses several challenges. Among these challenges are the following:

1. Although Android apps are developed in JAVA, they are not compiled to JAVA bytecode but to Android's own DALVIK bytecode.

2. Standard JAVA applications use a single entry point (main), but Android apps have a multitude of possible entry points which are triggered by the Android system throughout the execution of the app

3. Android apps employ *intents*, a message-passing mechanism to exchange data and start external apps' components, which requires to also analyze information flows between apps.

These challenges are not specific to Android. For example, many JAVA applications with graphical user interfaces (GUI) also have multiple entry points which handle user input. However, different frameworks require different models specifying how these entry-points are used and JOANA has no naturally built-in mechanism to specify such models[21]. Similar considerations can be made for intents: intents are comparable to other message-passing mechanisms which are commonly found in client-server-applications but currently JOANA does not provide a general mechanism which applies to a wide variety of message-passing mechanisms. We therefore consider our work on extending JOANA to Android as a starting point to addressing these more general challenges.

In the following, I present the work we have already done to address the challenges just sketched. In section subsubsection 4.4.3.1, I give a short overview of architectural aspects of Android apps, in subsubsection 4.4.3.2, I outline how we address the above mentioned challenges. After that, I conclude in subsubsection 4.4.3.3 by giving an outlook on future work.

4.4.3.1 Overview of Android Applications

In the following, I briefly discuss the architecture of Android applications. This overview is largely based on Android's API documentation [72].

An Android application usually consists of multiple, loosely coupled components. In the simplest case, these components can either be *Activities*, *Broadcast Receivers*, *Services* or *Content Providers*. I now give a quick summary of what these components do and which roles they play.

[21] This problem has been tackled in a bachelor's thesis under my supervision [19].

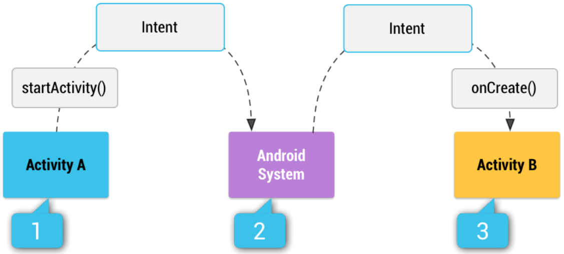


Figure 4.10: Illustration of how an implicit intent is delivered through the Android system to start another activity (see [73, Figure 1]); The sending activity [1] passes an action description, the Android system [2] selects an appropriate receiver component and starts it [3].

- *Activities:* An activity is an application component that provides a screen with which users can interact in order to do something. Each activity is given a window in which to draw its user interface.
- *Broadcast Receivers:* A broadcast receiver responds to system-wide broadcast announcements. Many broadcasts originate from the system, but can also be initiated by arbitrary app components. Broadcast receivers do not display a user interface. Typically, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.
- *Services:* A service runs in the background to perform long-running operations or to perform work for remote processes. It does not provide a user interface. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it, using a special kind of inter-process communication.
- *Content Providers:* Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

Android components use *intents* to exchange messages with each other. In particular, intents are used to start components. Intents can be *explicit*

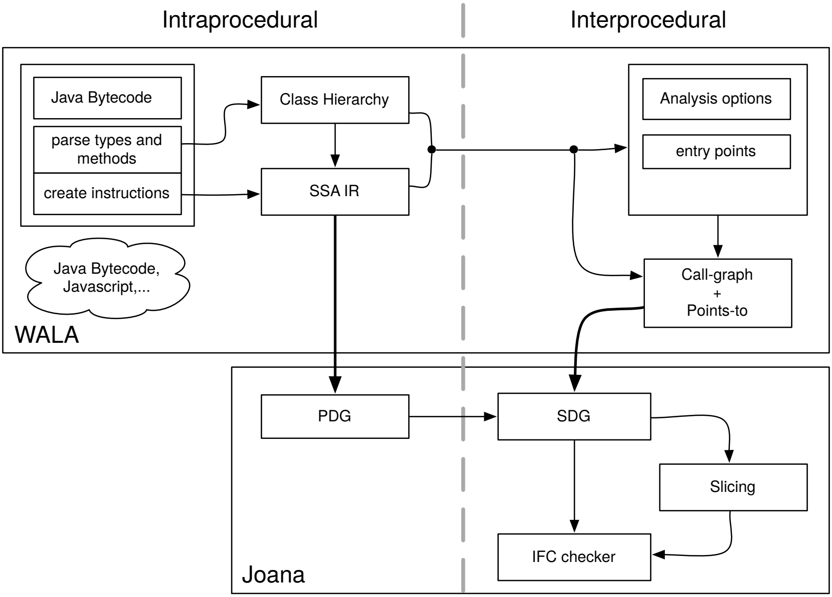


Figure 4.11: Overview of the architecture of JOANA

or *implicit*. Explicit intents specify a receiver, whereas implicit intents leave it up to the Android system and/or the user to resolve their receiver. Figure 4.10 (taken from Android's API documentation [73]) shows how Android processes an implicit intent.

4.4.3.2 Approach

In this section, I explain how we address the challenges I mentioned in subsection 4.4.3.

DALVIK front-end Figure 4.11 shows the general architecture of JOANA. As can be seen in Figure 4.11, the PDG builder of JOANA only depends on WALA's analysis results and hence is decoupled from WALA's front-end. As a consequence, we only needed to adapt WALA's front end to be able to process Android apps. For this, we integrated the WALA front end code of SCanDroid [62], a security analysis tool which is also based on WALA. This was a first step to extend JOANA to handle also Android apps.

Life cycle modelling. Classic JAVA applications have a single entry point and every execution of the application starts with an invocation of this method. Clearly, this assumption is not met by Android apps: As we already mentioned, Android apps have multiple callbacks, which may be triggered by the user or the Android system as a reaction to certain events. However, the order and the way these callbacks are triggered is not arbitrary, but follows certain rules. More specifically, the components of an Android app are driven by their *life cycles*. The life cycle of an activity can be seen in Figure 4.12.

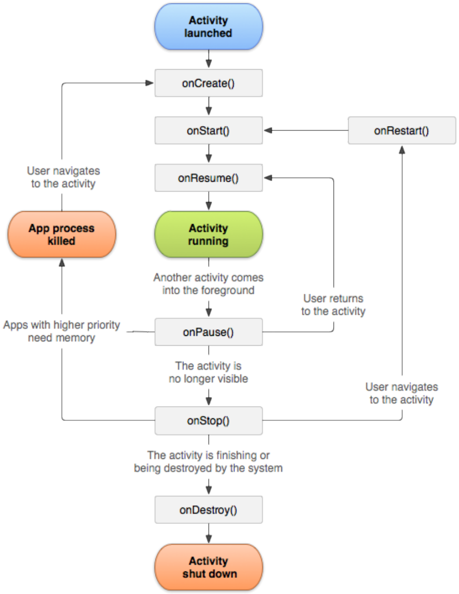

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 4.12: Lifecycle of an activity (see [74, Figure 1])

It is not an option to run a separate analysis for each entry point, since there may be information flows which only occur if multiple callbacks are executed in sequence. Listing 4.7 (adapted version of a sample from DroidBench [61]) presents an example.

```
 1 public class MyActivity extends Activity {
 2   static String addr =
 3   "http://www.google.de/search?q=";
 4   void onCreate(Bundle savedInstanceState) {
 5     telephonyManager = (TelephonyManager)
 6     getSystemService(
 7       Context.TELEPHONY_SERVICE
 8     );
 9     /** retrieve secret data of telephone (source) */
10     imei = telephonyManager.getDeviceId();
11     /** extend request by secret data */
12     addr = URL.concat(imei);
13   }
14   void onStart() {
15     super.onStart();
16       try{
17         url = new URL(addr);
18         conn = (HttpURLConnection) url
19                                   .openConnection();
20         conn.setRequestMethod("GET");
21         conn.setDoInput(true);
22         /** send request to network (sink) */
23         conn.connect();
24       } catch(Exception ex){}
25     }
26 }
```

Listing 4.7: Example for an information flow across entry points

When `onCreate()` is executed, line 10 reads the IMEI of the phone and later, upon the invocation of `onStart()`, line 23 sends the IMEI to a server on the internet. However, such an information flow would not be detected if `onStart()` and `onCreate()` were each analyzed in isolation, since neither calls the other but both are called by the Android framework.

To cover such flows as well, our approach synthesizes an entry method that simulates the Android framework by invoking all callbacks of the given app.

```
public class MyActivity extends Activity {
    static String URL=
            "http://www.google.de/search?q=";
    void onCreate(Bundle savedInstanceState){
        ...
        conn.connect();
        ...
    }
    void onStart() {
        ...
        imei = telephonyManager.getDeviceId();
        URL = URL.concat(imei);
    }
}
```

Listing 4.8: An example in which there is no information flow between entry points

In order to lose not too much precision, we take the life cycles of the app’s components into account. Consider again Figure 4.12: When `onCreate()` is called, either the activity has just been launched, or the app’s process has been destroyed and re-created. In either case, `onCreate()` is called on a fresh heap which cannot have been influenced by any other of the activity’s entry points. Thus, it is safe to assume that none of the activity’s entry points is called before `onCreate`. An example of how this assumption can be exploited to rule out impossible information flows and thus leads to increased precision is shown in Listing 4.8: In this variant of Listing 4.7, the source is contained in `onStart()` and the sink is contained in `onCreate()`. Since `onCreate` is never executed after `onStart`, the sink cannot be influenced by the source.

Intents. Our model also provides basic support for intents.
In order to incorporate the intents an application may react to, the application’s manifest is inspected, the possible intent targets are resolved and appropriate method calls are inserted into the artificial entry method. Similarly, our approach handles intents which may be issued during the execution of the application and whose target can be resolved to a component within the same application. Listing 4.9 shows an example of an activity

ShareActivity which declares the kinds of intents that it reacts to. It does so by using an *intent filter*. An intent filter specifies a set of possible intents an activity may react to. There are three aspects that can be used to specify intents: actions, categories and data. For example, in Listing 4.9 the activity ShareActivity can react to intents which specify android.intent.action.SEND as action, belong to the category android.intent.category.DEFAULT and send data of type text/plain. Note that in general, an intent filter may declare multiple action, category and data items. If an intent filter declares multiple action items, it matches all intents that match at least one of the declared action items. The same rule applies to categories and data. In Listing 4.10, we see exemplary code which issues an intent matching the intent filter depicted in Listing 4.9. Jodroid handles such code in the following way: It analyzes the app's manifest and records for each activity the possible intents it may react to.

Now suppose that during call graph construction, a piece of code like Listing 4.10 is encountered. Jodroid then inspects the object passed as parameter in the call in line 9. If the action can be resolved statically and matches the intent filter of a given activity, the call is interpreted as a call to the onCreate method of that activity.

This analysis can be improved by a static approximation of strings. Such an approximation was implemented in the scope of a bachelor's thesis under my supervision [167] and can be integrated into Jodroid.

```
<activity android:name="ShareActivity">
  <intent-filter>
  <action android:name="android.intent.action.SEND"/>
  <category
    android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

Listing 4.9: An exemplary section of an app's manifest where a component declares that it reacts to certain intents – taken and adapted from the Android documentation [73]

```
public void foo() {
 Intent s = new Intent();
 s.setAction(Intent.ACTION_SEND);
 s.putExtra(Intent.EXTRA_TEXT, "secret");
 s.setType("text/plain");
 // Verify that the intent will resolve to an
 //activity
 if (s.resolveActivity(getPackageManager()) != null) {
  startActivity(s);
 }
}
```

Listing 4.10: Example of how to invoke an activity using an implicit intent – taken and adapted from [73]

4.4.3.3 Limitations and Future Work

Now, I elaborate on the work that is left to do.

At the moment, Jodroid cannot handle callbacks of graphical user interfaces. The graphical user interfaces of Android apps are typically described in separate files and these files also reference the callbacks which are invoked on user input, e.g. when a button is pressed.

Hence, to also cover GUI callbacks, they have to be extracted from the separate files and integrated into the artificial main method appropriately. Another current limitation is that Jodroid only analyzes information flows inside single apps, but no information flows between different apps. This could be achieved by simply analyzing all the apps simultaneously. However, such an analysis would have to be re-done each time an app is added. Additionally, the analysis would have to be adapted in order not to assume that all the apps under analysis share a single heap (normally, different apps run in different virtual machines and hence have separate heaps).

An alternative, more modular approach is outlined in Figure 4.13: First, the intra-app flows in each single application are analyzed and summaries are generated from these analysis results. After that, a *communication graph* is built by connecting one app's summary with another app's summary if one of the former app's components may trigger one of the latter app's components by issuing an intent. The paths of such a graph represent the possible information flows between the apps.

4.5 RIFL

In this section, I report on RIFL ("RS^3 Information Flow Language") [57, 25], a joint effort of multiple researchers within the RS^3 project. The goal was to develop a language in which security requirements can be expressed. One main design goal for RIFL was to be tool-independent, i.e. not be tailored to a specific information-flow analysis. Tool-independence enables to create case studies that are suitable for multiple tools, so that multiple tools can be evaluated, compared and possibly even combined. As a consequence of its tool-independence, RIFL is a semi-formal language: It has a formally defined syntax with a specific intuition behind it, but no fixed security semantics.
In the following, I describe the syntax of RIFL and the intended meaning of a RIFL specification. For this, I will base on the technical report on RIFL 1.1 [25], to which I also contributed. Furthermore, I will report on JOANA's

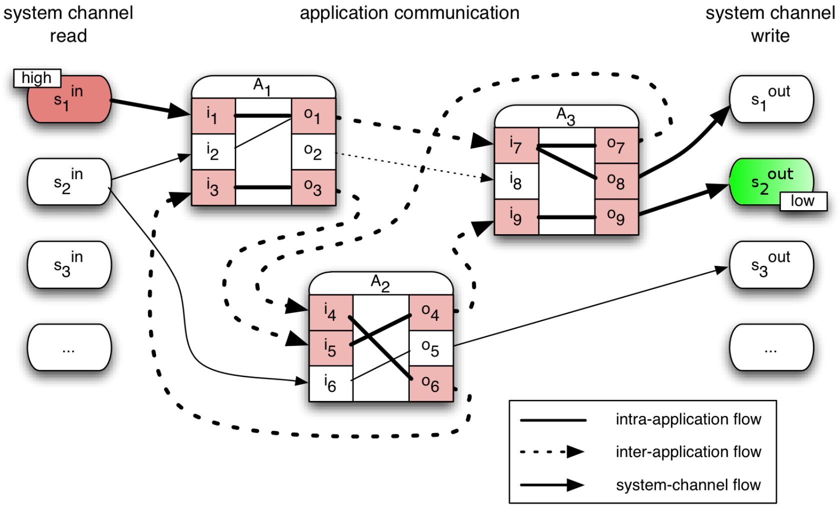


Figure 4.13: Possible approach to capture inter-app flows

support for RIFL and in particular will describe how a RIFL specification maps to an information flow query for Joana.
As an application of RIFL, we also developed a benchmark suite for information flow analysis tools. I will describe this benchmark suite and some results from it in section 4.6.

4.5.1 Description of RIFL

In this subsection, I describe the syntax of RIFL and shed some light on the intended meanings behind it. To keep the description brief, I will refrain from showing the syntax explicitly but rather show examples on which I explain the different elements. For further information, I refer the interested reader to the technical report on RIFL 1.1 [25], which contains all details.
Intuitively, a security requirement specification describes the allowed information flows within a program. Such a description usually consists of

- information *sources* / *sinks*, i.e. locations at which the program imports / exports information,
- a set of domains together with a *flow relation* that specifies between which domains information is allowed to flow, and
- a domain assignment which maps each source and sink to a domain.

A RIFL specification roughly consists of these ingredients. The syntax of RIFL is XML-based and a Document Type Definition (DTD) is provided.
In order to enable re-use, RIFL is separated into a language-independent part, which can be re-used for each concrete supported programming language and a language-dependent part, which provides the respective specifics for the supported languages. Currently,< the supported languages are Java Source Code (JSC), Java Bytecode (JBC) and Dalvik Bytecode (DBC). Joana and its Android-variant Jodroid support both Java Bytecode and Dalvik Bytecode. In my descriptions, I will focus on Java Bytecode. The DBC front-end is syntactically identical to JBC and the JSC front-end only differs in the notation of method and field signatures.
In the following, I will describe the different parts of a RIFL specification. This description is based on the technical report on RIFL 1.1 [25].

4.5.1.1 Interface Specification

The interface specification declares where a program imports and exports information. It specifies where in the program code a program reads input from the environment (*sources*) and where it provides output (*sinks*) to the environment.

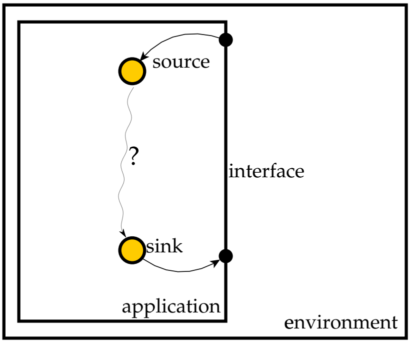


Figure 4.14: RIFL's program model

.

RIFL also provides a grouping mechanism. Sources and sinks can be grouped into *categories* and categories can be organized in a hierarchy: Categories may either contain sources or sinks or further categories.
A rough sketch of RIFL's program model can be seen in Figure 4.14. A program is basically regarded as a black-box which interacts with its environment (for example, the operating system) through a well-defined interface. This interface can be used to get data from outside (source) or provide output to the environment (sink). A RIFL specification describes exactly the parts of the environment which are used as sources and sinks for the program.
An interface specification consists of multiple *assignables*. An assignable has an identifier which is called a *handle* and contains either a *source*, a *sink* or a *category*. The sources and sinks themselves are language-dependent since they refer to explicit locations in the program's code and are explained later. A category has an identifier, its *name*, and may contain arbitrarily many sources, sinks or further categories.

It may appear redundant to have identifiers both for assignables and categories. But this has the simple reason that an assignable may consist of a single source or sink. Sources and sinks themselves do not have an identifier. Instead, their containing assignable provides one through its handle. The identifiers of categories and assignables are important since they are used for referring from other parts of a RIFL specification.

Figure 4.15 shows the logical structure of an exemplary interface specification. The corresponding RIFL representation can be found Listing 4.11. The sources and sinks in this example are simplified. Their concrete structure will be discussed later.
The specification declares the three handles `fileshandle`, `HTTPhandle` and `HTTPShandle`. `HTTPhandle` and `HTTPShandle` each consist of one sink (`sendViaHTTP` and `sendViaHTTPS`, respectively). The third handle `fileshandle` consists of the single category `files` which contains one bare sink `storeToTmpFile` and two sub-categories `file-sources` and `file-sinks`, in which the source `loadFromFile` and the sink `storeToFile` are located.

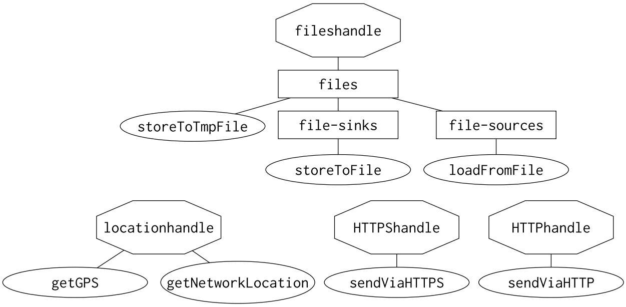


Figure 4.15: Logical structure of an exemplary interface specification – the actual RIFL specification snippet can be found in Listing 4.11 – handles are represented by octagons, categories by rectangles and sources/sinks by ovals

```
<interfacespec>
  <assignable handle=”locationhandle”>
    <category name=”location”>
      <source name=”getGPS” />
      <source name=”getNetworkLocation” />
    </category>
  </assignable>
  <assignable handle=”fileshandle”>
    <category name=”files”>
      <category name=”file-sources”>
        <source name=”loadFromFile”/>
      </category>
      <category name=”file-sinks”>
        <sink name=”storeToFile” />
      </category>
      <sink name=”storeToTmpFile”/>
    </category>
  </assignable>
  <assignable handle=”HTTPhandle”>
    <sink name=”sendViaHTTP” />
  </assignable>
  <assignable handle=”HTTPShandle”>
    <sink name=”sendViaHTTPS” />
  </assignable>
</interfacespec>
```

Listing 4.11: RIFL representation of the exemplary interface specification from Figure 4.15

4.5.1.2 Security Domains and Flow Relation.

As usual in the information-flow world, RIFL uses *security domains* to model different levels of confidentiality. A *flow relation* $\rightsquigarrow \subseteq D \times D$ over the set D of security domains is used to describe the allowed flows. Formally, a flow between security domains d_1 and d_2 is allowed according to the given RIFL specification iff $d_1 \rightsquigarrow d_2$.

In RIFL, both the security domains and the flow relation are specified by declaring mere lists. RIFL aims to make as much explicit as possible. All pairs of domains which are related via $\rightsquigarrow$ have to be listed explicitly. All

domains occurring in the flow relation have to be declared in the `<domain>`-section. The only implicit assumption made about the flow relation is that it is reflexive. Consequently, transitive relations like lattices may lead to a considerable rise of verbosity in a RIFL specification.
The specifications shown in Listing 4.12 describe a diamond lattice: On the left-hand side, four security domains are declared, whereas the right-hand side specifies the diamond lattice structure on them.

```
<domains>                      <flowrelation>
  <domain name="low" />          <flow from="low" to="mid1"/>
  <domain name="mid1" />         <flow from="low" to="mid2"/>
  <domain name="mid2" />         <flow from="low" to="high"/>
  <domain name="high" />         <flow from="mid1" to="high"/>
</domains>                       <flow from="mid2" to="high"/>
                               </flowrelation>
```

Listing 4.12: Specification of security domains and a flow relation in RIFL

Since RIFL makes the implicit assumption that the specified flow relation is reflexive, declarations such as

```
<flow from="low" to="low"/>
```

need not be declared. However, it is neccessary to declare

```
<flow from="low" to="high"/>
```

since flow relations in RIFL do not need to be transitive.

4.5.1.3 Escape Hatches.

For the sake of completeness, I mention here that RIFL also supports a form of *declassification* [144], namely *what-declassification*. This is realized by the usage of *escape hatches* [143]. An escape hatch specifies that certain information may be declassified to a given security domain.
Joana also supports a kind of declassification, *where*-declassification [87]. It does however not support the what-declassification mechanism implemented by RIFL. Joana does not reject a RIFL specification containing escape hatches, it rather ignores the escape hatches and hence treats such

a specification as if they were not there. Consequently, it checks the compliance of the given program with a stricter policy, which does not hurt soundness.

For details on how escape hatches work in RIFL, I refer the interested reader to the technical report of RIFL 1.1 [25].

4.5.1.4 Domain Assignment.

RIFL's `domainassignment` describes a mapping of the declared sources and sinks to the declared security domains. It employs the handles of declared `assignables` to refer to the sources and sinks declared within that `assignable`. In particular, if the `assignable` contains a `category`, then all sources and sinks declared within that category (directly or indirectly) are referred to by the handle of the `assignable`.

Listing 4.13 shows an example for a domain assignment, assuming the flow relation from Listing 4.12 and the interface specification of Listing 4.11. The third `assign` declaration refers to the `assignable` with the handle `fileshandle` and hence assigns the security domain `low` to all sources and sinks contained in `fileshandle`, namely `storeToTmpFile`, `storeToFile` and `loadFromFile`.

```
<domainassignment>
  <assign handle=”locationhandle” domain=”high” />
  <assign handle=”HTTPShandle” domain=”high” />
  <assign handle=”fileshandle” domain=”low” />
  <assign handle=”HTTPhandle” domain=”low” />
</domainassignment>
```

Listing 4.13: Exemplary domain assignment in RIFL

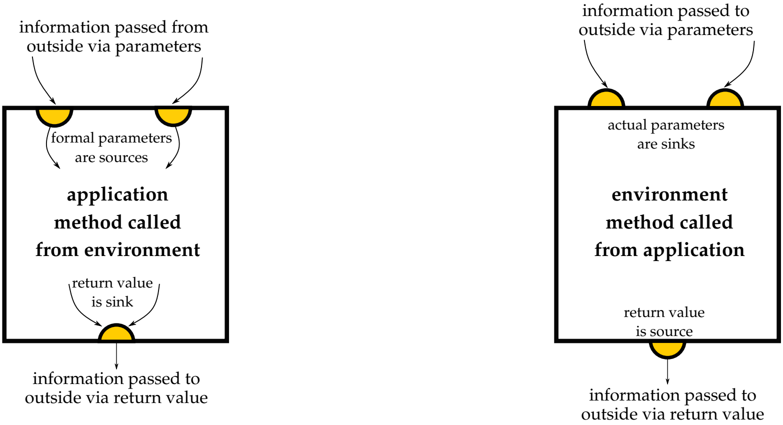
4.5.1.5 Sources and Sinks

Sources and sinks in RIFL are language-specific, i.e. which kinds of sources and sinks are available and the concrete syntax depend on the concrete programming language. In RIFL, there are specializations for Java Source Code, Java Bytecode and Dalvik Bytecode. In the following, I will focus on Java and Dalvik Bytecode since the JBC and the JSC front-ends of

RIFL support the same kinds of sources and sinks and only differ in the syntax of method and field identifiers. Note that the JAVA Bytecode and DALVIK Bytecode front-ends are syntactically identical, therefore I will only consider the JAVA Bytecode front-end.

4.5.1.6 Method Parameters and Return Values.

In Figure 4.16, we see two views on Java methods that may be employed when thinking about their role in specifying sources and sinks.



information passed from outside via parameters
formal parameters are sources
application method called from environment
return value is sink
information passed to outside via return value
information passed to outside via parameters
actual parameters are sinks
environment method called from application
return value is source
information passed to outside via return value

Figure 4.16: Two views on methods

On the left, we see an *internal view*. This is the view which is used for application-internal methods which are called from the environment. Examples for this include the `main` method of a simple JAVA application or any callback of an Android app. When such an application-internal method is called from the environment, its parameters can be used to pass information from the environment to the application. In other words, using the intuitions of RIFL, any parameter of an application-internal method can be considered a source. Conversely, when the application-internal method finishes, it passes its return value to the environment. In other words, the return value of an application-internal method can be considered a sink.

On the right-hand side of Figure 4.16, we see the external view on a method. This view is to be employed for external environment methods which are called from the application. Examples for such usage of library methods include writing into or reading from files. Here, the application uses the parameters of the external methods to pass information to the environment (for example, the next line of text to be written) and the return value to import information from the environment (the next line from a text file). Since both views are valid in their respective context, method parameters and return values can be both specified as sources and sinks. Table 4.1 summarizes the different usages of method parameters and return values as sources and sinks.

	source	sink
application method called from outside	parameters	return value
call of environment method	return value	parameters

Table 4.1: Method parameters and return values as sources or sinks

4.5.1.7 Heap Locations.

A Java application may exchange information with the environment not only through method parameters and return values but also through the heap. RIFL supports the specification of the following kinds of heap locations:

Object fields / static fields Fields of objects and static fields can be specified both as sources and sinks. The specification of an object field is to be understood in an object-insensitive way. That means that if the object field `f` of the class `C` is specified as a source, then `o.f` is considered a source for all instances `o` of class `C`. Static fields are specified in the same way as object fields.

Content and length of arrays Arrays are treated like objects with two special fields `content` and `length`. That is, it is possible to e.g. specify the contents of every `int[]` array as source or sink.

Fields of objects received as parameters of methods If an application method receives its parameters from the outside through a parameter of non-primitive type (i.e. an object), it may be unsuitable to treat the parameter itself as a source since it may only be a reference to heap locations which contain the actual information. Therefore, RIFL provides the possibility to not only specify parameters but also reachable fields as sources. Note that, until version 1.1, RIFL does not allow to specify fields reachable from parameters of external method calls or from return values of internal or external methods.

4.5.1.8 Exceptions.

In Java, exceptions constitute an implicit channel of information. This especially applies to the communication between an application and its environment through internal or external methods. Therefore, RIFL makes it possible to specify exceptions as sources and/or sinks. The intuition is that a method not only returns its ordinary return value but also whether it has terminated abnormally and also the type of the occurred exception. Applying the intuition expressed in Figure 4.16, RIFL considers exceptions as sources for external methods and as sinks for internal methods. This enables to express for example that a given parameter may not influence whether a given application-internal method terminates abnormally.

4.5.2 JOANA's Support for RIFL

In the following, I will explain how JOANA's RIFL front-end works. In particular I will carry out how a RIFL specification is interpreted by JOANA and how the translation of sources and sinks is performed. I will, as in most parts of this thesis, consider only sequential programs.
JOANA implements RIFL 1.1 in most parts. RIFL's declassification is not supported since JOANA only supports a form of *where-declassification* [86] but no *what-declassification*. Furthermore, JOANA currently does not support the specification of an array's length as source or sinks. However, this has only implementation reasons and should be fixable with reasonable effort.
The major part of JOANA's RIFL front-end consists of translating a RIFL specification $\mathcal{S} = (D, \rightsquigarrow, Src, Snk, dom)$ into queries answerable by JOANA. The objective is to check whether the RIFL specification $\mathcal{S}$ is satisfied. Intuitively, a source s may influence a sink t only if information classified as $dom(s)$ is allowed to influence information classified as t. More formally, this can be expressed as

Algorithm 7: Routine for checking a RIFL policy

```
Input: a program $p$, a set $Src$ of sources, a set $Snk$ of sinks, a RIFL
       specification with flow relation $\rightsquigarrow$
Result: whether one of the given sources may influence one of
        the given sinks although it must not according to $\rightsquigarrow$
foreach $s \in Src$ do
    foreach $t \in Snk$ do
        if $dom(s) \not\rightsquigarrow dom(t) \wedge s$ possibly influences $t$ then
            return false
return true
```

$$\forall s \in Src.\ \forall t \in Snk.\ s \text{ possibly influences } t \implies dom(s) \rightsquigarrow dom(t).$$

In RIFL, sources and sinks lie at the boundary between the application and the environment. At a source, information enters the application from the environment and at a sink it leaves the application to the environment. With ordinary sources and sinks, it is not possible in RIFL to specify sources or sinks which completely lie within the application. Furthermore, once information is outside the application, it cannot be tracked by Joana anymore. So if it leaves the application through a sink and immediately enters it again unmodified through a source, it is treated like a fresh piece of information with no connection to the piece of information that left the application just before.
But if there can be no intermediate steps within the application, it is sufficient to consider each pair (s, t) of sources and sinks individually. A check routine is shown in Algorithm 7. It receives a program and a RIFL specification and returns whether it can be verified that the program satisfies the RIFL specification. If the assigned domains of s and t are related, no checking is neccessary: Even if there were an information flow between s and t, that would be permitted since their security domains relate. Hence, an actual check is performed just for those (s, t), where $s \not\rightsquigarrow t$.
It remains to implement the check whether a source *s possibly influences* a sink t using Joana. This is done in a two-step process:

1. Translate s into a set N_{Src} of PDG nodes and t into a set N_{Snk} of PDG nodes.

2. s cannot influence t if

$$BS(N_{Snk}) \cap N_{Src} = \emptyset \tag{4.3}$$

Wasserrab has shown [164, Theorem 6.1] that a check like the one expressed by (4.3) is sufficient for guaranteeing non-interference. Note that such a check can be replaced by other more sophisticated checks like RLSOD.
In the special case that the flow relation $\rightsquigarrow$ on D forms a lattice, we can also perform a single instance of Hammer's IFC check [86] instead of performing an individual check for each source-sink-pair.
In the following, I describe how the first step of the process sketched above is performed, namely how RIFL sources and sinks are translated to PDG nodes.

4.5.2.1 Mapping of method parameters, return values and exceptional return values

Joana's interprocedural PDG representation has special-purpose nodes for method parameters and the return values, both at the caller's and the callee's side. Hence, these kinds of RIFL sources and sinks can be mapped directly to PDG nodes. The specifics are summarized in Table 4.2. By using the term "root parameter", I acknowledge the fact that Joana not only has parameter nodes for the parameters themselves but also for fields reachable from parameters which are read or written within the method, either directly by the method itself or indirectly by called methods. Every parameter node represents a sets of heap locations which may be modified or read. Parameter nodes are connected via *parameter structure edges*: p is connected to q via a parameter structure edge if a heap location represented by q may be obtained by dereferencing one of p's heap locations using a field access operation. More details about this can be found in the PhD thesis of Graf [78].

4.5.2.2 Mapping of static fields and object fields

Static and object fields do not have a single counterpart in Joana's PDG representation. Instead they are mapped to all corresponding heap access

	source	sink
application method called from outside	formal-in node for the (root) parameter	formal-out node for the return value
call of environment method	actual-out node for the return value	actual-in node for the (root) parameter

Table 4.2: Method parameters and return values as sources or sinks on JOANA's layer

	source	sink
non-static field	all non-static heap reads	all non-static heap writes
static field	static heap reads	static heap writes

Table 4.3: Static and non-static fields as sources or sinks on JOANA's layer

operations. The specifics are summarized in Table 4.3. JOANA retains sufficient information in its PDG for identifying all reading or writing accesses of a given field and also whether these accesses are static or not. For example, if an object field `A.f` is specified as a source, then all non-static heap read operations on `A.f` are located.

How such a field read operation is represented in the PDG can be seen in Figure 4.17. One of the nodes in this structure represents the actual access to the heap (highlighted in blue). This node is selected as a source.

Accordingly, if an object field `A.f` is specified as a sink, then first all write operations to this field are located and the actual field node in the corresponding PDG structure (see Figure 4.18) is selected.

Static fields are handled analogously.

4.5.2.3 Mapping of arrays.

RIFL allows for the specification of the content and length of arrays as sources and sinks. Since JOANA's RIFL front-end does not support lengths of arrays as sources or sinks, I only consider the contents of an array.

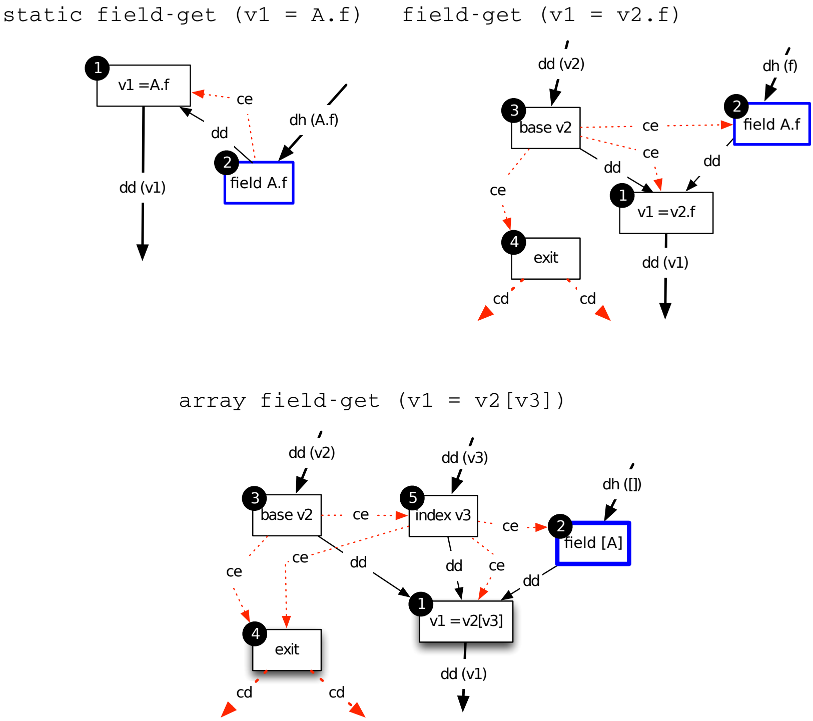


Figure 4.17: JOANA's PDG node structures corresponding to the various heap read operations (taken and adapted from [78, Figure 2.31]) – the node to which a particular kind of source is mapped is highlighted in blue.

JOANA models arrays as classes with exactly one field for the contents of the array. Individual array cells are not distinguished. Hence, array contents can be handled analogously to object fields. The only difference is that one has to be coarser when selecting the appropriate instructions: RIFL only distinguishes arrays by element type. For example, it can be specified that all **`int`** arrays are sources. In such a case, all reads on **`int`** arrays are located and for each of them the actual content access node is selected (see the bottom of Figure 4.17 and Figure 4.18, respectively).

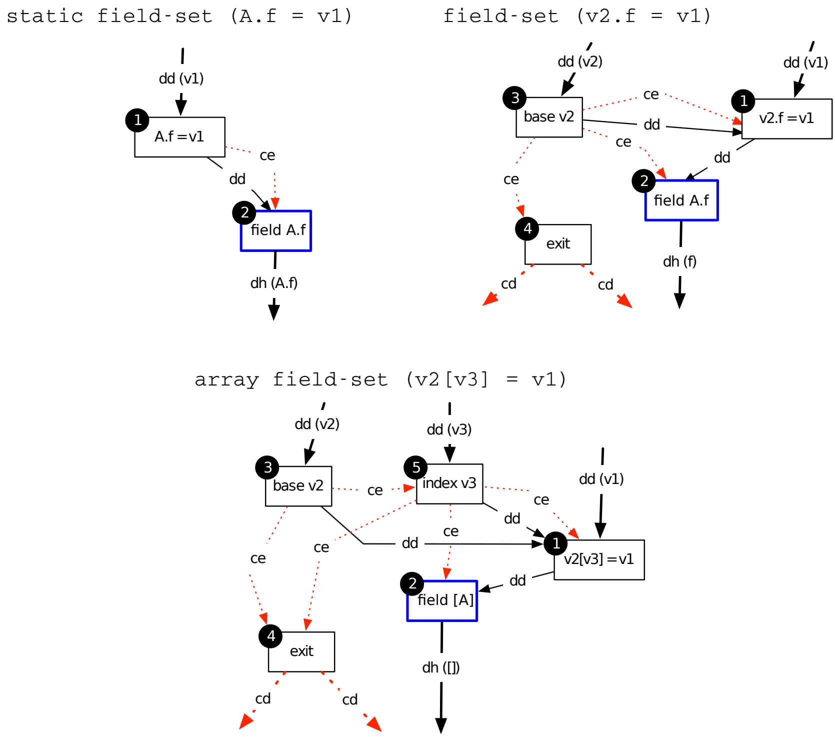


Figure 4.18: JOANA's PDG node structures corresponding to the various heap write operations (taken and adapted from [78, Figure 2.32]) – the node to which a particular kind of sink is mapped is highlighted in blue.

4.6 IFSPEC: An Information-Flow Security Benchmark Suite

In the scope of our work on RIFL, we developed IFSPEC, a benchmark suite for information flow analysis tools. This was joint work with RS^3 researchers from the groups of Mantel at TU Darmstadt and Beckert at KIT, with support from many other RS^3 researchers. In the following, I will describe the motivation behind and the structure of IFSPEC. Furthermore, I will show some of the results we have produced using IFSPEC. The following text is based on the resulting publication [85], to which I contributed JOANA support.

Benchmark suites exist for various areas of computer science, like compiler research, SAT/SMT solving, theorem proving or model checking. They allow for the evaluation of a tool or technique with respect to different quality metrics like performance, correctness, precision or completeness. With such evaluable quality metrics, it is possible to compare different tools. Hence, benchmark suites can be regarded as driving forces of innovation and technical progress.

In a benchmark suite that assesses some form of correctness, samples should contain some kind of specification of the expected behavior of the benchmarked tool. For example, benchmark suites for compilers usually consist of sample programs to be compiled together with several test cases for the correctness of the compiled programs.If the compiled program fails one of these test cases, there is evidence that the compiler does not behave correctly. Hence, it is crucial to have such test cases in order for benchmark suites that are evaluated with respect to correctness.

Ideally, the expectation is specified *formally*, so that it can be read and processed automatically. For instance, the SPEC compiler benchmark suites contain test drivers, which execute the compiled samples and compare their actual outputs with expected values.

Formal specifications of expectations can also be found in benchmark suites for SMT solving. Here the expectation concerns for example the satisfiability of a given instance. SMT-Lib requires benchmark instances to state their solvability in the metadata [24, §3.9.3].

In the area of information-flow security, we found two benchmark suites: SecuriBench [148] and DroidBench [21, 61]. SecuriBench consists of several web applications found "in the wild" with known vulnerabilities. Later, SecuriBench Micro [149] was distilled from SecuriBench. SecuriBench Micro consists of 122 very small web applications, each of which focuses on a small set of vulnerabilities.

Although SecuriBench and SecuriBench Micro were developed to benchmark tools for the analysis of vulnerabilities in web applications, they are also suitable for information-flow analysis tools since web vulnerabilities can also be interpreted from an information-flow security perspective. Indeed, SecuriBench Micro has been used to evaluate information-flow analyzers targeting JAVA (e.g. [168]).

Arzt et al. [21] presented *DroidBench* as a benchmark suite for comparing their taint-analysis tool FlowDroid with existing tools for Android. The original version consists of 35 small Android apps, each of which focuses on some feature of Android. Of these 35 apps, 25 are insecure and 10 are secure (according to an intuitive specification). Later, DroidBench was extended considerably: The current DroidBench 2.0 consists of 119 apps, 99 of which are insecure and 20 of which are secure.

For a benchmark suite in information-flow security, such a formal specification consists of two parts: A formal specification of the security requirements for the given sample and the *ground truth*, a short information whether the given program satisfies this specification. Then, an information-flow can be evaluated automatically as follows: First, it is fed with the program and the requirements specification and performs its security analysis. Then it outputs whether it deems the given program secure or insecure with respect to the given specification. This output is then compared with the expected output.

Neither SecuriBench Micro nor DroidBench provides a machine-readable specification of the information-flow requirements. Instead, they provide some hints in the comments of their samples.

With ifspec, we provide a collection of samples together with a machine-readable specification so that tools can be evaluated and compared automatically. We use RIFL to provide formal security requirements and a short text file to indicate whether the program satisfies these formally specified security requirements or not.

In detail, each sample consists of the following data:

Sample Kernel The *sample kernel* comprises the program itself together a RIFL specification and a ground truth. The RIFL specification describes formally what it means for this program to be secure. The ground truth specifies whether the given program is expected to satisfy the RIFL specification.

Sample Meta-Information The *sample meta-information* provides further descriptions of the sample. For one, it associates the sample with a number of *tags*, which serve as a categorization mechanism for the samples. Another meta-information is the minimal RIFL version that a benchmarked tool must support to parse the RIFL specification. Lastly, since RIFL has no formal semantics, each sample needs to provide some *security semantics* that have been considered when classifying a sample as secure or insecure. For the classification of the samples in IFSPEC, we consider four formal security properties: termination-insensitive non-interference for the Abstract DALVIK Language (TIN-ADL) [121], sequential and probabilistic non-interference (SN/PN) [31], and the flow*-predicate [27]. These are sufficient for JOANA, JODROID, Cassandra and KeY, the four tools considered in this work. The security property TIN-ADL is enforced by Cassandra[121], SN/PN is enforced by JOANA as well as JODROID for sequential (resp. concurrent) programs, and the flow*-predicate is enforced by KeY[27].

Sample Interpretation The purpose of the *sample interpretation* is to provide convincing arguments that the sample kernel is meaningful. It consists of three parts:

- a *description* of the program itself,
- a description of the intuitive *security requirements* for this program, i.e. what it means intuitively for this program to be secure, and
- a *faithfullness argument* that the RIFL specification matches the intuitive security requirements.

The core of IFSPEC consists of 80 samples. The core samples have been provided by RS^3 researchers over the course of several years. Apart from that, IFSPEC also comprises the 122 original samples of SecuriBench Micro and 30 additional samples that were derived from them.

As an extension, IFSPEC incorporates a machine-processable version of DroidBench 2.0. A second extension comprises examples whose RIFL specification make use of declassification.

To demonstrate the usefulness of IFSPEC, we ran four information-flow tools on its samples and reported and discussed our findings.

In the following, I summarize these discussions, with a focus on JOANA's and JODROID's results.

We use terms that are commonly used to assess properties of classification tools. In the following, I introduce these terms. An information-flow tool that processes a sample produces two possible *analysis results*: Either it considers the sample secure or it considers it insecure. We refer to the former as a *positive* result (as in *the analysis could not find a potential information flow*) and to the latter as a *negative* result (as in *the analysis found a potential information flow*).

ground truth	analysis result	combined
secure	negative	true negative
secure	positive	false positive
insecure	negative	false negative
insecure	positive	true positive

Table 4.4: The four possible analysis results when rated with respect to the ground truth

Apart from that and as I already elaborated on, each of IFSPEC's samples comes with a *ground truth* which specifies the sample as *secure* or *insecure*. Now, if we compare the analysis result with the actual ground truth, we yield four possible combinations that can be thought of as "rated analysis results", i.e. the analysis result together with the assessment whether the analysis result matches the sample's ground truth.

The four possible combinations are listed in Table 4.4.

This can also be expressed in terms of the formalisms introduced in section 3.1. Evaluating the analyses on a number of samples can be thought of evaluating empirically the four sets

$$\begin{array}{ll}
\{\mathcal{P} \mid \mathcal{P} \not\models \phi \land \mathcal{P} \nvdash_{\mathcal{A}} \phi\} & \text{(true positive)} \\
\{\mathcal{P} \mid \mathcal{P} \not\models \phi \land \mathcal{P} \vdash_{\mathcal{A}} \phi\} & \text{(false negative)} \\
\{\mathcal{P} \mid \mathcal{P} \models \phi \land \mathcal{P} \nvdash_{\mathcal{A}} \phi\} & \text{(false positive)} \\
\{\mathcal{P} \mid \mathcal{P} \models \phi \land \mathcal{P} \vdash_{\mathcal{A}} \phi\} & \text{(true negative)}
\end{array}$$

for a property ϕ that expresses that the given program is secure w.r.t. to the specification[22].

Let $\#S$ be the number of secure samples and $\#I$ be the number of insecure samples, respectively. For a fixed analysis tool, we use $\#TP$, $\#FP$, $\#TN$, $\#FN$ to refer to the number of true positive, false positive, true negative and false negative analysis results, respectively. Analogously, let $\#P$ be the number of positive analysis results and $\#N$ be the number of negative analysis results. Hence, we have

$$\#S = \#TN + \#FP$$

[22] Note that ϕ has to express security instead of *in*security due to our usage of soundness.

$$\#I = \#TP + \#FN$$

$$\#P = \#TP + \#FP$$

$$\#N = \#TN + \#FN$$

Based on these numbers, several quantities can be derived that can be useful in assessing an analysis tool. In our paper about IFSPEC, we used two of them, *recall* and *precision*. These two quantities also have been used to assess other information flow analysis tools like *FlowDroid* [21][23].

- The *recall* measures how many insecure samples yield a positive analysis result.

  $$recall = \frac{\#TP}{\#I} = \frac{\#TP}{\#TP + \#FN}$$

  The recall always lies between 0 and 1. A recall of 0 means that the analysis tool never returns a positive result for any insecure sample, whereas a recall of 1 means that the analysis tool yields a positive result for all insecure samples. Looking at the formula, we see that the recall is low if the number of false negatives is high. In this sense, one could say that the recall is a measure of soundness of an analysis tool, at least for the given set of samples.

- The *precision* measures how many samples with positive analysis result are actually insecure.

  $$precision = \frac{\#TP}{\#P} = \frac{\#TP}{\#TP + \#FP}$$

  The precision also lies between 0 and 1. A precision of 0 means that the analysis never returns a positive result for an insecure sample or,

[23] Note that there are other metrics that may be more adequate in assessing an information flow tools' actual precision than the one defined in the following. However, also note that the metrics we picked are supposed to be an example for evaluations that can be performed with IFSPEC. Hence, such a discussion lies outside of the scope of this work.

conversely, that all its positive results are false positives. Conversely, a precision of 1 means that the analysis tool has no false positives. In this sense, one could say that this quantity is a measure for actual the precision of an analysis tool, at least for the given set of samples.

tool	target language	#samples	#soap samples	TP	TN	FP	FN	recall	precision
Cassandra	DBC	232	109	68+79	15	40+30	0	100%	67.7%
Joana	JBC	232	0	139+0	35	50+0	8	94.6%	73.5%
Jodroid	DBC	232	3	136+2	32	52+1	9	93.9%	72.3%
KeY	JSC	232	208	7+138	12	5+70	0	100%	65.9%

Legend: JSC=Java source code, JBC=Java bytecode, DBC=Dalvik bytecode

Figure 4.19: Overview of benchmark results – cf. [85, Fig. 5]

Figure 4.19 shows the results of running the four analysis tools Cassandra, KeY, Joana and Jodroid on ifspec. It lists the respective numbers of true positives, true negatives, false positives and false negatives and also the recall and precision values derived from these numbers.

Note that apart from returning a normal result, an analysis tool may also crash for a given sample. This may be due to a bug but also due to the fact that the sample uses a feature that cannot be treated by the respective analysis tool. We interpreted such cases as *positive* results and call them *soundly over-approximated*, or *soap* for short. The number of soap samples are reported separately for each analysis tool, both the total number and how many of false positives and true positives where due to sound over approximation (denoted as +n).

In the following, I want to discuss Joana's and Jodroid's results in more detail. Discussions of the other tools' results can be found in our article on ifspec [85].

The results of Joana match the ground truths for 174 of the samples in ifspec. The 50 false positives are mainly caused by the fact that Joana over-approximates actual program behavior. For instance, Joana does not reason about values and does not rule out control flow which is actually impossible due to algebraic invariants. Other sources of imprecision include array handling (Joana does not distinguish between different cells of the same array) and exceptional control flow.

The eight false negatives are due to two reasons. Seven false negatives are caused by the usage of reflection: Joana tries to handle reflective code but leaves it unresolved if it fails in doing so. The resulting PDG is then incomplete. The second reason is that Joana models static initializers improperly: In one example, the leak is caused by the fact that in Java, class initializers are executed lazily. Joana on the other hand assumes that all class initializers are executed upfront and hence misses the leak because it assumes that the leaking statement is executed at a time when no secret information is available yet.

The benchmarking results for Jodroid show differences in 11 samples. These appear to be caused by Jodroid's Dalvik frontend, which not only reads in the bytecode but also performs simple intraprocedural analyses on it. In three examples, Joana could deliver a result while Jodroid crashed. In five examples, Joana did not report a flow and JoDroid did. Possible reasons for this may include differences in the handling of static initializers and the analysis of exceptional control-flow. Three more differences appear to stem from a bug in Jodroid's modelling of multidimensional arrays.

We also ran Jodroid on the 119 DroidBench samples that are integrated into IFSpec. JoDroid delivered the expected results on 67 of them (54 true positives, 13 true negatives) and unexpected results on 52 samples (seven false positives, 45 false negatives) – this corresponds to a recall of 54.6% and a precision of 88.5%. The false negatives shed light on Jodroid's limits which I already elaborated on in subsection 4.4.3 (in particular subsubsection 4.4.3.3): It currently only has rudimentary support for Android features like intents and dynamic broadcast receivers and does not detect entry points corresponding to graphical interfaces. Also, the results clearly show that the stubs we used for Jodroid are insufficient as they do not reflect the dependencies of the actual library methods.

4.7 SHRIFT– System-Wide HybRid Information Flow Tracking

In the following, I report on SHRIFT, a collaboration with the group of Pretschner at TU Munich. The general idea of the SHRIFT approach is to use static information flow analysis to improve the precision and runtime performance of a usage control and enforcement system. We also implemented the approach using Joana and demonstrated it on a case study.

The following summary is largely based on the resulting publication [122]. I concentrate on the motivation for this work and a brief description on its approach, with a focus on aspects of my contribution that were not covered by the paper. For detailed results and their discussion, I refer the interested reader to the original article.

4.7.1 Background and Motivation

Usage control [132] is an extension of access control. Apart from the question "who is allowed to access this data?" it is also concerned with usage policies ("how is this data allowed to be used after access has been granted?"), data flow tracking mechanisms ("what is allowed to happen to this data?" or "what must happen to this data?") and runtime enforcement mechanisms ("what happens if the policy is violated?").

A major challenge for effective usage control is the fact that data may exist in different representations and/or on different system layers. For example, "don't copy this picture" may mean *don't send an e-mail to which this picture is attached* in an e-mail client, *don't copy this file* on the operating system layer or *don't copy&paste this picture* in an image editor.

As a possible solution to this challenge, it has been proposed (a) to model usage control polices in a representation-independent language and (b) to track and enforce these policies on multiple layers of abstraction using a distributed approach [135]. However, multiple monitors running in parallel and communicating with each other may incur a significant runtime overhead and it may be the case that monitors are not available for every layer of abstraction.

A remedy for the absence of a dedicated monitor is to rely on conservative estimation: For example, if a dedicated monitor for a process is not available, an OS-level monitor would treat the process as a "black box" and assume that every output of the process may result from any sensitive input this process has come in touch with. However, this may lead to a phenomenon called *label creep*: Due to over-approximation, the system falsely assumes that a piece of data is compromised with high data and prevents necessary actions on it because according to the system's policy, they are not allowed. In the worst case, this may lead to an unusable system.

4.7.2 Approach

SHRIFT aims to improve on the "black box"-approach described above. Using a static information flow analysis, we compute an over-approximation of the data flows between the sources and the sinks a given application imports data from and exports data to, respectively. Then the application is instrumented with a lightweight runtime monitor which, instead of performing full data-flow tracking, consults the result of the static analysis phase every time a sink is executed to report to the OS-level monitor a list of sources which may have contributed to the piece of data which is exported by the sink.

This way, SHRIFT may increase precision in comparison with the "black box"-approach and at the same time reduce the runtime overhead of a dedicated application runtime monitor communicating with the system's monitor.

We provided and exemplified an implementation of the SHRIFT approach by using JOANA as the static information flow analysis. Moreover, we evaluated our approach in terms of precision gain with respect to the black box approach and performance gain with respect to a fully dynamic analysis. Our evaluation showed that by employing a lightweight runtime monitor using the result of a static information flow analysis like JOANA, it is possible to obtain a significant performance gain in comparison with a fully dynamic approach and being more precise than the black box approach while at the same time retaining a reasonable amount of soundness.

We applied our approach to the following example[24], which is visualized in Figure 4.20:

A company enforces the policy "upon logout, delete every local copy of customer data" *to prevent clerks from working with outdated material. Upon every login, a clerk must download from a central server a fresh version of the customer data he is interested in. In this setting, a clerk uses the* JZip *application to compress multiple customer data (**E**, **F**) into a single archive file (*`File 3`*), which he then sends to the company server using* JFTP.

This example illustrates that precision is crucial: If data-flow tracking is imprecise, then not only customer data but also additional resources which are vital to the system's functionality, such as the Zipper's configuration

[24] Here, I show a slightly adapted version of the scenario description in the original paper [122, page 372].

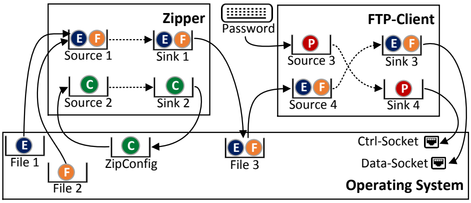


Figure 4.20: Example scenario on which we demonstrated our SHRIFT approach (taken from [122, p. 372])

file, may be deleted. Moreover, it is important that the usage control system is able to distinguish the two different channels FTP works with: A data channel is used for the data to be sent (in this case the zipped customer data), whereas a control channel is used for commands and credentials. With the black box approach, once customer data has been read, every write operation is assumed to contain customer data. Hence, even the credentials written to the control channel by the client and the database are subject to deletion.

In the following, I describe how JOANA was used to execute the static analysis phase.

Input for the static analysis is a list of sources and sinks to consider. It is important to notice that usually the list of descriptors is provided to the analysis by a security expert. In general, this list may depend on the application under analysis, since an application can e.g. use JNI to call its own native libraries.

In general, to be independent from the concrete application, we represent sources and sinks as pairs (m, p) where m is a method and p is a parameter of m. In the following, we call such pairs *descriptors*. A descriptor (m, p) represents parameter p of every invocation of m in the application code. It is notated in the format which is also used by JAVA's class file format[25]. For example, the descriptor `(FileInputStream.read([B), param 1)` represents the byte array passed as first parameter to any call of the method `read(byte[])` of class `FileInputStream`.

Consider the code snippet shown in Listing 4.14, taken from our running example. Here, we want JOANA to consider the first parameter of the

[25] See e.g. *The JAVA Virtual Machine Specification, Java SE 8 Edition*, §4.3.3.

call at line 10 as a source and the first parameter of the call at line 11 as a sink. To a certain extent, the descriptors have to be chosen manually, e.g. by reading the API documentation and then deciding which methods/parameters are relevant. For example, one may consider all variants of `FileOutputStream.write()` together with appropriate parameters as sinks.

```
 1 FileOutputStream fos = new FileOutputStream(file);
 2 ZipOutputStream zos = new ZipOutputStream(fos);
 3 List<String> fileList = this.generateFileList();
 4 byte[] buffer = new byte[1024];
 5 for (String file : fileList) {
 6   ZipEntry ze = new ZipEntry(file);
 7   zos.putNextEntry(ze);
 8   FileInputStream in = new FileInputStream(file);
 9   int len;
10   while ((len = in.read(buffer)) > 0)
11     zos.write(buffer, 0, len);
12   in.close();
13 }
```

Listing 4.14: Java code fragment for Zipper application

However, it may be the case that applications do not invoke source or sink methods directly. Consider again Listing 4.14: In line 2, a `FileOutputStream` is wrapped into a `ZipOutputStream`. When line 11 is executed, ultimately `FileOutputStream.write()` will be called, but not directly from application code. To also cover such cases, the descriptors have to be more general. One approach is to list every method manually and explicitly. This may be error-prone because methods may be missed. We chose another approach: We list only methods of the most general I/O classes `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`. After that, they are extended automatically using the following rule: *If (m, p) is a source descriptor and m' overrides m, then also (m', p) is a descriptor*. This rule can be implemented by analyzing the class hierarchy of the given program.

Still, this may not suffice. For example, JZip also contains a call to the method `Properties.load()` which takes an input stream as parameter and uses it to fill a properties table. This method is not included by the above rule because `Properties` itself is not an I/O class. For this reason, the descriptors are again extended automatically by the following rule: *If* (m, p) *is a descriptor,* m' *may call* m *and* p' *is a parameter of* m'*, then* (m', p') *is also a descriptor.* This rule can be implemented using a call graph of the application, which is also built and used during PDG construction, so it can be reused here.

Once the descriptors have been extended, they can be used to find the locations of the sources and sinks in the application and map them to appropriate PDG nodes.

Joana then computes the outcome of this phase: a table that lists, for each sink, all the sources that may influence this sink. An example is depicted in listing 4.15.

```
<source>
  <id>Source1</id>
  <location>JZip.zipIt(Ljava/lang/String;Ljava/lang/String;)V:191</location>
  <signature>java.io.FileInputStream.read([B)I</signature>
  <return/>
</source>
<source>
  <id>Source2</id>
</source>
...
<sinks>
  <sink>
    <id>Sink1</id>
  </sink>
  <location>JZip.zipIt(Ljava/lang/String;Ljava/lang/String;)V:185</location>
  <signature>java.util.zip.ZipOutputStream.write([BII)V</signature>
  <param index="1"/>
</sink>
<flows>
  <sink id="Sink1">
    <source id="Source1"/>
  </sink>
</flows>
```

Listing 4.15: Static analysis report listing sinks, sources and their dependencies

4.7.3 Results

In the following, I give a summary of the results of our evaluation of the SHRIFT approach. I concentrate on the Joana part.

As Zipper application, we used *JZip*, a simple command-line application written by us that uses the built-in ZIP functionality of the Java standard library and Apache Commons CLI [60] (Version 1.2) to implement command-line features. Without libraries, it has 293 LoC.

The FTP client *JavaFTP* was downloaded from SourceForge [109]. It does not use any libraries apart from Java's standard library and consists of 2082 LoC.
All measurements were conducted on a system with a 2.6 GHz Xeon-E5 CPU and 3GB of RAM. We ran Joana on JZip and JavaFTP with four different points-to analyses. For each points-to analysis, we considered two variants with respect to control dependencies. In variant DI ("direct and indirect flows"), Joana built the normal PDG, including control-dependencies. In variant D ("only direct flows") it built a PDG without control dependencies. This was realized by first building the normal graph, removing control-dependencies and summary edges from this graph and finally re-computing the summary edges.
Each run consists of four steps: First, Joana built the call graph of the given program and extended the given sources and sinks as described in subsection 4.7.2. Then, Joana built the program dependence graph and identified sources and sinks in it according to the extended lists. Finally, Joana used context-sensitive slicing to count the number of source-sink-pairs that were connected in the PDG.

4.7.3.1 Performance

Table 4.5 shows the overall time that Joana took for each configuration, along with rough estimates of the sizes of the respective PDGs. The times for the D and DI variants are aggregated by reporting the time for the slower variant. Note that although the D variant takes more time for the PDG construction, it may take less time in the slicing phase since there are less edges – the overall time for the D variant may therefore be roughly the same or even smaller. The graph sizes are reported for the DI variant. We can see that the points-to analysis has a massive impact on the graph size and also on the overall time needed to perform the analysis. For object-sensitive points-to, the number of edges can be 6.4-6.6 times as high as for 0-1-CFA, resulting in an overall time that is 6.7-6.9 times as high.

4.7.3.2 Precision

Table 4.6 shows the number of source-sink-connections for both examples for all considered configurations and variants, which we call *flows* for

	points-to	graph size (DI)		time (sec.)
		#nodes	#edges	
JavaFTP	0-1-CFA	6.82×10^4	8.65×10^5	32
	1-CFA	1.77×10^5	1.95×10^6	64
	2-CFA	3.98×10^5	4.54×10^6	153
	object-sensitive	3.06×10^5	5.73×10^6	220
JZip	0-1-CFA	1.10×10^5	1.44×10^6	53
	1-CFA	2.04×10^5	2.13×10^6	82
	2-CFA	3.47×10^5	3.61×10^6	185
	object-sensitive	4.50×10^5	9.15×10^6	353

Table 4.5: Overall time of static analysis phase and PDG sizes for JavaFTP and JZip

short. Moreover, Table 4.6 gives a simple and coarse estimation of the precision gain obtained. This value is computed as

$$precision = 1 - \frac{\#flows}{\#sources \cdot \#sinks}$$

and compares the respective static analysis result to a conservative black box approach where every source is assumed to possibly flow to every sink. Such an approach would correspond to a static analysis with a *precision* of 0. Hence, the higher the *precision* value, the larger the precision gain of using the respective static analysis is[26].

According to this measure, we see that the choice of points-to analysis is crucial for Joana's precision and has to be adapted to the application under analysis. For JavaFTP in the DI variant, object-sensitive points-to analysis makes the analysis 6.6 times as expensive as 0-1-CFA, without

[26] Note however, that the term "precision" is not entirely appropriate, since we deliberately give up soundness in the D configurations. In [122], we argue why it may be appropriate to ignore control dependencies for our application.

[27] The actual sources and sinks may vary with different points-to analyses because Joana uses points-to analysis for call graph construction and particularly for the identification of reachable code. Also, [122] reports 84% for JZip/object-sensitive/D, possibly due to a typo in the paper.

	points-to	#sources	#sinks	#flows (precision %)	
				DI	D
JavaFTP	0-1-CFA	9	46	274 (38%)	214 (51%)
	1-CFA	9	46	187 (58%)	119 (73%)
	2-CFA	9	46	187 (58%)	118 (73%)
	object-sensitive	9	46	272 (38%)	114 (74%)
JZip	0-1-CFA	10	56	428 (24%)	321 (43%)
	1-CFA	10	55	414 (25%)	257 (53%)
	2-CFA	10	55	246 (55%)	123 (78%)
	object-sensitive	10	55	239 (57%)	91 (83%)

Table 4.6: precision values for JavaFTP and JZip[27]

any precision gain. Also 2-CFA does not have any advantage over 1-CFA although its runtime is 2.4 times as high. For JZip in the DI variant, on the other hand, the higher costs seem to pay off. For the D variants, the precision gain of a more precise points-to analysis is more visible. Joana uses points-to analysis not only to construct its call graph but also heavily for the computation of the heap dependencies. Hence it is plausible that a more precise points-to results in more precise heap dependency and therefore data dependency graph.

4.8 Modular Verification of Information Flow Security in Component-Based Systems

In a collaboration with the group of Beckert at KIT [83], we also applied Joana in the area of information flow security verification of component-based systems.

In an instantiation of the approach proposed in the article, we used Joana to verify user-provided service-level security properties. From these properties, first-order formulas are generated that express that component-level security follows from service-level security and that system-level security follows from component-level security. These formulas can be discharged using a first-order theorem prover like KeY.

The approach is both modular with respect to services and components and with respect to service-level security properties. This not only means that the security of the whole can be derived from the security of its parts but also that service-level properties verified by Joana can be re-used to show different overall security properties.
We applied the approach to a case study, in which we verified the security of a system implemented in Java.

4.9 Summary and Conclusion

In this chapter, I gave an overview of some of the activities within the RS^3 priority program, with a focus on the achievements of the program paradigms group and the sub-project *Information Flow for mobile components*. In section 4.2, I described the advances in our work on *probabilistic non-interference*, particularly how we developed *relaxed low-security observational determinism* (RLSOD), a criterion that (a) can be verified using PDGs and control-flow checks, (b) improves the precision (under certain scheduling assumptions) on earlier criteria based on observational determinism and (c) enforces probabilistic non-interference.
After that, I reported on seven collaborations within RS^3 in which Joana was involved.
In section 4.3, I elaborated on how Joana and the KeY theorem prover can be combined to verify cryptographic properties of a prototypical E-Voting system.
In section 4.4, I presented Jodroid, an extension of Joana for Android apps and showed how it was integrated into the *RS^3 certifying app store*, the artifact of the RS^3 reference scenario *Software Security for mobile devices*. Subsequently, in section 4.5, I reported on the joint work on the *RS^3 Information Flow Language* (RIFL), a language to specify information-flow properties in a language- and tool-neutral way. Then, I described Joana's RIFL support in subsection 4.5.2. An application of RIFL was shown in section 4.6: RIFL was used to provide an information-flow security benchmark suite that is also fully supported by Joana.
Last but not least, I elaborated on two other collaborations that showed how a static information flow control tool like Joana can be applied (a) to improve the precision and performance of usage control (section 4.7)

and (b) to aid a theorem prover in the verification of information-flow properties for component-based systems (section 4.8).

All in all, this chapter demonstrated that our progress and our collaborations within the RS^3 project contributed to the establishment of PDG-based static analysis techniques in the realm of security analyses. Joana is a matured tool whose theoretical foundation has been firmly stabilized. It can be applied to a wide variety of scenarios, ranging from advanced security checks of mobile apps over the verification of cryptographic security properties to the improvement of usage control systems and the simplification of theorem proving approaches to the verification of component-based systems. With Joana's support for RIFL and ifspec, a well-founded baseline can be drawn that should drive and foster the state-of-the art of static security analysis tools in the future.

And in the end, the love you take
is equal to the love you make.
The Beatles

5

A Common Generalization Of Program Dependence Graphs and Control-Flow Graphs

In chapter 3, I considered the interprocedural versions of slicing on program dependence graphs (PDGs) and data-flow analysis on control-flow graphs. In subsection 3.3.4, I identified similarities and argued that slicing on PDGs can be considered as a very simple data-flow analysis instance.

In this chapter, I am going to further develop these ideas. I will introduce *interprocedural graphs* (IGs), a general graph model that is less restricted than interprocedural control-flow graphs (ICFGs), yet still enables data-flow analysis. Both interprocedural control-flow graphs and interprocedural program dependence graphs can be considered as IGs. This makes available a whole range of data-flow analyses for PDGs.

IGs generalize ICFGs as defined in subsubsection 3.2.1.2 in several aspects. For one, the procedures of ICFGs each have only one entry and one exit. IGs lift this restriction and allow for multiple entries and exits. Secondly, in an ICFG, each call has exactly one corresponding return. In contrast, IGs allow for arbitrary corresponding relations between calls and returns. A third aspect in which IGs generalize ICFGs is that ICFGs – at least in the context of classical data-flow analysis – usually make some reachability assumptions. For data-flow analysis on IGs, these assumptions are not neccessary.

The data-flow analysis variant I introduce in this chapter is also a generalization of its counterpart on interprocedural control-flow graphs that I already considered in subsubsection 3.2.2.2. These generalizations are neccessary to properly consider slicing as a data-flow analysis.

As I have carved out in subsection 3.3.4, two generalizations are necessary to make this idea work. In the following, I am going to briefly describe them.

The first generalization is concerned with the notion of *interprocedurally valid paths*. Remember that subsubsection 3.2.2.2 introduced this notion to characterize the properties of interprocedural paths considered by a data-flow analysis (cf. page 56).

Intuitively, an interprocedurally valid path π respects call semantics. This amounts to two properties. Firstly, if a procedure is called on π and it returns later, then the call site to which it returns must match the call site from which the call started in the first place. Secondly, if a procedure returns on π, then π must also contain a matching call. The notions developed in this chapter only require that the first property is satisfied. This matches the notion of interprocedural validness that one usually makes on PDGs to define slicing.

The second generalization is concerned with the fact that a slice is defined usually not with respect to a fixed entry point but rather with respect to an arbitrary node. In subsection 5.4.2, this leads to a version of the objective function *MOVP* that has – unlike the version in subsubsection 3.2.2.2 (cf. equation (3.8)) – not one argument, but two.

This chapter is organized in four parts. In section 5.1, I develop the notion of *valid sequences* and important variants. I base these notions on classic results from the theory of balanced parenthesis. My approach is to first introduce intuitive definitions and then derive inductive definitions from them. After that, in section 5.2, I present interprocedural graphs and my notion of valid paths, which is based on the theory developed so far. Next, section 5.3 presents data-flow analysis on interprocedural graphs. Finally, I conclude this chapter in section 5.4 by showing a variety of use cases for data-flow analysis on IGs, including already existing PDG-based approaches and several graph-theoretic notions.

5.1 Nesting Properties of Symbol Sequences

In this section, I develop the notion of *validness* for symbol sequences. Valid sequences form the basis of valid paths, which I will define in section 5.2 along with interprocedural graphs.

Validness is concerned with symbol sequences that essentially consist of opening and closing parentheses and connects two different ways of assigning closing parentheses to opening parentheses. One of them counts parentheses to assign every opening parenthesis at most one closing parenthesis such that the part between the two is *balanced*. The other way is expressed as a correspondence relation that is used to relate opening parentheses and closing parentheses that are compatible. Validness then demands that if two parentheses match, then they must be compatible with respect to the correspondence relation.

This section is organized as follows: In subsection 5.1.1, I introduce the *matching relation* that assigns each opening parenthesis in a given sequence at most one closing parenthesis and vice versa. This relation makes use of *balanced sequences*, which in turn can be characterized using a formalization of the process of counting parentheses. It turns out that relation-theoretic properties of the matching relation can be used to characterize two different kinds of *partially balanced sequences* that become important in defining the building blocks of valid sequences and paths. Partially-balanced sequences are introduced in subsection 5.1.2. After that, subsection 5.1.3 is concerned with the second kind of assigning opening and closing parentheses to each other: It introduces a relation that specifies which parentheses are compatible. Using this relation, I define *valid sequences*. Additionally, I introduce valid counterparts for (partially-)balanced sequences. Finally, after I have introduced valid sequences in subsection 5.1.3, I derive inductive definitions for valid sequences and their partially-balanced variants in subsection 5.1.4.

5.1.1 Balanced Sequences

In this section, I introduce *balanced sequences*. I do this according to Knuth [107] with the help of two functions which formalize the process of counting parentheses.

In the following I consider symbol sequences over $E = E_{intra} \cup E_{call} \cup E_{ret}$ where E_{intra}, E_{call} and E_{ret} are pairwise disjoint. Elements of E_{call} are called *opening parentheses*, *call symbols* or just *calls*. Elements of E_{ret} are called *closing parentheses*, *return symbols*, or just *returns*. Elements of E_{intra} are called *inner symbols*. Later, the opening parentheses will model calls of procedures and the closing parentheses will model returns from procedures. The inner symbols will later model intraprocedural edges. With $Callpos(\pi) := \{i \in$

range$(\pi) \mid \pi_i \in E_{call}\}$ and $Retpos(\pi) := \{i \in range(\pi) \mid \pi_i \in E_{ret}\}$ I denote the set of *call* and *return positions* in π, respectively.

Definition 5.1. *Let $\pi \in E^\star$ be a symbol sequence.*

1. *The* content $c(\pi)$ *of* π *is defined as follows:*

$$c(\epsilon) = 0$$

$$c(\pi \cdot e) = \begin{cases} c(\pi) + 1 & \textit{if } e \in E_{call} \\ c(\pi) - 1 & \textit{if } e \in E_{ret} \\ c(\pi) & \textit{if } e \in E_{intra} \end{cases}$$

2. *The* deficiency $d(\pi)$ *of* π *is defined as follows:*

$$d(\epsilon) = 0$$

$$d(\pi \cdot e) = \begin{cases} max\{d(\pi), -c(\pi)\} & \textit{if } e \in E_{call} \cup E_{intra} \\ max\{d(\pi), -c(\pi) + 1\} & \textit{if } e \in E_{ret} \end{cases}$$

Given a symbol sequence π, the function c computes the difference between the number of call symbols and the number of return symbols of a given symbol sequence, wheres d computes the maximal shortage of call symbols among the prefixes of π. I give some examples to illustrate how c and d work.

Example 5.2. *Let $E_{intra} = \{\star\}$, $E_{call} = \{(\}$ and $E_{ret} = \{)\}$.*

1. *For $\pi_1 = ()(\star)$, we have*

i	0	1	2	3	4
π_1^i	(	)	(	$\star$	)
$c(\pi_1^{\leq i})$	1	0	1	1	0
$d(\pi_1^{\leq i})$	0	0	0	0	0

2. *For $\pi_2 = ()))$, we have*

i	0	1	2	3
π_2^i	(	)	)	)
$c(\pi_2^{\leq i})$	1	0	-1	-2
$d(\pi_2^{\leq i})$	0	0	1	2

3. *For $\pi_3 = (()())$, we have*

i	0	1	2	3	4	5
π_3^i	(	(	)	(	)	)
$c(\pi_3^{\leq i})$	1	2	1	2	1	0
$d(\pi_3^{\leq i})$	0	0	0	0	0	0

4. *For $\pi_4 =)()((\star$, we have*

i	0	1	2	3	4	5
π_4^i	)	(	)	(	(	★
$c(\pi_4^{\leq i})$	-1	0	-1	0	1	1
$d(\pi_4^{\leq i})$	1	1	1	1	1	1

5. *For $\pi_5 = ()(\star(\star$, we have*

i	0	1	2	3	4	5
π_5^i	(	)	(	★	(	★
$c(\pi_5^{\leq i})$	1	0	1	1	2	2
$d(\pi_5^{\leq i})$	0	0	0	0	0	0

Remark 5.3 and Lemma 5.4 formalize important properties of c and d.

Remark 5.3. *c is additive: $c(\pi_1 \cdot \pi_2) = c(\pi_1) + c(\pi_2)$.*

Proof. This follows by induction on π_2. □

Lemma 5.4. *For every $\pi \in E^\star$, we have*

1. $c(\pi) = |Callpos(\pi)| - |Retpos(\pi)|$

2. $d(\pi) = \max\{-c(\theta) \mid \theta \in Prefixes(\pi)\}$

3. $d(\pi) \geq 0$

4. $d(\pi) = 0 \implies c(\pi) \geq 0$

Proof.
- The first two claims can be proven by a straightforward induction on the length of π.
- For the third claim, we observe that (1) ϵ is always a prefix of π, (2) $c(\epsilon) = 0$ (by definition of c) and conclude from the second claim that (3) $d(\pi) \geq -c(\epsilon) = 0$.

- Since π is a prefix of itself, we may apply the second claim and obtain $d(\pi) \geq -c(\pi)$ or, equivalently, $-d(\pi) \leq c(\pi)$. For $d(\pi) = 0$, this implies $c(\pi) \geq 0$.

□

Now I define what it means for a symbol sequence to be *balanced*. Intuitively, in a balanced sequence we can match the call positions with the return positions in a *well-nested* fashion.

However, initially I define balancedness as a mere property about the counts of opening and closed parentheses in a symbol sequence. Balancedness has two requirements: Firstly, there must be as many call symbols as return symbols in π. Secondly, for every prefix of π there cannot be more return symbols than call symbols. The first property guarantees that a bijective function between the call positions and the return positions is possible, while the second ensures that it is always possible to map each call position to a *later* return position.

Going back to Example 5.2, we see that π_1 and π_3 are intuitively balanced, whereas π_2, π_4 and π_5 are not: π_2 has more return symbols than call symbols, while π_5 has more call symbols than return symbols. Moreover, as π_4 starts with a return symbol, it cannot be extended to a balanced sequence.

The examples show that the balancedness of sequences can indeed by characterized using c and d.

Definition 5.5. *A sequence $\pi \in E^\star$ is called* balanced *if $c(\pi) = d(\pi) = 0$. The set of balanced symbol sequences is written as $Bal(E)$.*

The following lemma gives an easy characterization of balancedness which we will make use of later.

Lemma 5.6. *A sequence $\pi \in E^\star$ is balanced iff $c(\pi) = 0$ and $c(\theta) \geq 0$ for all prefixes θ of π.*

Proof. This follows easily from Definition 5.5 and Lemma 5.4. □

5.1.1.1 The Matching Relation

Now I introduce a relation ν_π which relates opening and closing positions in a symbol sequence π to each other. The relation was also introduced by

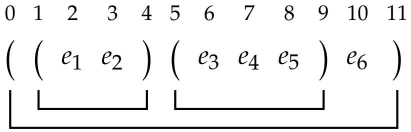


Figure 5.1: An example sequence with its matching relation. The e_i are inner symbols, there is one call symbol "(" and one return symbol ")". Connected symbols are related by the matching relation.

Knuth [107], albeit not formally. An example for the intuition behind ν_π can be seen in Figure 5.1.

Definition 5.7. *For a sequence $\pi \in E^\star$ we define the* matching relation *$\nu_\pi \subseteq Callpos(\pi) \times Retpos(\pi)$ by:*

$$(i, j) \in \nu_\pi \iff i < j \wedge \pi_i \in E_{call} \wedge \pi_j \in E_{ret} \wedge \pi^{]i,j[} \textit{ is balanced}$$

In the following, I am going to state three important properties of ν_π that reflect the intuitive expectations toward a properly defined matching relation.

Firstly, Theorem 5.8 states that ν_π never relates a call position to multiple return positions or, conversely, a return position to multiple call positions (compare Knuth[107, p. 271]).

Theorem 5.8. *For any symbol sequence $\pi \in E^\star$, ν_π is left- and right-unique.*

Proof. We show left- and right-uniqueness separately.

right-uniqueness: Let $(i, j) \in \nu_\pi$ and $(i, j') \in \nu_\pi$. Assume, for the purpose of contradiction, that $j \neq j'$. Without loss of generality, we may assume that $j < j'$. Since $\pi^{]i,j'[}$ is balanced, we know that $d(\pi^{]i,j'[}) = 0$. Because $\pi^{]i,j]}$ is a prefix of $\pi^{]i,j'[}$, we get $c(\pi^{]i,j]}) \geq 0$ by Lemma 5.4. However, since $\pi_j \in E_{ret}$ and $\pi^{]i,j[}$ is balanced (which means that $c(\pi^{]i,j[}) = 0$), we also have:

$$c(\pi^{]i,j]}) = c(\pi^{]i,j[}) - 1 = 0 - 1 = -1$$

This is a contradiction, so the assumption must be false and it must be $j = j'$.

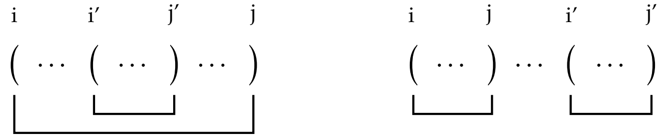


Figure 5.2: Illustration of the different cases of well-nestedness of ν_π

left-uniqueness: Let $(i, j) \in \nu_\pi$, $(i', j) \in \nu_\pi$. Assume, for the purpose of contradiction, $i \neq i'$. Without loss of generality, assume $i < i'$. Since $\pi^{]i,j[}$ is balanced, we have $c(\pi^{]i,j[}) = 0$. Since both $\pi^{]i,i'[}$ and $\pi^{]i,i']}$ are prefixes of $\pi^{]i,j[}$, we have $c(\pi^{]i,i'[}) \geq 0$ and $c(\pi^{]i,i']}) \geq 0$ by Lemma 5.4. In fact, since $\pi_{i'} \in E_{call}$, we even have $c(\pi^{]i,i']}) = c(\pi^{]i,i'[}) + 1 > 0$. But then, using Remark 5.3, we can conclude

$$c(\pi^{]i',j[}) = c(\pi^{]i,j[}) - c(\pi^{]i,i']}) < c(\pi^{]i,j[}) = 0,$$

so that $c(\pi^{]i',j[}) < 0$, which contradicts the balancedness of $\pi^{]i',j[}$.

□

Secondly, the one-sided totality properties of ν_π can be used to assess whether π is balanced or not. This is the statement of Theorem 5.9.

Theorem 5.9. *For any symbol sequence $\pi \in E^\star$, the following conditions are equivalent:*

(a) π is balanced.

(b) ν_π is left- and right-total, i.e. a bijective function $Callpos(\pi) \rightarrow Retpos(\pi)$.

The third property, which is given in Theorem 5.10, states that ν_π relates call and return positions in a *well-nested* fashion.
More specifically, the respective sections between two matching position pairs never overlap.
For an illustration, see Figure 5.2. Given two pairs $(i, j), (i', j') \in \nu_\pi$ of call and return positions that are related by ν_π, the corresponding index ranges $[i, j]$ and $[i', j']$ are either disjoint or one of them is contained in the other.

Theorem 5.10. *Given $\pi \in E^\star$, assume that $(i, j), (i', j') \in \nu_\pi$. Then one of the following statements is true:*

1. $[i, j] \subseteq [i', j']$
2. $[i', j'] \subseteq [i, j]$
3. $[i, j] \cap [i', j'] = \emptyset$

Proofs for Theorem 5.9 and Theorem 5.10 can be found in section A.1 and Theorem 5.9, respectively.

5.1.2 Partially-Balanced Sequences

Theorem 5.9 suggests that the totality properties of ν_π can be used to classify sequences with respect to their balancedness properties. This motivates the following definition.

Definition 5.11. *We denote by $Left(E) \subseteq E^\star$ the set of symbol sequences π such that ν_π is left-total and by $Right(E) \subseteq E^\star$ the set of symbol sequences π such that ν_π is right-total. If $\pi \in Left(E) \cup Right(E)$, we also call π* partially balanced*.*

Remark 5.12.

$$Left(E) \cap Right(E) = Bal(E)$$

Proof. This follows from Theorem 5.9. □

5.1.3 Valid Sequences

The property of balancedness only considers call and return symbols in terms of their numbers. In particular, in a balanced path it is only ensured that every call symbol is matched by a return symbol and vice versa. However, it is not ensured that return symbols also *correspond* to their matching call symbols.

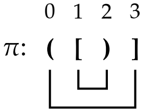


Figure 5.3: A balanced but invalid symbol sequence – positions related by ν_π are connected

As a simple example, consider the symbol sequence depicted in Figure 5.3. Here there are two distinct call symbols **(** and **[** and two distinct return symbols **)** and **]**. We find that $\nu_\pi = \{(0,3),(1,2)\}$, hence π is balanced. But the symbols at the positions related by ν_π do not belong together. To exclude such sequences, I introduce a *correspondence relation* $\Phi \subseteq E_{call} \times E_{ret}$ that specifies which call symbols belong to which return symbols. In the example, the obvious choice for Φ is $\{(\mathbf{(},\mathbf{)}),(\mathbf{[},\mathbf{]})\}$.

Definition 5.13. *Let $\pi \in E^\star$ be a symbol sequence.*

π is called valid *if*

$$\forall (i,j) \in Callpos(\pi) \times Retpos(\pi).(i,j) \in \nu_\pi \implies (\pi^i, \pi^j) \in \Phi$$

I will denote the set of valid symbol sequences as $Val(E)$.

Basically, this property says that if a call has a matching return (in terms of position in the path), this return actually corresponds to the call according to the relation. The sequence from our example is invalid, since $(0,3) \in \nu_\pi$ but $(\pi^0, \pi^3) \notin \Phi$.
In the following, I fix a correspondence relation $\Phi \subseteq E_{call} \times E_{ret}$.

Definition 5.14. *Let $\pi \in E^\star$ be a symbol sequence.*

1. *π is called* ascending *if π is valid and ν_π is left-total.*

2. *π is called* descending *if π is valid and ν_π is right-total.*

3. *π is called* same-level *if π is both ascending and descending.*

4. *With $AscSeq(E)$, $DescSeq(E)$, $SLSeq(E)$, I denote the sets of ascending, descending and same-level sequences, respectively.*

Example 5.15. *Let $E_{intra} = \{\bigstar\}$, $E_{call} = \{<_a, <_b\}$ and $E_{ret} = \{>_a, >_b\}$, Furthermore, assume that $\Phi = \{(<_a, >_a), (<_b, >_b)\}$ and consider*

$$\pi_1 \overset{def}{=} <_a >_b$$

$$\pi_2 \overset{def}{=} <_a <_b >_b >_a$$

$$\pi_3 \overset{def}{=} <_a <_b >_b >_a >_b$$

$$\pi_4 \overset{def}{=} <_a <_a >_a <_b >_b$$

$$\pi_5 \overset{def}{=} >_b >_b <_a <_a$$

1. Then π_1 is invalid: We have $(0,1) \in \nu_{\pi_1}$, but $(<_a, >_b) \notin \Phi$.

2. We have $Callpos(\pi_2) = \{0,1\}$, $Retpos(\pi_2) = \{2,3\}$ and $\nu_{\pi_2} = \{(0,3),(1,2)\}$. Now it is easy to see that π_2 is valid and both left- and right-total. Hence, π_2 is a same-level sequence.

3. We have $Callpos(\pi_3) = \{0,1\}$ and $Retpos(\pi_3) = \{2,3,4\}$. Moreover, we have $\nu_{\pi_3} = \{(0,3),(1,2)\}$. This means that π_3 is valid and ν_{π_3} is left-total, but not right-total, since there is no $i \in Callpos(\pi_3)$ with $(i,4) \in \nu_{\pi_3}$. Hence, π_3 is ascending, but not same-level.

4. Analogously, we see that π_4 is valid and that $\nu_{\pi_4} \subseteq \{0,1,3\} \times \{2,4\}$ is right-total, but not left-total. Hence, π_4 is descending, but not same-level.

5. Finally, $\nu_{\pi_5} = \emptyset \subseteq \{0,1\} \times \{2,3\}$, so π_5 is trivially valid but neither left- nor right-total. Hence, π_5 is neither ascending nor descending.

Since the left- and right-totality of ν_π is equivalent to the balanced-ness of π, the same-level property can also be expressed using balanced-ness.

Theorem 5.16. *$\pi \in E^\star$ is same-level if and only if it is balanced and valid.*

Proof. By definition, π is same-level if and only if it is valid and ν_π is bijective. By Theorem 5.9, this is the case if and only if π is valid and balanced. □

In the following, I show that valid, ascending and descending sequences are closed under taking contiguous sub-sequences. This becomes clear relatively quickly, if we think about cases in which the respective totality properties are maintained: Taking an arbitrary sub-sequence of π can only remove or shift the positions appearing in ν_π. Hence, validness is maintained by this operation. Left-totality may be destroyed, if we take a suffix and potentially remove a return position by this. However, taking prefixes maintains right-totality. Hence, we can conclude that descending sequences are closed under taking prefixes. Analogously, right-totality is maintained by taking suffixes since taking a suffix only removes return positions and does not hurt the matching for the remaining

return positions. Hence, ascending sequences are closed under taking suffixes.
For the proof of Theorem 5.18, I need a technical lemma, which I state and prove before I proceed with Theorem 5.18.

Lemma 5.17. *If $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3 \in E^\star$, then*

1. *$range_\pi(\pi_2) = \{i + |\pi_1| \mid i \in range(\pi_2)\}$*

2. *$\forall i, j \in range(\pi_2).\ (i, j) \in \nu_{\pi_2} \iff (i + |\pi_1|, j + |\pi_1|) \in \nu_\pi$*

Proof. We observe that

$$\pi_2 = \pi^{[|\pi_1|, |\pi_1| + |\pi_2| - 1]},$$

which is equivalent to

$$\forall i \in range(\pi_2).\ \pi_2^i = \pi^{|\pi_1| + i}.$$

From this, both claims can be proven easily. □

Theorem 5.18. *The following statements are true.*

1. *Valid sequences are closed under taking sub-sequences, in particular under taking suffixes and prefixes.*

2. *Ascending sequences are closed under taking suffixes.*

3. *Descending sequences are closed under taking prefixes.*

4. *Taking a prefix of an ascending sequence or a suffix of a descending sequence yields a valid sequence.*

Proof. 1. If π is valid and $\pi' = \pi^{[i,j]}$ is a sub-sequence of π, then π' still has the validity property, as can easily be seen.

2. Let π be ascending and $\pi^{\geq j}$ be a suffix of π. Write $n = |\pi|$. Let $k \in range(\pi^{\geq j})$ be a call position in $\pi^{\geq j}$. Then $(\pi^{\geq j})^k = \pi^{k+j}$. Because ν_π is left-total, there is a return position $l \in range(\pi)$ such that $(k + j, l) \in \nu_\pi$. From this, it follows that $l \geq k + j$. Hence, we can write $l = (l - j) + j$, so that $l - j \in range(\pi^{\geq j})$. Then, $(k + j, l) \in \nu_\pi$ implies $(k, l - j) \in \nu_{\pi^{\leq j}}$ by Lemma 5.17. Thus, we have shown that $\nu_{\pi^{\geq j}}$ is left-total.

3. Let π be descending and $\pi^{\leq j}$ be a prefix of π. Let

$$l \in Retpos(\pi^{\leq j}) \subseteq Retpos(\pi)$$

be a return position in $\pi^{\leq j}$. Due to right-totality of ν_π, we find a $k \in Callpos(\pi)$ such that $(k,l) \in \nu_\pi$. This means in particular that $k < l$. With $l \leq j$ we get $k \leq j$ so that we can conclude $(k,l) \in \nu_{\pi^{\leq j}}$. This proves that $\nu_{\pi^{\leq j}}$ is left-total.

4. Both ascending and descending paths are valid and according to the first statement, valid paths are closed under taking suffixes and prefixes.

□

5.1.4 Inductive Definitions

In this subsection, I derive inductive definitions for the various classes of sequences that I have introduced so far. I start with characterizing balanced, left-total and right-total and finally all sequences inductively. Afterwards, I consider their valid counterparts and give inductive definitions for same-level, ascending, descending and valid sequences.
Proofs for the following three theorems can be found in section A.3.

Theorem 5.19. *$Bal(E)$ is the least subset X of $E^\star$ with the following properties:*

$$(Bal1)\ \frac{}{\epsilon \in X} \qquad (Bal2)\ \frac{\pi \in X \qquad e \in E_{intra}}{\pi \cdot e \in X}$$

$$(Bal3)\ \frac{\pi \in X \qquad \pi' \in X \qquad e_{call} \in E_{call} \qquad e_{ret} \in E_{ret}}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Theorem 5.20. *$Left(E)$ is the least subset X of $E^\star$ which has the following properties:*

$$(Left1)\ \frac{}{\epsilon \in X} \qquad (Left2)\ \frac{\pi \in X \qquad e \in E_{intra} \cup E_{ret}}{\pi \cdot e \in X}$$

$$(Left3)\ \frac{\pi \in X \qquad \pi' \in Bal(E) \qquad e_{call} \in E_{call} \qquad e_{ret} \in E_{ret}}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Theorem 5.21. *$Right(E)$ is the least subset X of $E^\star$ which has the following properties:*

$$(Right1)\ \frac{}{\epsilon \in X} \qquad (Right2)\ \frac{\pi \in X \qquad e \in E_{intra} \cup E_{call}}{\pi \cdot e \in X}$$

$$(Right3)\ \frac{\pi \in X \qquad \pi' \in Bal(E) \qquad e_{call} \in E_{call} \qquad e_{ret} \in E_{ret}}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Next, Theorem 5.22 that every sequence can be split up into a left- and right-total sequence in a specific way. This is a specific version of a theorem already considered by Knuth (cf. [107, Lemma 1]). I am going to apply this property in chapter 7.

Theorem 5.22. *For every symbol sequence, there is $i \in range(\pi)$ such that every symbol sequence $\pi \in E^\star$ can be split up into $\pi = \pi^{<i} \cdot \pi^{\geq i}$ such that*

$$\pi^{<i} \in Left(E) \tag{5.1}$$

$$\pi^{\geq i} \in Right(E) \tag{5.2}$$

$$\pi \notin Left(E) \implies i \in Callpos(\pi). \tag{5.3}$$

Proof. Define the set of unfinished call positions in π as

$$\mathcal{U}_{call} \overset{def}{=} \{i \in Callpos(\pi) \mid \forall j \in range(\pi).\ (i, j) \notin \nu_\pi\}.$$

Now we make a case distinction on whether $\mathcal{U}_{call}$ is empty or not. If $\mathcal{U}_{call} = \emptyset$, then $\pi \in Left(E)$. Then we can choose $i = |\pi|$.
Now consider the case that $\mathcal{U}_{call} \neq \emptyset$. Let $i \in \mathcal{U}_{call}$ be the least element of $\mathcal{U}_{call}$. Now we show

$$\pi_1 \overset{def}{=} \pi^{<i} \in Left(E) \tag{1}$$

$$\pi_2 \overset{def}{=} \pi^{\geq i} \in Right(E). \tag{2}$$

(1) Let $i' \in Callpos(\pi_1) \subseteq Callpos(\pi)$. Then $i' < i$ and, due to the choice of i, there must be $j \in range(\pi)$ such that $(i', j) \in \nu_\pi$. This implies $\pi^{]i',j[}$ is balanced and hence that $j \leq i$, since otherwise $\pi^{]i',j[}$ would contain

the unmatched call π_i, in contradiction to the balancedness of $\pi^{]i',j[}$. In addition to $j \leq i$ we also observe $j \neq i$, since $\pi^i \in E_{call}$ and $\pi^j \in E_{ret}$ and $E_{call} \cap E_{ret} = \emptyset$. Thus, we have $j \in range(\pi_1)$ and therefore $(i', j) \in \nu_{\pi_1}$.

(2) We show $\pi_2 \in Right(E)$ by induction on $\pi_2 \in E^\star$. This is clear for $\pi = \epsilon$. So let $\pi_2 = \pi_2' \cdot e$. Our induction hypothesis says that $\pi_2' \in Right(E)$ and we have to show that $\pi_2 \in Right(E)$ as well. We proceed with a case distinction on e. If $e \in E_{intra} \cup E_{call}$, then we have $\pi_2 \in Right(E)$ by Theorem 5.21. So we assume $e \in E_{ret}$. Let $j \in Retpos(\pi_2)$. We have to find $i_0 \in Callpos(\pi_2)$ such that $(i_0, j) \in \nu_{\pi_2}$. This follows from the induction hypothesis if $j \in Retpos(\pi_2')$, so we may assume $j = |\pi_2| - 1$. In this case, we choose i_0 as the greatest unmatched call position in π_2. Such a position must exist since i is an unmatched call position in π and therefore in π_2. Furthermore, we have $i_0 < j$. This is because $E_{call} \cap E_{ret} = \emptyset$ and $\pi_2^j = e \in E_{ret}$. Finally, we have to show that $\pi_2^{]i_0,j[}$ is balanced. By Theorem 5.9, it suffices to show that $\pi_2^{]i_0,j[}$ contains neither unmatched call positions nor unmatched returns positions. We show the two claims separately:

- Assume, for the purpose of contradiction, that $\pi_2^{]i_0,j[}$ contains an unmatched call position i_1. This position would have the property $i_1 > i_0$, which is a contradiction to the maximality of i_0. Hence, such a position cannot exist.
- Assume, for the purpose of contradiction, that $\pi_2^{]i_0,j[}$ contains an unmatched return position and let j_0 be the least such position. Then $\pi_2^{]i_0,j_0[}$ would be balanced. Firstly, it cannot contain unmatched call positions, due to the maximality of i_0. Secondly, it also cannot contain unmatched return positions, due to the minimality of j_0. Moreover, we have $i_0 < j_0$, $\pi_2^{i_0} \in E_{call}$ and $\pi_2^{j_0} \in E_{ret}$, so that we obtain $(i_0, j_0) \in \nu_{\pi_2}$, a contradiction to the choice of j_0 as unmatched return position.

□

Corollary 5.23 (cf. [107], Lemma 1). *Every symbol sequence $\pi \in E^\star$ can be split up into $\pi = \pi_1 \cdot \pi_2$ such that ν_{π_1} is left-total and ν_{π_2} is right-total.*

Proof. This is a direct consequence of Theorem 5.22. □

5.1.4.1 Inductive Definitions for valid sequences

Before I actually derive and prove correct inductive definitions for the valid variants of partially-balanced sequences, I compile a couple of observations about the closure properties of same-level, ascending, descending and valid sequences. Proofs can be found in section A.3.
Firstly, I observe that appending inner symbols does not do any harm.

Lemma 5.24. *1. If $\pi \in AscSeq(E)$ and $e \in E_{intra} \cup E_{ret}$, then $\pi \cdot e \in AscSeq(E)$.*

2. If $\pi \in DescSeq(E)$ and $e \in E_{intra} \cup E_{call}$, then $\pi \cdot e \in DescSeq(E)$.

3. If $\pi \in SLSeq(E)$ and $e \in E_{intra}$, then $\pi \cdot e \in SLSeq(E)$.

Secondly, appending a same-level sequence destroys neither validness nor the respective totality property of the matching relation.

Lemma 5.25. *Let $\pi \in Val(E)$ and $\pi' \in SLSeq(E)$. Then the following statements hold:*

1. $\pi \cdot \pi'$ is valid.

2. If $\pi \in Bal(E)$, then $\pi \cdot \pi'$ is same-level.

3. If $\pi \in Left(E)$, then $\pi \cdot \pi'$ is ascending.

4. If $\pi \in Right(E)$, then $\pi \cdot \pi'$ is descending.

Lastly, same-level sequences are closed under surrounding with a corresponding call-return pair.

Lemma 5.26. *If $\pi \in SLSeq(E)$, $e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$ and $(e_{call}, e_{ret}) \in \Phi$, then $e_{call} \cdot \pi \cdot e_{ret} \in SLSeq(E)$.*

With these three observations in mind, it is now relatively clear how the inductive definitions for the partially-balanced sequences have to be modified in order to obtain inductive definitions for their valid counterparts. All we have to do is adapt the respective clause that is concerned with appending balanced sequence to make sure that it maintains validity.

Theorem 5.27. *The same-level sequences are the least subset X of $E^\star$ with the following closure properties:*

$$(\textsc{sl-seq}_{empty})\,\frac{}{\epsilon \in X} \qquad (\textsc{sl-seq}_{intra})\,\frac{\pi \in X \qquad e \in E_{intra}}{\pi \cdot e \in X}$$

$$(\textsc{sl-seq}_{inter})\,\frac{\pi \in X \quad \pi' \in X \quad e_{call} \in E_{call} \quad e_{ret} \in E_{ret} \quad (e_{call}, e_{ret}) \in \Phi}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Proof. By Theorem 2.19, we have to show that

1. $SLSeq(E)$ satisfies $\textsc{sl-seq}_{empty}$,$\textsc{sl-seq}_{intra}$ and $\textsc{sl-seq}_{inter}$.

2. $SLSeq(E)$ is contained in the least subset $X_0 \subseteq E^\star$that satisfies $\textsc{sl-seq}_{empty}$,$\textsc{sl-seq}_{intra}$ and $\textsc{sl-seq}_{inter}$.

We prove the two claims separately.

1. $SLSeq(E)$ satisfies $\textsc{sl-seq}_{empty}$, because ϵ is balanced by $Bal1$ and trivially valid. Moreover, with Lemma 5.24 we see that $SLSeq(E)$ also satisfies $\textsc{sl-seq}_{intra}$. Finally, Lemma 5.25 and Lemma 5.26 imply $\textsc{sl-seq}_{inter}$.

2. By structural induction on $\pi \in Bal(E)$ we show

$$\forall \pi \in Bal(E).\ \pi \in Val(E) \implies \pi \in X_0.$$

This implies $SLSeq(E) = Bal(E) \cap Val(E) \subseteq X_0$.

*Bal*1 If $\pi = \epsilon$, then $\pi \in X_0$ by $\textsc{sl-seq}_{empty}$.

*Bal*2 Let $\pi = \pi' \cdot e$ with $\pi' \in Bal(E)$ and $e \in E_{intra}$. Assume that $\pi \in Val(E)$. Then $\pi' \in Val(E)$ by Theorem 5.18. This implies $\pi' \in X_0$ by induction hypothesis. With $\textsc{sl-seq}_{intra}$, we get $\pi = \pi' \cdot e \in X_0$.

*Bal*3 Let $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$ with $\pi', \pi'' \in Bal(E)$, $e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$ and assume that $\pi \in Val(E)$. Choose i, j such that $\pi^{<i} = \pi'$, $\pi^i = e_{call}$, $\pi^{]i,j[} = \pi''$ and $\pi^j = e_{ret}$. Then we observe that $(i, j) \in \nu_\pi$ by definition. Since $\pi \in Val(E)$, it follows that $(e_{call}, e_{ret}) \in \Phi$. Furthermore, by Theorem 5.18, both π' and π'' are valid. Application of the induction hypothesis to π' and π'' yields $\pi', \pi'' \in X_0$. Now we can apply $\textsc{sl-seq}_{inter}$ obtain that $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret} \in X_0$.

□

Theorem 5.28. *The set of ascending sequences is the least subset X of $E^\star$ with the following properties:*

$$(\textsc{asc-seq}_{empty})\,\frac{}{\epsilon \in X} \qquad (\textsc{asc-seq}_{asc})\,\frac{\pi \in X \qquad e \in E_{intra} \cup E_{ret}}{\pi \cdot e \in X}$$

$$(\textsc{asc-seq}_{sl})\,\frac{\pi \in X \;\; \pi' \in SLSeq(E) \;\; e_{call} \in E_{call} \;\; e_{ret} \in E_{ret} \;\; (e_{call}, e_{ret}) \in \Phi}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Proof. By Theorem 2.19, we have to show that

1. $AscSeq(E)$ satisfies $\textsc{asc-seq}_{empty}$,$\textsc{asc-seq}_{asc}$ and $\textsc{asc-seq}_{sl}$, and

2. $AscSeq(E)$ is contained in the least subset $X_0 \subseteq E^\star$ that satisfies $\textsc{asc-seq}_{empty}$,$\textsc{asc-seq}_{asc}$ and $\textsc{asc-seq}_{sl}$.

We prove the two claims separately.

1. $AscSeq(E)$ satisfies $\textsc{asc-seq}_{empty}$, because $\epsilon \in Left(E)$ by $Left1$ and ϵ is trivially valid. Moreover, with Lemma 5.24 we see that $AscSeq(E)$ also satisfies $\textsc{asc-seq}_{asc}$. Finally, Lemma 5.26 and Lemma 5.25 imply that $AscSeq(E)$ satisfies $\textsc{asc-seq}_{sl}$.

2. Let X_0 be the least subset of $E^\star$ that has the closure properties $\textsc{asc-seq}_{empty}$,$\textsc{asc-seq}_{asc}$ and $\textsc{asc-seq}_{sl}$. By structural induction on $\pi \in Left(E)$ we show

$$\forall \pi \in Left(E).\, \pi \in Val(E) \implies \pi \in X_0.$$

This implies $AscSeq(E) = Left(E) \cap Val(E) \subseteq X_0$.

$Left1$ If $\pi = \epsilon$, then $\pi \in X_0$ by $\textsc{asc-seq}_{empty}$.

$Left2$ Let $\pi = \pi' \cdot e$ with $\pi' \in Left(E)$ and $e \in E_{intra} \cup E_{ret}$. Assume that $\pi \in Val(E)$. Then $\pi' \in Val(E)$ by Theorem 5.18. With $\pi' \in Left(E)$, we can apply the induction hypothesis and get $\pi' \in X_0$. With $\textsc{asc-seq}_{asc}$, we get $\pi = \pi' \cdot e \in X_0$.

$Left3$ Let $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$ with $\pi' \in Left(E), \pi'' \in Bal(E), e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$ and assume that $\pi \in Val(E)$. Define i, j such that $\pi^{<i} = \pi'$, $\pi^i = e_{call}$, $\pi^{]i,j[} = \pi''$ and $\pi^j = e_{ret}$. Then we observe that $(i, j) \in \nu_\pi$ by definition. Since $\pi \in Val(E)$, it follows that $(e_{call}, e_{ret}) \in \Phi$. Furthermore, by Theorem 5.18, both π' and π'' are valid. Application of the induction hypothesis to π' yields $\pi' \in X_0$. Furthermore, π'' is both balanced and valid, which means that $\pi'' \in SLSeq(E)$. Now we can apply ASC-SEQ$_{sl}$ and yield that $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret} \in X_0$.

□

Theorem 5.29. *The set of descending sequences is the least subset X of $E^\star$ with the following properties:*

$$(\textsc{desc-seq}_{empty})\ \frac{}{\epsilon \in X} \qquad (\textsc{desc-seq}_{desc})\ \frac{\pi \in X \qquad e \in E_{intra} \cup E_{call}}{\pi \cdot e \in X}$$

$$(\textsc{desc-seq}_{sl})\ \frac{\pi \in X \ \ \pi' \in SLSeq(E) \ \ e_{call} \in E_{call} \quad e_{ret} \in E_{ret} \ \ (e_{call}, e_{ret}) \in \Phi}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Proof. The proof is very similar to the proof of Theorem 5.28. □

Theorem 5.30. *The valid sequences are exactly the concatenations of the ascending and the descending sequences. In particular:*

1. If π_1 is ascending and π_2 is descending, then $\pi_1 \cdot \pi_2$ is valid.

2. Every valid symbol sequence $\pi \in E^\star$ can be split up into $\pi = \pi_1 \cdot \pi_2$ such that π_1 is ascending and π_2 is descending.

Proof. I show both statements separately.

1. Let $\pi = \pi_1 \cdot \pi_2$ and $(i, j) \in \nu_\pi$. First we observe that it must be either $i, j \in range_\pi(\pi_1)$ or $i, j \in range_\pi(\pi_2)$: Assume, for the purpose of contradiction, that this is not the case. Then, since $i < j$, it must be $i \in range_\pi(\pi_1)$ and $j \in range_\pi(\pi_2)$. But note that π_1 is ascending. Hence, there is $j' \in range_\pi(\pi_1)$ with $(i, j') \in \nu_{\pi_1} \subseteq \nu_\pi$. But then we have $(i, j) \in \nu_\pi$ and $(i, j') \in \nu_\pi$ with $j' < j$, in contradiction to Theorem 5.8. Thus, the assumption must be false and we have either $i, j \in range_\pi(\pi_1)$ or

$i, j \in range_\pi(\pi_2)$. In either case, it follows that $(\pi^i, \pi^j) \in \Phi$, since both π_1 and π_2 are valid.

2. Let π be a valid symbol sequence. By Corollary 5.23, we can split up π into $\pi = \pi_1 \cdot \pi_2$ such that ν_{π_1} is left-total and ν_{π_2} is right-total. Since π is valid, by Theorem 5.18 both π_1 and π_2 are valid. It follows that π_1 is ascending and π_2 is descending.

□

5.2 Interprocedural Graphs

In this section, I introduce the graph model over which I will later define data-flow analyses. This graph model is intended to be general enough to cover both the classical interprocedural control-flow graphs (cf. Definition 3.2) and also other graphs like program dependence graphs.

Definition 5.31. *Given a finite set P of procedure labels, an* interprocedural graph

$$G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi, N_{entry}, N_{exit})$$

consists of

- *a set of nodes $N = \bigcup_{p \in P} N_p$ such that*

$$\forall p, p' \in P.\ p \neq p' \implies N_p \cap N_{p'} = \emptyset$$

- *a set of* intraprocedural edges $E_{intra} = \bigcup_{p \in P} E_p$ *such that*

$$\forall p \in P.\ E_p \subseteq N_p \times N_p \text{ and } \forall p, p' \in P.\ E_p \cap E'_p = \emptyset$$

- *sets $E_{call}, E_{ret} \subseteq \bigcup_{p,p' \in P} N_p \times N'_p$ of* call edges *and* return edges *with the property*

$$E_{intra} \cap E_{call} = E_{intra} \cap E_{ret} = E_{call} \cap E_{ret} = \emptyset,$$

- *a* correspondence relation $\Phi \subseteq E_{call} \times E_{ret}$, *and*

- *sets $N_{entry}, N_{exit} \subseteq N$ of* entry *and* exit nodes*, respectively, such that*
  - *every node with an incoming call edge is an entry node,*

$$\forall n \in N.\ (\exists n' \in N.\ \exists e \in E_{call}.\ n' \xrightarrow{e} n \implies n \in N_{entry}),$$

  - *every node with an outgoing return edge is an exit node, and*

$$\forall n \in N.\ (\exists n' \in N.\ \exists e \in E_{ret}.\ n \xrightarrow{e} n' \implies n \in N_{exit}),$$

  - *no node is an entry node and an exit node at the same time.*

$$N_{entry} \cap N_{exit} = \emptyset.$$

I define $E_{inter} \overset{def}{=} E_{call} \cup E_{ret}$ and call its elements *interprocedural edges*. I refer to each (N_p, E_p) as *procedure graph*. According to the definition of interprocedural graphs, for each $n \in N$ there is exactly one $p \in P$ such that $n \in N_p$. I call p the *procedure of n* and write it as $proc(n)$. Occasionally, I will consider an interprocedural graph as a directed graph (N, E). Then I ignore the additional structure and use

$$E \overset{def}{=} E_{intra} \cup E_{call} \cup E_{ret}.$$

In the rest of this thesis, I will assume that N_{entry} and N_{exit} are given but will not mention them explicitly when specifying an interprocedural graph.

Definition 5.32. *Let $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ be an interprocedural graph. Then I define the following special nodes:*

1. A call node *is a node that has outgoing call edges. I write N_{call} for the set of call nodes.*

2. A return node *is a node that has incoming return edges. I write N_{ret} for the set of return nodes.*

The following example shows that both interprocedural control-flow graphs and interprocedural program dependence graphs are subsumed by Definition 5.31.

Example 5.33. *1. Interprocedural control-flow graphs according to Definition 3.2 can be considered as interprocedural graphs with the following additional properties:*

- *Every procedure graph has exactly one entry node s_p and exactly one exit node e_p.*
- *Φ is a bijective function, i.e. it is left-total, left-unique, right-total and right-unique.*

2. Interprocedural program dependence graphs as described in subsubsection 3.3.2.2 on page 65 can be considered as interprocedural graphs. We consider parameter-in edges as elements of E_{call} and parameter-out edges as elements of E_{ret}. Then the entry nodes are either ordinary procedure entries or formal-in nodes. Exit nodes are either ordinary procedure exits or formal-out nodes. Call nodes are either ordinary call nodes or actual-in nodes. Return nodes are either ordinary return nodes or actual-out nodes.

Definition 5.34. *Let $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ be an interprocedural graph and let $s, t \in N$.*

1. A path $\pi \in Paths(G)$ is called

- same-level path *if $\pi \in SLSeq(E)$,*
- ascending path *if $\pi \in AscSeq(E)$,*
- descending path *if $\pi \in DescSeq(E)$, and*
- valid path *if $\pi \in Val(E)$,*

where $SLSeq(E)$, $AscSeq(E)$, $DescSeq(E)$ are defined with respect to the correspondence relation Φ.

2. I define

$$SL, ASC, DESC, VP : N \times N \to 2^{Paths(G)}$$

by

$$SL(s,t) \overset{def}{=} SLSeq(E) \cap Paths_G(s,t)$$
$$ASC(s,t) \overset{def}{=} AscSeq(E) \cap Paths_G(s,t)$$

$$DESC(s,t) \overset{def}{=} DescSeq(E) \cap Paths_G(s,t)$$
$$VP(s,t) \overset{def}{=} ValSeq(E) \cap Paths_G(s,t)$$

In the following, I will use these functions also as path sets. Particularly, I will write SL for the set of paths π such that there is $s,t \in N$ with $\pi \in SL(s,t)$ – analogously for ASC, DESC and VP.

By combining the inductive definitions of the valid sequences and their partially-balanced variants with the inductive definition of $Paths(G)$, I can derive inductive definitions for *SL*, *ASC*, *DESC* and *VP*, respectively.

Theorem 5.35. *SL is the least subset of $N \times N \times 2^{E^\star}$ with the following closure properties:*

$$(\textsc{sl-empty})\ \frac{}{\varepsilon \in X(s,s)} \qquad (\textsc{sl-intra})\ \frac{\pi \in X(s,t') \quad t' \xrightarrow{e} t \quad e \in E_{intra}}{\pi \cdot e \in X(s,t)}$$

$$(\textsc{sl-sl})\ \frac{\pi \in X(s,n) \quad n \xrightarrow{e_{call}} n_0 \quad \pi' \in X(n_0,n_1) \quad n_1 \xrightarrow{e_{ret}} t \quad (e_{call},e_{ret}) \in \Phi}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X(s,t)}$$

Proof. According to Theorem 2.19, we need to show

1. *SL* has the closure properties sl-empty, sl-intra and sl-sl, and

2. $SL \subseteq X_0$ where X_0 is the least subset of $N \times N \times 2^{E^\star}$ with the closure properties sl-empty, sl-intra and sl-sl.

We show both claims separately.

1. We show that *SL* has the properties sl-empty, sl-intra and sl-sl.

sl-empty Let $s \in N$. Then $\epsilon \in Paths_G(s,s)$ by path-empty. Moreover, $\epsilon \in SLSeq(E)$ by sl-seq$_{empty}$. Together, it follows that $\epsilon \in Paths_G(s,s) \cap SLSeq(E) = SL(s,s)$.

sl-intra Let $s,t',t \in N$, $\pi \in SL(s,t')$ and $t' \xrightarrow{e} t$ with $e \in E_{intra}$. From $\pi \in SL(s,t')$ we have $\pi \in Paths_G(s,t')$ and $\pi \in SLSeq(s,t')$. From $\pi \in Paths_G(s,t')$ and $t' \xrightarrow{e} t$ we get $\pi \cdot e \in Paths_G(s,t)$ and from $\pi \in SLSeq(E)$ and $e \in E_{intra}$ we have $\pi \cdot e \in SLSeq(E)$ by sl-seq$_{intra}$. Together we have $\pi \cdot e \in Paths_G(s,t) \cap SLSeq(E) = SL(s,t)$.

SL-SL Let $s, n, n_0, n_1, t \in N$, $e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$, $\pi \in SL(s,n)$ and $\pi' \in SL(n_0, n_1)$ with $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$ and $(e_{call}, e_{ret}) \in \Phi$. Then we have $\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in SLSeq(E)$ by SL-SEQ$_{inter}$. Moreover, $\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in Paths_G(s,t)$ follows from $SL(s,n) \subseteq Paths_G(s,n)$ and $SL(n_0,n_1) \subseteq Paths_G(n_0,n_1)$ by PATH-EXTEND and Lemma 2.23.

2. Let X_0 be the least subset of $N \times N \times 2^{E^\star}$ with the closure properties SL-EMPTY, SL-INTRA and SL-SL. Using the induction principle induced by Theorem 5.27, we show

$$\forall \pi \in SLSeq.\ \forall s,t \in N.\ \pi \in Paths_G(s,t) \implies \pi \in X_0(s,t)$$

Let $\pi \in SLSeq$ and $s,t \in N$ with $\pi \in Paths_G(s,t)$. Then we need to show $\pi \in X_0(s,t)$.

SL-SEQ$_{empty}$: If $\pi = \epsilon$, then from $\pi \in Paths_G(s,t)$ we get $s = t$, hence $\pi \in X_0(s,t)$ by SL-EMPTY.

SL-SEQ$_{intra}$: Assume $\pi = \pi' \cdot e$ with $\pi' \in SLSeq$ and $e \in E_{intra}$. By Lemma 2.24, from $\pi \in Paths_G(s,t)$ we yield $t' \in N$ with $\pi' \in Paths_G(s,t')$ and $t' \overset{e}{\to} t$. By induction hypothesis, we get $\pi' \in X_0(s,t')$. This implies $\pi \in X_0(s,t)$ by SL-INTRA.

SL-SEQ$_{inter}$: Assume $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$ with $\pi', \pi'' \in SLSeq$ and $(e_{call}, e_{ret}) \in \Phi$. By splitting up π and applying Lemma 2.24 to $\pi \in Paths_G(s,t)$, we obtain n, n_0, n_1 with $\pi' \in Paths_G(s,n)$, $n \overset{e_{call}}{\to} n_0$, $\pi'' \in Paths_G(n_0,n_1)$ and $n_1 \overset{e_{ret}}{\to} t$. By induction hypothesis, applied to π' and π'', we get $\pi' \in X_0(s,n)$ and $\pi'' \in X_0(n_0,n_1)$. This implies $\pi \in X_0(s,t)$ by SL-SL.

□

Theorem 5.36. *ASC is the least element $X \in N \times N \to 2^{E^\star}$ with the following closure properties*

$$(\textsc{asc-empty})\ \frac{}{\varepsilon \in X(s,s)} \qquad (\textsc{asc-asc})\ \frac{\pi \in X(s,t') \quad t' \overset{e}{\to} t \quad e \in E_{intra} \cup E_{ret}}{\pi \cdot e \in X(s,t)}$$

$$(\textsc{asc-sl})\ \frac{\pi \in X(s,n) \quad n \overset{e_{call}}{\to} n_0 \quad \pi' \in SL(n_0,n_1) \quad n_1 \overset{e_{ret}}{\to} t \quad (e_{call}, e_{ret}) \in \Phi}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X(s,t)}$$

Proof. This can be shown analogously to Theorem 5.35. □

Theorem 5.37. *DESC is the least element $X \in N \times N \to 2^{E^\star}$ with the following closure properties*

$$(\textsc{desc-empty})\,\frac{}{\varepsilon \in X(s,s)}$$

$$(\textsc{desc-desc})\,\frac{\pi \in X(s,t') \quad t' \overset{e}{\to} t \quad e \in E_{intra} \cup E_{call}}{\pi \cdot e \in X(s,t)}$$

$$(\textsc{desc-sl})\,\frac{\pi \in X(s,n) \quad n \overset{e_{call}}{\to} n_0 \quad \pi' \in SL(n_0,n_1) \quad n_1 \overset{e_{ret}}{\to} t \quad (e_{call}, e_{ret}) \in \Phi}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X(s,t)}$$

Proof. This can be shown analogously to Theorem 5.35. □

Theorem 5.38. *For all $s,t \in N$, $VP(s,t)$ can be characterized as follows:*

(valid-asc-desc)
$$VP(s,t) = \{\pi_1 \cdot \pi_2 \mid \exists n \in N. \pi_1 \in ASC(s,n) \wedge \pi_2 \in DESC(n,t)\}$$

Proof. This follows from Theorem 5.30, Lemma 2.23 and Lemma 2.24. □

Theorem 5.39. *The following statemens are true.*

1. Valid paths are closed under taking sub-paths, in particular under taking suffixes and prefixes.

2. Ascending paths are closed under taking suffixes.

3. Descending paths are closed under taking prefixes.

4. Taking a prefix of an ascending path or a suffix of a descending path yields a valid path.

Proof. This can be derived by combining Remark 2.25 and Theorem 5.18. □

Theorem 5.40. *For all $s,t,t' \in N$, we have*

(5.4) $\forall \pi \in ASC(s,t).\ \forall \pi' \in ASC(t,t').\ \pi \cdot \pi' \in ASC(s,t')$

(5.5) $\forall \pi \in DESC(s,t).\ \forall \pi' \in DESC(t,t').\ \pi \cdot \pi' \in DESC(s,t')$

(5.6) $\forall \pi \in VP(s,t).\ \forall \pi' \in SL(t,t').\ \pi \cdot \pi' \in SL(s,t')$

Proof. The first two statements can be shown using Lemma 2.23, Theorem 5.20, Theorem 5.21 and the definitions of ascending and descending paths.

For the third statements, by Theorem 5.38 it suffices to show the respective property for descending paths. But this follows from Theorem 5.37.

□

With an additional regularity restriction of Φ, I can show that same-level paths end in the same procedure that they started in.

Remark 5.41. *Assume that the correspondence relation Φ has the following property*

$$(e_{call}, e_{ret}) \in \Phi \implies proc(src(e_{call})) = proc(tgt(e_{ret})) \tag{5.7}$$

Then the following statement holds: If π is a same-level path from s to t, then $proc(s) = proc(t)$.

Proof. By induction on $\pi \in SL(s, t)$. □

Table 5.1 shows an overview of some notions of validness in the literature. The publications there can roughly be split into two groups: Those that characterize valid paths as suffixes of descending paths and those that characterize them as concatenations of ascending and descending paths. While the former notion is sensible for contexts in which valid paths are supposed to correspond to actual program executions[28], the latter notion is more general. According to Theorem 5.39, suffixes of descending paths are always valid. Conversely, however, interprocedural graphs in general may contain valid paths that are not suffixes of any descending path.

The two notions coincide if additional assumptions are made, as Theorem 5.42 states. Note that such assumptions are *not* made for the rest of this thesis.

Theorem 5.42. *Let $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ be an interprocedural graph. Assume that every procedure graph G_p has a distinguished entry s_p and that the following conditions hold:*

[28] They are also called *realizable* paths for this reason.

definition in literature	notion of validness
[86, Definition 2.13]	suffixes of descending paths
[137, p. 4]	suffixes of descending paths
[164, p. 110ff]	suffixes of descending paths
[136, Definition 2.3]	suffixes of descending paths
[65, Definition 2.3]	valid paths
[138, Definition 2.1]	valid paths

Table 5.1: Notions of validness in the literature

There is a procedure main $\in P$ *such that the entry* s_{main} *of* G_{main} *reaches every* s_p *using a descending path.*

(main-reach) $\quad \exists main \in P.\ \forall p' \in Proc.\ SL(s_{main}, s_p) \neq \emptyset$

In every procedure graph G_p*, every node* $n \in N_p$ *is same-level reachable from* s_p*.*

(sl-reach) $\quad \forall p \in .\ \forall n \in N_p.\ SL(s_p, n) \neq \emptyset$

For every return edge e, there is a corresponding call edge that enters the procedure that e starts in:

(ret-call) $\quad \forall e \in E_{ret}.\ \exists e' \in E_{call}.(e', e) \in \Phi \wedge tgt(e') = s_{proc(src(e))}$

Then every valid path is the suffix of a descending path starting in s_{main}*:*

$$\begin{aligned}(5.8)\qquad & \forall \pi_2 \in VP.\ \forall m, n \in N.\\ & \pi_2 \in Paths_G(m, n)\\ & \quad \Longrightarrow \exists \pi_1 \in Paths_G(s_{main}, m).\ \pi_1 \cdot \pi_2 \in DESC(s_{main}, n).\end{aligned}$$

Proof. It suffices to show that (5.8) holds for the ascending paths:

$$\begin{aligned}(5.9)\qquad & \forall \pi_2 \in ASC.\ \forall m, n \in N.\\ & \pi_2 \in Paths_G(m, n)\\ & \quad \Longrightarrow \exists \pi_1 \in Paths_G(s_{main}, m).\ \pi_1 \cdot \pi_2 \in DESC(s_{main}, n).\end{aligned}$$

Suppose that (5.9) is true and let $\pi \in VP(m, n)$ be a valid path. Then by Theorem 5.38, we obtain $m_0 \in N$, $\pi_1 \in ASC(m, m_0)$, $\pi_2 \in DESC(m_0, n)$ such

that $\pi = \pi_1 \cdot \pi_2$. Now we apply (5.9) to π_1 and obtain $\sigma \in Paths_G(s_{main}, m)$ such that $\sigma \cdot \pi_1 \in DESC(s_{main}, m_0)$. With $\pi_2 \in DESC(m_0, n)$, we have $\sigma \cdot \pi_1 \cdot \pi_2 \in DESC(s_{main}, n)$, as desired.
It remains to show (5.9). We proceed by induction on the number k of unmatched return positions in $\pi_2 \in ASC$.

base case ($k = 0$): If π_2 has no unmatched return positions, then $\pi_2 \in SL$, because $\pi_2 \in ASC$, which means that π_2 has also no unmatched call positions. Now let $m, n \in N$ such that $\pi_2 \in SL(m, n)$ and let $p = proc(m)$. We apply (main-reach) to s_p and obtain $\pi_1' \in DESC(s_{main}, s_p)$. Moreover, we apply (sl-reach) to $m \in N_p$ and obtain $\pi_1'' \in SL(s_p, m)$. Now consider $\pi_1 \overset{def}{=} \pi_1' \cdot \pi_1''$. With the help of Theorem 5.40, from $\pi_1' \in DESC(s_{main}, s_p)$, $\pi_1'' \in SL(s_p, m)$ and $\pi_2 \in SL(m, n)$, we conclude $\pi_1' \cdot \pi_1'' \cdot \pi_2 = \pi_1 \cdot \pi_2 \in DESC(s_{main}, n)$, as desired.

induction step ($k \to k + 1$): Let π_2 be an ascending path with $k + 1$ unmatched return positions. The induction hypothesis states that the claim is true for all ascending paths with k unmatched return positions. Let $m, n \in N$ such that $\pi_2 \in ASC(m, n)$ and let j_0 be the least unmatched return position in π_2. Then we write π_2 as

$$\pi_2 = \pi_2' \cdot e_{ret} \cdot \pi_2''$$

where

$$\begin{aligned} \pi_2' &\overset{def}{=} \pi_2^{<j_0} \in Paths_G(m, n_0), \\ e_{ret} &\overset{def}{=} \pi_2^{j_0} \text{ with } n_0 \overset{e_{ret}}{\to} n_1, \text{ and} \\ \pi_2'' &\overset{def}{=} \pi_2^{>j_0} \in Paths_G(n_1, n). \end{aligned}$$

for some $n_0, n_1 \in N$, $e_{ret} \in E_{ret}$.

Due to the maximality property of j_0, π_2' contains no unmatched return positions. Moreover, it cannot contain any unmatched call position. Assume, for the purpose of contradiction, that π_2' contains an unmatched call position. Then the greatest such position is also a call position in π_2, which would be matched by the return position j_0, in contradiction to the choice of j_0 as unmatched return position. Hence, the assumption is false, which means that all call positions in π_2' must be matched. Hence, $\pi_2' \in SL(m, n_0)$. Moreover, π_2'' is a suffix of the ascending path π_2 and therefore ascending because of Theorem 5.39.

Now let $p = proc(m)$. We apply (ret-call) to e_{ret} and obtain $m_0 \overset{e_{call}}{\to} s_p$ such that $(e_{call}, e_{ret}) \in \Phi$. Moreover, we apply (sl-reach) to m and obtain $\pi_0 \in SL(s_p, m)$. Now consider the path

$$\theta \overset{def}{=} e_{call} \cdot \pi_0 \cdot e_{ret} \cdot \pi_2 = e_{call} \cdot \pi_0 \cdot e_{ret} \cdot \pi_2' \cdot e_{ret} \cdot \pi_2''$$

from m_0 to n_0: It is the concatenation of the same-level path $e_{call} \cdot \pi_0 \cdot e_{ret}$ and the ascending path π_2 and therefore ascending itself. Moreover, it contains k unmatched return positions because by choice of e_{call}, we have $(0, 1 + j_0) \in \nu_\theta$. Hence, we can apply the induction hypothesis to θ and obtain $\sigma \in Paths_G(s_{main}, m_0)$ such that

$$\sigma \cdot \theta \in DESC(s_{main}, n).$$

With

$$\pi_1 \overset{def}{=} \sigma \cdot e_{call} \cdot \pi_0,$$

we have $\pi_1 \cdot \pi_2 = \sigma \cdot \theta \in DESC(s_{main}, n)$, as desired.

□

5.3 Data-Flow Analysis on Interprocedural Graphs

Next, I define data-flow analysis instances for interprocedural graphs. This is a general and formal version[29] of the notions I have already described in subsubsection 3.2.2.1 on page 48.

Definition 5.43. *A data-flow analysis instance $\mathcal{F} = (G, L, F, \rho)$ consists of*

- *an interprocedural graph $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$,*

[29] Note that Definition 5.43 omits the initial information *init*. I will discuss this slight modification in subsection 9.2.2.

- *a complete lattice* $(L, \leq)$,
- *a set* $F \subseteq L \rightarrow_{mon} L$ *that contains* id_L*, is closed under function composition and forms a complete lattice with point-wise ordering, and*
- *a function* $\rho : E \rightarrow (L \rightarrow_{mon} L)$ *which assigns a monotone transfer function to each edge of* G*.*

Instead of $\rho(e)$ I will write f_e. I extend this notation to arbitrary paths by defining

(5.10) $$f_\epsilon \overset{def}{=} id$$

(5.11) $$f_{\pi \cdot e} \overset{def}{=} f_e \circ f_\pi.$$

Next, I want to introduce my generalized variant of the *merge-over-all-valid-paths* solution $MOVP$. This version of $MOVP$ is more general in two aspects:

1. it not only considers paths that start in a fixed entry node, but takes the starting node as additional argument,

2. it not only considers descending paths but also paths with an ascending prefix.

Moreover, I explicitly do not make any assumptions about reachability. Hence, the value $MOVP(s,t)$ not only reflects the resulting value if we merge the path functions for all valid paths from s to t, but it also communicates whether $VP(s,t)$ is empty or not. In traditional data-flow analyses, this is never the case because they only consider s, t where $VP(s,t)$ is not empty. However, $\perp_F$ can still be a valid analysis result value, even if $VP(s,t) \neq \emptyset$.

Hence, in order to be able to distinguish between analysis results for non-empty and empty path sets, I adjoin F with an additional element $\boxtimes$ that represents *undefinedness*. Before I discuss the properties of $\boxtimes$, I give the definition of $MOVP$ in Definition 5.44. Since I will also consider *Merge-Over-$\mathcal{P}$-solutions* for other sets of paths than VP, Definition 5.44 is more general than needed right now.

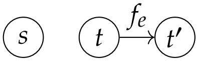


Figure 5.4: Example in which it is important that $\boxtimes$ kills everything – e is an intraprocedural edge

Definition 5.44. *Let $\mathcal{P} \subseteq Paths(G)$ be a set of paths. The* Merge-Over-$\mathcal{P}$-solution

$$\mathrm{MO}\mathcal{P} : N \times N \to F_{\boxtimes}$$

is defined by

$$\mathrm{MO}\mathcal{P}(s,t) = \bigsqcup_{\pi \in \mathcal{P} \cap Paths_G(s,t)} f_\pi.$$

Now I discuss the assumptions about and properties of $\boxtimes$ more closely. Firstly, I assume that $\boxtimes$ is smaller than any element of F.

(5.12) $\quad \forall f \in F_{\boxtimes}.\ \boxtimes \leq f$

(5.13) $\quad \forall f \in F.\ \boxtimes \neq f$

This means that it never coincides with any f_π:

(5.14) $\quad \forall e \in E.\ f_e \neq \boxtimes$

(5.15) $\quad \forall f, g \in F.\ f \circ g \neq \boxtimes$

These properties ensure that

$$MOVP(s,t) = \boxtimes \text{ if and only if } VP(s,t) = \emptyset.$$

Moreover, I extend the function composition on F to $F_{\boxtimes}$ in a way such that $\boxtimes$ kills every value already computed:

(5.16) $\quad \forall f \in F_{\boxtimes}.\ f \circ \boxtimes = \boxtimes \circ f = \boxtimes.$

I make this assumption because I want to compose different values of *MOVP* consistently without explicitly thinking about $\boxtimes$ or whether the corresponding *VP* sets are empty or not. For illustration, consider the graph in Figure 5.4. Since $VP(s,t') = \emptyset$, we have $MOVP(s,t') = \boxtimes$. (5.16) ensures that I get the same result by computing $f_e \circ MOP(s,t)$.
Instead of adjoining F with $\boxtimes$, I could have made two alternative choices that one could make to achieve the same goals. For one, I also could

use partial functions $N \times N \to F$ for my solution space. Then I could say that (s, t) does not belong to the domain of *MOVP* if *VP* is empty. In my approach, I communicate this by letting $MOVP(s, t)$ be ⊠ in such cases. The other approach would be to restrict F to only allow *strict* functions. A function $f : L \to L$ is called *strict*, if $f(x) = \bot_L$ is equivalent to $x = \bot_L$. If all $f \in F$ are strict, then I can use $\lambda x.\bot$ as bottom element of F to represent the merge over the empty path set.

I decided to use ⊠ explicitly because I did not want to restrict the transfer functions but also did not want to introduce additional notational overhead for dealing with partial functions. I will make use of one additional convention: At several places, I will use elements $\psi : N \times N \to F_{\boxtimes}$ as functions. Whenever I do this and do not explicitly discuss whether $\psi = \boxtimes$ or not, I will silently assume $\psi \neq \boxtimes$. For instance, if I state equations such as $\psi(x) = y$, then I will silently assume that ψ is indeed a function.

I conclude this section by considering *distributivity*, an important property of transfer functions and frameworks. *Distributivity* ensures that the constraint systems that I show in chapter 6 coincide with their respective MO$\mathcal{P}$ solutions.

Definition 5.45. • *Let L be a complete lattice and $F \subseteq L \to_{mon} L$ be a complete lattice of monotone functions that is closed under function composition. Then*

1. *$f \in F$ is called strict, if*

$$f \circ \bot = \bot \tag{5.17}$$

2. *$f \in F$ is called* distributive, *if*

$$\forall g, h \in F. f \circ (g \sqcup h) = f \circ g \sqcup f \circ h \tag{5.18}$$

3. *$f \in F$ is called* positive-distributive, *if*

$$\forall A \subseteq F. A \neq \emptyset \implies f \circ \bigsqcup A = \bigsqcup \{f \circ g \mid g \in A\} \tag{5.19}$$

4. *$f \in F$ is called* universally distributive, *if it is strict and positive-distributive, i.e. if*

$$\forall A \subseteq F. f \circ \bigsqcup A = \bigsqcup \{f \circ g \mid g \in A\} \tag{5.20}$$

5. *F is called strict, distributive, positive-distributive, universally distributive if all $f \in F$ have the respective property.*

- *A data-flow framework instance $\mathcal{F} = (G, L, F, \rho)$ is called strict, distributive, positive-distributive, universally distributive if F has the respective property.*

In the following, I will use "u.d." to abbreviate the term "universally distributive".

The data-flow framework instances that I consider in this thesis are automatically strict because of (5.13), (5.12) and (5.16). Under these assumptions, a data-flow analysis framework instance is universally distributive if $F_{\boxtimes}$ is positive-distributive. However, it is worth mentioning that this applies to $F_{\boxtimes}$ and *not* to F: In order for $F_{\boxtimes}$ to be positive-distributive, F needs to be universally distributive. In this sense, adjoining $\boxtimes$ does not magically make F strict, but at least makes it possible to distinguish reachable nodes from unreachable parts of the given graph. In chapter 7, I will only consider data-flow analysis framework instances $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ in which $F_{\boxtimes}$ additionally satisfies (ACC). For such instances, distributivity is equivalent to universal distributivity.

5.4 Example Instances

In this section, I want to discuss various data-flow analyses that can be expressed and solved using the abstract data-flow framework presented earlier.

5.4.1 Traditional Data-Flow Analyses on Interprocedural Control-Flow Graphs

According to Example 5.33, interprocedural control-flow graphs can be regarded as interprocedural graphs. Hence, all traditional data-flow analyses in the sense of subsubsection 3.2.2.2 can be expressed as data-flow analysis instances in the sense of Definition 5.43. The solution *MOVP* defined in Definition 5.44 is more general, since it does not use the entry s_{main} of the *main* procedure as starting point but rather considers all paths between arbitrary nodes s and t. Moreover, *MOVP* merges over all valid paths and not over all descending paths and also takes reachability into account. I already discussed this in subsection 5.4.2 and section 5.3.

5.4.2 Slicing

As I already pointed out in subsection 3.3.4, slicing on program dependence graphs can be expressed as *reachability*, a very simple data-flow analysis instance. We can generalize this further by considering slicing on interprocedural graphs, which are a generalization of interprocedural program dependence graphs according to Example 5.33.
Let $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ be an interprocedural graph.
As for IPDGs, the *backwards slice* of a node $n \in N$ can be characterized as the set of nodes which may reach n by a valid path. Analogously, the *forward slice* can be characterized as the set of nodes which may be reached by a valid path.

$$BS(n) = \{m \in N \mid VP(m, n) \neq \emptyset\} \tag{5.21}$$

$$FS(n) = \{m \in N \mid VP(n, m) \neq \emptyset\} \tag{5.22}$$

As in subsection 3.3.4, the reachability data-flow analysis instance (G, L, F, ρ) is defined as follows:

$$\begin{aligned} L &\overset{def}{=} \{\bot, \top\} \\ F &\overset{def}{=} \{\lambda x.\bot, id, \lambda x.\top\} \\ \rho &\overset{def}{=} \lambda e.id \end{aligned} \tag{5.23}$$

Now, observe that

$$\forall \pi \in VP(s, t). f_\pi = id$$

Hence, we have

$$MOVP^{\mathcal{F}}(s, t) = \bigsqcup_{\pi \in VP(s,t)} f_\pi = \begin{cases} id & \text{if } VP(s, t) \neq \emptyset \\ \boxtimes & \text{otherwise} \end{cases}$$

This allows us to rewrite BS and FS as

$$BS(n) = \{m \in N \mid MOVP(m, n) \neq \boxtimes\}$$
$$FS(n) = \{m \in N \mid MOVP(n, m) \neq \boxtimes\}$$

So, by computing $MOVP$, we can extract BS and FS.

5.4.3 Chopping

Chopping [48, 138] was proposed as a means to make slicing on PDGs more focussed. Roughly, the idea is not to go back or forward from a single node, but to consider all nodes that may lie on paths *between* two nodes. Like slicing, chopping can also be considered on a general interprocedural graph $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$. Given $s, t \in N$, the chop between s and t is defined as

$$CH(s,t) \overset{def}{=} \{n \in N \mid \exists \pi \in VP(s,t).\, n \in nodes(\pi)\},$$

where for a sequence of edges $\pi \in E^{\star}$, $nodes(\pi)$ is the set of nodes that occur in the symbol sequence:

$$nodes(\pi) \overset{def}{=} \{n \in N \mid \exists i \in range(\pi).\, n = src(\pi^i) \lor n = tgt(\pi^i)\}. \tag{5.24}$$

Note that chops and slices actually have a strong connection. For $s \neq t$, we have $s \in BS(t)$ if and only if $CH(s,t) \neq \emptyset$. Hence, applications such as slicing-based information flow control can also be expressed using chopping.
Chopping can be expressed as a data-flow analysis instance as follows.

$$\begin{aligned} L &\overset{def}{=} (2^N, \subseteq) \\ F &\overset{def}{=} \{\lambda X.(X \cap A) \cup B \mid A, B \in 2^N\} \\ f_e(X) &\overset{def}{=} X \cup \{src(e), tgt(e)\} \end{aligned} \tag{5.25}$$

It can easily be seen that (G, L, F, ρ) is indeed a data-flow framework instance with respect to Definition 5.43.
Moreover, using induction on the length of paths, we can show that

$$\forall \pi \in Paths_G(s,t).f_\pi(X) = X \cup nodes(\pi). \tag{5.26}$$

Hence, we have

$$\begin{aligned}
CH(s,t) &= \bigcup_{\pi \in VP(s,t)} nodes(\pi) && \{\text{ definition }\} \\
&= \bigcup_{\pi \in VP(s,t)} f_\pi(\emptyset) && \{\text{ by (5.26) }\} \\
&= \left(\bigsqcup_{\pi \in VP(s,t)} f_\pi \right)(\emptyset) && \{\text{ rewriting }\} \\
&= MOVP(s,t)(\emptyset). && \{\text{ definition }\}
\end{aligned}$$

Analogously, we can consider the *edge chop*

(5.27) $$ECH(s,t) \overset{def}{=} \{n \in N \mid \exists \pi \in VP(s,t).\, n \in edges(\pi)\},$$

where

(5.28) $$edges(\pi) \overset{def}{=} \{e \in E \mid \exists i \in range(\pi).\, e = \pi^i.\}$$

Edge chops can be computed using the following data-flow analysis instance.

(5.29) $$\begin{aligned}
L &\overset{def}{=} (2^E, \subseteq) \\
F &\overset{def}{=} \{\lambda X.(X \cap A) \cup B \mid A, B \in 2^E\} \\
f_e(X) &\overset{def}{=} X \cup \{e\}
\end{aligned}$$

Like for node chops, it can be easily verified that this indeed satisfies Definition 5.43.

5.4.4 Strong Bridges and Strong Articulation Points

Strong bridges [95] are a graph theoretical concept that describes the connectivity properties of a given directed graph. Intuitively, a strong bridge is an edge that disconnects a graph if it is removed. Strong bridges are interesting for applications such as slicing-based information flow control. Consider a PDG $G = (N, E)$ and two nodes $s, t \in N$. Remember that slicing-based IFC works by computing the backwards slice $BS(t)$ of t and checking whether $s \in BS(t)$ or not. If $s \notin BS(t)$, then it is definitely not the case that s influences t in any way and if $s \in BS(t)$, this may be the case. Equivalently, we can also compute the chop $CH(s, t)$ of s and t and consider the *chop graph* $C_{s,t} = (CH(s, t), ECH(s, t))$. Now, suppose that $C_{s,t}$ contains a bridge and we can show that this bridge is actually not justified, i.e. G would not contain it if it was obtained using a more precise analysis. Then this means that s actually does not influence t. Hence, strong bridges can be a tool for eliminating false alarms. This kind of reasoning has been applied to PDGs by Beckert, Bischof et al. [26].

Strong bridges as considered by, e.g., Italiano et al. [95], can be generalized to interprocedural graphs.

Given an interprocedural graph $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ and $s, t \in N$, a *strong bridge* with respect to s and t is an edge $e \in E$ such that $e \in edges(\pi)$ for all $\pi \in VP(s, t)$.

The set of strong bridges is then defined as

$$SB(s, t) = \{e \in E \mid \forall \pi \in VP(s, t).\ e \in edges(\pi)\}. \tag{5.30}$$

Note that this is dual to the edge chops defined in (5.27). Hence, a data-flow analysis instance for expressing strong bridges can be obtained by reversing the partial order used for the data-flow analysis defined by (5.29):

$$\begin{aligned} L &= (2^E, \supseteq) \\ F &= \{\lambda X.(X \cap A) \cup B \mid A, B \subseteq E\} \\ f_e(A) &= A \cup \{e\} \end{aligned} \tag{5.31}$$

It is easy to see that F is indeed closed under composition and contains the identity function. We also note that if $f = \lambda X.(X \cap A_1) \cup B_1$ and $g = \lambda X.(X \cap A_2) \cup B_2$, then

$$(f \sqcup g)(X) = f(X) \cap g(X) = (X \cap A_3) \cup B_3$$

with $A_3 = (A_1 \cap A_2) \cup (A_1 \cap B_2) \cup (A_2 \cap B_1)$ and $B_3 = B_1 \cap B_2$.
Due to the finiteness of E, this makes F a complete lattice with respect to the functional join induced by the join $\cap$ of $(2^E, \supseteq)$.
The node analogon to strong bridges are *strong articulation point*. A strong articulation point is a node that disconnects a given graph if removed. A data-flow analysis instance that is able to compute the strong articulation points of a given graph can be specified as follows:

$$\begin{aligned} L &= (2^N, \supseteq) \\ F &= \{\lambda X.(X \cap A) \cup B \mid A, B \subseteq N\} \\ f_e(A) &= A \cup \{src(e), tgt(e)\}. \end{aligned} \tag{5.32}$$

5.4.5 Restricting to Paths With Regular Properties

The reachability analysis shown in subsection 5.4.2 can be generalized to language-restricted reachability. In this subsection, I am going to demonstrate this for forward reachability and regular languages. I also will show two special cases of this.
Remember that a *finite automaton* is a quintuple

$$\mathcal{A} = (Q, A, q_0, \Delta_0, Q_F). \tag{5.33}$$

The components of are

- a set Q of *states*,
- an alphabet A,
- a distinguished state $q_0 \in Q$ which is called the *initial state* of $\mathcal{A}$,
- a set $\Delta_0 \subseteq Q \times A \times Q$ of *transition rules*,
- a set $Q_F \subseteq Q$ of *final states*.

Now, I recall the definition of the *language recognized by a finite automaton* $\mathcal{A} = (Q, A, q_0, \Delta_0, Q_F)$. For this, I define

$$\Delta : E^\star \to 2^Q \to 2^Q$$

by

(5.34) $$\Delta(\epsilon, S) = S$$

(5.35) $$\Delta(\pi \cdot e, S) = \{q' \in Q \mid \exists q \in \Delta(\pi, S).\, q' \in \Delta_0(q, e)\}$$

Then the *language recognized by $\mathcal{A}$* can be defined as

(5.36) $$\mathcal{L}(\mathcal{A}) = \{\pi \in E^\star \mid \Delta(\pi, \{q_0\}) \cap Q_F \neq \emptyset\}$$

It is well-known that the regular languages are exactly the languages that are recognized by a finite automaton.

Now let $R \subseteq E^\star$ be a regular language of edge sequences. I define the *R-restricted forward slice of s* by

(5.37) $$FS(s, R) \overset{def}{=} \{t \in N \mid VP(s, t) \cap R \neq \emptyset\}.$$

In the following, I show how to compute $FS(s, R)$ using a data-flow analysis instance. For this, let $\mathcal{A} = (Q, E, q_0, \Delta_0, Q_F)$ be a finite automaton with

(5.38) $$\mathcal{L}(\mathcal{A}) = R.$$

The idea is that the information computed at each node consists of the set of states in Q that are reachable during a run of $\mathcal{A}$. Consequently, the transfer functions are defined by Δ. Before I define this data-flow analysis instance, I need some helping notions and observations. Firstly, I observe that for each $e \in E$, the function $\lambda A.\, \Delta(e, A)$ is monotone: If $A \subseteq B$, then $\Delta(e, A) \subseteq \Delta(e, B)$. Secondly, for a complete lattice L and a set $X \subseteq L \to_{mon} L$ of monotone functions on X, I define $cl(X) \subseteq L \to_{mon} L$ as the least (with respect to set inclusion) subset Y of $L \to_{mon} L$ that (a) contains X, (b) is a complete lattice and (c) is closed under function composition. It is easy to see that $cl(X)$ is unique and always exists.

Now we are ready to define the data-flow analysis instance.

$$L = 2^Q \tag{5.39}$$

$$F = cl(\{\lambda A.\ \Delta(e, A) : 2^Q \to 2^Q \mid e \in E\}) \tag{5.40}$$

$$f_e = \lambda A.\Delta(e, A) : 2^Q \to 2^Q \tag{5.41}$$

Theorem 5.46. *For every $s \in N$, we have*

$$FS(s, R) = \{t \in N \mid MOVP(s, t)(\{q_0\}) \cap Q_F \neq \emptyset\} \tag{5.42}$$

Proof. Let $s \in N$. Firstly, by induction on $\pi \in E^\star$, it can easily be shown that

$$\forall \pi \in E^\star . \forall A \subseteq Q.\ f_\pi(A) = \Delta(\pi, A). \tag{5.43}$$

Secondly, we observe that

$$\mathrm{MO}VP(s, t)(\{q_0\}) = \bigcup_{\pi \in VP(s,t)} f_\pi(q_0). \tag{5.44}$$

This follows from the definition of the given data-flow framework instance and the definition of $MOVP$.
From (5.43) and (5.44), we can derive the claimed set equality.

$$\begin{aligned}
t \in FS(s, R) &\iff VP(s, t) \cap R \neq \emptyset && \{\text{ definition }\}\\
&\iff \exists \pi \in VP(s, t).\ \pi \in R && \{\text{ rewriting }\}\\
&\iff \exists \pi \in VP(s, t).\ \Delta(\pi, \{q_0\}) \cap Q_F \neq \emptyset && \{\ (5.38)\ \}\\
&\iff \exists \pi \in VP(s, t).\ f_\pi(\{q_0\}) \cap Q_F \neq \emptyset && \{\ (5.43)\ \}\\
&\iff MOVP(s, t)(\{q_0\}) \cap Q_F \neq \emptyset && \{\ (5.44)\ \}
\end{aligned}$$

□

5.4.5.1 Barrier Slicing

A simple yet important special case of language-restricted slices is *barrier slicing*. Barrier slicing was introduced for PDGs by Krinke [111] as a means to make slicing more focussed. The idea is to introduce a *barrier*, i.e. a set B of nodes that is not to be passed. In the following, I demonstrate that

barrier slicing can be expressed as a regular language-restricted reachability analysis instance on interprocedural graphs.
Given an interprocedural graph $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ (e.g., a program dependence graph), the *(backwards) barrier slice* of $t \in N$ with respect to B is defined to be the set

$$BBS(t, B) \overset{def}{=} \{s \in N \mid \exists \pi \in VP(s, t).\, nodes(\pi) \cap B = \emptyset\}. \tag{5.45}$$

Analogously, the *forward barrier slice* can be defined as

$$FBS(s, B) \overset{def}{=} \{t \in N \mid \exists \pi \in VP(s, t).\, nodes(\pi) \cap B = \emptyset\}. \tag{5.46}$$

The property of being a sequence from $E^\star$ that avoids nodes from B can be expressed as the language

$$A(B) \overset{def}{=} \{\pi \in E^\star \mid \forall i \in range(\pi).\, src(\pi^i) \notin B \wedge tgt(\pi^i) \notin B\}. \tag{5.47}$$

It can easily be seen that $A(B)$ is regular. Hence $FBS(s, B)$ and $BBS(t, B)$ can both be determined using a regular language-restricted reachability analysis.

5.4.5.2 Explicit Information Flow

As a second example for regular language restricted reachability analysis, I want to consider a heuristic property of slices that I call *explicit information flow*. This property is based on the observation that static helper analyses may cause the insertion of spurious control dependencies. Such control dependencies can easily lead to program dependence graphs in which everything that happens after some critical statement is control-dependent on this statement, solely because of the fact that this statement may fail due to an exception that could not be ruled out.
As an example, consider Figure 5.5b. The array access does not fail and hence the program should be secure. But with a static analysis that is too imprecise to prove this, the resulting PDG contains a control dependency between the array access and the print statement. Moreover, whether the

```
1  int h = inputPIN();
2  int x;
3  if (h > 42) {
4    arr[i] = 17;
5    x = 17;
6  }
7  print(x);
```

(a)

```
1  int h = inputPIN();
2
3  if (h > 42) {
4    arr[i] = 17;
5
6  }
7  print("OK");
```

(b)

Figure 5.5: Analysis precision and control dependencies – assume that $arr \neq null$ and $0 \leq i < arr.length$

array access happens depends on the secret value. Hence, in this PDG, line 1 is connected to 7.

In contrast, the PDG of Figure 5.5a contains a path from the secret source to the public sink, even if a static analysis could prove that the array access never fails. The path has the property that it *ends in a data dependency*, where as the Figure 5.5b does not.

Explicit information flow only considers PDG paths that end in a data dependency. This way, PDG paths such as the one in Figure 5.5b are ignored, whereas the path in Figure 5.5a is covered.

We also can ignore the path in Figure 5.5b by simply ignoring *all* control dependencies (like we did in one variant of the SHRIFT approach in section 4.7), but then we would also ignore the path in Figure 5.5a that is in a sense more direct.

I consider explicit information flow as an example for a helper analysis for the further analysis of a PDG's valid paths. The setting that I have in mind is that a PDG-based information flow control tool like Joana fails to verify a given information flow requirement and the analyst tries to find out why. If Joana succeeds to verify the requirement with restriction to explicit information flow, then this may indicate that Joana's exception analysis is too imprecise.

In the following, I formally describe explicit information flow as a data-flow analysis framework instance.

Given a program dependence graph $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$, I decompose E into $E = DD \cup CD$, where DD is the set of all data dependencies and CD is the set of all control dependencies. Then, *the explicit information*

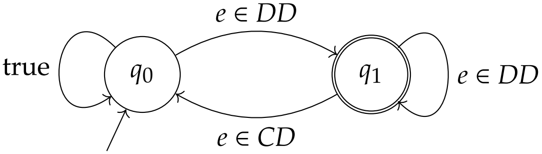


Figure 5.6: A finite automaton for explicit information flow analysis – $P(e)$ is short for $\{e \in E \mid P(e)\}$.

flow (forward) slice of $s \in N$ consists of all nodes t such that at least one valid path from s to t ends in a data dependency:

$$EIF(s) \overset{def}{=} \{t \in N \mid VP(s,t) \cap E^*DD^+ \neq \emptyset\}.$$

As E^*DD^+ is a regular language, EIF is an instance of (5.37):

$$EIF(s) = FS(s, E^*DD^+).$$

Figure 5.6 shows a finite automaton that recognizes E^*DD^+.

5.4.6 Hammer's Approach to IFC

In this section, I want to take a closer look at Hammer's PDG-based approach to IFC [86].

Hammer uses a finite lattice $(L, \leq)$, where the levels $l \in L$ are confidentiality levels; $x \leq y$ means that y is as confidential as or more confidential than x. Partial functions P and R are used to annotate sources and sinks. Sources have a *provided level* $P(n)$ whereas sinks have *required level* $R(n)$. For now, we assume that $dom(P) \cap dom(R) = \emptyset$.

P and R are expanded to all nodes by defining

$$P'(n) = \begin{cases} P(n) & \text{if } n \in dom(P) \\ \bot & \text{otherwise} \end{cases} \tag{5.48}$$

$$R'(n) = \begin{cases} R(n) & \text{if } n \in dom(R) \\ \top & \text{otherwise} \end{cases}. \tag{5.49}$$

Intuitively, information leaving n has confidentiality level at least $P(n)$ and information entering n has at most confidentiality level $R(n)$. The goal is to propagate the confidentiality levels along the paths of G, or, respectively, whether such a propagation is possible without contradiction.
The following monotone constraint system (compare [86, (4.7) on p. 102]) describes a function $S : N \to L$ that *propagates* confidentiality levels from $dom(P)$. Solutions of Constraint System 5.1 can be considered to conservatively describe the information flows of the given program with respect to the confidentiality specification given by P.

Constraint System 5.1.

$$\frac{x \in dom(P) \qquad y \to_G x}{S(x) \geq S(y) \sqcup P(x)} \qquad\qquad \frac{x \notin dom(P) \qquad y \to_G x}{S(x) \geq S(y)}$$

To be able to assess whether the given program is secure with respect to the complete security specification (P, R), Hammer introduces the notion of *maintaining confidentiality*. For this, he states a further set of constraints ([86, (4.8) on p. 102]):

$$\forall x \in dom(R).\ S(x) \leq R(x). \tag{5.50}$$

For G to maintain confidentiality, Hammer requires that Constraint System 5.1 and (5.50) are simultaneously satisfied ([86, Definition 4.1]). Note that the joint constraint system consisting of Constraint System 5.1 and Equation 5.50 is *not* monotone: Constraint System 5.1 and (5.50) use $\leq$ in different directions.
However, whether G maintains confidentiality can be checked by looking at the least solution $\underline{S}$ of the monotone constraint system Constraint System 5.1.

Lemma 5.47. *The following two statements are equivalent:*

1. *G maintains confidentiality.*
2. *The least solution $\underline{S}$ of Constraint System 5.1 satisfies* (5.50) .

Proof. Since $\underline{S}$ is a solution of Constraint System 5.1, it is clear that G maintains confidentiality if $\underline{S}$ also satisfies (5.50).
For the converse direction, assume that G maintains confidentiality. Then there is a function $S : N \rightarrow L$ that satisfies both (5.1) and (5.50). Now, consider the least solution $\underline{S}$ of (5.1). We have to show that $\underline{S}$ satisfies (5.50). So let $x \in dom(R)$. Then $S(x) \leq R(x)$. Moreover, since S is a solution of (5.1) and $\underline{S}$ is the least solution of (5.1), we have $\underline{S}(x) \leq S(x)$. Together, it follows that $\underline{S}(x) \leq R(x)$, as desired. □

Hammer does not state this directly, but suggests that he aims for the least solution of Constraint System 5.1 by stating that "Equation (4.11) is satisfied in the most precise way, and hence the risk that equation (4.8) is violated is minimized, if the inequality for S turns into equality"[30] [86, p. 103] and later referring to the solution as least fixed-point (e.g. [86, Theorem 4.2]).
The simple approach showed so far is conservative but not precise: In fact, Constraint System 5.1 propagates provided levels along arbitrary paths and not only along valid paths.
Hammer describes a slicing-based check that aims to solve this precision problem. A PDG G is said to *ensure non-interference* with respect to R and P, if

$$\forall n \in N. \bigsqcup_{m \in BS(n)} P'(m) \leq R'(n)$$

This amounts to computing

$$S(n) = \bigsqcup_{m \in BS(n)} P'(m) \tag{5.51}$$

and checking that $\forall n.\ S(n) \leq R'(n)$.
As Hammer also notices, (5.51) can be computed using data-flow analysis. In my notation, an appropriate data-flow analysis instance is given by the following ingredients.

[30] Equation (4.11) is a simplified version of Equation (4.7) in [86] or Constraint System 5.1, respectively.

1. L is the security lattice
2. $F = \{\lambda x.\, x \sqcup l \mid l \in L\}$
3. $\rho : e \mapsto f_e$ is defined by

$$f_e(x) \overset{def}{=} x \sqcup P'(src(e)) \sqcup P'(tgt(e)) \tag{5.52}$$

Then

$$f_\pi(x) = x \sqcup \bigsqcup_{n \in nodes(\pi)} P'(n) \tag{5.53}$$

and

$$MOVP(s,t)(x) = x \sqcup \bigsqcup_{\pi \in VP(s,t)} \bigsqcup_{n \in nodes(\pi)} P'(n). \tag{5.54}$$

Hammer provides an algorithm [86, Algorithm 7] to compute (5.51). Specifically, this algorithm generates an appropriate subset of Constraint System 5.1. (5.51) then turns out to be a solution of that system and can be checked against an appropriate set of R constraints that is also generated by the algorithm [86, Theorem 4.2].
Now I show how (5.51) can be extracted from (5.54).
For this, I note that

$$BS(m) = \{m\} \cup \bigcup_{n \in N} \bigcup_{\pi \in VP(n,m)} nodes(\pi). \tag{5.55}$$

This enables me to re-write (5.51) as follows.

$$\begin{aligned}
S(n) &= \bigsqcup_{m \in BS(n)} P'(m) && \{\ (5.51)\ \} \\
&= P(n) \sqcup \bigsqcup_{l \in N} \bigsqcup_{\pi \in VP(l,n)} \bigsqcup_{m \in nodes(\pi)} P'(m) && \{\ (5.55)\ \}
\end{aligned}$$

$$
\begin{aligned}
&= \bigsqcup_{l \in N} \bigsqcup_{\pi \in VP(l,n)} \left(P(n) \sqcup \bigsqcup_{m \in nodes(\pi)} P'(m) \right) && \{ \text{ re-writing } \} \\
&= \bigsqcup_{l \in N} \bigsqcup_{\pi \in VP(l,n)} f_\pi(P'(n)) && \{ \text{ (5.53) } \} \\
&= \left(\bigsqcup_{l \in N} \bigsqcup_{\pi \in VP(l,n)} f_\pi \right) (P'(n)) && \{ \text{ re-writing } \} \\
&= \left(\bigsqcup_{l \in N} MOVP(l,n) \right) (P'(n)). && \{ \text{ Def. of MO}VP \text{ } \}
\end{aligned}
$$

Note that (5.51) does *not* define a solution to Constraint System 5.1, as Constraint System 5.1 is too simplistic, hence demands too much. Hammer proposes an algorithm based on two-phase slicing that generates a subsystem of Constraint System 5.1 for which (5.51) indeed is a solution and which still adequately describes the property of maintaining confidentiality.

5.4.6.1 IFC With Declassification

Hammer also supports a form of *where-declassification*, i.e. the specification of points in the program where confidential information is transformed into benign information which is allowed to be made available to lower observers. For this, he introduces a set $D \subseteq N$ of *declassification nodes*. If information enters a declassification node $d \in D$ with a level of at most r, then d downgrades it (e.g. by sanitizing or removing classified information) to level $p \leq r$.

A declassification node has both a required and a provided level (cf. [86, (4.18)]):

$$
(5.56) \qquad x \in D \implies (x \in dom(P) \cap dom(R)) \wedge R(x) \geq P(x)
$$

Hammer then adapts Constraint System 5.1 and (5.50) to also support declassification nodes (cf. [86, (4.19)]). This leads to the following constraint system.

Constraint System 5.2.

$$\frac{x \in D \qquad y \to_G x}{S(x) \geq P(x)} \qquad\qquad \frac{x \notin D \qquad y \to_G x}{S(x) \geq S(y) \sqcup P'(x)}$$

The check (5.50) is adapted accordingly (cf. [86, (4.20)]):

$$\forall x \in dom(R) \setminus D.\, S(x) \leq R(x) \wedge \forall x \in D. \bigsqcup_{y \to_G x} S(y) \leq R(x). \tag{5.57}$$

The approach is now just like the in the case without declassification: The given program is considered secure if and only if the least solution $\underline{S}$ of Constraint System 5.2 satisfies Equation 5.57. Hammer's Algorithm 7 is also able to handle declassification nodes and generates appropriate subsets of Constraint System 5.2 and (5.57) that can be used to compute and check a solution (cf. [86, Theorem 4.6]).

In the following, I show how (5.57) can be expressed using a data-flow analysis instance.

The complete lattices L and F stay the same as in the non-declassification case, only the transfer functions need to be adapted. Note that Hammer does not give a solution representation for the declassification case like (5.51), so that I have to extract the transfer function from Constraint System 5.2. The definition of f_e particularly accounts for the case that both $src(e)$ and $tgt(e)$ may be declassification nodes.

$$f_e \overset{def}{=} \begin{cases} \lambda x.\, x \sqcup P'(src(e)) \sqcup P'(tgt(e)) & \text{if } src(e) \notin D \wedge tgt(e) \notin D \\ \lambda x.\, P(tgt(e)) & \text{if } tgt(e) \in D \\ \lambda x.\, P(src(e)) & \text{if } src(e) \in D \wedge tgt(e) \notin D \end{cases} \tag{5.58}$$

Next, I want to consider f_π more concretely: In the case that π contains no declassification nodes, it is easy to see that

$$f_\pi = \lambda x.\, x \sqcup \bigsqcup_{n \in nodes(\pi)} P'(n). \tag{5.59}$$

Hence, for $D = \emptyset$, (5.59) is compatible with the declassification-less instance defined in (5.52).

To describe f_π if π contains declassification nodes, I need some additional notations. First of all, I call an edge e such that $src(e) \in D$ or $tgt(e) \in D$ a *declassification edge*. For $\pi \in Paths_G(s,t)$ such that $nodes(\pi) \cap D \neq \emptyset$, I define $last_D(\pi) \in range(\pi)$ as the greatest i such that π^i is a declassification edge and $q(\pi) \overset{def}{=} \pi^{\geq last_D(\pi)}$, the suffix of π that starts with the last declassification edge occurring in π, and $p(\pi) \overset{def}{=} \pi^{<last_D(\pi)}$ as the corresponding prefix.

With the help of $q(\pi)$, I can show that f_π discards all provided levels up to the last declassification node in $q(\pi^0)$.

Theorem 5.48. *If $nodes(\pi) \cap D \neq \emptyset$, f_π can be characterized as follows:*

$$\begin{array}{c} src(q(\pi)^0) \in D \\ \wedge tgt(q(\pi)^0) \notin D \end{array} \implies f_\pi = \lambda x. \bigsqcup_{n \in nodes(\pi)} P'(n) \tag{5.60}$$

$$tgt(q(\pi)^0) \in D \implies f_\pi = \lambda x. \bigsqcup_{e \in edges(\pi)} P'(tgt(e)) \tag{5.61}$$

Proof. First, we observe that it is enough to show (5.60) and (5.61) for $f_{q(\pi)}$:

$$\begin{array}{c} src(q(\pi)^0 <) \in D \\ \wedge tgt(q(\pi)^0) \notin D \end{array} \implies f_{q(\pi)} = \lambda x. \bigsqcup_{n \in nodes(q(\pi))} P'(n) \tag{5.62}$$

$$tgt(q(\pi)^0) \in D \implies f_{q(\pi)} = \lambda x. \bigsqcup_{e \in edges(q(\pi))} P'(tgt(e)) \tag{5.63}$$

The reason is that the right-hand sides of (5.62) and (5.63) are constant functions, i.e. if (5.62) and (5.63) hold, then we have

$$\forall x, y \in L.\ f_{q(\pi)}(x) = f_{q(\pi)}(y) \tag{5.64}$$

Now assume that (5.62) and (5.63) are true and let $x \in L$. Then we have

$$\begin{aligned} f_\pi(x) &= f_{p(\pi) \cdot q(\pi)}(x) && \{ \text{ definition of } p(\pi), q(\pi) \ \} \\ &= f_{q(\pi)}(f_{p(\pi)}(x)) && \{ \text{ properties of } f_\star \ \} \end{aligned}$$

$$= f_{q(\pi)}(x). \qquad \{\, (5.64) \,\}$$

It remains to show (5.62) and (5.63). For this, we consider $q(\pi)$ more closely: It is not empty and consists of exactly one declassification edge at the beginning, i.e. $q(\pi)^0$ is a declassification edge and $q(\pi)^{>0}$ contains no declassification edges and hence no declassification nodes. Hence, for every $x \in L$, we have

$$\begin{aligned}
f_{q(\pi)}(x) &= f_{q(\pi)^0 \cdot q(\pi)^{>0}}(x) && \{\text{ definition }\}\\
&= f_{q(\pi)^{>0}}(f_{q(\pi)^0}(x)) && \{\text{ properties of } f_\star \}\\
&= f_{q(\pi)^0}(x) \sqcup \bigsqcup_{n \in nodes(q(\pi)^{>0})} P'(n) && \{\, (5.59) \,\}
\end{aligned}$$

Now, (5.62) and (5.63) follow from the different cases in the definition (5.58) of $f_{q(\pi)^0}$. Because $q(\pi)^0$ is a declassification edge, we do not need to consider first case in (5.58). For the remaining two cases, we argue as follows:

- If $src(q(\pi)^0) \in D \wedge tgt(q(\pi)^0) \notin D$, then we have

$$\begin{aligned}
&f_{q(\pi)^0}(x) \sqcup \bigsqcup_{n \in nodes(q(\pi)^{>0})} P'(n)\\
&= P'(src(q(\pi)^0)) \sqcup \bigsqcup_{n \in nodes(q(\pi)^{>0})} P'(n) && \{\, (5.58) \,\}\\
&= \bigsqcup_{n \in nodes(q(\pi))} P'(n). && \{\text{ definition of } nodes \}
\end{aligned}$$

- If $tgt(q(\pi)) \in D$, then we have

$$\begin{aligned}
&f_{q(\pi)^0}(x) \sqcup \bigsqcup_{n \in nodes(q(\pi)^{>0})} P'(n)\\
&= P'(tgt(q(\pi)^0)) \sqcup \bigsqcup_{n \in nodes(q(\pi)^{>0})} P'(n) && \{\text{ definition }\}\\
&= \bigsqcup_{e \in edges(q(\pi))} P'(tgt(e)). && \{\, (\star) \,\}
\end{aligned}$$

To conclude the proof, we need to justify the last step ($\star$). Its validity can easily be seen in the case that $q(\pi)^{>0} = \epsilon$. For $q(\pi)^{>0} \neq \epsilon$, we note

$$nodes(q(\pi)^{>0}) = \{tgt(e) \mid e \in edges(q(\pi))\}.$$

Moreover, since $q(\pi)$ is a path, we have

$$tgt(q(\pi)^0) = src(\left(q(\pi)^{>0}\right)^0) \in nodes(q(\pi)^{>0}).$$

Together, this implies

$$nodes(q(\pi)^{>0}) = tgt(q(\pi)^0) \cup \{tgt(e) \mid e \in edges(q(\pi))\}.$$

□

5.4.7 Least Distances

The problem of computing least distances can also be expressed as the instance of a data-flow framework. Computing least distances can be considered a flexible tool to help in PDG-based IFC analysis. By computing least distances, useful information about the valid paths of a given interprocedural graph can be generated. Such information could be used to classify the set of valid paths between two nodes, e.g. with respect to the kinds of dependencies. For example, a least distances analysis could be used to compute the minimum number of control dependencies between two nodes s and t. If this number is 0, then s and t are connected via a chain of data dependencies and the higher the number, the more control dependencies are required to transmit information between s and t.

In the following, I describe the data-flow framework for least distances in the simplest case that every edge is given a weight of 1.

Consider the set $\mathbb{N} \cup \{\infty\}$, partially ordered by $\sqsubseteq_\infty$, which extends the natural ordering $\geq$ on $\mathbb{N}$ such that ∞ is the least element with respect to $\sqsubseteq_\infty$. Note that, in comparison to $\leq$, $\sqsubseteq_\infty$ is "upside down".

With respect to $\sqsubseteq_\infty$, $\mathbb{N} \cup \{\infty\}$ is a complete lattice: Since every non-empty subset $A \subseteq \mathbb{N}$ has a least element with respect to $\leq$, the least upper bound has the following characterization:

$$\bigsqcup A = \begin{cases} \infty & \text{if } A = \emptyset \\ \min A & \text{if } A \neq \emptyset \land \infty \notin A \\ \min (A - \{\infty\}) & \text{if } \infty \in A \end{cases}$$

The space F of transfer functions can be chosen as

$$F := \{\lambda x.\, x + d \mid d \in \mathbb{N}_\infty\}$$

As can easily be seen, all $f \in F$ are monotone, and F is closed under arbitrary joins and function composition.
In the easiest setting, we give each edge the distance 1:

$$\rho(e)(x) = x + 1.$$

Now let $G = (N, E_{intra}, E_{call}, E_{ret}, P, \Phi)$ be an interprocedural graph. For a path $\pi \in Paths(G)$, f_π is then the function that adds the length of π to its argument. Hence, if $VP(s,t) \neq \emptyset$, $MOVP(s,t)$ adds the minimal length of a path between s and t to its argument. Generally, $MOVP(s,t) \in F_\boxtimes$ can be characterized as follows[31]:

$$MOVP(s,t)(x) = \begin{cases} x + \min\{|\pi| \mid \pi \in VP(s,t)\} & \text{if } VP(s,t) \neq \emptyset \\ \boxtimes & \text{otherwise} \end{cases}$$

Note that although there are large similarities, the least-distances-along-valid-paths problem cannot be cast as an instance of the traditional shortest-paths problem on directed graphs for which there are established approaches [56]. Knuth [108] considered a version of the least-distances-problem that can be considered similar to our setting. Traditional approaches compute the length of a shortest *arbitrary* path in the given graph, while we are interested in the length of a shortest *valid* path. A

[31] As I pointed out in section 5.3, I always adjoin a the set F of transfer functions with an additional element $\boxtimes$. For this particular instance, this is technically not necessary, because $\lambda x.\infty$ has all the properties that I assume about $\boxtimes$.

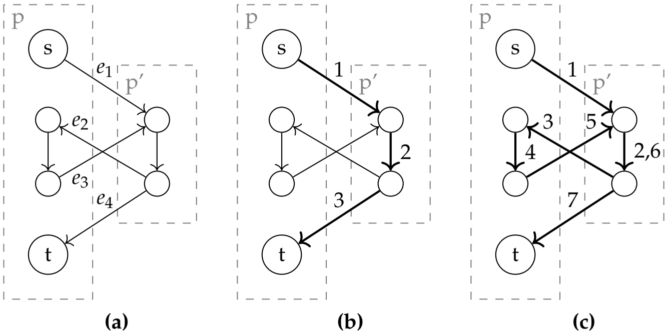


Figure 5.7: A small example with $\Phi = \{(e_1, e_2), (e_3, e_4)\}$ which shows that the shortest valid path may differ from the shortest arbitrary path. (b) shows the shortest arbitrary path, which has length 3 and is invalid, (c) shows the shortest valid path which has length 7.

simple example of this restriction is shown in Figure 5.7. In this example, the shortest arbitrary path between s and t does not respect the correspondence relation ϕ and therefore is invalid. The shortest valid path is significantly longer because both of its sections that enter p' have to leave it through the corresponding return edge.

It is also worth noting that distance calculations on PDGs have been considered before, e.g. by Krinke [110]. Consider a PDG with summary edges. If every edge, including summary edges, is annotated with a weight, a two-phase approach can be used to compute least weights along paths *including summary edges*. In his description, Krinke [110] does not specify the weight that he assigns to summary edges, but I suspect that he uses a weight of 1. Hence, he consequently under-estimates the length of same-level paths and, in the terminology of the data-flow framework instance discussed in this section, computes an *over-approximation*[32] of *MOVP*. Hence, Krinke's approach is correct and more precise than computing least distances w.r.t. arbitrary paths, but it is imprecise w.r.t. *MOVP*.

The generalized algorithms that I present in chapter 7 can be used to (a) annotate each summary edge between to nodes n_0 and n_1 with the *length of*

[32] Remember that the partial order is upside down.

a shortest same-level path between n_0 and n_1 and then to (b) *precisely* compute the least distances along valid paths.

People asking questions, lost in confusion.
Well, I tell them there's no problem,
only solutions.
John Lennon

6 Two Approaches to Abstract Data-Flow Analysis on Interprocedural Graphs

In this chapter, I describe versions of the *functional approach* and the *call-string approach* to interprocedural data-flow analysis for my generalized data-flow frameworks. For classical data-flow frameworks, I already described these two approaches in chapter 3. The two approaches use different ideas to tackle a fundamental problem that arises in interprocedural data-flow analysis, namely that arbitrary paths in interprocedural graphs do not necessarily reflect valid calling and returning behavior of actual program executions. In chapter 3, I already described this problem: An arbitrary path may contain returns that do not return to the call site from which the call started.

The idea of the functional approach is to first solve a helper problem whose solution describes the data-flows along same-level paths. In a second step, this helper solution is then used to describe the data-flows along realizable paths.

In contrast, the call-string approach simulates the call stack usage of paths in order to rule out paths with invalid calling and returning behavior.

Sharir and Pnueli [154] showed that both approaches lead to correct approximations of MO*DESC* in interprocedural control-flow graphs and, for distributive frameworks, can compute MO*DESC* exactly. However, the unrestricted call-string approach uses stacks of unlimited height and therefore does not lead to effectively solvable constraint systems. This is why one usually uses k-bounded stacks, whose height is not greater than k items, and which lead to a correct approximation.

In this chapter, I generalize these results. I describe versions of the functional approach and the call string approach that are based on *interprocedural graphs*, the general graph model that I introduced in chapter 5. Interprocedural graphs cover both interprocedural control-flow graphs and program dependence graphs. Furthermore, I show that both the functional approach and the unrestricted call-string approach enjoy the same properties as in the classical case. Moreover, for the call-string approach, I introduce *stack abstractions*, a general technique that makes it possible to obtain correct approximations to the call-string approach. I demonstrate the applicability of stack abstractions by using them to show that my version of the call-string approach is correct for k-bounded stacks. The following sections are structured as follows. In section 6.1, I make some preparations and fixtures. After that, section 6.2 considers the generalized functional approach. Lastly, I describe my version of the call-string approach in section 6.3.

6.1 Preliminaries

In section 6.2 and section 6.3, I will specify several constraint systems and examine the properties of their solutions. This section makes several necessary preparations. Subsection 6.1.1 specifies the expressions that appear in the constraint systems of this chapter and makes clear their semantics. Subsequently, subsection 6.1.1 introduces *correctness* and *precision* as quality criteria of solutions to constraint systems with respect to a given MO$\mathcal{P}$ function.

For this chapter, I fix a data-flow analysis framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ in the sense of section 5.3.

6.1.1 Syntax and Semantics

First, I have to make clear my assumptions about the expressions and their interpretations from which the constraint systems in this and the next chapter are formed. The set of variables X will always become clear from context and is left unspecified for the moment. Moreover, I assume that the set $\underline{\mathcal{F}}$ of function symbols contains

- a unary symbol $\underline{f}$ for every $f \in F_{\boxtimes}$ and

- a binary symbol $\underline{\circ}$.

In order to provide the semantics functional $[\![\cdot]\!] : \mathit{Expr}(\underline{\mathcal{F}}, X) \to F_{\boxtimes}$, I need to specify the interpretation function α that assigns every function symbol a function on with the appropriate number of inputs from $F_{\boxtimes}$. My obvious choice for α is

$$\alpha(\underline{f}) = f \text{ for all } f \in F_{\boxtimes}$$

$$\alpha(\underline{\circ})(f,g) = \begin{cases} f \circ g & \text{if } f, g \in F \\ \boxtimes & \text{if } f = \boxtimes \lor g = \boxtimes \end{cases}$$

6.1.2 Correctness and Precision of Solutions

The constraint systems that I will present in the following sections are supposed to approximate MO$\mathcal{P}$ for respective path sets $\mathcal{P}$. Definition 6.1 provides the main criteria for assessing the quality of a given function relative to MO$\mathcal{P}$.

Definition 6.1. *Let $\mathcal{P} \subseteq \mathit{Paths}(G)$ be a set of paths, $E \subseteq N \times N$ a set of node pairs and $A : N \times N \to F_{\boxtimes}$ be a function. Then A is called*

1. $(\mathcal{P}, E)$-domain-correct *if it yields a defined value for every $(s,t) \in E$ that is connected by a path in $\mathcal{P}$:*

$$\forall s,t \in N.\ (s,t) \in E \land \mathit{Paths}_G(s,t) \cap \mathcal{P} \neq \emptyset \implies A(s,t) \neq \boxtimes.$$

2. $(\mathcal{P}, E)$-domain-precise *if it does not yield a defined value for $(s,t) \in E$ that is not connected by a path in $\mathcal{P}$:*

$$\forall s,t \in N.\ (s,t) \in E \land A(s,t) \neq \boxtimes \implies \mathit{Paths}_G(s,t) \cap \mathcal{P} \neq \emptyset.$$

3. $(\mathcal{P}, E)$-correct *if it over-approximates* MO$\mathcal{P}$ *on E:*

$$\forall s,t \in N.\ (s,t) \in E \implies A(s,t) \geq \bigsqcup_{\pi \in \mathit{Paths}_G(s,t) \cap \mathcal{P}} f_\pi$$

4. $(\mathcal{P}, E)$-precise *if it does not exceed* MO$\mathcal{P}$ *on E:*

$$\forall s,t \in N.\ (s,t) \in E \implies A(s,t) \leq \bigsqcup_{\pi \in \mathit{Paths}_G(s,t) \cap \mathcal{P}} f_\pi.$$

*I call A just $\mathcal{P}$-*correct/precise*, if it has the corresponding property with respect to $(\mathcal{P}, N \times N)$.*

Definition 6.1 is more general than needed in chapter 6: Specifically, Definition 6.1 also takes solutions into account that are $\neq \boxtimes$ only for a part of $N \times N$. I will need this in chapter 7, which is concerned with computing partial solutions. As my solutions not only give analysis values but also communicate whether there is a value or not, an additional item of comparison is the domain. Hence, Definition 6.1 also incorporates notions that enable to compare the domains of solutions to the domains of objective functions.

It is easy to see that $\mathcal{P}$-correctness of a given function A can be verified by simply showing that every f_π is incorporated in A. This will be my main tool to prove correctness, so I note it here.

Remark 6.2. *A function $A : N \times N \to F_\boxtimes$ is $\mathcal{P}$-correct if and only if*

$$\forall s, t \in N.\ \forall \pi \in Paths_G(s,t).\ (s,t) \in E \land \pi \in \mathcal{P} \implies f_\pi \leq A(s,t).$$

In order to show precision, I will use the argument that is formalized by Remark 6.3.

Remark 6.3. *Let C be a set of constraints over variables $N \times N$ and $\mathcal{P} \subseteq Paths(G)$. Then $lfp(F_C)$ is $\mathcal{P}$-precise if* MO$\mathcal{P}$ *is a solution of C.*

6.2 The Functional Approach

This section describes the functional approach to solve a given interprocedural data-flow framework instance.

The pattern of the functional approach is analogous to the successive construction of the same-level, the ascending, the descending and finally the valid paths that we saw in chapter 5.

Each step considers a set $\mathcal{P}$ of paths (where $\mathcal{P} = SL, ASC, DESC, VP$ in this order) and uses the results from earlier steps to specify a monotone constraint system $C_\mathcal{P}$, whose solutions

$$X_\mathcal{P} : N \times N \to F_\boxtimes$$

are $\mathcal{P}$-correct, i.e. are over-approximations of MO$\mathcal{P}$. For simplicity, I say that both $C_\mathcal{P}$ and its solutions *correctly describe the data transfer along paths from $\mathcal{P}$.*

6.2.1 Constraint Systems

The constraint systems themselves are constructed in a fashion that is very similar to the inductive definitions of the corresponding path sets. First, I introduce a constraint system to describe the data transfer along same-level paths.

Constraint System 6.1.

$$X_{SL} : N \times N \to F_{\boxtimes}$$

is a same-level-solution *if it satisfies all constraints from the following system:*

$$(\textsc{sl-sol-(i)})\ \frac{}{X_{SL}(s,s) \geq id}$$

$$(\textsc{sl-sol-(ii)})\ \frac{t' \overset{e}{\to} t \qquad e \in E_{intra}}{X_{SL}(s,t) \geq f_e \circ X_{SL}(s,t')}$$

$$(\textsc{sl-sol-(iii)})\ \frac{n \overset{e_{call}}{\to} n_0 \qquad n_1 \overset{e_{ret}}{\to} t \qquad (e_{call}, e_{ret}) \in \Phi}{X_{SL}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}} \circ X_{SL}(s,n)}$$

For the sake of presentation, my notation of Constraint System 6.1 and later constraint systems is somewhat sloppy. For one, I use the actual functions from F and function composition where I actually mean their corresponding function symbols. Also note that the $X_{SL}(\_,\_)$ actually denote corresponding variables. Thirdly, I want to point out that the clauses of Constraint System 6.1 – and most other constraint systems shown in this thesis – are to be understood as rules that specify when to include a constraint to the system. This is why in the following, I will refer to these clauses as *rules*.
All in all, sl-sol-(iii) reads:

> For all $n \overset{e_{call}}{\to} n_0$ and $n_1 \overset{e_{ret}}{\to} t$ with $(e_{call}, e_{ret}) \in \Phi$, the system contains the constraint $X_{SL}(s,t) \geq \underline{f_{e_{ret}}\circ}X_{SL}(s,n)\underline{\circ f_{e_{call}}\circ}X_{SL}(s,n)$.

In the following, I want explain the intuition behind Constraint System 6.1. It is supposed to specify a function X_{SL} that correctly describes the data transfer along same-level paths. More concretely, $X_{SL}(s,t)$ shall correctly

approximate MO$SL(s,t)$, i.e. incorporate all path functions f_π along same-level paths π from s to t. Hence, it is natural to construct Constraint System 6.1 in correspondence with the inductive definition of SL.
To illustrate this, I consider sl-sol-(iii) more closely. Let $(e_{call}, e_{ret}) \in \Phi$ with $n \overset{e_{call}}{\to} n_0$ and $n_1 \overset{e_{ret}}{\to} t$. In order to describe the data transfer along same-level paths from s to t, we have to ensure that

$$(6.1) \qquad X_{SL}(s,t) \geq f_{e_{ret}} \circ f_{\pi''} \circ f_{e_{call}} \circ f_{\pi'}$$

for all $\pi' \in SL(s,n)$ and all $\pi'' \in SL(n_0,n_1)$.
Suppose that $X_{SL}(n_0,n_1)$ describes the data transfer along same-level paths from n_0 to n_1 and that $X_{SL}(s,n)$ describes the data transfer along same-level paths from s to n. Then (6.1) can be satisfied for all $\pi' \in SL(s,n)$ and all $\pi'' \in SL(n_0,n_1)$ if

$$(6.2) \qquad X_{SL}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}} \circ X_{SL}(s,n).$$

Rule sl-sol-(iii) specifies that Constraint System 6.1 contains a constraint of the form (6.2) for all $(e_{call}, e_{ret}) \in \Phi$ with $n \overset{e_{call}}{\to} n_0$ and $n_1 \overset{e_{call}}{\to} t$. The three rules together cover all possibilities to construct a same-level path.
Next I use solutions X_{SL} of Constraint System 6.1 to describe the data-flow along ascending paths. The construction principle behind these two constraint systems is the same as for Constraint System 6.1 and their intuition can be explained similarly.

Constraint System 6.2. *Let $X_{SL} : N \times N \to F_\boxtimes$ be a function. Then*

$$X_{ASC} : N \times N \to F_\boxtimes$$

is an ascending-path-solution (relative to X_{SL}) if it satisfies all constraints from the following system:

$$(\textsc{asc-sol-(i)})\ \frac{}{X_{ASC}(s,s) \geq id}$$

$$(\textsc{asc-sol-(ii)})\ \frac{t' \xrightarrow{e} t \qquad e \in E_{intra} \cup E_{ret}}{X_{ASC}(s,t) \geq f_e \circ X_{ASC}(s,t')}$$

$$(\textsc{asc-sol-(iii)})\ \frac{n \overset{e_{call}}{\to} n_0 \quad n_1 \overset{e_{ret}}{\to} t \quad (e_{call}, e_{ret}) \in \Phi \quad X_{SL}(n_0,n_1) \neq \boxtimes}{X_{ASC}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}} \circ X_{ASC}(s,n)}$$

At this point, I want to highlight yet another subtlety in Constraint System 6.2 that applies to all constraint systems in this thesis that make use of some helper functions. Unlike in Constraint System 6.1, $X_{SL}(n_0, n_1)$ in the lower part of asc-sol-(iii) is *not* a variable but a constant. In particular, if $X_{SL}(n_0, n_1) = f \in F$, I actually mean $\underline{f}$ when I write $X_{SL}(n_0, n_1)$ in a constraint. Furthermore, it is also worth mentioning that since the upper part of asc-sol-(iii) is *not* part of the actual constraint but rather specifies when to include the constraint, $X_{SL}(n_0, n_1)$ in the upper part is used as value.

In particular, no constraint is contained in the same situation but $X_{SL}(n_0, n_1) = \boxtimes$. For the solutions of Constraint System 6.2, it is not important whether $X_{SL}(n_0, n_1) \neq \boxtimes$ is demanded or not for a constraint to be present. However, in chapter 7 I will make statements about variables on the left-hand sides of constraints and I need this assumption in order for these statements to make sense.

The data transfer along descending paths is described by Constraint System 6.3, which is completely analogous to Constraint System 6.2.

Constraint System 6.3. *Let $X_{SL} : N \times N \to F_{\boxtimes}$ be a function. Then*

$$X_{DESC} : N \times N \to F_{\boxtimes}$$

is a descending-path-solution (relative to X_{SL}) if it satisfies all constraints from the following system:

$$(\text{desc-sol-(i)}) \; \frac{}{X_{DESC}(s, s) \geq id}$$

$$(\text{desc-sol-(ii)}) \; \frac{t' \xrightarrow{e} t \qquad e \in E_{intra} \cup E_{call}}{X_{DESC}(s, t) \geq f_e \circ X_{DESC}(s, t')}$$

$$(\text{desc-sol-(iii)}) \; \frac{n \xrightarrow{e_{call}} n_0 \quad n_1 \xrightarrow{e_{ret}} t \quad (e_{call}, e_{ret}) \in \Phi \quad X_{SL}(n_0, n_1) \neq \boxtimes}{X_{DESC}(s, t) \geq f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}} \circ X_{DESC}(s, n)}$$

Finally, like valid paths are constructed from the ascending and descending paths, the data transfer along valid paths are described using solutions of Constraint System 6.2 and Constraint System 6.3. This directly leads to Constraint System 6.4.

Constraint System 6.4. *Let*

$$X_{ASC} : N \times N \to F_{\boxtimes}$$

and

$$X_{DESC} : N \times N \to F_{\boxtimes}$$

be two functions. Then

$$X_{VP} : N \times N \to F_{\boxtimes}$$

is a valid-path-solution (relative to X_{ASC} and X_{DESC}) if it satisfies all constraints from the following system:

$$(\textsc{valid-sol})\ \frac{X_{ASC}(s,n) \neq \boxtimes \qquad X_{DESC}(n,t) \neq \boxtimes}{X_{VP}(s,t) \geq X_{DESC}(n,t) \circ X_{ASC}(s,n)}$$

6.2.2 Correctness and Precision

In the following, I show important properties of the solutions of the constraint systems I just introduced. First, I show that the constraint systems indeed meet their purpose, i.e. that their solutions correctly describe the data transfer along the corresponding paths. Given the construction principle behind the constraint system that I described at the beginning of this section, this should be no surprise.
Before I start my actual proofs, I state two elementary properties about F that I will need later at various places.

Remark 6.4. *Function composition is monotone on F, both in the left and in the right argument:*

$$\forall f_1, g_1, f_2, g_2 \in F. f_1 \leq f_2 \wedge g_1 \leq g_2 \implies f_1 \circ g_1 \leq f_2 \circ g_2 \tag{6.3}$$

Proof. This is an easy calculation. □

Remark 6.5. *The function*

$$\bigsqcup : 2^F \to F \tag{6.4}$$

is monotone in the following sense:

$$\forall A, B \in 2^F.\ A \subseteq B \implies \bigsqcup A \leq \bigsqcup B. \tag{6.5}$$

I proceed with showing that solutions of Constraint System 6.1 are *SL*-correct.

Theorem 6.6. *Every same-level solution X_{SL} is SL-correct.*

Proof. We show

$$f_\pi \leq X_{SL}(s, t)$$

by induction on the definition of $\pi \in SL(s, t)$.

1. For $\pi = \epsilon$, we have $f_\pi = id \leq X_{SL}(s, s)$ by definition and constraint SL-SOL-(I).

2. Let $\pi = \pi' \cdot e$ with $\pi' \in SL(s, t')$ and $t' \xrightarrow{e} t$. By induction hypothesis we know

$$(\mathrm{IH}_{intra}) \qquad f_{\pi'} \leq X_{SL}(s, t')$$

Hence

$$\begin{aligned} f_\pi &= f_e \circ f_{\pi'} && \{ \text{ by definition } \} \\ &\leq f_e \circ X_{SL}(s, t') && \{ \text{ by } (\mathrm{IH}_{intra}), (6.3) \} \\ &\leq X_{SL}(s, t) && \{ \text{ by constraint SL-SOL-(II) } \} \end{aligned}$$

3. Let $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$ with $\pi' \in SL(s, n)$, $n \xrightarrow{e_{call}} n_0$, $\pi'' \in SL(n_0, n_1)$, $n_1 \xrightarrow{e_{ret}} t$ and $(e_{call}, e_{ret}) \in \Phi$. By induction hypothesis we know

$$(\mathrm{IH}_{sl}) \qquad f_{\pi'} \leq X_{SL}(s, t') \wedge f_{\pi''} \leq X_{SL}(n_0, n_1)$$

Hence

$$\begin{aligned} f_\pi &= f_{e_{ret}} \circ f_{\pi''} \circ f_{e_{call}} \circ f_{\pi'} && \{ \text{ by definition } \} \\ &\leq f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}} \circ X_{SL}(s, n) && \{ (\mathrm{IH}_{sl}), (6.3) \} \\ &\leq X_{SL}(s, t) && \{ \text{ by SL-SOL-(III) } \} \end{aligned}$$

□

Next, I consider the ascending paths. Theorem 6.7 states that solutions of Constraint System 6.2 are *ASC*-correct, provided that Constraint System 6.2 is defined with respect to an *SL*-correct function X_{SL}. According to Theorem 6.6, X_{SL} *may* be obtained as a solution of Constraint System 6.1, but could also be given in any other way – as long as it is *SL*-correct.

Theorem 6.7. *If X_{SL} is SL-correct and X_{ASC} is an ascending-path solution relative to X_{SL}, then X_{ASC} is ASC-correct:*

$$X_{ASC} \geq \mathrm{MO}ASC$$

Proof. We show

$$\forall \pi \in ASC(s,t).\ f_\pi \leq X_{ASC}(s,t)$$

by induction on $\pi \in ASC$. The cases ASC-SEQ$_{empty}$ and ASC-SEQ$_{asc}$ are very similar to the corresponding cases in the proof of Theorem 6.6, so we only consider ASC-SEQ$_{sl}$.

Let $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$ where $\pi' \in ASC(s,n), n_0, \pi'' \in SL(n_0,n_1), n_1 \overset{e_{ret}}{\to} t$ and $(e_{call}, e_{ret}) \in \Phi$. By assumption, we know

$$(\mathrm{Ass}_{sl}) \qquad f_{\pi''} \leq X_{SL}(n_0,n_1),$$

and, by induction hypothesis,

$$(\mathrm{IH}_{sl}) \qquad f_{\pi'} \leq X_{ASC}(s,n).$$

Furthermore, we know that $f_{\pi''} \neq \boxtimes$ by (5.14) and (5.15). By (Ass$_{sl}$), this means that also $X_{SL}(n_0,n_1) \neq \boxtimes$. Hence, using (6.3),

$$\begin{aligned} f_\pi &= f_{e_{ret}} \circ f_{\pi''} \circ f_{e_{call}} \circ f_{\pi'} && \{\text{ definition }\} \\ &\leq f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}} \circ X_{ASC}(s,n) && \{\ (\mathrm{Ass}_{sl}), (\mathrm{IH}_{sl})\ \} \\ &\leq X_{ASC}(s,t). && \{\text{ by ASC-SOL-(III) }\} \end{aligned}$$

□

Similar to Theorem 6.7, Theorem 6.8 states that solutions of Constraint System 6.3 are *DESC*-correct, provided that Constraint System 6.3 is defined with respect to an *SL*-correct function X_{SL}. The proof is omitted as it is completely analogous to the proof of Theorem 6.7.

Theorem 6.8. *If X_{SL} is SL-correct and X_{DESC} is a descending-path solution relative to X_{SL}, then X_{DESC} is DESC-correct.*

Finally, Theorem 6.9 states that any solution of Constraint System 6.4 with respect to an ASC-correct function X_{ASC} and a $DESC$-correct function X_{DESC} is VP-correct.

Theorem 6.9. *Let X_{ASC} be a ASC-correct, X_{DESC} be DESC-correct and X_{VP} be a valid-path solution relative to X_{ASC} and X_{DESC}. Then X_{VP} is VP-correct.*

Proof. We show

$$\forall \pi \in VP(s,t).\ f_\pi \leq X_{VP}.$$

by induction on $\pi \in VP(s,t)$.
So let $n \in N$, $\pi_1 \in ASC(s,n)$ and $\pi_2 \in DESC(n,t)$. By assumption, we have

$$f_{\pi_1} \leq X_{ASC}(s,n) \wedge f_{\pi_2} \leq X_{DESC}(n,t)$$

Furthermore, we know that $f_{\pi_1} \in F$ and $f_{\pi_2} \in F$ using (5.14) and (5.15). Hence we can conclude

$$\begin{aligned} f_\pi &= f_{\pi_2} \circ f_{\pi_1} && \{ \text{ definition } \} \\ &\leq X_{DESC}(n,t) \circ X_{ASC}(s,n) && \{ \text{ by assumption, monotonicity of } \circ \} \\ &\leq X_{VP} && \{ \text{ by constraint VALID-SOL. } \} \end{aligned}$$

□

Next, I want to consider the question under which circumstances the constraint systems not only correctly describe the data transfer along their corresponding path set $\mathcal{P}$ but are also $\mathcal{P}$-precise. For classical interprocedural data-flow analysis, this is the case if they are universally distributive [154]. Such a result can also be given for my generalized setup.
Using the argument from Remark 6.3, it suffices to show that MO$\mathcal{P}$ is a solution of the corresponding constraint system $C_{\mathcal{P}}$ in order to guarantee precision of $lfp(F_{C_{\mathcal{P}}})$. This is stated and proven in Theorem 6.10.

Theorem 6.10. *Assume that $\mathcal{F}_{\boxtimes}$ is universally distributive. Then the following statements are true:*

1. *MOSL is a same-level solution.*

2. If X_{SL} is SL-precise, then $\mathrm{MO}ASC$ *is an ascending-path solution with respect to X_{SL}.*

3. If X_{SL} is SL-precise, then $\mathrm{MO}DESC$ *is a descending-path solution with respect to X_{SL}.*

4. If X_{ASC} is ASC-precise and X_{DESC} is DESC-precise, then $\mathrm{MO}VP$ *is a valid-path solution with respect to X_{ASC} and X_{DESC}.*

Proof. 1. We show that $\mathrm{MO}SL$ satisfies SL-SOL-(I), SL-SOL-(II) and SL-SOL-(III).

SL-SOL-(I) : For every $s \in N$, $\mathrm{MO}SL(s,s) \geq id$ is satisfied because $\epsilon \in SL(s,s)$.

SL-SOL-(II) : Let $s \in N, e \in E_{intra}, t' \in N$ such that $t' \xrightarrow{e} t$. Then we have

$$SL(s,t) \supseteq \{\pi' \cdot e \mid \pi' \in SL(s,t')\}$$

This means that

$$\begin{aligned}
\mathrm{MO}SL(s,t) &\geq \bigsqcup\{f_{\pi'\cdot e} \mid \pi' \in SL(s,t')\} && \{\,(6.5)\,\}\\
&= \bigsqcup\{f_e \circ f_{\pi'} \mid \pi' \in SL(s,t')\} && \{\text{ def. of } f_\pi \}\\
&= f_e \circ \bigsqcup\{f_{\pi'} \mid \pi' \in SL(s,t')\} && \{\, f_e \text{ is u.d. }\}\\
&= f_e \circ \mathrm{MO}SL(s,t') && \{\text{ def. of } \mathrm{MO}SL \}
\end{aligned}$$

SL-SOL-(III) : Let $s \in N$, $e_c \in E_{call}$, $e_r \in E_{ret}$, and $n, n_0, n_1 \in N$ such that $(e_c, e_r) \in \Phi$, $n \xrightarrow{e_c} n_0$ and $n_1 \xrightarrow{e_r} t$. Then

$$SL(s,t) \supseteq \{\pi_1 \cdot e_c \cdot \pi_2 \cdot e_r \mid \pi_1 \in SL(s,n), \pi_2 \in SL(n_0,n_1)\}.$$

This means that

$$\begin{aligned}
&\mathrm{MO}SL(s,t)\\
\geq &\bigsqcup\{f_{e_r} \circ f_{\pi_2} \circ f_{e_c} \circ f_{\pi_1} \mid \pi_1 \in SL(s,n), \pi_2 \in SL(n_0,n_1)\}\\
&\{\,(6.5), \text{definition}\,\}\\
= &f_{e_r} \circ \bigsqcup\{f_{\pi_2} \mid \pi_2 \in SL(n_0,n_1)\} \circ f_{e_c} \circ \bigsqcup\{f_{\pi_1} \mid \pi_1 \in SL(s,n)\}\\
&\{\,\mathcal{F} \text{ is u.d. }\}\\
= &f_{e_r} \circ \mathrm{MO}SL(n_0,n_1) \circ f_{e_c} \circ \mathrm{MO}SL(s,n)\\
&\{\text{ definition of } \mathrm{MO}SL \}
\end{aligned}$$

2. We show that MO$ASC$ satisfies the constraints from Constraint System 6.2 with respect to X_{SL}.

ASC-SOL-(I) For every $s \in N$, MO$ASC(s,s) \geq id$ is satisfied because $\epsilon \in ASC(s,s)$.

ASC-SOL-(II) Let $s, t, t' \in N$, $e \in E_{intra} \cup E_{ret}$ with $t' \xrightarrow{e} t$. Then we have $\pi' \cdot e \in ASC(s,t)$ for every $\pi' \in ASC(s,t')$. Hence,

$$\{\pi' \cdot e \mid |\pi' \in ASC(s,t')\} \subseteq ASC(s,t),$$

which translates to

$$\bigsqcup\{f_{\pi' \cdot e} \mid \pi' \in ASC(s,t')\} \leq \mathrm{MO}ASC(s,t).$$

Thus we can conclude

$$\begin{aligned}
& \mathrm{MO}ASC(s,t) \\
\geq & \bigsqcup\{f_{\pi' \cdot e} \mid \pi' \in ASC(s,t')\} && \{\text{ see above }\} \\
= & \bigsqcup\{f_e \circ f_{\pi'} \mid \pi' \in ASC(s,t')\} && \{\text{ definition }\} \\
= & f_e \circ \bigsqcup\{f_{\pi'} \mid \pi' \in ASC(s,t')\} && \{\ F_{\boxtimes} \text{ is u.d. }\} \\
= & f_e \circ \mathrm{MO}ASC(s,t'). && \{\text{ definition }\}
\end{aligned}$$

ASC-SOL-(III) Let $s, t \in N$, $e_c \in E_{call}$, $e_r \in E_{ret}$ with $n \xrightarrow{e_c} n_0$, $n_1 \xrightarrow{e_r} t$, $(e_c, e_r) \in \Phi$ and $X_{SL}(n_0, n_1) \neq \boxtimes$. Then, according to the closure properties of ASC, we have $\pi' \cdot e_c \cdot \pi'' \cdot e_r \in ASC(s,t)$ for all $\pi' \in ASC(s,n)$, $\pi'' \in SL(n_0, n_1)$. Hence we have

$$\{\pi' \cdot e_c \cdot \pi'' \cdot e_r \mid \pi' \in ASC(s,n), \pi'' \in SL(n_0,n_1)\} \subseteq ASC(s,t),$$

which, due to (6.3), implies that

$$\begin{aligned}
& \bigsqcup\{f_{\pi' \cdot e_c \cdot \pi'' \cdot e_r} \mid \pi' \in ASC(s,n), \pi'' \in SL(n_0,n_1)\} \\
\leq & \mathrm{MO}ASC(s,t).
\end{aligned} \tag{6.6}$$

We conclude as follows:

$$\mathrm{MO}ASC(s,t)$$

$$
\begin{aligned}
&\geq \bigsqcup\{f_{\pi_1 \cdot e_c \cdot \pi_2 \cdot e_r} \mid \pi_1 \in ASC(s,t'), \pi_2 \in SL(n_0,n_1)\} \\
&\quad \{\text{ see above }\} \\
&= \bigsqcup\{f_{e_r} \circ f_{\pi_2} \circ f_{e_c} \circ f_{\pi_1} | \pi_1 \in ASC(s,t'), \pi_2 \in SL(n_0,n_1)\} \\
&\quad \{\text{ definition }\} \\
&= f_{e_r} \circ \bigsqcup\{f_{\pi_2} \mid \pi_2 \in SL(n_0,n_1)\} \circ f_{e_c} \circ \bigsqcup\{f_{\pi_1} | \pi_1 \in ASC(s,t')\} \\
&\quad \{\ F_{\boxtimes} \text{ is u.d. }\} \\
&= f_{e_r} \circ \mathrm{MO}SL(n_0,n_1) \circ f_{e_c} \circ \mathrm{MO}ASC(s,n) \\
&\quad \{\text{ definition }\} \\
&\geq f_{e_r} \circ X_{SL}(n_0,n_1) \circ f_{e_c} \circ \mathrm{MO}ASC(s,n). \\
&\quad \{\ X_{SL} \text{ is } SL\text{-precise, (6.5) }\}
\end{aligned}
$$

3. This is very similar to the proof of 2.

4. For every $n \in N$, we have

$$
\begin{aligned}
&\mathrm{MO}VP(s,t) \\
&= \bigsqcup\{f_{\pi_2} \circ f_{\pi_1} \mid \pi_1 \in ASC(s,n), \pi_2 \in DESC(n,t)\} \\
&\quad \{\text{ definition }\} \\
&= \bigsqcup\{f_{\pi_1} \mid \pi_1 \in DESC(n,t)\} \circ \bigsqcup\{f_{\pi_1} \mid \pi_1 \in ASC(s,n)\} \\
&\quad \{\ F_{\boxtimes} \text{ is u.v. }\} \\
&\geq X_{DESC}(n,t) \circ X_{ASC}(s,n). \\
&\quad \{\text{ assumptions about } X_{ASC} \text{ and } X_{DESC}, \text{ (6.3) }\}
\end{aligned}
$$

□

6.3 The Call-String Approach

The basic idea of the call-string approach, which I already explained in chapter 3, is to additionally simulate a *call stack*: Each time a procedure is called, the call site is pushed to the stack, and each time the procedure returns, the call site it is supposed to return to is popped off the stack. Sharir and Pnueli showed for classical interprocedural data-flow analysis that the call string approach with unbounded stacks yields the same solution as the functional approach [154]. However, unbounded stacks also lead to infinite constraint systems and lattices that do not satisfy (ACC). This is why in practice, one usually uses approximate approaches such as bounded stack heights.

In this section, I will give a general and formal presentation of the call-string approach under my more general assumptions. The general results of this section will be that (a) I can achieve the same result as Sharir and Pnueli under my more general assumptions and (b) that I provide a general framework from which one can derive correctness results for approximative call-string approaches that include but are potentially not limited to bounded stack heights.

This section is divided into four subsections. In subsection 6.3.1, I introduce the abstract concept of *stack spaces*, which provide enough structure to simulate the calling behavior of interprocedural programs but are general enough to at least cover both unbounded and bounded stack heights. Given a stack space $\mathcal{S}$, I then introduce the $\mathcal{S}$-*acceptable* paths, i.e. the set of paths of G that exhibit valid stack usage with respect to $\mathcal{S}$. Moreover, I define a monotone constraint system and prove a correctness and a precision result for solutions of this system. These results are similar to the corresponding results in section 6.2, but do not compare solutions to *MOVP* but to the merge over all $\mathcal{S}$-acceptable paths. In order to get results with respect to *MOVP*, I need to relate the $\mathcal{S}$-acceptable paths to *VP* for specific stack spaces and this is the subject of the other subsections. As a first step, I consider the space of unbounded stacks in subsection 6.3.2. For this stack space, I can indeed show that the acceptable paths coincide with *VP*. After that, in subsection 6.3.3, I describe a way that allows to transfer correctness results between stack spaces. For this, I introduce the concept of *stack abstractions*, which allows to relate two stack spaces using the more general and well-known concept of *galois connection*. The main result of subsection 6.3.3 is that if there is a stack abstraction between two stack spaces $\mathcal{S}$ and $\mathcal{S}^{\#}$, then all $\mathcal{S}$-acceptable paths are also $\mathcal{S}^{\#}$-acceptable. Finally, subsection 6.3.4 gives an example of an application of this main result and proves that the call-string approach using the space of k-bounded stacks yields a correct approximation of *MOVP*.

6.3.1 Stack Spaces

Definition 6.11 introduces *stack spaces*. A stack space is an abstract structure that provides the interface needed by an interprocedural program to implement procedure calls.

Definition 6.11. *A* stack space *over the alphabet A is a partially ordered set $(S, \leq)$ (whose elements are called* stacks*) with a distinguished element ϵ (the empty stack) and functions*

$$\begin{aligned} push &: A \times S \to S \\ pop &: (S \setminus \{\epsilon\}) \to S \\ top &: (S \setminus \{\epsilon\}) \to A \end{aligned}$$

such that the following conditions hold:

(EpsMax) *ϵ is the greatest element w.r.t. $\leq$: $\forall \sigma \in S.\ \sigma \leq \epsilon$*

(PopMon) *pop is monotone w.r.t. $\leq$.*

(PushMon) *For every $a \in A, push(a, \cdot)$ is monotone w.r.t. $\leq$.*

(PushNE) *Pushing preserves non-emptiness:*
$\forall a \in A.\ \forall \sigma \in S.\ \sigma \neq \epsilon \implies push(a, \sigma) \neq \epsilon$

(TopPush) *Pushed elements can be found at the top:*
$\forall a \in A.\ \forall \sigma \in S.\ push(a, \sigma) \neq \epsilon \implies top(push(a, \sigma)) = a$

(NEPushPop) *Non-empty stacks can be expressed with push and pop:*
$\forall \sigma \in S.\ \sigma \neq \epsilon \implies (\exists a \in A.\ \sigma = push(a, pop(\sigma)))$

(TopVsLe) *top is compatible with $\leq$:*
$\forall \sigma' \in S.\ \sigma \neq \epsilon \wedge \sigma' \neq \epsilon \wedge \sigma \leq \sigma' \implies top(\sigma) = top(\sigma')$

(PopPushLe) $\forall a \in A.\ \forall \sigma \in S.\ push(a, \sigma) \neq \epsilon \implies \sigma \leq pop(push(a, \sigma))$

The axioms (TopPush), (NEPushPop) and (PushNE) should not be surprising as they essentially describe how a stack works.
Additionally, stack spaces provide a partial order. The intuition behind $\leq$ is that a stack space not necessarily provides full and precise information about actual call stacks but may only give partial and approximate information. We can use $\leq$ to compare stacks with respect to the information they provide: Given two stacks σ, σ', $\sigma \leq \sigma'$ is supposed to represent that

σ' does not provide more information than σ. The axioms (PushMon), (PopMon) and (TopVsLe) specify that the main operations *push*, *pop* and *top* are compatible with $\leq$.

A special case is the empty stack ϵ that provides no information. This is formalized by (EpsMax). Finally, I want to give some explanation for (PopPushLe). It considers a stack σ and an element a and compares σ to the result of first pushing a to σ and then popping one element off the top. Normally, one would expect that the result of these two subsequent operations should be just σ. This is indeed the case for unbounded stacks but in general, I cannot demand this since stack spaces are supposed to also cover *bounded stacks*. For bounded stacks, which I will formally show in Example 6.13, there is a $k > 0$ such that no stack is higher than k. Consequently, such a stack can be *full*. If we push an element to a full stack and want to preserve the top, we simply remove the element at the bottom. Popping off the top element then does not yield the full stack but only a prefix of it. This is why (PopPushLe) only demands that $\sigma \leq pop(push(a, \sigma))$.

The following two examples show to important stack spaces, namely the spaces of unbounded and k-bounded stacks, respectively.

Example 6.12. *Consider $\mathcal{S}_\infty = (E^\star_{call}, \leq_\infty, \epsilon_\infty, push_\infty, pop_\infty, top_\infty)$ with*

$$\sigma \leq_\infty \sigma' \overset{def}{\iff} \sigma' \in Prefixes(\sigma)$$

$$\epsilon_\infty \overset{def}{=} \epsilon \quad \textit{(empty sequence)}$$

$$push_\infty(e, \sigma) \overset{def}{=} e \cdot \sigma$$

$$pop_\infty(e \cdot \sigma) \overset{def}{=} \sigma = (e \cdot \sigma)^{\geq 1}$$

$$top_\infty(e \cdot \sigma) \overset{def}{=} e = (e \cdot \sigma)^0$$

$\leq_\infty$ is a partial order, as can easily be verified, and $\mathcal{S}_\infty$ is a stack space over E_{call}:

(EpsMax) *ϵ is a prefix of every symbol sequence.*

(PopMon) *If $e \cdot \sigma'$ is a prefix $e' \cdot \sigma$, then $e = e'$ and σ' is a prefix of σ.*

(PushMon) *If σ' is a prefix of σ, then $e \cdot \sigma$ is a prefix of $e \cdot \sigma'$ for every $e \in E_{call}$.*

(PushNE) *It is clear that $\sigma \neq \epsilon$ implies $a \cdot \sigma \neq \epsilon$.*

(TopPush) *It is also easy to see that* $top_\infty(push_\infty(a, \sigma)) = top_\infty(a \cdot \sigma) = a$.

(NEPushPop) *Every non-empty sequence* σ *can be expressed as* $e \cdot \sigma^{\geq 1}$ *for some* $e \in E_{call}$.

(TopVsLe) *If* σ, σ' *are both not empty and* σ *is a prefix of* σ', *then obviously* $\sigma^0 = \sigma'^0$.

(PopPushLe) *For every* $\sigma \in E_{call}^\infty$, *we have* $pop_\infty(push(a, \sigma)) = \sigma \geq_\infty \sigma$.

Example 6.13. *For* $k \in \mathbb{N}$, *consider* $\mathcal{S}_k = (E_{call}^{\leq k}, \leq_k, \epsilon_k, push_k, pop_k, top_k)$ *with*

$$\begin{aligned} \sigma \leq_k \sigma' &\overset{def}{\iff} \sigma' \in Prefixes(\sigma) \\ \epsilon_k &\overset{def}{=} \epsilon \quad \text{(empty sequence)} \\ push_k(e, \sigma) &\overset{def}{=} (e \cdot \sigma)^{\leq k} \\ pop_k(e \cdot \sigma) &\overset{def}{=} \sigma \\ top_k(e \cdot \sigma) &\overset{def}{=} e \end{aligned}$$

Then $\mathcal{S}_k$ *is a stack space over* E_{call}: *This is trivial for* $k = 0$, *so that we only consider the case that* $k > 0$. $\leq_k$ *is a partial order with greatest element* ϵ_k, *so that EpsMax is satisfied.*

(PopMon) *See Example 6.12.*

(PushMon) $push_k(e, \cdot)$ *is monotone with respect to* $\leq_k$ *because it is the composition of* $\lambda\sigma.\ e \cdot \sigma$ *and* $\lambda\sigma.\ \sigma^{\leq k}$, *which are both monotone with respect to* $\leq_k$.

(PushNE) *Let* $\sigma \in E_{call}^{\leq k}$ *with* $\sigma \neq \epsilon$. *Then* $push_k(e, \sigma) = (e \cdot \sigma)^{\leq k} = e \cdot (\sigma^{\leq k-1}) \neq \epsilon$.

(TopPush) *The argument from above shows in particular that*

$$top(push(e, \sigma)) = e$$

for every $\sigma \in E_{call}^{\leq k}$, $e \in E_{call}$.

(NEPushPop) *Let $\sigma \in E_{call}^{\leq k}$ be non-empty. Then $\sigma = e \cdot \sigma^{\geq 1}$ for some $e \in E_{call}$. Since $\sigma \in E_{call}^{\leq k}$, we have $\sigma = \sigma^{\leq k} = (e \cdot \sigma^{\geq 1})^{\leq k} = (e \cdot pop_k(\sigma))^{\leq k} = push_k(e, pop_k(\sigma))$.*

(TopVsLe) *See Example 6.12.*

(PopPushLe) *Let $\sigma \in E_{call}^{\leq k}$. We make a case distinction of whether $|\sigma| < k$ or $|\sigma| = k$:*

- *If $|\sigma| < k$, then $|e \cdot \sigma| \leq k$, which means that $e \cdot \sigma = (e \cdot \sigma)^{\leq k} = push_k(e, \sigma)$. Hence, $pop_k(push_k(e, \sigma)) = pop_k((e \cdot \sigma)^{\leq k}) = pop_k(e \cdot \sigma) = \sigma$.*
- *If $|\sigma| = k$, then $push_k(e, \sigma) = e \cdot \sigma^{<k}$, so that $pop_k(push_k(e, \sigma)) = \sigma^{<k}$. Since $\sigma^{<k}$ is a prefix of σ, this shows that $\sigma \leq_k pop_k(push_k(a, \sigma))$.*

In the following, I consider a fixed stack space

$$\mathcal{S} = (E_{call}, S, \leq, \epsilon, push, pop, top)$$

over E_{call}.

Next, I want to introduce the paths in G that exhibit an acceptable stack behavior with respect to $\mathcal{S}$ and the correspondence relation Φ. For this, I use a function *cs* that takes as input an edge sequence $\pi \in E^{\star}$. This function traverses π and simulates π's usage of calls and returns with respect to $\mathcal{S}$ and Φ. Each time a call edge is encountered, *cs* applies the *push* function of $\mathcal{S}$ to the current stack. Each time a return edge is encountered, *cs* checks whether the top of the current stack corresponds to the return edge with respect to Φ. If so, the top is popped off using the *pop* function of $\mathcal{S}$ – if not, *cs* concludes that π is unacceptable or *invalid*.

In summary, *cs* returns either a call stack that is left behind by π, e.g. π just consists of a series of call edges, then the result of *cs* is the stack that consists of these unfinished calls) or a special symbol ▼ that signals unacceptable stack behavior. Occasionally, I will also call ▼ the *invalid stack*.

I extend S and $\leq$ by ▼ such that ▼ is the least element with respect to $\leq$ and

(6.7) $$push : E_{call} \times S_{\blacktriangledown} \to S_{\blacktriangledown}$$

(6.8) $$pop : (S \setminus \{\epsilon\}) \cup \{\blacktriangledown\} \to S_{\blacktriangledown}$$

such that

(6.9) $$\forall \sigma \in S.\ \forall e \in E_{call}.\ push(e, \sigma) = \blacktriangledown \iff \sigma = \blacktriangledown$$

$$\forall \sigma \in S.\, pop(\sigma) = \blacktriangledown \iff \sigma = \blacktriangledown \tag{6.10}$$

That is, if *push* and *pop* are applied to valid stacks, the result is guaranteed to be valid, and if they are applied to the invalid stack, the result stays invalid.

With the help of ▼, I give the definition of *cs* in Definition 6.14.

Definition 6.14.

$$cs : Paths(G) \to S_{\blacktriangledown}$$

is defined by

$$cs(\epsilon) = \epsilon$$

$$cs(\pi \cdot e) = \begin{cases} cs(\pi) & \text{if } e \in E_{intra} \\ push(e, cs(\pi)) & \text{if } e \in E_{call} \\ cs(\pi) = \epsilon & \text{if } e \in E_{ret} \text{ and } cs(\pi) = \epsilon \\ pop(cs(\pi)) & \text{if } e \in E_{ret} \wedge cs(\pi) \notin \{\blacktriangledown, \epsilon\} \\ & \wedge\ (top(cs(\pi)), e) \in \Phi \\ \blacktriangledown & \text{otherwise} \end{cases}$$

If a path π can be fully traversed without resulting in the invalid stack, I call π $\mathcal{S}$*-acceptable*[33].

Definition 6.15. *The $\mathcal{S}$-acceptable paths are given by*

$$AP_{\mathcal{S}}(s,t) \overset{def}{=} \{\pi \in Paths_G(s,t) \mid cs(\pi) \neq \blacktriangledown\}.$$

Next, I introduce Constraint System 6.5, which can be used to describe the data-flows along paths from *AP*. Like the previous constraint systems, its idea is to simulate what happens when we extend a path by one edge. In addition to the application of the corresponding edge function, Constraint System 6.5 also considers the effect of the edge on the stack – this is why the variables in Constraint System 6.5 have an additional stack component.

Constraint System 6.5. *Let $\mathcal{S} = (E_{call}, S, \leq, \epsilon, push, pop, top)$ be a stack space over E_{call}. Then $X_{\mathcal{S}} : N \times N \times S \to F_{\boxtimes}$ is a $\mathcal{S}$-solution if it satisfies the following*

[33] A similar definition can be found in the work of Sharir and Pnueli[154, p. 227].

constraints:

$$(\textsc{empty}_{\mathcal{S}})\ \frac{}{X_{\mathcal{S}}(s,s,\epsilon) \geq id} \qquad (\textsc{intra}_{\mathcal{S}})\ \frac{e \in E_{intra} \qquad t' \xrightarrow{e} t}{X_{\mathcal{S}}(s,t,\sigma) \geq f_e \circ X_{\mathcal{S}}(s,t',\sigma)}$$

$$(\textsc{call}_{\mathcal{S}})\ \frac{e_{call} \in E_{call} \qquad t' \xrightarrow{e_{call}} t \qquad \sigma = push(e_{call},\sigma')}{X_{\mathcal{S}}(s,t,\sigma) \geq f_{e_{call}} \circ X_{\mathcal{S}}(s,t',\sigma')}$$

$$(\textsc{ret}^{(1)}_{\mathcal{S}})\ \frac{e_{ret} \in E_{ret} \qquad t' \xrightarrow{e_{ret}} t}{X_{\mathcal{S}}(s,t,\epsilon) \geq f_{e_{ret}} \circ X_{\mathcal{S}}(s,t',\epsilon)}$$

$$(\textsc{ret}^{(2)}_{\mathcal{S}})\ \frac{t' \xrightarrow{e_{ret}} t \;\; (e_{call},e_{ret}) \in \Phi \;\; pop(push(e_{call},\sigma)) = \sigma \;\; push(e_{call},\sigma) \neq \epsilon}{X_{\mathcal{S}}(s,t,\sigma) \geq f_{e_{ret}} \circ X_{\mathcal{S}}(s,t',push(e_{call},\sigma))}$$

Like for previous constraint systems, I want to make some remarks on how to read and interpret Constraint System 6.5.

First of all, the set of variables of Constraint System 6.5 is $N \times N \times S$. Hence, solutions to Constraint System 6.5 live in the complete lattice

$$N \times N \times S \rightarrow F_{\boxtimes}$$

Moreover, I extend the function symbols and their interpretation function to also cover the stack operations. Analogously to previous constraint systems, for each occurrence of such a stack operation in a constraint in the lower part of one of the rules, I actually mean the corresponding function symbol. Regarding the upper part, I again want to remind the reader that the upper part of the rules specifies conditions under which the constraint in the lower part of the rule is contained in the set of constraints that makes up Constraint System 6.5. As an example, consider (6.5): It contains a constraint of the form

$$X_{\mathcal{S}}(s,t,\sigma) \geq f_{e_{call}} \circ X_{\mathcal{S}}(s,t',\sigma')$$

for all $s,t,t' \in N$, $e_{call} \in E$ and $\sigma,\sigma' \in S$ such that $e \in E_{call}$, $t' \xrightarrow{e_{call}} t$ and $\sigma = push(e_{call},\sigma')$.

Theorem 6.16 gives a correctness result for Constraint System 6.5.

Theorem 6.16. *Let $X : N \times N \times S \to F_{\boxtimes}$ be a $\mathcal{S}$-solution. Then for arbitrary $s, t \in N$ we have*

$$\forall \pi \in Paths_G(s,t).\ \pi \in AP_{\mathcal{S}}(s,t) \implies f_\pi \leq X(s,t,cs(\pi)) \tag{6.11}$$

In particular, $\tilde{X}(s,t) = \bigsqcup_{\sigma \in S} X(s,t,\sigma)$ defines a MO*$AP_{\mathcal{S}}$-correct solution.*

Proof. First assume that (6.11) is proven. Then

$$\forall \pi \in AP_{\mathcal{S}}(s,t).\ f_\pi \leq X(s,t,cs(\pi)) \leq \bigsqcup_{\sigma \in S} X(s,t,\sigma) = \tilde{X}(s,t)$$

so that

$$\mathrm{MO}AP_{\mathcal{S}}(s,t) = \bigsqcup_{\pi \in AP_{\mathcal{S}}(s,t)} f_\pi \leq \tilde{X}(s,t)$$

Now we prove (6.11) by induction on the length of paths. Let $n \in \mathbb{N}$. The induction hypothesis is

$$\forall \pi \in Paths_G(s,t).\ |\pi| < n \wedge \pi \in AP_{\mathcal{S}}(s,t) \implies f_\pi \leq X(s,t,cs(\pi)) \tag{IH}$$

We show

$$\forall \pi \in Paths_G(s,t).\ |\pi| = n \wedge \pi \in AP_{\mathcal{S}}(s,t) \implies f_\pi \leq X(s,t,cs(\pi)) \tag{6.12}$$

by case distinction on n.

$n = 0$: clear by constraint EMPTY$_{\mathcal{S}}$.

$n > 0$: Let π be a path from s to t with $|\pi| = n > 0$ and $\pi \in AP_{\mathcal{S}}(s,t)$. Hence, we can write $\pi = \pi' \cdot e$ with $\pi' \in Paths_G(s,t')$ and $t' \xrightarrow{e} t$. Moreover, since $\pi \in AP_{\mathcal{S}}(s,t)$, it must be $\pi' \in AP_{\mathcal{S}}(s,t)$ as well (this follows from the definition of cs). By definition, (IH), and (6.3), we see that

$$f_\pi \leq f_e \circ X(s,t',cs(\pi')).$$

Then we show

$$f_e \circ X(s,t',cs(\pi')) \leq X(s,t,cs(\pi))$$

by case distinction on e:

- $e \in E_{intra}$: Then $cs(\pi) = cs(\pi')$ and hence

$$\begin{aligned} f_e \circ X(s,t',cs(\pi')) &\leq X(s,t,cs(\pi')) && \{ \text{ constraint } \textsc{intra}_{\mathcal{S}} \} \\ &= X(s,t,cs(\pi)) && \{ \text{ since } cs(\pi') = cs(\pi) \} \end{aligned}$$

- $e = e_{call} \in E_{call}$: Then $cs(\pi) = push(e_{call}, cs(\pi'))$ and hence we can conclude

  $$f_{e_{call}} \circ X(s,t',cs(\pi')) \leq X(s,t,cs(\pi)))$$

  by $\textsc{call}_{\mathcal{S}}$.

- $e = e_{ret} \in E_{ret}$: By definition of cs and since $cs(\pi) \neq \blacktriangledown$ by assumption, it must be either $cs(\pi) = cs(\pi') = \epsilon$ or $cs(\pi') \notin \{\blacktriangledown, \epsilon\}$ and $(top(cs(\pi'), e_{ret})) \in \Phi$. We consider each of these cases individually.

  1. If $cs(\pi') = cs(\pi) = \epsilon$, we can make the following reasoning:

  $$\begin{aligned} f_{e_{ret}} \circ X(s,t',cs(\pi')) &= f_{e_{ret}} \circ X(s,t',\epsilon) && \{ cs_\infty(\pi') = \epsilon \} \\ &\leq X(s,t,\epsilon) && \{ \text{ by } \textsc{ret}^{(1)}_{\mathcal{S}} \} \\ &= X(s,t,cs(\pi)) && \{ cs(\pi) = \epsilon \} \end{aligned}$$

  2. If $cs(\pi') \notin \{\blacktriangledown, \epsilon\}$ and $(top(cs(\pi'), e_{ret})) \in \Phi$, then for $e_{call} \overset{def}{=} top(cs(\pi'))$ we have $(e_{call}, e_{ret}) \in \Phi$ and $cs(\pi) = pop(cs(\pi'))$ (by definition of cs), hence

  $$cs(\pi') = push(e_{call}, cs(\pi)) = push(e_{call}, pop(cs(\pi'))) \tag{6.13}$$

  by (NEPushPop) and (TopPush). Hence, we can apply $\textsc{ret}^{(2)}_{\mathcal{S}}$ and conclude:

  $$\begin{aligned} & f_{e_{ret}} \circ X(s,t',cs(\pi')) \\ = {} & f_{e_{ret}} \circ X(s,t',push(e_{call}, cs(\pi))) && \{ (6.13) \} \\ \leq {} & X(s,t,cs(\pi)) && \{ \text{ by } \textsc{ret}^{(2)}_{\mathcal{S}} \} \end{aligned}$$

□

Theorem 6.16 states that MO$AP_{\mathcal{S}}$ can be correctly approximated by computing a solution of Constraint System 6.5 and subsequently joining over all stacks.
In order to give a precision result for Constraint System 6.5, I need a variant of MO$AP_{\mathcal{S}}$ that has an additional stack component. This variant is defined in Definition 6.17. $\mathrm{MO}^{\text{(Stack)}}_{AP_{\mathcal{S}}}(s,t,\sigma)$ only merges over those paths that leave behind stack σ.

Definition 6.17. *The stack-based Merge-Over-All-$\mathcal{S}$-Acceptable-Paths solution*

$$\mathrm{MO}^{(Stack)}_{AP_{\mathcal{S}}} : N \times N \times S \rightarrow_{mon} F_{\boxtimes}$$

is defined by

$$\mathrm{MO}^{(Stack)}_{AP_{\mathcal{S}}}(s,t,\sigma) = \bigsqcup_{\pi \in VP(s,t), cs_{\infty}(\pi)=\sigma} f_{\pi}$$

Now I show that Constraint System 6.5 is also able to precisely describe MO$AP_{\mathcal{S}}$, provided that $\mathcal{F}$ is universally distributive.

Theorem 6.18. *Let $\mathcal{F}$ be a universally distributive framework. Then* $\mathrm{MO}^{(Stack)}_{AP}$ *is an $\mathcal{S}$-solution. In particular, for the least $\mathcal{S}$-solution $\underline{X}$, we have $\underline{X} \leq$* $\mathrm{MO}^{(Stack)}_{AP}$.

Proof. We show that $\mathrm{MO}^{\text{(Stack)}}_{AP}$ satisfies all the constraints of Constraint System 6.5.

EMPTY$_{\mathcal{S}}$ Let $s \in N$. Then $\mathrm{MO}^{\text{(Stack)}}_{AP}(s,s,\epsilon) = id$. So, EMPTY$_{\mathcal{S}}$ is satisfied.

INTRA$_{\mathcal{S}}$ Let $s,t,t' \in N$, $e \in E_{intra}$, $t' \xrightarrow{e} t$ and $\sigma \in S$. Then for every $\pi' \in AP_{\mathcal{S}}(s,t')$ with $cs_{\infty}(\pi') = \sigma$ we have $\pi' \cdot e \in AP_{\mathcal{S}}(s,t)$ and $cs(\pi' \cdot e) = cs(\pi') = \sigma$. This implies

$$\begin{aligned}
& \mathrm{MO}^{\text{(Stack)}}_{AP}(s,t,\sigma) \\
= & \bigsqcup \{f_{\pi} \mid \pi \in AP_{\mathcal{S}}(s,t), cs(\pi) = \sigma\} && \{\text{ definition }\} \\
\geq & \bigsqcup \{f_{\pi' \cdot e} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \sigma\} && \{\text{ see above }\}
\end{aligned}$$

$$
\begin{aligned}
&= \bigsqcup\{f_e \circ f_{\pi'} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \sigma\} && \{\text{ definition }\}\\
&= f_e \circ \bigsqcup\{f_{\pi'} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \sigma\} && \{\ f_e \text{ is u.d. }\}\\
&= f_e \circ \mathrm{MO}^{(\text{Stack})}_{AP}(s,t',\sigma) && \{\text{ definition }\}
\end{aligned}
$$

$\textsc{call}_{\mathcal{S}}$ Let $s,t,t' \in N$, $e_{call} \in E_{call}$, $t' \overset{e_{call}}{\to} t$ and $\sigma \in S$. Then for every $\pi' \in AP(s,t')$ with $cs(\pi') = \sigma$ we have $\pi' \cdot e_{call} \in AP_{\mathcal{S}}(s,t)$ with $cs(\pi' \cdot e_{call}) = push(e_{call}, cs(\pi')) = push(e_{call}, \sigma)$. Hence

$$
\begin{aligned}
&\mathrm{MO}^{(\text{Stack})}_{AP}(s,t,push(e_{call},\sigma))\\
&= \bigsqcup\{f_\pi \mid \pi \in AP_{\mathcal{S}}(s,t), cs_\infty(\pi) = push(e_{call},\sigma)\} && \{\text{ definition }\}\\
&\geq \bigsqcup\{f_{\pi'\cdot e_{call}} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \sigma\} && \{\text{ see above }\}\\
&= \bigsqcup\{f_{e_{call}} \circ f_{\pi'} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \sigma\} && \{\text{ definition }\}\\
&= f_{e_{call}} \circ \bigsqcup\{f_{\pi'} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \sigma\} && \{\ f_{e_{call}} \text{ is u.d. }\}\\
&= f_{e_{call}} \circ \mathrm{MO}^{(\text{Stack})}_{AP}(s,t',\sigma) && \{\text{ definition }\}
\end{aligned}
$$

$\textsc{ret}^{(1)}_{\mathcal{S}}$ Let $s,t,t' \in N$, $e_{ret} \in E_{ret}$, $t' \overset{e_{ret}}{\to} t$, $\sigma \in S$ and $\pi' \in AP_{\mathcal{S}}(s,t')$ with $cs(\pi') = \epsilon$. Then $\pi' \cdot e_{ret} \in AP_{\mathcal{S}}(s,t)$ and $cs(\pi' \cdot e_{ret}) = \epsilon$. Hence, we may conclude

$$
\begin{aligned}
&\mathrm{MO}^{(\text{Stack})}_{AP}(s,t,\epsilon)\\
&= \bigsqcup\{f_\pi \mid \pi \in AP_{\mathcal{S}}(s,t), cs(\pi) = \epsilon\} && \{\text{ definition }\}\\
&\geq \bigsqcup\{f_{\pi'\cdot e_{ret}} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \epsilon\} && \{\text{ see above }\}\\
&= \bigsqcup\{f_{e_{ret}} \circ f_{\pi'} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \epsilon\} && \{\text{ definition }\}\\
&= f_{e_{ret}} \circ \bigsqcup\{f_{\pi'} \mid \pi' \in AP_{\mathcal{S}}(s,t'), cs(\pi') = \epsilon\} && \{\ f_{e_{ret}} \text{ is u.d. }\}\\
&= f_{e_{ret}} \circ \mathrm{MO}^{(\text{Stack})}_{AP}(s,t',\epsilon) && \{\text{ definition }\}
\end{aligned}
$$

$\textsc{ret}^{(2)}_{\mathcal{S}}$ Let $\sigma \in S$ with $push(e_{call},\sigma) \neq \epsilon$ and $pop(push(e_{call},\sigma)) = \sigma$. Furthermore, let $s,t,t' \in N$, $e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$, $t' \overset{e_{ret}}{\to} t$, $(e_{call}, e_{ret}) \in \Phi$, and

$\pi' \in AP_{\mathcal{S}}(s,t')$ with $cs(\pi') = push(e_{call},\sigma)$. Then $\pi' \cdot e_{ret} \in AP_{\mathcal{S}}(s,t)$ and $cs(\pi' \cdot e_{ret}) = pop(cs(\pi')) = pop(push(e_{call},\sigma)) = \sigma$. This justifies that we conclude

$$\begin{aligned}
&\mathrm{MO}^{(\text{Stack})}_{AP}(s,t,\sigma) \\
&= \bigsqcup\{f_\pi \mid \pi \in AP(s,t), cs(\pi) = \sigma\} && \{\text{ definition }\} \\
&\geq \bigsqcup\{f_{\pi'\cdot e_{ret}} \mid \pi' \in AP(s,t'), cs(\pi') = push(e_{call},\sigma)\} && \{\text{ see above }\} \\
&= \bigsqcup\{f_{e_{ret}} \circ f_{\pi'} \mid \pi' \in AP(s,t'), cs(\pi') = push(e_{call},\sigma)\} && \{\text{ definition }\} \\
&= f_{e_{ret}} \circ \bigsqcup\{f_{\pi'} \mid \pi' \in AP(s,t'), cs(\pi') = push(e_{call},\sigma)\} && \{\ f_{e_{ret}} \text{ is u.d. }\} \\
&= f_{e_{ret}} \circ \mathrm{MO}^{(\text{Stack})}_{AP}(s,t',push(e_{call},\sigma)) && \{\text{ definition }\}
\end{aligned}$$

□

6.3.2 The Unbounded Stack Space

Up until now, we only have compared solutions to the abstract set $AP_{\mathcal{S}}$. The subject of this and the following subsection is to examine stack spaces for which $AP_{\mathcal{S}}$ relates to VP in a meaningful way. From the results, I will derive VP-correctness and VP-precision results in subsection 6.3.4

In this subsection, I consider the very important special case of the space $\mathcal{S}_\infty$ of unbounded stacks. My main result for this subsection is that the $\mathcal{S}_\infty$-acceptable paths coincide with the valid paths.

The proof is split up into two parts. In the first part, I show that every valid path is $\mathcal{S}_\infty$-acceptable. After that, I show every $\mathcal{S}_\infty$-acceptable path is valid.

6.3.2.1 Valid Paths Are $\mathcal{S}_\infty$-Acceptable

Theorem 6.21 states that every valid path is $\mathcal{S}_\infty$-acceptable. It makes use of two elementary properties of cs_∞ concerning same-level and ascending paths.

The first property states that appending a same-level path does not change the stack. This is not surprising as same-level paths are balanced and valid, which means that for every call there is a return and that every return corresponds to the innermost call.

Lemma 6.19. *If π' is a same-level path, then*

$$\forall \pi \in Paths(G).cs_\infty(\pi \cdot \pi') = cs_\infty(\pi)$$

Proof. Induction on $\pi' \in SL$ with arbitrary π. □

The second property, which is stated in Lemma 6.20, considers ascending paths. Basically, an ascending path is like a same-level path with an excess of return edges. Hence, if we start with an empty stack and have traversed the same-level prefix, the stack is empty. Then traversing the rest of the path leaves the stack empty. Note that this property also highlights a specific characteristic of my treatment of stacks. Because I not only consider descending paths, I have to handle the case of returning from an empty stack, which is not a valid situation in a normal program. In my definition of cs_∞, I allow returning to any call site if the stack is empty, as long as stack usage is acceptable in the case that the stack is not empty.

Lemma 6.20. *If π is an ascending path, then $cs_\infty(\pi) = \epsilon$.*

Proof. By induction on $\pi \in ASC$.

- The claim is clear for $\pi = \epsilon$.
- Suppose that the claim holds for $\pi' \in ASC(s, t')$ and we have $\pi = \pi' \cdot e$ with $e \in E_{intra} \cup E_{ret}$ and $t' \xrightarrow{e} t$. Then we distinguish two cases:
  - If $e \in E_{intra}$, it follows that

$$\begin{aligned} cs_\infty(\pi' \cdot e) &= cs_\infty(\pi') && \{\text{ by definition }\} \\ &= \epsilon && \{\text{ by induction hypothesis }\} \end{aligned}$$

  - If $e \in E_{ret}$, we know $cs_\infty(\pi') = \epsilon$ by induction hypothesis and can thus follow $cs_\infty(\pi) = \epsilon$ by definition of cs.
- Suppose that the claim holds for $\pi' \in ASC(s, n)$, we have $n \xrightarrow{e_{call}} n_0, \pi'' \in SL(n_0, n_1), n_1 \xrightarrow{e_{ret}} t$ and $(e_{call}, e_{ret}) \in \Phi$. Then we have

$$\begin{aligned} cs_\infty(\pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}) &= cs_\infty(\pi') && \{\text{ by Lemma 6.19 }\} \\ &= \epsilon && \{\text{ by induction hypothesis }\} \end{aligned}$$

□

With help of Lemma 6.19 and Lemma 6.20, I can prove acceptability of valid paths.

Theorem 6.21. *If $\pi \in VP(s,t)$, then $cs_\infty(\pi) \neq \blacktriangledown_\infty$.*

Proof. We use Theorem 5.30. Let $\pi_1 \in ASC(s,t')$ be an ascending path starting with s and ending in some arbitrary $t' \in N$, then it suffices to prove

$$\forall \pi_2 \in DESC.\ \forall t \in N.\ \pi_2 \in DESC(t',t) \implies cs_\infty(\pi_1 \cdot \pi_2) \neq \blacktriangledown_\infty.$$

We prove this statement by induction on $\pi_2 \in DESC$.

- If $\pi_2 = \epsilon$, then $\pi_1 \cdot \pi_2 \in ASC(s,t)$. Hence, $cs_\infty(\pi_1 \cdot \pi_2) = \epsilon$ by Lemma 6.20.
- Otherwise, we are in case DESC-SEQ$_{desc}$ or DESC-SEQ$_{sl}$ and have $\pi_2 = \pi_2' \cdot \pi_2''$ for some $\pi_2' \in DESC$ and either $\pi_2'' \in E_{call}$ or $\pi_2'' \in SL$.

  In either case, we can apply the respective induction hypothesis and get

  $$\text{(IH)} \qquad cs_\infty(\pi_1 \cdot \pi_2') \neq \blacktriangledown_\infty.$$

  We consider these two cases separately.

  - Let $\pi_2'' = e_{call} \in E_{call}$, with $t'' \overset{e_{call}}{\to} t'$. Then , we can conclude

    $$\begin{aligned} & cs_\infty(\pi_1 \cdot \pi_2 \cdot e_{call}) \\ = {} & push_\infty(e_{call}, cs_\infty(\pi_1 \cdot \pi_2)) && \{ \text{definition} \} \\ \neq {} & \blacktriangledown_\infty && \{ \text{(IH), (6.9)} \} \end{aligned}$$

  - Let $\pi_2 = \pi_2' \cdot \pi_2''$ with $\pi_2' \in DESC(t',t'')$ and $\pi_2'' \in SL(t'',t)$ for some $t'' \in N$. Then, we can conclude

    $$\begin{aligned} cs_\infty(\pi_1 \cdot \pi_2' \cdot \pi_2'') &= cs_\infty(\pi_1 \cdot \pi_2') && \{ \pi_2'' \in SL, \text{Lemma 6.19} \} \\ &\neq \blacktriangledown_\infty. && \{ \text{(IH)} \} \end{aligned}$$

□

6.3.2.2 $\mathcal{S}_\infty$-Acceptable Paths Are Valid

For the property that cs_∞ only accepts valid paths, which I will show and prove in Lemma 6.24, I need two additional properties of cs_∞ that are also of interest on their own.

The first property formally describes the possible shapes of a path π for which $cs_\infty(\pi)$ is not empty: Suppose that $cs_\infty(\pi) = e_{call} \cdot \sigma$. Then intuitively, it can only be the case that (a) some prefix of π left behind stack σ, (b) then e_{call} was encountered and (c) the part after e_{call} is a same-level path that leaves the stack unchanged. Lemma 6.22 states that this is indeed the case.

Lemma 6.22. *If $\pi \in Paths_G(s,t)$ and $cs_\infty(\pi) = e_{call} \cdot \sigma$, then π can be split up into $\pi = \pi' \cdot e_{call} \cdot \pi''$ where $cs_\infty(\pi') = \sigma$ and π'' is a same-level path.*

Proof. We fix $s \in N$ and prove the result for all $\pi \in Paths_G(s,t)$ with arbitrary $t \in N$, $e \in E_{call}$ and $\sigma \in E^\star_{call}$ by induction on the length of paths. Let $n \in \mathbb{N}$ and assume that the claim is true for all π' with $|\pi'| < n$. Let $\pi_0 \in Paths_G(s,t)$, $e_{call} \in E_{call}$, $\sigma \in E^\star_{call}$ such that $|\pi_0| = n$ and $cs_\infty(\pi_0) = e_{call} \cdot \sigma$.

First we observe that $n > 0$, since $cs_\infty(\epsilon) = \epsilon$. So $\pi_0 = \pi \cdot e$ for some $e \in E$. We make a case distinction on e:

$e \in E_{intra}$: Then $cs_\infty(\pi \cdot e) = cs_\infty(\pi)$. Hence, we can apply the induction hypothesis on π and get a decomposition $\pi = \pi' \cdot e_{call} \cdot \pi''$ where $cs_\infty(\pi') = \sigma$ and π'' is a same-level path. Hence $\pi \cdot e = \pi' \cdot e_{call} \cdot (\pi'' \cdot e)$ is a decomposition of $\pi \cdot e$ with the desired properties: $\pi'' \cdot e$ is a same-level path, since $e \in E_{intra}$.

$e \in E_{call}$: Then $cs_\infty(\pi \cdot e) = e \cdot cs_\infty(\pi) = e_{call} \cdot cs_\infty(\pi)$, so $e = e_{call}$ and $\sigma = cs_\infty(\pi) \neq \blacktriangledown_\infty$. Now we make a case distinction on whether $cs_\infty(\pi)$ is empty or not.

- If $cs_\infty(\pi)$ is not empty, then $cs_\infty(\pi) = e' \cdot \sigma'$ for some $e' \in E_{call}$ and $\sigma' \in E^\star_{call}$. We apply the induction hypothesis on π and obtain π' with $cs_\infty(\pi') = \sigma'$ and a same-level path π'' such that

$$\pi = \pi' \cdot e' \cdot \pi''$$

Then

$$\pi \cdot e = \pi' \cdot e' \cdot \pi'' \cdot e = (\pi' \cdot e' \cdot \pi'') \cdot e \cdot \epsilon \tag{6.14}$$

ϵ is a same-level path, so in order to show that 6.14 is a decomposition of $\pi \cdot e$ with the desired properties, we only need to show $cs_\infty(\pi' \cdot e' \cdot \pi'') = cs_\infty(\pi) = e' \cdot \sigma'$. But this can be seen as follows:

$$\begin{aligned} cs_\infty(\pi' \cdot e' \cdot \pi'') &= cs_\infty(\pi' \cdot e') && \{\, \pi'' \in SL, \text{Lemma 6.19} \,\} \\ &= e' \cdot cs_\infty(\pi') && \{\, \text{by definition of } cs_\infty, e' \in E_{call} \,\} \\ &= e' \cdot \sigma' && \{\, cs_\infty(\pi') = \sigma' \,\} \end{aligned}$$

- If $cs_\infty(\pi) = \epsilon$, then $\sigma = \epsilon = cs_\infty(\pi)$ so that $\pi \cdot e = \pi \cdot e \cdot \epsilon$ is a decomposition with the desired properties.

$e \in E_{ret}$: We have $e_{call} \cdot \sigma = cs_\infty(\pi \cdot e) = pop_\infty(cs_\infty(\pi))$, so that by definition $cs_\infty(\pi) = e' \cdot (e_{call} \cdot \sigma)$ for some $e' \in E_{call}$ with $(e', e) \in \Phi$. We apply the induction hypothesis on π ($|\pi| < n$) and get π', π'' such that $cs_\infty(\pi') = e_{call} \cdot \sigma$, π'' is a same-level path and

$$\pi = \pi' \cdot e' \cdot \pi''$$

Next we apply the induction hypothesis on π' ($|\pi'| < n$) and get π'_1, π'_2 such that $cs_\infty(\pi'_1) = \sigma$, π'_2 is a same-level path and

$$\pi' = \pi'_1 \cdot e_{call} \cdot \pi'_2$$

That means $\pi \cdot e$ can be decomposed into

$$\pi \cdot e = \pi'_1 \cdot e_{call} \cdot \pi'_2 \cdot e' \cdot \pi' \cdot e = \pi'_1 \cdot e_{call} \cdot (\pi'_2 \cdot e' \cdot \pi'' \cdot e) \tag{6.15}$$

Since $(e', e) \in \Phi$ and π'' is a same-level path, $e' \cdot \pi'' \cdot e$ is a same-level path. Since π'_2 is also a same-level path, $\pi'_2 \cdot e' \cdot \pi'' \cdot e$ is a same-level path. Furthermore, $cs_\infty(\pi'_1) = \sigma$. In summary, 6.15 is a decomposition of $\pi \cdot e$ with the desired properties.

□

The second property that will be needed in my proof of Lemma 6.24 is the converse of Lemma 6.20: Empty stacks can only be left behind by ascending paths. In particular, even though cs_∞ allows returning to arbitrary call sites in the face of the empty stack, apart from this exception, it still requires that paths exhibit valid stack usage.

Lemma 6.23. *If $\pi \in Paths_G(s,t)$ with $cs_\infty(\pi) = \epsilon_\infty$, then $\pi \in ASC(s,t)$.*

Proof. I prove the result by induction on the length of paths.
The induction hypothesis is

$$\text{(IH)} \qquad \forall \pi \in Paths_G.\ \forall s,t \in N.\ |\pi| < n \land \pi \in Paths_G(s,t) \land cs_\infty(\pi) = \epsilon \implies \pi \in ASC(s,t)$$

Let $\pi \in Paths_G$ with $|\pi| = n$ and $s,t \in N$ such that $\pi \in Paths_G(s,t)$. Furthermore, assume $cs_\infty(\pi) = \epsilon$. We need to show $\pi \in ASC(s,t)$. For this, we distinguish two cases:

$n = 0$: Then π must be empty and is hence ascending, so that the claim holds in this case.

$n > 0$: Then $\pi = \pi' \cdot e$ and $\pi' \in Paths_G(s,t')$ for some $t' \in N, e \in E, t' \xrightarrow{e} t$. Since $cs_\infty(\pi) = \epsilon$, it cannot be the case that $e \in E_{call}$, so we only need to distinguish the cases $e \in E_{intra}$ and $e \in E_{ret}$:

$e \in E_{intra}$: Then $cs_\infty(\pi) = cs_\infty(\pi')$, so that $cs_\infty(\pi') = \epsilon$. Hence we may apply (IH) to π' and conclude $\pi' \in ASC(s,t')$. Thus $\pi = \pi' \cdot e \in ASC(s,t)$ by ASC-ASC.

$e \in E_{ret}$: By definition, $cs_\infty(\pi) = \epsilon$ implies that either $cs_\infty(\pi') = \epsilon$ or that $cs_\infty(\pi') = e_{call}$ for some $e_{call} \in E_{call}$ with $(e_{call}, e) \in \Phi$.

In the former case, we may reason just as for the case "$e \in E_{intra}$": Applying (IH), we obtain $\pi' \in ASC(s,t')$ and appending a return edge to an ascending path results in an ascending path.

In the latter case, we apply Lemma 6.22 and decompose π' into $\pi' = \pi'' \cdot e_{call} \cdot \pi'''$ such that $cs_\infty(\pi'') = \epsilon$ and π''' is a same-level path. Then we apply (IH) to π'' and conclude that π'' is an ascending path. Furthermore, since $(e_{call}, e) \in \Phi$ and π''' is same-level, $\pi = \pi'' \cdot e_{call} \cdot \pi''' \cdot e$ is ascending by ASC-SEQ$_{sl}$. With $\pi \in Paths_G(s,t)$, we get $\pi \in ASC(s,t)$ by definition.

□

Lemma 6.24. *If $\pi \in Paths_G(s,t)$ with $cs_\infty(\pi) \neq \blacktriangledown_\infty$, then π is a valid path.*

Proof. If $cs_\infty(\pi) \neq \blacktriangledown_\infty$, then $cs_\infty(\pi) \in E^\star_{call}$, so we prove the claim by induction on the length of $cs_\infty(\pi)$. So let $n \in \mathbb{N}$.
The induction hypothesis is

$$\text{(IH)} \qquad \forall \pi \in Paths_G.\ |cs_\infty(\pi)| < n \wedge cs_\infty(\pi) \neq \blacktriangledown_\infty \implies \pi \in VP$$

Let $\pi \in Paths_G(s,t)$ with $cs_\infty(\pi) \neq \blacktriangledown_\infty$, $|cs_\infty(\pi)| = n$. We distinguish two cases:

$n = 0$: Then $cs_\infty(\pi) = \epsilon$. By Lemma 6.23, $\pi \in ASC \subseteq VP$.

$n > 0$: Then $cs_\infty(\pi) = e \cdot \sigma$ for some $e \in E_{call}$. By Lemma 6.22, π can be decomposed into $\pi = \pi' \cdot e \cdot \pi''$ where $cs_\infty(\pi') = \sigma$ and π'' is a same-level path. In particular, $cs_\infty(\pi') \neq \blacktriangledown_\infty$ and $|cs_\infty(\pi')| < n$, so we may apply (IH) and get that $\pi' \in VP$. Furthermore, since $e \in E_{call}$ and π'' and is same-level, $e \cdot \pi''$ is descending by Theorem 5.29. Furthermore, $e \cdot \pi''$ is a path because it is a sub-sequence of π and π is a path. Hence, $e \cdot \pi'' \in DESC$ by Theorem 5.37.

Thus, π is the concatenation of a valid sequence and a descending sequence and therefore valid by Theorem 5.30. With $\pi \in Paths_G$, we get $\pi \in VP$ by definition.

□

6.3.2.3 Summary

After I have proven the two subset relations, I formally state the main result of this subsection, namely that $AP_{\mathcal{S}_\infty}$ coincides with VP, for later reference.

Theorem 6.25. *The valid paths are exactly the $\mathcal{S}$-acceptable paths for $\mathcal{S} = \mathcal{S}_\infty$:*

$$AP_{\mathcal{S}_\infty} = VP$$

Proof. "$\subseteq$" is shown by Lemma 6.24 and "$\supseteq$" by Theorem 6.21. □

As a corollary, I conclude that for distributive frameworks, the least solution Constraint System 6.5 w.r.t. $\mathcal{S}_\infty$ coincides with $MOVP$ and hence provides the same result as the functional approach. Thus, I can confirm the corresponding result, that Sharir and Pnueli [154, Theorem 7-4.6] showed for classical data-flow analyses on interprocedural graphs in my more general setting.

6.3.3 Stack Abstractions

In the last subsections, we have seen that Constraint System 6.5 correctly and precisely describes the data-flows along the valid paths in the case that Constraint System 6.5 is based on the stack space $\mathcal{S}_\infty$ of unbounded stacks. In this sense, the unbounded call-string approach *coincides* with the functional approach.

In this and the next subsection, I examine general stack spaces. Specifically, I present an approach for providing correctness results for general stack spaces. Therefore, this subsection introduces the concept of *stack abstractions*. A stack abstraction is a function between stack spaces that preserves enough structure to maintain acceptability of paths. My main result is that if a stack space $\mathcal{S}$ is related to a stack space $\mathcal{S}^\#$ via a stack abstraction, then all $\mathcal{S}$-acceptable paths are $\mathcal{S}^\#$-acceptable. This enables to conclude VP-correctness of $\mathrm{MO}AP_{\mathcal{S}^\#}$ from VP-correctness of $\mathrm{MO}AP_{\mathcal{S}}$.

In subsection 6.3.4, I will apply this result to obtain a VP-correctness result for $\mathrm{MO}AP_{\mathcal{S}_k}$.

Stack abstractions make use of *galois connections*, a concept that is well-known in mathematics and also static program analysis.

A galois connection can be seen as a generalization of inverse functions for partial orders. Definition 6.26 can be found e.g. in [130, Section 4.3] or [50, 7.23, p. 155].

Definition 6.26. *Let $(L, \leq_L)$ and $(M, \leq_M)$ be two partially ordered sets and*

$$\alpha : L \to M$$
$$\gamma : M \to L$$

be two monotone functions. Then (L, α, γ, M) is called a galois connection *if $\alpha \circ \gamma \leq_M id_M$ and $id_L \leq_L \gamma \circ \alpha$.*

Lemma 6.27 gives a simple characterization of galois connections, that I will make use of occasionally in the following.

Lemma 6.27. *(L, M, α, γ) is a galois connection between the partially ordered sets $(L, \leq_L)$ and $(M, \leq_M)$ if and only if $\alpha : L \to M$ and $\gamma : M \to M$ are monotone functions such that*

$$\forall l \in L.\, \forall m \in M.\, \alpha(l) \leq_M m \iff l \leq_L \gamma(m)$$

Proof. See [130, Proposition 4.20]. Note that [130] actually assumes that L and M are complete lattices, which is not necessary for the proof to be valid. □

Next, Definition 6.28 introduces stack abstractions, the main concept of this subsection.

Definition 6.28. *A stack abstraction between two stack spaces*

$$(S, \leq, \epsilon, top, push, pop)$$

and

$$(S^\#, \leq^\#, \epsilon^\#, top^\#, push^\#, pop^\#)$$

is a pair (α, γ) of functions

$$\begin{aligned} \alpha :& S_\blacktriangledown \to S^\#_{\blacktriangledown^\#} \\ \gamma :& S^\#_{\blacktriangledown^\#} \to S_\blacktriangledown \end{aligned}$$

such that

(Gal) $(S_\blacktriangledown, \alpha, \gamma, S^\#_{\blacktriangledown^\#})$ *is a galois connection.*

(Bot) $\gamma(\blacktriangledown^\#) = \blacktriangledown$

(Eps) $\forall \sigma^\# \in S^\#.\, \gamma(\sigma^\#) = \epsilon \implies \sigma^\# = \epsilon^\#$

(PushHom) $\forall \sigma^\# \in S^\#.\, \alpha(push(e, \gamma(\sigma^\#))) \leq^\# push^\#(e, \sigma^\#)$

(PopHom) $\forall \sigma^\# \in S^\# - \{\epsilon^\#\}.\, \alpha(pop(\gamma(\sigma^\#))) \leq^\# pop^\#(\sigma^\#)$

(TopHom) $\forall \sigma^\# \in S^\# - \{\epsilon^\#\}.\, top(\sigma^\#) = top(\gamma(\sigma^\#))$

Before I show an example of Definition 6.28, I want to discuss a bit the intuition behind stack abstraction and their properties.
For this, consider a stack abstraction $(S, \alpha, \gamma, S^\#)$ between two stack spaces $\mathcal{S} = (S, \leq, \epsilon, top, push, pop)$ and $\mathcal{S}^\# = (S, \leq, \epsilon, top, push, pop)$.

A stack abstraction is supposed to "abstract away" from unnecessary details in a stack space while still preserving enough structure to make meaningful statements. For example, it shall enable to conclude from the validness of $cs(\pi)$, that $cs^{\#}(\pi)$ is also valid, i.e. it shall be possible to prove $cs(\pi) \neq \blacktriangledown \implies cs^{\#}(\pi) \neq \blacktriangledown$.

First of all, (Gal) provides a formalization of the intuitive notion of *abstraction*. The function α, which is in the program analysis context also called *abstraction function*, can be thought of "abstracting away the details" from a concrete stack from $\mathcal{S}$, whereas γ, also called *concretization function*, maps an abstract stack to a corresponding concrete stack.

Intuitively, we expect two properties of abstractions and their concretizations. For one, if we abstract a stack and then concretize again, then we get a concrete stack that provides at most as much as the original stack. In particular, this stack should be comparable to the original stack. Secondly, if we concretize an abstract stack and then abstract it, then we get a stack that provides at least as much information as the original abstract stack.

This is exactly what is achieved by the galois connection provided by (Gal). For every $\sigma \in S$, there is a corresponding stack $\alpha(\sigma) \in S^{\#}$, which can be thought of as "at most as detailed" as σ. This is to be understood in the following sense: If we apply γ to $\alpha(\sigma)$, we arrive at a stack $\gamma(\alpha(\sigma))$ with $\sigma \leq \gamma(\alpha(\sigma))$.

Conversely, for every stack $\sigma^{\#} \in S^{\#}$, there is a corresponding stack $\gamma(\sigma^{\#}) \in \mathcal{S}$ in S, which can be though of as "at least as detailed". This is to be understood in the following sense: If we apply α to $\gamma(\sigma^{\#})$, we arrive at a stack $\alpha(\gamma(\sigma^{\#})) \leq \sigma^{\#}$.

Next, I consider the other properties. (Gal) in connection with (Bot) and (Eps) is strong enough to fully preserve ϵ and $\blacktriangledown$, as Remark 6.29 shows.

Remark 6.29. *For every stack abstraction, we have*

$$\alpha(\epsilon) = \epsilon^{\#} \tag{6.16}$$

$$\gamma(\epsilon^{\#}) = \epsilon \tag{6.17}$$

$$\alpha(x) = \blacktriangledown^{\#} \iff x = \blacktriangledown \tag{6.18}$$

Proof. First, we consider (6.16) and (6.17). By (Gal), we have $\epsilon \leq \gamma(\alpha(\epsilon))$. Since ϵ is the greatest element of S with respect to $\leq$, this means that $\gamma(\alpha(\epsilon)) = \epsilon$. Hence, $\alpha(\epsilon) = \epsilon^{\#}$ by (Eps). Again by application of (Gal), we have $\epsilon \leq \gamma(\alpha(\epsilon)) = \gamma(\epsilon^{\#})$, so $\gamma(\epsilon^{\#}) = \epsilon$ by $\leq$-maximality of ϵ.

Finally, we prove (6.18). For all $\sigma^{\#} \in S^{\#}$ we have $\blacktriangledown \leq \gamma(\sigma^{\#})$ since $\blacktriangledown$ is the least element of $S_{\blacktriangledown}$. By (Gal) and Lemma 6.27, this means that $\alpha(\blacktriangledown) \leq \sigma^{\#}$ for all $\sigma^{\#} \in S^{\#}$, which implies $\alpha(\blacktriangledown) = \blacktriangledown^{\#}$. Conversely, for $\sigma \in S_{\blacktriangledown}$ with $\alpha(\sigma) = \blacktriangledown^{\#}$ by (Gal) and (Bot) we have $\sigma \leq \gamma(\alpha(\sigma)) = \gamma(\blacktriangledown^{\#}) = \blacktriangledown$, which implies $\sigma = \blacktriangledown$. □

The other properties in Definition 6.28 ensure that α and γ are well-behaved with respect to the stack operations. Property (TopHom) ensures that the top elements of a stack and its abstract versions are the same. The other two properties consider two different ways to perform a push or pop operation on abstract stacks: The first way applies the respective abstract operation, whereas the second way first concretizes the stack, applies the concrete operation and then abstracts the result. The two properties (PushHom) and (PopHom) state that the first way must provide as most as much information as the second way.

In the following, I give an important example of a family of stack abstractions. To support the uniform presentation of the following example, I define $A^{\leq\infty} \overset{def}{=} A^{\star}$ for an alphabet A, $\sigma^{\leq\infty} \overset{def}{=} \sigma$ for every $\sigma \in A^{\star}$ and $\infty - 1 \overset{def}{=} \infty$. Furthermore, I assume that the natural order $\leq$ on $\mathbb{N}$ is extended to $\mathbb{N}_{\infty} \overset{def}{=} \mathbb{N} \cup \{\infty\}$ by defining $x \leq \infty$ for every $x \in \mathbb{N}_{\infty}$.

Example 6.30. *For $k, l \in \mathbb{N}_{\infty}$, $l \leq k$, consider the two stack spaces $\mathcal{S}_k = (E_{call}^{\leq k}, \leq_k, \epsilon_k, push_k, pop_k, top_k)$ and $\mathcal{S}_l = (E_{call}^{\leq l}, \leq_l, \epsilon_l, push_l, pop_l, top_l)$. Define the functions $\alpha_{k,l}$ and $\gamma_{k,l}$ by*

$$\begin{aligned} \alpha_{k,l}(\sigma) &= \sigma^{\leq l} \\ \alpha_{k,l}(\blacktriangledown_k) &= \blacktriangledown_l \\ \gamma_{k,l}(\sigma) &= \sigma \\ \gamma_{k,l}(\blacktriangledown_l) &= \blacktriangledown_k \end{aligned}$$

Then $(\mathcal{S}_k, \alpha_{k,l}, \gamma_{k,l}, \mathcal{S}_l)$ is a stack abstraction.

Proof. (Gal) $(\mathcal{S}_k, \alpha_{k,l}, \gamma_{k,l}, \mathcal{S}_l)$ is a galois connection.

- For $\sigma \in E_{call}^{\leq k}$ we have

$$\sigma \leq_k \sigma^{\leq l} \qquad \{ \ \sigma^{\leq l} \text{ is a prefix of } \sigma \ \}$$

$$
\begin{aligned}
&= \gamma_{k,l}(\sigma^{\leq l}) && \{ \text{ definition of } \gamma_{k,l} \text{ } \} \\
&= \gamma_{k,l}(\alpha_{k,l}(\sigma)) && \{ \text{ definition of } \alpha_{k,l} \text{ } \}
\end{aligned}
$$

- For $\sigma \in E_{call}^{\leq l}$ we have

$$
\begin{aligned}
\alpha_{k,l}(\gamma_{k,l}(\sigma)) &= \alpha_{k,l}(\sigma) && \{ \text{ definition of } \gamma_{k,l} \text{ } \} \\
&= \sigma^{\leq l} && \{ \text{ definition of } \alpha_{k,l} \text{ } \} \\
&= \sigma && \{ \text{ since } \sigma \in E_{call}^{\leq l} \text{ } \} \\
&\leq_l \sigma && \{ \text{ Reflexivity of } \leq_l \text{ } \}
\end{aligned}
$$

(Bot) clear by definition.

(Eps) This is clear since $\gamma_{k,l}$ is the identity function.

(PushHom) Let $\sigma \in E_{call}^{\leq l}$. Then we have

$$
\begin{aligned}
\alpha_{k,l}(push_k(e, \gamma_{k,l}(\sigma))) &= \alpha_{k,l}(push_k(e, \sigma)) && \{ \text{ definition of } \gamma_{k,l} \text{ } \} \\
&= \alpha_{k,l}((e \cdot \sigma)^{\leq k}) && \{ \text{ definition of } push_k \text{ } \} \\
&= ((e \cdot \sigma)^{\leq k})^{\leq l} && \{ \text{ definition of } \alpha_{k,l} \text{ } \} \\
&= (e \cdot \sigma)^{\leq l} && \{ \text{ } l \leq k \text{ } \} \\
&= push_l(e, \sigma) && \{ \text{ definition of } push_l \text{ } \}
\end{aligned}
$$

(PopHom) The claim is trivial for $l = 0$, so we may assume $l > 0$. For $e \cdot \sigma \in E_{call}^{\leq l} \setminus \{\epsilon\}$, we have

$$
\begin{aligned}
\alpha_{k,l}(pop_k(\gamma_{k,l}(e \cdot \sigma))) &= \alpha_{k,l}(pop_k(e \cdot \sigma)) && \{ \text{ definition of } \gamma_{k,l} \text{ } \} \\
&= \alpha_{k,l}(\sigma)) && \{ \text{ definition of } pop_k \text{ } \} \\
&= \sigma^{\leq l} && \{ \text{ definition of } \alpha_{k,l} \text{ } \} \\
&= \sigma && \{ \text{ } \sigma \in E_{call}^{\leq l-1} \subseteq E_{call}^{\leq l} \text{ } \} \\
&= pop_l(e \cdot \sigma) && \{ \text{ definition of } pop_l \text{ } \}
\end{aligned}
$$

(TopHom) Again, the claim is trivial for $l = 0$, so we only consider $l > 0$. For $e \cdot \sigma \in E_{call}^{\leq l} \setminus \{\epsilon\}$, we have $\gamma_{k,l}(\sigma) = \sigma$ and therefore $top_k(\gamma_{k,l}(\sigma)) = top_l(\sigma)$.

□

After I have introduced stack abstractions and gave an example, I provide the main result of this subsection in Theorem 6.31. If $\mathcal{S}$ and $\mathcal{S}^{\#}$ are two stack spaces and α and γ form a stack abstraction between $\mathcal{S}$ and $\mathcal{S}^{\#}$, then every $\mathcal{S}$-acceptable path is also $\mathcal{S}^{\#}$-acceptable.
The proof of Theorem 6.31 is an easy consequence of the following key property, which I will prove later.

Lemma 6.32. *Let $(S, \alpha, \gamma, S^{\#})$ be a stack abstraction between the stack spaces $(S, \leq, \epsilon, top, push, pop)$ and $(S^{\#}, \leq^{\#}, \epsilon^{\#}, top^{\#}, push^{\#}, pop^{\#})$ over E_{call}. Then the following statement is true for any $\pi \in Paths(G)$:*

$$\alpha(cs(\pi)) \leq cs^{\#}(\pi) \tag{6.21}$$

Lemma 6.32 can be used to show the main result of this section.

Theorem 6.31. *Let*

$$\mathcal{S} = (E_{call}, S, \leq, \epsilon, top, push, pop)$$

and

$$\mathcal{S}^{\#} = (E_{call}, S^{\#}, \leq^{\#}, \epsilon^{\#}, top^{\#}, push^{\#}, pop^{\#})$$

be two stack spaces over E_{call}. Furthermore, let $(S, \alpha, \gamma, S^{\#})$ be a stack abstraction between $\mathcal{S}$ and $\mathcal{S}^{\#}$. Then

$$\forall s, t \in N.\ AP_{\mathcal{S}}(s,t) \subseteq AP_{\mathcal{S}^{\#}}(s,t) \tag{6.19}$$

$$\mathrm{MO}AP_{\mathcal{S}} \leq \mathrm{MO}AP_{\mathcal{S}^{\#}} \tag{6.20}$$

Proof. First, we show (6.19). Let $\pi \in AP_{\mathcal{S}}(s,t)$. Then $cs(\pi) \neq \blacktriangledown$ which by Remark 6.29 implies $\alpha(cs(\pi)) \neq \blacktriangledown^{\#}$. By Lemma 6.32 it follows that $\alpha(cs(\pi)) \leq cs^{\#}(\pi)$, which entails $cs^{\#}(\pi) \neq \blacktriangledown^{\#}$. Thus $\pi \in AP_{\mathcal{S}^{\#}}(s,t)$.
With (6.19), we see that $\mathrm{MO}AP_{\mathcal{S}^{\#}}$ joins over more paths than $\mathrm{MO}AP_{\mathcal{S}}$. This shows (6.20). □

All that is left to do for this subsection is to prove Lemma 6.32.

Lemma 6.32. *Let $(S, \alpha, \gamma, S^{\#})$ be a stack abstraction between the stack spaces $(S, \leq, \epsilon, top, push, pop)$ and $(S^{\#}, \leq^{\#}, \epsilon^{\#}, top^{\#}, push^{\#}, pop^{\#})$ over E_{call}. Then the following statement is true for any $\pi \in Paths(G)$:*

$$\alpha(cs(\pi)) \leq cs^{\#}(\pi) \tag{6.21}$$

Proof. First, we observe that

$$\alpha(cs(\pi)) \leq cs^{\#}(\pi) \iff cs(\pi) \leq \gamma(cs^{\#}(\pi)) \tag{6.22}$$

This follows from (Gal) and Lemma 6.27.
Next we show

$$\forall \pi \in Paths(G).\ \alpha(cs(\pi)) \leq cs^{\#}(\pi)$$

by induction on the length of π. So let $n \in \mathbb{N}$, $\pi \in Paths(G)$ with $|\pi| = n$. The induction hypothesis is

$$\textbf{(IH)} \qquad \forall \pi' \in Paths(G).\ |\pi'| < n \implies \alpha(cs(\pi)) \leq cs^{\#}(\pi)$$

By (6.22), this is equivalent to

$$\textbf{(IH')} \qquad \forall \pi' \in Paths(G).\ |\pi'| < n \implies cs(\pi') \leq \gamma(cs^{\#}(\pi'))$$

Now we make a case distinction on n.

$n = 0$: Then $\pi = \epsilon$. Evaluation of both sides yields that they are equal: $\alpha(cs(\pi)) = \alpha(\epsilon) = \epsilon^{\#}$ and $cs^{\#}(\pi) = \epsilon^{\#}$.

$n > 0$: Then $\pi = \pi' \cdot e$ with $|\pi'| = n - 1$ and $e \in E_{intra} \cup E_{call} \cup E_{ret}$.

First we consider the case that $cs(\pi') = \blacktriangledown$. Then $cs(\pi) = \blacktriangledown$ and hence, by (6.18), $\alpha(cs(\pi)) = \blacktriangledown^{\#} \leq cs^{\#}(\pi)$.

Next assume $cs^{\#}(\pi') = \blacktriangledown^{\#}$. Then **(IH)** implies $\alpha(cs(\pi')) \leq \blacktriangledown^{\#}$. Hence, $\alpha(cs(\pi')) = \blacktriangledown^{\#}$, and this implies $cs(\pi') = \blacktriangledown$ by (6.18). As in the previous case, we can conclude that $\alpha(cs(\pi)) \leq cs^{\#}(\pi)$ also holds in this case.

Now we can assume $cs(\pi') \neq \blacktriangledown$ and $cs^{\#}(\pi') \neq \blacktriangledown^{\#}$ and make a case distinction on e:

$e \in E_{intra}$:

$$\begin{array}{ll}
\ \ \alpha(cs(\pi)) & \\
= \alpha(cs(\pi')) & \{\text{ by definition, } e \in E_{intra} \} \\
\leq cs^{\#}(\pi') & \{\ \textbf{(IH)}\ \} \\
= cs^{\#}(\pi) & \{\text{ by definition, } e \in E_{intra} \}
\end{array}$$

$e \in E_{call}$:

$$\begin{aligned}
&\alpha(cs(\pi)) \\
&= \alpha(push(e, cs(\pi'))) && \{ \text{ by definition, } e \in E_{call} \} \\
&\leq \alpha(push(e, \gamma(cs^{\#}(\pi')))) && \{ \textbf{ (IH')}, \text{ mon. of } push(e, \cdot) \text{ and } \alpha \} \\
&\leq push^{\#}(e, cs^{\#}(\pi')) && \{ \textbf{ (PushHom) } \} \\
&= cs^{\#}(\pi) && \{ \text{ by definition, } e \in E_{call} \}
\end{aligned}$$

case $e \in E_{ret}$: By definition of cs, we distinguish two cases:

$cs(\pi') = \epsilon$: By **(IH')**, this implies $\epsilon \leq \gamma(cs^{\#}(\pi'))$. It follows that $\gamma(cs^{\#}(\pi')) = \epsilon$. Hence $cs^{\#}(\pi') = \epsilon^{\#}$ and, by definition of $cs^{\#}$, $cs^{\#}(\pi) = \epsilon^{\#}$.

case $cs(\pi') \neq \epsilon$ By assumption, we have $cs(\pi') \notin \{\blacktriangledown, \epsilon\}$.

First consider the case that $(top(cs(\pi')), e) \notin \Phi$. Then $cs(\pi) = \blacktriangledown$ by definition of cs. Hence, we can justify (6.21) as follows:

$$\begin{aligned}
\alpha(cs(\pi)) &= \alpha(\blacktriangledown) && \{ \text{ see above } \} \\
&= \blacktriangledown^{\#} && \{ \text{ (6.18) } \} \\
&\leq cs^{\#}(\pi). && \{ cs^{\#}(\pi) \in S^{\#}, \blacktriangledown^{\#} \text{ is the least element of } S^{\#} \}
\end{aligned}$$

Now we may assume that $cs(\pi') \notin \{\blacktriangledown, \epsilon\} \wedge (top(cs(\pi')), e) \in \Phi$. Consider the case that $cs^{\#}(\pi') = \epsilon^{\#}$. Then $cs^{\#}(\pi) = \epsilon^{\#}$, so that $\alpha(cs(\pi)) \leq cs^{\#}(\pi)$ holds.

Now assume $cs^{\#}(\pi') \neq \epsilon^{\#}$. Together with $cs^{\#}(\pi') \neq \blacktriangledown^{\#}$, we have $cs^{\#}(\pi') \notin \{\blacktriangledown^{\#}, \epsilon^{\#}\}$. By **(IH')**, we have $cs(\pi') \leq \gamma(cs^{\#}(\pi'))$. Since $cs^{\#}(\pi') \neq \epsilon^{\#}$, it must be $\gamma(cs^{\#}(\pi')) \neq \epsilon$ (by (Eps)).

Because $cs^{\#}(\pi') \neq \epsilon^{\#}$, $cs(\pi') \neq \epsilon$ and $\gamma(cs^{\#}(\pi')) \neq \epsilon$, we can conclude from $cs(\pi') \leq \gamma(cs^{\#}(\pi'))$ that

$$top(cs(\pi')) = top(\gamma(cs^{\#}(\pi'))) = top^{\#}(cs^{\#}(\pi'))$$

by (TopVsLe) and (TopHom)). Together with $(top(cs(\pi)), e) \in \Phi$, it follows that $(top(cs^{\#}(\pi)), e) \in \Phi$, so that $cs^{\#}(\pi) = pop^{\#}(cs^{\#}(\pi'))$ by definition of $cs^{\#}$. Now we can conclude

$$
\begin{aligned}
& \alpha(cs(\pi)) \\
= \; & \alpha(pop(cs(\pi'))) && \{ \text{ definition of } cs\text{, assumptions } \} \\
\leq \; & \alpha(pop(\gamma(cs^{\#}(\pi')))) && \{ \textbf{ (IH')}\text{, mon. of } pop \text{ and } \alpha \; \} \\
\leq \; & pop^{\#}(cs^{\#}(\pi')) && \{ \textbf{ (PopHom) } \} \\
= \; & cs^{\#}(\pi) && \{ \text{ definition of } cs^{\#}\text{, see above } \}
\end{aligned}
$$

□

6.3.4 Effective and Correct Approximations of the Unbounded Call-String Approach

In subsection 6.3.2, we saw that solutions of Constraint System 6.5 w.r.t. $\mathcal{S}_\infty$ enjoy the same correctness and precision results as the functional approaches. Unfortunately, Constraint System 6.5 is not *effective*. For one, Constraint System 6.5 w.r.t. $\mathcal{S}_\infty$ is infinite. Secondly, the complete lattice that forms the solution space of Constraint System 6.5 is

$$N \times N \times S^\infty \to F_\boxtimes,$$

which does not satisfy the ascending chain condition. Particularly this second fact makes it impossible to use algorithms such as Algorithm 8, which rely on the ascending chain condition for termination.

However, in Example 6.13 I showed the family of $(\mathcal{S}_k)_{k\in\mathbb{N}}$ of stack spaces that lead to finite versions of Constraint System 6.5 and to solution spaces that satisfy the ascending chain condition. Hence algorithms like Algorithm 8 are applicable to Constraint System 6.5 w.r.t. $\mathcal{S}_k$. The idea of $\mathcal{S}_k$, which is at least since the work of Sharir and Pnueli [154] well-known in the literature, is to only consider the k topmost stack elements, so that all stacks are k-bounded. As Example 6.30 showed, $\mathcal{S}_\infty$ and $\mathcal{S}_k$ are connected via a stack abstraction, where the abstraction function projects a stack to the k topmost items.

Consequently, the main result Theorem 6.31 from the last subsection enables me to transfer the correctness result for $\mathcal{S}_\infty$ to $\mathcal{S}_k$.

Corollary 6.33. *1. Let $\mathcal{S}$ be a stack space such that there is a stack abstraction between $\mathcal{S}_\infty$ and $\mathcal{S}$. Then the following statements hold:*

$$\forall s, t \in N.\ VP(s,t) \subseteq AP_{\mathcal{S}}(s,t) \tag{6.23}$$

$$\forall s, t, \in N.\ \mathrm{MO}VP(s,t) \leq \mathrm{MO}AP(s,t) \tag{6.24}$$

2. Let $k, l \in \mathbb{N} \cup \{\infty\}$ with $l \leq k$. Then the following statements hold:

$$\mathrm{MO}VP \leq \mathrm{MO}AP_{\mathcal{S}_k} \tag{6.25}$$

$$\mathrm{MO}AP_{\mathcal{S}_k} \leq \mathrm{MO}AP_{\mathcal{S}_l} \tag{6.26}$$

Proof. 1. (6.23) follows from Theorem 6.31 and Theorem 6.25. The second statement (6.24) is an easy consequence of (6.23).

2. In Example 6.30 we have seen that $(\mathcal{S}_k, \alpha_{k,l}, \gamma_{k,l}, \mathcal{S}_l)$ is a stack abstraction if $k, l \in \mathbb{N} \cup \{\infty\}$ with $l \leq k$. This means that we can apply (6.23) and (6.24) to obtain the desired statements.

□

Together with Theorem 7.34, this entails that solutions of Constraint System 6.5 w.r.t. $\mathcal{S}_k$ are VP-correct. Hence, using k-bounded stacks in the call-strings approach is indeed correct. This is a generalization of the corresponding result that was obtained by Sharir and Pnueli for classical data-flow analyses on interprocedural control-flow graph [154, Theorem 7-6.5]. For one, my assumptions about the given graph and its valid paths are weaker (as I already explained in chapter 5). Moreover, there may be other stack spaces than $\mathcal{S}_k$ that can be obtained from $\mathcal{S}_\infty$ via a stack abstraction and lead to an effective version of Constraint System 6.5. The theory that I developed in this section provides a way to prove VP-correctness for all these stack spaces.

I have suffered for this book; now it's your turn.
George Harrison

7 A Common Generalization of Interprocedural Data-Flow Analysis and Slicing

After chapter 6 introduced constraint systems for the functional and the call-string approach to interprocedural data-flow analysis for interprocedural graphs, this chapter is concerned with the effective solution of meaningful portions of these constraint systems in order to compute correct and possibly precise approximations to MO$VP$.
In chapter 3, I compared slicers such as Algorithm 5 and Algorithm 6 with algorithms to solve data-flow problems, such as Algorithm 3. My conclusion was that although both algorithm groups are worklist-based, they differ in what I coined as *worklist policy*. While the slicers usually assume that worklist items just have been updated, update the values of their successors and put them on the worklist if their value has changed, Algorithm 3 acts on the assumption that worklist items have to be updated, updates them using all their predecessors and then puts all successors on the worklist. This difference becomes obvious if one considers how the two approaches handle reachability. While slicers are reachability analyses and naturally explore graphs iteratively using a reachability frontier, Algorithm 3 is geared towards situations where the goal is to compute values for every variable and the usual assumption is that every variable is reachable. Hence, Algorithm 3 proceeds rather clumsily if the analysis information to be computed is reachability itself: It uses the rule "this variable is reachable if one of its predecessors is reachable" and not – like it is natural – "if this variable is reachable, then all its successors are reachable". It *is possible* to use Algorithm 3 for data-flow problems where reachability

is part of the analysis result, even without modifications. However, this necessitates modifications of the data-flow problems. Moreover, the initial loop, which iterates through *all* variables cannot be eliminated – its task is to assess that all unreachable variables are indeed unreachable because all their predecessors are. I take a different path to integrate reachability and data-flow analysis and modify Algorithm 3 in such a way that it is geared towards reachability and computes the actual analysis result for reachable variables as a side-product. Particularly, Algorithm 8, my modified version of Algorithm 3, does not need to visit every variable at least once but only the reachable variables.

The presentation and analysis of Algorithm 8 will be the subject of section 7.1.

Moreover, section 7.1 considers reachability and partial solution on the level of constraint systems. This is particularly necessary to be able to make statements about the properties of Algorithm 8. Moreover, I investigate under which circumstances the least solution of the restricted constraint system coincides with the least solution of the full constraint system on the reachable variables. This can also be seen as *slicing of constraint systems*. It turns out that the modified version of Algorithm 3 is indeed capable of computing such partial solutions.

Subsequently, I apply the theory developed in section 7.1 to the constraint systems given in chapter 6 in order to obtain algorithms that compute correct data-flow solutions on forward slices. Section 7.2 lays out the setup and gives an overview of the general scheme. Then, section 7.3 and section 7.4 consider the functional approach and the call-string approach, respectively.

7.1 Integrating Reachability Into the Solution Process

As was already mentioned at the very beginning of this chapter, the goal of this section is to present Algorithm 8, a variant of Algorithm 3 that additionally takes a set X_0 of initial variables and solves only the part of the given constraint system that is *reachable* from this initial set of variables. This set cannot be chosen arbitrarily: The properties that X_0 has to satisfy in order for Algorithm 8 are one of the topics of this section. How X_0 is chosen concretely is dependent on the context and will be discussed in later sections of this chapter.

In the following, I consider a complete lattice L that satisfies the ascending chain condition. The idea behind Algorithm 8 is that it starts with applying the defining constraints of X_0 and then proceeds by applying all constraints that use variables from X_0. Then it considers the variables whose values have changed in the same way and continues this procedure until no more value is changed. In the end, Algorithm 8 has computed a partial solution of the given constraint system and explored the variables and constraints that are reachable from X_0 using a chain of def-use relations.

To make this idea work, I need a way to distinguish variables that have been explored and whose solution value may be $\perp_L$ from those which have not yet been explored and never will be[34]. For this purpose, I introduce a fresh value $\boxtimes$ that represents undefinedness and extend L to $L_\boxtimes$. Moreover, I also extend the interpretation so that it can also take $\boxtimes$ as input and assume that the extended interpretation has the properties (7.1) and (7.2). Remember from chapter 2 that $FV(t)$ denotes the set of variables occurring in t.

$$\begin{array}{lll}
(7.1) & \forall x \in L.\ \boxtimes \leq x & \\
(7.2) & \forall x \geq t \in C.\ \forall \psi : X \to L_\boxtimes. & [\![t]\!](\psi) = \boxtimes \\
 & & \iff \exists x' \in FV(t).\ \psi(x') = \boxtimes.
\end{array}$$

These two properties ensure that reachable variables which are set to the ordinary $\perp$ element of L can be distinguished from those variables which have not been reached yet or will never be reached. Particularly, the second property ensures that reachability is maintained.

Also, the extension of L to $L_\boxtimes$ is always possible without destroying important properties of L. $L_\boxtimes$ is still complete lattice that satisfies the ascending chain condition.

Note that in general (7.2) restricts the possible interpretations, since (7.2) essentially enforces that a constraint can be omitted if not all variables on its right-hand side have defining constraints.

[34] Note that the unreachable variables do not actually exist in the memory of a machine executing Algorithm 8. In practice, the reachability of a variable can be assessed by evaluating whether Algorithm 8 already has assigned a value to this variable. Nevertheless, I need this additional element $\boxtimes$ to reason theoretically about unreachable variables.

However, the semantics functional $[\![\cdot]\!] : Expr(\underline{\mathcal{F}}_{\boxtimes}, X) \to F_{\boxtimes}$ that corresponds to a data-flow framework instance $\mathcal{F} = (G, L, F, \rho)$ and that I specified in section 6.1 satisfies both (7.1) and (7.2): (7.1) follows from (5.12) and (5.13), while (7.2) can be shown by induction with the help of (5.16) and (5.15).

Next, I want to give a formal definition of reachable constraints and variables. For motivation, I give an example. Consider the constraint system C that consists of the constraints

$$\begin{aligned} c_1 &: v_1 \geq f(v_2, v_3) \\ c_2 &: v_2 \geq g(v_3) \\ c_3 &: v_3 \geq a \\ c_4 &: v_4 \geq h(v_3, v_5) \\ c_5 &: v_4 \geq b \end{aligned}$$

where f, g, h, a, b are function symbols with appropriate interpretations and the v_i are variables.

If we start with v_3 and follow the def-use chains in C, we see that c_1, c_2, c_3 and c_4 are the constraints from C that are reachable from v_3. Moreover, the reachable variables are v_1, v_2 and v_3.

Also note that c_4 is not really useful as it uses the variable v_5, which does not have any defining constraints in C. Hence, no solution of C needs to assign v_5 any defined value. Particularly, the least solution $lfp(F_C)$ must map v_4 to $\boxtimes$, which implies that $lfp(F_C)(v_4) = lfp(F_{C \setminus \{c_4\}})(v_4)$. Similarly, we can conclude that it is not really useful to start with variables that have defining constraints with non-constant right-hand side, if one wants to preserve the least solution for the reachable variables. For example, if we start with v_2, then we get the reachable constraints c_1 and c_2, but both use v_3, which is defined by neither c_1 nor c_2, so that $lfp(F_C)(v_1)$ and $lfp(F_C)(v_2)$ both become $\boxtimes$ if we restrict C to $\{c_1, c_2\}$.

Given a variable set $X_0 \subseteq X$, the following definition introduces the sets $\mathcal{C}\mathrm{ore}(C, X_0)$ and $\mathcal{C}\mathrm{oreVars}(C, X_0)$. The former captures the set of constraints that are obtained by starting at X_0 and following the def-use chains in C, while the latter characterizes the variables defined by constraints from $\mathcal{C}\mathrm{ore}(C, X_0)$.

Remember from chapter 2 that $Vars(C)$ is the set of variables that occur on the left-hand sides of constraints in C and that $Def(x)$ is the set of constraints for which x occurs on the left-hand side, respectively.

Definition 7.1 (core and core variables of a constraint system)**.** *Let C be a constraint system and $X_0 \subseteq X$ be a set of variables.*

1. $\mathcal{C}\text{ore}(C, X_0)$, the core *of C with respect to X_0, is defined as the least subset $C_0 \subseteq C$ with the following closure properties:*

$$
\begin{array}{ll}
\text{(C-B{\scriptsize ASE})} & \forall x \geq t \in C.\ FV(t) = \emptyset \wedge x \in X_0 \Longrightarrow\ x \geq t \in C_0 \\
\text{(C-S{\scriptsize TEP})} & \forall x \geq t \in C.\ FV(t) \neq \emptyset \\
 & \wedge \forall x' \in FV(t).\ Def(x') \cap C_0 \neq \emptyset\ \Longrightarrow\ x \geq t \in C_0
\end{array}
$$

2. $\mathcal{C}\text{oreVars}(C, X_0)$, the core variables *of C, is defined as the least subset $X_c \subseteq Vars(C)$ that has the following closure properties:*

$$
\begin{array}{ll}
\text{(V-B{\scriptsize ASE})} & \forall x \geq t \in C.\ FV(t) = \emptyset \wedge x \in X_0 \qquad \Longrightarrow\ x \in X_c \\
\text{(V-S{\scriptsize TEP})} & \forall x \geq t \in C.\ FV(t) \neq \emptyset \wedge FV(t) \subseteq X_c \Longrightarrow\ x \in X_c
\end{array}
$$

The following two lemmas shed some light on the properties of $\mathcal{C}$ore and $\mathcal{C}$oreVars and will be of great use later.

Lemma 7.2.

$$Vars(\mathcal{C}\text{ore}(C, X_0)) = \mathcal{C}\text{oreVars}(C, X_0)$$

Proof. $\supseteq$: Show that $Vars(\mathcal{C}\text{ore}(C, X_0))$ has the closure properties of $\mathcal{C}\text{oreVars}(C, X_0)$:

1. Let $x \geq t \in C$ with $FV(t) = \emptyset$ and $x \in X_0$. Then $x \geq t \in \mathcal{C}\text{ore}(C, X_0)$, i.e. $x \in Vars(\mathcal{C}\text{ore}(C, X_0))$.

2. Let $x \geq t \in C$ with $FV(t) \neq \emptyset$ and $FV(t) \subseteq Vars(\mathcal{C}\text{ore}(C, X_0))$. Then for every $x' \in FV(t)$, $Def(x') \cap \mathcal{C}\text{ore}(C, X_0) \neq \emptyset$. Hence, $x \geq t \in \mathcal{C}\text{ore}(C, X_0)$, which means that $x \in Vars(\mathcal{C}\text{ore}(C, X_0))$.

$\subseteq$: Define

$$C_0 \overset{def}{=} \{x \geq t \in C \mid x \in \mathcal{C}\text{oreVars}(C, X_0)\}$$

Then

$$(\star) \qquad Vars(C_0) = \mathcal{C}\text{oreVars}(C, X_0).$$

This can be seen as follows:

- If $x \in Vars(C_0)$, then there is $x \geq t \in C_0$. By definition, this means that $x \in \mathcal{C}\text{oreVars}(C, X_0)$.
- If $x \in \mathcal{C}\text{oreVars}(C, X_0)$, then it can easily be seen that $Def(x) \neq \emptyset$. However, since $x \in \mathcal{C}\text{oreVars}(C, X_0)$, it follows that $Def(x) \subseteq C_0$, so that $x \in Vars(C_0)$.

Next we show that $\mathcal{C}\text{ore}(C, X_0) \subseteq C_0$. From this, it follows that

$$Vars(\mathcal{C}\text{ore}(C, X_0)) \subseteq Vars(C_0) = \mathcal{C}\text{oreVars}(C, X_0).$$

To prove $\mathcal{C}\text{ore}(C, X_0) \subseteq C_0$, we show that C_0 has the closure properties (C-Base) and (C-Step).

1. If $x \geq t \in C$, $FV(t) = \emptyset$ and $x \in X_0$, we have $x \in \mathcal{C}\text{oreVars}(C, X_0)$ by (V-Base), hence $x \geq t \in C_0$ by definition.

2. Let $x \geq t \in C$ with $FV(t) \neq \emptyset$ and

$$\forall x' \in FV(t).\, Def(x') \cap C_0 \neq \emptyset.$$

Then

$$FV(t) \subseteq Vars(C_0) \overset{(\star)}{=} \mathcal{C}\text{oreVars}(C, X_0).$$

Hence, $x \in \mathcal{C}\text{oreVars}(C, X_0)$ by property (V-Step), which means that $x \geq t \in C_0$ by definition of C_0.

□

Lemma 7.3. *Let C be a constraint system and $X_0 \subseteq X$ be a set of variables. Then the following statements are equivalent:*

(1) $x \geq t \in \mathcal{C}\text{ore}(C, X_0)$

(2) $(FV(t) = \emptyset \wedge x \in X_0) \vee (FV(t) \neq \emptyset \wedge FV(t) \subseteq \mathcal{C}\text{oreVars}(C, X_0))$

Proof. (1) $\Longrightarrow$ (2) Assume $x \geq t \in \mathcal{C}\mathrm{ore}(C, X_0)$. Then either $FV(t) = \emptyset$ and $x \in X_0$ or $FV(t) \neq \emptyset$ and $\forall x' \in FV(t).\, Def(x') \cap \mathcal{C}\mathrm{ore}(C, X_0) \neq \emptyset$. In the former case, (2) holds trivially. In the latter case, we conclude

$$\forall x' \in FV(t).\, x' \in Vars(\mathcal{C}\mathrm{ore}(C, X_0)),$$

which is, due to Lemma 7.2, equivalent to

$$\forall x' \in FV(t).\, x \in \mathcal{C}\mathrm{oreVars}(C, X_0).$$

(2) $\Longrightarrow$ (1) We prove the claim by case distinction.

1. Assume that

$$x \in X_0 \wedge FV(t) = \emptyset.$$

Then $x \geq t \in \mathcal{C}\mathrm{ore}(C, X_0)$ by (C-Base).

2. Assume that

$$FV(t) \neq \emptyset \wedge FV(t) \subseteq \mathcal{C}\mathrm{oreVars}(C, X_0).$$

With the definition of *Vars* and Lemma 7.2, this implies

$$\forall x' \in FV(t).\, Def(x') \cap \mathcal{C}\mathrm{ore}(C, X_0) \neq \emptyset.$$

Thus $x \geq t \in \mathcal{C}\mathrm{ore}(C, X_0)$ by (C-Step).

□

$\mathcal{C}\mathrm{ore}(C, X)$ only eliminates useless constraints. Therefore, the least solution of $\mathcal{C}\mathrm{ore}(C, X)$ must coincide with the least solution of C on the whole variable set.

Lemma 7.4. $\mathcal{C}\mathrm{ore}(C, X)$ *has the same least solution as* C.

Proof. First, we show that the two least solutions must coincide on $Vars(\mathcal{C}\mathrm{ore}(C, X))$.
It is clear that $lfp(F_{\mathcal{C}\mathrm{ore}(C,X)}) \leq lfp(F_C)$, since $\mathcal{C}\mathrm{ore}(C, X) \subseteq C$.
Next, we observe

$$\forall x \in X \setminus Vars(\mathcal{C}\mathrm{ore}(C, X)).\, lfp(F_{\mathcal{C}\mathrm{ore}(C,X)})(x) = \boxtimes. \tag{7.3}$$

The reason is that by definition, $\mathcal{C}\mathrm{ore}(\mathcal{C}, X)$ has no defining constraints for variables outside of $Vars(\mathcal{C}\mathrm{ore}(\mathcal{C}, X))$.
Finally, we show

$$lfp(F_{\mathcal{C}}) \leq lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)}) \tag{7.4}$$

by fixed-point induction (cf. Corollary 2.7).
Let

$$\mathcal{P} \overset{def}{=} \{\psi : X \to L \mid \psi \leq lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})\}.$$

It is easy to see that $const(\boxtimes) \in \mathcal{P}$ and that $\mathcal{P}$ is closed under arbitrary joins. It remains to show that $\mathcal{P}$ is also closed under $F_{\mathcal{C}}$. For this, it suffices to show

$$\forall \psi \in \mathcal{P}.\ \forall x \geq t \in \mathcal{C}.\ \llbracket t \rrbracket(\psi) \leq lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})(x)$$

So let $\psi \in \mathcal{P}$ and $x \geq t \in \mathcal{C}$. We have to show that

$$\llbracket t \rrbracket(\psi) \leq \llbracket t \rrbracket(lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})) \leq lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})(x). \tag{7.5}$$

This follows from the two properties

$$\llbracket t \rrbracket(\psi) \leq \llbracket t \rrbracket(lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})) \tag{7.6}$$

and

$$\llbracket t \rrbracket(lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})) \leq lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})(x), \tag{7.7}$$

which we are going to show in the following.

(7.6) By our assumption that $\psi \in \mathcal{P}$, we have in particular

$$\forall x' \in FV(t).\ \psi(x') \leq lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})(x').$$

From this, (7.6) can be shown using Lemma 2.10.

(7.7) We distinguish two cases:

1. $x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}, X)$: Then (7.7) holds because $lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C},X)})$ is a solution of $\mathcal{C}\mathrm{ore}(\mathcal{C}, X)$.

2. $x \geq t \notin \mathcal{C}\text{ore}(C, X)$. Then it must be the case that

$$FV(t) \neq \emptyset \wedge \exists x' \in FV(t).\, x' \in X \setminus Vars(\mathcal{C}\text{ore}(C, X)),$$

since otherwise the closure properties of $\mathcal{C}\text{ore}(C, X)$ would immediately imply $x \geq t \in \mathcal{C}\text{ore}(C, X)$.

But then, due to (7.3), there must be one $x' \in FV(t)$ with

$$lfp(F_{\mathcal{C}\text{ore}(C,X)})(x') = \boxtimes.$$

Now, (7.7) follows with (7.2) and (7.1).

□

Now I want to consider how $\mathcal{C}$ore behaves for $X_0 \neq X$.
The following lemma shows that $\mathcal{C}\text{ore}(C, X_0)$ can be obtained by first computing $\mathcal{C}\text{ore}(C, X)$ and then restricting the variables to X_0. The first step eliminates all constraints that do not contribute anything, while the second step eliminates all constraints that may contribute to the solution but depend on variables not in X_0.

Lemma 7.5. *Applying* $\mathcal{C}\text{ore}(\_, X_0)$ *to* C *is the same as applying it to* $\mathcal{C}\text{ore}(C, X)$*:*

$$\mathcal{C}\text{ore}(C, X_0) = \mathcal{C}\text{ore}(\mathcal{C}\text{ore}(C, X), X_0). \tag{7.8}$$

Proof. First, we observe that $\mathcal{C}\text{ore}(C, X)$ is a subset of C and $\mathcal{C}$ore is monotone in its first argument. This implies

$$\mathcal{C}\text{ore}(C, X_0) \supseteq \mathcal{C}\text{ore}(\mathcal{C}\text{ore}(C, X), X_0).$$

It remains to show "$\subseteq$". For this, we show that $\mathcal{C}\text{ore}(\mathcal{C}\text{ore}(C, X), X_0)$ has the properties (C-Base) and (C-Step) with respect to C and X_0.

1. Let $x \geq t \in C$ with $FV(t) = \emptyset$ and $x \in X_0$. Then in particular $x \in X$. Hence, by applying (C-Base) to $x \geq t \in C$ and $x \in X$, we get

$$x \geq t \in \mathcal{C}\text{ore}(C, X).$$

Then we apply (C-Base) to $x \in X_0$ and $x \geq t \in \mathcal{C}\text{ore}(C, X)$ and get our desired result

$$x \geq t \in \mathcal{C}\text{ore}(\mathcal{C}\text{ore}(C, X), X_0).$$

2. Let $x \geq t \in C$ with $FV(t) \neq \emptyset$ and

$$\forall x' \in FV(t).\ Def(x') \cap \mathcal{C}\mathrm{ore}(\mathcal{C}\mathrm{ore}(C, X), X_0) \neq \emptyset.$$

Since $\mathcal{C}\mathrm{ore}(C, X) \subseteq C$ and $X_0 \subseteq X$, and due to the monotonicity of $\mathcal{C}\mathrm{ore}$, we have

$$\mathcal{C}\mathrm{ore}(\mathcal{C}\mathrm{ore}(C, X), X_0) \subseteq \mathcal{C}\mathrm{ore}(C, X),$$

so that we can weaken the second condition to

$$\forall x' \in FV(t).\ Def(x') \cap \mathcal{C}\mathrm{ore}(C, X) \neq \emptyset.$$

With (C-Step) for C and X, we conclude

$$x \geq t \in \mathcal{C}\mathrm{ore}(C, X).$$

Another application of (C-Step), this time for $\mathcal{C}\mathrm{ore}(C, X)$ and X_0, finally yields

$$x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}\mathrm{ore}(C, X), X_0)$$

□

The intuition of $\mathcal{C}\mathrm{ore}$ is that the least solution of $\mathcal{C}\mathrm{ore}(C, X_0)$ coincides with C on a relevant part of X, namely $\mathcal{C}\mathrm{oreVars}(C, X_0)$. But this is not true for every choice of X_0.

Example 7.6. *Assume that $Expr(\mathcal{F}, \mathcal{X})$ contains appropriate function symbols for expressing subsets of $\{a, b\}$ and set constraints over these sets. Moreover, assume an appropriate interpretation.*

1. *Consider the following constraint system C:*

$$\begin{aligned} x &\supseteq y \\ x &\supseteq \{a\} \\ y &\supseteq \{b\} \end{aligned}$$

Now consider $X_0 = \{x\}$. Then we get

$$\begin{aligned} \mathcal{C}\mathrm{ore}(C, X_0) &= \{x \supseteq \{a\}\} \\ \mathcal{C}\mathrm{oreVars}(C, X_0) &= \{x\} \\ lfp(F_{\mathrm{Core}(C, X_0)})(x) &= \{a\} \\ lfp(F_C)(x) &= \{a, b\} \end{aligned}$$

2. Consider the following constraint system C'.

$$\begin{aligned} y &\supseteq x \\ x &\supseteq \{a\} \\ y &\supseteq \{b\} \end{aligned}$$

Now consider $X'_0 = \{x\}$. Then we get

$$\begin{aligned} \mathcal{C}\mathrm{ore}(C', X'_0) &= \{x \supseteq \{a\}, y \supseteq x\} \\ \mathcal{C}\mathrm{oreVars}(C', X'_0) &= \{x, y\} \\ lfp(F_{\mathcal{C}\mathrm{ore}(C', X'_0)})(y) &= \{a\} \\ lfp(F_{C'})(y) &= \{a, b\} \end{aligned}$$

The two examples each highlight a characteristic problem that prevents $\mathcal{C}\mathrm{ore}(C, X_0)$ from exhibiting the same least solution as C, if X_0 is not chosen appropriately.
In the first example, the problem is that the constraint $x \supseteq y$ is not contained in $\mathcal{C}\mathrm{ore}(C, X_0)$, as it does not depend on a constant constraint that defines x (or, more generally, a variable from X_0).
In the second example, $y \supseteq \{b\}$ is not included in $\mathcal{C}\mathrm{ore}(C', X'_0)$ because y does not belong to X'_0.
An appropriate choice of X_0 hence ensures two things:

1. Every constraint in $\mathcal{C}\mathrm{ore}(C, X_0)$ transitively depends on a constant constraint defining some variable from X_0.

2. If a variable $v \in \mathcal{C}\mathrm{oreVars}(C, X_0)$, then all constant constraints defining v are contained in $\mathcal{C}\mathrm{ore}(C, X_0)$.

In the following, I introduce *definition-completeness* as a property of constraint systems that ensures the coincidence of least solutions. After that, I consider conditions on variable sets that ensure the definition-completeness of the corresponding $\mathcal{C}\mathrm{ore}$ set.

Definition 7.7. *$C_0 \subseteq C$ is called* definition-complete *if it has the property*

$$\forall x \geq t \in C.\ Def(x) \cap C_0 \neq \emptyset \implies Def(x) \subseteq C_0$$

Definition-completeness is indeed sufficient for the coincidence of the least solutions on the core-variables. This is formally stated by Theorem 7.9. Before I can prove that, I need a technical lemma that enables me to perform the fundamental proof steps.

Lemma 7.8. *If* $C_0 \overset{def}{=} Core(C, X_0)$ *is definition-complete, then*

$$\psi \leq_{Vars(C_0)} lfp(F_{C_0}) \implies F_C(\psi) \leq_{Vars(C_0)} F_C(lfp(F_{C_0})) \tag{7.9}$$

$$lfp(F_C) \leq_{Vars(C_0)} lfp(F_{C_0}) \tag{7.10}$$

Proof. First, assume that (7.9). Then we can show (7.10) by fixed-point induction (Corollary 2.7). Let

$$\mathcal{P} \overset{def}{=} \{\psi : X \to L \mid \psi \leq_{Vars(C_0)} lfp(F_{C_0})\}.$$

We need to show that $const(\boxtimes) \in \mathcal{P}$, that $\mathcal{P}$ is closed under arbitrary joins and that $\mathcal{P}$ is closed under F_C. The first two statements can easily be seen and the third claim is proven by (7.9).

Now we prove (7.9). Assume $\psi \leq_{Vars(C_0)} lfp(F_{C_0})$. We need to show that $F_C(\psi) \leq_{Vars(C_0)} F_C(lfp(F_{C_0}))$ and for this it is enough to show

$$\forall x \geq t \in C.\ x \in Vars(C_0) \implies [\![t]\!](\psi) \leq lfp(F_{C_0})(x).$$

So let $x \geq t \in C$ with $x \in Vars(C_0)$. Since C_0 is definition-complete, it must be $x \geq t \in C_0$. By Lemma 7.2 and Lemma 7.3, it follows that $FV(t) \subseteq Vars(C_0)$. But according to our assumption this means that

$$\forall x' \in FV(t).\ \psi(x') \leq lfp(F_{C_0})(x').$$

Using Lemma 2.10, this entails

$$[\![t]\!](\psi) \leq [\![t]\!](lfp(F_{C_0})), \tag{$\star$}$$

and because $lfp(F_{C_0})$ satisfies $x \geq t$ we have

$$[\![t]\!](lfp(F_{C_0})) \leq lfp(F_{C_0})(x). \tag{$\star\star$}$$

From ($\star$) and ($\star\star$) we get

$$[\![t]\!](\psi) \leq lfp(F_{C_0})(x),$$

as desired. □

Theorem 7.9. *If* $\mathcal{C}\text{ore}(C, X_0)$ *is definition-complete, then*

$$lfp(F_C) =_{\mathcal{C}\text{oreVars}(C,X_0)} lfp(F_{\mathcal{C}\text{ore}(C,X_0)}) \tag{7.11}$$

Proof. With $C_0 \overset{def}{=} \mathcal{C}\text{ore}(C, X_0)$, we need show

$$lfp(F_C) \leq_{Vars(C_0)} lfp(F_{C_0}) \tag{7.12}$$

$$lfp(F_{C_0}) \leq_{Vars(C_0)} lfp(F_C) \tag{7.13}$$

With the given assumptions, (7.12) follows directly from Lemma 7.8. Hence, it remains to show (7.13), which can be justified as follows. $\mathcal{C}\text{ore}(C, X_0)$ is a subset of C. Therefore, every solution of C is a solution of $\mathcal{C}\text{ore}(C, X_0)$. Hence, $lfp(F_C)$ is a solution of $\mathcal{C}\text{ore}(C, X_0)$. Since $lfp(F_{\mathcal{C}\text{ore}(C,X_0)})$ is the least solution of $\mathcal{C}\text{ore}(C, X_0)$, this implies $lfp(F_{\mathcal{C}\text{ore}(C,X_0)}) \leq lfp(F_C)$. Clearly, this also holds when restricting to $\mathcal{C}\text{oreVars}(C, X_0)$. □

Theorem 7.10 states conditions on X_0 that characterize definition-completeness. This is basically a formalization of the intuition given in Example 7.6.

Theorem 7.10. $\mathcal{C}\text{ore}(C, X_0)$ *is definition-complete if and only if it satisfies the following two conditions:*

(i) $\forall x \geq t \in C.\ x \in \mathcal{C}\text{oreVars}(C, X_0) \wedge FV(t) = \emptyset$
$\Longrightarrow\ x \in X_0$

(ii) $\forall x \geq t \in C.\ \forall x' \in FV(t).\ x \in \mathcal{C}\text{oreVars}(C, X_0)$
$\Longrightarrow\ x' \in \mathcal{C}\text{oreVars}(C, X_0)$

Proof. We show the two directions separately.

1. Assume that $\mathcal{C}\text{ore}(C, X_0)$ is definition-complete and let $x \geq t \in C$ and $x \in \mathcal{C}\text{oreVars}(C, X_0)$. Then $x \in Vars(\mathcal{C}\text{ore}(C, X_0))$ by Lemma 7.2, which means that $Def(x) \cap \mathcal{C}\text{ore}(C, X_0) \neq \emptyset$. Since $\mathcal{C}\text{ore}(C, X_0)$ is definition-complete, this entails that $x \geq t \in \mathcal{C}\text{ore}(C, X_0)$.

Now we can show (i) and (ii) by considering the two cases $FV(t) = \emptyset$ and $FV(t) \neq \emptyset$:

- If $FV(t) = \emptyset$, $x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0)$ implies $x \in X_0$ by Lemma 7.3, which shows (i).
- Assume that $FV(t) \neq \emptyset$. Together with $x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0)$, we get $FV(t) \subseteq \mathcal{C}\mathrm{oreVars}(\mathcal{C}, X_0)$ by Lemma 7.3. This shows (ii).

2. Assume that (i) and (ii) hold and let $x \in X$ with

$$Def(x) \cap \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0) \neq \emptyset \wedge x \geq t \in Def(x).$$

Together with Lemma 7.2, this implies $x \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}, X_0)$.

Now we show $x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0)$ by case distinction on whether $FV(t) = \emptyset$ or not.

- If $FV(t) = \emptyset$, then with $x \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}, X_0)$ we get $x \in X_0$ by (i). But this means $x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0)$ due to (C-Base).
- If $FV(t) \neq \emptyset$, then with $x \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}, X_0)$ we get

  $$\forall x' \in FV(t).\, x' \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}, X_0)$$

  by (ii). But by definition and Lemma 7.2 this is the same as

  $$\forall x' \in FV(t).\, Def(x) \cap \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0) \neq \emptyset.$$

  This implies $x \geq t \in \mathcal{C}\mathrm{ore}(\mathcal{C}, X_0)$ by (C-Step).

□

Now I am ready to present Algorithm 8, a variant of Algorithm 3 that also uses a worklist approach to compute the solution of a given constraint system, but at the same time also performs a reachability analysis. More specifically, Algorithm 8 takes a set X_0 of variables and, starting with X_0, traverses the constraint dependency graph from usage to definition.

In the following, I state and prove a correctness result for Algorithm 8. This proof is a a variation of the correctness proof for the ordinary worklist algorithm [130] that I already mentioned in chapter 2. This proof is combined with correctness arguments for the reachability parts taken from Takai [160][35].

[35] Tamai [160] also considers integrating reachability into graph problems that can be encoded as constraint systems. They propose an algorithm whose worklist policy is the same as the one for Algorithm 8 and their correctness proof works similarly as the proof of Theorem 7.11. However, Tamai does not encode unreachability explicitly into the lattice.

Algorithm 8: A variant of the worklist algorithm where the items taken off the worklist have just been updated but changes have not been propagated yet

```
Input: a monotone constraint system $C$ with interpretation over
       the variables $X$, a set $X_0 \subseteq X$ of initial variables
Result: a function $X \to L$ with properties as stated in Theorem 7.11
$\mathcal{A} \leftarrow const(\boxtimes)$
foreach $x \geq t \in C$ s.t. $x \in X_0 \wedge FV(t) = \emptyset$ do
    $\mathcal{A}(x) \leftarrow \mathcal{A}(x) \sqcup [\![t]\!](\mathcal{A})$
    $W \leftarrow W \cup \{x\}$
while $W \neq \emptyset$ do
    $x \leftarrow remove(W)$
    foreach $x' \geq t \in C$ such that $x \in FV(t)$ do
        if $\forall y \in FV(t).\ y \neq x \implies \mathcal{A}(y) \neq \boxtimes$ then
            $old \leftarrow \mathcal{A}(x')$
            $\mathcal{A}(x') \leftarrow \mathcal{A}(x') \sqcup [\![t]\!](\mathcal{A})$
            if $\mathcal{A}(x') \neq old$ then
                $W \leftarrow W \cup \{x'\}$
return $\mathcal{A}$
```

For abbreviation, I define

$$C_0 \overset{def}{=} \mathcal{C}\mathrm{ore}(C, X_0)$$

$$\mathcal{V}_0 \overset{def}{=} Vars(C_0)$$

Theorem 7.11. *Algorithm 8 always terminates and upon termination, we have*

$$\mathcal{A}(x) = \begin{cases} lfp(F_{C_0})(x) & \textit{if } x \in \mathcal{V}_0 \\ \boxtimes & \textit{otherwise.} \end{cases} \tag{7.14}$$

Moreover, if C_0 is definition-complete with respect to $\mathcal{C}\mathrm{ore}(C, X)$, then

$$\mathcal{A} =_{\mathcal{V}_0} lfp(F_{\mathcal{C}\mathrm{ore}(C,X)}) = lfp(F_C) \tag{7.15}$$

Proof. First, assume that (7.14) has been shown and that C_0 is definition-complete with respect to $\mathcal{C}\mathrm{ore}(C, X)$. Then

$$\begin{aligned}
\mathcal{A} &=_{\mathcal{V}_0} lfp(F_{C_0}) && \{\ (7.14)\ \}\\
&=_{\mathcal{V}_0} lfp(F_{\mathcal{C}\mathrm{ore}(C,X)}) && \{\ \text{Theorem 7.9}\ \}\\
&= lfp(F_C). && \{\ \text{Lemma 7.4}\ \}
\end{aligned}$$

It remains to show (7.14). This follows from the following three statements:

(A) Algorithm 8 terminates.

(B) Algorithm 8 maintains the invariant

$$\{x \in X \mid \mathcal{A}(x) \neq \boxtimes\} \subseteq \mathcal{V}_0$$

and upon termination, we have

$$\{x \in X \mid \mathcal{A}(x) \neq \boxtimes\} = \mathcal{V}_0.$$

(C) Algorithm 8 maintains the invariant

$$\mathcal{A} \leq lfp(F_{C_0})$$

and upon termination, we have

$$\mathcal{A} =_{\mathcal{V}_0} lfp(F_{C_0}).$$

For a proof of (A), see Lemma 7.12. (B) is going to be shown in Lemma 7.13 and, finally, I will prove (C) in Lemma 7.14. □

The remainder of this section consists of the three statements (A), (B), and (C) that were left to show in the proof of Theorem 7.11.

Lemma 7.12. *Algorithm 8 terminates.*

Proof. It is sufficient to show that the main loop in the lines 5–13 is only traversed finitely often. For this purpose, we define $T \overset{def}{=} (X \to L) \times 2^X$. Elements of T can be used to represent the state which is maintained by the algorithm: The first component is the currently computed solution and the second component is the current worklist. Moreover, define on T the relation

$$(A_1, W_1) \leq_T (A_2, W_2) \overset{def}{\iff} A_1 >_{(X \to L)} A_2 \vee (A_1 = A_2 \wedge W_1 \subseteq W_2)$$

Then $\leq_T$ is a partial order and satisfies the descending chain condition, since $X \to L$ satisfies the ascending chain condition and 2^X is finite. Let $(\mathcal{A}, W) \in T$ be the state of the algorithm at any time.

Now consider an iteration of the main loop. Let $(\mathcal{A}_{old}, W_{old})$ and $(\mathcal{A}_{new}, W_{new})$ be the state at the beginning and end of this iteration, respectively. Now two cases are possible and we show that $(\mathcal{A}_{new}, W_{new}) <_T (\mathcal{A}_{old}, W_{old})$ must hold in either case.

First, assume that $\mathcal{A}_{old}$ has not been changed in the iteration. Then we have $\mathcal{A}_{new} = \mathcal{A}_{old}$ and the iteration has removed exactly one element of W_{old} and did not add any elements to it, i.e. $W_{new} \subsetneq W_{old}$, which means that

$$(\mathcal{A}_{new}, W_{new}) <_T (\mathcal{A}_{old}, W_{old})$$

in this case.

Now assume that $\mathcal{A}_{old}$ has indeed been touched. Then this can only have happened by executing line 10, which changes $\mathcal{A}_{old}$ upwards. Hence, we have $\mathcal{A}_{old}(x) < \mathcal{A}_{new}(x)$ for some $x \in X$. This entails

$$(\mathcal{A}_{new}, W_{new}) <_T (\mathcal{A}_{old}, W_{old}),$$

as desired.

Now we have seen that $(\mathcal{A}, W)$ becomes strictly smaller in each iteration of the main loop. Hence, since the partial order $(T, \leq_T)$ satisfies the descending chain condition, the main loop can only be traversed finitely often, as desired. □

Lemma 7.13. *1. Algorithm 8 maintains the invariant*

$$\{x \in X \mid \mathcal{A}(x) \neq \boxtimes\} \subseteq \mathcal{V}_0.$$

2. Upon termination, we have

$$\{x \in X \mid \mathcal{A}(x) \neq \boxtimes\} = \mathrm{CoreVars}(C, X_0).$$

Proof. 1. For $\psi : X \to L$ define

$$dom(\psi) \overset{def}{=} \{x \in X \mid \psi(x) \neq \boxtimes\}.$$

Then we have to show that

$$\text{(Inv)} \qquad dom(\mathcal{A}) \subseteq \mathcal{V}_0$$

holds after the execution of each statement in Algorithm 8. We only need to consider executions of line 3 or 10, since all the other statements leave $\mathcal{A}$ unchanged and therefore maintain (Inv) trivially.

So consider any execution of line 3 or 10 for a given constraint $x \geq t$. Let $\mathcal{A}_{old}$ be the value of $\mathcal{A}$ before this execution, $\mathcal{A}_{new}$ be the value of $\mathcal{A}$ after this execution.

Assume that (Inv) holds for $\mathcal{A}_{old}$.

If the considered execution does not change $\mathcal{A}$, (Inv) is maintained trivially. So assume that the considered execution indeed changes $\mathcal{A}$. In this case we have

$$dom(\mathcal{A}_{new}) = dom(\mathcal{A}_{old}) \cup \{x\},$$

i.e. we must show that $x \in \mathcal{V}_0$.

For an execution of line 3, this is indeed the case: This follows by (V-Base), because line 3 is executed only if $x \in X_0$ and $FV(t) = \emptyset$.

Next consider any execution of line 10. Then it must be the case that $FV(t)$ is not empty. Moreover, since the considered execution of line 10 changes $\mathcal{A}$, it cannot be the case that $\mathcal{A}_{old}(x) = \boxtimes$. Otherwise, $[\![t]\!](\mathcal{A}_{old})$ would be $\boxtimes$ in the case that $x \in FV(t)$, because of (7.2). Together with line 8, this ensures that $FV(t) \subseteq dom(\mathcal{A}_{old})$. By (Inv), this means that

$$\forall y \in FV(t).\ y \in \mathcal{V}_0.$$

By Lemma 7.3, this entails $x \geq t \in \mathcal{C}_0$, or, equivalently, $x \in \mathcal{V}_0$.

2. By induction on $x \in \mathcal{V}_0$, we show that for every $x \in \mathcal{V}_0$, $\mathcal{A}(x)$ is written to at some point in Algorithm 8.

(V-Base) The initial loop applies every constraint $x \geq t$ with $x \in X_0$ and no variables on the right-hand side. Since $\mathcal{A}$ is only changed upwards, and because of (7.2), this guarantees that upon termination, we have $\mathcal{A}(x) \neq \boxtimes$ for every $x \in X_0$ for which there is a constraint $x \geq t$ with $FV(t) = \emptyset$.

(V-Step) Consider a constraint $x \geq t$ such that

$$\emptyset \neq FV(t) \subseteq \mathcal{V}_0.$$

By induction hypothesis, we may assume that for all $y \in FV(t)$, $\mathcal{A}(y)$ is written to for the first time at some point in Algorithm 8. Since $FV(t) \neq \emptyset$, we may further assume that there is an iteration in which this happens for the last $y_0 \in FV(t)$. Particularly, at the beginning of this iteration, $\mathcal{A}(y) \neq \boxtimes$ for all $y \in FV(t) \setminus \{y_0\}$ and $\mathcal{A}(y_0) = \boxtimes$ and at the end of this iteration $\mathcal{A}(y) \neq \boxtimes$ for all $y \in FV(t)$. This means that $\mathcal{A}(y_0)$ is changed in this iteration and since $y_0 \in FV(t)$, x is added to W. Hence, at the end of the iteration, $x \in W$. Since Algorithm 8 terminates (Lemma 7.12), x is eventually considered in some later iteration. We may assume that $\mathcal{A}(x) = \boxtimes$ at the beginning of this iteration, since otherwise our claim follows trivially. Furthermore, we notice that

$$\forall y \in FV(t).\, \mathcal{A}(y) \neq \boxtimes$$

is maintained until Algorithm 8 terminates. So, this property holds particularly in the iteration in which x is removed from the worklist. During this iteration, $x \geq t$ is eventually considered. Again, we assume that $\mathcal{A}(x) = \boxtimes$ until $x \geq t$ is considered, since otherwise there is nothing to show. Now, when $x \geq t$ is eventually processed, the check in line 8 passes and $\mathcal{A}(x)$ is written to for the first time.

□

Lemma 7.14. *1. Algorithm 8 maintains the invariant*

$$\mathcal{A} \leq lfp(F_{C_0}).$$

2. Upon termination, we have

$$\mathcal{A} =_{\mathcal{V}_0} lfp(F_{C_0}).$$

Proof. We show the two statements separately.

1. We have to show that

$$\text{(Inv)} \qquad \mathcal{A} \leq lfp(F_{C_0})$$

is maintained by every execution of each statement in Algorithm 8. We only need to consider executions of line 3 or 10, since all the other statements leave $\mathcal{A}$ unchanged and therefore maintain (Inv) trivially.

So consider any execution of 3 or 10. Let $x \geq t$ be the constraint that is applied. First, we notice that line 3 is only executed if $x \in X_0$ and t contains no variables. Hence, $x \geq t \in C_0$. Secondly, we see that line 10 is only executed if

$$\forall y \in FV(t).\, \mathcal{A}(y) \neq \boxtimes,$$

and this implies, according to the invariant in Lemma 7.13, that $FV(t) \subseteq \mathcal{V}_0$. Hence, in both lines 3 and 10 we have $x \geq t \in C_0$ and therefore

$$(\star) \qquad lfp(F_{C_0})(x) \geq [\![t]\!](lfp(F_{C_0})).$$

Let $\mathcal{A}_{old}$ be the value of $\mathcal{A}$ before the execution and $\mathcal{A}_{new}$ be the value of $\mathcal{A}$ after the execution of line 3 or 10. Then we have

$$(\star\star) \qquad \mathcal{A}_{new}(x) = \mathcal{A}_{old}(x) \sqcup [\![t]\!](\mathcal{A}_{old}).$$

Now assume that (Inv) holds for $\mathcal{A}_{old}$. To show (Inv) for $\mathcal{A}_{new}$, we only need to consider $\mathcal{A}_{new}(x)$ since the rest is left unchanged.

Then, we can argue as follows:

$$\begin{array}{ll} \mathcal{A}_{new}(x) & \\ = \mathcal{A}_{old}(x) \sqcup [\![t]\!](\mathcal{A}_{old}) & \{\, (\star\star) \,\} \\ \leq lfp(F_{C_0})(x) \sqcup [\![t]\!](lfp(F_{C_0})) & \{\, \text{(Inv), Lemma 2.10} \,\} \\ = lfp(F_{C_0})(x) & \{\, (\star) \,\} \end{array}$$

2. We show that the main loop of Algorithm 8 maintains the invariant

$$\text{(Inv)} \qquad \begin{aligned} \forall x \geq t \in C.\;\; & x \in dom(\mathcal{A}) \\ & \wedge FV(t) \subseteq dom(\mathcal{A}) \\ & \wedge FV(t) \cap W = \emptyset \\ & \Longrightarrow \mathcal{A}(x) \geq [\![t]\!](\mathcal{A}). \end{aligned}$$

This is sufficient: Upon termination, W is empty. Hence, the following property holds:

$$\forall x \geq t \in C.\ x \in dom(\mathcal{A}) \wedge FV(t) \subseteq dom(\mathcal{A}) \implies \mathcal{A}(x) \geq [\![t]\!](\mathcal{A}).$$

Using Lemma 7.13, this is equivalent to

$$\forall x \in \mathcal{V}_0.\ \forall x \geq t \in C.\ FV(t) \subseteq \mathcal{V}_0 \implies \mathcal{A}(x) \geq [\![t]\!](\mathcal{A}). \tag{7.16}$$

Now consider any constraint $x \geq t \in \mathcal{C}\mathrm{ore}(C, X_0)$. If $x \in X_0$ and $FV(t) = \emptyset$, then (7.16) trivially implies that $\mathcal{A}$ satisfies $x \geq t$. Next, consider the case that $FV(t) \neq \emptyset$ and

$$\forall y \in FV(t).\ Def(y) \cap C_0 \neq \emptyset.$$

ByLemma 7.3 and Lemma 7.2, it follows that $FV(t) \subseteq \mathcal{V}_0$. Again, by application of (7.16) we see that $x \geq t$ is satisfied by $\mathcal{A}$.

It remains to show that (Inv) holds at the beginning of the main loop and is maintained by each of its iterations.

(Inv) **holds just before the first iteration of the main loop:** In the initialization loop, $\mathcal{A}$ was only updated for $x \in X_0$ and only for those constraints $x \geq t$ where $FV(t) = \emptyset$. Hence, if $x \geq t \in C$ with $FV(t) \cap W = \emptyset$ and $\mathcal{A}(x) \neq \boxtimes$ and $\forall y \in FV(t).\ \mathcal{A}(y) \neq \boxtimes$, it must be the case that $x \in X_0$ and $FV(t) = \emptyset$. Due to the initialization, $\mathcal{A}(x) \geq [\![t]\!](\mathcal{A})$ holds since this was ensured by the assignment in line 3 and could not be invalidated by any other execution of this line.

(Inv) **is maintained by any iteration:** Let i be some iteration of the main loop. Let $\mathcal{A}_{old}$ and W_{old} be the values of $\mathcal{A}$ and W at the beginning of this iteration and $\mathcal{A}_{new}$ and W_{new} the respective values at the end. Assume that (Inv) holds at the beginning of i:

$$\begin{aligned} \forall x \geq t \in C.\ \ & x \in dom(\mathcal{A}_{old}) \\ & \wedge FV(t) \subseteq dom(\mathcal{A}_{old}) \\ & \wedge FV(t) \cap W_{old} = \emptyset \\ & \implies \mathcal{A}_{old}(x) \geq [\![t]\!](\mathcal{A}_{old}). \end{aligned} \tag{Inv_{pre}}$$

Now consider a constraint $x \geq t \in C$ with $\mathcal{A}_{new}(x) \neq \boxtimes$ and $FV(t) \cap W_{new} = \emptyset$ and $FV(t) \subseteq dom(\mathcal{A}_{new})$. We have to show $\mathcal{A}_{new}(x) \geq [\![t]\!](\mathcal{A}_{new})$.

First, we make two important observations: Firstly, it must be the case that

$$\mathcal{A}_{new} \geq \mathcal{A}_{old}. \tag{7.17}$$

Secondly, from $FV(t) \cap W_{new} = \emptyset$, we can conclude

$$\forall y \in FV(t).\ \mathcal{A}_{new}(y) = \mathcal{A}_{old}(y). \tag{7.18}$$

Now we make a case distinction:
a) $\mathcal{A}_{old}(x) \geq [\![t]\!](\mathcal{A}_{old})$. Then we can show our claim as follows:

$$\begin{array}{ll} \ \ \mathcal{A}_{new}(x) & \\ \geq \mathcal{A}_{old}(x) & \{\ (7.17)\ \} \\ \geq [\![t]\!](\mathcal{A}_{old}) & \{\ \text{assumption}\ \} \\ = [\![t]\!](\mathcal{A}_{new}) & \{\ (7.18), \text{Lemma 2.10}\ \} \end{array}$$

b) $\mathcal{A}_{old}(x) \ngeq [\![t]\!](\mathcal{A}_{old})$. Then because of (Inv$_{pre}$), one of the following statements must be true:

$$\exists y \in FV(t).\ \mathcal{A}_{old}(y) = \boxtimes \tag{7.19}$$

$$\mathcal{A}_{old} = \boxtimes \tag{7.20}$$

$$FV(t) \cap W_{old} \neq \emptyset \tag{7.21}$$

We consider each of these cases separately.

- For (7.19), we argue as follows:

$$\begin{array}{ll} \ \ [\![t]\!](\mathcal{A}_{new}) & \\ = [\![t]\!](\mathcal{A}_{old}) & \{\ (7.18)\ \} \\ = \boxtimes & \{\ (7.2)\ \} \\ \leq \mathcal{A}_{new}(x) & \{\ (7.1)\ \} \end{array}$$

- If (7.20) holds, then line 10 must have been executed for $x \geq t$. Afterwards, $\mathcal{A}(x)$ could only have changed upwards, so we can conclude:

$$\mathcal{A}_{new}(x)$$

$$\begin{aligned}
&\geq \mathcal{A}_{old}(x) \sqcup [\![t]\!](\mathcal{A}_{old}) && \{\text{ see above }\}\\
&\geq [\![t]\!](\mathcal{A}_{old}) && \{\text{ properties of } \sqcup \}\\
&= [\![t]\!](\mathcal{A}_{new}) && \{\text{ (7.18), Lemma 2.10 }\}
\end{aligned}$$

- If (7.21) is true, then a variable x'' was removed from W and every $x' \geq t'$ with $x'' \in FV(t')$ was considered and $\mathcal{A}(x')$ was updated if needed. Since $x'' \in W_{old}$ and $FV(t) \cap W_{old} \neq \emptyset$ and $FV(t) \cap W_{new} = \emptyset$ and x'' was the only element which was removed from W, it must be $x'' \in FV(t)$. Hence, line 10 was particularly executed for $x \geq t$. Now we can argue just as in case (7.20).

□

7.2 Integration of Interprocedural Slicing and Interprocedural Data-Flow Analysis

This section gives an outline of sections 7.3 and 7.4, in which I am going to apply the results from section 7.1 to the constraint systems that I showed and discussed in chapter 6.

Section 7.3 is dedicated to the functional approach, while section 7.4 considers the call-string approach.

For my purposes, I fix a data-flow framework $\mathcal{F} = (G, L, F, \rho)$. Moreover, I fix a set $Src \subseteq N$ of *source nodes*.

The goal of the following sections is to derive algorithms that compute a VP-correct solution on the forward slice of Src. I do this by instantiating Algorithm 8 for solving relevant parts of the constraint systems that we saw in chapter 6.

To be able to handle unreachability, I adjoin $\mathcal{F}$ with an element ⊠ and extend F and $[\![.]\!]$ like outlined in section 5.3 and section 6.1, respectively. In particular, this is compatible with (7.1) and (7.2), so that I can actually apply the results from section 7.1 to the constraint systems chapter 6.

Next, I want to describe a recurring theme of the following sections. For simplicity, I exclude the call-string approach for the moment. I will handle its specifics in section 7.4.

Consider a constraint system $C_{\mathcal{P}}$ corresponding to a set of paths $\mathcal{P} \subseteq Paths_G$. Then I want to use an instantiation of Algorithm 8 to compute a solution $\mathcal{A}_{\mathcal{P}}$ along the paths from $\mathcal{P}$ such that

$$\forall (s,t) \in \mathcal{V}_0^{(\mathcal{P})} . \mathcal{A}_{\mathcal{P}}(s,t) = lfp(C_P)(s,t),$$

where

$$\mathcal{V}_0^{(\mathcal{P})} \overset{def}{=} \{(s,t) \in N \times N \mid Paths_G(s,t) \cap \mathcal{P} \neq \emptyset\}.$$

According to Theorem 7.11, Algorithm 8 computes such an $\mathcal{A}_{\mathcal{P}}$ for $\mathcal{V}_0^{(\mathcal{P})} = \mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{P}}, X_0)$, where $X_0 \subseteq X$ is chosen appropriately and $\mathcal{C}\mathrm{ore}(\mathcal{C}_{\mathcal{P}}, X_0)$ is definition-complete in $\mathcal{C}\mathrm{ore}(\mathcal{C}_{\mathcal{P}}, X)$. Hence, the general steps of the following sections will be

1. specify X_0,

2. instantiate Theorem 7.11,

3. show that

$$\mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{P}}, X_0) = \{(s,t) \in N \times N \mid Paths_G(s,t) \cap \mathcal{P} \neq \emptyset\}$$

and, finally,

4. show that $\mathcal{C}\mathrm{ore}(\mathcal{C}_{\mathcal{P}}, X_0)$ is definition-complete in $\mathcal{C}\mathrm{ore}(\mathcal{C}_{\mathcal{P}}, X)$.

7.3 Functional Approach

I want to remind the reader of Constraint System 6.4, which was defined as follows:

$$(\textsc{valid-sol})\ \frac{X_{ASC}(s,n) \neq \boxtimes \qquad X_{DESC}(n,t) \neq \boxtimes}{X_{VP}(s,t) \geq X_{DESC}(n,t) \circ X_{ASC}(s,n)}$$

Given X_{ASC} and X_{DESC}, a valid-paths solution X_{VP} could be obtained by evaluating the equation

$$X_{VP}(s,t) = \bigsqcup_{n \in N} X_{DESC}(n,t) \circ X_{ASC}(s,n). \tag{7.22}$$

One direct and naive strategy would be to first compute both X_{ASC} and X_{DESC} on the whole set $N \times N$ and then use these functions to evaluate (7.22). However, such an approach would perform a lot of unnecessary

work: In fact, one only needs to compute $X_{DESC}(n,t) \circ X_{ASC}(s,n)$ for those $s,n,t \in N$ for which $s \in Src$, $X_{ASC}(s,n) \neq \boxtimes$ and $X_{DESC}(n,t) \neq \boxtimes$.
We can reduce the amount of unnecessary work by first computing X_{ASC} on $dom(ASC)$ and then use X_{ASC} to compute an integral part of X_{VP}. Roughly, the idea is to compute a solution X_{NASC} such that X_{VP} can be written as $X_{ASC} \sqcup X_{NASC}$. For this, I need to introduce another set of paths, the *non-ascending* paths.

Definition 7.15. *$\pi \in Paths_G(s,t)$ is called* non-ascending *if π can be written as $\pi_1 \cdot \pi_2$ such that*

(i) $$\pi_1 \in ASC(s,n)$$

(ii) $$\pi_2 \in DESC(n,t)$$

(iii) $$n \in N_{call}$$

(iv) $$\pi_1 \cdot \pi_2 \notin ASC(s,t)$$

I denote the set of non-ascending paths from s to t with $NASC(s,t)$.

Definition 7.15 is motivated by the fact that any valid path is either ascending or non-ascending. This is formalized by Lemma 7.16 and Remark 7.17.

Lemma 7.16. *If $\pi \in VP(s,t)$, then either of the following is true:*

1. $\pi \in ASC(s,t)$

2. There are $c \in N_{call}$, $\pi_1 \in ASC(s,c)$ and $\pi_2 \in DESC(c,t)$ such that $\pi = \pi_1 \cdot \pi_2$

Proof. Let $\pi \in VP(s,t) \setminus ASC(s,t)$. By Theorem 5.22 there is $i \in range(\pi)$ such that $\pi^{<i} \in Left(E)$, $\pi^{\geq i} \in Right(E)$ and $\pi^i \in E_{call}$. Define

$$c \overset{def}{=} src(\pi^i),$$

$$\pi_1 \overset{def}{=} \pi^{<i}, \text{ and}$$

$$\pi_2 \overset{def}{=} \pi^{\geq i}.$$

Then $c \in N_{call}$, since π^i is an outgoing call edge of π^i. Moreover, because $\pi \in Paths_G(s,t)$, we have $\pi_1 \in Paths_G(s,c)$ and $\pi_2 \in Paths_G(c,t)$. Lastly, by Theorem 5.39, we have $\pi_1 \in Val(E)$ and $\pi_2 \in Val(E)$. In summary, π_1, π_2 and c have the desired properties. □

Remark 7.17. *$\pi \in NASC(s,t)$ if and only if $\pi \in VP(s,t) \setminus ASC(s,t)$.*

Proof. " $\Longleftarrow$ " is covered by Lemma 7.16.
So let $\pi \in NASC(s,t)$. Then $\pi \in VP(s,t)$ by Theorem 5.30 and since $\pi \in Paths_G(s,t)$. Furthermore, $\pi \notin ASC(s,t)$ by Definition 7.15. □

Lemma 7.16 can be applied as follows: Since any valid path is either ascending or non-ascending, we can obtain a valid-paths solution by joining an ascending-paths solution with a solution X_{NASC} that merges over all *non*-ascending paths.
The rest of this section is dedicated to computing a valid-paths-solution using the approach that I just sketched. First, subsection 7.3.1 considers the computation of the least same-level solution $\mathcal{A}^{(SL)}$. Then, subsection 7.3.2 shows how to use a same-level solution such as $\mathcal{A}^{(SL)}$ to compute the least ascending-paths solution $\mathcal{A}^{(ASC)}$. After that, subsection 7.3.3 is dedicated to the extension of a given ascending-paths solution like $\mathcal{A}^{(ASC)}$ along the descending paths to obtain the least non-ascending-paths-solution: First, a constraint system is given that characterizes non-ascending-paths solutions and then the usual scheme is used to compute the least non-ascending-paths solution $\mathcal{A}^{(NASC)}$.
The last three subsections consider the combination of $\mathcal{A}^{(ASC)}$ and $\mathcal{A}^{(NASC)}$. Subsection 7.3.4 shows that joining $\mathcal{A}^{(ASC)}$ and $\mathcal{A}^{(NASC)}$ actually yields a valid-paths-correct solution, provided that $\mathcal{A}^{(ASC)}$ and $\mathcal{A}^{(NASC)}$ are correct relative to their respective path sets. Moreover, it compares the solution obtained by this approach with the original least valid-paths solution. After that, subsection 7.3.5 integrates all the sections before in a simple algorithm and shows its correctness. This algorithm uses the same ideas as the two-phase slicer by Horwitz et al. that I already discussed in chapter 3, however there are still some differences. Subsection 7.3.6 explores these differences, modifies the algorithm from subsection 7.3.5 in such a way that it essentially becomes the two-phase slicer and sketches a correctness proof.

7.3.1 Computing the Same-Level Solution

In the following, I consider the constraint system from Constraint System 6.1, which I denote with $C^{(SL)}$.

I define

$$X_0^{(SL)} \overset{def}{=} \{(s,s) \mid s \in N_{entry}\} \tag{7.23}$$

$$\mathcal{C}_0^{(SL)} \overset{def}{=} \mathcal{C}\mathrm{ore}(\mathcal{C}^{(SL)}, X_0^{(SL)}) \tag{7.24}$$

$$\mathcal{V}_0^{(SL)} \overset{def}{=} \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(SL)}, X_0^{(SL)}) \tag{7.25}$$

Algorithm 9 is an instantiation of Algorithm 8 for the constraint system $\mathcal{C}^{(SL)}$, using $X_0^{(SL)}$ as set of initial variables.
The loop in lines 2–4 corresponds to the initialization loop in Algorithm 8. It processes all constant constraints in Constraint System 6.1 whose left-hand side is in $X_0^{(SL)}$.
The loop in lines 5–18 corresponds to the main loop Algorithm 8. All constraints $x \geq u$ with $(s,t) \in FV(u)$ are enumerated and the respective $\mathcal{A}^{(SL)}(x)$ is updated, just like in Algorithm 8. However, the constraint enumeration loop in Algorithm 9 is split into three parts.
The first of these parts, in lines 7–9, enumerates all constraints of the form SL-SOL-(II).
The other two parts are dedicated to constraint SL-SOL-(III). This is because Algorithm 9 propagates from the right-hand side of a constraint to its left-hand side. In contrast to SL-SOL-(II), there are *two* free variables on the right-hand side of SL-SOL-(III)-constraints, so that (s,t) can occur at two positions. To illustrate this, let us take a look at such a constraint. It has the form

$$X_{SL}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}} \circ X_{SL}(s,n)$$

with $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$ and $(e_{call}, e_{ret}) \in \Phi$.
Hence, each of the two variables on the right-hand side of a rule have to be considered separately.
The loop in lines 10–13 takes care of the case that the variable currently processed is (s,n) in the above situation, whereas the loop in lines 15–18 takes care of the other case: Here, the variable currently processed is (n_0,n_1) in the above situation.
In summary, the two loops enumerate all constraints c of the form SL-SOL-(III) with $(s,t) \in FV(rhs(c))$.

Algorithm 9: Algorithm for computing the least same-level solution

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295
Result: least same-level solution for $\mathcal{F}$, as stated in Theorem 7.18
$\mathcal{A}^{(SL)} \leftarrow const(\boxtimes)$
foreach $s \in N_{entry}$ do
    $\mathcal{A}^{(SL)}(s,s) \leftarrow id$
    $W = W \cup \{(s,s)\}$
while $W \neq \emptyset$ do
    $(s,t) \leftarrow remove(W)$
    foreach $t' \in N \; s.t. \; t \xrightarrow{e} t' \wedge e \in E_{intra}$ do
        $\mathcal{A}^{(SL)}(s,t') \leftarrow \mathcal{A}^{(SL)}(s,t') \sqcup f_e \circ \mathcal{A}^{(SL)}(s,t)$
        $W \leftarrow W \cup \{(s,t')\}$ if $\mathcal{A}^{(SL)}(s,t')$ has changed
    foreach $(e_c, e_r) \in \Phi \; s.t. \; t \xrightarrow{e_c} n_0 \wedge n_1 \xrightarrow{e_r} t' \wedge \mathcal{A}^{(SL)}(n_0,n_1) \neq \boxtimes$ do
        $slSol \leftarrow f_{e_r} \circ \mathcal{A}^{(SL)}(n_0,n_1) \circ f_{e_c}$
        $\mathcal{A}^{(SL)}(s,t') \leftarrow \mathcal{A}^{(SL)}(s,t') \sqcup s \circ \mathcal{A}^{(SL)}(s,t)$
        $W \leftarrow W \cup \{(s,t')\}$ if $\mathcal{A}^{(SL)}(s,t')$ has changed
    foreach $(e_c, e_r) \in \Phi \; s.t. \; a \xrightarrow{e_c} s \wedge t \xrightarrow{e_r} b$ do
        foreach $u \in N_{entry} \; s.t. \; \mathcal{A}^{(SL)}(u,a) \neq \boxtimes$ do
            $slSol \leftarrow f_{e_r} \circ \mathcal{A}^{(SL)}(s,t) \circ f_{e_c}$
            $\mathcal{A}^{(SL)}(u,b) \leftarrow \mathcal{A}^{(SL)}(u,b) \sqcup slSol \circ \mathcal{A}^{(SL)}(u,a)$
            $W \leftarrow W \cup \{(u,b)\}$ if $\mathcal{A}^{(SL)}(u,b)$ has changed
return $\mathcal{A}^{(SL)}$
```

Now that the reader is convinced that Algorithm 9 is an instantiation of Algorithm 8, I want to instantiate the correctness result of Algorithm 8 for Algorithm 9.

For the moment, I assume that $C_0^{(SL)}$ is definition-complete with respect to $C^{(SL)}$. Then I can use Theorem 7.11 to prove the following correctness result.

Theorem 7.18. *The following statements hold:*

1. *Algorithm 9 always terminates and upon termination, we have*

$$\mathcal{A}^{(SL)}(s,t) = \begin{cases} lfp(F_{C^{(SL)}})(s,t) & \text{if } (s,t) \in \mathcal{V}_0^{(SL)} \\ \boxtimes & \text{otherwise} \end{cases} \tag{7.26}$$

2. *Upon termination, $\mathcal{A}^{(SL)}$ is always $(SL, \mathcal{V}_0^{(SL)})$-correct.*

3. *If $\mathcal{F}$ is u.d., then $\mathcal{A}^{(SL)}$ is $(SL, \mathcal{V}_0^{(SL)})$-precise.*

Two things are left to do. Firstly, we have to characterize $\mathcal{V}_0^{(SL)}$ appropriately and secondly, we have to show that $C_0^{(SL)}$ is indeed definition-complete with respect to $C^{(SL)}$.

We start with the characterization of $\mathcal{V}_0^{(SL)}$. Ideally, considering the fact that $C^{(SL)}$ is defined along the same-level paths of G, same-level reachability should be the right property to characterize $C_0^{(SL)}$. Moreover, taking $X_0^{(SL)}$ into account, we can only expect $(s,t) \in \mathcal{V}_0^{(SL)}$ if $s \in N_{entry}$.

Lemma 7.19 formally confirms that $\mathcal{V}_0^{(SL)}$ indeed has the desired characterization.

Lemma 7.19. $\mathcal{V}_0^{(SL)}$ *and* $\mathcal{C}\text{oreVars}(C, N \times N)$ *can be characterized as follows:*

$$\mathcal{V}_0^{(SL)} = \{(s,t) \in N \times N \mid SL(s,t) \neq \emptyset\} \cap N_{entry} \times N \tag{7.27}$$

$$\mathcal{C}\text{oreVars}(C, N \times N) = \{(s,t) \in N \times N \mid SL(s,t) \neq \emptyset\} \tag{7.28}$$

In particular, $\mathcal{V}_0^{(SL)}$ *can be obtained from* $\mathcal{C}\text{oreVars}(C, N \times N)$ *by restricting the first components to* N_{entry}*:*

$$\mathcal{V}_0^{(SL)} = \mathcal{C}\text{oreVars}(C, N \times N) \cap N_{entry} \times N \tag{7.29}$$

Proof. The claim (7.29) follows directly from (7.27).

It remains to show (7.27) and (7.28). We only show (7.27), since the proof for (7.28) is very similar.

We prove (7.27) by showing the two subset relations separately.

1. In order to prove "$\supseteq$", we show

$$\forall \pi \in SL. \forall (s,t) \in N_{entry} \times N.\ \pi \in SL(s,t) \\ \implies (s,t) \in \mathcal{V}_0^{(SL)}$$

by induction on $\pi \in SL$. So let $\pi \in SL$ and $(s,t) \in N_{entry} \times N$ such that $\pi \in SL(s,t)$.

a) Suppose that $\pi = \epsilon$. Then we have $s = t$ and $s \in N_{entry}$, so that $(s,t) = (s,s) \in X_0^{(SL)}$. By SL-SOL-(I), $\mathcal{C}^{(SL)}$ contains the constraint

$$X(s,s) \geq id,$$

which does not have variables on the right-hand side. Hence, $(s,s) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(SL)}, X_0^{(SL)})$ by (V-BASE).

b) Assume that $\pi = \pi' \cdot e$, where $e \in E_{intra}$, $t' \xrightarrow{e} t$ and $\pi' \in SL(s,t')$. Since $\pi' \in SL(s,t')$, we can apply the induction hypothesis to π': $s \in N_{entry}$ implies that

$$(s,t') \in \mathcal{V}_0^{(SL)}.$$

By SL-SOL-(II), $\mathcal{C}^{(SL)}$ contains the constraint

$$X(s,t) \geq f_e \circ X(s,t').$$

Since $(s,t') \in \mathcal{V}_0^{(SL)}$ we may apply (V-STEP) and conclude

$$(s,t) \in \mathcal{V}_0^{(SL)}.$$

c) Suppose that

$$\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$$

with $\pi' \in SL(s,n)$, $\pi'' \in SL(n_0,n_1)$, $n \xrightarrow{e_{call}} n_0$, $n_1 \xrightarrow{e_{ret}} t$ and $(e_{call}, e_{ret}) \in \Phi$. We apply the induction hypothesis to $\pi' \in SL(s,n)$ and conclude from $s \in N_{entry}$ that

$$(s,n) \in \mathcal{V}_0^{(SL)}. \tag{7.30}$$

Moreover, note that n_0 has an incoming call edge. Hence, we have $n_0 \in N_{entry}$. With $\pi'' \in SL(n_0, n_1)$ we can apply the induction hypothesis to π'' and obtain

$$(n_0, n_1) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(SL)}, X_0^{(SL)}). \tag{7.31}$$

Moreover, by SL-SOL-(III), $\mathcal{C}^{(SL)}$ contains the constraint

$$X(s,t) \geq f_{e_{ret}} \circ X(n_0, n_1) \circ f_{e_{call}} \circ X(s,n).$$

With (V-STEP), we conclude $(s,t) \in \mathcal{V}_0^{(SL)}$ from (7.30) and (7.31).

2. For "$\subseteq$", we show

$$(s,t) \in \mathcal{V}_0^{(SL)} \implies SL(s,t) \neq \emptyset \land s \in N_{entry}$$

by induction over $\mathcal{V}_0^{(SL)} = \mathcal{C}\text{oreVars}(\mathcal{C}^{(SL)}, X_0^{(SL)})$.

Let $(s,t) \in \mathcal{V}_0^{(SL)}$. Let $c \in \mathcal{C}_0^{(SL)}$ with $lhs(c) = (s,t)$. Our induction hypothesis states

$$\forall (x,y) \in FV(rhs(c)).\ SL(x,y) \neq \emptyset \land s \in N_{entry}$$

a) Suppose that $FV(rhs(c)) = \emptyset$. Then c must be of the form

$$X_{SL}(s,s) \geq id$$

Clearly, $SL(s,s) \neq \emptyset$, since $\epsilon \in SL(s,s)$.

b) Suppose that $FV(rhs(c)) \neq \emptyset$. Then c is either of the form SL-SOL-(II) or SL-SOL-(III). We only consider SL-SOL-(III), since SL-SOL-(II) is similar.

So assume that c has the form

$$X_{SL}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}} \circ X_{SL}(s,n)$$

such that $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$ and $(e_{call}, e_{ret}) \in \Phi$. From $c \in \mathcal{C}_0^{(SL)}$, we conclude $(s,n) \in \mathcal{V}_0^{(SL)}$ and $(n_0, n_1) \in \mathcal{V}_0^{(SL)}$. Hence, we can apply the induction hypothesis to (s, n) and (n_0, n_1) and yield $s \in N_{entry}$, $\pi' \in SL(s, n)$ and $\pi'' \in SL(n_0, n_1)$. These two paths can be used to obtain

$$\pi \overset{def}{=} \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret} \in SL(s, t),$$

as desired.

□

In order to complete the proof of Theorem 7.18, we show the definition-completeness of $\mathcal{C}_0^{(SL)}$ with respect to $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(SL)}, N \times N)$.

Lemma 7.20. *$\mathcal{C}_0^{(SL)}$ is definition-complete with respect to its superset*

$$\mathcal{C}\mathrm{ore}(\mathcal{C}^{(SL)}, N \times N).$$

Proof. We use Theorem 7.10. Let $c \in \mathcal{C}\mathrm{ore}(\mathcal{C}^{(SL)}, N \times N)$ with $lhs(c) \in \mathcal{V}_0^{(SL)}$. Then we have to show

$$FV(rhs(c)) = \emptyset \implies lhs(c) \in X_0^{(SL)} \tag{7.32}$$

$$FV(rhs(c)) \neq \emptyset \land x \in FV(rhs(c)) \implies x \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(SL)}, X_0^{(SL)}) \tag{7.33}$$

- For (7.32), assume that $FV(rhs(c)) = \emptyset$. Then c is of the form

  $$X_{SL}(s, s) \geq id$$

  From $lhs(c) = (s, s) \in \mathcal{V}_0^{(SL)}$ we get $s \in N_{entry}$ by (7.27). Hence, $(s, s) \in X_0^{(SL)}$.

- For (7.33), assume that $FV(rhs(c)) \neq \emptyset$. Then there are two possible shapes of c.

  1. Let c be of the form SL-SOL-(III), i.e.

     $$X_{SL}(s, t) \geq f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}} \circ X_{SL}(s, n)$$

     with $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$ and $(e_{call}, e_{ret}) \in \Phi$.

From $lhs(c) = (s,t) \in \mathcal{V}_0^{(SL)}$, we conclude $s \in N_{entry}$ by (7.27). Since $c \in \mathcal{C}_0^{(SL)}$, we have

$$(\star) \qquad X_{SL}(s,n) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(SL)}, N \times N), \text{ and}$$

$$(\star\star) \qquad X_{SL}(n_0,n_1) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(SL)}, N \times N).$$

We apply (7.29) and conclude from $s \in N_{entry}$, $(\star)$ and $(\star\star)$ that $X_{SL}(s,n) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(SL)}, X_0^{(SL)})$. Moreover, note that $n_0 \in N_{entry}$, since it has an incoming call edge. Hence, $X_{SL}(n_0,n_1) \in \mathcal{V}_0^{(SL)}$ by (7.29).

With (C-Step), it follows from $(\star)$ and $(\star\star)$ that $c \in \mathcal{C}_0^{(SL)}$, and, with (V-Step), $(s,t) \in \mathcal{V}_0^{(SL)}$. This concludes the proof of (7.33) for c.

2. The argument for the form sl-sol-(ii) is very similar.

□

7.3.2 Computing the Ascending Solution

In the following, I consider the constraint system from Constraint System 6.2, which I denote with $\mathcal{C}^{(ASC)}$.
I define

$$X_0^{(ASC)} \overset{def}{=} \{(s,s) \mid s \in Src\} \tag{7.34}$$

$$\mathcal{C}_0^{(ASC)} \overset{def}{=} \mathcal{C}\text{ore}(\mathcal{C}^{(ASC)}, X_0^{(ASC)}) \tag{7.35}$$

$$\mathcal{V}_0^{(ASC)} \overset{def}{=} \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, X_0^{(ASC)}) \tag{7.36}$$

Algorithm 10 is a straight-forward instantiation of Algorithm 8 to solve Constraint System 6.2 with respect to a function

$$X_{SL} : N \times N \to F_{\boxtimes}.$$

This function could have been computed by Algorithm 9, but can also have been obtained in any other way.

Theorem 7.21 gives a correctness result for Algorithm 10. Similar to Theorem 7.18, the first item in Theorem 7.21 directly follows from the generic result Theorem 7.11, once I have shown that $\mathcal{C}_0^{(ASC)}$ is definition-complete with respect to $\mathcal{C}^{(ASC)}$. However, this first item only is concerned with the relation between the result of Theorem 7.18 and the least solution of $\mathcal{C}^{(ASC)}$. In order to get full correctness and precision results, I need additional requirements for X_{SL}, relative to a set $\Psi \subseteq N \times N$.

In principle, there are multiple choices for Ψ. I just need to ensure that (a) Ψ is large enough so that $\mathcal{C}^{(ASC)}$ and Algorithm 10 never access X_{SL} outside of Ψ and that (b) the output of Algorithm 9 is indeed (SL, Ψ)-correct. The following choice of Ψ has both properties:

$$(7.37) \quad (n_0, n_1) \in \Psi \overset{def}{\Leftrightarrow} \quad (n_0, n_1) \in N_{entry} \times N_{exit} \\ \wedge \exists n, t \in N.\ \exists (e_{call}, e_{ret}) \in \Phi.\ n \overset{e_{call}}{\to} n_0 \wedge n_1 \overset{e_{ret}}{\to} t.$$

Property (a) can be seen by inspection of $\mathcal{C}^{(ASC)}$ and Algorithm 10. Moreover, Ψ also has property (b). (SL, Ψ)-correctness of $\mathcal{A}^{(SL)}$ follows from the $(SL, \mathcal{V}_0^{(SL)})$-correctness of $\mathcal{A}^{(SL)}$ and the fact that $\Psi \subseteq N_{entry} \times N$, which entails

$$\Psi \cap \{(s,t) \mid SL(s,t) \neq \emptyset\} \subseteq N_{entry} \times N \cap \{(s,t) \mid SL(s,t) \neq \emptyset\} \overset{(7.27)}{=} \mathcal{V}_0^{(SL)}.$$

Thus, if $(s,t) \in \Psi$ with $\pi \in SL(s,t)$, then $(s,t) \in \mathcal{V}_0^{(SL)}$ and hence, by Theorem 7.18, $f_\pi \leq \mathcal{A}^{(SL)}(s,t)$.

A similar argument shows that $(SL, \mathcal{V}_0^{(SL)})$-precision of $\mathcal{A}^{(SL)}$ implies (SL, Ψ)-precision of $\mathcal{A}^{(SL)}$ for universally distributive frameworks.

Using an appropriate choice of Ψ and assuming that $\mathcal{C}_0^{(ASC)}$ is definition-complete with respect to $\mathcal{C}^{(ASC)}$, I can state the correctness property of Theorem 7.21.

Theorem 7.21. *Let $X_{SL} : N \times N \to F_{\boxtimes}$ be a function and let $\mathcal{C}^{(ASC)}$ the set of constraints in Constraint System 6.2 with respect to X_{SL}. Consider X_{SL} as input for Algorithm 10.*

1. *Algorithm 10 terminates and upon termination, we have*

$$\mathcal{A}(s,t) = \begin{cases} lfp(F_{C^{(ASC)}})(s,t) & \text{if } (s,t) \in \mathcal{V}_0^{(ASC)} \\ \boxtimes & \text{otherwise} \end{cases} \tag{7.38}$$

2. *If X_{SL} is (SL,Υ)-correct, then $\mathcal{A}$ is $(ASC,\mathcal{V}_0^{(ASC)})$-correct.*

3. *If X_{SL} is (SL,Υ)-precise and $\mathcal{F}$ is universally distributive, then $\mathcal{A}$ is $(ASC,\mathcal{V}_0^{(ASC)})$-precise.*

In the following, I am going to prove Theorem 7.21. Firstly, I assume that item 1 is proven and consider items 2 and 3. In order to prove item 2, I need to generalize Theorem 6.7: Remember that Theorem 6.7 states that $lfp(F_{C^{(ASC)}})$ is ASC-correct if X_{SL} is SL-correct. But an inspection of its proof shows that actually (SL,Υ)-correctness of X_{SL} is sufficient for the ASC-correctness of $lfp(F_{C^{(ASC)}})$. Hence, the proof of Theorem 6.7 actually shows the following statement:

If X_{SL} is (SL,Υ)-correct, then $lfp(F_{C^{(ASC)}})$ is $(ASC,\mathcal{V}_0^{(ASC)})$-correct.

With (7.38), this shows item 2. With a similar argument, item 3 can be shown.

Two things are left to be done. In order to complete the proof of Theorem 7.21, I need to show that $C_0^{(ASC)}$ is definition-complete with respect to $C^{(ASC)}$. Moreover, I need to characterize $\mathcal{V}_0^{(ASC)}$ appropriately, to be sure that Theorem 7.21 indeed proves Algorithm 10 correct.

Lemma 7.22 gives a characterization of CoreVars$(C^{(ASC)}, N \times N)$ and $\mathcal{V}_0^{(ASC)}$. Analogously to Lemma 7.19, it provides a strong connection to ASC-reachability. Since $C^{(ASC)}$ is defined with respect to X_{SL}, Lemma 7.22 has to make additional assumptions about X_{SL}. However, for Lemma 7.22 to be valid, full SL-correctness or SL-precision of X_{SL} is not neccessary. It suffices to require that $X_{SL} \neq \boxtimes$ for the right points.

Lemma 7.22. *Define $dom(ASC) \overset{def}{=} \{(s,t) \in N \times N \mid ASC(s,t) \neq \emptyset\}$. Then the following statements are true:*

1. *If X_{SL} is (SL,Υ)-domain-correct, then we have*

$$dom(ASC) \cap Src \times N \subseteq \mathcal{V}_0^{(ASC)} \tag{7.39}$$

Algorithm 10: Given a function $X_{SL} : N \times N \to F_{\boxtimes}$, computes the least ascending solution with respect to X_{SL}

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295, a function $X_{SL} : N \times N \to F_{\boxtimes}$
Result: the least ascending solution with respect to $X_{SL}$, as stated in Theorem 7.21
$\mathcal{A} \leftarrow const(\boxtimes)$
foreach $s \in Src$ do
    $\mathcal{A}(s,s) \leftarrow id$
    $W \leftarrow W \cup \{(s,s)\}$
while $W \neq \emptyset$ do
    $(s,t') \leftarrow remove(W)$
    foreach $e \in E_{intra} \cup E_{ret}$ such that $t' \xrightarrow{e} t$ do
        $\mathcal{A}(s,t) \leftarrow \mathcal{A}(s,t) \sqcup f_e \circ \mathcal{A}(s,t')$
        $W \leftarrow W \cup \{(s,t)\}$ if $\mathcal{A}(s,t)$ has changed
    foreach $(e_{call}, e_{ret}) \in \Phi$ such that $t' \xrightarrow{e_{call}} n_0 \wedge n_1 \xrightarrow{e_{ret}} t \wedge$
      $X_{SL}(n_0,n_1) \neq \boxtimes$ do
        $sameLevelInfo \leftarrow f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}}$
        $\mathcal{A}(s,t) \leftarrow \mathcal{A}(s,t) \sqcup sameLevelInfo \circ \mathcal{A}(s,t')$
        $W \leftarrow W \cup \{(s,t')\}$ if $\mathcal{A}(s,t)$ has changed
return $\mathcal{A}$
```

$$dom(ASC) \subseteq \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, N \times N) \tag{7.40}$$

2. *If X_{SL} is (SL, Ψ)-domain-precise, then we have*

$$\mathcal{V}_0^{(ASC)} \subseteq dom(ASC) \cap Src \times N \tag{7.41}$$

$$\mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, N \times N) \subseteq dom(ASC) \tag{7.42}$$

3. $\mathcal{V}_0^{(ASC)}$ *and* $\mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, N \times N)$ *have the following connection:*

$$\mathcal{V}_0^{(ASC)} = \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, N \times N) \cap Src \times N. \tag{7.43}$$

Proof. Regarding the first two items, we only show (7.39) and (7.41), since (7.40) and (7.42) can be proven using very similar arguments.

1. Assume that X_{SL} is (SL, Ψ)-domain-correct. Then (7.39) is implied by the statement

$$\forall \pi \in ASC.\, \forall (s,t) \in Src \times N.\, \pi \in ASC(s,t) \implies (s,t) \in \mathcal{V}_0^{(ASC)}. \tag{7.44}$$

We prove this statement by induction on $\pi \in ASC$. The proof is largely similar to the corresponding part of the proof of Lemma 7.19. We only highlight the parts where the assumptions about X_{SL} are used. Hence, we only consider the case that $\pi = \pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret}$ with $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$, $\pi' \in ASC(s,n)$, $\pi'' \in SL(n_0, n_1)$ and $(e_{call}, e_{ret}) \in \Phi$. Note that $(n_0, n_1) \in \Psi$. With the (SL, Ψ)-domain-correctness of X_{SL} this means that $X_{SL}(n_0, n_1) \neq \boxtimes$. Hence, by ASC-SOL-(III), $\mathcal{C}^{(ASC)}$ contains a constraint

$$X(s,t) \geq f_{e_{ret}} \circ X(n_0, n_1) \circ f_{e_{call}} \circ X(s,n).$$

By induction hypothesis, we get $(s,n) \in \mathcal{V}_0^{(ASC)}$. We conclude

$$X(s,t) \in \mathcal{V}_0^{(ASC)}$$

by (V-BASE).

2. Assume that X_{SL} is (SL, Ψ)-domain-precise. In order to show (7.42), we prove

$$\forall (s,t) \in \mathcal{V}_0^{(ASC)}.\, ASC(s,t) \neq \emptyset \tag{7.45}$$

by induction on $(s,t) \in \mathcal{V}_0^{(ASC)} = \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, X_0^{(ASC)})$. So let $(s,t) \in \mathcal{V}_0^{(ASC)}$. Then there is $c \in \mathcal{C}_0^{(ASC)}$ with $lhs(c) = (s,t)$. Because the other cases are similar to the proof of Lemma 7.19, we only consider the case that c is of the form

$$X_{ASC}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}} \circ X_{ASC}(s,n)$$

with $(e_{call}, e_{ret}) \in \Phi$, $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$ and $X_{SL}(n_0, n_1) \neq \boxtimes$. Then we have $(n_0, n_1) \in \Psi$. Since X_{SL} is (SL, Ψ)-domain-precise, $X_{SL}(n_0, n_1) \neq \boxtimes$ implies that there is $\pi'' \in SL(n_0, n_1)$. Furthermore, we may apply the induction hypothesis to (s,n) and get $\pi' \in ASC(s,n)$. Together with the assumptions about e_{call} and e_{ret}, we get $\pi' \cdot e_{call} \cdot \pi'' \cdot e_{ret} \in ASC(s,t)$, as desired.

3. In order to show (7.43), we consider the two subset relations separately.
   - For $\subseteq$, it is clear that

     $$\mathcal{V}_0^{(ASC)} \subseteq \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(ASC)}, N \times N),$$

     since $Src \times N \subseteq N \times N$ and $\mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(ASC)}, \_)$ is monotone.
     It remains to show

     $$\forall (s,t) \in \mathcal{V}_0^{(ASC)}.(s,t) \in Src \times N.$$

     The proof is a straight-forward induction along $\mathcal{V}_0^{(ASC)}$, where the induction steps work by noticing that the first components of the variables on the left-hand and right-hand sides of constraints in $\mathcal{V}_0^{(ASC)}$ are the same.
   - For $\supseteq$, induction on $(s,t) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(ASC)}, N \times N)$ shows that

     $$\forall (s,t) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(ASC)}, N \times N).\ (s,t) \in Src \times N \\ \implies (s,t) \in \mathcal{V}_0^{(ASC)}.$$

□

With Lemma 7.22, I can provide the last puzzle piece to the proof of Theorem 7.21: The definition-completeness of $\mathcal{C}_0^{(ASC)}$ with respect to $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(ASC)}, N \times N)$.

Lemma 7.23. $\mathcal{C}_0^{(ASC)}$ *is definition-complete with respect to its superset*

$$\mathcal{C}\mathrm{ore}(\mathcal{C}^{(ASC)}, N \times N).$$

Proof. Like in the proof of Lemma 7.20, we use Theorem 7.10. Consider $c \in \mathcal{C}\mathrm{ore}(\mathcal{C}^{(ASC)}, N \times N)$ with $(s,t) = lhs(c) \in \mathcal{V}_0^{(ASC)}$. Then we have to show

$$FV(rhs(c)) = \emptyset \implies (s,t) \in X_0^{(ASC)} \tag{7.46}$$

$$(s',t') \in FV(rhs(c)) \implies (s',t') \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(ASC)}, X_0^{(ASC)}) \tag{7.47}$$

As the other cases are very similar, we only consider the case that $c \in \mathcal{C}\text{ore}(\mathcal{C}^{(ASC)}, N \times N)$ is of the form

$$X_{ASC}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}} \circ X_{ASC}(s,n)$$

with $(e_{call}, e_{ret}) \in \Phi$, $n \overset{e_{call}}{\to} n_0$, $n_1 \overset{e_{ret}}{\to} t$, $X_{SL}(n_0, n_1) \neq \boxtimes$, and $(s,t) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, X_0^{(ASC)})$. We need to show

$$(s,n) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, X_0^{(ASC)}).$$

But note that

(1) $\qquad (s,n) \in \mathcal{C}\text{oreVars}(\mathcal{C}^{(ASC)}, N \times N)$, and

(2) $\qquad s \in Src$.

Statement (1) is implied by $c \in \mathcal{C}\text{ore}(\mathcal{C}^{(ASC)}, N \times N)$, while statement (2) follows, by application of Lemma 7.22, from $(s,t) \in \mathcal{V}_0^{(ASC)}$. Both statements together imply $(s,n) \in \mathcal{V}_0^{(ASC)}$ by Lemma 7.22. □

7.3.3 Extending the Solution Along the Non-Ascending Paths

Constraint System 7.1 is a modified version of Constraint System 6.3 that describes the data flows along the non-ascending paths.
The constant constraints of Constraint System 7.1 use the values $X_{ASC}(s,c)$ with $X_{ASC}(s,c) \neq \boxtimes$ to initialize the solution. Note that c can be restricted to N_{call} according to Lemma 7.16. The non-constant constraints then extend the solution along the descending paths.

Constraint System 7.1. *Given functions $X_{ASC}, X_{SL} : N \times N \to F_{\boxtimes}$, the function*

$$X_{NASC} : N \times N \to F_{\boxtimes}$$

is a non-ascending solution *with respect to X_{ASC} and X_{SL} if it satisfies all constraints from the following system:*

$$(\textsc{non-asc-(i)})\ \frac{t' \xrightarrow{e} t \qquad e \in E_{call} \qquad X_{ASC}(s,t') \neq \boxtimes}{X_{NASC}(s,t) \geq f_e \circ X_{ASC}(s,t')}$$

$$(\textsc{non-asc-(ii)})\ \frac{t' \xrightarrow{e} t \qquad e \in E_{intra} \cup E_{call}}{X_{NASC}(s,t) \geq f_e \circ X_{NASC}(s,t')}$$

$$(\textsc{non-asc-(iii)})\ \frac{n \xrightarrow{e_{call}} n_0 \ \ n_1 \xrightarrow{e_{ret}} t \ \ (e_{call}, e_{ret}) \in \Phi \ \ X_{SL}(n_0,n_1) \neq \boxtimes}{X_{NASC}(s,t) \geq f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}} \circ X_{NASC}(s,n)}$$

Using a similar argument as in Theorem 6.7, I can prove that Constraint System 7.1 indeed describes *NASC*-correct solutions, provided that X_{SL} and X_{ASC} enjoy their respective correctness properties.

Theorem 7.24. *Let $X_{SL} : N \times N \to F_{\boxtimes}$ be SL-correct, $X_{ASC} : N \times N \to F_{\boxtimes}$ be ASC-correct and let X_{NASC} be a NASC-solution with respect to X_{SL} and X_{ASC}. Then X_{NASC} is NASC-correct.*

Proof. Let $s \in N$, $c \in N_{call}$ and $\pi_1 \in ASC(s,c)$. Using a straight-forward induction along $\pi_2 \in DESC$ with arguments similar to the ones in the proof of, e.g., Theorem 6.6, we show

$$\begin{aligned}&\forall \pi_2 \in DESC.\ \forall t \in N.\\ &\qquad \pi_2 \in DESC(c,t) \land \pi_1 \cdot \pi_2 \notin ASC(s,t)\\ &\qquad \implies f_{\pi_1 \cdot \pi_2} \leq X_{NASC}(s,t).\end{aligned}$$

Using Definition 7.15, this shows that

$$\forall \pi \in NASC.\ \forall s,t \in N. \pi \in NASC(s,t) \implies f_\pi \leq X_{NASC}(s,t),$$

as desired. □

Moreover, for universally distributive frameworks and with the respective precision requirements satisfied for X_{ASC} and X_{SL}, the least solution of Constraint System 7.1 coincides with *MONASC*. The proof is very similar to the proof of Theorem 6.10 and is not repeated here.

Algorithm 11: computes the least non-ascending solution on $\mathcal{V}_0^{(NASC)}$ with respect to X_{ASC} and X_{SL}

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_\boxtimes, \rho)$ as described on page 295, functions $X_{SL}, X_{ASC} : N \times N \to F_\boxtimes$
Result: the least non-ascending solution with respect to $X_{SL}$, as stated in Theorem 7.26
$\mathcal{A} \leftarrow const(\boxtimes)$
foreach $(s,c) \in Src \times N$ with $X_{ASC}(s,c) \neq \boxtimes$ do
    foreach $e \in E_{call}$, $t \in N$ such that $c \xrightarrow{e} t$ do
        $\mathcal{A}(s,t) \leftarrow \mathcal{A}(s,t) \sqcup f_e \circ X_{ASC}(s,c)$
        $W \leftarrow W \cup \{(s,t)\}$
while $W \neq \emptyset$ do
    $(s,t) \leftarrow remove(W)$
    foreach $e \in E_{intra} \cup E_{call}$ such that $t \xrightarrow{e} t'$ do
        $\mathcal{A}(s,t') \leftarrow \mathcal{A}(s,t') \sqcup f_e \circ \mathcal{A}(s,t)$
        $W \leftarrow W \cup \{(s,t')\}$ if $\mathcal{A}(s,t')$ has changed
    foreach $(e_{call}, e_{ret}) \in \Phi$ such that $t \xrightarrow{e_{call}} n_0 \wedge n_1 \xrightarrow{e_{ret}} t' \wedge X_{SL}(n_0,n_1) \neq \boxtimes$ do
        $sumInfo \leftarrow f_{e_{ret}} \circ X_{SL}(n_0,n_1) \circ f_{e_{call}}$
        $\mathcal{A}(s,t') \leftarrow \mathcal{A}(s,t') \sqcup sumInfo \circ \mathcal{A}(s,t)$
        $W \leftarrow W \cup \{(s,t')\}$ if $\mathcal{A}(s,t')$ has changed
return $\mathcal{A}$
```

Theorem 7.25. *Let $\mathcal{F}$ be universally distributive. Furthermore, let X_{SL} be SL-precise and X_{ASC} be ASC-precise. Then* MO*NASC is a NASC-solution with respect to X_{SL} and X_{ASC}.*
In particular, $lfp(F_{C_{NASC}})$ is NASC-precise, if C_{NASC} is defined with respect to X_{ASC} and X_{SL}.

All that remains to do is to instantiate Algorithm 8 appropriately and prove the usual correctness result. The instantiation can be seen in Algorithm 11. I fix two functions

$$X_{ASC}, X_{SL} : N \times N \to F_\boxtimes,$$

and let $\mathcal{C}^{(NASC)}$ be the set of constraints defined by Constraint System 7.1, with respect to X_{SL} and X_{ASC}.
I define

$$X_0^{(NASC)} \stackrel{def}{=} \{(s,c) \mid s \in Src \wedge c \in N_{call} \wedge X_{ASC}(s,c) \neq \boxtimes\} \tag{7.48}$$

$$\mathcal{C}_0^{(NASC)} \stackrel{def}{=} \mathcal{C}\text{ore}(\mathcal{C}^{(NASC)}, X_0^{(ASC)}) \tag{7.49}$$

$$\mathcal{V}_0^{(NASC)} \stackrel{def}{=} \mathcal{C}\text{oreVars}(\mathcal{C}^{(NASC)}, X_0^{(ASC)}) \tag{7.50}$$

Like before, an inspection of Constraint System 7.1 shows that Algorithm 11 is indeed an instance of Algorithm 8. Thus, assuming definition-completeness of $\mathcal{C}_0^{(NASC)}$ in $\mathcal{C}\text{ore}(\mathcal{C}^{(NASC)}, N \times N)$, I can give the following correctness result for Algorithm 11.

Theorem 7.26. *Let $X_{SL}, X_{ASC} : N \times N \to F_{\boxtimes}$ be two functions and let C_{NASC} be the instance of Constraint System 7.1 with respect to X_{SL} and X_{ASC}. Consider X_{SL} and X_{ASC} as input for Algorithm 11.*

1. Algorithm 11 terminates and upon termination, we have

$$\mathcal{A}(s,t) = \begin{cases} lfp(F_{C_{NASC}})(s,t) & \text{if } s \in \mathcal{V}_0^{(NASC)} \\ \boxtimes & \text{otherwise.} \end{cases} \tag{7.51}$$

2. If X_{SL} is (SL, Ψ)-correct and X_{ASC} is $(ASC, Src \times N)$-correct, then $\mathcal{A}$ is $(NASC, \mathcal{V}_0^{(NASC)})$-correct.

3. If $\mathcal{F}_{\boxtimes}$ is u.d., X_{SL} is (SL, Ψ)-precise, and X_{ASC} is $(ASC, Src \times N)$-precise, then $\mathcal{A}$ is $(NASC, \mathcal{V}_0^{(NASC)})$-precise.

In the following, I sketch a proof for Theorem 7.26. First, assume that the first item is shown. Then the second item follows if I can show that $lfp(F_{\mathcal{C}^{(NASC)}})$ is $(NASC, \mathcal{V}_0^{(NASC)})$-correct. But this can be concluded from the (SL, Ψ)-correctness of X_{SL} and the $(ASC, Src \times N)$-correctness of X_{ASC} with a similar argument as in the proof of Constraint System 7.1.
The third item in Theorem 7.26 can be shown similarly by adapting Theorem 7.25 appropriately.

It remains to characterize the variable$\mathcal{V}_0^{(NASC)}$ and to prove the definition-completeness of $\mathcal{C}_0^{(NASC)}$ in $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(NASC)}, N \times N)$. Lemma 7.27 gives a characterization of $\mathcal{V}_0^{(NASC)}$ that is analogous to Lemma 7.22.

Lemma 7.27. *Define* $dom(NASC) \overset{def}{=} \{(s,t) \in N \times N \mid NASC(s,t) \neq \emptyset\}$*. Then the following statements are true:*

1. If X_{SL} is (SL, Ψ)-domain-correct and X_{ASC} is $(ASC, Src \times N)$-domain-correct, then we have

$$dom(NASC) \cap Src \times N \subseteq \mathcal{V}_0^{(NASC)} \tag{7.52}$$

$$dom(NASC) \subseteq \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(NASC)}, N \times N) \tag{7.53}$$

2. If X_{SL} is (SL, Ψ)-domain-precise and X_{ASC} is $(ASC, Src \times N)$-domain-precise, then we have

$$\mathcal{V}_0^{(NASC)} \subseteq dom(NASC) \cap Src \times N \tag{7.54}$$

$$\mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(NASC)}, N \times N) \subseteq dom(NASC) \tag{7.55}$$

3. $\mathcal{V}_0^{(NASC)}$ and $\mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(NASC)}, N \times N)$ have the following connection:

$$\mathcal{V}_0^{(NASC)} = \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(NASC)}, N \times N) \cap Src \times N. \tag{7.56}$$

Proof. 1. We only consider (7.52), since the proof of (7.53) is very similar. Assume that X_{SL} is (SL, Ψ)-domain-correct and that X_{ASC} is $(ASC, Src \times N)$-domain-correct. Let $s, c \in N$ and $\pi_1 \in ASC(s,c)$. Then, we can show

$$\begin{aligned} \forall \pi_2 \in DESC.\ \forall t \in N.\ \pi_2 \in DESC(c,t) \wedge \pi_1 \cdot \pi_2 \notin ASC(s,t) \\ \implies (s,t) \in \mathcal{V}_0^{(NASC)} \end{aligned}$$

by induction on $\pi_2 \in DESC$. The details are very similar to the corresponding part of the proof of Lemma 7.22 and are omitted here. By Definition 7.15, this implies (7.52).

2. Assume that X_{SL} is (SL, Ψ)-domain-precise and, moreover, that X_{ASC} is $(ASC, Src \times N)$-domain-precise. Then we can show

$$\forall (s,t) \in \mathcal{V}_0^{(NASC)}.\ (s,t) \in dom(NASC) \cap Src \times N$$

by induction on $(s,t) \in \mathcal{V}_0^{(NASC)}$. For each case, the argument uses a combination of the respective induction hypothesis and the assumptions about X_{SL} and X_{ASC} to obtain an appropriate non-ascending path. The details are very similar to Lemma 7.22 and I do not repeat them here. The proof of (7.54) is similar.

3. This is completely analogous to the proof of (7.43).

□

Using Lemma 7.27, the definition-completeness of $\mathcal{C}_0^{(NASC)}$ with respect to its superset $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(NASC)}, N \times N)$ can be shown. The proof is completely analogous to the proofs of Lemma 7.23 and Lemma 7.20 and is omitted here.

Lemma 7.28. $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(ASC)}, X_0)$ *is definition-complete with respect to its superset* $\mathcal{C}\mathrm{ore}(\mathcal{C}_{NASC}, N \times N)$.

7.3.4 Combining the Ascending Solution and the Non-Ascending Solution

Constraint System 7.2 combines two given functions X_{ASC} and X_{NASC} by simply joining them.

Constraint System 7.2. *Given functions* $X_{ASC}, X_{NASC} : N \times N \to F_{\boxtimes}$, *the function*

$$X_{VP'} : N \times N \to F_{\boxtimes}$$

is an alternative valid-paths-solution *with respect to* X_{ASC} *and* X_{NASC} *if it satisfies all constraints from the following system:*

$$(\text{VP}'\text{-(I)})\ \frac{X_{ASC}(s,t) \neq \boxtimes}{X_{VP'}(s,t) \geq X_{ASC}(s,t)} \qquad (\text{VP}'\text{-(II)})\ \frac{X_{NASC}(s,t) \neq \boxtimes}{X_{VP'}(s,t) \geq X_{NASC}(s,t)}$$

Constraint System 7.2 is so simple that I can give a clear and direct characterization of $lfp(F_{C(VP')})$, which is easy to see:

$$lfp(F_{C(VP')}) = X_{ASC} \sqcup X_{NASC}. \tag{7.57}$$

Alternative valid-paths solutions are indeed VP-correct, if X_{ASC} and X_{DESC} satisfy their respective correctness conditions. This is an easy consequence of Remark 7.17, which is why I omit the proof.

Theorem 7.29. *Let $X_{ASC}, X_{NASC} : N \times N \rightarrow F_{\boxtimes}$ be two functions. Then the following statements hold:*

1. Let $X_{VP'}$ be an alternative valid-paths solution with respect to X_{ASC} and X_{NASC}. If X_{ASC} is ASC-correct and X_{NASC} is NASC-correct, then $X_{VP'}$ is VP-correct.

2. If X_{ASC} is ASC-precise and X_{NASC} is NASC-precise, then $lfp(F_{C_{VP'}})$ is VP-precise.

In chapter 6, we considered Constraint System 6.4 as a characterization for valid-paths solutions. For the following considerations, let $C^{(VP)}$ be the set of constraints from Constraint System 6.4.

Note that Theorem 7.29 does not say anything about the relation of $lfp(F_{C(VP)})$ and $lfp(F_{C(VP')})$ in general, even if they are considered with respect to the same helper solutions. As $C^{(VP)}$ and $C^{(VP')}$ make different choices about when they join and when they compose, we cannot expect in general that they coincide[36]. In general, $C^{(VP)}$ and $C^{(VP')}$ can simply be seen as alternative approaches to characterize an over-approximation to $MOVP$. This is especially relevant if $\mathcal{F}$ is not distributive and the helper solutions are correct and as precise as possible (e.g. if they were obtained using the algorithms from the previous sections), but also if the helper solutions themselves are only over-approximations, even for the distributive case.

[36] My conjecture is (a) that $lfp(F_{C(VP)}) \leq lfp(F_{C(VP')})$ under relatively general assumptions and (b) that, under the same assumptions, there are examples for which $lfp(F_{C(VP')}) \not\leq lfp(F_{C(VP)})$, since $C^{(VP)}$ "joins later" than $C^{(VP')}$. I will however not attempt to prove this within the scope of this thesis.

For distributive frameworks and the most precise helper solutions, $lfp(F_{C^{(VP')}})$ can be shown to coincide with $lfp(F_{C^{(VP)}})$.

Corollary 7.30. *Assume that $\mathcal{F}$ is universally-distributive. Consider the following constraint systems:*

$$\begin{aligned}
C^{(ASC)} &\textit{ with respect to } lfp(F_{C^{(SL)}})\\
C^{(DESC)} &\textit{ with respect to } lfp(F_{C^{(SL)}})\\
C^{(NASC)} &\textit{ with respect to } lfp(F_{C^{(SL)}}) \textit{ and } lfp(F_{C^{(ASC)}})\\
C^{(VP)} &\textit{ with respect to } lfp(F_{C^{(ASC)}}) \textit{ and } lfp(F_{C^{(DESC)}}), \textit{ and}\\
C^{(VP')} &\textit{ with respect to } lfp(F_{C^{(ASC)}}) \textit{ and } lfp(F_{C^{(NASC)}}).
\end{aligned}$$

Then $lfp(F_{C^{(VP)}}) = lfp(F_{C^{(VP')}})$.

Proof. By Theorem 6.7 and Theorem 6.10, $lfp(F_{C^{(ASC)}})$ is both ASC-correct and ASC-precise. Moreover, Theorem 7.24 and Theorem 7.25 imply that $lfp(F_{C^{(NASC)}})$ is both $NASC$-correct and $NASC$-precise. Hence, Theorem 7.29 implies that $lfp(F_{C^{(VP')}})$ is both VP-correct and VP-precise, i.e.

$$lfp(F_{C^{(VP')}}) = MOVP.$$

Analogously, using Theorem 6.7, Theorem 6.8, Theorem 6.10 and Theorem 6.9, one can conclude

$$lfp(F_{C^{(VP)}}) = MOVP.$$

□

Lastly, I want to consider the connection between the domain of $lfp(F_{C_{VP'}})$ and forward slices. Remember that the forward slice $FS(s)$ of a given node $s \in N$ is the set of nodes $t \in N$ such that $VP(s,t) \neq \emptyset$.

Theorem 7.31. *Let $X_{ASC}, X_{NASC} : N \times N \to F_{\boxtimes}$ be two functions. Then the following statements hold:*

1. Let $X_{VP'}$ be an alternative valid-paths solution with respect to X_{ASC} and X_{NASC}. If X_{ASC} is ASC-domain-correct and X_{NASC} is NASC-domain-correct, then $X_{VP'}$ is VP-domain-correct. In particular, we have

$$\forall s \in N.\, FS(s) \subseteq \{t \in N \mid lfp(F_{C_{VP'}})(s,t) \neq \boxtimes\}. \tag{7.58}$$

2. If X_{ASC} is ASC-domain-precise and X_{NASC} is NASC-domain-precise, then $lfp(F_{C_{VP'}})$ is VP-domain-precise. In particular, we have

$$\forall s \in N.\, \{t \in N \mid lfp(F_{C_{VP'}})(s,t) \neq \boxtimes\} \subseteq FS(s). \tag{7.59}$$

Proof. For given $s \in N$, we define

$$\begin{aligned} FS_{ASC}(s) &\overset{def}{=} \{t \in N \mid ASC(s,t) \neq \emptyset\}, \text{ and} \\ FS_{NASC}(s) &\overset{def}{=} \{t \in N \mid NASC(s,t) \neq \emptyset\}. \end{aligned}$$

By Lemma 7.16, we have $FS(s) = FS_{ASC}(s) \cup FS_{NASC}(s)$ for all $s \in N$. Moreover, by definition we have

$$\begin{aligned} FS_{ASC}(s) &\subseteq dom(X_{ASC}) && \text{if } X_{ASC} \text{ is } ASC\text{-domain-correct,} \\ FS_{ASC}(s) &\supseteq dom(X_{ASC}) && \text{if } X_{ASC} \text{ is } ASC\text{-domain-precise,} \\ FS_{NASC}(s) &\subseteq dom(X_{NASC}) && \text{if } X_{NASC} \text{ is } NASC\text{-domain-correct, and} \\ FS_{NASC}(s) &\supseteq dom(X_{NASC}) && \text{if } X_{NASC} \text{ is } NASC\text{-domain-precise.} \end{aligned}$$

From this, all claimed statements can be shown using (7.57). □

7.3.5 Putting It All Together

In this section, I combine the algorithms from the previous sections to yield algorithms that compute the least alternative valid-paths solution with the most precise helper solutions.

First, I show a simple algorithm that is always correct and then I consider a variant that works for distributive frameworks.

The simple approach, which can be seen in Algorithm 12, takes the set $Src \subseteq N$ of source nodes as input and uses the pre-computed result $\mathcal{A}^{(SL)}$ of running Algorithm 9 on $X_0^{(SL)}$.

As its first step, Algorithm 12 invokes Algorithm 10 to compute the least ascending-paths solution $\mathcal{A}^{(ASC)}$ with respect to $\mathcal{A}^{(SL)}$. After that, it runs Algorithm 11 to obtain the least non-ascending solution $\mathcal{A}^{(NASC)}$ with respect to $\mathcal{A}^{(SL)}$ and $\mathcal{A}^{(ASC)}$. Finally, it joins $\mathcal{A}^{(ASC)}$ and $\mathcal{A}^{(NASC)}$ to obtain its result $\mathcal{A}^{(VP')}$.
For Algorithm 12 I can give the following simple correctness result.

Theorem 7.32. *Consider*

$$\begin{aligned} C^{(VP')} &\text{ with respect to } lfp(C^{(ASC)}) \text{ and } lfp(C^{(NASC)}) \\ C^{(NASC)} &\text{ with respect to } lfp(C^{(ASC)}) \text{ and } lfp(C^{(SL)}), \text{ and} \\ lfp(C^{(ASC)}) &\text{ with respect to } lfp(C^{(SL)}). \end{aligned}$$

Then Algorithm 12 has the following properties:

1. *Algorithm 12 always terminates and upon termination, we have*

$$\mathcal{A}^{(VP')}(s,t) = \begin{cases} lfp(F_{C^{(VP')}})(s,t) & \text{if } t \in \mathcal{V}_0^{(ASC)} \cup \mathcal{V}_0^{(NASC)} \\ \boxtimes & \text{otherwise} \end{cases} \tag{7.60}$$

2. *Define* $F \overset{def}{=} \bigcup_{s \in Src} FS(s)$. *If* $\mathcal{F}$ *is universally distributive, then*

$$\mathcal{A}^{(VP')} =_F \text{MO}VP. \tag{7.61}$$

Algorithm 12: Computes the least alternative valid-paths solution on $Src \times N$ with respect to the most precise helper functions

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295, least same-level solution $\mathcal{A}^{(SL)}$, node set $Src \subseteq N$
Result: least alternative valid-paths solution, as stated in Theorem 7.32
$\mathcal{A}^{(ASC)} \leftarrow ComputeAscendingSolution(Src, \mathcal{A}^{(SL)})$
$\mathcal{A}^{(NASC)} \leftarrow ComputeNonAscendingSolution(\mathcal{A}^{(SL)}, \mathcal{A}^{(ASC)})$
$\mathcal{A}^{(VP')} \leftarrow \mathcal{A}^{(ASC)} \sqcup \mathcal{A}^{(DESC)}$
return $\mathcal{A}^{(VP')}$
```

Proof. 1. (7.60) is an easy consequence of (7.57), considering the final value of $\mathcal{A}^{(VP')}$ in Algorithm 12.

2. Assume that $\mathcal{F}$ is universally distributive. Then (7.61) is an easy consequence of

$$lfp(F_{C^{(VP')}}) = MOVP, \text{ and} \tag{7.62}$$

$$\mathcal{V}_0^{(ASC)} \cup \mathcal{V}_0^{(NASC)} = \bigcup_{s \in Src} FS(s). \tag{7.63}$$

These two facts follow from the universal distributivity of $\mathcal{F}$, Theorem 7.29 and Theorem 7.31.

□

7.3.6 Towards the Two-Phase Slicer

Remember that Algorithm 6 works in two phases: The first visits all nodes that are *ASC*-reachable from the given start nodes. It uses summary edges to skip call edges and instead collects the call nodes on a second worklist W_2. The second phase starts with W_2 and extends the set of visited nodes along the descending paths of the given graph, using summary edges to skip return edges.

I can employ this pattern in my more general setting. The result is Algorithm 13, a variant of Algorithm 12 that integrates the two main steps of Algorithm 12 and offers large similarities to Algorithm 6. This variant also computes the alternative valid-paths solution on the forward slice of *Src*, provided that the given framework $\mathcal{F}$ is distributive.

Like Algorithm 12, Algorithm 13 proceeds in two steps.

The first step, which is implemented by Algorithm 14, is a variant of Algorithm 10 that computes the least ascending solution $\mathcal{A}^{(ASC)}$ with respect to the pre-computed least same-level solution and additionally collects all $(s, c) \in Src \times N_{call}$ such that $ASC(s, c) \neq \emptyset$ in a set W_2, which it also returns. It can easily be verified that Algorithm 14 has the same properties as Algorithm 10 but additionally has the property that, upon termination, W_2 contains the set of *all* $(s, c) \in Src \times N$ such that $c \in N_{call}$ and $ASC(s, c) \neq \emptyset$.

The second step, which is implemented by Algorithm 15, then takes $\mathcal{A}^{(ASC)}$ and W_2 and executes a variant of Algorithm 11. Algorithm 15 differs from

Algorithm 13: Computes the least alternative valid-paths solution on $Src \times N$ with respect to the most precise helper functions

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295, least same-level solution $\mathcal{A}^{(SL)}$, node set $Src \subseteq N$
Result: least alternative valid-paths solution
$(\mathcal{A}^{(ASC)}, W_2) \leftarrow ComputeAscendingSolution'(Src, \mathcal{A}^{(SL)})$
$\mathcal{A}^{(VP')} \leftarrow ExtendAlongNonAscSolution'(\mathcal{A}^{(SL)}, \mathcal{A}^{(ASC)}, W_2)$
return $\mathcal{A}^{(VP')}$
```

Algorithm 11 in two key aspects: Firstly, it does not initialize the solution with $const(\boxtimes)$, but rather initializes it with $\mathcal{A}^{(ASC)}$ and secondly, it does not have an initial loop. However, it also starts with a non-empty worklist that, by the additional correctness property of Algorithm 15, contains all $(s, c) \in Src \times N$ such that $n \in N_{call}$ and $ASC(s, c) \neq \boxtimes$. It can be verified that the initial loop is actually integrated in the main loop.

Assuming the distributivity of $\mathcal{F}$, we can show that Algorithm 15 indeed computes $lfp(C_{VP'})$. The proof is analogous to the proof of Theorem 7.11 and proceeds in three steps, which I state here but omit the proofs.

1. Algorithm 15 always terminates.

2. Algorithm 15 maintains the invariant

$$\{(s, t) \in Src \times N \mid \mathcal{A}^{(VP')}(s, t) \neq \boxtimes\} \subseteq \mathcal{V}_0^{(ASC)} \cup \mathcal{V}_0^{(NASC)}$$

and upon termination, we have

$$\{(s, t) \in Src \times N \mid \mathcal{A}^{(VP')}(s, t) \neq \boxtimes\} = \mathcal{V}_0^{(ASC)} \cup \mathcal{V}_0^{(NASC)}$$

3. Algorithm 15 maintains the invariant

$$\mathcal{A}^{(VP')} \leq lfp(F_{C^{(ASC)}}) \sqcup lfp(F_{C^{(NASC)}})$$

and upon termination, we have

$$\mathcal{A}^{(VP')} = lfp(F_{C^{(ASC)}}) \sqcup lfp(F_{C^{(NASC)}}) = lfp(F_{C^{(VP')}}).$$

The invariant in the last step actually uses the distributivity of $\mathcal{F}$. My conjecture, which I will not prove within the scope of this thesis is that (a) Algorithm 13 always computes an over-approximation $lfp(C_{VP'})$ and that (b) there are non-distributive frameworks for which $\mathcal{A}^{(VP')} > lfp(C_{VP'})$.

7.4 Call-String Approach

This section is dedicated to computing a VP-correct solution using a call-string-based constraint system. The general pattern is the same as in the

Algorithm 14: Implementation of ComputeAscendingSolution′

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_\boxtimes, \rho)$ as described on page 295, least same-level solution $\mathcal{A}^{(SL)}$, node set $Src \subseteq N$
Result: Ascending solution $\mathcal{A}$
set $W_2 \subseteq Src \times N$ such that $(s, n) \in W_2$ implies that $n \in N_{call}$ and $n$ is $ASC$-reachable from $Src$
$\mathcal{A} \leftarrow const(\boxtimes)$
foreach $s \in Src$ do
    $\mathcal{A}(s, s) \leftarrow id$
    $W \leftarrow W \cup \{(s, s)\}$
while $W \neq \emptyset$ do
    $(s, t') \leftarrow remove(W)$
    foreach $e \in E_{intra} \cup E_{ret}$ such that $t' \xrightarrow{e} t$ do
        $\mathcal{A}(s, t) \leftarrow \mathcal{A}(s, t) \sqcup f_e \circ \mathcal{A}(s, t')$
        $W \leftarrow W \cup \{(s, t)\}$ if $\mathcal{A}(s, t)$ has changed
    foreach $(e_{call}, e_{ret}) \in \Phi$ such that $t' \overset{e_{call}}{\to} n_0 \wedge n_1 \overset{e_{ret}}{\to} t \wedge X_{SL}(n_0, n_1) \neq \boxtimes$ do
        $sumInfo \leftarrow f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}}$
        $\mathcal{A}(s, t) \leftarrow \mathcal{A}(s, t) \sqcup sumInfo \circ \mathcal{A}(s, t')$
        $W_2 \leftarrow W_2 \cup \{(s, t')\}$
        $W \leftarrow W \cup \{(s, t')\}$ if $\mathcal{A}(s, t)$ has changed
return $\mathcal{A}$
```

Algorithm 15: Implementation of ExtendAlongNonAscSolution′

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as
       described on page 295, least same-level solution $\mathcal{A}^{(SL)}$,
       node set $Src \subseteq N$, ascending solution $\mathcal{A}^{(ASC)}$, set $W$ of call
       nodes that are $ASC$-reachable from $Src$
Result: alternative valid-paths solution $\mathcal{A}^{(VP')}$
$\mathcal{A}^{(VP')} \leftarrow \mathcal{A}^{(ASC)}$
while $W \neq \emptyset$ do
    $(s, t) \leftarrow remove(W)$
    foreach $e \in E_{intra} \cup E_{call}$ such that $t \xrightarrow{e} t'$ do
        $\mathcal{A}^{(VP')}(s, t') \leftarrow \mathcal{A}^{(VP')}(s, t') \sqcup f_e \circ \mathcal{A}^{(VP')}(s, t)$
        $W \leftarrow W \cup \{(s, t')\}$ if $\mathcal{A}^{(VP')}(s, t')$ has changed
    foreach $(e_{call}, e_{ret}) \in \Phi$ such that $t \xrightarrow{e_{call}} n_0 \wedge n_1 \xrightarrow{e_{ret}} t' \wedge$
      $X_{SL}(n_0, n_1) \neq \boxtimes$ do
        $sumInfo \leftarrow f_{e_{ret}} \circ X_{SL}(n_0, n_1) \circ f_{e_{call}}$
        $\mathcal{A}^{(VP')}(s, t') \leftarrow \mathcal{A}^{(VP')}(s, t') \sqcup sumInfo \circ \mathcal{A}^{(VP')}(s, t)$
        $W \leftarrow W \cup \{(s, t')\}$ if $\mathcal{A}^{(VP')}(s, t')$ has changed
return $\mathcal{A}^{(VP')}$
```

last sections. First, I am going to instantiate Algorithm 8 so that it solves a sufficient portion of Constraint System 6.5. After that, I will state an appropriate instance of Theorem 7.11. Lastly, I give a characterize the core variables of the subsystem solved by the algorithm and convince myself that the algorithm indeed solves a definition-complete subsystem.

I fix a stack space $\mathcal{S} = (E_{call}, S, \leq, \epsilon, push, pop, top)$ such that S is finite. The finiteness of S ensures that the complete lattice

$$N \times N \times S \rightarrow_{mon} F_{\boxtimes}$$

satisfies the ascending chain condition.

Before I am able to instantiate Algorithm 8 to compute the least solution of Constraint System 6.5, I need to modify Constraint System 6.5 a bit: In Algorithm 8, constraints are applied by updating the left-hand sides for given updated right-hand sides. However, Constraint System 6.5 does

not present all constraints in such a form. Particularly, $\text{RET}^{(2)}_{\mathcal{S}}$ needs to be transformed.
The $\text{RET}^{(2)}_{\mathcal{S}}$-constraints have the form

$$(\text{RET}^{(2)}_{\mathcal{S}})\frac{t' \overset{e_{ret}}{\to} t \quad (e_{call}, e_{ret}) \in \Phi \quad pop(push(e_{call}, \sigma)) = \sigma \quad push(e_{call}, \sigma) \neq \epsilon}{X_{\mathcal{S}}(s, t, \sigma) \geq f_{e_{ret}} \circ X_{\mathcal{S}}(s, t', push(e_{call}, \sigma))}$$

I fix $e_{call} \in E_{call}$ and consider the following two sets:

$$\begin{aligned} A = \{(\sigma, push(e_{call}, \sigma)) \mid \ & \sigma \in S \\ & \land pop(push(e_{call}, \sigma)) = \sigma \\ & \land push(e_{call}, \sigma) \neq \epsilon\} \\ B = \{(pop(\sigma), \sigma) \mid & \ \sigma \in S \land \sigma \neq \epsilon \land top(\sigma) = e_{call}\} \end{aligned}$$

The elements of A describe the relation between the two stacks appearing on the left-hand side and the right-hand side of a $\text{RET}^{(2)}_{\mathcal{S}}$-rule involving e_{call}, whereas the elements of B describe this relation for a respective transformed rule.
In the following, I am going to show that $A = B$ by proving that they are contained in each other.
So let $(\sigma, push(e_{call}, \sigma)) \in A$. Then we have

$$pop(push(e_{call}, \sigma)) = \sigma \qquad \text{and} \qquad push(e_{call}, \sigma) \neq \epsilon.$$

Define $\tilde{\sigma} \overset{def}{=} push(e_{call}, \sigma)$. Then $\tilde{\sigma} \neq \epsilon$, $top(\tilde{\sigma}) = e_{call}$ and

$$pop(\tilde{\sigma}) = pop(push(e_{call}, \sigma)) = \sigma.$$

This means that

$$(\sigma, push(e_{call}, \sigma)) = (pop(\tilde{\sigma}), \tilde{\sigma}) \in B.$$

Conversely, let $(pop(\sigma), \sigma) \in B$. Then $\sigma \neq \epsilon$ and $top(\sigma) = e_{call}$. Define $\tilde{\sigma} \overset{def}{=} pop(\sigma)$. Then

$$push(e_{call}, \tilde{\sigma}) = push(e_{call}, pop(\sigma)) = \sigma \neq \epsilon$$

and

$$pop(push(e_{call}, \tilde{\sigma})) = pop(\sigma) = \tilde{\sigma}.$$

This means that

$$(pop(\sigma), \sigma) = (\tilde{\sigma}, push(e_{call}, \tilde{\sigma})) \in A.$$

The equality of A and B implies that the $\textsc{ret}^{(2)}_{\mathcal{S}}$-constraints from Constraint System 6.5 can be replaced by the following equivalent $\textsc{ret}'^{(2)}_{\mathcal{S}}$-constraints:

$$(\textsc{ret}'^{(2)}_{\mathcal{S}})\,\frac{e_{ret} \in E_{ret} \quad t' \overset{e_{ret}}{\to} t \quad (e_{call}, e_{ret}) \in \Phi \quad \sigma \neq \epsilon \quad top(\sigma) = e_{call}}{X_{\mathcal{S}}(s, t, pop(\sigma)) \geq f_{e_{ret}} \circ X_{\mathcal{S}}(s, t', \sigma)}$$

The replacement of $\textsc{ret}^{(2)}_{\mathcal{S}}$ by $\textsc{ret}'^{(2)}_{\mathcal{S}}$ in Constraint System 6.5 leads to Constraint System 7.3.

Constraint System 7.3. *Let $\mathcal{S} = (E_{call}, S, \leq, \epsilon, push, pop, top)$ be a stack space over E_{call}. Then $X_{\mathcal{S}} : N \times N \times S \to_{mon} F$ is an $\mathcal{S}$-solution if it satisfies the following constraints:*

$$(\textsc{empty}_{\mathcal{S}})\,\frac{s \in N}{X_{\mathcal{S}}(s, s, \epsilon) \geq id}\,(\textsc{intra}_{\mathcal{S}})\,\frac{e \in E_{intra} \qquad t' \overset{e}{\to} t}{X_{\mathcal{S}}(s, t, \sigma) \geq f_e \circ X_{\mathcal{S}}(s, t', \sigma)}$$

$$(\textsc{call}_{\mathcal{S}})\,\frac{e_{call} \in E_{call} \qquad t' \overset{e_{call}}{\to} t}{X_{\mathcal{S}}(s, t, push(e_{call}, \sigma')) \geq f_{e_{call}} \circ X_{\mathcal{S}}(s, t', \sigma')}$$

$$(\textsc{ret}^{(1)}_{\mathcal{S}})\,\frac{e_{ret} \in E_{ret} \qquad t' \overset{e_{ret}}{\to} t}{X_{\mathcal{S}}(s, t, \epsilon) \geq f_{e_{ret}} \circ X_{\mathcal{S}}(s, t', \epsilon)}$$

$$(\textsc{ret}'^{(2)}_{\mathcal{S}})\,\frac{e_{ret} \in E_{ret} \quad t' \overset{e_{ret}}{\to} t \quad (e_{call}, e_{ret}) \in \Phi \quad \sigma \neq \epsilon \quad top(\sigma) = e_{call}}{X_{\mathcal{S}}(s, t, pop(\sigma)) \geq f_{e_{ret}} \circ X_{\mathcal{S}}(s, t', \sigma)}$$

Lemma 7.33. *Constraint System 6.5 and Constraint System 7.3 are equivalent: X is a solution of Constraint System 6.5 if and only if it is a solution of Constraint System 7.3.*

Proof. See the previous considerations. □

Now let $\mathcal{C}^{(\mathcal{S})}$ be the set of constraints from Constraint System 7.3 with respect to the abstract stack space $\mathcal{S}$. Furthermore, I define

$$X_0^{(\mathcal{S})} \overset{def}{=} \{(s, s, \epsilon) \mid s \in Src\} \tag{7.64}$$

$$\mathcal{C}_0^{(\mathcal{S})} \overset{def}{=} \mathcal{C}\mathrm{ore}(\mathcal{C}^{(\mathcal{S})}, X_0^{(\mathcal{S})}) \tag{7.65}$$

$$\mathcal{V}_0^{(\mathcal{S})} \overset{def}{=} \mathcal{C}\mathrm{oreVars}(\mathcal{C}^{(\mathcal{S})}, X_0^{(\mathcal{S})}) \tag{7.66}$$

Algorithm 16 is an instantiation of Algorithm 8 that is supposed to solve $\mathcal{C}_0^{(\mathcal{S})}$. The initialization loop can be seen in lines 2–4 and the main loop in lines 5–11. The body of the main loop enumerates all constraints c with $(s, t, \sigma) \in FV(rhs(c))$ by inspecting the stack σ and the outgoing edge e and determining which constraint to apply.

Assuming that $\mathcal{C}_0^{(\mathcal{S})}$ is definition-complete in $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(\mathcal{S})}, N \times N \times S)$, we get the following correctness result for Algorithm 16.

Theorem 7.34. *Algorithm 16 always terminates and upon termination, we have*

$$\mathcal{A}_{\mathcal{S}}(s, t, \sigma) = \begin{cases} lfp(F_{\mathcal{C}_0^{(\mathcal{S})}})(s, t, \sigma) & \textit{if } (s, t, \sigma) \in \mathcal{V}_0^{(\mathcal{S})} \\ \boxtimes & \textit{otherwise} \end{cases} \tag{7.67}$$

Moreover, we have

$$\mathcal{A}_{\mathcal{S}} =_{\mathcal{V}_0^{(\mathcal{S})}} lfp(F_{\mathcal{C}\mathrm{ore}(\mathcal{C}^{(\mathcal{S})}, N \times N \times S)}) = lfp(F_{\mathcal{C}^{(\mathcal{S})}}) \tag{7.68}$$

To fully establish the correctness and meaningfulness of Theorem 7.34, it remains to show that $\mathcal{C}_0^{(\mathcal{S})}$ is definition-complete in $\mathcal{C}\mathrm{ore}(\mathcal{C}^{(\mathcal{S})}, N \times N \times S)$ and give an appropriate characterization of $\mathcal{V}_0^{(\mathcal{S})}$. I start with the latter. The following lemma does not fully solve this task, but at least shows that $\mathcal{V}_0^{(\mathcal{S})}$ is large enough[37].

[37] My conjecture is that an appropriate converse can also be shown, once a suitable formalization of reachability on $N \times N \times S$ is available. I will however not consider this within the scope of this thesis.

Algorithm 16: Algorithm for computing the least $\mathcal{S}$-solution

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295, set $Src \subseteq N$ of sources
Result: the least $\mathcal{S}$-solution, as stated in Theorem 7.34
$\mathcal{A}_{\mathcal{S}} \leftarrow const(\boxtimes)$
foreach $s \in Src$ do
    $\mathcal{A}_{\mathcal{S}}(s, s, \epsilon) \leftarrow id$
    $W \leftarrow \{(s, s, \epsilon)\}$
while $W \neq \emptyset$ do
    $(s, t, \sigma) \leftarrow remove(W)$
    foreach $e \in E$ such that $t \xrightarrow{e} t'$ do
        $\sigma' \leftarrow \begin{cases} \sigma & \text{if } e \in E_{intra} \lor e \in E_{ret} \land \sigma = \epsilon \\ push(e, \sigma) & \text{if } e \in E_{call} \\ pop(\sigma) & \text{if } \sigma \neq \epsilon \land (top(\sigma), e) \in \Phi \\ \blacktriangledown & \text{otherwise} \end{cases}$
        if $\sigma' \neq \blacktriangledown$ then
            $\mathcal{A}_{\mathcal{S}}(s, t', \sigma') \leftarrow \mathcal{A}_{\mathcal{S}}(s, t', \sigma') \sqcup f_e \circ \mathcal{A}_{\mathcal{S}}(s, t, \sigma)$
            $W \leftarrow W \cup \{(s, t', \sigma')\}$ if $\mathcal{A}_{\mathcal{S}}(s, t', \sigma')$ has changed
return $\mathcal{A}_{\mathcal{S}}$
```

Lemma 7.35. *For all $s, t \in N$, we have*

$$AP_{\mathcal{S}}(s, t) \neq \emptyset \land s \in Src \implies \exists \sigma \in S.\ (s, t, \sigma) \in CoreVars(C_{\mathcal{S}}, X_0)$$

Proof. We show the statement

$$\forall \pi \in Paths_G.\ \forall s, t \in N.\ \pi \in Paths_G(s, t) \land s \in Src \land cs(\pi) \neq \blacktriangledown \implies (s, t, cs(\pi)) \in CoreVars(C_{\mathcal{S}}, X_0)$$

by induction on $\pi \in Paths_G$.
So let $\pi \in Paths_G$ be a path with $\pi \in Paths_G(s, t)$, $s \in Src$ and $cs(\pi) \neq \blacktriangledown$.

1. Assume that $\pi = \epsilon$. Then $s = t$ and $(s, t, \epsilon) = (s, t, cs(\pi)) \in X_0$ since $s \in Src$. Furthermore, $C_{\mathcal{S}}$ contains a constraint

$$X_{\mathcal{S}}(s, s, \epsilon) \geq id$$

by $\textsc{empty}_{\mathcal{S}}$. Thus, $(s,t,\epsilon) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{S}}, X_0)$.

2. Assume that $\pi = \pi' \cdot e$ with $\pi' \in Paths_G(s,t')$ and $t' \overset{e}{\to} t$. By definition of cs, $cs(\pi) \neq \boxtimes$ implies that $cs(\pi') \neq \boxtimes$. Hence by induction hypothesis we may assume

$$(s,t',cs(\pi')) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{S}}, X_0).$$

We observe that $\mathcal{C}_{\mathcal{S}}$ contains the constraint

$$(\star) \qquad X_{\mathcal{S}}(s,t,cs(\pi)) \geq f_e \circ X_{\mathcal{S}}(s,t',cs(\pi')).$$

To see this, we make a case distinction on e.

$e \in E_{intra}$: Then $cs(\pi) = cs(\pi')$ and we obtain the constraint by $\textsc{intra}_{\mathcal{S}}$.

$e \in E_{call}$: Then $cs(\pi) = push(e_{call}, cs(\pi'))$ and we obtain the constraint by $\textsc{call}_{\mathcal{S}}$.

$e \in E_{ret}$: Then one of the following statements holds:

$$(1) \qquad cs(\pi) = cs(\pi') = \epsilon$$

$$(2) \qquad cs(\pi') \neq \epsilon \wedge (top(cs(\pi')), e) \in \Phi \wedge cs(\pi) = pop(cs(\pi')).$$

In case (1), we obtain the constraint by $\textsc{ret}^{(1)}_{\mathcal{S}}$ and in case (2) by $\textsc{ret}'^{(2)}_{\mathcal{S}}$.

Since $(s,t',cs(\pi')) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{S}}, X_0)$, constraint $(\star)$ is contained in $\mathcal{C}\mathrm{ore}(\mathcal{C}_{\mathcal{S}}, X_0)$ and we obtain

$$(s,t,cs(\pi')) \in \mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{S}}, X_0),$$

as desired.

□

To prove the definition-completeness, I need a connection between $\mathcal{V}^{(\mathcal{S})}_0$ and $\mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{S}}, N \times N \times S)$. Its proof is analogous to the proof of, e.g., (7.43) in Lemma 7.22.

Lemma 7.36. *For all $s,t \in N$ and $\sigma \in S$, we have*

$$\mathcal{V}^{(\mathcal{S})}_0 = \mathcal{C}\mathrm{oreVars}(\mathcal{C}_{\mathcal{S}}, N \times N \times S) \cap Src \times N \times S$$

Now I can give the proof of the definition-completeness of $\mathcal{V}_0^{(\mathcal{S})}$ in $\mathcal{C}\text{oreVars}(C_{\mathcal{S}}, N \times N \times S)$.

Lemma 7.37. $\mathcal{C}\text{ore}(C_{\mathcal{S}}, X_0)$ *is definition-complete with respect to its superset* $\mathcal{C}\text{ore}(C_{\mathcal{S}}, N \times N \times S)$*.*

Proof. We use Theorem 7.10. Let $c \in \mathcal{C}\text{ore}(C_{\mathcal{S}}, N \times N \times S)$ with $lhs(c) \in \mathcal{C}\text{oreVars}(C_{\mathcal{S}}, X_0)$. Then we have to show

$$FV(rhs(c)) = \emptyset \implies lhs(c) \in X_0 \tag{7.69}$$

$$FV(rhs(c)) \neq \emptyset \wedge x \in FV(rhs(c)) \implies x \in \mathcal{C}\text{oreVars}(C_{\mathcal{S}}, X_0) \tag{7.70}$$

We show the two claims separately.

(7.69): If $FV(rhs(c)) = \emptyset$, then c is of the form

$$X_{\mathcal{S}}(s, s, \epsilon) \geq id$$

Since $(s, s, \epsilon) = lhs(c) \in \mathcal{C}\text{oreVars}(C_{\mathcal{S}}, X_0)$, it must be the case that $(s, s, \epsilon) \in X_0$ by Lemma 7.36.

(7.70): Let $FV(rhs(c)) \neq \emptyset$. Inspection of Constraint System 7.3 shows that c is of the form

$$X_{\mathcal{S}}(s, t, \sigma) \geq f_e \circ X_{\mathcal{S}}(s, t', \sigma')$$

where $t' \xrightarrow{e} t$ and $(s, t, \sigma) = lhs(c)$. Note that the s on the right-hand side is the same as on the left-hand side. Since $c \in \mathcal{C}\text{ore}(C_{\mathcal{S}}, N \times N \times S)$, we have $(s, t', \sigma') \in \mathcal{C}\text{oreVars}(C_{\mathcal{S}}, N \times N \times S)$. Moreover, since $(s, t, \sigma) \in \mathcal{C}\text{oreVars}(C_{\mathcal{S}}, X_0)$, we get $s \in Src$ by Lemma 7.36. Thus $(s, t', \sigma') \in \mathcal{C}\text{oreVars}(C_{\mathcal{S}}, X_0)$ by Lemma 7.36.

□

Theorem 7.34 states that Algorithm 16 solves $C^{(\mathcal{S})}$ for an appropriate subset of $N \times N \times S$. In particular, for every $(s, t) \in Src \times N$ such that $AP(s, t) \neq \emptyset$, there is some $\sigma \in S$ such that the result $\mathcal{A}_{\mathcal{S}}(s, t, \sigma) = lfp(F_{C^{(\mathcal{S})}})(s, t, \sigma)$. It remains to establish a connection between $lfp(F_{C^{(\mathcal{S})}})$ and $MOVP$ to be fully convinced that Algorithm 16 indeed yields a $(VP, Src \times N)$-correct solution.

Now define

$$\mathcal{A}(s,t) \overset{def}{=} \bigsqcup_{\sigma \in S} \mathcal{A}_{\mathcal{S}}(s,t,\sigma).$$

Then the above considerations and Theorem 6.16 entail that

$$AP(s,t) \neq \emptyset \implies \mathcal{A}(s,t) \geq MOAP(s,t). \tag{7.71}$$

Moreover, if $\mathcal{F}$ is distributive, then by Theorem 6.18, I can refine this to

$$AP(s,t) \neq \emptyset \implies \mathcal{A}(s,t) = MOAP(s,t)$$

Finally, if $\mathcal{S}$ is chosen appropriately, it follows from Corollary 6.33 that Algorithm 16 can be used to compute a $(VP, Src \times N)$-correct solution.

Theorem 7.38. *Assume that there is a stack abstraction between $\mathcal{S}_\infty$ and $\mathcal{S}$. Let $\mathcal{A}_{\mathcal{S}}$ be the result of Algorithm 16 and define*

$$\mathcal{A}(s,t) \overset{def}{=} \bigsqcup_{\sigma \in S} \mathcal{A}_{\mathcal{S}}(s,t,\sigma).$$

Then $\mathcal{A}$ is $(VP, Src \times N)$-correct.

Proof. Assume $(s,t) \in Src \times N$. Then we have to show $MOVP(s,t) \leq \mathcal{A}(s,t)$. For $VP(s,t) = \emptyset$, there is nothing to show, so we may assume $VP(s,t) \neq \emptyset$. Because of our assumption about $\mathcal{S}$, we may apply Corollary 6.33 and yield $VP(s,t) \subseteq AP(s,t)$ and

$$MOVP(s,t) \leq MOAP(s,t) \tag{7.72}$$

Because $VP(s,t) \neq \emptyset$, we get $AP(s,t) \neq \emptyset$. Hence, we may apply (7.71) and get

$$MOAP(s,t) \leq \mathcal{A}(s,t) \tag{7.73}$$

(7.72) and (7.73) imply $MOVP(s,t) \leq \mathcal{A}(s,t)$, as desired. □

And I think it's gonna be a long long time.
Elton John

8 Implementation and Evaluation

In the previous chapters, I theoretically developed and examined various algorithms to conduct generalized interprocedural data-flow analysis on interprocedural graphs. The purpose of this chapter is to demonstrate that these theoretically stated algorithms also work in practice. To this end, I implemented the algorithms in the Joana framework and conducted an evaluation of both performance and precision.

This chapter is organized as follows. First, I am going to give an overview of my implementation in section 8.1. The other sections are dedicated to the evaluation.

Section 8.2 describes several aspects on the setup of my evaluation, particularly which samples and instances I chose. Then, I describe and discuss the results of the performance evaluation in section 8.3. Lastly, section 8.4 is dedicated to the precision evaluation.

8.1 Notes on the Implementation

My implementation allows for arbitrary data-flow framework instances. In order to enable evaluations and comparison with Joana's summary edge computation and slicers, my implementation works on Joana's graph data structure.

In the next subsections, I am going to consider several aspects of my implementation in more detail. Subsection 8.1.1 is dedicated to my implementation of the functional approach and subsection 8.1.2 considers my implementation of the call string approach. Finally, in subsection 8.1.3, I will conduct a worst-case complexity analysis on all implemented algorithms.

8.1.1 Functional Approach

As we still remember from previous chapters, the functional approach consists of a preprocessing phase that computes same-level information, and a two-phase algorithm that computes the actual data-flow analysis solution using the same-level information. In the following, I will give a brief overview of my implementations of both steps, in this order.
I implemented two variants of the same-level info computations, which I will elaborate on in subsubsection 8.1.1.2 and subsubsection 8.1.1.3, respectively. Before that, in subsubsection 8.1.1.1 I give some general hints on how my implementations represent same-level information.
After having described my summary info computation implementations, I consider my implementation of the actual two-phase data-flow analysis in subsubsection 8.1.1.4.

8.1.1.1 Representing Same-Level Information

Recall that both the summary edge algorithm and the two-phase slicer employ *summary edges* between actual-in and actual-out nodes in order to avoid descending into callees.
My implementations of the generic same-level problem use a generalized version of summary edges: In addition to the mere information that there is a same-level path, my generalized summary edges in addition are annotated with the result of the same-level solution for the given pair of nodes.
More specifically, a common difference to Algorithm 9 is that expressions of the form $f_{e_{ret}} \circ \mathcal{A}^{(SL)}(n_0, n_1) \circ f_{e_{call}}$ are stored separately in a map $sumInfo$. Moreover, my implementation of the generalized two-phase approach assumes that $sumInfo$ is available. Practically, I persist $sumInfo$ in a separate file using the JSON format [94].
I implemented two variants of Algorithm 9, *consequent* and *optimized*. I discuss them briefly in the following.

8.1.1.2 Consequent Summary Info Computation

The *consequent* variant is very close to Algorithm 9 and only specifies an order in which the worklist items are traversed. In the following, I give some hints on how this ordering is chosen.

Remember that Algorithm 9 maintains pairs (s, t) of nodes on its worklist, where s and t belong to the same procedure. The ordering aims to ensure that callees are processed before callers. The idea is that, if we process a given procedure p, we want to use the most recent summary information to avoid re-computation. Hence we aim to ensure that the procedures called by p are processed before p is processed. This works perfectly as long as there are no recursive cycles in the call graph.
The ordering consists of two parts: The first part orders procedures on the call graph, while the second part orders the nodes within a procedure.
The ordering on the call graph is obtained as follows.

1. We are given a call graph $C = (P, E)$ of the given PDG G where P consists of the procedures of the given PDG G and two nodes p and p' are connected by a directed edge $p \to p'$ if p contains a call of p'.

2. Reverse the edges of C and obtain graph $C' = (P, E')$.

3. Compute the *condensation* of C', that is, a graph $C'' = (S, E'')$ such that the nodes $S \subseteq 2^P$ of C'' are the strongly connected components of C' and $A \to_{C''} B$ if and only if $\exists p \in A.\ \exists p' \in B.\ p \to_{C'} p'$. Note that C'' is acyclic [59, p. 98].

4. Now sort C'' topologically. This yields a total order $<_{top}$ of C''.

5. By carefully enumerating the nodes in each SCC A in a well-defined fashion, use $<_{top}$ to obtain a function $i : P \to \mathbb{N}$ of G' such that $i(p) \neq i(p')$ iff $p \neq p'$ and $i(p) < i(p')$ if p and p' belong to two different SCCs A and B with $A <_{top} B$.

Orderings within the procedures were obtained by enumerating the nodes in depth-first order. Note that this relies on Joana's node iteration order for its internal PDG structure and therefore can be subject to non-determinism on some level.

8.1.1.3 Optimized Summary Info Computation

Variant *optimized*, which is depicted in Algorithm 17, is based on the observation that Algorithm 9 actually computes a global same-level solution, although it only needs a relatively small part of it, namely the summary information for pairs of call nodes and their return counterparts. Instead,

the *optimized* variant only maintains the actual summary information, re-computes intraprocedural parts of the solution as needed and discards non-essential parts of them as soon as possible. Also, instead of node pairs that need to be updated, it maintains procedures that need to be re-processed on its worklist.
Processing a procedure p means to completely re-compute the intraprocedural result for p and to propagate this result to p's callers. If this propagation leads to a change in some summary information between two nodes m and n, the procedure that contains m and n is scheduled for re-computation by putting it on the worklist. For worklist ordering, a callee-caller-ordering similar to the procedure ordering of the *consequent* variant is used.
Initially, all procedures are put on the worklist to ensure that each intraprocedural result is computed at least once.

8.1.1.4 Generalized Two-Phase Approach

For the two-phase approach, I implemented Algorithm 13, which can give precision guarantees for distributive problems. My implementation is relatively straight-forward. The most notable deviation from Algorithm 13 is that the implementation uses summary information instead of same-level information, as described in subsubsection 8.1.1.1.

8.1.2 Call-string Approach

In my implementation of the call-string approach, I use a variant of Algorithm 16 that exploits fundamental properties of program dependence graphs with respect to their correspondence relation Φ.
Recall that Algorithm 16 uses stacks, i.e. sequences of call edges, to remember to which caller the analysis shall return after exiting a given procedure. If such a stack $e_{call} \cdot \sigma$ is given and the algorithm is about to leave procedure p through the return edge e_{ret}, it needs to check that $(e_{call}, e_{ret}) \in \Phi$, i.e. that the top of the stack corresponds to e_{ret}, in order to proceed with $tgt(e_{ret})$ and stack σ.
In a program dependence graph, we generally have $(e_{call}, e_{ret}) \in \Phi$ if and only if $src(e_{call})$ and $tgt(e_{ret})$ belong to the same call site and $tgt(e_{call})$ and $src(e_{ret})$ belong to the same procedure. Hence, for PDGs, the stack does

not need to contain call edges, but only the call sites. A call site can be represented by the node that describes the actual call instruction.
Hence, my implementation of Algorithm 16 uses sequences of call *nodes* as stacks. Using sequences of call nodes instead of parameter edges as stacks mainly saves a lot of space. This is especially beneficial for calls of procedures with many parameters. With the ordinary stack representation, all these parameters induce different, equivalent call stacks.

Algorithm 17: A variant of Algorithm 9 that trades recomputation of intra-procedural results for a more compact solution and worklist representation – for *updateIntraprocResult*, see Algorithm 18

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295
Result: least summary information for $\mathcal{F}$
procedure computeSumInfoOptimized
    $sumInfo \leftarrow const(\boxtimes)$
    $W \leftarrow Proc$
    while $W \neq \emptyset$ do
        $p \leftarrow remove(W)$
        foreach $s \in Entries_p$ do
            $newResults \leftarrow updateIntraprocResult(s, sumInfo)$
            foreach $t \in Exits_p$ do
                foreach $(e_{call}, e_{ret}) \in \Phi$ and $m, n$ s.t. $m \overset{e_{call}}{\to} s \wedge t \overset{e_{ret}}{\to} n$ do
                    $old \leftarrow sumInfo(m, n)$
                    $sumInfo(m, n) \leftarrow sumInfo(m, n) \sqcup f_{e_{ret}} \circ newResults(s, t) \circ f_{e_{call}}$
                    if $sumInfo(m, n) \neq old$ then
                        $W \leftarrow W \cup \{proc(m)\}$
    return $sumInfo$
```

Algorithm 18: Intraprocedural part of Algorithm 17

```
Input: a data-flow framework instance $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ as described on page 295, $s \in N$, function sumInfo: $N \times N \to F_{\boxtimes}$
Result: updated version of $sumInfo$ on $\{s\} \times Exits_p$
procedure $updateIntraprocResult(s\colon N, sumInfo\colon N \times N \to F_{\boxtimes})$
    $Analysis \leftarrow const(\boxtimes)$
    $W = W \cup \{s\}$
    $Analysis(s,s) \leftarrow id$
    while $W \neq \emptyset$ do
        $t \leftarrow remove(W)$
        foreach $t' \in N$ such that $t \xrightarrow{e} t' \wedge e \in E_{intra}$ do
            $old \leftarrow Analysis(s,t')$
            $Analysis(s,t') \leftarrow Analysis(s,t') \sqcup f_e \circ Analysis(s,t)$
            if $Analysis(s,t') \neq old$ then
                $W \leftarrow W \cup \{t'\}$
        foreach $t' \in N$ such that $sumInfo(t,t') \neq \boxtimes$ do
            $old \leftarrow Analysis(s,t')$
            $Analysis(s,t') \leftarrow Analysis(s,t') \sqcup sumInfo(t,t') \circ Analysis(s,t)$
            if $Analysis(s,t') \neq old$ then
                $W \leftarrow W \cup \{t'\}$
    return $Analysis \restriction \{s\} \times Exits_p$
```

8.1.3 Complexity Considerations

For data-flow frameworks (L, F) where F has finite height n, an asymptotic upper bound on the time taken to execute Algorithm 8 can be given in terms of the number of applications of constraints, as this is the most elementary and most often executed operation in this algorithm. Any constraint on a given system can be applied at most n times. Hence, the total number of constraint applications is no more than $\mathcal{O}(|C| \cdot n) = \mathcal{O}(|C| \cdot |\mathcal{F}|)$. For frameworks where F does not have finite height but satisfies (ACC), such a bound cannot be given in the general case, so that the concrete instance has to be taken into account.

Next, I am going to give worst-case time bounds for the different algorithms I presented in chapter 7. For simplicity, I assume that F has finite height. Subsection 8.1.3.1 considers the algorithms of the functional approach, while subsubsection 8.1.3.2 consider the call-string approach.

8.1.3.1 Functional Approach

8.1.3.1.1 Consequent Same-Level Problem Solver In order to give a worst-case time bound, I give an upper bound for the size of the subset C'_{SL} of Constraint System 6.1 that is solved by Algorithm 9:

$$\begin{aligned}
&|C'_{SL}| \\
&\leq |N_{entry}| \\
&+ \sum_{p \in Proc} |N_p^{entry}| \cdot |E_p^{intra}| \\
&+ \sum_{p \in Proc} |N_p^{entry}| \cdot |\Phi \cap \{(n \overset{e_{call}}{\to} n_0, n_1 \overset{e_{ret}}{\to} t) \mid proc(n) = proc(t) = p\}|
\end{aligned}$$

where $N_p^{entry} \overset{def}{=} N_p \cap N_{entry}$ and $E_p^{intra} \overset{def}{=} E_p \cap E_{intra}$.
The first term counts the number of constraints of the form SL-SOL-(I): C'_{SL} contains such a constraint for every $s \in N_{entry}$. The second term approximates the number of constraints of the form SL-SOL-(II): C'_{SL} contains such a constraint at most for every $s \in N_{entry}$ and every $e \in E_{intra}$ whose source and target both lie in the same procedure as s. Finally, the third term approximates the number of constraints of the form SL-SOL-(III): C'_{SL} contains such a constraint at most for every $s \in N_{entry}$ and every $(e_{call}, e_{ret}) \in \Phi$ such that $src(e_{call})$ and $tgt(e_{ret})$ lie in the same procedure as s. The second and third terms are only approximations since they do not take into account the reachability analysis performed by Algorithm 9.
$|N_{entry}|$ can be bounded by $|N|$ and $|E_p^{intra}|$ can be bounded by $|E|$. Moreover,

$$|\Phi \cap \{(n \overset{e_{call}}{\to} n_0, n_1 \overset{e_{ret}}{\to} t) \mid proc(n) = proc(t) = p\}|$$

can be bounded by $|E|^2$. In summary, an asymptotic upper bound to the overall time needed by Algorithm 9 in terms of the size of the given PDG G can be given by

$$\mathcal{O}(|F| \cdot (|N| + |N| \cdot |E| + |N| \cdot |E|^2)) = \mathcal{O}(|F| \cdot |N| \cdot |E|^2).$$

8.1.3.1.2 Optimized Same-Level Problem Solver For the optimized same-level problem solver, I give a coarse yet simple upper bound that only uses $|N|$, $|E|$, $|F|$ and the number $|P|$ of procedures in the given graph. First, consider `updateIntraprocResult`. It consists of an instantiation of Algorithm 8 that solves a part of Constraint System 6.1. Using the argument from above, an invocation of `updateIntraprocResult` takes no more than $\mathcal{O}(|F| \cdot |E|^2)$ constraint applications. Next, consider one iteration of the main loop in Algorithm 17: It consists of no more than $|N|$ invocations of `updateIntraprocResult` and no more than $|N| \cdot |N|$ constraint applications. Hence, one iteration of the loop does not take more than $\mathcal{O}(|N| \cdot |N| \cdot |F| \cdot |E|^2 + |N| \cdot |N|) = \mathcal{O}(|N| \cdot |N| \cdot |F| \cdot |E|^2)$ rule applications. A given procedure p can be put at most $|N| \cdot |N| \cdot |F|$ times onto the worklist ($sumInfo$'s domain consists of pairs of nodes and every value can change at most $|F|$ times), hence the loop can be executed no more than $|P| \cdot |N|^2 \cdot |F|$ times. All in all, Algorithm 17 executes no more than $\mathcal{O}(|P| \cdot |N|^4 \cdot |F|^2 \cdot |E|^2)$ constraint applications.

Again, I would like to point out that this bound is indeed very coarse and formally appears to be worse than the bound for the consequent same-level problem solver. I suspect that it is possible to conduct a more elaborate analysis and yield a tighter upper bound that is closer to the bound for the consequent variant. The evaluation will show that the optimized variant can perform better than the consequent variant in practice.

8.1.3.1.3 Generalized Two-Phase Approach For Algorithm 13, similar considerations can be made as in paragraph 8.1.3.1.1. Hence, we arrive at the same rough upper bound $\mathcal{O}(|F| \cdot |N| \cdot |E|^2)$. However, there is one important aspect that needs to be highlighted. The summary information computation only needs to be conducted once and can then be incorporated into the given graph. A usual approach is to insert summary edges into the given graph and annotate them with the respective value of the same-level analysis result. Then, Algorithm 13 does not operate on the same graph as Algorithm 17, but on an extended graph with more edges. This lowers the upper bound of

$$|\Phi \cap \{(n \overset{e_{call}}{\to} n_0, n_1 \overset{e_{ret}}{\to} t) \mid proc(n) = proc(t) = p\}|$$

to $|E|$ and the overall bound to $\mathcal{O}(|F| \cdot |N| \cdot |E|)$.
To put it more shortly, using the functional approach makes the actual problem solver linear in the graph size, but may cause the graph's number of edges to increase quadratically. Note that this also applies to simple slicing.

8.1.3.2 Call-String Approach

For the analysis of the call-string approach, I assume a fixed source node $s \in N$ and the stack space $\mathcal{S}_k$ for $k \in \mathbb{N}$ and consider the part of Constraint System 6.5 where the variables have s as first component. A closer look then shows that there is one constraint for every edge $e \in E$ and every stack $\sigma \in S^k$. This means that the size of the constraint system is bounded to $\mathcal{O}(|E| \cdot |E_{call}|^k) = \mathcal{O}(|E|^{k+1})$, which results in an upper bound of $\mathcal{O}(|F| \cdot |E|^{k+1})$ for the costs of Algorithm 16.
As I mentioned in subsection 8.1.2, my implementation uses node sequences instead of edge sequences to represent stacks which changes the bound to $\mathcal{O}(|F| \cdot |N|^{k+1})$.

8.2 Description of the Setup

In this section, I describe the setup of all my evaluations. Subsection 8.2.1 describes the environment of the evaluation. After that, subsection 8.2.2 gives an overview of the programs that I ran my analyses on. Lastly, subsection 8.2.3 describes the data-flow framework instances that I selected for this evaluation.

8.2.1 Evaluation Environment

The environment of my evaluation consists of the hardware and the software of the setup that I used but not provided myself. An overview of this is given in Table 8.1.

CPU	Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Memory	512 GB RAM
Operating System	Ubuntu 18.04.4 LTS
Kernel Version	4.15.0-72-generic
JDK	openjdk version "11.0.7" 2020-04-14
JVM	OpenJDK Runtime Environment (11.0.7+10-post-Ubuntu-2ubuntu218.04)

Table 8.1: Characteristics of the machine used for performance evaluations

The machine that I ran my experiments on consisted of 80 cores. I ran multiple experiments in parallel, in order to save time. This may have slightly disturbed the results, however, I believe that this effect is negligible.

8.2.2 Samples

The programs I used for the evaluation are described in Table 8.2.
As a preliminary step to all evaluations, I used Joana to construct PDGs without summary edges for all sample programs. I assumed a sequential setting, i.e. I configured Joana to not analyze threads. Moreover, I used context-insensitive points-to analysis, parameter modelling based on object graphs [78] and interprocedural exception analysis.
Including all libraries added to the respective analysis scope, the example programs have between 654k and 15.7M bytecode instructions. After call graph construction, on average 95% were classified as unreachable. Additionally, Joana pruned away 53% of the remaining instructions, so that the parts of the programs that were covered by the resulting program dependence graphs had between 1.6k and 119.3k bytecode instructions.
I divided the example programs in two groups: One group consists of four very large programs, namely `javap`, `dacapo-eclipse`, `hsqldb` and `freecs`, the other group consists of the rest of the programs. In the following, I refer to these programs as *large*, and to the others as *non-large*.
The reason for this grouping is that the large programs performed substantially worse than all other programs (across all algorithms and instances), which is why I had to choose different evaluation parameters.
The sizes of the corresponding PDGs are shown in Table 8.3.

sample name	description
ant	Apache Ant (version 1.9.6), as found in Ubuntu 16.04.3 LTS
bibtex2website	a tool for managing the publications of the programming paradigms group
dacapo-antlr	sample from the Dacapo benchmark suite (version 2006-10-MR2) [32]
dacapo-eclipse	sample from the Dacapo benchmark suite
dacapo-fop	sample from the Dacapo benchmark suite
dacapo-hsqldb	sample from the Dacapo benchmark suite
dacapo-luindex	sample from the Dacapo benchmark suite
dacapo-lusearch	sample from the Dacapo benchmark suite
dacapo-xalan	sample from the Dacapo benchmark suite
eVotingMachine	a case study from the E-Voting Reference Scenario [115]
freecs	a version of freecs (a chat server) that was also used for previous evaluations [78]
hsqldb	a version of hsqldb (a database engine) that was also used for previous evaluations [78]
jasypt-decrypt	decryption command-line tool from Jasypt[99] (version 1.9.2)
jasypt-digest	command-line tool from Jasypt for computing message digests
jasypt-encrypt	digest command-line tool from Jasypt
javap	Java disassembler from Oracle's JDK (version 1.7.0_80-b15)
jftp	case study from Lovat et al. [122]
jzip	case study from Lovat et al. [122]
jlex	A Lexical Analyzer Generator for Java [29, 30]
lethal	a demo program from lethal, a tree and hedge automata library developed in a student's project at the University of Münster [98]
maven	Apache Maven (version 3.3.9), as found in Ubuntu 16.04.3 LTS
mixServer	a case study from the E-Voting Reference Scenario [163]

Table 8.2: Description of the sample programs used for this evaluation

name	no. PDG nodes	no. PDG edges
mixServer	3865	21271
eVotingMachine	5655	33580
ant	7542	55397
bibtex2website	11031	71115
jftp	22550	170813
jlex	33525	254996
dacapo-hsqldb	37159	209545
dacapo-xalan	48387	279980
jzip	57637	349359
dacapo-fop	87453	507111
dacapo-luindex	140974	959292
jasypt-decrypt	154842	1020183
jasypt-encrypt	154868	1020173
jasypt-digest	162970	1075326
dacapo-lusearch	204374	1502063
lethal	218346	3559662
maven	238963	1795599
dacapo-antlr	344254	2677337
javap	827135	6907387
dacapo-eclipse	1129723	12031685
freecs	1497110	13494055
hsqldb	2001268	17497510

Table 8.3: Sizes of the PDGs in the sample considered for my evaluation

8.2.3 Instances

I considered the three data-flow framework instances `reach` (cf. subsection 5.4.2), `explicit-info-flow` (cf. subsubsection 5.4.5.2) and `dist` (cf. subsection 5.4.7). Considering the `reach` instance enables me to compare the performance of my generic algorithms with JOANA's hand-optimized algorithms for summary edge computation and two-phase slicing. Instance `dist` has a lattice with unbounded height. Lastly, I included `explicit-info-flow` (occasionally abbreviated by `eif`) as a simple example for language-restricted reachability (cf. subsection 5.4.5) that does not need additional parameters like a barrier.

8.3 Performance Evaluation

In this section, I describe how I conducted my performance evaluation and discuss its result.

My performance evaluation falls into two groups: Same-level problem solvers and data-flow analyses. I consider the former in subsection 8.3.1, while subsection 8.3.2 is dedicated to the latter.

8.3.1 Same-Level Problem Solvers

I evaluated both the consequent (`cons`) and the optimized (`opt`) variants of the same-level problem solver for all three considered instances. For comparison, I also evaluated the summary edge computation which JOANA performs currently as part of SDG construction. As I explained earlier, this corresponds to the `reach` instance. This algorithm will be abbreviated as `classic`. A graphical overview of the runtimes for the non-large programs can be seen in Figure 8.1. For full results, see Table 8.5 and Table 8.6.

The remainder of this subsection is structured as follows: In subsubsection 8.3.1.1, I describe the method that I applied to conduct my measurement. After that, I discuss various aspects of the results: Subsection 8.3.1.2 gives a general overview, subsubsection 8.3.1.3 describes how the graph size influences the runtime of the algorithms, subsubsection 8.3.1.4 compares the runtime performance of the different algorithms and subsubsection 8.3.1.5 discusses how the instance affects the runtime. Lastly, I give a short summary in subsubsection 8.3.1.6.

8.3.1.1 Method

I ran the different algorithms m times, distributed over n JVM invocations. For each of the $m \cdot n$ runs, I performed w warm-up iterations. An overview of the choices of these parameters for the different samples and algorithms is given in Table 8.4.

The times that I report for every sample, algorithm and instance are the average times of the $m \cdot n$ runs.

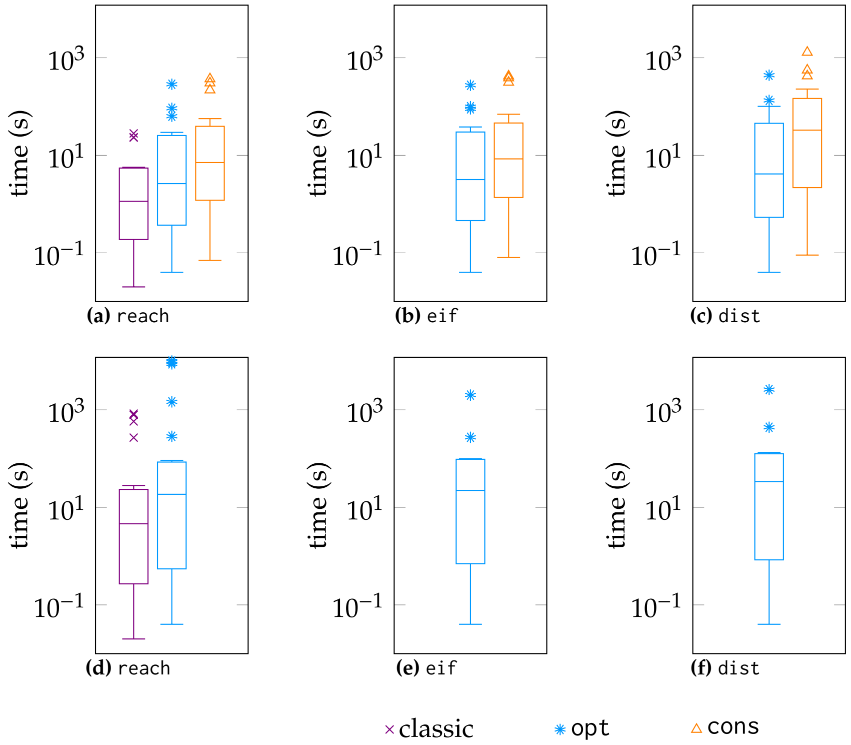


Figure 8.1: Runtime distributions for the various same-level problem solvers and instances – a-c: only non-large, d-f: including large

8.3.1.2 General Impression of Runtimes and Error Discussion

Generally, for all instances and most programs, all algorithms finished within reasonable time. For large programs, the runtimes ranged between several minutes (`classic`/`reach`) and up to several hours (`opt`). Algorithm `cons` was not able to finish its computation on the large programs within the given time and space constraints.

In order to assess the quality of these averages, I also computed confidence intervals for confidence levels $1 - \alpha = 0.95$, assuming that the runtimes are normally distributed [124, §4.2]. The radii of these intervals can be found in the *err* columns of Table 8.5 and Table 8.6. With respect to this metric, it

example group	algorithm	chosen parameters				
		m	n	w	heap	timeout
non-large large	all `reach`	10	10	3	32 GB	2 hours
large	`opt` `cons`	1	3	0	64 GB	12 hours

Table 8.4: Overview of the chosen parameters for the evaluation of the different same-level problem solvers – m is the number of JVM invocations, n is the number of iterations per JVM invocation, w is the number of warmup iterations before each iteration

can be said that the accuracy of the reported runtimes corresponds to the number of runs performed to obtain the runtimes: For large programs and non-`classic` algorithms, the error can be up to 27% relative to the reported time. Conversely, for the rest of the configurations, errors tend to be small (up to 8%, 1-2% on average). However, the benefit of performing more runs on large programs is negligible with respect to the effort, since the runtimes are an order of magnitude larger than for the non-large programs. Also note that the runtimes are subject to distortions caused by continuous optimizations and garbage collection performed by the JVM. Apart from trying to distribute the runs over multiple JVM invocations, I do not consider these issues in the scope of this thesis. For further information, see Georges et al. [64].

8.3.1.3 Relationship Between Graph Size and Runtime

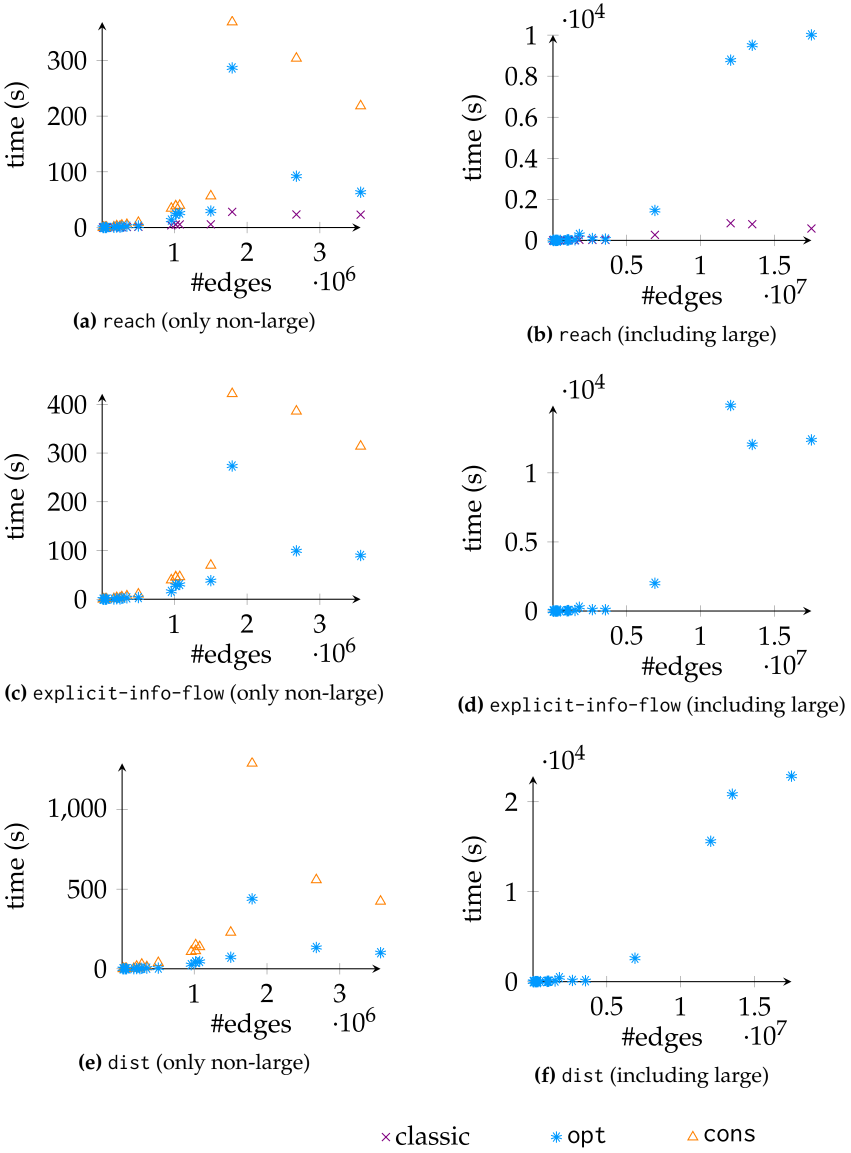
Figure 8.2 visualizes how the runtimes of the various same-level problem solvers relate to the graph size. Graph size is measured in the number of edges. We see that generally the runtime increases with the number of edges. The regression appears to be somewhere between linear and quadratic, although it is rather difficult to make a reliable statement here since the number of programs is too small and has some notable outliers. For example, all solvers consistently take longer on `maven` than on `dacapo-antlr`, although `maven`'s PDG has fewer edges than `dacapo-antlr`'s PDG (compare Table 8.3, Table 8.5 and Table 8.6). Conversely, `hsqldb` performs better relative to `freecs` than the ratio of their respective graph sizes suggests. Still, the number of edges in a program dependence graph appears to be a sufficiently reliable criterion for estimating the runtime of the measured algorithms and does not contradict the theoretical complexity analysis conducted in subsection 8.1.3, which suggests that the runtimes of all same-level problem solvers grows no more than quadratically with the number of edges.

8.3.1.4 Comparison of the Algorithms

Figure 8.3 shows how the different algorithms perform relative to each other on the chosen set of programs and instances. For this, I computed for each pair of algorithms (a_1, a_2) and each sample s the ratio of the runtimes that a_1 and a_2 took on s. Figure 8.3 shows boxplots of the resulting distributions.
Two key observations can be made here.

8.3.1.4.1 `reach` Instance For one, Joana's hand-optimized summary edge computation algorithm is clearly superior to the two generic algorithms. It can be up to 10 times as fast as the `opt` variant. This observation is not very surprising, since the summary edge computation is tailored to the `reach` instance and was optimized especially to work on large PDGs.

8.3.1.4.2 Comparison of Opt and Cons The other observation is that the `opt` variant performs moderately faster than the `cons` variant. Apart from the fact that for the majority of configurations, `opt` performed 2.5 to 3 times as fast as `cons`, `opt` was able to finish the computation on the large samples for all instances within the given time and space constraints, whereas `cons` was not. Recall from subsection 8.1.1 that the improvement in `opt` mainly consists of a more compact worklist item representation: The worklist contains procedures to be processed as opposed to pairs of entry/exit nodes. This results in a much smaller worklist, hence less memory consumption. The processing of a procedure consists of a complete intraprocedural traversal and re-computation for all intra-procedural node pairs. The price of this modification is that procedures are re-processed even if only a small part of them changed – resulting in potentially lots of spurious re-computations. This is also reflected in the runtime results.



(a) `reach` (only non-large)

(b) `reach` (including large)

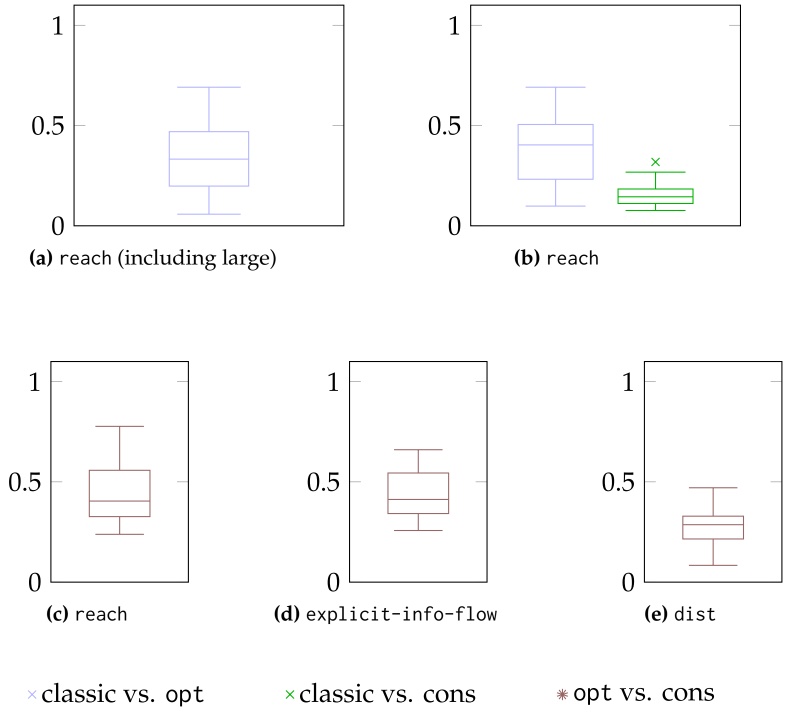
(c) `explicit-info-flow` (only non-large)

(d) `explicit-info-flow` (including large)

(e) `dist` (only non-large)

(f) `dist` (including large)

Figure 8.2: Relationship between graph size and runtime for the various same-level problem solvers and instances



(a) reach (including large)

(b) reach

(c) reach

(d) explicit-info-flow

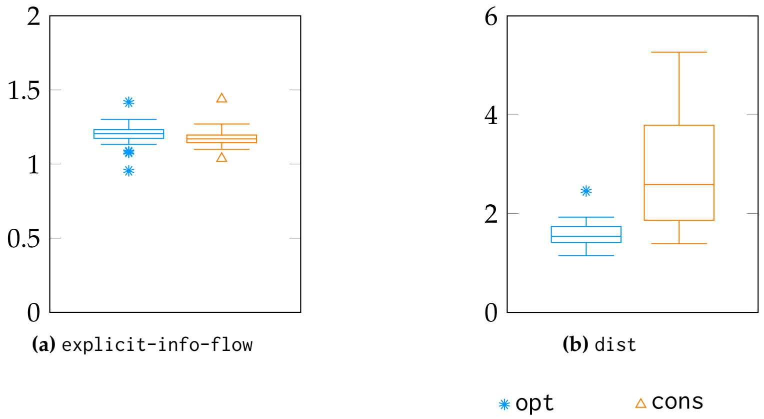
(e) dist

Figure 8.3: Comparison between the evaluated algorithms for same-level computation – a and b compare classic with opt and cons, respectively, c–e compare opt with cons

The opt variant is consistently faster than the cons variant, but only by a moderately small constant factor. Moreover, opt is able to process programs with rather large PDGs, whereas cons is not.

8.3.1.5 Comparison Between Different Instances

Figure 8.4 compares the different instances with respect to the additional effort relative to the reach instance. For this purpose, I computed for each program the ratio between the required runtime for the respective instance and the reach instance. Figure 8.4 shows a boxplot of the distributions of these ratios.



(a) `explicit-info-flow`

(b) `dist`

Figure 8.4: Additional effort of same-level computation for non-`reach` instances (non-large programs)

For both generic algorithms, instance `explicit-info-flow` takes less additional effort than `dist`. Moreover, it can be seen that algorithm `opt` performs slightly better on `dist`: For `cons`, the majority of samples take at least 2.5 times as much time as for `reach`. The corresponding factor for `opt` is less than 2.

8.3.1.6 Summary

The key takeaways of the runtime evaluation of the same-level problem solvers are:

- all evaluated problem solvers take more time on larger PDGs – the runtimes appear to grow no more than quadratically with the number of PDG edges
- Joana's hand-optimized summary edge algorithm performs much better than the generic algorithms on the `reach` instance,
- it is possible to solve the same-level problems for non-trivial data-flow analyses on fairly large PDGs in reasonable time,
- more complex data-flow framework instances generally require more time,

- it is plausible that this additional work grows with the height of an instance's lattice, and
- the improved variant `opt` of the generic algorithm performs moderately better than the consequent variant `cons`– in particular, it is able to compute a result for fairly large PDGs, whereas `cons` is not.

	reach						eif		dist	
name	classic	err	cons	err	opt	err	opt	err	opt	err
mixServer	0.02	<0.01	0.07	<0.01	0.04	<0.01	0.04	<0.01	0.04	<0.01
ant	0.05	<0.01	0.25	0.01	0.09	<0.01	0.12	<0.01	0.12	<0.01
eVotingMachine	0.05	<0.01	0.16	0.01	0.08	<0.01	0.09	<0.01	0.11	<0.01
bibtex2website	0.07	<0.01	0.31	0.01	0.11	<0.01	0.13	<0.01	0.13	0.01
jftp	0.18	<0.01	0.95	0.02	0.34	0.01	0.41	0.01	0.48	0.02
jlex	0.21	0.01	1.95	0.04	0.46	0.01	0.60	0.02	0.70	0.02
dacapo-hsqldb	0.45	0.01	2.83	0.05	0.81	0.02	1.00	0.02	1.25	0.03
dacapo-xalan	0.68	0.02	3.93	0.07	1.65	0.03	1.95	0.03	2.74	0.04
jzip	0.94	0.02	5.22	0.09	2.39	0.04	2.86	0.04	3.88	0.06
dacapo-fop	1.34	0.03	8.97	0.13	2.86	0.05	3.49	0.08	4.42	0.07
dacapo-luindex	4.01	0.04	34.40	0.36	13.38	0.22	16.19	0.22	25.80	0.33
jasypt-decrypt	5.32	0.07	38.61	0.37	23.72	0.26	28.40	0.53	41.86	0.46
jasypt-encrypt	5.20	0.06	38.42	0.31	23.87	0.25	29.02	0.49	43.47	0.79
jasypt-digest	5.54	0.06	39.57	0.39	25.97	0.58	30.41	0.38	45.96	0.92
dacapo-lusearch	5.69	0.07	56.51	0.64	29.53	0.71	38.15	0.72	72.56	1.96
lethal	23.10	0.24	218.20	1.81	63.36	1.43	89.79	1.76	100.85	1.91
dacapo-antlr	23.48	0.23	303.69	2.33	92.15	0.79	99.33	0.96	133.66	1.77
maven	28.13	0.24	368.86	2.54	286.50	4.60	273.42	2.81	439.29	2.95
dacapo-eclipse	836.63	6.27	–	–	8780.16	2133.87	14873.90	88.54	15608.08	985.87
hsqldb	578.80	4.67	–	–	10010.17	566.98	12384.62	3308.67	22847.64	3783.21
freecs	788.76	12.00	–	–	9505.95	1617.24	12053.72	1896.42	20841.18	2678.90
javap	269.36	2.21	–	–	1458.44	203.93	2007.98	246.15	2606.31	612.43

Table 8.5: Performance results for the summary information computation, part 1

	eif		dist	
name	cons	err	cons	err
mixServer	0.08	<0.01	0.09	<0.01
ant	0.30	0.01	0.41	0.01
eVotingMachine	0.17	0.01	0.33	0.01
bibtex2website	0.37	0.01	0.61	0.02
jftp	1.08	0.02	1.70	0.03
jlex	2.21	0.03	3.59	0.05
dacapo-hsqldb	3.11	0.05	14.88	0.16
dacapo-xalan	4.70	0.07	27.66	0.33
jzip	6.57	0.09	11.79	0.14
dacapo-fop	10.33	0.13	38.09	0.41
dacapo-luindex	39.41	0.60	107.71	1.02
jasypt-decrypt	44.93	0.56	112.53	0.78
jasypt-encrypt	45.91	0.64	149.28	1.20
jasypt-digest	46.05	0.56	137.91	1.06
dacapo-lusearch	69.56	0.78	228.65	2.96
lethal	313.85	2.46	423.53	5.76
dacapo-antlr	385.81	3.56	558.31	6.87
maven	421.70	4.73	1290.46	7.60
dacapo-eclipse	–	–	–	–
hsqldb	–	–	–	–
freecs	–	–	–	–
javap	–	–	–	–

Table 8.6: Performance results for the summary information computation, part 2

8.3.2 Data-Flow Solvers

I evaluated three data-flow solvers for all three instances:

- `v2p`– the generic two-phase approach Algorithm 13
- `cs0`– call-string approach with a depth of 0
- `cs1`– call-string approach with a depth of 1

Note that all evaluated instances are distributive, so that Algorithm 13 indeed produces a precise result.

For comparison, I also evaluated Joana's standard two-phase slicer for the `reach` instance (`v2p-classic`). The full results can be found in Table 8.8 and Table 8.9. A graphical overview is shown in Figure 8.5.

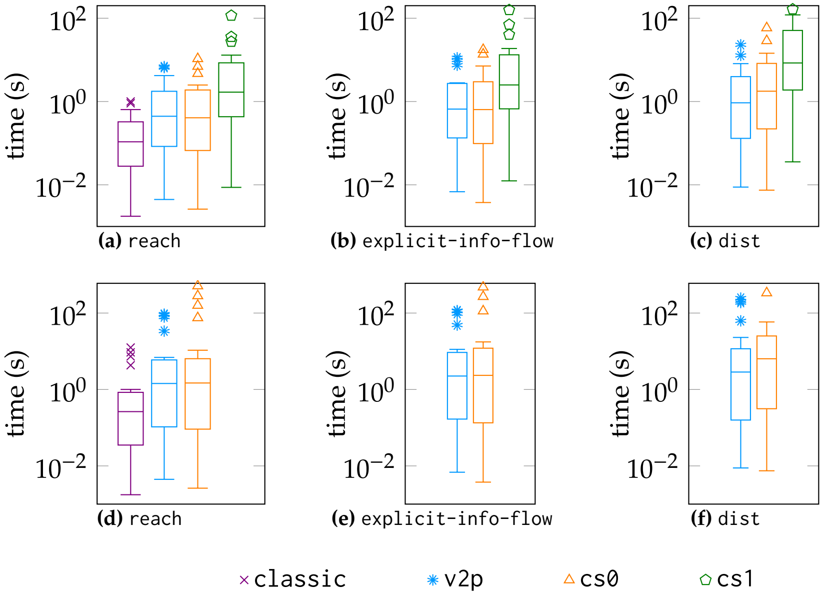


Figure 8.5: Overview of the runtime distributions of the evaluated data-flow solvers; a–c: only non-large programs, d–f: including large programs

Next, in subsubsection 8.3.2.1, I am going to describe the method that I used to obtain the results. After that, I will discuss several aspects of the results. Subsection 8.3.2.2 gives a general impression, subsubsection 8.3.2.3 considers the influence of the graph size on the runtimes, subsubsection 8.3.2.4 compares the different data-flow analysis algorithms and subsubsection 8.3.2.5 discusses the influence of the framework instances on the runtimes. Finally, I sum up my observations in subsubsection 8.3.2.6.

8.3.2.1 Method

In order to obtain a runtime measurement for the various data-flow solvers, I took for each program p a sample S_p of n nodes from p's PDG and then followed the following recipe:

- For each instance i, algorithm a and node $s \in S_p$, run a's instantiation for i on p's PDG and s m times[38] (with w warm-up iterations) in a single JVM fork. This results in m measured runtimes $t_1^{(i,a,p,s)}, \dots, t_m^{(i,a,p,s)}$.
- Let $time(i,a,p,s) = \frac{1}{m}\sum_{k=1}^{m} t_k^{(i,a,p,s)}$ be the average time along the m runs. Then let $[l(i,a,p), u(i,a,p)]$ be an estimation interval for the q-th quantile of the times $(time(i,a,p,s))_{s \in S}$ for confidence level $1 - \alpha$.
- Report the midpoint $\frac{l(i,a,p)+u(i,a,p)}{2}$ and $\frac{u(i,a,p)-l(i,a,p)}{2}$ as the time and error for program p, instance i and algorithm a.

Table 8.7 gives an overview of the parameters I chose for each configuration. For all configurations, I chose $q = 0.5$ (the median). To determine the parameters l and u, I used a statistical method for conservatively determining estimates of confidence intervals of quantiles without assumptions about the underlying distribution[124, §5.2.2].

8.3.2.2 General Impression of the Runtimes and Error Discussion

Generally, the measured runtimes show that the `v2p` solver delivers reasonably fast times for all programs under evaluation, although it is clearly inferior to JOANA's classic slicer on the `reach` instance. The `cs0` algorithm was able to finish in all configurations, while the `cs1` solver did not finish on large program within a reasonable amount of time.
The errors show the same pattern as in the evaluation for the same-level problem solvers: For most configurations, they are relatively small. For the large programs, where the number of nodes and runs was smaller, they can be very large. Hence, the times for the large programs must be read with caution. Keeping this in mind, I will not further discuss errors in the following.

[38] For `v2p` and `v2p-classic`, where most times tended to be very short, I let each iteration run in a loop for at least 1 second and reported the average times.

example group	algorithm	chosen parameters					
		n	m	w	l	u	$1-\alpha$
non-large	v2p-classic v2p cs0	100	10	3	42	64	≈ 0.95
non-large	cs1	10	5	3	3	8	≈ 0.89
large	v2p-classic v2p	100	10	3	42	64	≈ 0.95
large	cs0	10	5	3	3	8	≈ 0.89
large	cs1	(not evaluated)					

Table 8.7: Overview of the chosen parameters for the evaluation of the different data-flow solvers

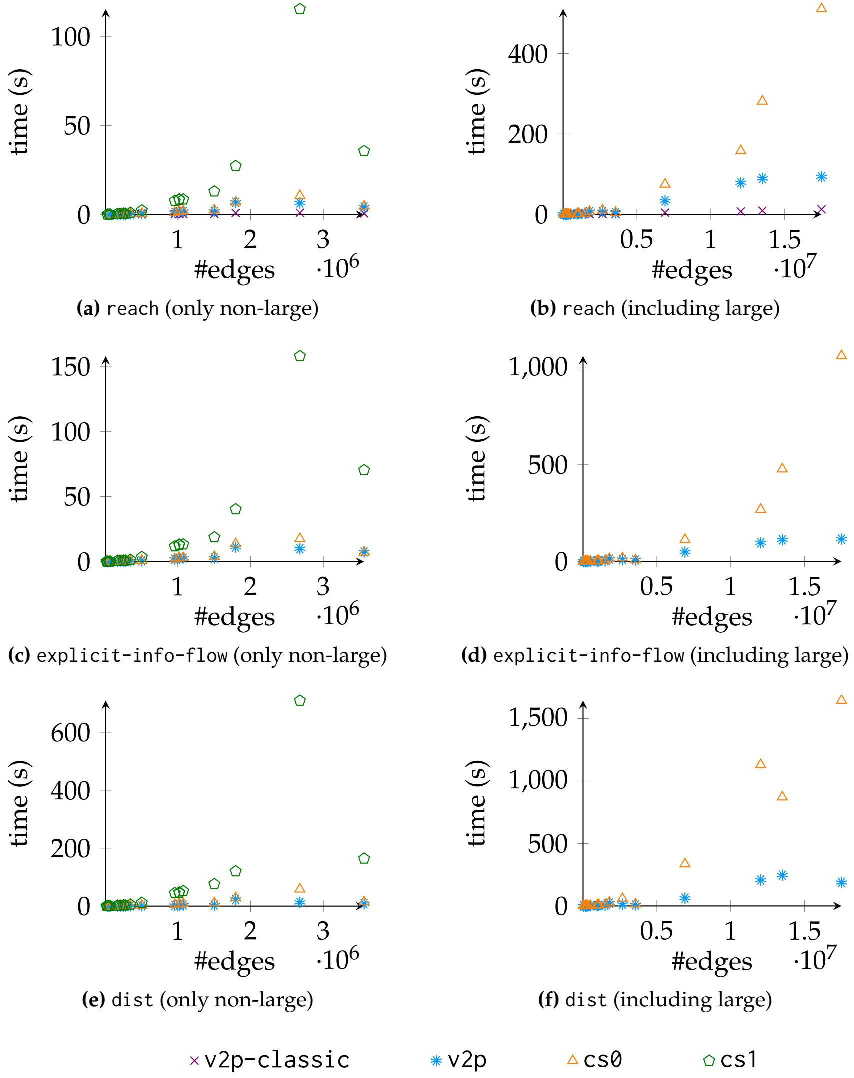
8.3.2.3 Relationship Between Graph Size and Runtime

Figure 8.6 visualizes how the runtime of the various data-flow analysis algorithms relate to graph size. Graph size is measured in the number of edges.

Generally, it can be seen that the runtime increases with graph size. Both for the classic slicer and the `v2p` algorithm, the runtime appears to be roughly linear in the graph size, whereas the call-string algorithms suggest a super-linear regression. We also see that there are outliers. For example, the times for `hsqldb` is consistently much lower than the times for `dacapo-eclipse`, although `hsqldb` is the program whose PDG has the highest number of edges. A reason for this may be that the evaluated algorithms not only solve a constraint system but also perform a reachability analysis and their runtime is also affected by the size of their forward slices: If `hsqldb` is less connected than `dacapo-eclipse`, its forward slices are smaller and data-flow analyses on `hsqldb` take less time than on `dacapo-eclipse`.

8.3.2.4 Comparison of the Algorithms

Both callstring-based algorithms perform worse than the two-phase approach. This can be seen in Figure 8.7, which visualizes the distribution of the respective runtime ratios. While `cs0` shows roughly the same runtime



(a) `reach` (only non-large)

(b) `reach` (including large)

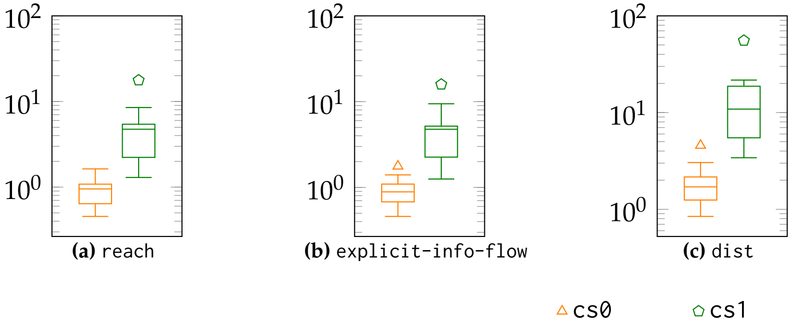
(c) `explicit-info-flow` (only non-large)

(d) `explicit-info-flow` (including large)

(e) `dist` (only non-large)

(f) `dist` (including large)

Figure 8.6: Relationship between graph size and runtime for the various DFA algorithms and instances



(a) reach **(b)** explicit-info-flow **(c)** dist

Figure 8.7: Performance comparison between the call-string-based algorithms and v2p

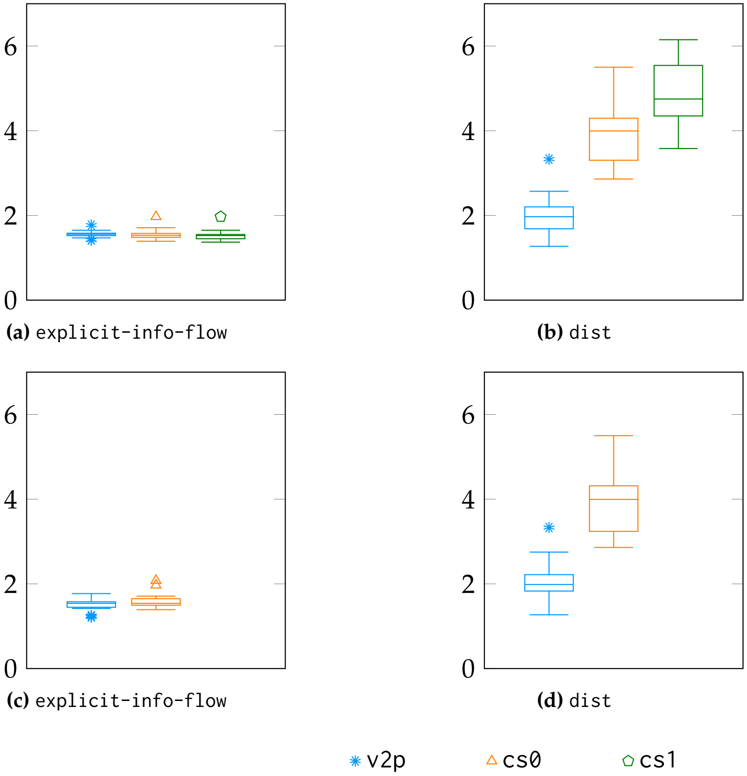
behavior as v2p for the reach and the explicit info flow instances, it can however take up to twice as much time. The dist instance shows a clear deviation – here cs0 generally takes twice as much time and can take up to 3 or 4 times as much time.
The other evaluated callstring-variant cs1 shows a much worse behavior: It runs 5-10 times as long as v2p and may take up to 60 times.

8.3.2.5 Comparison Between Different Instances

Another observation is that, like for the same-level problem solvers, it appears that more complex instances require more effort. This can be seen in Figure 8.8. It shows that the explicit-info-flow instance take roughly twice as much time as reach. This is consistent among all evaluated algorithm. The dist instance shows a more heterogeneous picture. Figure 8.8 shows that at least twice as much effort is required for dist relative to reach. However, this additional effort is much higher for the callstring-based approaches and appears to grow with the depth of the call-strings.

8.3.2.6 Summary

The key takeaways of the runtime evaluation of the data-flow problem solvers are:



(a) `explicit-info-flow`

(b) `dist`

(c) `explicit-info-flow`

(d) `dist`

Figure 8.8: Additional effort of DFA for complex instances relative to `reach`; a/b: only non-large programs, c/d: including large programs

- Joana's hand-optimized slicer performs better than the generic algorithms on the `reach` instance,
- it is possible to compute precise solutions to non-trivial interprocedural data-flow analyses on fairly large PDGs in reasonable time,
- more complex data-flow framework instance generally require more time,
- it is plausible that this additional work grows with the height of an instance's lattice, and
- the call-strings approach performs much worse than the functional approach and is not able to produce results in reasonable time even for very small stack bounds.

	classic		v2p		cs0		cs1	
name	time	err	time	err	time	err	time	err
mixServer	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
ant	<0.01	<0.01	0.02	<0.01	<0.01	<0.01	0.02	<0.01
eVotingMachine	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	<0.01
bibtex2website	<0.01	<0.01	0.02	<0.01	0.01	<0.01	0.05	<0.01
jftp	0.03	<0.01	0.08	<0.01	0.06	<0.01	0.43	0.06
jlex	0.04	<0.01	0.10	<0.01	0.09	<0.01	0.54	<0.01
dacapo-hsqldb	0.03	0.01	0.12	0.05	0.10	<0.01	0.45	0.07
dacapo-xalan	0.06	<0.01	0.25	0.03	0.17	0.01	0.52	0.04
jzip	0.09	<0.01	0.38	<0.01	0.24	<0.01	0.98	0.02
dacapo-fop	0.13	0.01	0.50	0.05	0.57	<0.01	2.37	0.09
dacapo-luindex	0.22	0.04	1.15	0.19	1.15	0.13	7.68	0.29
jasypt-decrypt	0.31	0.01	1.72	0.03	1.87	0.07	8.37	0.36
jasypt-encrypt	0.31	0.02	1.73	0.09	1.80	0.06	8.49	0.25
jasypt-digest	0.33	0.01	1.78	0.08	1.90	0.04	8.50	0.31
dacapo-lusearch	0.39	0.02	1.77	0.11	2.49	0.03	12.99	0.26
lethal	0.64	0.02	4.19	0.14	4.68	0.36	35.66	2.42
dacapo-antlr	1.00	0.06	6.51	0.42	10.63	2.20	115.31	5.51
maven	0.91	0.06	6.90	0.37	6.95	1.55	27.28	1.06
dacapo-eclipse	7.35	0.42	78.70	2.67	158.14	15.23	–	–
hsqldb	12.51	0.65	93.57	12.04	511.14	159.47	–	–
freecs	9.35	0.52	89.34	2.23	281.24	39.69	–	–
javap	4.30	0.32	33.70	2.09	74.81	6.84	–	–

Table 8.8: Runtime performance of the various data-flow solvers for reach instance

	explicit info flow						dist					
name	v2p	err	cs0	err	cs1	err	v2p	err	cs0	err	cs1	err
mixServer	<0.01	<0.01	<0.01	<0.01	0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.04	<0.01
ant	0.03	<0.01	0.01	<0.01	0.03	<0.01	0.02	<0.01	0.03	<0.01	0.08	<0.01
eVotingMachine	0.01	<0.01	<0.01	<0.01	0.02	<0.01	0.01	<0.01	0.01	<0.01	0.05	<0.01
bibtex2website	0.03	<0.01	0.02	<0.01	0.07	<0.01	0.03	<0.01	0.05	<0.01	0.19	0.02
jftp	0.12	<0.01	0.09	<0.01	0.64	0.08	0.13	0.03	0.20	0.03	1.84	0.43
jlex	0.16	<0.01	0.13	<0.01	0.85	0.12	0.14	<0.01	0.30	0.04	2.76	0.31
dacapo-hsqldb	0.18	0.07	0.15	0.01	0.74	0.05	0.22	0.10	0.36	0.05	2.04	0.34
dacapo-xalan	0.39	0.04	0.26	<0.01	0.81	0.13	0.54	0.13	0.66	0.08	2.42	0.32
jzip	0.60	0.03	0.41	0.06	1.40	0.02	0.85	0.11	1.05	0.23	5.31	0.66
dacapo-fop	0.72	0.07	0.87	0.01	3.61	0.11	1.02	0.15	2.48	0.49	11.56	2.00
dacapo-luindex	1.87	0.24	1.90	0.27	11.79	0.16	2.20	0.46	4.87	0.49	44.96	8.49
jasypt-decrypt	2.66	0.05	2.92	0.09	12.76	0.41	4.41	0.69	7.83	0.62	45.65	6.72
jasypt-encrypt	2.63	0.09	2.78	0.05	13.14	0.26	3.85	0.25	8.25	1.45	47.30	6.09
jasypt-digest	2.81	0.06	3.00	0.08	13.27	0.45	4.00	0.32	8.13	1.05	52.00	10.39
dacapo-lusearch	2.74	0.11	3.83	0.12	18.74	0.24	3.52	0.25	10.71	1.14	76.27	8.46
lethal	7.44	0.47	7.13	0.42	70.27	23.16	8.12	1.33	14.42	1.63	164.23	48.68
dacapo-antlr	9.94	0.53	17.57	1.67	157.80	4.18	12.78	1.68	58.45	8.25	709.11	188.56
maven	11.21	0.33	13.66	0.59	40.24	0.70	22.98	4.01	28.70	7.24	120.18	21.15
dacapo-eclipse	96.27	1.47	268.60	11.83	–	–	207.31	23.98	1129.82	680.68	–	–
hsqldb	115.56	8.54	1062.36	218.09	–	–	187.77	11.52	1644.78	463.92	–	–
freecs	111.79	1.90	477.32	78.03	–	–	246.01	24.67	871.36	180.35	–	–
javap	48.46	2.79	112.79	5.93	–	–	62.51	6.03	335.39	57.12	–	–

Table 8.9: Runtime performance of the evaluated data-flow solvers for complex instances

8.4 Precision Evaluation

In my evaluation, I not only measured performance, but also the precision of the considered data-flow analyses. In particular, I practically compared the precision of `cs0` and `cs1`.

It is not obvious how to measure precision of a data-flow analysis. Hence, in subsection 8.4.1, I am going to consider this aspect more closely and describe a general method that allows to practically assess the precision of a given data-flow analysis. In subsection 8.4.2, I will present the results of my precision evaluation and describe how I obtained them. After that, I will discuss the result in subsection 8.4.3.

8.4.1 How to Measure the Precision of Data-Flow Analyses

In program analysis theory, precision is usually a *binary* concept – a program analysis is either precise with respect to a given ideal baseline, or it is not.

However, for practical purposes, it is desirable to perceive precision as *comparative*. With a comparative notion of precision, one can make statements like "analysis A is more precise than analysis B".

In chapter 6, we already encountered results that may be read as pointers to comparative precision statements.

For example, in Corollary 6.33 we saw that the MO$AP_{\mathcal{S}_k}$ is correct with respect to MO$AP_{\mathcal{S}_l}$ if $k \leq l$. In other words, MO$AP_{\mathcal{S}_l}$ is always at least as precise as MO$AP_{\mathcal{S}_l}$ if $k \leq l$. Moreover, it is easy to give examples for which MO$AP_{\mathcal{S}_l}$ yields a result that is strictly more precise than the result for MO$AP_{\mathcal{S}_k}$.

However, it is very challenging if not impossible to theoretically and generally assess the precision of program analyses comparatively[39].

It is therefore more promising to concentrate on practical evaluation that usually evaluates the different analyses on a concrete sample of example programs.

[39] According to Jansen [97], different approaches to interprocedural data-flow analyses are comparable, but only in simple cases.

One possibility to establish a comparative notion of precision is to *quantify* it using some metric that assigns an analysis a number that assesses *how precise* this analysis is. Then, the precision of multiple approaches can be compared using this metric.

In chapter 4, we saw two examples for this. Firstly, in the scope of the SHRIFT approach we compare different points-to analyses of Joana by relating the number of reported information flows with the number that a black box approach would report that just assumes that every sink depends on every source. Secondly, in our work on ifspec, we compare the precision of different information flow tools by relating the number of insecure programs with the number of programs that were classified as insecure by the respective tool.

In the following, I describe a method for practically evaluating the precision of a given data-flow analysis approach. The method is independent of instances and algorithms and only makes a few general assumptions. The basic idea is to compute a metric that indicates how close a given solution of a given data-flow analysis approach is to the respective section of the *MOVP* solution. By computing this number for a multitude of solutions, we get a distribution for the given analysis. Multiple analyses can then be compared with respect to this distribution.

Let $\mathcal{F} = (G, L, F_{\boxtimes}, \rho)$ be a data-flow analysis framework and let $\mathcal{D}$ be a data-flow analysis. Moreover, I fix a node $s \in N$. Analysis $\mathcal{D}$ takes s as input and outputs a function $\mathcal{A}^{(s)} : N \to F_{\boxtimes}$. I want to compare $\mathcal{A}^{(s)}$ with the portion of *MOVP* where the first argument is s. Therefore, I introduce the function $\mathrm{MO}VP^{(s)} : N \to F_{\boxtimes}$ that is defined by

$$\mathrm{MO}VP^{(s)}(t) \overset{def}{=} MOVP(s, t).$$

I assume that, regardless of s, $\mathcal{D}$ only produces $(MOVP, \{s\} \times N)$-correct solutions, i.e. $\forall t \in N.\ \mathcal{A}(t) \geq MOVP(s, t)$. Moreover, I assume that $\mathcal{F}$ allows for practically evaluating *MOVP*, e.g. that it is distributive and allows for an effective execution of Algorithm 9 and Algorithm 13.

A straight-forward way to assess the precision of $\mathcal{A}^{(s)}$ is to evaluate the fraction of t for which $\mathcal{A}^{(s)}(t)$ coincides with $MOVP^{(s)}(t)$. I call this metric the *value precision* and define it as follows:

$$vp(s) \overset{def}{=} \frac{|\{t \in dom(\mathcal{A}^{(s)}) \mid \mathcal{A}^{(s)}(t) = MOVP^{(s)}(t)\}|}{|dom(\mathcal{A}^{(s)})|}.$$

The function vp assumes values between 0 and 1 and measures the coincidence between $\mathcal{A}^{(s)}$ and $MOVP^{(s)}$. A value of 0 means that $\mathcal{A}^{(s)}$ does not coincide at all with $MOVP^{(s)}$, where as a value of 1 means that $vp(s)$ perfectly coincides with $MOVP^{(s)}$. However, one issue of vp is that it does not differentiate between the two main reasons for $\mathcal{A}^{(s)}(t)$ and $MOVP^{(s)}(t)$ to differ. For one, it may be the case that $t \in dom(MOVP^{(s)})$ and $Analysis^{(s)}(t) \neq MOVP^{(s)}(t)$, i.e. that t is a node for which every $(MOVP, \{s\} \times N)$-correct analysis must actually compute a result. The second case is that $t \in dom(\mathcal{A}^{(s)}) \setminus dom(MOVP^{(s)})$, i.e. that $\mathcal{A}$ computes a value for t, although this is not absolutely necessary for a context-sensitive analysis (and therefore imprecise). To distinguish between these two cases, I introduce two additional metrics, namely the *relative slice size*

$$ss(s) \overset{def}{=} \frac{|dom(MOVP^{(s)})|}{|dom(\mathcal{A}^{(s)})|}, \text{ and}$$

the *value precision on the common core*

$$vp_{cc}(s) \overset{def}{=} \frac{|\{t \in dom(MOVP^{(s)}) | \mathcal{A}^{(s)}(t) = MOVP^{(s)}(t)\}|}{|dom(MOVP^{(s)})|}.$$

Both $ss(s)$ and $vp_{cc}(s)$ also assume values between 0 and 1 and larger value reflect more precision – while ss focuses on the slice, vp_{cc} focuses on values. This is reflected by the equation

$$vp(s) = ss(s) \cdot vp_{cc}(s). \tag{8.1}$$

The validity of (8.1) can be seen as follows: From $t \in dom(MOVP^{(s)})$ and $\mathcal{A}^{(s)}(t) = MOVP^{(s)}(t)$, it follows that $t \in dom(\mathcal{A}^{(s)})$. Hence, we can also write vp_{cc} as

$$vp_{cc}(s) = \frac{|\{t \in dom(\mathcal{A}^{(s)}) | \mathcal{A}^{(s)}(t) = MOVP^{(s)}(t)\}|}{|dom(MOVP^{(s)})|},$$

and from this, Equation 8.1 follows by an easy calculation.
With help of (8.1), I can identify two important special cases. For one, if $vp_{cc}(s) = 1$, then $\mathcal{A}^{(s)}$ assumes perfectly context-sensitive values on $dom(MOVP^{(s)})$ but may be too large (if $ss(s) < 1$). Secondly, if $ss(s) = 1$, then $dom(\mathcal{A}^{(s)})$ coincides with $dom(MOVP^{(s)})$ but may contain different results (if $vp_{cc} < 1$).

8.4.2 Results

In my precision evaluation I applied the methodology described in subsection 8.4.1 to obtain a practical precision comparison between the two call-string approaches `cs0` and `cs1`. I considered the programs from Table 8.2 and the three instances `reach`, `explicit-info-flow` and `dist` that I already considered for my performance evaluation. Note that all instances are distributive and can be solved precisely using the functional approach, so that I can use the `v2p` algorithm to provide *MOVP*-solutions. For each of the programs and the instances, I evaluated all three metrics *vp*, *ss* and vp_{cc} for a sample of randomly selected nodes in the respective program's PDG. For the non-large programs, I took 100 nodes, whereas for the large programs, I took 10 nodes.
The distributions of the evaluated metrics are shown in Figure 8.9.

8.4.3 Discussion

In the following, I want to briefly discuss the results that are visualized in Figure 8.9. Generally, the *ss* distributions for `cs0` and `cs1` are very similar. In particular, with respect to the relative slice size, `cs1` offers only a little precision gain in comparison with `cs0`. Notably, for the `reach` instance, `cs0` and `cs1` differ only in their *ss* distributions. The values on the common core coincide completely, so that vp_{cc} is 1 for all measurements. This is indeed no wonder because data-flow solutions for the `reach` instance can assume exactly one value.
On the complex instances `explicit info flow` and `dist`, `cs1` delivers more precise results than `cs0` on the common core. I suspect that the reason for this is that the `dist` instance offers a larger space of possible values with more possibility for the different data-flow analyses to differ. On the other

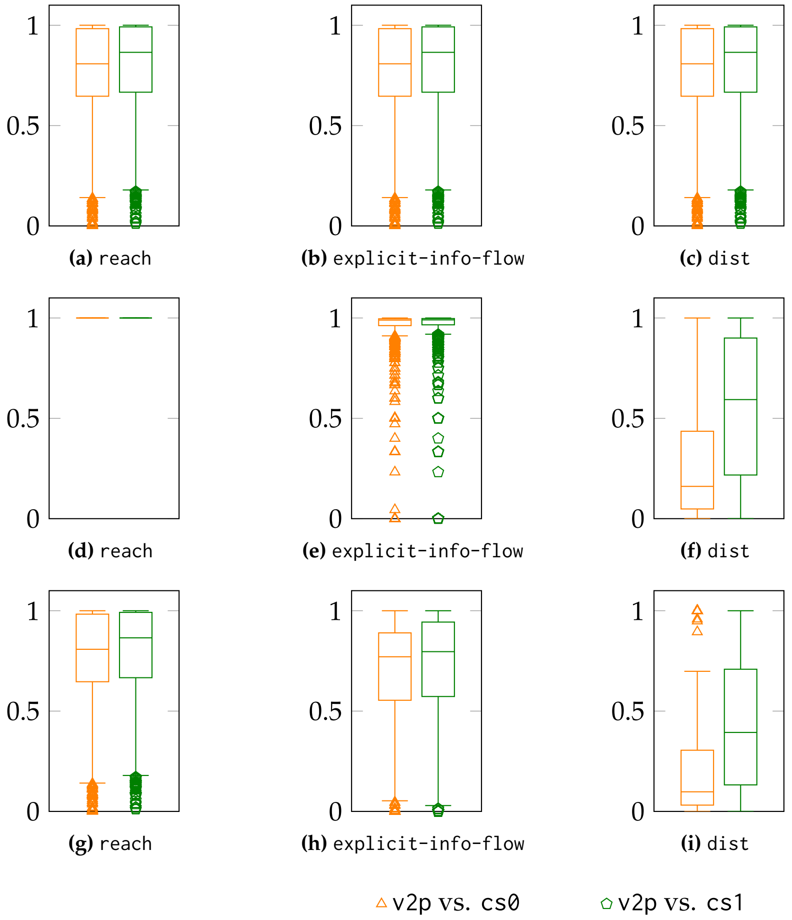


Figure 8.9: Comparison of cs0 and cs1 with respect to precision; a – c: slice sizes, d – f: value precision on common core, g – i: overall value precision

hand, we also see that vp_{cc} for the dist instance offers more variability for cs1 than for cs0.

All in all, we see that, based on this sample, cs0 and cs1 offer similar result with respect to the amount of spurious values. Regarding the computed values on the common core, cs1 tends to deliver a more precise value on the common core, especially for complex instances. It can be imagined that call-string approaches with a higher bound deliver even more precise results. However, with regards to the performance results, the question is whether the precision gain is worth the additional effort.

Looking through the bent-backed tulips,
to see how the other half live.
The Beatles

9 Discussion and Related Work

In this chapter, I critically discuss the approach that I developed in the last chapters and compare it to the existing literature.

In section 9.2, I discuss restrictions of my approach and simplifications that I deliberately applied for the sake of presentation. After that, I look at some of the benefits of my approach and discuss possible improvements in section 9.3. Lastly, in section 9.4, I consider other approaches that could also be used to generalize slicing.

Before I start with the actual discussion, I want to give some general remarks that may help to place the work in this thesis in the right context.

9.1 The Role of Data-Flow Analysis and Slicing in Program Analysis

Schmidt and Steffen [146] propose the view that a multitude of program analyses mainly consists of three steps: Given a program and an operational semantics, the program is first transformed into a model that adequately captures its semantics. This program model is usually some kind of labeled transition system – i.e. a (possibly infinite) graph whose nodes represent the program state and whose edges represent the possible state transitions. The second step consists of an abstraction that abstracts from irrelevant details or makes the model more tractable and still respects the semantics. Lastly, this abstraction of the program model is analyzed with respect to graph-theoretic properties.

If all steps indeed respect the given program semantics, graph theoretic properties of the program model abstraction can indeed be mapped to actual properties of the program.

Following this view, both data-flow analysis and context-sensitive slicing are techniques that operate as third step. That is, control-flow graphs and program dependence graphs are abstractions of underlying program models. One main result of chapters 5–7 is that program dependence graphs and control-flow graphs can be seen as instances of the same general graph model. However, this generalization ignores program semantics. Whether or not the results that generalized data-flow analyses yield can be transferred to actual program properties is out of the scope of this thesis. For data-flow analyses on control-flow graphs, we can state that if they are constructed properly, they can give semantic guarantees. However, such a statement cannot easily be generalized to data-flow analyses on interprocedural graphs, let alone transferred to program dependence graphs. I will consider semantics more closely in subsection 9.2.4.

Hence, within the scope of this work, generalized data-flow analysis is to be understood as an abstract technique for analyzing graphs with respect to their path sets.

9.2 Simplification and Restrictions

In the following, I discuss some aspects that I chose to simplify for the sake of presentation and uniformity. Moreover, I look at some of the restrictions of generalized data-flow analysis on interprocedural graphs and discuss possible ways to lift them.

The first two subsections are dedicated to the simplifications and the last two subsections discuss the restrictions.

9.2.1 Forward Analysis vs. Backward Analysis

Forward analyses consider the propagation of data-flow information along the paths of the given directed graph, in the direction given by the graph's edges. In contrast, backward analyses consider the propagation of data-flow information against the direction given by the graph's edges.

There are numerous program analyses that are naturally expressed as backward data-flow analyses (cf. [130, Figure 2.6]). Another important example of a backward analysis is slicing in its original formulation. As I explained in subsection 3.3.1, program slicing was originally introduced as a technique for focusing on the parts of a program that contribute to the value of a given variable at a given program location. This naturally leads to the notion of *backward slices*. Hence, subsequent work on slicing mainly focused on computing backward slices. Even the summary edge computation was originally presented as a backward analysis [93, 137], although this is not strictly necessary, since the property of same-level path reachability is symmetric.

Although I mainly concentrated on forward analyses, all my considerations apply to backward analyses as well. Adapting my framework to backward analyses would result in algorithms that are even closer to the original slicing and summary edge computation approaches.

9.2.2 Functional Level vs. Ordinary Level and Initial Values

My representation of data-flow analyses uses an ordinary lattice L of possible values and a lattice F that consists of monotone functions on L, contains the identity function and is closed under function composition. The goal function $MOVP$ has values in F – i.e. it assigns a pair (s, t) a function $MOVP(s, t)$ that represents the transformation of data-flow facts along all valid paths from s to t. After $MOVP$ has been computed or approximated, one can apply $MOVP(s, t)$ to some value $l \in L$ to yield the transformed value $MOVP(s, t)(l)$.

Classically, one is not interested in the function $MOVP(s, t)$ but rather the value $MOVP(s, t)(init)$ for a specific element $init \in L$, called *initial information*. This element is traditionally associated with the entry or exit node of the procedure or program to be analyzed. Hence, it is customary to include *init* into a data-flow instance and focus on the computation of data-flow analysis solutions that aim to approximate $MOVP(s, t)(init)$ for $t \in N$. This is also how I introduced classical intra-procedural data-flow analysis in subsubsection 3.2.2.1. If s is not a fixed node, then we can also consider *init* as a function $N \rightarrow L$.

Both the functional and the call-string approach to interprocedural data-flow analysis can in principle be formulated in this way: It is not hard to come up with variations of Constraint System 6.5, Constraint System 7.1, Constraint System 6.2 and the corresponding algorithms and correctness results that employ the initial information. The only component that requires the functional level is the same-level solution. The reason is that we want to use MO$SL$ (or the least same-level solution, respectively) to “fill the gap” between an entry node n_0 and an exit node n_1 (or between a call node and a corresponding return node, respectively), without knowing the value that arrives at n_0. That is, we explicitly need a function that describes the transformation of data-flow facts along same-level paths from n_0 to n_1. I chose to completely stay on the functional level because it is required for the same-level problem and I wanted my presentation to be uniform.

9.2.3 Concurrency

My analysis framework only considers nesting structures that occur in sequential programs. For both control-flow graphs and program dependence graphs, extensions have been proposed to support concurrency. In the following, I want to discuss these extensions briefly.

For control-flow graphs, it has been shown that it is possible to support simple parallelism constructs [151] and even dynamic thread creation [117] and enable a limited yet important class of data-flow analyses.

Program Dependence Graphs have also been extended to support multithreading [110, 86, 65, 66]. As I briefly explained in subsection 4.2.2, additional edges called *interference edges* model data dependencies across thread borders. Multi-threaded PDGs can also be sliced using an *iterated two-phase slicing approach* that was first described by Nanda et al. [129] and later also considered by Hammer [86] and extended by Giffhorn [65]. The basic idea is to employ an additional loop around the two-phase slicer that invokes a two-phase slice each time an interference edge is encountered. A possible generalization of my framework to concurrent PDGs would be to formally characterize the paths that are traversed by the iterated two-phase slicer and then establish a monotone constraint system that characterizes the data-flow along these paths. A worklist algorithm that solves this system could then turn out to be a generalization of Nanda’s iterated two-phase slicer.

Note that for the iterated two-phase slicer, we cannot expect that the iterated two-phase backward slice can be used to verify a non-interference-like property (or a result such as the slicing theorem, respectively). As I explained in subsection 4.2.1, additional dependencies have to be taken into account to obtain such a result. However, I think it is possible to re-formalize e.g. the RLSOD check, which I described in subsection 4.2.1, in such a way that it performs a form of slicing instead of checking, possibly on an extension of the multi-threaded PDG by an additional type of dependency. As for the iterated two-phase slicer, one could establish a monotone constraint system that characterizes the data-flow along the paths that are traversed by the RLSOD slicer.

In summary, I think that it is possible to extend my framework for multi-threaded PDGs. However, such an extension would probably be specific to PDGs and I suspect that the resulting data-flow analysis cannot be unified with data-flow analysis for multi-threaded control-flow graphs as described by Lammich and Müller-Olm et al. [117].

9.2.4 Semantics

Classic data-flow analyses have a strong connection to program semantics. Control-flow graphs can be considered as static approximations of the possible program executions. This connection can be used to formally characterize the program properties that a given data-flow analysis verifies [45].

Program Dependence Graphs can also be connected to program semantics, albeit not that directly: As we saw in subsection 3.4.1, PDGs have been semantically justified in the sense that equivalent programs have isomorphic PDGs and the reachability instance, i.e. PDG-based slicing, has been shown to verify non-interference. However, it is unclear how such results can be extended to other data-flow analyses on PDGs.

In particular, it is not clear what a given data-flow analysis result along the paths of a program's dependence graph tells about the executions of that program. For example, the least distances analysis from subsection 5.4.7 provides purely graph-theoretic information about the structure of the given graph.

Hence, generalized data-flow analyses on interprocedural graphs per se only compute properties of the given graph and additional arguments are necessary to provide the connection to a reference semantics such as program semantics. Nonetheless, I still think that such generalized analyses can be useful. For example, the least distances analysis from subsection 5.4.7 could be extended in such a way that it also constructs a shortest path[40] that can serve as a "simplest witness" for the connectedness of two nodes. Moreover, strong bridges or strong articulation points (as described in subsection 5.4.4) can be computed in order to automatically infer parts of a program dependence graph where it may be promising to increase analysis precision.

9.3 Benefits and Possible Improvements of My Approach

In this section, I discuss possible improvements and benefits of my approach.

9.3.1 Applicability of Existing Extensions and Improvements

Generally, the approaches that I describe in this work are extensions of the two approaches presented by Sharir and Pnueli [154]. As such, they inherit all benefits and drawbacks.

The functional approach yields the most precise result but only works *effectively* for a given data-flow framework instance $\mathcal{F}$ if the lattice of functions of $\mathcal{F}$ satisfies the ascending chain condition and functions can be encoded effectively.

In the literature, we can find several contributions that improve certain aspects of the functional approach and that can also be applied in the context that I consider here.

For example, Reps et al. [136] consider an important class of data-flow problems that can be encoded particularly well. For this class, which consists of data-flow analyses with a finite subset lattice and distributive transformers, it is possible to encode the summary functions in such a way that the whole data-flow analysis can be reduced to graph reachability. Sagiv et al. [145] extend this work to data-flow problems in which the

[40] I suspect that such an algorithm would generally consist of two parts: (a) a two-phase approach that treats the edge functions and same-level information as weights and works analogously to a classic algorithm and (b) a second step that exploits same-level information to iteratively replace summary edges by shortest *same-level paths*.

data-flow facts are mappings from variables to lattice values but the transformers still enjoy distributivity properties.

Moreover, one shortcoming of the functional approach of Sharir and Pnueli is that it does not properly support local variables. Knoop and Steffen [106] present a solution for this: They extend a given data-flow framework instance by a stack component and additional transformers for modelling parameter-passing. This technique works solely on the level of the data-flow framework instance and therefore only needs little adaption of the analysis approach itself.

9.3.2 Benefits of My Framework Compared to Adhoc Approaches

My theory gives a formal characterization of the results of data-flow analyses in terms of information transformers along certain path sets and provides generic algorithms to generate these results. This is beneficial in situations where algorithms on PDGs are considered that can be expressed as data-flow analyses. Having available a general result that only needs to be instantiated to a concrete framework instances eliminates the necessity of a separate correctness argument.

One notable example is Hammer's approach to information flow control, which I considered in subsection 5.4.6. While Hammer notes that his approach to IFC can be expressed as a data-flow analysis, he only applies this fact in the intraprocedural case [86, p. 103]. For the interprocedural case with declassification, he presents adapted versions of the well-known summary edge algorithm [86, Algorithm 8/9] and the two-phase slicer ([86, Algorithm 7]) and gives dedicated correctness arguments (see [86, Theorem 4.10] and [86, Theorem 4.4], respectively).

With my framework, such separate correctness arguments are not necessary: It is only necessary to show that Hammer's approach to IFC with declassification can be performed by computing the *MOVP* solution of an appropriate data-flow analysis (as I did in subsection 5.4.6). Then, e.g., Algorithm 9 and Algorithm 12 can safely be used to compute the least solution, in connection with appropriate checks.

Another example, which I want to discuss at this point, is barrier slicing. I already considered barrier slicing in subsection 5.4.5.

Krinke [111] proposes a two-phase approach with a barrier-specific pre-processing phase to compute context-sensitive barrier slices. The pre-processing phase relies on the existence of summary edges. It starts with the assumption that all summary edges are *blocked*, i.e. that all same-level paths contain nodes from the given barrier. Then, it iteratively unblocks all summary edges for which it can construct a barrier-free same-level path. The following two-phase approach then uses only unblocked summary edges and itself skips the barrier. While the correctness of Krinke's approach is intuitively plausible, he does not prove formally that his approach indeed computes context-sensitive barrier slices. Moreover, Krinke's approach is tailored to the problem of computing barrier slices. It is unclear how his approach needs to be adapted in order to compute slices with other properties. As I pointed out in subsection 5.4.5, barrier slicing can be viewed as an instance of a more general problem, namely the problem of computing all nodes that are reachable using paths of a given regular language. This means that barrier slices can be computed with the help of Algorithm 9 and Algorithm 12, which both come with formally proven correctness properties. Moreover, if we want to consider slices with other regular properties, all that we need to do is change the data-flow framework instance and use the algorithms for the other instance. However, it is worth pointing out that this flexibility comes at a price, at least when using the functional approach: Since summary information is specific to the framework instance, it can only be re-used for problems of the same instance. Moreover, the data-flow framework instance for barrier slicing is dependent on the barrier. Hence, the summary information has to be re-computed if the barrier changes. This problem was also noted by Krinke [111, p. 4].

There are two potential remedies to this drawback of using a generic approach. Firstly, one could use a call-string approach. Call-string approaches are also generic but do not need barrier-specific pre-processing phases. However, as we saw in chapter 8, they are significantly less precise and considerably more costly. Secondly, one could try to use a more elaborate data-flow framework instance. Such an instance would propagate the information "this path skips the following nodes" – i.e. node sets – instead of the binary information "this path skips the given barrier". I suspect that this would result in a more expensive summary information computation phase, since its lattice is more complex, but potentially increases the re-usability for a variety of barriers.

9.4 Alternative Approaches to Generalize Slicing

In this thesis, I consider context-sensitive slicing on program dependence graphs as a form of data-flow analysis on a generalized interprocedural graph model. However, data-flow analysis is not the only technique that can be unified with slicing. In the following subsections, I take a look at three formalisms from the literature for which I suspect that they are also suitable to represent program dependence graphs and context-sensitive slicing.

9.4.1 Pushdown Systems

One alternative approach is to employ *pushdown systems* [36, 90]. Pushdown systems are extensions of finite state machines that are able to represent the control-flow in sequential programs with recursive procedure calls. A *configuration* of a pushdown system consists of a *control state* and a *stack*. The control state may assume finitely many values and the stack consists of a (finite but arbitrarily long) list of *stack symbols* from a finite alphabet. Possible transitions of a pushdown system may alter the control state and manipulate the stack, depending on the stack's top symbol.

An interprocedural (program dependence) graph G can be represented as a pushdown system as follows: The possible control states are the nodes of G, while the stack alphabet consists of the G's call edges. The transitions can then be defined in a similar fashion as the constraints from Constraint System 6.5.

Pushdown systems can be analyzed with respect to their configuration space: For any regular set C of configurations, the set $pre^{\star}(C)$ of configurations that reach C by a sequence of allowed transitions is also regular [36]. In particular, if C is given as a finite automaton, one can use a saturation procedure to construct a finite automaton that accepts $pre^{\star}(C)$. This result can be applied to obtain a context-sensitive slicer that is more flexible than classical two-phase slicing. The slicing criterion does not need to be a set of plain nodes but can also encode regular properties about the possible call stacks. A simple version of such a context-restricted slicer was already considered by Krinke [112].

The analysis of pushdown systems is not restricted to reachability. A pushdown system $\mathcal{P}$ can also be equipped with *weights* and it is possible to compute a form of merge-over-all-paths solution on the configuration transition graph of $\mathcal{P}$ [147]. The structure of these weights is largely similar to the transfer functions considered in data-flow analysis. This idea, which was already noted by Schwoon et al. [147], was further developed by Reps et al. [139], who showed that weighted pushdown systems are general enough to express important special cases of interprocedural data-flow analysis.

9.4.2 Recursive State Machines

Another formalism to represent PDGs, which I want to briefly mention, are *recursive state machines* [13]. Recursive state machines are a model of sequential, imperative, recursive programs and consist of multiple components, which may have multiple entries and exits. Slicing then can be expressed as reachability analysis on recursive state machines. Such an analysis can be performed using a functional approach [13].

9.4.3 Visibly Push-Down Languages

The valid paths considered in this thesis appear to be an example of *visibly push-down languages* [15, 14]. Their key feature is that they are recognized by a class of push-down automata that are restricted in their stack manipulation operations. This restriction is still general enough to be useful in program analysis – yet, visibly push-down languages enjoy nice closure properties. For example, they are closed under intersection (unlike general context-free languages) and union (unlike deterministic context-free languages). Hence, by employing visibly push-down languages, one could define and compute language-restricted slices with respect to properties that are expressible by visibly push-down languages – a generalization of the regular language-restricted slices considered in subsection 5.4.5.

All things must pass.
George Harrison

10 Conclusion

10.1 Summary and Main Theses

In the following, I give a summary of this dissertation and revisit the main theses stated in section 1.3.

10.1.1 Applications to Software Security

Summary Chapter 3 gave a general overview of static analysis techniques and data structures, including data-flow analysis on control-flow graphs and slicing on program dependence graphs. It also mentioned the connection between slicing and information flow control. The last section of chapter 3 described the PDG-based information flow control tool Joana and several analysis techniques for object-oriented languages such as Java. Subsequently, in chapter 4 I reported on the contributions of the programming paradigms group at KIT to the priority program RS^3.
I presented research results of the sub-project "Information Flow Control for Mobile Components" concerning information flow control for concurrent languages. In particular, I described a static PDG-based check to guarantee probabilistic non-interference that was developed in our group within the scope of RS^3.
I also reported on the contributions of our group to two of the three reference scenarios of RS^3, namely "Security In E-Voting" and "Software Security For Mobile Devices". In the former, Joana is combined with a theorem prover to verify cryptographic properties of prototypical electronic voting systems, and in the latter, Joana provides static checks of user-defined security policies in the server component of a secure app store.

Lastly, I described several collaborations within RS^3. In these collaborations, we demonstrate that Joana can be used to increase the precision and performance of dynamic usage control and to simplify the security verification obligations in component-based systems. A third cooperation is concerned with the development of RIFL, a machine-readable language dedicated to the specification of security properties. RIFL specifications can not only be read by machines, but also be checked by information flow analysis tools. I contributed a Joana-back-end for RIFL. An application of RIFL is ifspec, a benchmark for information flow analysis tools.

Main Thesis 1: PDG-based information flow control is useful, practically applicable and relevant. As shown in chapter 4, applications of Joana range from highly relevant scenarios such as mobile security and electronic voting systems to the support of both theorem provers and dynamic usage control systems.

10.1.2 Systematic Approaches to Advanced Information Flow Analysis

Summary In chapter 5, I developed a general notion of valid paths and described a graph-based model and a data-flow framework that incorporates both interprocedural data-flow analysis on control-flow graphs and PDG-based slicing as special cases. I discussed several examples from the literature that can be expressed systematically within this framework.
Chapter 6 demonstrated that instances of the general framework developed in chapter 5 can indeed be solved with the two classical approaches of Sharir and Pnueli [154] – the functional approach and the call-string approach, respectively. I specified monotone constraint systems whose least solutions can be used to compute an over-approximation of the *merge-over-all-valid-paths (*MO*VP)* solution given by the framework instance. Similar to classic results, I showed that the functional approach and the unrestricted call-string approach both fully characterize the MO*VP* solution. For the call-string approach, I gave a sufficient criterion under which it yields a correct over-approximation of MO*VP* using a possibly finite constraint system. I showed that this criterion is in particular satisfied for call-strings whose length is at most k.

In chapter 7, I showed how the constraint systems developed in chapter 6 can be solved algorithmically. Here, I combined a classical worklist-based solving algorithm with a reachability analysis that explores the *relevant core* of the given constraint system. Given a set of variables to start with, only the constraints that the initial variables may influence are solved, provided that the initial variables satisfy a regularity condition. Roughly speaking, this can be imagined as computing a *(forward) slice of the given constraint system*. I instantiated the resulting algorithm multiple times to obtain solving algorithms for the constraint systems of the functional and the call-string approach. Using this method, I showed that the functional approach can be performed analogously to the methods proposed by Horwitz et al. [93, 137] for context-sensitive slicing: First, the same-level problem needs to be solved. This can be understood as a generalization of the summary edge computation. After that, a two-phase algorithm can be used to solve the actual problem. I also showed that the call-string approach can be performed using an appropriate instance of the general algorithm.

Within the scope of this thesis, I not only developed a general framework and its solution approaches theoretically, but also implemented it in Joana and evaluated it to demonstrate that it is practically feasible. In chapter 8, I presented my implementation and discussed some of the practical choices I made. In addition, I discussed the methods and results of my evaluation. Last but not least, in chapter 9 I discussed my approach and put it into the context of related and similar work. I pointed out the restrictions and possible improvements and extensions. Moreover, I discussed other formalisms that also appear to be a natural generalization of context-sensitive slicing.

Main Thesis 2: Data-flow analysis can be systematically applied to program dependence graphs. The theory that I developed in chapters 5–7 proposes to view context-sensitive slicing as a special case of a generalized version of the classical technique of interprocedural data-flow analysis. Particularly, this applies to earlier PDG-based approaches such as Hammer's IFC [86, 87] and Krinke's barrier slicing [111, 110].

Hence, PDG-based approaches can profit from the advantages of a rich, generic toolkit for systematically deriving sophisticated analyses: If a given problem on a PDG can be expressed as a data-flow analysis instance, the

framework provides generic and re-usable solution algorithms with general correctness guarantees that give formal descriptions of the solutions. This also includes correctness arguments such as the ones given by Hammer and Krinke for their respective problems.
Moreover, I demonstrated that generalized data-flow problems can be described with both a functional approach and a call-string approach. The resulting constraint systems can then be solved with a general solution algorithm that integrates a classical worklist algorithm with a reachability analysis. The instantiations of this algorithm for the functional approach can be modified in a way such that they resemble a generalization of the summary edge computation and the two-phase approach known for context-sensitive slicing [93, 137], respectively.

Main Thesis 3: Data-flow analysis on PDGs can be practically conducted. My evaluation demonstrates that it is also practically feasible to solve complex data-flow analysis problems on program dependence graphs. Moreover, it shows that the functional approach outperforms the call-string approach with respect to both performance and precision. A side product of the precision evaluation is the formal description of an instance- and approach-independent method for the precision evaluation of data-flow analyses. Future work can use this method to ensure comparability.

10.2 Future Work

I want to close this thesis by giving an outlook on future work. I restrict this outlook to the area of generalized data-flow analysis on interprocedural graphs like program dependence graphs, to which I consider this thesis as a starting point.

10.2.1 Approximation of Same-Level Information

One characteristic of the functional approach is that the two-phase algorithm that solves the actual constraint system relies on a same-level solution. The most precise same-level solution can be computed by an algorithm like Algorithm 9 or Algorithm 17. The evaluation in chapter 8 shows that this can be quite expensive. However, my theory provides a remedy for the case that precision can be sacrificed. In fact, the correctness properties in section 7.3 state that the algorithms for computing ascending and non-ascending-path solutions still produce correct results if they are fed with a same-level solution that is not as precise as possible. This opens up a possibility for less precise yet faster ways to produce correct same-level solutions.

An extreme example would be to simply use a same-level solution whose value is always the greatest element $\top$. This is surely SL-correct because of the maximality property of $\top$. More generally, we could exploit domain-specific knowledge about the data-flow framework instance and use a value for which we know that it is a universal upper bound of MO$SL$. For example, consider the `dist` instance that was described in subsection 5.4.7 and evaluated in chapter 8: According to its definition we have $\top = 0$. Using such a simplified same-level information would amount to a distance calculation that would assume that entries and exits of the same procedure are connected by a path with length 0. Such a distance calculation would result in statements like "nodes s and t are connected by a valid path that has a length of at least n", which are still correct if the length of same-level paths is coarsely underestimated[41].

Other approximations of same-level information are imaginable. One variant, which I considered more closely within the scope of this thesis but did not evaluate practically, uses a $\mathbb{N}$-indexed family of constraint systems C_n, so that for all $n \in \mathbb{N}$ we have

(a) $\quad lfp(C_n) \geq MOSL$

(b) $\quad lfp(C_{n+1}) \leq lfp(C_n)$

Property (a) ensures that we can correctly use any $lfp(C_n)$ as same-level information, while property (b) entails that we can get more precision by just increasing n. A special case of this scheme uses constraint systems C_n that are defined in a similar fashion as Constraint System 6.1, but have a different SL-SOL-(III)-clause: Instead of recursively relying on the solution characterized by the constraint system itself, it consults $lfp(C_k)$

[41] As I already briefly mentioned in subsection 5.4.7, Krinke [110] also considered distances in system dependence graphs and since he does not provide details about how he computed distances of same-level paths, I firmly suspect that he assigned summary edges a value of 1 (like any other edge), which also is a very coarse underestimation.

for some $k < n$ as a helper solution for same-level information[42]. Property (a) ensures that this indeed characterizes a solution that is correct with respect to MO$SL$. Every $lfp(C_n)$ can then be computed by subsequently computing $lfp(C_0), \dots, lfp(C_{n-1})$ and then, finally, computing $lfp(C_n)$.

10.2.2 Further Exploration of Stack Spaces and *MOVP*-Correct Abstractions

In chapter 6, I introduced stack spaces as an abstract structure for representing call stacks. The constraint system for the call-string approach is parameterized with a given stack space. This enables me to not only consider one call-string approach but multiple concrete instances that differ only in the stack space parameter. I also provide *stack abstractions* as a tool to relate two stack spaces. With the help of stack abstractions, I state a sufficient criterion for when the call-string approach with respect to a given stack space leads to a *MOVP*-correct solution. This criterion requires that there has to be a stack abstraction from the stack space $\mathcal{S}_\infty$ of unbounded stacks to the given stack space. Examples for the satisfiability of this criterion, and hence for stack spaces that lead to *MOVP*-correct solutions, are the k-bounded stack spaces $\mathcal{S}_k$.
There are two open questions in this context:

1. Is the criterion necessary for obtaining a *MOVP*-correct solution?

2. Are there, apart from the $\mathcal{S}_k$, other stack spaces that lead to *MOVP*-correct solutions?

If these two questions can be answered, we could either be assured that there are no other stack spaces that lead to *MOVP*-correct solutions, or other appropriate stack spaces could be found that offer a better compromise between precision and performance.
Regarding the first question, stack abstractions as introduced in this work have the restriction that they do not change the alphabet. It may be sensible to explore whether and how this restriction can be lifted. If it is

[42] This idea can indeed be used to construct such a family $(C_n)_{n \in \mathbb{N}}$ of constraint systems from Constraint System 6.1 as follows: C_0 demands $X(s,t) \geq \top$ for all $s, t \in N$, and each C_n is a copy of Constraint System 6.1 where the SL-SOL-(III)-clause is modified such that the right-hand side relies on $lfp(C_{n-1})$ as a helper solution for same-level information.

possible to obtain stack spaces with $MOVP$-correct solutions from $\mathcal{S}_\infty$ via such generalized stack abstractions, then it may also be possible to obtain $MOVP$-correct stack spaces for which there is no ordinary stack abstraction, so that the first question would have to be answered negatively.
I suspect that the answer to the second question is that there are indeed stack spaces other than $\mathcal{S}_k$ that satisfy the criterion. It seems possible that there are stack abstractions that do not apply the same bound to all stacks but crop each stack depending on their content. More formally, given a function $d : E_{call}^{\star} \rightarrow \mathbb{N}$, one can define

$$\alpha(\sigma) = \sigma^{\leq d(\sigma)}$$
$$\gamma(\sigma) = \sigma,$$

which should be a stack abstraction. Note that d has to be bounded so that the resulting constraint system is finite.

10.2.3 Relation Between the Results of Data-Flow Analysis on PDGs and Program Semantics

An important special case of my general data-flow framework is data-flow analysis on PDGs. All the examples that are described in this thesis are inherently graph-theoretic, i.e. they do not allow for conclusions about semantic properties of the program represented by the PDG. From my point of view, this gap needs to be examined further. Some interesting questions in this area include:

- To what extent can data-flow analyses on PDGs be generated systematically from a given program's semantics (as discussed in subsection 9.2.4)?
- What kind of properties can be examined or verified this way? To what extent can a connection between the valid paths in a PDG and the set of program executions be established?
- Or, respectively, do we need some other kind of program semantics artifact that can serve as basis for a set of valid PDG paths?
- How does data-flow analysis on PDGs have to be adapted to make such a connection possible?

10.2.4 Generalization of Chopping

Chopping [96] was introduced as a generalization of slicing to enable the extraction of more focused program parts. A chop has two nodes s and t as parameter and is defined as the set of nodes that lie on valid paths between s and t. Like for slicing, approaches for the computation of context-sensitive chops have been proposed [138].
I think that it is possible to generalize chopping in a way that is similar to how slicing was generalized in this thesis. The objective function of such a generalization would take not two arguments, like $MOVP$, but three arguments s, t and n. Its value $MOVP_{ch}(s,t,n)$ could be defined as the merge-over-all-valid paths from s to t that also contain n. Analogously to $MOVP$, one could try to approximate $MOVP_{ch}$ by means of monotone constraint systems. I suspect that this can be done both with a functional and a call-string approach. A benefit of such an analysis would be significantly more detailed results, and hence more information than for the slicing variant. For example, for the `dist` instance, one could yield statements like "all paths from s to t that pass n have a length of at least k" – that is, with the additional argument, one would get a whole spectrum of results instead of just one.

10.2.5 Extensions to Concurrent Programs

As already discussed in chapter 9, the framework that I develop in this thesis is restricted to sequential constructs. Hence, future work should explore how this restriction can be lifted. Possible ideas for PDGs, which I considered more closely in subsection 9.2.3, include (a) formalizing and generalizing slicers for concurrent PDGs, like for example the iterated two-phase slicer, and (b) examining and generalizing the constraint system that are solved by IFC checkers on multi-threaded PDGs such as the RLSOD approach [31].

10.2.6 Exploration of Other Generalizations of Context-Sensitive Slicing

As already pointed out in subsection 9.4.1, other formalisms than data-flow analysis could be considered as possible generalizations of context-sensitive slicing. For example, Push-Down Systems enable slicers that allow for significantly more flexible queries. Such an approach could also be generalized to an analysis that not only computes a slice, but also computes data-flow analysis results (or weights, respectively).

Bibliography

[1] International Data Corporation (IDC). *Smartphone Market Share (by Operating System)*. version of 2021-03-19. URL: https://web.archive.org/web/20200319171549/https://www.idc.com/promo/smartphone-market-share/os (visited on 09/22/2021).

[2] Reliably Secure Software Systems (RS^3) – DFG Priority Programme 1496. *Original Proposal (German)*. version of 2020-02-28. URL: https://web.archive.org/web/20200228202215/http://www.reliably-secure-software-systems.de/About/Proposal/original (visited on 09/22/2021).

[3] Reliably Secure Software Systems (RS^3) – DFG Priority Programme 1496. *Projects Sorted by Reference Scenarios*. version of 2020-04-22. URL: https://web.archive.org/web/20200422154911/http://www.reliably-secure-software-systems.de/ReferenceScenarios/Projects (visited on 09/22/2021).

[4] Reliably Secure Software Systems (RS^3) – DFG Priority Programme 1496. *Proposal*. version of 2020-02-28. URL: https://web.archive.org/web/20200228202112/http://www.reliably-secure-software-systems.de/About/Proposal (visited on 09/22/2021).

[5] Reliably Secure Software Systems (RS^3) – DFG Priority Programme 1496. *Security in E-Voting*. version of 2020-02-28. URL: https://web.archive.org/web/20200228205933/http://www.reliably-secure-software-systems.de/EVoting (visited on 09/22/2021).

[6] Reliably Secure Software Systems (RS^3) – DFG Priority Programme 1496. *The Priority Programme at a Glance*. version of 2020-02-28. URL: https://web.archive.org/web/20200228201946/http://

www.reliably-secure-software-systems.de/About (visited on 09/22/2021).

[7] Peter Aczel. "An Introduction to Inductive Definitions". In: *HANDBOOK OF MATHEMATICAL LOGIC*. Vol. 90. Studies in Logic and the Foundations of Mathematics. 1977, pp. 739–782. DOI: 10.1016/S0049-237X(08)71120-0.

[8] Ole Agesen. "The Cartesian Product Algorithm". In: *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*. 1995, pp. 2–26. DOI: 10.1007/3-540-49538-X_2.

[9] Gagan Agrawal and Liang Guo. "Evaluating Explicitly Context-Sensitive Program Slicing". In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '01. 2001, pp. 6–12. DOI: 10.1145/379605.379630.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Second Edition. Addison-Wesley, 2007. ISBN: 0-321-48681-1. URL: http://www.worldcat.org/oclc/179803237.

[11] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book. From Theory to Practice*. Cham: Springer, 2016. DOI: 10.1007/978-3-319-49812-6.

[12] Frances E. Allen. "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*. 1970, pp. 1–19. DOI: 10.1145/800028.808479.

[13] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. "Analysis of Recursive State Machines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27 (4 July 2005), pp. 786–818. ISSN: 0164-0925. DOI: 10.1145/1075382.1075387.

[14] Rajeev Alur and P. Madhusudan. "Adding Nesting Structure to Words". In: *Developments in Language Theory*. 2006, pp. 1–13. DOI: 10.1007/11779148_1.

[15] Rajeev Alur and P. Madhusudan. "Visibly Pushdown Languages". In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*. 2004, pp. 202–211. DOI: 10.1145/1007352.1007390.

[16] Lars Ole Andersen. "Program analysis and specialization for the C programming language". PhD thesis. University of Copenhagen, 1994.

[17] Jason Andress. *The Basics of Information Security*. Second Edition. Syngress, 2014. DOI: 10.1016/C2013-0-18642-4.

[18] AppBrain. *Number of Android apps on Google Play*. version of 2020-03-19. URL: http://web.archive.org/web/20200319174734/https://www.appbrain.com/stats/number-of-android-apps (visited on 09/22/2021).

[19] Martin Armbruster. "Eine Sprache zur Spezifikation von Lebenszyklen framework-basierter Anwendungen". German. bachelor's thesis. IPD Snelting at Karlsruhe Institute of Technology (KIT), 2018.

[20] Steven Arzt, Alexandre Bartel, Richard Gay, Steffen Lortz, Enrico Lovat, Heiko Mantel, Martin Mohr, Benedikt Nordhoff, Matthias Perner, Siegfried Rasthofer, David Schneider, Gregor Snelting, Artem Starostin, and Alexandra Weber. "Software Security for Mobile Devices". Poster at the 2015 IEEE Symposium on Security and Privacy (S&P). URL: https://www.ieee-security.org/TC/SP2015/posters/paper_13.pdf (visited on 09/21/2021).

[21] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. 2014, pp. 259–269. DOI: 10.1145/2594291.2594299.

[22] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. "Termination-Insensitive Noninterference Leaks More Than Just a Bit". In: *Computer Security - ESORICS 2008*. Ed. by Sushil Jajodia and Javier Lopez, pp. 333–348. DOI: 10.1007/978-3-540-88313-5_22.

[23] David F. Bacon and Peter F. Sweeney. “Fast Static Analysis of C++ Virtual Function Calls”. In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’96. 1996, pp. 324–341. DOI: `10.1145/236337.236371`.

[24] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, 2017. URL: `https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf` (visited on 09/22/2021).

[25] Thomas Bauereiß, Simon Greiner, Mihai Herda, Michael Kirsten, Ximeng Li, Heiko Mantel, Martin Mohr, Matthias Perner, David Schneider, and Markus Tasch. *RIFL 1.1: A Common Specification Language for Information-Flow Requirements*. Tech. rep. TUD-CS-2017-0225. TU Darmstadt, 2017.

[26] Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning. “Combining Graph-Based and Deduction-Based Information-Flow Analysis”. In: *Proceedings of the 5th Workshop on Hot Issues in Security Principles and Trust — Hotspot 2017*. 2017, pp. 6–25. URL: `%5Curl%7Bhttps://web.archive.org/web/20210917164048/https://hotspot2017.sec.uni-stuttgart.de/proceedings.pdf%7D` (visited on 09/17/2021).

[27] Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. “Information Flow in Object-Oriented Software”. In: *Logic-Based Program Synthesis and Transformation*. 2014, pp. 19–37. DOI: `10.1007/978-3-319-14125-1_2`.

[28] Rudolf Berghammer. *Ordnungen und Verbände*. Springer Fachmedien Wiesbaden, 2013. DOI: `10.1007/978-3-658-02711-7`.

[29] Elliot Joel Berk and C. Scott Ananian. *JLex: A Lexical Analyzer Generator for* JAVA *(TM)*. version of 2020-09-09. URL: `https://web.archive.org/web/20200909164142/https://www.cs.princeton.edu/~appel/modern/java/JLex/` (visited on 09/22/2021).

[30] Elliot Joel Berk and C. Scott Ananian. *JLex: A Lexical Analyzer Generator for* JAVA *(TM) – Source Code of Main Class in Version 1.2.6*. version of 2019-12-21. URL: `https://web.archive.org/web/20191221224412/https://www.cs.princeton.edu/~appel/modern/java/JLex/Archive/1.2.6/Main.java` (visited on 09/22/2021).

[31] Simon Bischof, Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. "Low-Deterministic Security for Low-Deterministic Programs". In: *Journal of Computer Security* 26 (3 Apr. 2018), pp. 335–336. DOI: `10.3233/JCS-17984`.

[32] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. 2006, pp. 169–190. DOI: `10.1145/1167473.1167488`.

[33] Tobias Blaschke. "Automatische Modellierung des Lebenszyklus von Android-Anwendungen". German. diploma thesis. IPD Snelting at Karlsruhe Institute of Technology (KIT), 2014.

[34] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. "ABCD: Eliminating Array Bounds Checks on Demand". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. 2000, pp. 321–333. DOI: `10.1145/349299.349342`.

[35] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. "Refining Data Flow Information Using Infeasible Paths". In: *Software Engineering — ESEC/FSE'97*. 1997, pp. 361–377. DOI: `10.1007/3-540-63531-9_25`.

[36] Ahmed Bouajjani, Javier Esparza, and Oded Maler. "Reachability analysis of pushdown automata: Application to model-checking". In: *CONCUR '97: Concurrency Theory*. 1997, pp. 135–150. DOI: `10.1007/3-540-63141-0_10`.

[37] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. "Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems". In: *CONCUR 2005 – Concurrency Theory*. 2005, pp. 473–487. DOI: `10.1007/11539452_36`.

[38] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. "Simple and Efficient Construction of Static Single Assignment Form". In: *Compiler Construction*. 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6.

[39] Joachim Breitner. "Lazy Evaluation: From natural semantics to a machine-checked compiler transformation". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2016. DOI: 10.5445/IR/1000054251.

[40] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. "On Improvements Of Low-Deterministic Security". In: *Principles of Security and Trust. 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 68–88. DOI: 10.1007/978-3-662-49635-0_4.

[41] Daniel Bruns, Huy Quoc Do, Simon Greiner, Mihai Herda, Martin Mohr, Enrico Scapin, Tomasz Truderung, Bernhard Beckert, Ralf Küsters, Heiko Mantel, and Richard Gay. "Security in E-Voting". Poster at the 2015 IEEE Symposium on Security and Privacy (S&P). URL: https://www.ieee-security.org/TC/SP2015/posters/paper_10.pdf (visited on 09/21/2021).

[42] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. "Verified Construction of Static Single Assignment Form". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. 2016, pp. 67–76. DOI: 10.1145/2892208.2892211.

[43] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[44] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Third Edition. MIT Press. ISBN: 978-0-262-03384-8.

[45] P. Cousot. "Semantic Foundations of Program Analysis". In: *Program Flow Analysis: Theory and Applications*. Ed. by S.S. Muchnick and N.D. Jones. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981. Chap. 10, pp. 303–342.

[46] P. Cousot and R. Cousot. “Systematic design of program analysis frameworks”. In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas: ACM Press, New York, NY, 1979, pp. 269–282.

[47] 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”. In: *ACM Trans. Program. Lang. Syst.* 13 (4 Oct. 1991), pp. 451–490. DOI: 10.1145/115372.115320.

[48] Daniel Jackson and Eugene J. Rollins. “A New Model of Program Dependences for Reverse Engineering”. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. 1994, pp. 2–10. DOI: 10.1145/193173.195281.

[49] Manuvir Das, Sorin Lerner, and Mark Seigle. “ESP: Path-Sensitive Program Verification in Polynomial Time”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. 2002, pp. 57–68. DOI: 10.1145/512529.512538.

[50] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Second Edition. Cambridge University Press, 2002. ISBN: 978-0-521-78451-1.

[51] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. “A Practical Interprocedural Dominance Algorithm”. In: *ACM Transactions on Programming Languages and Systems* 29 (4 Aug. 2007). DOI: 10.1145/1255450.1255452.

[52] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis”. In: *ECOOP’95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*. 1995, pp. 77–101. DOI: 10.1007/3-540-49538-X_5.

[53] Dorothy E. Denning and Peter J. Denning. “Certification of Programs for Secure Information Flow”. In: *Commun. ACM* 20 (7 July 1977), pp. 504–513. DOI: 10.1145/359636.359712.

[54] Dinakar Dhurjati, Manuvir Das, and Yue Yang. “Path-Sensitive Dataflow Analysis with Iterative Refinement”. In: *Static Analysis*. 2006, pp. 425–442. DOI: 10.1007/11823230_27.

[55] Reinhard Diestel. “The Basics”. In: *Graph Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 1–34. ISBN: 978-3-662-53622-3. DOI: 10.1007/978-3-662-53622-3_1.

[56] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1 (1 Dec. 1959), pp. 269–271. DOI: 10.1007/BF01386390.

[57] Sarah Ereth, Heiko Mantel, and Matthias Perner. *Towards a Common Specification Language for Information-Flow Security in RS^3 and Beyond: RIFL 1.0 - The Language*. Tech. rep. TUD-CS-2014-0115. TU Darmstadt, 2014. URL: http://www.mais.informatik.tu-darmstadt.de/WebBib/papers/2014/RIFL1.0-TechnicalReport-Revision1.pdf (visited on 09/22/2021).

[58] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9 (3 July 1987), pp. 319–349. DOI: 10.1145/24039.24041.

[59] L.R. Foulds. *Graph Theory Applications*. Universitext. Springer New York, 2012. ISBN: 9781461209331. DOI: 10.1007/978-1-4612-0933-1.

[60] The Apache Software Foundation. *Commons CLI*. URL: http://commons.apache.org/proper/commons-cli/ (visited on 09/22/2021).

[61] Christian Fritz, Steven Arzt, and Siegfried Rasthofer. *DroidBench*. URL: https://github.com/secure-software-engineering/DroidBench (visited on 09/22/2021).

[62] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. CS-TR-4991. University of Maryland, Department of Computer Science, 2009. URL: https://web.archive.org/web/20210225122912/https://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf (visited on 09/22/2021).

[63] Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. "Join-Lock-Sensitive Forward Reachability Analysis for Concurrent Programs with Dynamic Process Creation". In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'11. 2011, pp. 199–213. DOI: 10.1007/978-3-642-18275-4_15.

[64] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '07. 2007, pp. 57–76. DOI: 10.1145/1297027.1297033.

[65] Dennis Giffhorn. "Slicing of Concurrent Programs and its Application to Information Flow Control". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2012.

[66] Dennis Giffhorn and Gregor Snelting. "A new algorithm for low-deterministic security". In: *International Journal of Information Security* 14 (3 June 2015), pp. 263–287. DOI: 10.1007/s10207-014-0257-6.

[67] Stephan Gocht. *Detectable*. version of 2020-03-18. URL: http://web.archive.org/web/20200318171149/https://github.com/StephanGocht/WALA/blob/672876c5951623590e135c601f3d899ae6150909/com.ibm.wala.core.testdata/src/arraybounds/Detectable.java (visited on 09/22/2021).

[68] Stephan Gocht. *NotDetectable*. version of 2020-03-18. URL: http://web.archive.org/web/20200318171525/https://github.com/StephanGocht/WALA/blob/672876c5951623590e135c601f3d899ae6150909/com.ibm.wala.core.testdata/src/arraybounds/NotDetectable.java (visited on 09/22/2021).

[69] Stephan Gocht. *Pull Request #89: Adding array index out of bound analysis and exception analysis.* version of 2020-03-18. URL: http://web.archive.org/web/20200318170158/https://github.com/wala/WALA/pull/89 (visited on 09/22/2021).

[70] J. A. Goguen and J. Meseguer. "Unwinding and Inference Control". In: *1984 IEEE Symposium on Security and Privacy*. 1984, pp. 75–85. DOI: 10.1109/SP.1984.10019.

[71] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[72] Google. *Android API Guide*. URL: https://developer.android.com/guide/components/fundamentals?hl=en (visited on 09/22/2021).

[73] Google. *Android API Guide, Intents and Intent Filters*. URL: https://developer.android.com/guide/components/intents-filters.html (visited on 09/22/2021).

[74] Google. *Understand the Activity Lifecycle*. URL: https://developer.android.com/guide/components/activities/activity-lifecycle (visited on 09/23/2021).

[75] Anjana Gosain and Ganga Sharma. "A Survey of Dynamic Program Analysis Techniques and Tools". In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. 2015, pp. 113–122. DOI: 10.1007/978-3-319-11933-5_13.

[76] Anjana Gosain and Ganga Sharma. "Static Analysis: A Survey of Techniques and Tools". In: *Intelligent Computing and Applications*. 2015, pp. 581–591.

[77] Juergen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. "Sicherheitsanalyse mit JOANA". In: *Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit*. Bonn: Gesellschaft für Informatik e.V., 2016, pp. 11–21. DOI: 20.500.12116/869.

[78] Jürgen Graf. "Information Flow Control with System Dependence Graphs - Improving Modularity, Scalability and Precision for Object Oriented Languages". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2016. DOI: 10.5445/IR/1000068211.

[79] Jürgen Graf, Martin Hecker, and Martin Mohr. "Using JOANA for Information Flow Control in Java Programs - A Practical Guide". In: *Software Engineering 2013 - Workshopband*. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 123–138.

[80] Jürgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. "Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks". https://pp.ipd.kit.edu/uploads/publikationen/pdgwithdpn2013id.pdf. 1st International Workshop on Interference and Dependence – no formal proceedings. Jan. 2013.

[81] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. "Checking Applications using Security APIs with JOANA". https://pp.ipd.kit.edu/uploads/publikationen/joana15asa.pdf. 8th International Workshop on Analysis of Security APIs – http://www.dsi.unive.it/~focardi/ASA8/ – no formal proceedings. Jan. 2015.

[82] Susan L. Graham and Mark N. Wegman. "A Fast and Usually Linear Algorithm for Global Flow Analysis". In: *Journal of the ACM* 23 (1 Jan. 1976), pp. 172–202. DOI: 10.1145/321921.321939.

[83] Simon Greiner, Martin Mohr, and Bernhard Beckert. "Modular Verification of Information Flow Security in Component-Based Systems". In: *Software Engineering and Formal Methods*. 2017, pp. 300–315. DOI: 10.1007/978-3-319-66197-1_19.

[84] David Grove and Craig Chambers. "A Framework for Call Graph Construction Algorithms". In: *ACM Transactions on Programming Languages and Systems* 23 (6 Nov. 2001), pp. 685–746. DOI: 10.1145/506315.506316.

[85] Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. "A Uniform Information-Flow Security Benchmark Suite for Source Code and Bytecode". In: *Secure IT Systems*. 2018, pp. 437–453. DOI: 10.1007/978-3-030-03638-6_27.

[86] Christian Hammer. "Information flow control for java : a comprehensive approach based on path conditions in dependence Graphs". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2009. DOI: 10.5445/KSP/1000012049.

[87] Christian Hammer and Gregor Snelting. "Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs". In: *International Journal of Information Security* 8 (6 Dec. 2009), pp. 399–422. DOI: 10.1007/s10207-009-0086-1.

[88] Michael Hind. “Pointer Analysis: Haven’t We Solved This Problem Yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’01. 2001, pp. 54–61. DOI: 10.1145/379605.379665.

[89] L. Howard Holley and Barry K. Rosen. “Qualified Data Flow Problems”. In: *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’80. 1980, pp. 68–82. DOI: 10.1145/567446.567454.

[90] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd Edition. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0-321-45536-3.

[91] S. Horwitz, J. Prins, and T. Reps. “On the Adequacy of Program Dependence Graphs for Representing Programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1988, pp. 146–157. DOI: 10.1145/73560.73573.

[92] S. Horwitz, T. Reps, and D. Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. 1988, pp. 35–46. DOI: 10.1145/53990.53994.

[93] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (1 Jan. 1990), pp. 26–60. DOI: 10.1145/77606.77608.

[94] *Introducing JSON*. URL: https://www.json.org/json-en.html (visited on 09/22/2021).

[95] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. “Finding strong bridges and strong articulation points in linear time”. In: *Theoretical Computer Science* 447 (Aug. 2012), pp. 74–84. DOI: 10.1016/j.tcs.2011.11.011.

[96] Daniel Jackson and Eugene J. Rollins. “A New Model of Program Dependences for Reverse Engineering”. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT ’94. 1994, pp. 2–10. DOI: 10.1145/193173.195281.

[97] Dorothea Jansen. “Über die Präzision interprozeduraler Analysen”. German. diploma thesis. University of Münster, 2011. URL: http://arxiv.org/abs/1703.10179.

[98] Dorothea Jansen, Martin Mohr, Irene Thesing, Anton Reis, Maria Schatz, Philipp Claves, and Sezar Jarrous. LETHAL. URL: https://github.com/mohrm/LETHAL (visited on 09/22/2021).

[99] JASYPT. URL: http://www.jasypt.org/ (visited on 09/22/2021).

[100] John B. Kam and Jeffrey D. Ullman. “Global Data Flow Analysis and Iterative Algorithms”. In: *Journal of the ACM* 23 (1 Jan. 1976), pp. 158–171. DOI: 10.1145/321921.321938.

[101] John B. Kam and Jeffrey D. Ullman. “Monotone data flow analysis frameworks”. In: *Acta Informatica* 7 (3 Sept. 1977). DOI: 10.1007/BF00290339.

[102] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. 1973, pp. 194–206. DOI: 10.1145/512927.512945.

[103] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. “Implicit Flows: Can’t Live with ‘Em, Can’t Live without ‘Em”. In: *Information Systems Security*. Ed. by R. Sekar and Arun K. Pujari. 2008, pp. 56–70. DOI: 10.1007/978-3-540-89862-7_4.

[104] The Programming Paradigms Group at KIT. *JOANA (JAVA Object-sensitive ANAlysis) - Information Flow Control Framework for JAVA*. URL: https://pp.ipd.kit.edu/projects/joana/ (visited on 09/22/2021).

[105] Gerwin Klein and Tobias Nipkow. “A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler”. In: *ACM Transactions on Programming Languages and Systems* 28 (4 July 2006), pp. 619–695. DOI: 10.1145/1146809.1146811.

[106] Jens Knoop and Bernhard Steffen. “The Interprocedural Coincidence Theorem”. In: *Compiler Construction*. 1992, pp. 125–140. DOI: 10.1007/3-540-55984-1_13.

[107] Donald E. Knuth. “A Characterization of Parenthesis Languages”. In: *Information and Control* 11 (3 Sept. 1967), pp. 269–289. DOI: 10.1016/S0019-9958(67)90564-5.

[108] Donald E. Knuth. “A generalization of Dijkstra's algorithm”. In: *Information Processing Letters* 6 (1 Feb. 1977), pp. 1–5. doi: 10.1016/0020-0190(77)90002-3.

[109] Tobias Kranz and Sebastian Schipper. *The Java-FTP-Client project*. version of 2014-06-16. url: https://web.archive.org/web/20140616134349/http://sourceforge.net/projects/javaftp/ (visited on 09/22/2021). Note: Unfortunately this page was removed shortly after we downloaded the source code. See https://web.archive.org/web/20200731153651/https://sourceforge.net/p/forge/site-support/8398/.

[110] Jens Krinke. “Advanced Slicing of Sequential and Concurrent Programs”. PhD thesis. Universität Passau, Fakultät für Informatik und Mathematik, Apr. 2003. URN: urn:nbn:de:bvb:739-opus-375.

[111] Jens Krinke. “Barrier Slicing and Chopping”. In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. 2003, pp. 81–87. doi: 10.1109/SCAM.2003.1238034.

[112] Jens Krinke. “Effects of Context on Program Slicing”. In: *Journal of Systems and Software* 79 (9 Sept. 2006), pp. 1249–1260. doi: 10.1016/j.jss.2006.02.040.

[113] Ralf Küsters. “Simulation-Based Security with Inexhaustible Interactive Turing Machines”. In: *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 2006, pp. 309–320. doi: 10.1109/CSFW.2006.30.

[114] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. “Extending and Applying a Framework for the Cryptographic Verification of Java Programs”. In: *Principles of Security and Trust*. 2014, pp. 220–239. doi: 10.1007/978-3-642-54792-8_12.

[115] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. “A Hybrid Approach for Proving Noninterference of Java Programs”. In: *2015 IEEE 28th Computer Security Foundations Symposium*. 2015, pp. 305–319. doi: 10.1109/CSF.2015.28.

[116] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. “A Framework for the Cryptographic Verification of Java-like Programs”. In: *2012 IEEE 25th Computer Security Foundations Symposium*. 2012, pp. 198–212. doi: 10.1109/CSF.2012.9.

[117] Peter Lammich and Markus Müller-Olm. "Precise Fixpoint-Based Analysis of Programs with Thread-Creation and Procedures". In: *CONCUR 2007 – Concurrency Theory*. 2007, pp. 287–302. DOI: 10.1007/978-3-540-74407-8_20.

[118] Peter Lammich, Markus Müller-Olm, and Alexander Wenner. "Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints". In: *Computer Aided Verification*. 2009, pp. 525–539. DOI: 10.1007/978-3-642-02658-4_39.

[119] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Transactions on Programming Languages and Systems* 1 (1 July 1979), pp. 121–141. DOI: 10.1145/357062.357071.

[120] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. "Cassandra: Towards a Certifying App Store for Android". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014, pp. 93–104. DOI: 10.1145/2666620.2666631.

[121] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. "Cassandra: Towards a Certifying App Store for Android". In: *Proceedings of the 4th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '14. 2014, pp. 93–104. DOI: 10.1145/2666620.2666631.

[122] Enrico Lovat, Alexander Fromm, Martin Mohr, and Alexander Pretschner. "SHRIFT System-Wide HybRid Information Flow Tracking". In: *ICT Systems Security and Privacy Protection*. Springer International Publishing, 2015, pp. 371–385. DOI: 10.1007/978-3-319-18467-8_25.

[123] Martin Mohr, Jürgen Graf, and Martin Hecker. "JoDroid: Adding Android Support to a Static Information Flow Control Tool". In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015. Dresden, 17.-18. März 2015.* Vol. 1337. CEUR-WS.org, 2015, pp. 140–145. URN: urn:nbn:de:0074-1337-4. URL: http://ceur-ws.org/Vol-1337/paper25.pdf (visited on 09/23/2021).

[124] William Q. Meeker, Gerald J. Hahn, and Luis A. Esobar. *Statistical Intervals: A Guide for Practitioners and Researchers*. Second Edition. John Wiley & Sons, Inc., Hoboken, New Jersey, 2017. ISBN: 978-1-118-59484-1. DOI: 10.1002/9781118594841.

[125] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. "Parameterized Object Sensitivity for Points-to Analysis for Java". In: *ACM Transactions on Software Engineering and Methodology* 14 (1 Jan. 2005), pp. 1–41. DOI: 10.1145/1044834.1044835.

[126] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN: 978-1-55860-320-2.

[127] Markus Müller-Olm. *Variations on Constants: Flow Analysis of Sequential and Parallel Programs*. Vol. 3800. Lecture Notes in Computer Science. Springer, 2006. DOI: 10.1007/11871743.

[128] Markus Müller-Olm and Helmut Seidl. "Precise Interprocedural Analysis through Linear Algebra". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2004, pp. 330–341. DOI: 10.1145/964001.964029.

[129] Mangala Gowri Nanda and S. Ramesh. "Interprocedural Slicing of Multithreaded Programs with Applications to Java". In: *ACM Transactions on Programming Languages and Systems* 28 (6 Nov. 2006), pp. 1088–1144. DOI: 10.1145/1186632.1186636.

[130] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. corrected 2nd printing 2005. Berlin Heidelberg: Springer-Verlag, 1999. ISBN: 978-3-642-08474-4. DOI: 10.1007/978-3-662-03811-6.

[131] Benedikt Nordhoff, Markus Müller-Olm, and Peter Lammich. "Iterable Forward Reachability Analysis of Monitor-DPNs". In: *Electronic Proceedings in Theoretical Computer Science*. EPTCS 129 (2013), pp. 384–403. DOI: 10.4204/EPTCS.129.24.

[132] Jaehong Park and Ravi Sandhu. "The UCONABC usage control model". In: *ACM Transactions on Information and System Security* 7 (1 Feb. 2004), pp. 128–174. DOI: 10.1145/984334.984339.

[133] Williams Pelegrin. *A flashlight app stole 50-100 million people's location data, gets a slap on the wrist*. version of 2020-03-19. URL: https://web.archive.org/web/20200319171944/https://www.digitaltrends.com/mobile/brightest-flashlight-ftc-punishment/ (visited on 09/22/2021).

[134] Birgit Pfitzmann and Michael Waidner. "A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission". In: *IEEE Symposium on Security and Privacy*. 2001, pp. 184–200. DOI: 10.1109/SECPRI.2001.924298.

[135] Alexander Pretschner, Enrico Lovat, and Matthias Büchler. "Representation-Independent Data Usage Control". In: *Data Privacy Management and Autonomous Spontaneus Security*. 2012, pp. 122–140. DOI: 10.1007/978-3-642-28879-1_9.

[136] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. 1995, pp. 49–61. DOI: 10.1145/199448.199462.

[137] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. "Speeding Up Slicing". In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '94. 1994, pp. 11–20. DOI: 10.1145/193173.195287.

[138] Thomas Reps and Genevieve Rosay. "Precise Interprocedural Chopping". In: *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '95. 1995, pp. 41–52. DOI: 10.1145/222124.222138.

[139] Thomas Reps, Stefan Schwoon, and Somesh Jha. "Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis". In: *Static Analysis*. 2003, pp. 189–213. DOI: 10.1007/3-540-44898-5_11.

[140] Thomas Reps and Wuu Yang. "The semantics of Program Slicing and Program Integration". In: *TAPSOFT '89. Proceedings of the International Joint Conference on Theory and Practice of Software Development Barcelona, Spain, March 13–17, 1989*. 1989, pp. 360–374. DOI: 10.1007/3-540-50940-2_47.

[141] IBM Research. *T.J. Watson Libraries for Analysis (WALA)*. version of 2020-05-04. URL: https://web.archive.org/web/20200504164652/http://wala.sourceforge.net/wiki/index.php/Main_Page (visited on 09/22/2021).

[142] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74 (2 1953), pp. 358–366. ISSN: 00029947. DOI: 10.2307/1990888.

[143] Andrei Sabelfeld and Andrew C. Myers. "A Model for Delimited Information Release". In: *Proceedings of the 2nd Mext-WSF-JSPS International Symposium on Software Security*. 2004, pp. 174–191. DOI: 10.1007/978-3-540-37621-7_9.

[144] Andrei Sabelfeld and David Sands. "Declassification: Dimensions and principles". In: *Journal of Computer Security* 17 (5 Oct. 2009), pp. 517–548. DOI: 10.3233/JCS-2009-0352.

[145] Mooly Sagiv, Thomas Reps, and Susan Horwitz. "Precise interprocedural dataflow analysis with applications to constant propagation". In: *Theoretical Computer Science* 167 (1–2 1996), pp. 131–170. DOI: 10.1016/0304-3975(96)00072-2.

[146] David Schmidt and Bernhard Steffen. "Program Analysis as Model Checking of Abstract Interpretations". In: *Static Analysis*. 1998, pp. 351–380. DOI: 10.1007/3-540-49727-7_22.

[147] S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. "On generalized authorization problems". In: *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* 2003, pp. 202–216. DOI: 10.1109/CSFW.2003.1212714.

[148] *SecuriBench*. version of 2018-08-17. URL: http://web.archive.org/web/20180817120009/https://suif.stanford.edu/~livshits/securibench/ (visited on 09/22/2021).

[149] *SecuriBench Micro*. version of 2018-08-17. URL: http://web.archive.org/web/20180817122244/https://suif.stanford.edu/~livshits/work/securibench-micro/ (visited on 09/22/2021).

[150] Jérôme Segura. *Uncovering an Android botnet involved in SMS fraud*. version of 2020-03-19. URL: `http://web.archive.org/web/20200319171154/https://blog.malwarebytes.com/cybercrime/2013/09/uncovering-an-android-botnet-involved-in-sms-fraud/` (visited on 09/22/2021).

[151] Helmut Seidl and Bernhard Steffen. "Constraint-Based Inter-Procedural Analysis of Parallel Programs". In: *Programming Languages and Systems*. 2000, pp. 351–365. DOI: `10.1007/3-540-46425-5_23`.

[152] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. "Foundations and Intraprocedural Optimization". In: *Compiler Design: Analysis and Transformation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–114. ISBN: 978-3-642-17548-0. DOI: `10.1007/978-3-642-17548-0_1`.

[153] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Übersetzerbau: Band 3: Analyse und Transformation*. Berlin, Heidelberg: Springer, 2010. DOI: `10.1007/978-3-642-03331-5`.

[154] Micha Sharir and Amir Pnueli. "Two approaches to interprocedural data flow analysis". In: *Program Flow Analysis: Theory and Applications*. Ed. by Steven S Muchnick and Neil D Jones. Englewood Cliffs (N.J.) : Prentice-Hall, 1981. Chap. 7, pp. 189–233. ISBN: 978-0-137-29681-1. URL: `http://www.worldcat.org/oclc/876050229`.

[155] Olin. Shivers. "Control-flow analysis of higher-order languages: or taming lambda". PhD thesis. School of Computer Science, Carnegie Mellon University, 1991.

[156] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. "Checking Probabilistic Noninterference Using JOANA". In: 56 (6 Nov. 2014), pp. 280–287. DOI: `10.1515/itit-2014-1051`.

[157] Gregor Snelting, Torsten Robschink, and Jens Krinke. "Efficient Path Conditions in Dependence Graphs for Software Safety Analysis". In: *ACM Transactions on Software Engineering and Methodology* 15 (4 Oct. 2006), pp. 410–457.

[158] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. "Alias Analysis for Object-Oriented Programs". In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer Berlin Heidelberg, 2013, pp. 196–232.

[159] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. 1996, pp. 32–41. DOI: 10.1145/237721.237727.

[160] Tetsuo Tamai. "A class of fixed-point problems on graphs and iterative solution algorithms". In: *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 76 (8 1993), pp. 25–36. DOI: 10.1002/ecjc.4430760803.

[161] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications". In: *Pacific Journal of Mathematics* 5 (2 1955), pp. 285–309. DOI: 10.2140/pjm.1955.5-2.

[162] Positive Technologies. *Vulnerabilities and threats in mobile applications, 2019*. version of 2020-03-19. URL: https://web.archive.org/web/20200319173815/https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/ (visited on 09/22/2021).

[163] *Verification of the MixServer of sElect*. URL: https://github.com/escapin/MixServerVerification (visited on 09/22/2021).

[164] Daniel Wasserrab. "From Formal Semantics to Verified Slicing : A Modular Framework with Applications in Language Based Security". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2011. DOI: 10.5445/KSP/1000020678.

[165] Mark Weiser. "Program Slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. 1981, pp. 439–449. DOI: 10.5555/800078.802557.

[166] Mark Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10 (4 July 1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[167] Maik Wiesner. "Intent-Analyse von Android-Applikationen". German. bachelor's thesis. IPD Snelting at Karlsruhe Institute of Technology (KIT), 2016.

[168] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. “SAILS: static analysis of information leakage with sample”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. 2012, pp. 1308–1313. DOI: 10.1145/2245276.2231983.

[169] S. Zdancewic and A. C. Myers. “Observational determinism for concurrent program security”. In: *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* 2003, pp. 29–43. DOI: 10.1109/CSFW.2003.1212703.

A

Proofs

A.1 Proof of Theorem 5.9

Lemma A.1. *Let $\pi \in Bal(E)$ be a balanced symbol sequence. Assume that $(i, j) \in \nu_\pi$ and that either of the following holds:*

(i) i is the least call position.

(ii) j is the greatest return position.

Then both $\pi^{<i}$ and $\pi^{>j}$ are balanced.

Proof. We first consider case (i). Since π is balanced, we have $c(\pi^{<i'}) \geq 0$ for any $i' \leq i$. Furthermore, $c(\pi^{<i})$ must be 0 because $\pi^{<i}$ contains no call symbols. This together proves that $\pi^{<i}$ is balanced. Now we show that $\pi^{>j}$ is balanced. It suffices to show that (1) $c(\pi^{>j}) = 0$ and (2) $c(\pi^{]j,k[}) \geq 0$ for every $k \in]j, n-1]$, where $n \overset{def}{=} |\pi|$.

We have $c(\pi) = 0$ since π is balanced. Moreover, we already have argued that $c(\pi^{<i}) = 0$. Furthermore, because $(i, j) \in \nu_\pi$, $\pi^{]i,j[}$ is balanced, so we have $c(\pi^{]i,j[}) = 0$. Together we can conclude:

$$\begin{aligned}
0 = c(\pi) &= c(\pi^{\leq j}) + c(\pi^{>j}) \\
&= c(\pi^{<i}) + c(\pi^{i}) + c(\pi^{]i,j[}) + c(\pi^{j}) + c(\pi^{>j}) \\
&= 0 + 1 + 0 + (-1) + c(\pi^{>j}) \\
&= 0 + c(\pi^{>j}) \\
&= c(\pi^{>j})
\end{aligned}$$

It remains to show that $c(\pi^{]j,k[}) \geq 0$ for every $k \in]j, n-1]$. For this, note that

$$c(\pi^{<j}) = c(\pi^{<i}) + c(\pi^{i}) + c(\pi^{]i,j[}) = 0 + 1 + 0 = 1$$

and $c(\pi^{j}) = -1$.
Now pick any $k \in]j, n-1]$. Then we have

$$\begin{aligned} 0 &\leq c(\pi^{<k}) \\ &= c(\pi^{<j}) + c(\pi^{j}) + c(\pi^{]j,k[}) \\ &= 1 + (-1) + c(\pi^{]j,k[}) \\ &= c(\pi^{]j,k[}). \end{aligned}$$

Now consider case (ii). We have $c(\theta) \geq 0$ for every prefix θ of $\pi^{<i}$ since π is balanced and $c(\theta') \geq 0$ for every prefix θ' of $\pi^{>j}$, since $\pi^{>j}$ does not contain any return symbols. Furthermore, due to the balancedness of $\pi^{]i,j[}$ we can derive

$$\begin{aligned} 0 = c(\pi) &= c(\pi^{<i}) + c(\pi_i) + c(\pi^{]i,j[}) + c(\pi_j) + c(\pi^{>j}) \\ &= c(\pi^{<i}) + 1 + 0 + (-1) + c(\pi^{>j}) \\ &= c(\pi^{<i}) + c(\pi^{>j}) \end{aligned}$$

Since both $c(\pi^{<i})$ and $c(\pi^{>j})$ are not negative, they must both be 0. In summary, we have shown that both $c(\pi^{<i})$ and $c(\pi^{>j})$ must be balanced. □

Theorem 5.9. *For any symbol sequence $\pi \in E^{\star}$, the following conditions are equivalent:*

(a) π is balanced.

(b) ν_{π} is left- and right-total, i.e. a bijective function $Callpos(\pi) \rightarrow Retpos(\pi)$.

Proof. (a) $\Longrightarrow$ (b)

We show that the claim

$$\forall \pi \in E^{\star}.\ \pi \text{ balanced} \implies \nu_{\pi} \text{ bijective}$$

by strong induction on the number $K \in \mathbb{N}$ of call symbols in π.

induction hypothesis: The claim is proven for all π' which contain $L < K$ call symbols.

Let $\pi \in E^\star$ be a symbol sequence with K call symbols and assume that π is balanced. Then, by Lemma 5.4, π also contains K return symbols. Let $n = |\pi|$ be the length of π.

If $K = 0$, i.e. if π does not contain any call or return symbols, then ν_π is the empty relation and therefore trivially fulfills the conditions of a bijective function between empty sets.

Now assume that π contains $K > 0$ call and K return symbols. We show left- and right-totality separately.

left-totality Let $i \in range(\pi)$ be the least index such that $\pi_i = e_{call} \in E_{call}$. Hence, we can write

$$\pi = \pi^{<i} \cdot e_{call} \cdot \pi^{>i} \tag{A.1}$$

First, we show that there must be a $k \in]i, n[$ such that $c(\pi^{]i,k]}) < 0$. This can be seen as follows. First, due to the choice of i, $\pi^{<i}$ does not contain any call symbols. Moreover, because π is balanced, we have $c(\pi^{<i}) \geq 0$. But this means that $\pi^{<i}$ cannot contain return symbols either. Hence, $c(\pi^{<i}) = 0$ and from this, we derive that $c(\pi^{>i}) < 0$ by the following computation:

$$\begin{aligned}
0 &= c(\pi) && \{\ \pi \text{ balanced}\ \} \\
&= c(\pi^{<i}) + c(e_{call}) + c(\pi^{>i}) && \{\ \text{(A.1), additivity of } c\ \} \\
&= c(\pi_i) + c(\pi^{>i}) && \{\ c(\pi^{<i}) = 0\ \} \\
&> c(\pi^{>i}). && \{\ \pi^i \in E_{call}\ \}
\end{aligned}$$

But by definition of c, $c(\pi^{>i}) < 0$ implies that there must be $k \in]i, n[$ such that $c(\pi^{]i,k]}) < 0$.

Now let j be the smallest $k \in]i, n[$ such that $c(\pi^{]i,k]}) < 0$. Obviously, we have $i < j$. Moreover:

- $\pi_j \in E_{ret}$: If this were not the case, it would follow that

  $$c(\pi^{]i,j-1]}) < 0,$$

  a contradiction to the choice of j.

- For $k' \in]i,j[$, consider the prefix $\pi^{]i,k']}$ of $\pi^{]i,j[}$. Then $c(\pi^{]i,k']}) \geq 0$, since $k' < j$ and j was chosen to be the smallest k with $c(\pi^{]i,k]}) < 0$.
- $c(\pi^{]i,j[}) = 0$: From the previous fact we know already that $c(\pi^{]i,j[}) \geq 0$. Furthermore, because $0 > c(\pi^{]i,j]}) = c(\pi^{]i,j[}) + c(\pi_j) = c(\pi^{]i,j[}) + (-1)$, we can also conclude that $c(\pi^{]i,j[}) \leq 0$.

From the former two statements we conclude that $\pi^{]i,j[}$ is balanced. Now we have

$$\pi = \pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j}$$

and have shown that $\pi^{]i,j[}$ is balanced. Together with $i < j$ and $\pi_j = e_{ret} \in E_{ret}$ this entails $(i,j) \in \nu_\pi$.

Now, because π is balanced (by assumption) and due to the choice of i we can apply Lemma A.1 and additionally conclude that $\pi^{>j}$ is balanced. Since both $\pi^{]i,j[}$ and $\pi^{>j}$ contain at most $K-1$ call symbols, we can apply the induction hypothesis to them and gain that $\nu_{\pi^{]i,j[}}$ and $\nu_{\pi^{>j}}$ are bijective, so in particular left-total.

Let $i' \neq i$ be another call position of π. Due to the choice of i, it must be $i' > i$ and since $\pi_j \in E_{ret}$, we conclude that either $i' < j$ or $i' > j$. If $i' < j$ then by using that $\pi^{i'} = (\pi^{]i,j[})^{i'-(i+1)}$ and the left-totality of $\nu_{\pi^{]i,j[}}$, we obtain a j' such that $(i'-(i+1), j') \in \nu_{\pi^{]i,j[}}$ which means by Lemma 5.17 that $(i, j'+(i+1)) \in \nu_\pi$. Similarly, we find a j' with $(i', j') \in \nu_\pi$ in the case that $i' > j$. This concludes the proof of the left-totality of ν_π.

right-totality Let $j \in range(\pi)$ be the greatest index such that $\pi^j \in E_{ret}$. Then we have

$$0 \geq c(\pi^{\leq j}) = c(\pi^{<j}) + c(\pi^j) = c(\pi^{<j}) - 1,$$

which means that $c(\pi^{\leq j}) > 0$. Hence, by the properties of c, there must be a $k \in [0,j[$ such that $c(\pi^{[k,j[}) > 0$. We choose i to be the greatest such k. Then we have $i < j$. Moreover, due to the maximality of i, we have $c(\pi^{]i,j[}) \leq 0$. Lastly, we have

$$0 < c(\pi^{[i,j[}) = c(\pi^i) + c(\pi^{]i,j[}).$$

Because $c(\pi^{]i,j[}) \leq 0$, this necessarily entails $c(\pi^i) > 0$, i.e. $c(\pi^i) = 1$, and $c(\pi^{]i,j[}) = 0$.

Now let $k \in]i, j[$. Then we have

$$0 = c(\pi^{]i,j[}) = c(\pi^{]i,k]}) + c(\pi^{]k,j[}),$$

but since $k > i$ and because of the maximality property of i, it must be $c(\pi^{]k,j[}) \leq 0$, which means that $c(\pi^{]i,k]}) \geq 0$.

Together this shows $(i, j) \in \nu_\pi$. By Lemma A.1 and the choice of j, $\pi^{<i}$ is balanced. For the other return positions $j' < j$ we proceed similarly to the left-totality part: We apply the induction hypothesis (noting that $\pi^{]i,j[}$ and $\pi^{<i}$ contain less call symbols) and Lemma 5.17 to obtain i' such that $(i', j') \in \nu_\pi$.

(b) $\Longrightarrow$ (a) Let $\pi \in E^\star$ be a symbol sequence such that

$$\nu_\pi : \mathit{Callpos}(\pi) \to \mathit{Retpos}(\pi)$$

is a bijective function. We must show that π is balanced. By Lemma 5.6 it suffices to show that

$$\text{(1)} \qquad c(\pi) = 0$$

$$\text{(2)} \qquad \forall i \in \mathit{range}(\pi).\, c(\pi^{\leq i}) \geq 0$$

Claim (1) is clear because of the bijectivity of ν_π and the first statement in Lemma 5.4.

It remains to show (2). We proceed by strong induction on i. Let $i \in \mathit{range}(\pi)$. The induction hypothesis is

$$\text{(IH)} \qquad \forall i' < i. c(\pi^{\leq i'}) \geq 0$$

We have to show $c(\pi^{\leq i}) \geq 0$. For this, we make a case distinction on whether $i = 0$ or not.

- If $i = 0$, the claim follows easily from the definition of c.
- Now assume $i \neq 0$ and consider π^i. For $\pi^i \in E_{intra} \cup E_{call}$, we have $c(\pi^i) \geq 0$. Moreover, by (IH), we have $c(\pi^{\leq i-1}) \geq 0$. These two facts entail $c(\pi^{\leq i}) = c(\pi^{<i}) + c(\pi^i) \geq 0$.

Now consider the case that $\pi_i \in E_{ret}$. Since ν_π is bijective, there is $j < i$ such that $\pi_j \in E_{call}$ and $\pi^{]j,i[}$ is balanced. We conclude

$$c(\pi^{<i}) = \underbrace{c(\pi^{<j})}_{\geq 0 \text{ by (IH)}} + \underbrace{c(\pi^j)}_{=1} + \underbrace{c(\pi^{]j,i[})}_{=0} > 0.$$

This proves that $c(\pi^{\leq i}) \geq 0$.

□

A.2 Proof of Theorem 5.10

Theorem 5.10. *Given $\pi \in E^\star$, assume that $(i,j), (i',j') \in \nu_\pi$. Then one of the following statements is true:*

1. $[i,j] \subseteq [i',j']$

2. $[i',j'] \subseteq [i,j]$

3. $[i,j] \cap [i',j'] = \emptyset$

Proof. For $(i,j) = (i',j')$, the theorem is trivially true. If $(i,j) \neq (i',j')$, then left- and right-uniqueness of ν_π (Theorem 5.8) gives us $i \neq i' \land j \neq j'$. Also note that $i \neq j'$ and $j \neq i'$, because $\pi^i, \pi^{i'} \in E_{call}$ and $\pi^j, \pi^{j'} \in E_{ret}$. Since $(i,j) \in \nu_\pi$, π can be split up into

$$\pi = \pi^{<i} \cdot \pi^i \cdot \pi^{]i,j[} \cdot \pi^j \cdot \pi^{>j},$$

where $\pi_i \in E_{call}$, $\pi_j \in E_{ret}$ and $\pi^{]i,j[}$ is balanced. Now, we make a case distinction of where i' lies relative to i and j and show for every case that one of the three conditions from the claim must be true.

$i' < i$: Then j' cannot be in $]i,j[$. Assume, for the purpose of contradiction, that it were. Since $\pi^{]i,j[}$ is balanced, we may apply Theorem 5.9 and find i'' such that $(i'', j' - (i+1)) \in \nu_{\pi^{]i,j[}}$. By shifting, we see that $(i'' + i + 1, j') \in \nu_\pi$. Since ν_π is left-unique, this means that $i' = i'' + i + 1$. But, since $i + 1 > 0$, it must be $i' > i$, which contradicts the case we consider currently. So, the assumption that j' is in $]i,j[$ is false. It follows that either $j' < i$ or $j' > j$. In the former case, since $j' > i'$, we have $[i,j] \cap [i',j'] = \emptyset$ and in the latter case we have $[i,j] \subseteq [i',j']$.

$i' > i \wedge i' < j$: First, we note that $j' > i$. This follows from $(i', j') \in \nu_\pi$ and $i' > i$. Moreover, we observe that j' cannot be greater than j. This can be shown analogously to the previous case, by shifting around, by using that ν_π is right-unique and by exploiting the balancedness of $\pi^{]i,j[}$. In summary, we have shown $[i', j'] \subseteq [i, j]$.

$i' > j$: Since $(i', j') \in \nu_\pi$ implies $i' < j'$, it must also be $j' > j$. So we have $[i, j] \cap [i', j'] = \emptyset$.

□

A.3 Proof of Theorem 5.19

Lemma A.2. *We have $E_{intra} \subseteq Bal(E)$, $E_{call} \subseteq Right(E)$ and $E_{ret} \subseteq Left(E)$.*

Proof.

- For $e \in E_{intra}$, ν_e is both left- and right-total, since $Callpos(e) = Retpos(e) = \emptyset$. This implies $e \in Bal(E)$ by Theorem 5.9.
- For $e \in E_{call}$, ν_e is right-total, since $Retpos(e) = \emptyset$.
- For $e \in E_{ret}$, ν_e is left-total, since $Callpos(e) = \emptyset$.

□

Lemma A.3. *$Left(E)$, $Right(E)$ and $Bal(E)$ are all closed under concatenation.*

1. *If $\pi, \pi' \in Left(E)$, then $\pi \cdot \pi' \in Left(E)$.*
2. *If $\pi, \pi' \in Right(E)$, then $\pi \cdot \pi' \in Right(E)$.*
3. *If $\pi, \pi' \in Bal(E)$, then $\pi \cdot \pi' \in Bal(E)$.*

Proof. 1. Let $i \in range(\pi \cdot \pi')$. Then either $i \in range(\pi)$ or $i = |\pi| + i'$ with $i' \in range(\pi')$. We consider each case separately:

a) If $i \in range(\pi)$, then because $\pi \in Left(E)$, there is $j \in range(\pi) \subseteq range(\pi \cdot \pi')$ with $(i, j) \in \nu_\pi$.

b) If $i = |\pi| + i'$ with $i' \in range(\pi')$, then because $\pi' \in Left(E)$, we find $j' \in range(\pi')$ with $(i', j') \in \nu_{\pi'}$. Then $(i' + |\pi|, j' + |\pi|) = (i, j' + |\pi|) \in \nu_{\pi \cdot \pi'}$ by Lemma 5.17.

2. This follows by an analogous argument as the first statement.

3. This is a consequence of Theorem 5.9 and the first two statements.

□

Lemma A.4. *If $\pi \in Bal(E)$, $e_{call} \in E_{call}$ and $e_{ret} \in E_{ret}$, then $e_{call} \cdot \pi \cdot e_{ret} \in Bal(E)$.*

Proof. Let $\pi \in Bal(E)$, $e_{call} \in E_{call}$ and $e_{ret} \in E_{ret}$ and define $\pi' \overset{def}{=} e_{call} \cdot \pi \cdot e_{ret}$. We show that

(A) $\quad c(\pi') = 0$

(B) $\quad \forall \theta \in Prefix(\pi').\, c(\theta) \geq 0$

This implies that $\pi' \in Bal(E)$ by Lemma 5.6.

(A) Since $\pi \in Bal(E)$ we have $c(\pi) = 0$. This implies

$$c(\pi') = c(e_{call}) + c(\pi) + c(e_{ret}) = 1 + c(\pi) + (-1) = 1 + 0 + (-1) = 0.$$

(B) Let $\theta \in Prefix(\pi')$. The case $\theta = \pi'$ is already covered by (A). Also, $c(\epsilon) = 0$ holds by definition. It remains the case that $\theta = e_{call} \cdot \theta'$ for some prefix θ' of π. But $\pi \in Bal(E)$, which implies $c(\theta') \geq 0$ by Lemma 5.6. It follows

$$c(\theta) = c(e_{call} \cdot \theta') = c(e_{call}) + c(\theta') = 1 + c(\theta') > 0.$$

□

Theorem 5.19. *$Bal(E)$ is the least subset X of $E^\star$ with the following properties:*

$$(Bal1)\ \frac{}{\epsilon \in X} \qquad (Bal2)\ \frac{\pi \in X \qquad e \in E_{intra}}{\pi \cdot e \in X}$$

$$(Bal3)\ \frac{\pi \in X \qquad \pi' \in X \qquad e_{call} \in E_{call} \qquad e_{ret} \in E_{ret}}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Proof. First, we show that $Bal(E)$ satisfies the properties $Bal1$, $Bal2$, $Bal3$. It is clear that $Bal(E)$ satisfies $Bal1$. Furthermore, Lemma A.2, Lemma A.3 and Lemma A.4 imply that it also has the properties $Bal2$ and $Bal3$.

It remains to show that $Bal(E)$ is the least subset of $E^\star$ with properties $Bal1$, $Bal2$, $Bal3$. For this, let $X \subseteq E^\star$ be a set of symbol sequences that satisfies the closure properties $Bal1$, $Bal2$ and $Bal3$. We show $Bal(E) \subseteq X$ by strong induction on the lengths of symbol sequences. The induction hypothesis for $n \in \mathbb{N}$ is

$$\forall \pi \in Bal(E).|\pi| < n \implies \pi \in X \tag{A.2}$$

Now let $\pi \in Bal(E)$, $|\pi| = n$. We show $\pi \in X$ by a case distinction on whether $Retpos(\pi) = \emptyset$ or not.

1. If $Retpos(\pi) = \emptyset$, then we also have $Callpos(\pi) = \emptyset$, because π is balanced. Hence, π is either empty or solely consists of intraprocedural symbols. Thus, $\pi \in X$ can be derived by repeated application of $Bal1$ and $Bal2$.

2. If $Retpos(\pi) \neq \emptyset$, then let j be the maximum of $Retpos(\pi)$. Since π is balanced, there must be $i \in range(\pi)$ with $(i, j) \in \nu_\pi$. With $e_{call} \overset{def}{=} \pi^i$ and $e_{ret} \overset{def}{=} \pi^j$, π can be decomposed into

$$\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j}$$

From $(i, j) \in \nu_\pi$ it follows that $\pi^{]i,j[}$ is balanced. From this, the balancedness of π and the choice of j, we can conclude by application of Lemma A.1 that both $\pi^{<i}$ and $\pi^{>j}$ are balanced.

Since both $\pi^{<i}$ and $\pi^{]i,j[}$ are balanced and shorter than π, we obtain $\pi^{<i} \in X$ and $\pi^{]i,j[} \in X$ by induction hypothesis. With $Bal3$ we get

$$\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \in X \tag{A.3}$$

Due to the choice of j, we have $Retpos(\pi^{>j}) = \emptyset$. Since $\pi^{>j}$ is balanced, this implies that $Callpos(\pi^{>j}) = \emptyset$, too. Hence, from (A.3) we get

$$\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j} \in X$$

by repeated application of $Bal2$.

□

A.4 Proof of Theorem 5.20

Definition A.5. *For a given symbol sequence $\pi \in E^\star$, we define the set $MRet(\pi)$ of* matched return positions *as follows:*

$$MRet(\pi) \overset{def}{=} \{j \in Retpos(\pi) \mid \exists i \in range(\pi).\ (i, j) \in \nu_\pi\}$$

Lemma A.6. *Let $\pi \in Left(E)$ such that $MRet(\pi) \neq \emptyset$. Assume that j is the greatest element of MRet and let $i = \nu_\pi^{-1}(j)$. Then the following statements hold:*

1. Both $\pi^{<i}$ and $\pi^{>j}$ are left-total.

2. $\pi^{>j}$ consists only of symbols from $E_{intra} \cup E_{ret}$

Proof. We show the two statements separately.

1. Let $i' \in Callpos(\pi^{<i})$. Because $Callpos(\pi^{<i}) \subseteq Callpos(\pi)$ and $\pi \in Left(E)$, we find $j' \in range(\pi)$ with $(i', j') \in \nu_\pi$. This j' is a member of $MRet(\pi)$: $j' \in MRet(\pi)$. Due to the choice of j, we have $j' < j$. With Theorem 5.10 and $i' < i$ we get $j' < i$. Therefore, $(i', j') \in v_{\pi^{<i}}$. This shows $\pi^{<i} \in Left(E)$.

2. This is an easy consequence of the maximality of j.

□

Theorem 5.20. *$Left(E)$ is the least subset X of $E^\star$ which has the following properties:*

$$(Left1)\ \frac{}{\epsilon \in X} \qquad (Left2)\ \frac{\pi \in X \qquad e \in E_{intra} \cup E_{ret}}{\pi \cdot e \in X}$$

$$(Left3)\ \frac{\pi \in X \qquad \pi' \in Bal(E) \qquad e_{call} \in E_{call} \qquad e_{ret} \in E_{ret}}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Proof. First, we observe that $Left(E)$ has the closure properties $Left1$, $Left2$, $Left3$. This is clear for $Left1$, the other two properties follow from Lemma A.2, Lemma A.3, Lemma A.4 and Remark 5.12.
Next, let $X \subseteq E^\star$ be a set with the closure properties $Left1$, $Left2$ and $Left3$.

Now we show $Left(E) \subseteq X$ by strong induction on the length of sequences from $E^\star$.
For $n \in \mathbb{N}$, the induction hypothesis is

$$\forall \pi \in E^\star . |\pi| < n \land \pi \in Left(E) \implies \pi \in X \tag{A.4}$$

Now let $\pi \in E^\star$ be a symbol sequence with $|\pi| = n$ and $\pi \in Left(E)$. We show $\pi \in X$ by case distinction on whether $MRet(\pi) = \emptyset$ or not.

1. If $MRet(\pi) = \emptyset$, then $Callpos(\pi)$ must be empty, too: Assume, for the purpose of contradiction, that $Callpos(\pi)$ contains some call position i. Then i cannot be matched, because $MRet(\pi) = \emptyset$. But this is a contradiction to $\pi \in Left(E)$. Hence, the assumption is false and $Callpos(\pi) = \emptyset$.

Because $Callpos(\pi) = \emptyset$, π solely consists of symbols from $E_{intra} \cup E_{ret}$. Hence, $\pi \in X$ can be derived by repeated application of $Left1$ and $Left2$.

2. If $MRet(\pi) \neq \emptyset$, then let j be the greatest element of $MRet(\pi)$ and $i = \nu_\pi^{-1}(j)$. With $e_{call} \overset{def}{=} \pi^i$ and $e_{ret} \overset{def}{=} \pi^j$, π can be written as

$$\pi = \pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j}$$

By definition, $\pi^{]i,j[}$ is balanced. Plus, since $\pi \in Left(E)$ and due to the choice of j, by application of Lemma A.6 we obtain $\pi^{<i} \in Left(E)$, $\pi^{>j} \in Left(E)$ and that $\pi^{>j}$ consists only of symbols from $E_{intra} \cup E_{ret}$.

Equipped with these observations, we can finish the proof as follows:

a) Because $|\pi^{<i}| < |\pi|$ and $\pi^{<i} \in Left(E)$, we may apply (A.4) to $\pi^{<i}$ and obtain that $\pi^{<i} \in X$.

b) Now we apply $Left3$ to $\pi^{<i} \in X$, $\pi^{]i,j[} \in Bal(E)$, $e_{call} \in E_{call}$ and $e_{ret} \in E_{ret}$ and get $\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \in X$.

c) Finally, because $\pi^{>j} \in (E_{intra} \cup E_{ret})^\star$, we obtain

$$\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j} \in X$$

by repeated application of $Left2$ to $\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \in X$.

□

A.5 Proof of Theorem 5.21

Lemma A.7. *1. If $\pi \in Left(E)$ and $\pi^{\geq k}$ is a suffix of π, then $\pi^{\geq k} \in Left(E)$.*

2. If $\pi \in Right(E)$ and $\pi^{\leq j}$ is a prefix of π, then $\pi^{\leq j} \in Right(E)$.

Proof. 1. Consider an arbitrary $\pi \in Left(E)$ and let $\pi^{\geq k}$ be a suffix of π. We need to show that $\nu_{\pi^{\geq k}}$ is left-total. Let $i \in Callpos(\pi^{\geq k})$. Then we show that there is $j \in Retpos(\pi^{\geq k})$ such that $(i, j) \in \nu_{\pi^{\geq k}}$.

First, we observe $(\pi^{\geq k})^i = \pi^{k+i}$. Hence, we have $i + k \in Callpos(\pi)$. Because ν_π is left-total, there is $j' \in Retpos(\pi)$ such that $(i + k, j') \in \nu_\pi$. Moreover, because $j' > k + i \geq k$, we can write j' as $j' = (j' - k) + k$. In particular, we have $j' - k \in range(\pi^{\geq k})$. With Lemma 5.17, it follows from $(i + k, (j' - k) + k) \in \nu_\pi$ that $(i, j' - k) \in \nu_{\pi^{\geq k}}$, which concludes the proof.

2. Let $\pi \in Right(E)$ and $\pi^{\leq j}$ be a prefix of π. Let

$$l \in Retpos(\pi^{\leq j}) \subseteq Retpos(\pi)$$

be a return position in $\pi^{\leq j}$. Due to right-totality of ν_π, we find a $k \in Callpos(\pi)$ such that $(k, l) \in \nu_\pi$. This means in particular that $k < l$. With $l \leq j$ we get $k \leq j$ so that we can conclude $(k, l) \in \nu_{\pi^{\leq j}}$. This proves that $\nu_{\pi^{\leq j}}$ is right-total.

□

Lemma A.8. *Let $\pi \in Right(E)$ such that $Retpos(\pi) \neq \emptyset$. Assume that j is the greatest element of $Retpos(\pi)$ and let $i = \nu_\pi^{-1}(j)$. Then both $\pi^{<i} \in Right(E)$ and $\pi^{>j} \in Right(E)$.*

Proof. Since j is the greatest return position, we have $Retpos(\pi^{>j}) = \emptyset$. Hence, $\pi^{>j}$ can only consist of symbols from $E_{intra} \cup E_{call}$, so that $\pi^{>j} \in Right(E)$ holds trivially. Furthermore, $\pi^{<i} \in Right(E)$ holds because $Right(E)$ is closed under prefixes (by Lemma A.7). □

Theorem 5.21. *Right*(E) *is the least subset* X *of* $E^\star$ *which has the following properties:*

$$(Right1)\ \frac{}{\epsilon \in X} \qquad (Right2)\ \frac{\pi \in X \qquad e \in E_{intra} \cup E_{call}}{\pi \cdot e \in X}$$

$$(Right3)\ \frac{\pi \in X \qquad \pi' \in Bal(E) \qquad e_{call} \in E_{call} \qquad e_{ret} \in E_{ret}}{\pi \cdot e_{call} \cdot \pi' \cdot e_{ret} \in X}$$

Proof. First, we observe that $Right(E)$ has the closure properties *Right*1, *Right*2, *Right*3. This is clear for *Right*1, the other two properties follow from Lemma A.2, Lemma A.3, Lemma A.4 and Remark 5.12.
Next, let $X \subseteq E^\star$ be a set with the closure properties *Right*1, *Right*2 and *Right*3. We show $Right(E) \subseteq X$ by strong induction on the length of sequences from $E^\star$.
For $n \in \mathbb{N}$, the induction hypothesis is

$$\text{(A.5)} \qquad \forall \pi \in E^\star . |\pi| < n \land \pi \in Right(E) \implies \pi \in X$$

Now let $\pi \in E^\star$ be a symbol sequence with $|\pi| = n$ and $\pi \in Right(E)$. We show $\pi \in X$ by case distinction on whether $Retpos(\pi) = \emptyset$ or not.

1. If $Retpos(\pi) = \emptyset$, then π does not contain any return symbols. Hence, π is either empty or solely consists of symbols from $E_{intra} \cup E_{call}$. In both cases, $\pi \in X$ can be derived by repeated application of *Right*1 and *Right*2.

2. If $Retpos(\pi) \neq \emptyset$, then let j be the greatest element of $Retpos(\pi)$ and $i = \nu_\pi^{-1}(j)$. This is well-defined because $\pi \in Right(E)$. With $e_{call} \overset{def}{=} \pi^i$ and $e_{ret} \overset{def}{=} \pi^j$, π can be written as

$$\pi = \pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j}$$

By definition, $\pi^{]i,j[}$ is balanced. Plus, since $\pi \in Right(E)$ and due to the choice of j, application of Lemma A.8 yields that $\pi^{<i} \in Right(E)$.

Now we can finish the proof as follows:

a) Because $|\pi^{<i}| < |\pi|$ and $\pi^{<i} \in Right(E)$, we may apply (A.5) to $\pi^{<i}$ and obtain that $\pi^{<i} \in X$.

b) Now we apply *Right*3 to $\pi^{<i} \in X$, $\pi^{]i,j[} \in Bal(E)$, $e_{call} \in E_{call}$ and $e_{ret} \in E_{ret}$ and get $\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \in X$.

c) Finally, we note that, due to the choice of j, $\pi^{>j}$ only contains symbols from $E_{intra} \cup E_{call}$. Hence, we obtain

$$\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \cdot \pi^{>j} \in X$$

by repeated application of *Right*2 to $\pi^{<i} \cdot e_{call} \cdot \pi^{]i,j[} \cdot e_{ret} \in X$.

□

A.6 Proof of Lemma 5.24

Lemma A.9. *For $\pi \in E^{\star}$ and $e \in E$, we have $\nu_{\pi \cdot e} = \nu_{\pi}$ if one of the following conditions is satisfied:*

(i) $$e \in E_{intra} \cup E_{call}$$

(ii) $$e \in E_{ret} \wedge \pi \in Left(E)$$

Proof. By definition, it is clear that $\nu_{\pi} \subseteq \nu_{\pi \cdot e}$. For the other inclusion we assume either condition and show that $\nu_{\pi \cdot e} \subseteq \nu_{\pi}$ holds in both cases.

(i) Assume that $e \in E_{intra} \cup E_{call}$. Let $(i, j) \in \nu_{\pi \cdot e}$. Then $i < j$ and $(\pi \cdot e)^j \in E_{ret}$. Since $E_{ret} \cap (E_{intra} \cup E_{call}) = \emptyset$, $(\pi \cdot e)^j \in E_{ret}$ implies that $j \neq |\pi|$. Hence $j \in range(\pi)$. But with $i < j$ this means that $i \in range(\pi)$. Hence $(i, j) \in \nu_{\pi}$.

(ii) Assume $e \in E_{ret}$ and $\pi \in Left(E)$. Let $(i, j) \in \nu_{\pi \cdot e}$. From $(i, j) \in \nu_{\pi \cdot e}$ we get $i < j$ so that $i \in range(\pi)$. Moreover, $\pi \in Left(E)$, hence there is $j' \in range(\pi)$ with $(i, j') \in \nu_{\pi} \subseteq \nu_{\pi \cdot e}$. From Theorem 5.8, we get $j = j'$ and thus $(i, j) \in \nu_{\pi}$.

□

Lemma 5.24. *1. If $\pi \in AscSeq(E)$ and $e \in E_{intra} \cup E_{ret}$, then $\pi \cdot e \in AscSeq(E)$.*

2. If $\pi \in DescSeq(E)$ and $e \in E_{intra} \cup E_{call}$, then $\pi \cdot e \in DescSeq(E)$.

3. If $\pi \in SLSeq(E)$ and $e \in E_{intra}$, then $\pi \cdot e \in SLSeq(E)$.

Proof. 1. Let $\pi \in \mathit{AscSeq}(E)$ and $e \in E_{intra} \cup E_{ret}$. Then $\pi \in \mathit{Left}(E) \cap \mathit{Val}(E)$. It follows that $\pi \cdot e \in \mathit{Left}(E)$ by Theorem 5.20. Moreover, $\nu_{\pi \cdot e} = \nu_\pi$ by Lemma A.9. Thus, $\pi \cdot e \in \mathit{Val}(E)$ follows from $\pi \in \mathit{Val}(E)$.

2. Let $\pi \in \mathit{DescSeq}(E)$ and $e \in E_{intra} \cup E_{call}$. Then $\pi \in \mathit{Right}(E) \cap \mathit{Val}(E)$. It follows that $\pi \cdot e \in \mathit{Right}(E)$ by Theorem 5.21. Moreover, $\nu_{\pi \cdot e} = \nu_\pi$ by Lemma A.9. Thus, validness of $\pi \cdot e$ follows from validness of $\pi \in \mathit{Val}(E)$.

3. This follows from a combination of the first two statements.

□

A.7 Proof of Lemma 5.25

Lemma 5.25. *Let $\pi \in \mathit{Val}(E)$ and $\pi' \in \mathit{SLSeq}(E)$. Then the following statements hold:*

1. *$\pi \cdot \pi'$ is valid.*

2. *If $\pi \in \mathit{Bal}(E)$, then $\pi \cdot \pi'$ is same-level.*

3. *If $\pi \in \mathit{Left}(E)$, then $\pi \cdot \pi'$ is ascending.*

4. *If $\pi \in \mathit{Right}(E)$, then $\pi \cdot \pi'$ is descending.*

Proof.

The first statement can be seen as follows: Let $(i, j) \in \nu_{\pi \cdot \pi'}$. Then either $j \in \mathit{range}(\pi)$ or there is $j' \in \mathit{range}(\pi')$ with $j = |\pi| + j'$. We show that $(i, j) \in \Phi$ in either case.

1. Assume $j \in \mathit{range}(\pi)$. Then $i \in \mathit{range}(\pi)$ since $i < j$. Furthermore we have $(i, j) \in \nu_\pi$ by definition of ν_π. This entails $(i, j) \in \Phi$ because π is valid.

2. Assume that $j = |\pi| + j'$ for $j' \in \mathit{range}(\pi')$. Since π' is balanced and hence right-total, there is $i' \in \mathit{range}(\pi')$ with $(i', j') \in \nu_{\pi'}$. By Lemma 5.17, this means that $(i' + |\pi|, j) \in \nu_{\pi \cdot \pi'}$ and because $\nu_{\pi \cdot \pi'}$ is right-unique, it follows that $i = i' + |\pi'|$. Since π' is valid, we get

$$((\pi \cdot \pi')^i, (\pi \cdot \pi')^j) = ({\pi'}^{i'}, {\pi'}^{j'}) \in \Phi.$$

The other three statements are implied by the first statement and Lemma A.3 after noticing that $\mathit{SLSeq}(E) \subseteq \mathit{Bal}(E)$, $\mathit{SLSeq}(E) \subseteq \mathit{Left}(E)$ and $\mathit{SLSeq}(E) \subseteq \mathit{Right}(E)$, respectively. □

A.8 Proof of Lemma 5.26

Lemma 5.26. *If $\pi \in SLSeq(E)$, $e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$ and $(e_{call}, e_{ret}) \in \Phi$, then $e_{call} \cdot \pi \cdot e_{ret} \in SLSeq(E)$.*

Proof. Define $\pi' \overset{def}{=} e_{call} \cdot \pi \cdot e_{ret}$. From $\pi \in Bal(E)$, $e_{call} \in E_{call}$, $e_{ret} \in E_{ret}$, we conclude that $\pi' \in Bal(E)$ by Lemma A.4. It remains to show that $\pi' \in Val(E)$. Let $(i, j) \in \nu_{\pi'}$. Then we show by case distinction on whether $i = 0$ or not that $(\pi^i, \pi^j) \in \Phi$.

1. If $i = 0$, then we must have $j = |\pi'| - 1$: Since $\nu_{\pi'}$ is left-unique by Theorem 5.8, there can be only one j with $(0, j) \in \nu_{\pi'}$. Furthermore, by definition of $\nu_{\pi'}$, we have $(0, |\pi'| - 1) \in \nu_{\pi'}$:

- We have $0 < |\pi'| - 1$, since $|\pi'| \geq 2$.
- We have $\pi'^0 = e_{call} \in E_{call}$ and $\pi'^{|\pi'|-1} = e_{ret} \in E_{ret}$
- We have $\pi'^{]0,|\pi'|-1[} = \pi \in Bal(E)$ by assumption.

Thus $(\pi'^i, \pi'^j) = (\pi'^0, \pi'^{|\pi'|-1}) = (e_{call}, e_{ret}) \in \Phi$ by assumption.

2. Assume $i \neq 0$. Then $i \in [1, |\pi'| - 1[$ because $\pi'^i \in E_{call}$ and $\pi'^{|\pi'|-1} \in E_{ret}$ and $E_{call} \cap E_{ret} = \emptyset$. Furthermore, $j \in]1, |\pi'| - 1[$: $j > 1$ follows from $0 < i$ and $i < j$. Moreover, since $\nu_{\pi'}$ is right-unique and $(0, |\pi'| - 1) \in \nu_{\pi'}$ and $i \neq 0$, we have $j \neq |\pi'| - 1$. Hence, by Lemma 5.17 and because π is valid, we get $(\pi'^i, \pi'^j) \in \Phi$.

□

B

List of Figures

C

List of Tables

D
List of Listings

E

List of Algorithms

Index

I report on applications of slicing and program dependence graphs (PDGs) to software security. Moreover, I propose a framework that generalizes both data-flow analysis on control-flow graphs and slicing on program dependence graphs. Such a framework enables to systematically derive data-flow-like analyses on program dependence graphs that go beyond slicing.

The main theses of my work are:

1. PDG-based information flow control is useful, practically applicable and relevant.
2. Data-flow analysis can be systematically applied to program dependence graphs.
3. Data-flow analysis on PDGs can be practically conducted.

ISBN 978-3-7315-1275-2

9 783731 512752

Gedruckt auf FSC-zertifiziertem Papier