Università
Ca'Foscari
Venezia

Dottorato di ricerca
in Informatica
Scuola di dottorato in Scienze e Tecnologie
Ciclo XXIV
(A.A. 2010 - 2011)

Information Flow Analysis by Abstract Interpretation

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di dottorato di Matteo Zanioli, matricola 955626

Coordinatore del Dottorato	**Tutore del dottorando**
Prof. Antonino Salibra	**Prof. Agostino Cortesi**
	Prof. Radhia Cousot

Università Ca' Foscari di Venezia

Dipartimento di Scienze Ambientali, Informatica e Statistica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Information Flow Analysis by Abstract Interpretation

Matteo Zanioli

Supervisors
Prof. Agostino Cortesi
Prof. Radhia Cousot

PhD Coordinator
Prof. Antonino Salibra

March, 2012

Author's Web Page: `http://www.dsi.unive.it/~zanioli`

Author's e-mail: `zanioli@dsi.unive.it`

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: `http://www.dsi.unive.it`

To Annarita, who offered me
unconditional love and support
throughout the course of this thesis.

Abstract

Protecting the confidentiality of information stored in a computer system or transmitted over a public network is a relevant problem in computer security. The goal of this thesis is to provide both theoretical and experimental results towards the design of an information flow analysis for the automatic verification of absence of sensitive information leakage.

Our approach is based on Abstract Interpretation, a theory of sound approximation of program semantics. We track the dependencies among program's variables using propositional formulae, namely the Pos domain. We study the main ways to improve the accuracy (by combination of abstract domains) and the efficiency (by combination of widening and narrowing operators) of the analysis. The reduced product of the logical domain Pos and suitable numerical domains yields to an analysis strictly more accurate with respect to the ones already in the literature. The modular construction of our analysis allows to deal with the trade-off between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators.

Finally, we introduce Sails, a new information flow analysis tool for mainstream languages like Java, that does not require any manual annotation. Sails combines the information leakage analysis with different heap abstractions, inferring information leakage over programs dealing with complex data structures too. We applied Sails to the analysis of the SecuriBench-micro suite and the preliminary experimental results outline the effectiveness of our approach.

Sommario

Proteggere la segretezza delle informazioni nei sistemi informatici o all'interno di reti pubbliche è uno dei principali problemi riguardanti la sicurezza informatica. L'obiettivo di questa tesi è fornire sia risultati teorici che sperimentali attraverso la progettazione di un'analisi di flussi di informazioni volta a verificare che i dati sensibili rimangano tali e non vengano resi pubblici.

Il nostro approccio si fonda sull'Interpretazione Astratta, una teoria riguardante l'approssimazione della semantica dei programmi. Tracciamo le dipendenze tra le variabili attraverso formule proposizionali, in particolare usando il dominio Pos. Analizziamo i principali metodi per incrementare la precisione (tramite la combinazione di domini astratti) e l'efficienza (tramite l'utilizzo degli operatori di widening and narrowing) dell'analisi. Il prodotto ridotto tra il dominio logico Pos ed opportuni domini numerici fornisce un analisi più accurata rispetto quelle presenti in letteratura. La costruzione modulare della nostra analisi permette di gestire al meglio il trade-off tra efficienza e precisione regolando la granularità dell'astrazione e la complessità degli operatori astratti.

In fine, introduciamo Sails, un nuovo strumento per l'analisi di flussi di informazioni per linguaggi tradizionali come Java, che non richiede nessuna annotazione manuale. Sails combina l'analisi dei flussi con differenti astrazioni dell'heap, inferendo i flussi anche su programmi che utilizzano strutture dati complesse. Abbiamo poi analizzato con Sails la suite SecuriBench-micro ottenendo dei risultati preliminari che hanno confermato l'efficacia del nostro approccio.

Résumé

Protéger la confidentialité de l'information numérique stockée ou en transfert sur des réseaux publics est un problème récurrent dans le domaine de la sécurité informatique. Le but de cette thèse est de fournir des résultats théoriques et expérimentaux sur une analyse de flue permettant la vérification automatique de l'absence de fuite possible d'information sensible.

Notre approche est basée sur la théorie de l'Interprétation Abstraite et consiste à manipuler une approximation de la sémantique des programmes. Nous détectons les différentes dépendances entre les variables d'un programme en utilisant des formules propositionnelles avec notamment le domaine Pos. Nous étudions les principales façon d'améliorer la précision (en combinant des domaines abstraits) et l'efficacité (en associant des opérateurs d'élargissement et de rétrécissement) de l'analyse. Le produit réduit du domaine logique Pos et d'un domaine numérique choisi permet une analyse strictement plus précise que celles précédemment présentent dans la littérature. La construction modulaire de notre analyse permet de choisir un bon compromis entre efficacité et précision en faisant varier la granularité de l'abstraction et la complexité des opérateurs abstraits.

Pour finir, nous introduisons Sails une nouvelle analyse de flue destinée à des langages de haut niveau sans annotation tel que Java. Sails combine une analyse de fuite possible d'information à différentes abstraction de la mémoire (du tas), ce qui lui permet d'inférer des résultats sur des programmes manipulant des structures complexes. De premiers résultats expérimentaux permettent de pointer l'efficacité de notre approche en appliquant Sails à l'analyse de SecuriBench-micro.

Acknowledgments

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

First of all, I would like to thank my PhD advisors, Agostino Cortesi and Radhia Cousot. They introduced me to the field of Abstract Interpretation and supported my work throughout all my thesis. Their encouragements and suggestions were very helpful to me.

David A. Schmidt and Roberto Giacobazzi accepted to be the reviewers of my thesis: I am very proud of that and I thanks them for the time spent to read and comments my work.

During my thesis, I had the opportunity to work in the Abstract Interpretation group at École Normale Supérieure. I would like to thank all the members for the discussions about Abstract Interpretation and the suggestions which they gave me about my work: Patrick Cousot, Jérôme Feret, Antoine Miné, Xavier Rival, Julien Bertrane, Axel Simon, Liqian Chen and Ferdinanda Camporesi.

Special thanks go to Pietro Ferrara. I met him when I started my PhD and since than he has always helped me. I'm very grateful to him for everything he did it.

Many thanks to all my french friends, and in particular Miriam, Federico, Jérémy, Vincenzo, Cherine, Mohamed, Tao and Enrique. I will remember the time I spent in Paris with them forever.

I will always be indebted to all the people who shared with me the doctoral studies: it was a pleasure to work with Luca, Matteo, Raju, Gian-Luca, Alvise and the others Ph.D. students.

I am particularly grateful to my family that supported me all along my life and in particular in these last three years. In particular, I want to mention my mother Maria Teresa, my father Antonio and my brothers Andrea and Alberto.

Last but not least, my deepest thank goes to Annarita, that strongly encouraged and sustained me and my work.

Contents

1 Introduction **1**
1.1 Key Concepts . . . 1
1.2 Contribution . . . 2
1.2.1 Dependency analysis . . . 2
1.2.2 Efficiency and Accuracy . . . 3
1.2.3 Domains' combination . . . 3
1.2.4 Implementation . . . 3
1.3 Overview of the Thesis . . . 4

2 Computer Security and Information Flow **5**
2.1 Introduction . . . 5
2.2 Noninterference . . . 6
2.3 Information Flow . . . 7
2.3.1 Type Systems . . . 7
2.3.2 Control Flow . . . 8
2.3.3 Abstract Interpretation . . . 8
2.4 Implementation . . . 8
2.5 Declassification . . . 9
2.6 Conclusions . . . 10

3 Abstract Interpretation Theory **11**
3.1 Introduction . . . 11
3.2 General Definitions . . . 11
3.2.1 Set and Lattice Theory . . . 11
3.2.2 Functions, Fixpoints and Traces . . . 12
3.2.3 Abstract Interpretation . . . 14
3.3 Combination of Domains . . . 17
3.4 Widening and Narrowing Operators . . . 18
3.4.1 Introduction . . . 18
3.4.2 Widening Operator . . . 19
3.4.3 Narrowing Operator . . . 22
3.4.4 Combination of Widening and Narrowing Operators . . . 24
3.5 Conclusions . . . 35

4 Information Flow Analysis by Abstract Interpretation **37**
4.1 Introduction . . . 37
4.2 The Concrete Domain . . . 37

4.2.1 The Language . . . 37
4.2.2 Concrete Domain . . . 39
4.3 Abstract Domain of Propositional Formulae . . . 41
4.3.1 Propositional Formulae . . . 41
4.3.2 Propositional Formulae Domain . . . 43
4.3.3 Abstract Domain for Pos . . . 44
4.3.4 An Instrumented Concrete Domain . . . 46
4.4 Properties . . . 48
4.4.1 Confidentiality . . . 49
4.4.2 Integrity . . . 49
4.5 Complexity of the Analysis . . . 49
4.6 Conclusions . . . 50

5 Enhancing the Information Flow Analysis by Combining Domains 51
5.1 Introduction . . . 51
5.2 Numerical Domains . . . 51
5.2.1 Intervals . . . 51
5.2.2 Karr Analysis . . . 52
5.2.3 Polyhedra . . . 52
5.2.4 Octagons . . . 53
5.3 The Reduced Product of Pos and Numerical Domains . . . 53
5.3.1 Efficiency and Accuracy: Complexity of the Analysis . . . 55
5.4 Further Refinements . . . 55
5.4.1 Subformulae Elimination . . . 56
5.5 Conclusions . . . 57

6 Experimental Results: SAILS 59
6.1 Introduction . . . 59
6.2 Presentation of Sample . . . 60
6.2.1 Scope of the Analyzer . . . 60
6.2.2 Architecture . . . 60
6.3 Integrating the Information Flow Analysis . . . 60
6.3.1 Analysis extension for Object Oriented Languages . . . 62
6.3.2 Implementation Choices . . . 63
6.3.3 Example . . . 66
6.3.4 Analysis Results . . . 67
6.4 Current Limits of the Analyzer . . . 68
6.5 Comparison with other Tools . . . 68
6.5.1 JIF - Java Information Flow . . . 69
6.5.2 Julia - Software Verification for Java and Android . . . 70
6.6 Conclusions . . . 71

7 Future Works **77**
7.1 Information Flow Analysis of JavaScript Code 77
7.2 Declassification 81
7.3 Multithreading 81
7.4 Conclusions 82

8 Conclusions **85**

A Dependency Analysis - Formal Proofs **87**

B Widening and Narrowing - Formal Proofs **89**

Bibliography **105**

List of Figures

4.1 Definition of the concrete transition semantics 42
4.2 Dependency analysis example 44
4.3 Definition of abstract transitional semantics 46

5.1 Reduce product example 54
5.2 Definition of transitional semantics for complex formulae 58

6.1 The structure of Sample 61
6.2 Definition of transition semantics for object oriented languages 63
6.3 Definition of abstract transition semantics for object oriented languages 73
6.4 A CFG not supported by Sails 74
6.5 A motivating example 74
6.6 The initial state of the heap abstraction 75
6.7 Simple strings example 76

7.1 Abstract semantics for JavaScript expressions - Part 1 83
7.2 Abstract semantics for JavaScript expressions - Part 2 84

List of Tables

4.1 Definition of the syntax for simple language 38
4.2 Definition of initial label function 39
4.3 Definition of final label function 39
4.4 Definition of action function 40
4.5 Definition of variables functions 40
4.6 Evaluation of expression 41
4.7 Evaluated of boolean conditions 41
4.8 Definition of BV function 45

6.1 Syntax for Object Oriented Languages 72
6.2 SecuriBench-micro suite 75
6.3 Jif case studies 76

7.1 JavaScript expressions 78
7.2 Table expression - identifier 79

1 Introduction

Since the birth of computer science, writing correct programs has always been considered a great challenge. Nowadays, another important aspect is requested for programs: the concept of security. The terms security means the collective processes and mechanisms by which sensitive and valuable information are protected from publication or tampering through unauthorized activities. Moreover, software complexity seems to follow Moore's law and grow exponentially with time, making it harder to debug and verify. In order to ensure the correctness and security of programs there are various formal methods which try to address these problems by providing mathematically sound techniques that guarantee a full coverage of all program behaviors. In this thesis, we wish to contribute to the field of formal methods used in the verification of the security of programs.

1.1 Key Concepts

Program Semantics *Semantics* is the branch of computer science devoted to associating a mathematical meaning to computer programs, providing a model of all possible behaviors in interaction with any possible environment and allowing formal reasoning about program and their properties [146, 134, 117].

Static Analysis The term *static analysis* denote the analysis of computer software performed without run the program. This concept is strictly related with the term *static analyzer*. Unlike testing or debugging, static analyzers can reason about infinite sets of unbounded computations in arbitrary contexts. The perfect static analyzer, by Rice's theorem, cannot exist, therefore all static analyzers will use some approximations. A sound static analyzer is one for which approximations do not compromise the truth of its results: if can output either a definite "yes", meaning that the property is indeed true regardless of approximations, or "I don't know", meaning that the approximations prevent the analyzer from issuing a definite answer [115].

Abstract Interpretation *Abstract Interpretation* is a mathematical theory of the approximation of program semantics developed by Patrick and Radhia Cousot about 30 years ago [48, 53]. It formalizes the idea that one formal proof can be done at some level of abstraction where irrelevant details are ignored. It can be applied to approximate undecidable or very complex problems in computer science. In particular, applied to static analysis of programs, abstract interpretation allows to approximate a concrete semantics with an abstract one [132, 133]. The main concepts, in this field, are the correspondence between concrete and abstract semantics, through Galois connection/insertion, and the feasibility of a fixed point computation of the abstract semantics, through the combination of widening and narrowing operators. Unlike other formal methods for reasoning on programs, once an abstract domain is designed, the static analysis performs fully automatically and directly on the source code.

Information Flow The focus of this thesis is the *information flow analysis* applied to the computer programs. Suppose that some sensitive information is stored on a computer system. How can we prevent it from being leaked improperly? Limit the access to the information by using encryption prevents that information being *released*, but it does not prevent it from being *propagated*. The approach of secure information flow analysis involves performing a static analysis of the program with the goal of proving that it will not leak sensitive information. If the program passes the analysis, then it can be executed safely [139].

1.2 Contribution

Our main goal during this thesis research is to design a static analysis tool, firmly grounded mathematically and of practical interest, focused on secure information flow. In order to achieve this goal, we combine, in a novel way, techniques generally applied in the context of generic code analysis, to solve a specific security problem: the detection of information leakage. We look into the possible way to perform more precise analysis. Then, we merge the results of different analysis to improve the results and, finally, we develop a tool which which works with mainstream languages like Java and it does not require any manual annotation.

1.2.1 Dependency analysis

The approach of information flow analysis involves performing a static analysis of the program with the aim of proving that there will not be leaks of sensitive information. We propose a new domain based on propositional formulae in order to analyze the dependencies among variables. The resulting analysis tracks all the dependencies in the program, but it is not very accurate. In fact the obtained results contain different kind of false alarms which makes the analysis unusable in real cases. Hence, we need some methods to increase the precision of the analysis and then improve the results.

1.2.2 Efficiency and Accuracy

The abstract interpretation framework allows to define the program analysis at various granularities. The main ways to get it are the combination of widening and narrowing operators, and the combination of different abstract domains.
Widening and narrowing operators play a crucial role in particular when infinite abstract domains are considered to ensure the scalability of the analysis to large software system. We report a formal definition of the widening and narrowing operations already introduced in the literature, the proof that the widening and narrowing operators are preserved by abstraction and an indication as how to construct widening operators for a product domain such as the reduced and cartesian products. Moreover we prove that, for Galois Insertions, widening and narrowing operators are preserved by abstraction and we show how the operators can be combined in the cartesian and reduced product of abstract domains.
One of the key-concepts in which abstract interpretation is based consists in the correspondence between concrete and abstract semantics through Galois connections/insertions and the possibility to combine different abstractions to each other. In this case the more precise combination is the reduced product which permits to each analysis in the abstract composition to benefit from the information brought by the other analyses.

1.2.3 Domains' combination

In order to obtain more accurate results we combine the variables dependency analysis, based on propositional formulae, and variables' value analysis, based on numerical domain, through a reduced product. The obtained analysis is strictly more accurate than the previous one and it gives the possibility to be used in practical cases. Moreover, its modular construction allows to deal with the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators.

1.2.4 Implementation

We introduce a new tool, called Sails, that combines Sample, a generic static analyzer, and our dependency analysis. Differently from the other tools in literature, Sails does not require to modify the original language, since it works with mainstream languages like Java, and it does not require any manual annotation. Sails can combine the information leakage analysis with various heap abstractions, inferring information leakage over programs dealing with complex data structures. Moreover, we applied Sails to the analysis of the SecuriBench-micro suite and the results show the effectiveness of our approach.

1.3 Overview of the Thesis

This thesis is organized as follows. Chapter 2 first gives the concepts of computer security, information flow, noninterference, declassification and then presents the main issues which we want to deal with this thesis. In Chapter 3 we recall the formal framework of Abstract Interpretation and we shows an overview about methods to refine the results: combination of different domains and the use of narrowing operator, in combianation with widening operator. In Chapter 4 we introduce the dependency analysis by Abstract Interpretation using Pos domain. The results obtained in this last chapter is not enough precise, thus Chapter 5 presents some numerical domains and it shows how we can combine them with our dependency analysis presented in Chapter 4, in order to refine the results. Moreover, in the same chapter, we propose an extension, using more complex formulae, which permits a further improvements of results. Chapter 6 shows some experimental results: we implemented a tool called Sails (Static Analysis of Information Leakage with Sample), we tested it over a set of web applications established as security and performance benchmarks and we compared it with other existing tools. In Chapter 7 we show the on going work; an extension to the dependency analysis to analyze JavaScript language, the implementation of declassification and some consideration about the application of our analysis in multithreading programs. Finally, in Chapter 8 we report the conclusions.

2 Computer Security and Information Flow

2.1 Introduction

Protecting data stored in a computer system or transmitted over a public network is a relevant issue in computer security.

In literature, attacks are commonly classified into three categories known as the *CIA* of computer security [57, 83]:

- *Confidentiality*, the attacker attempts to steal confidential information.
- *Integrity*, the attacker attempts to illegally alter parts of the system.
- *Availability*, the attacker attempts to disrupt the normal operation of a system.

These three categories of attacks are related and usually results of attacks in one category can be used to assist another attack in an other category. Information flow control tracks how information propagates through the program during execution to make sure that the program handles the information securely. More precisely, the two related categories with information flow analysis are confidentiality and integrity.

Most of the methods applied to secure information define some policy about the secrecy of the messages during the phase of information moving, for example by some particular cryptography protocols. In this way, the network communication between two peers using a fresh shared secret key prevents a malicious user to steal information by "listening" to the network traffic. Unfortunately we don't have any guarantee that one parts reveal in the clear all the exchanged messages. Secure information flow tracks data propagation: an information flow policy specifies the security classes (or security levels) for data in a system and a flow relation declares the legal path which the information can follow among the given security classes. Any information flow analysis involves performing a static analysis of the program with the aim of detecting sensitive information leakage. The starting point in secure information flow analysis is the classification of program variables into different security levels. In the simplest case, two levels are used: public (or `low`, `L`) and secret (or `high`, `H`). The main purpose is to prevent leak of sensitive information from an hight variable to a lower one. More generally, we might work with a lattice of security levels, and we would aim to ensure that sensitive information flows only upwards in the lattice [56, 20].

Information flow analysis could also be applied in other different cases. For example, if we consider some variables as containing possibly *tainted* information, then we may wish to prevent that information from such variables flowing into *untainted* variables [118]. Newsome and Song [114] used a lattice with *untainted* $\leq$ *tainted* to detect worms via a dynamic taint analysis.

This chapter provides general background on information flow security starting from the generic definition of non-interference property and introducing the most relevant literature about information flow analysis.

2.2 Noninterference

The first static information flow certification mechanism has been presented in 1977 by Denning and Denning [57] and was implemented by instrumenting the language semantics to detect any leakage. They provide a certification mechanism for verifying the secure flow of information. A program is certified if it does not involve flows in violation of the flow policy. There is an information flow from object `x` to object `y` whenever the information stored in `x` is transferred to, or used to derive information transferred to, object `y`. More generally, data confidentiality demands that private informations are never revealed to someone who is not authorized to access them. A program aiming at preserving data confidentiality can access and modify secret information but must not reveal anything about such data in its public outputs: confidential data must not influence public ones so that variations on private data does not produce any observable difference in the outputs. The prevailing basic semantic notion of secure information flow is *non-interference*. This concept was introduced in the 1982 by Goguen and Meseguer in [71] as follows: *"one group of users/processes/variables, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users/processes/variables can see"*. The idea behind non-interference is that someone observing the final values of public variables cannot conclude anything about the initial values of secret variables [139]. As the field has matured, numerous variations of noninterference [126], as well as other semantic characterizations have been explored [129].

Recently, information integrity has received attention [8, 23, 89]. Integrity has frequently been seen as the dual of confidentiality [22], through it can be argued that this description might ignore other important facets [23]. One important aspect of integrity lies in its interaction with *declassification* in order to prevent the attacker from controlling what information is declassified [112].

2.3 Information Flow

The language-based approach [126] is the most considered in the literature: this means that the information flow analysis takes place within computations of programs. In this case, illegal flows may only occur transferring information between variables in a given language. Such insecurity interferences may manifest themselves in different ways. For example, an information leak could happen due to an illegal assignment, e.g. public := private, or, more subtly through the program control flow:

$$\text{if secret} = 0 \text{ then public} := 0 \text{ else public} := 1.$$

Program can also leak information through their termination or nontermination. Consider while secret! = 0 do skip, it is clear that whenever the program terminates the value of secret is zero, therefore the private information is revealed by the fact that the computation terminates or diverges.
Mechanisms for signaling information through a computing system are known as channels. The covert channels can be divided into several categories [126]:

- implicit and explicit flows, through the control structure of a program;
- termination channels, through the termination or nontermination of a computation;
- timing channels, through the time at which an action occurs;
- probabilistic channels, through the changing the probability distribution of observable data;
- resource exhaustion channels, through the possible exhaustion of a finite, shared resource;
- power channels, through the power consumption.

2.3.1 Type Systems

There is a widespread literature on methods and techniques for checking secure information flows in software, but, probably, the most used technique consists in type systems [119, 148, 111, 76, 2]. In a security-typed language Volpano, Irvine and Smith [148] were the first to develop a type system to enforce information flow policies, where a type is inductively associated at compile-time with program statements in such a way that well-typed programs satisfy the non-interference property. The authors formulated the certification conditions of Denning's analysis [57, 123] as a simple type system for a deterministic language: basically, a formal system of type inference rules for making judgments about programs. More generally, type-based approaches are designed such that well-typed programs do not leak secrets.

A type is inductively associated at compile-time with program statements in such a way that any statement showing a potential low disclosing secrets is rejected. Type systems that enforce secure information flow have been designed for various languages, e.g. [138, 26, 120, 55, 154] and they have been used in different applications. Some of these approaches are, for example, applied to specific system, e.g. mobile ambients [27, 28], or to specific programs, e.g. written in VHDL [143], where the analysis of information flow is closely related to the context. Moreover, the secure information flow problem was also handled in different situation, for example with multi-threaded programs [140] or with programs that employ explicit cryptographic operations [66, 7].

2.3.2 Control Flow

A different approach is the use of standard control flow analysis to detect information leakage, e.g. [25, 83, 5, 88]. The idea, of this technique, is to conservatively find the program paths through which data may flow. Generally, the data flow analysis approach consists in a translation of a given program to capture and facilitate reasoning about the possible information flows. For example, Leino and Joshi, in [83], showed an application based on semantics, deriving a first-order predicate whose validity implies that an attacker cannot deduce any secure information from observing the public inputs, outputs and termination behavior of the program.

2.3.3 Abstract Interpretation

The use of Abstract Interpretation in language-based security is not new [109], even though there aren't many works that use the lattice of abstract interpretations for evaluating the security of programs (for example [153]).
The main work about information flow analysis by Abstract Interpretation is [68], where Giacobazzi and Mastroeni generalized the notion of non-interference making it parametric relatively to what an attacker can observe and use it to model attackers as abstractions. A program semantics was characterized as an abstract interpretation of its maximal trace semantics in the corresponding transition system. The authors gave a method for verifying abstract non-interference and they proved that checking abstract non-interference is a standard static program analysis problem. This method allows both to compare attackers and program secrecy by comparing the corresponding abstractions in the lattice of abstract interpretations, and to design automatic program certification tools for language-based security.

2.4 Implementation

Despite the number of works about information flow analysis, the implementations are very few. In early 2000, some works began the control of sensitive information in realistic languages [18, 110, 120], but, as far as we know, the main implementations are Jif [9] and Flow CAML [137].

According to [139], in exploring this issue further, it seems be helpful to distinguish between two different application scenarios: *developing secure software* and *stopping malicious software*. The first scenario is based on to secure information flow analysis to help the development of software that satisfies some security properties. In this case, the analysis serves as a program development tool. The static analysis tool would alert the programmer of potential leaks so that the developer could rewriting the code as necessary. An example of this scenario can be found in [9], where Askarov and Sabelfeld discusses the implementation of a "mental poker" protocol in Jif. In the second scenario, instead, the secure information flow analysis is used as a kind of filter to stop malicious software. In this case, we might imagine analyzing a piece of untrusted code before executing it, with the goal of guaranteeing its safety. This is much more challenging than first scenario: probably we would not have access to the source code and we would need to analyze the binary code. Analyzing binaries is more difficult than analyzing source code and has not received much attention in the literature (a Java bytecodes analysis is performed, for instance, by Barthe and Rezk in [19]).

Generally, secure information flow analysis has focused on enforcing noninterference, but in both scenarios the noninterference property does not seem to be what we want. Noninterference requires absolutely no flow of information and in many practical situations "small" information leaks are acceptable in practice. For example, a password checker must allow a user to enter a purported password, which it will either accept or reject. Of course, rejecting a password leaks some information about the correct password, by eliminating one possibility. Similarly, encrypting some secrets information would seems to make them public, but there is a flow of information from the plaintext to the ciphertext, since the ciphertext depends on the plaintext. We can conclude that, in many practical situations, enforcing non-interference on a static lattice of security levels is too heavy-handed. At the same time, it seems difficult to allow "small" information leaks without allowing a malicious program to exploit such loopholes to leak too much.

2.5 Declassification

For many applications a complete separation between secret and public is too restrictive. Consider for instance the login screen of an operating system; when a user tries logging in the response of the system gives away information about the password. If access is refused we know that the attempted password was not the correct one. Even though this gives away partial information about the password, we deem this secure. Another important class of examples is data aggregation. Consider for instance a program that computes average salaries; even though each individual salary may be secret we might want to be able to publish the average.

We need a way to declassify information, i.e. lowering the security classification of selected information [17]. Sabelfeld and Sands [128] identify four different dimensions of declassification: *what* is declassified, *who* is able to declassify, *where* the declassification occurs and *when* the declassification takes place.

What is important to be able to specify what information is declassified, e.g., the four last digits of a credit card number. Policies for partial release must guarantee an upper limit on what information is released. Some works related to the *what* dimension are [94, 127, 69]. Another important aspect is *who* controls the release of information. This pertains, in particular, to information integrity and has been investigated in the context of robustness [112], which controls on whose behalf declassification may occur. Works about this dimension are [95, 111]. Sabelfeld and Sands identify two principal forms of released locality (so the *where* dimension). Related to the *what* and *when* dimension, the where dimension is the most immediate interpretation of where in terms of code locality. The other form is level locality, describing where information may flow relative to the security levels of the system. Some works on this topic are [103, 100]. The temporal dimension of declassification is about *when* information is leaked. Always in [128], the authors identify three classes of temporal release classifications: time-complexity based, probabilistic and relative. The first one indicates that information will not be released until after a certain time, with the second one we can talk about the probability of a leak being very small and in the last one the leakage is related to other events in the system. Some works about this dimension are [149, 32].

2.6 Conclusions

A major challenge for secure information flow analysis is to develop a good formalism for specifying useful information flow policies that are more flexible than noninterference. The formalism must be general enough for a wide variety of applications, but not too complicated for users to understand.

In the rest of the thesis we combine various techniques, generally applied in the context of generic code analysis, in a novel way to provide a different approach to analyze the information flow within computer programs, which permits us to obtain more accurate results than existed systems. Moreover, we provide a new tool (called Sails, Chapter 6) to perform an information flow analysis on Scala language or Java Bytecode.

3 Abstract Interpretation Theory

3.1 Introduction

In this chapter, we introduce the mathematical background used in the rest of the thesis, in particular we recall some basic notation and some well-know results in lattice, fixpoint and Abstract Interpretation theory. Moreover, we investigate the combination of widening and narrowing operators. One of the main features of Abstract interpretation framework consists on defining the program analysis at various granularities, achieving a different trade-offs between efficiency and precision. In some practical cases, it is possible to refine the results through the combination of different domains and the combination of widening and narrowing operators. When the abstract domain does not satisfy the ascending chain condition, widening and narrowing operators should be used to ensure convergence and tune the cost/precision compromise. In this chatpter we investigate also the possible combination of wdening and narrowing operators ([41], extension of [35]).

3.2 General Definitions

3.2.1 Set and Lattice Theory

A *partially ordered set* (or *poset*) $\langle \mathsf{D}, \sqsubseteq \rangle$ is a non-empty set D together with a partial order $\sqsubseteq$, a binary relation such that it is:

- reflexive: $\forall \mathsf{d} \in \mathsf{D} : \mathsf{d} \sqsubseteq \mathsf{d}$
- antisymmetric: $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D} : \mathsf{d}_0 \sqsubseteq \mathsf{d}_1 \wedge \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \Rightarrow \mathsf{d}_0 = \mathsf{d}_1$
- transitive: $\forall \mathsf{d}_0, \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D} : \mathsf{d}_0 \sqsubseteq \mathsf{d}_1 \wedge \mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \Rightarrow \mathsf{d}_0 \sqsubseteq \mathsf{d}_2$

Given $\mathsf{D}' \subseteq \mathsf{D}$, $\mathsf{d} \in \mathsf{D}$ is an *upper bound* of D' iff $\forall \mathsf{d}' \in \mathsf{D}' : \mathsf{d}' \sqsubseteq \mathsf{d}$. It is the *least upper bound* (*lub*), denoted by $\bigsqcup \mathsf{D}'$, if $\forall \mathsf{d}' \in \mathsf{D}$ such that d' is a an upper bound of D', then $\mathsf{d} \sqsubseteq \mathsf{d}'$. Symmetrically we can define the *lower bounds* and *greatest lower bounds* (*glb*), denoted by $\bigsqcap \mathsf{D}'$.
The poset $\langle \mathsf{D}, \sqsubseteq \rangle$ has a *top element* (or *greatest element*) $\top$ iff $\top \in \mathsf{D} \wedge \forall \mathsf{d} \in \mathsf{D} : \mathsf{d} \sqsubseteq$

$\top$. Dually, it has a *bottom element* (or *least element*) $\bot$ iff $\bot \in \mathsf{D} \wedge \forall \mathsf{d} \in \mathsf{D} : \bot \sqsubseteq \mathsf{d}$. A poset with a least element will be called a *pointed poset* and it is denoted by $\langle \mathsf{D}, \sqsubseteq, \bot \rangle$. Note that any poset can be transformed into a pointed poset by adding a new element that is smaller then everyone.
A *cpo* is a poset which is *complete*, that is, every increasing chain of elements $(\mathsf{X}_i)_{i\in\mathbb{N}}, i \leq j \Rightarrow \mathsf{X}_i \sqsubseteq \mathsf{X}_j$ has a least upper bound $\sqcup_{i\in\mathbb{N}}\mathsf{X}_i$, which is called the *limit* of the chain. Note that a cpo is always pointed as the least element can be defined by $\bot \overset{\text{def}}{=} \bigsqcup \emptyset$.

A *lattice* $\langle \mathsf{D}, \sqsubseteq, \sqcup, \sqcap \rangle$ is a poset where each pair of elements $\mathsf{a}, \mathsf{b} \in \mathsf{D}$ has a least upper bound, denoted by $\mathsf{a} \sqcup \mathsf{b}$, and a greatest lower bound, denoted by $\mathsf{a} \sqcap \mathsf{b}$. A lattice is said to be *complete* if any set $\mathsf{D}' \subseteq \mathsf{D}$ has a least upper bound. A complete lattice is always a cpo; it has both a least element $\bot \overset{\text{def}}{=} \bigsqcup \emptyset$ and a greatest element $\top \overset{\text{def}}{=} \bigsqcup \mathsf{D}$; also, each set $\mathsf{D}' \subseteq \mathsf{D}$ has a greatest lower bound $\bigsqcap \mathsf{D}' \overset{\text{def}}{=} \bigsqcup \{\mathsf{X} \in \mathsf{D} \mid \forall Y \in \mathsf{D}', X \sqsubseteq Y\}$. In this case, it is denote by $\langle \mathsf{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$. An important example of complete lattice is the *power-set* $\langle \wp(\mathsf{S}), \subseteq, \cup, \cap, \emptyset, \mathsf{S} \rangle$ for any set S. A *join semi lattice* and a *meet semi lattice* are poset $\langle \mathsf{D}, \sqsubseteq \rangle$ such that each pair of elements $\mathsf{a}, \mathsf{b} \in \mathsf{D}$ has, respectively, least upper bound $(\mathsf{a} \sqcup \mathsf{b})$ and great lower bound $(\mathsf{a} \sqcap \mathsf{b})$.

3.2.2 Functions, Fixpoints and Traces

A *function* is a relation r which relates each element of the domain to at most one element of the co-domain, i.e. if $(\mathsf{x}, \mathsf{y}_0) \in r$ and $(\mathsf{x}, \mathsf{y}_1) \in r$, then $\mathsf{y}_0 = \mathsf{y}_1$. Therefore, given an element $\mathsf{x} \in \mathit{dom}(r)$ we denote the element in the co-domain by $r(\mathsf{x})$.
In order to define functions we use the *lambda notation*: by $f = \lambda\mathsf{x}.\mathsf{expr}$ we denote the function f which maps x to the value of the expression expr where x is a free variable. We will sometimes use the explicit notation $[\mathsf{x}_0 \mapsto \mathsf{expr}_0, \ldots, \mathsf{x}_\mathsf{n} \mapsto \mathsf{expr}_\mathsf{n}]$ to denote the application that associates the value of expr_i to x_i, or the notation $f[\mathsf{x} \mapsto \mathsf{y}]$ to represent a function that behaves as f except for the input x, for which it returns y. By the notation $f : \mathsf{X} \rightarrow \mathsf{Y}$ we mean that the domain of the function f is included in X and its co-domain is included in Y. Let $f : \mathsf{X} \rightarrow \mathsf{Y}$ and $g : \mathsf{Y} \rightarrow \mathsf{Z}$, then $g \circ f : \mathsf{X} \rightarrow \mathsf{Z}$ represents the composition of functions f and g, i.e. $\lambda\mathsf{x}.g(f(\mathsf{x}))$.
Let $\langle \mathsf{X}, \sqsubseteq_\mathsf{X} \rangle$ and $\langle \mathsf{Y}, \sqsubseteq_\mathsf{Y} \rangle$ be two posets, a function $f : \mathsf{X} \rightarrow \mathsf{Y}$ is:

- *monotonic* iff $\forall \mathsf{x}_0, \mathsf{x}_1 \in \mathsf{X} : \mathsf{x}_0 \sqsubseteq_\mathsf{X} \mathsf{x}_1 \Rightarrow f(\mathsf{x}_0) \sqsubseteq_\mathsf{Y} f(\mathsf{x}_1)$
- *join preserving* iff $\forall \mathsf{x}_0, \mathsf{x}_1 \in \mathsf{X} : f(\mathsf{x}_0 \sqcup_\mathsf{X} \mathsf{x}_1) = f(\mathsf{x}_0) \sqcup_\mathsf{Y} f(\mathsf{x}_1)$ where $\sqcup_\mathsf{X}$ and $\sqcup_\mathsf{Y}$ are, respectively, the lub on $\langle \mathsf{X}, \sqsubseteq_\mathsf{X} \rangle$ and $\langle \mathsf{Y}, \sqsubseteq_\mathsf{Y} \rangle$
- *complete join preserving* (or *complete* $\sqcup$-*morphism*) iff $\forall \mathsf{X}' \subseteq \mathsf{X}$ such that $\bigsqcup_\mathsf{X} \mathsf{X}'$ exists, then $f(\bigsqcup_\mathsf{X} \mathsf{X}') = \bigsqcup_\mathsf{Y} f(\mathsf{X}')$
- *complete meet preserving* (or *complete* $\sqcap$-*morphism*) iff $\forall \mathsf{X}' \subseteq \mathsf{X}$ such that $\bigsqcap_\mathsf{X} \mathsf{X}'$ exists, then $f(\bigsqcap_\mathsf{X} \mathsf{X}') = \bigsqcap_\mathsf{Y} f(\mathsf{X}')$

- *continuous* iff for all chains $\mathsf{C} \subseteq \mathsf{X}$ we have that $f(\bigsqcup_{\mathsf{X}} \mathsf{C}) = \bigsqcup_{\mathsf{Y}}\{f(\mathsf{c} \mid \mathsf{c} \in \mathsf{C}\}$

A poset $\langle \mathsf{D}, \sqsubseteq \rangle$ satisfies the ascending chain condition (ACC) if every ascending chain $\mathsf{c}_0 \sqsubseteq \mathsf{c}_1 \sqsubseteq \cdots$ of elements in D is eventually stationary, i.e. $\exists i \in \mathbb{N} : \forall j > i : \mathsf{c}_j = \mathsf{c}_i$. Dually, a poset satisfies the descending chain condition (DCC) if there is not any infinte decreasing chain.

An *operator* $f : \mathsf{D} \to \mathsf{D}$, which is a function from a poset D to the same poset, is said to be *extensive* if $\forall \mathsf{X}, \mathsf{X} \sqsubseteq f(X)$. A *fixpoint* of an operator f is an element x such that $f(\mathsf{x}) = \mathsf{x}$. Let f be a function on a poset $\langle \mathsf{D}, \sqsubseteq \rangle$. The *set of fixpoints* of f is $\mathsf{FP} = \{\mathsf{x} \in \mathsf{D} \mid f(\mathsf{x}) = \mathsf{x}\}$. An element $\mathsf{x} \in \mathsf{D}$ is:

- a *pre-fixpoint* iff $\mathsf{x} \sqsubseteq f(\mathsf{x})$
- a *post-fixpoint* iff $f(\mathsf{x}) \sqsubseteq \mathsf{x}$

In particular, $\bot$ and $\top$ are, respectively, a pre-fixpoint and a post-fixpoint for all operators. Moreover, we denote by $lfp_{\mathsf{x}}^{\sqsubseteq} f$ the *least fixpoint* of f that is greater than x with respect the order $\sqsubseteq$ and by $gfp_{\mathsf{x}}^{\sqsubseteq} f$ the *greatest fixpoint* of f smaller than x with respect the order $\sqsubseteq$.
The existence of the least and greatest fixpoints on a monotonic map is guaranteed by the following theorems.

Theorem 3.1 (Tarski's theorem [142]). *Let $\langle \mathsf{D}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice. Let $f : \mathsf{D} \to \mathsf{D}$ be a monotonic function on this lattice. Then the set of fixpoints is a not-empty complete lattice, and:*

$$
\begin{aligned}
lfp_{\bot}^{\sqsubseteq} f &= \sqcap\{\mathsf{x} \in \mathsf{D} \mid f(\mathsf{x}) \sqsubseteq \mathsf{x}\} \\
gfp_{\mathsf{x}}^{\sqsubseteq} f &= \sqcup\{\mathsf{x} \in \mathsf{D} \mid \mathsf{x} \sqsubseteq f(\mathsf{x})\}
\end{aligned}
$$

Theorem 3.2 (Constructive version of Tarski's theorem [49]). *Let $\langle \mathsf{D}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice. Let $f : \mathsf{D} \to \mathsf{D}$ be a monotonic function on this lattice. Define the following sequence:*

$$
\begin{aligned}
f^0 &= \bot \\
f^\delta &= f(f^{\delta-1}) \textit{for every successor odrinal } \delta \\
f^\delta &= \bigsqcup_{\alpha<\delta} f^\alpha \textit{for every limit ordinal } \delta
\end{aligned}
$$

Then the ascending chain $\{f^i \mid 0 \leq i \leq \delta\}$ (where δ is an ordinal) is ultimately stationary for some $\rho \in \mathbb{N}$ that is $f^\rho = lfp_{\bot}^{\leq} f$.

An important notion, in abstract interpretation, is the concept of *trace*. Informally, we can define a trace as an ordered sequence of elements such that it is defined on the first k elements. More formally, a trace is define as follows.

Definition 3.1 (Trace [60])**.** *Given a set* S*, a trace* $\tau : \mathbb{N} \to \mathsf{S}$ *is a partial function such that:*

$$\forall i \in \mathbb{N} : i \notin dom(\tau) \Rightarrow \forall j > i : j \notin dom(\tau)$$

The definition implies that the domain of all non-empty traces is a segment of $\mathbb{N}$. The *empty trace*, i.e. the trace τ such that $dom(\tau) = \emptyset$, is denoted by ϵ. Let be S a generic set of elements, we denote by S^+ the set of all the finite traces composed of elements in S. *len* : $\mathsf{S}^+ \to \mathbb{N}$ is the function that, given a trace, returns its length. Formally: $len(\tau) = \mathsf{i} + 1 : \mathsf{i} \in dom(\tau) \wedge \mathsf{i} + 1 \notin dom(\tau)$. If $\tau = \epsilon$, then $len(\tau) = 0$. Notice that a trace can be represented by a succession of states, i.e. $\sigma_0 \to \sigma_1 \to \cdots$. We define by $\mathsf{S}^+_{\xrightarrow{\mathsf{T}}}$ the set of traces in S^+ ending with a final state with respect to the transition relation $\xrightarrow{\mathsf{T}}$, i.e. $\mathsf{S}^+_{\xrightarrow{\mathsf{T}}} = \{\sigma_0 \to \cdots \to \sigma_i \mid \sigma_0 \to \cdots \to \sigma_i \in \mathsf{S}^+, \nexists \sigma_j \in \mathsf{S}^+ : \sigma_i \xrightarrow{\mathsf{T}} \sigma_j\}$.
Given a set of initial elements $\mathsf{S_0}$ and a transition relation $\xrightarrow{\mathsf{T}} \subseteq \Sigma \times \Sigma$, the *partial trace semantics* builds up all the traces that can be obtained by starting from traces containing only a single element from $\mathsf{S_0}$ and then iteratively applying the transition relation until a fixpoint is reached.

Definition 3.2 (Partial trace semantics [50])**.** *Let* Σ *be a set of states,* $\mathsf{S_0} \subseteq \Sigma$ *a set of initial elements, and* $\xrightarrow{\mathsf{T}} \subseteq \Sigma \times \Sigma$ *a transition relation. Let* $f : \wp(\Sigma) \to (\Sigma^+ \to \Sigma^+)$ *be the function defined as:*

$$\begin{aligned} F(\mathsf{S_0}) = \lambda \mathsf{X}.\{\sigma_0 \mid \sigma_0 \in \mathsf{S_0}\} \cup \\ \{\sigma_0 \to \cdots \to \sigma_{n-1} \to \sigma n \mid \sigma_0 \to \cdots \to \sigma_{n-1} \in \mathsf{X} \wedge \sigma_{n-1} \xrightarrow{\mathsf{T}} \sigma_n\} \end{aligned}$$

The partial trace semantics is defined as

$$\mathbb{PT}[\![\mathsf{S_0}]\!] = lfp_{\emptyset}^{\sqsubseteq}\ F(\mathsf{S_0})$$

3.2.3 Abstract Interpretation

Abstract Interpretation is a mathematical theory of semantics approximation developed by Patrick and Radhia Cousot about 30 years ago [43, 48, 50]. A core principle in the abstract interpretation theory is that all kinds of semantics can be expressed as fixpoints of monotonic operators in partially ordered structures, would it be operational, denotational, rule-based, axiomatic, based on rewriting systems, transition systems, abstract machines, etc [135, 136, 102]. In this way, beside comparing already existing semantics, Abstract Interpretation allows building new semantics by applying abstractions to existing ones. A key property of the semantics designed by abstraction is that they are guaranteed to be sound, by construction. Thus, a sound and fully automatic static analyzer can be designed by starting from the non-computable formal semantics of a program language, and composing abstractions until the resulting semantics is computable. Notice that the approximation is required, then the result is correct but incomplete, i.e. if a property is not inferred in the abstract semantics, it may still be satisfied by the concrete one.

The *concrete semantics* belongs to a *concrete semantic domain* D which is a partially ordered set $\langle \mathsf{D}, \sqsubseteq \rangle$. The *abstract semantics* also belongs to a partial order $\langle \mathsf{D}^\sharp, \sqsubseteq^\sharp \rangle$, which is ordered by the abstract version $\sqsubseteq^\sharp$ of the concrete approximation order $\sqsubseteq$ and which is called *abstract semantic domain.* We denote the abstract counterparts for concrete entities by a superscript, generally $\sharp$, $\flat$ or $\natural$.

Galois connection/insertion

Let D and $\mathsf{D}^\sharp$ be two posets[1] used as semantic domains. A *Galois connection*, as introduced in [48], between D and $\mathsf{D}^\sharp$ is a function pair (α,γ) such that:

Definition 3.3 (Galois connection)**.**

1. $\alpha : \mathsf{D} \to \mathsf{D}^\sharp$ *is monotonic;*

2. $\gamma : \mathsf{D}^\sharp \to \mathsf{D}$ *is monotonic;*

3. $\forall \mathsf{X}, \mathsf{X}^\sharp, \alpha(\mathsf{X}) \sqsubseteq^\sharp \mathsf{X}^\sharp \iff \mathsf{X} \sqsubseteq \gamma(\mathsf{X}^\sharp)$.

This is often pictured as follows: $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathsf{D}^\sharp$.

As a consequence, we have $(\alpha \circ \gamma)(\mathsf{X}^\sharp) \sqsubseteq^\sharp \mathsf{X}^\sharp$, i.e. $\alpha \circ \gamma$ is *reductive*, and $\mathsf{X} \sqsubseteq (\gamma \circ \alpha)(\mathsf{X})$, i.e. $\gamma \circ \alpha$ is *extensive.* The fact that $\alpha(\mathsf{X}) \sqsubseteq^\sharp \mathsf{X}^\sharp$, or equivalently that $\mathsf{X} \sqsubseteq \gamma(\mathsf{X}^\sharp)$, formalizes the fact that $\mathsf{X}^\sharp$ is a sound approximation (or sound *abstraction*) of X. α and γ are called, respectively, *abstraction function* and *concretization function.*

If the concretization γ is one-to-one, i.e. $\alpha \circ \gamma = Id$, then (α,γ) is called a *Galois insertion*, denoted by $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\rightarrow}} \mathsf{D}^\sharp$. Design an abstract domain linked to the concrete one through a Galois insertion corresponds to choosing, as abstract elements, a subset of the concrete ones and, as the abstract order, the order induced by the concrete domain [107].

Sometimes, it is not necessary to define both the concretization and the abstraction functions in a Galois connection: the missing function can be defined in a canonical way. We can use the following theorem.

Theorem 3.3 (Canonical α, γ [52])**.**

[1]From now, a poset $\langle \mathsf{D}^{\mathsf{x}}\ \sqsubseteq^{\mathsf{x}} \rangle$ will only refereed to as D^{x} when there is no ambiguity, that is, when there is only one partial order of interest on D^{x}. The same also holds for cpo, lattice and complete lattice: the same superscript x as the one of the set D^{x} is used when talking about its order ($\sqsubseteq^{\mathsf{x}}$) ,lub ($\sqcup^{\mathsf{x}}$), glb ($\sqcap^{\mathsf{x}}$), least ($\bot^{\mathsf{x}}$) and greatest ($\top^{\mathsf{x}}$) element, when they exists.

1. *If* D *has lubs for arbitrary sets and* $\alpha : \mathsf{D} \to \mathsf{D}^\sharp$ *is a complete* $\sqcup$*-morphism, then there exists a unique concretization* γ *that forms a Galois connection* $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathsf{D}^\sharp$*. This* γ *is defined as:*

$$\gamma(\mathsf{X}) \overset{\text{def}}{=} \sqcup\{\mathsf{Y} \mid \alpha(\mathsf{Y}) \sqsubseteq^\sharp \mathsf{X}\}$$

2. *Likewise, if* $\mathsf{D}^\sharp$ *has glbs for arbitrary sets and* $\gamma : \mathsf{D}^\sharp \to \mathsf{D}$ *is a complete* $\sqcap$*-morphism, then there exists a unique* α *that forms a Galois connection* $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathsf{D}^\sharp$*. It is defined as:*

$$\alpha(\mathsf{X}) \overset{\text{def}}{=} \sqcap^\sharp\{\mathsf{Y} \mid \mathsf{X} \sqsubseteq \gamma(\mathsf{Y})\}$$

An interesting property of Galois connection is that they are compositional, i.e., the composition of two Galois connections is still a Galois connection.

Theorem 3.4 (Composition of Galois connection)**.** *Let* $\langle \mathsf{A} \sqsubseteq_\mathsf{A} \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathsf{B} \sqsubseteq_\mathsf{B} \rangle$ *and* $\langle \mathsf{B} \sqsubseteq_\mathsf{B} \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathsf{C} \sqsubseteq_\mathsf{C} \rangle$ *be two Galois connection. Then*

$$\langle \mathsf{A} \sqsubseteq_\mathsf{A} \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle \mathsf{C} \sqsubseteq_\mathsf{C} \rangle$$

Fixpoint Approximation

Usually in abstract interpretation the concrete and abstract semantics are defined as the fixpoint computation of monotonic functions.
Given the concrete and abstract semantics, respectively, $\mathbb{A} : \mathsf{D} \to \mathsf{D}$ and $\mathbb{A}^\sharp : \mathsf{D}^\sharp \to \mathsf{D}^\sharp$, where $\mathsf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathsf{D}^\sharp$, we want to prove the correctness of the abstract semantics with respect to the concrete one. The abstract semantics is sound iff for all the pre-fixpoints $\mathsf{p}^\sharp \in \mathsf{Pfp}^\sharp \subseteq \mathsf{D}^\sharp$ of $\mathbb{A}^\sharp$, we have that $\gamma \circ \mathbb{A}^\sharp[\![\mathsf{p}^\sharp]\!] \geq \mathbb{A}[\![\gamma(\mathsf{p}^\sharp)]\!]$.

When applied to the static analysis of programs, the transfer function depends on a program $\mathcal{P}$. There are many different ways to prove that an abstract semantics is sound, based on some different properties of transfer functions, concrete and abstract lattices, concretization and abstraction functions [53]. In this thesis, we will rely on the following thorem.

Theorem 3.5 (Kleene-like, join-morphism-based fixpoint approximation [48])**.** *Let* $\langle \mathsf{L}, \sqsubseteq, \sqcup \rangle$ *and* $\langle \mathsf{L}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp \rangle$ *be complete lattices. Let* $F : \mathsf{L} \to \mathsf{L}$ *and* $F^\sharp : \mathsf{L}^\sharp \to \mathsf{L}^\sharp$ *be two monotone functions with respect to* $\sqsubseteq$ *and* $\sqsubseteq^\sharp$ *respectively. Let* $\alpha : \mathsf{L} \to \mathsf{L}^\sharp$ *be a join-morphism such that* $\alpha \circ F \dot{\sqsubseteq}^\sharp F^\sharp \circ \alpha$*, where* $\dot{\sqsubseteq}^\sharp$ *is the lifting of the ordering operator* $\sqsubseteq^\sharp$*to functions. Let* $\mathsf{a} \in \mathsf{L}$ *be a pre-fixpoint of* F*. Then* $\alpha(\mathit{lfp}_{\mathsf{a}}^{\sqsubseteq}\ F) \sqsubseteq^\sharp \mathit{lfp}_{\alpha(\mathsf{a})}^{\sqsubseteq}\ F^\sharp$*.*

Properties

According to [46], by the term *program property* we mean a property of program executions, i.e. a property of program trace semantics. Given a program semantic domain S, the corresponding program properties belong to $\wp(\mathsf{S})$.

Definition 3.4 (Valid property for a program)**.** *We said that a property* $\mathsf{Prop} \in \wp(\mathsf{S})$ *is valid for a program* $\mathcal{P}$ *(or that* $\mathcal{P}$ *satisfies/has property* Prop*) if and only if property* Prop *is implied by the semantics of the program that is* $\{\mathbb{PT}[\![\mathcal{P}]\!]\} \subseteq \mathsf{Prop}$.

Abstraction is the process of considering part of the program semantic properties which are of interest in some reasoning or computation. This considered subset of all possible program properties is called the set of *abstract properties.*

3.3 Combination of Domains

Since abstract interpretation framework was defined, an its important feature has been the possibility to combine different abstractions to each other. As write in [50], the ideal method in order to construct a program analyzer consist in a separate design and implementation of various complementary program analysis frameworks which could then be systematically combined using a once for all implemented assembler. In fact, one commonly used method to create more precise abstract domains is by combining simpler ones.

There are two frequently used notions of lattice combinations in the literature: the *cartesian product* (or *direct product*), and the *reduced product* [75]. Both these combinations yield a lattice whose elements are a cartesian product of the elements of the individual lattices. The difference is that the lattice operations in the direct product are performed component-wise, while in case of the reduced product the lattice operations take into account both components simultaneously.

Cartesian Product. Let $\langle \mathsf{C}, \sqsubseteq, \sqcup \rangle$, $\langle \mathsf{D}_0^\sharp, \sqsubseteq_0^\sharp \rangle$ and $\langle \mathsf{D}_1^\sharp, \sqsubseteq_1^\sharp \rangle$ be three complete lattices. More precisely, C is a concrete domain and D_0 and D_1 are two abstract domains such that $\mathsf{C} \underset{\gamma_0}{\overset{\alpha_0}{\leftrightarrows}} \mathsf{D}_0^\sharp$ and $\mathsf{C} \underset{\gamma_1}{\overset{\alpha_1}{\leftrightarrows}} \mathsf{D}_1^\sharp$. The cartesian product of $\mathsf{D}_0^\sharp$ and $\mathsf{D}_1^\sharp$ is $\langle \mathsf{D}_0^\sharp \times \mathsf{D}_1^\sharp \rangle$ and it is in relation with C by the functions $\alpha : \mathsf{C} \longrightarrow \langle \mathsf{D}_0^\sharp \times \mathsf{D}_1 \rangle$ and $\gamma : \langle \mathsf{D}_0^\sharp \times \mathsf{D}_1^\sharp \rangle \longrightarrow \mathsf{C}$ defined as follows. Consider $\mathsf{c} \in \mathsf{C}$ and $\langle \mathsf{d}_0^\sharp, \mathsf{d}_1^\sharp \rangle \in \langle \mathsf{D}_0^\sharp \times \mathsf{D}_1^\sharp \rangle$, $\alpha(\mathsf{c}) = \langle \alpha_0(\mathsf{c}), \alpha_1(\mathsf{c}) \rangle$ and $\gamma(\langle \mathsf{d}_0^\sharp, \mathsf{d}_1^\sharp \rangle) = \gamma_0(\mathsf{d}_0^\sharp) \sqcup \gamma_1(\mathsf{d}_1^\sharp)$. In this way we find the information about both domains at the same time, but we do not learn more by performing all analyses simultaneously than by compute them one after another and finally taking their conjunctions.

Reduced Product. Differently, the advantage of the reduced product is that each analysis in the abstract composition benefits from the information brought by the other analyses. The reduced product was proposed by Cousots [50] to overcome some of the limitations of the direct product. It is based on clustering into equivalence classes the elements of the direct product having the same concretization and working on the more precise representative of each class. Formally, consider the three complete lattice $\langle \mathsf{C}, \sqsubseteq, \sqcup \rangle$, $\langle \mathsf{D}_0^\sharp, \sqsubseteq_0^\sharp \rangle$ and $\langle \mathsf{D}_1^\sharp, \sqsubseteq_1^\sharp \rangle$ and the respective relations,

$\mathsf{C} \underset{\gamma_0}{\overset{\alpha_0}{\leftrightarrows}} \mathsf{D}_0^\sharp$ and $\mathsf{C} \underset{\gamma_1}{\overset{\alpha_1}{\leftrightarrows}} \mathsf{D}_1^\sharp$. Let $\varrho : \mathsf{D}_0^\sharp \times \mathsf{D}_1^\sharp \to \mathsf{D}_0^\sharp \times \mathsf{D}_1^\sharp$ be the reduce operator, defined as follows.

$$\varrho(\langle \mathsf{d}_0^\sharp, \mathsf{d}_1^\sharp \rangle) = \sqcap\{\langle \mathsf{e}_0^\sharp, \mathsf{e}_1^\sharp \rangle \mid \gamma_0(\mathsf{e}_0^\sharp) \sqcap \gamma_1(\mathsf{e}_1^\sharp) = \gamma_0(\mathsf{d}_0^\sharp) \sqcap \gamma_1(\mathsf{d}_1^\sharp)\}$$

The reduced product domain is the domain $\mathsf{D}^\natural = \{\varrho(\langle \mathsf{d}_0^\sharp, \mathsf{d}_1^\sharp \rangle) \mid \mathsf{d}_0^\sharp \in \mathsf{D}_0^\sharp \wedge \mathsf{d}_1^\sharp \in \mathsf{D}_1^\sharp\}$. It's clear that, in this way, we obtain a domain which is more precise than the cartesian product.

Concluding, the abstract domains combination provides a good way to increase the accuracy of the analysis. In some cases, specializations of these combined domains are developed. For instance, the *open product* [40] was defined to improve the direct product by letting the domains interact, by letting operations in one domain asks queries to other domains.

3.4 Widening and Narrowing Operators

3.4.1 Introduction

Abstract Interpretation is a general theory of approximation of mathematical structures based on two main key-concepts: the correspondence between concrete and abstract semantics through Galois connections/insertions, and the feasibility of a fixed point computation of the abstract semantics, through, the combination of widening (to get fast convergence) and narrowing operators (to improve the accuracy of the resulting analysis).

Galois connections have been widely studied, yielding to a suite of general techniques to manage the combination of abstract domains, e.g. different kind of products [50, 74, 40], and more sophisticated notions like the quotient [37], the complement [36], and the powerset [70] of abstract domains, but not much attention has been given to provide general results about widening and narrowing operators.

Nevertheless, widening and narrowing operators play a crucial role in particular when infinite abstract domains are considered to ensure the scalability of the analysis to large software systems, as it has been shown in the case of the Astrée project for analysis of absence of run-time error of avionic critical software [45].

The first infinite abstract domain (that of intervals) was introduced in [47]. This abstract domain was later used to prove that, thanks to widening and narrowing operators, infinite abstract domains can lead to effective static analyses for a given programming language that are strictly more precise and equally efficient than any other one using a finite abstract domain or an abstract domain satisfying chain conditions [51].

Specific widening and narrowing operators have been also designed not only for numerical domains but also for type graphs [145], in domains for reordering CLP(RLin) programs [121], and in the analysis of programs containing digital filters [61], just to name a few. More recently, widenings have been used also to infer loop invariants inside an STM solver [125], in trace partitioning abstract domains [122] and in string analysis for string-generating programs [78].

The main challenge for widening and narrowing operators is when considering numerical domains. For instance, the original widening operator proposed by Cousot and Halbwachs [54] for the domain of convex polyhedra, has been improved by recent works by Bagnara et al [12], and further refined for the domain of pentagons by Logozzo et.al. in [93]. In [13], the authors define three generic widening methodologies for a finite powerset abstract domain. The widening operators are obtained by lifting any widening operator defined on the base-level abstract domain. The proposed techniques are instantiated on powersets of convex polyhedra, a domain for which no non-trivial widening operator was previously known.

We observed that, with the noticeable exception of [51, 13], there is still a lack of general techniques that support the systematic construction of widening and narrowing operators. This is mainly due to the fact that the definition of widening provides extremely weak algebraic properties, while it is extremely demanding with respect to convergence and termination.

This section presents the results obtained in [41], which filled this gap, and to set the ground for a systematic design of widening and narrowing operators either when they are defined on sets and when they are redefined on pairs. The advantages of suitable combinations of widening and narrowing operators are illustrated on a suite of examples, ranging from interval to powerset domains.

Note that all the proofs of the theorems presented in this Chapter are in Appendix B.

3.4.2 Widening Operator

In Abstract Interpretation, the collecting semantics of a program is expressed as a least fix-point of a set of equations. The equations are solved over some abstract domain that captures the property of interest to be analyzed. Typically, the equations are solved iteratively; that is, successive approximations of the solution is computed until a fix-point is reached. However, for many useful abstract domains, such chains can be either infinite or too long to let the analysis be efficient. To make use of these domains, abstract interpretation theory provides very powerful tools, the widening operators, that attempt to predict the fix-point based on the sequence of approximations computed on earlier iterations of the analysis on a cpo or on a (complete) lattice. The degradation of precision of the solution obtained by widening can be partly restored by further applying a narrowing operator [51].

Definition 3.5 (set-widening [50, 53]). *Let* $(\mathsf{P}, \leq)$ *be a poset. A set-widening operator is a partial function* $\nabla_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *such that*

(i) Covering: Let S *be an element of* $\wp(\mathsf{P})$. *If* $\nabla_\star(\mathsf{S})$ *is defined, then* $\forall \mathsf{x} \in \mathsf{S}$, $\mathsf{x} \leq \nabla_\star(\mathsf{S})$.

(ii) Termination: For every ascending chain $\{\mathsf{x}_i\}_{i\geq 0}$*, the chain defined as*

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_i = \nabla_\star(\{\mathsf{x}_j \,|\, 0 \leq j \leq i\})$$

is ascending too, and it stabilizes after a finite number of terms.

The definition above has been used in [59], for fix-point computations over sets represented as automata, in a model checking approach.

Example 1. *Consider the lattice of intervals* $\mathsf{L} = \{\bot\} \cup \{[\ell, u] \mid \ell \in \mathbb{Z} \cup \{-\infty\},\ u \in \mathbb{Z} \cup \{+\infty\},\ \ell \leq u\}$*, ordered by:* $\forall \mathsf{x} \in \mathsf{L}, \bot \leq \mathsf{x}$ *and* $[\ell_0, u_0] \leq [\ell_1, u_1]$ *if* $\ell_1 \leq \ell_0$ *and* $u_0 \leq u_1$*. Let* k *be a fixed positive integer constant, and* I *be any set of indices. Consider the threshold widening operator defined on* L *by:*

$$\begin{gathered}\nabla^{\mathsf{k}}_\star(\{\bot\}) = \bot \\ \nabla^{\mathsf{k}}_\star(\{\bot\} \cup\ \mathsf{S}) = \nabla^{\mathsf{k}}_\star(\mathsf{S}) \\ \nabla^{\mathsf{k}}_\star(\{[\ell_i, u_i] : i \in I\}) = [h_1, h_2]\end{gathered}$$

where

$$\begin{gathered}h_1 = min\{\ell_i : i \in I\}\ \textit{if}\ \ min\{\ell_i : i \in I\} > -\mathsf{k},\ \ \textit{else}\ -\infty \\ h_2 = max\{u_i : i \in I\}\ \textit{if}\ \ max\{u_i : i \in I\}) < \mathsf{k},\ \ \textit{else}\ +\infty.\end{gathered}$$

Observe that for all k*,* $\nabla^{\mathsf{k}}_\star$ *is associative, and monotone. Observe that* ∇^{k}_* *may widen also the singletons. For instance, we get* $\nabla^{7}_\star(\{[-8, 4]\}) = [-\infty, 4]$*.*

Definition 3.6 (pair-widening [51], [116])**.** *Let* $(\mathsf{P}, \leq)$ *be a poset. A pair-widening operator is a binary operator* $\nabla : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *such that*

(i) Covering: $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x} \leq \mathsf{x}\nabla\mathsf{y}$*, and* $\mathsf{y} \leq \mathsf{x}\nabla\mathsf{y}$*.*

(ii) Termination: For every ascending chain $\{\mathsf{x}_i\}_{i\geq 0}$*, the ascending chain defined as*

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i\nabla\mathsf{x}_{i+1}$$

stabilizes after a finite number of terms.

Definition 3.7 (extrapolator)**.** *Let* $(\mathsf{P}, \leq)$ *be a poset. A binary operator* $\bullet : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *is called extrapolator if it satisfies the covering property, i.e.* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x} \leq \mathsf{x} \bullet \mathsf{y}$*, and* $\mathsf{y} \leq \mathsf{x} \bullet \mathsf{y}$*.*

Observe that pair-widening operators are not necessarily neither commutative neither monotone, nor associative, while these properties are crucial for chaotic iteration fixpoint algorithms [116].

Example 2. *Consider the binary operator introduced in [47] on the same lattice of Intervals of Example 1:*

$$\begin{aligned}\bot\nabla\mathsf{x} &= \mathsf{x} \\ x\nabla\bot &= \mathsf{x} \\ [\ell_0, u_0]\nabla[\ell_1, u_1] &= [\textit{if}\ \ell_1 < \ell_0\ \textit{then}\ -\infty\ \textit{else}\ \ell_0, \\ &\quad\ \ \textit{if}\ u_0 < u_1\ \textit{then}\ +\infty\ \textit{else}\ u_0].\end{aligned}$$

∇ is a pair-widening operator, as it satisfies both covering and termination requirements of Def.3.6.
Observe that the operator is not commutative, as for instance

$$\begin{aligned}[2,3]\nabla[1,4] &= [-\infty,+\infty]\\ [1,4]\nabla[2,3] &= [1,4]\end{aligned}$$

Moreover, in order to see that it is not monotone, consider $[0,1] \leq [0,3]$. *We have:*

$$\begin{aligned}[0,1]\nabla[0,2] &= [0+\infty]\\ [0,3]\nabla[0,2] &= [0,3].\end{aligned}$$

and of course $[0,+\infty]$ *is not smaller or equal to* $[0,3]$. *Finally, observe that associativity does not hold either:*

$$\begin{aligned}[0,2]\nabla([0,1]\nabla[0,2]) &= [0+\infty]\\ ([0,2]\nabla[0,1])\nabla[0,2] &= [0,2].\end{aligned}$$

Let us come back to the two definitions of widening operators introduced before. We see how to build a set-widening out of a pair-widening operator.

Theorem 3.6. *Let* $(\mathsf{P},\leq)$ *be a poset, and let* $\nabla : \mathsf{P}\times\mathsf{P}\to\mathsf{P}$ *be a pair-widening operator on* P. *Define* $\nabla_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *such that:*

- *dom*$(\nabla_\star) = \mathsf{R}_1 \cup \mathsf{R}_2$, *where*
  $\mathsf{R}_1 = \{\{x,y\} \mid x,y \in \mathsf{P}\}$, *and*
  $\mathsf{R}_2 = \{\mathsf{S} \subseteq \mathsf{P} \mid \mathsf{S}$ *is a finite ascending chain*$\}$.

- $\forall\{x,y\} \in \mathsf{R}_1$,
  $$\nabla_\star(\{\mathsf{x},\mathsf{y}\}) =_{def} \begin{cases} \mathsf{x}\nabla\mathsf{y} & \textit{if}\ \mathsf{x}\leq\mathsf{y}\\ \mathsf{z}\in\{\mathsf{x}\nabla\mathsf{y},\mathsf{y}\nabla\mathsf{x}\} & \textit{randomly, otherwise.}\end{cases}$$

- $\forall\mathsf{S} = \{\mathsf{x}_i \mid \mathsf{x}_0 \leq \mathsf{x}_1 \leq \cdots \leq \mathsf{x}_j\} \in \mathsf{R}_2$,
  $\nabla_\star(\mathsf{S}) =_{def} (((\mathsf{x}_0\nabla\mathsf{x}_1)\nabla\mathsf{x}_2\ldots)\nabla\mathsf{x}_j)$.

Then $\nabla_\star$ *is a set-widening operator.*

The notion of set-widening is weaker than the notion of pair-widening. This is why, in general, there is no way to prove the dual of Theorem 3.6, which can be stated only under restricted conditions.

Theorem 3.7. *Let* $(\mathsf{P},\leq)$ *be a poset, and let* $\nabla_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *be a set-widening operator on* P *such that*

- *dom*$(\nabla_\star) \supseteq \{\{\mathsf{x},\mathsf{y}\} \mid \mathsf{x},\mathsf{y}\in\mathsf{P}\}$, *and*

- $\forall\mathsf{S}\subseteq\mathsf{P},\ \forall\mathsf{x}\in\mathsf{P}$, *if* $\mathsf{S}\cup\{\mathsf{x}\} \subseteq$ *dom*$(\nabla_\star)$ *then also* $\mathsf{S} \subseteq$ *dom*$(\nabla_\star)$

- $\forall \mathsf{S} \subseteq \mathsf{P},\ \forall \mathsf{x} \in \mathsf{P},\ \nabla_\star(\mathsf{S} \cup \{\mathsf{x}\}) = \nabla_\star(\{\nabla_\star(\mathsf{S}), \mathsf{x}\})$.

Then, the binary operator $\nabla : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *defined by* $\mathsf{x}\nabla\mathsf{y} = \nabla_\star(\{\mathsf{x}, \mathsf{y}\})$ *is a pair-widening operator.*

Observe that the set-widening operator $\nabla^{\mathsf{k}}_\star$ of Example 1 satisfies the conditions of Theorem 3.7 above, yielding to a corresponding pair-widening operator.

3.4.3 Narrowing Operator

Similarly, two different general definitions of narrowing operator have been introduced. The first one defines a narrowing operator as a partial function on the powerset of a poset P, while the second one defines it as a binary (total) function on a poset P.

Definition 3.8 (set-narrowing [53, 58]). *Let* $(\mathsf{P}, \leq)$ *be a poset. A set-narrowing operator is a partial function* $\Delta_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *such that*

(i) Bounding: Let S *be an element of* $\wp(\mathsf{P})$*. If* $\Delta_{\star(\mathsf{S})}$ *is defined, then* $glb(\mathsf{S})$ *exists and there exists* $\mathsf{s} \in \mathsf{S}$ *such that* $glb(\mathsf{S}) \leq \Delta_\star(\mathsf{S}) \leq \mathsf{s}$.

(ii) Termination: For every decreasing chain $\mathsf{x}_0 \geq \mathsf{x}_1 \geq \ldots$*, the chain defined as*

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_i = \Delta_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\})$$

is descending too, and it stabilizes after a finite number of terms.

Example 3. *Let* L *be the lattice of intervals introduced in Example 1. We can define* $\Delta_\star$*, a narrowing operator, on* L *as follows.*

$$\begin{gathered}\Delta_\star(\{\bot\}) = \bot \\ \Delta_\star(\{\bot\} \cup \mathsf{S}) = \Delta_\star(\mathsf{S}) \\ \Delta_\star(\{[\ell_i, u_i] : i \in I\}) = [h_1, h_2]\end{gathered}$$

where

$$\begin{gathered}h_1 = max\{\ell_i : i \in I\} \\ h_2 = min\{u_i : i \in I\}\end{gathered}$$

It is easy to verify that it satisfy the termination condition, as it converges immediately on decreasing chains.
Observe that $\Delta_\star$ *is associative, and monotone. Observe that* $\Delta_\star$ *may narrow also the singletons. For instance, we get* $\Delta_\star(\{[-8, 4]\}) = [-8, 4]$ *and* $\Delta_\star(\{[-8, 6], [1, 5], [-9, 11]\}) = [1, 5]$.

Example 4. *Observe that both conditions (bounding and termination) are required in order to get a narrowing operator. For instance, on the lattice of intervals on* $\mathbb{R}$ *instead of* $\mathbb{Z}$*, the operator* Δ_* *defined in Example 3 fulfills the bounding condition but it does not satisfy the termination one. Therefore, it is a not narrowing operator.*

Definition 3.9 (pair-narrowing [50, 51]). *Let* $(\mathsf{P}, \leq)$ *be a poset. A pair-narrowng operator is a binary operator* $\Delta : \mathsf{P} \times \mathsf{P} \rightarrow \mathsf{P}$ *such that*

(i) Bounding: $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : (\mathsf{x} \leq \mathsf{y}) \Longrightarrow (\mathsf{x} \leq (\mathsf{y} \Delta \mathsf{x}) \leq \mathsf{y})$.

(ii) Termination: For every decreasing chain $\mathsf{x}_0 \geq \mathsf{x}_1 \geq \ldots$, *the decreasing chain defined as*

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i \Delta \mathsf{x}_{i+1}$$

stabilizes after a finite number of terms.

Observe that pair-narrowing operators are not necessarily neither commutative neither monotone, nor associative. Moreover observe also that if P is a meet-semi-lattice (the greatest lower bound $\mathsf{x} \sqcap \mathsf{y}$ exists for all $\mathsf{x}, \mathsf{y} \in \mathsf{P}$) satisfying the decreasing chain condition (no strictly decreasing chain in P can be infinite), then $\sqcap$ is a narrowing.

Example 5. *Consider the binary operator introduced in [51] on the same lattice of intervals on* $\mathbb{Z}$ *of Example 1:*

$$\begin{array}{rcl} \bot \Delta \mathsf{x} & = & \bot \\ \mathsf{x} \Delta \bot & = & \bot \\ [\ell_0, u_0] \Delta [\ell_1, u_1] & = & [\mathit{if}\ \ell_0 = -\infty\ \mathit{then}\ \ell_1\ \mathit{else}\ \ell_0, \\ & & \mathit{if}\ u_0 = +\infty\ \mathit{then}\ u_1\ \mathit{else}\ u_0] \end{array}$$

Δ *is a pair-widening operator, as it satisfies both bounding and termination requirements of Definition 3.9.*

$$\begin{array}{rcl} [-\infty, +\infty] \Delta [-\infty, 101] & = & [-\infty, 101] \\ [1, +\infty] \Delta [50, 100] & = & [1, 100] \\ [1, 4] \Delta [2, 3] & = & [1, 3] \end{array}$$

Let us come back to the two definitions of narrowing operators introduced above. Like in the case of widening, we study how we can build a set-narrowing operator out of a pair-narrowing operator, and viceversa.

Theorem 3.8. *Let* $(\mathsf{P}, \leq)$ *be a poset, and let* $\Delta : \mathsf{P} \times \mathsf{P} \rightarrow \mathsf{P}$ *be a pair-narrowing operator on* P. *Define* $\Delta_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *such that:*

- *dom*$(\Delta_\star) = \mathsf{R}_1 \cup \mathsf{R}_2$, *where*
  $\mathsf{R}_1 = \{\{\mathsf{x}, \mathsf{y}\} \mid \mathsf{x}, \mathsf{y} \in \mathsf{P}\ : \exists\ \mathit{glb}(\mathsf{x}, \mathsf{y})\}$, *and*
  $\mathsf{R}_2 = \{\mathsf{S} \subseteq \mathsf{P} \mid \mathsf{S}\ \mathit{is\ a\ finite\ descending\ chain}\}$.

- $\forall \{\mathsf{x}, \mathsf{y}\} \in \mathsf{R}_1$,
  $$\Delta_\star(\{\mathsf{x}, \mathsf{y}\}) =_{def} \begin{cases} \mathsf{y} \Delta \mathsf{x} & \mathit{if}\ \mathsf{x} \leq \mathsf{y} \\ \mathit{glb}(\{\mathsf{x}, \mathsf{y}\}) & \mathit{otherwise.} \end{cases}$$

- $\forall \mathsf{S} = \{\mathsf{x}_i \mid \mathsf{x}_0 \geq \mathsf{x}_1 \geq \cdots \geq \mathsf{x}_j\} \in \mathsf{R}_2$,
  $\Delta_\star(\mathsf{S}) =_{def} ((((\mathsf{x}_0 \Delta \mathsf{x}_1) \Delta \mathsf{x}_2) \Delta) \ldots \Delta \mathsf{x}_j)$.

Then $\Delta_\star$ is a set-narrowing operator.

Theorem 3.9. *Let $(\mathsf{P}, \leq)$ be a poset, and let $\Delta_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ be a set-narrowing operator on P such that*

1. *$dom(\Delta_\star) \supseteq \{\{\mathsf{x}, \mathsf{y}\} \mid \mathsf{x}, \mathsf{y} \in \mathsf{P}\}$, and*
2. *$\forall \mathsf{S} \subseteq \mathsf{P},\ \forall \mathsf{x} \in \mathsf{P},$ if $\mathsf{S} \cup \{\mathsf{x}\} \subseteq dom(\Delta_\star)$ then also $\mathsf{S} \subseteq dom(\Delta_\star)$*
3. *$\forall \mathsf{S} \subseteq \mathsf{P},\ \forall \mathsf{x} \in \mathsf{P},\ \Delta_\star(\mathsf{S} \cup \{\mathsf{x}\}) = \Delta_\star(\{\Delta_\star(\mathsf{S}), \mathsf{x}\})$.*

Then, the binary operator $\Delta : \mathsf{P} \times \mathsf{P} \rightarrow \mathsf{P}$ defined $\mathsf{x}\Delta\mathsf{y} = \Delta_\star(\{\mathsf{x}, \mathsf{y}\})$ is a pair-narrowing operator.

3.4.4 Combination of Widening and Narrowing Operators

Widening and Narrowing

In order to better understand how widening and a narrowing operators can be combined in an effective way, consider the following example on the finite powerset domain of intervals.

The design of a successful widening is a very delicate task that is not only dependent on the considered abstract domain but also on the particular analysis or verification application. An important contribution in such context is [13], which introduces three methodologies for the design of widening operators. All of these methodologies are based on the same extrapolator, while they differ on the termination property: the first one poses constraints on the cardinality of the arguments, the second one uses connectors (as, for example, Egli-Milner Connectors), and the last one is certificate-based.
Notice that these generic widening constructions are applicable to any finite powerset abstract domain, encoding either numerical or symbolic information.

Example 6. *Let L be the lattice of intervals introduced in Example 1, and let $\wp_\mathsf{f}(\mathsf{L})$ be its finite powerset. For $\mathsf{A}, \mathsf{B} \in \wp_\mathsf{f}(\mathsf{L})$, we say $\mathsf{A} \trianglelefteq \mathsf{B}$ if and only if $\forall \mathsf{x} \in \mathsf{A}, \exists \mathsf{y} \in \mathsf{B}$ such that $\mathsf{x} \leq \mathsf{y}$. Consider the function reduce $: \wp_\mathsf{f}(\mathsf{L}) \rightarrow \wp_\mathsf{f}(\mathsf{L})$ defined as $\forall \mathsf{A} \subseteq \mathsf{L}$, reduce$(\mathsf{A})$ is the maximal subset of A such that $\forall \mathsf{x}, \mathsf{y} \in \mathsf{A} : \mathsf{x} < \mathsf{y} \Rightarrow \mathsf{x} \notin reduce(\mathsf{A})$. Observe that $reduce(\mathsf{A}) \trianglelefteq \mathsf{A}$ and $\mathsf{A} \trianglelefteq reduce(\mathsf{A})$.*
The closure of $\mathsf{A} \subseteq \mathsf{L}$, denoted by $\overline{\mathsf{A}}$, is the superset of A such that $\forall \mathsf{x}, \mathsf{y} \in \mathsf{A}$, such that $\mathsf{x} \sqcap \mathsf{y} \neq \emptyset$, the least upper bound $\mathsf{x} \sqcup \mathsf{y} \in \overline{\mathsf{A}}$.
For $\mathsf{X} \subseteq \mathsf{L}$, we denote by $min(\mathsf{X})$ the minimal value, and by $max(\mathsf{X})$ the maximal value, e.g. $min(\{[3,8],[2,5],[1,4]\}) = 1$ and $max(\{[3,8],[2,5],[1,4]\}) = 8$.
By $minInt(\mathsf{X})$ we denote the interval which have $min(\mathsf{X})$ as bottom value and, analogously, by $maxInt(\mathsf{X})$ the interval having $max(\mathsf{X})$ as top value, e.g. $minInt(\{[3,8],[2,5],[1,4]\}) = [1,4]$ and $maxInt(\{[3,8],[2,5],[1,4]\}) = [3,8]$.
For any positive constant k, we can define the pair-widening $\nabla^\mathsf{k} : \wp_\mathsf{f}(\mathsf{L}) \times \wp_\mathsf{f}(\mathsf{L}) \rightarrow \wp_\mathsf{f}(\mathsf{L})$ as follows.
Let A, B be elements of $\wp_\mathsf{f}(\mathsf{L})$.

- *If the cardinality of reduce*$(\mathsf{A} \cup \mathsf{B})$ *is smaller or equal to* k*, then* $\mathsf{A}\nabla^{\mathsf{k}}\mathsf{B} =$ *reduce*$(\mathsf{A} \cup \mathsf{B})$*.*
- *Otherwise, let* $\mathsf{R} =$ *reduce*$(\overline{\mathsf{W}})$ *where* W *is obtained by:*
  - $\mathsf{W} = \mathsf{A} \cup \mathsf{B}$
  - *If the cardinality of reduce*(W) *is greater than* k *and* $\exists \mathsf{s}' \in \mathsf{B}$ *such that* $max(\mathsf{s}') > max(\mathsf{A})$ *then:*
    let $\mathsf{r} \in \mathsf{A}$ *be the interval such that* $max(\mathsf{r}) = max(\mathsf{A})$ *then* $\mathsf{W} = \mathsf{W} \cup [min(\mathsf{r}), +\infty]$*.*
  - *And if the cardinality of reduce*(W) *is greater than* k *and* $\exists \mathsf{s}' \in \mathsf{B}$ *such that* $min(\mathsf{s}') < min(\mathsf{A})$ *then:*
    let $\mathsf{r} \in \mathsf{A}$ *be the interval such that* $min(\mathsf{r}) = min(\mathsf{A})$ *then* $\mathsf{W} = \mathsf{W} \cup [-\infty, max(\mathsf{r})]$*.*
- *While the cardinality of* R *is greater than* k*:*
  let $\mathsf{s}, \mathsf{s}' \in \mathsf{R}$ *such that* $|min(\mathsf{s}') - max(\mathsf{s})|$ *is minimal in* R*, then* $\mathsf{R} = (\mathsf{R} \backslash \{\mathsf{s}, \mathsf{s}'\}) \cup \{(\mathsf{s} \sqcup \mathsf{s}')\}$*.*
- $\mathsf{A}\nabla^{\mathsf{k}}\mathsf{B} = \mathsf{R}$*.*

Observe that if $\mathsf{Y} = \mathsf{A}\nabla^{\mathsf{k}}\mathsf{B}$*, then the cardinality of* Y *is always smaller or equal to* k*. For instance, if*

$$\mathsf{A} = \{[-5, 2], [1, 6], [11, 23], [27, 33], [30, 35], [36, 40]\}$$

and

$$\mathsf{B} = \{[-2, 3], [9, 15], [32, 35], [37, 42]\}$$

then

$$\mathsf{A}\nabla^{3}\mathsf{B} = \{[-5, 6], [9, 23], [27, +\infty]\}.$$

In fact , as the cardinality of reduce$(\mathsf{A}\cup\mathsf{B})$ *is* 4*, which is greater than* $\mathsf{k} = 3$*, and there exists an interval* s *in* B *(namely* $[37, 42]$*) such that* $min(\mathsf{s}) > max(\mathsf{A})$*, the set* $\mathsf{W} = \mathsf{A} \cup \mathsf{B} \cup \{[36, +\infty]\}$ *is computed. Then, its closure* $\overline{\mathsf{W}} = \mathsf{W} \cup \{[-5, 6], [-5, 3], [-2, 3], [9, 23], [27, 35], [36, 42]\}$ *is computed. Finally, the reduce operator is applied, yielding to reduce*$(\overline{\mathsf{W}}) = \{[-5, 6], [9, 23], [27, +\infty]\}$*.*

Notice that this widening operator satisfies the constraints in [13], as it merges an extrapolation heuristics "∇*-covered" and a "k-collapsor", as requested to obtain a "cardinality-based" widening. In addition, our operator can be applied also on not comparable elements, whereas the generic construction provided by Bagnara et.al. requires that the first element is less than second one.*

Similarly, we can define a corresponding pair-narrowing operator $\Delta^{\mathsf{k}} : \wp_{\mathsf{f}}(\mathsf{L}) \rightarrow \wp(\mathsf{L})$*. Let* $\mathsf{A}, \mathsf{B} \in \wp_{\mathsf{f}}(\mathsf{L})$ *such that* $\mathsf{A} \trianglelefteq \mathsf{B}$*,* $\mathsf{B}\Delta^{\mathsf{k}}\mathsf{A}$ *is defined as follows.*

- *Let* $\mathsf{A}' = \mathit{reduce}(\mathsf{A}) = \{\mathsf{s}_1, \ldots, \mathsf{s}_n\}$, $\mathsf{B}' = \mathit{reduce}(\mathsf{B}) = \{\mathsf{q}_1, \ldots, \mathsf{q}_m\}$ *and* $\mathsf{R} = \mathsf{A}'$.
- *If* $\mathit{min}(\mathsf{A}') = -\infty$ *then* $\mathsf{R} = \mathsf{R} \setminus \{\mathit{minInt}(\mathsf{A}')\} \cup \{\mathit{minInt}(\mathsf{B}')\}$, *and if* $\mathit{max}(\mathsf{A}') = +\infty$ *then* $\mathsf{R} = \mathsf{R} \setminus \{\mathit{maxInt}(\mathsf{A}')\} \cup \{\mathit{maxInt}(\mathsf{B}')\}$.
- *For each* $\mathsf{q}_i \in \mathsf{B}'$.
  - *Let* $\mathsf{H}_i = \{\mathsf{s}_j \in \mathsf{R} \mid \mathsf{s}_j \leq \mathsf{q}_i\}$
  - *If cardinality of* H_i *is greater than* 1.
    - *Let* $\mathsf{a}, \mathsf{b} \in \mathsf{H}_i$ *such that* $|\mathit{min}(\mathsf{b}) - \mathit{max}(\mathsf{a})|$ *is minimal in* H_i.
    - $\mathsf{R} = \mathit{reduce}(\mathsf{R} \setminus \{\mathsf{a}, \mathsf{b}\} \cup \{\mathit{lub}(\mathsf{a}, \mathsf{b})\})$.
  - *If the cardinality of* R *is smaller than* k *then break.*
- $\mathsf{B}\Delta^{\mathsf{k}}\mathsf{A} = \mathsf{R}$.

Observe that Δ^{k} *is a pair-narrowing operator, as it satisfies both bounding and termination properties.*
For instance, if

$$\mathsf{A} = \{[-10, -6], [-5, -2], [-1, 0], [1, 3], [9, 13], [18, 20], [23, 27], [29, +\infty]\}$$

and

$$\mathsf{B} = \{[-10, 3], [7, 13], [16, +\infty]\}$$

then we have

$$\mathsf{B}\Delta^3\mathsf{A} = \{[-10, 3], [9, 13], [16, +\infty]\}$$

As an example on how the widening and narrowing operators just introduced can be combined in order to accelerate the fix-point computation without loosing too much accuracy, consider the function $F : \wp_{\mathsf{f}}(\mathsf{L}) \rightarrow \wp_{\mathsf{f}}(\mathsf{L})$ *defined by*

$$F \equiv \lambda\mathsf{X}.((((\mathsf{X} \sqcup \{[1, 2]\}) \cup \{(\mathit{maxInt}(\mathsf{X}) \oplus [2, 2])\}) \setminus \{\perp\}) \sqcap \{[100, 100]\})$$

where $\oplus : \mathsf{L} \times \mathsf{L} \rightarrow \mathsf{L}$ *such that* $\perp \oplus \mathsf{X} = \mathsf{X} \oplus \perp = \perp$ *and* $[\ell_0, u_0] \oplus [\ell_1, u_1] = [\ell_0 + \ell_1, u_0 + u_1]$. *The computation of the fix-point of* F *starting from the* $\perp$ *element, would require at least 50 steps, getting to the fixpoint element* $\{[1, 2], [3, 4], \ldots, [100, 100]\}$. *In order to accelerate the fix-point computation of* F, *first we use the widening operator* ∇^3 *defined above.*

$$\begin{array}{lcl}
\mathsf{X}_0 & = & \{\perp\} \\
\mathsf{X}_1 & = & \mathsf{X}_0\nabla^3((((\mathsf{X}_0 \sqcup \{[1, 2]\}) \cup \{(\mathit{maxInt}(\mathsf{X}_0) \oplus [2, 2])\}) \setminus \{\perp\}) \sqcap \{[100, 100]\}) \\
 & = & \{[1, 2]\} \\
\mathsf{X}_2 & = & \{[1, 2], [3, 5]\} \\
 & \vdots & \\
\mathsf{X}_4 & = & \{[1, 2], [3, 4], [5, +\infty]\} = \mathsf{X}_5
\end{array}$$

The fix-point is obtained in 5 steps, but the accuracy of the result is not satisfactory as it completely looses the rightmost bound. Nevertheless, this lack of precision can be recovered by applying the narrowing operator Δ^3.

$$\begin{array}{rcl}
\mathsf{Y}_0 & = & \mathsf{X}_4 \\
\mathsf{Y}_1 & = & \mathsf{Y}_0\Delta^3((((\mathsf{Y}_0 \sqcup \{[1,2]\}) \cup \{(maxInt(\mathsf{Y}_0) \oplus [2,2])\}) \setminus \{\bot\}) \sqcap \{[100,100]\}) \\
 & = & \{[1,2],[3,4],[5,100]\} \\
\mathsf{Y}_2 & = & \{[1,2],[3,4],[5,100]\} = \mathsf{Y}_1
\end{array}$$

Observe that Δ^3 *requires only 3 steps to converge. We obtain as a final result the set* $\{[1,2],[3,4],[5,100]\}$, *which is not as precise as the least fix-point computation mentioned above, but that has the advantage of being reached dramatically quicker, and of preserving accuracy about the rightmost bound of the possible values of* F.

Widening, Narrowing and Cartesian Product

The next theorems show how pair-widening and pair-narrowing operators can be combined when considering the cartesian product of posets.

Theorem 3.10. *Let* ∇_A *and* ∇_D *be pair-widening operators defined on the posets* A *and* D*, respectively.*
The binary operator $\nabla : (\mathsf{A} \times \mathsf{D}) \times (\mathsf{A} \times \mathsf{D}) \to (\mathsf{A} \times \mathsf{D})$ *defined by* $\forall\langle \mathsf{a},\mathsf{d}\rangle, \langle \mathsf{a}',\mathsf{d}'\rangle \in \mathsf{A} \times \mathsf{D} : \langle \mathsf{a},\mathsf{d}\rangle \nabla \langle \mathsf{a}',\mathsf{d}'\rangle = \langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}'\rangle$ *is a pair-widening operator.*

Theorem 3.11. *Let* Δ_{A} *and* Δ_{D} *be pair-narrowing operators defined on the posets* A *and* D*, respectively.*
The binary operator $\Delta : (\mathsf{A} \times \mathsf{D}) \times (\mathsf{A} \times \mathsf{D}) \to (\mathsf{A} \times \mathsf{D})$ *defined by* $\forall\langle \mathsf{a},\mathsf{d}\rangle, \langle \mathsf{a}',\mathsf{d}'\rangle \in \mathsf{A} \times \mathsf{D} : \langle \mathsf{a},\mathsf{d}\rangle \Delta \langle \mathsf{a}',\mathsf{d}'\rangle = \langle \mathsf{a}\Delta_{\mathsf{A}}\mathsf{a}', \mathsf{d}\Delta_{\mathsf{D}}\mathsf{d}'\rangle$ *is a pair-narrowing operator.*

A corresponding result can be obtained also for set-widening and set-narrowing operators.

Widening and Narrowing Operators on the same poset

What happens when more than one widening (or narrowing) operators are defined on a poset P? Is it possible to get a more precise and/or a more efficient operator by combining them in a suitable way? Unfortunately, in general the answer is negative. And the reason relies on the fact that the possibly non monotonic behavior of the widening (or narrowing) operators becomes an issue when trying to prove termination of their combination on an ascending (and descending for narrowing) chain. However, as soon as stronger termination conditions are guaranteed on the poset P, some positive results can be easily derived.

Theorem 3.12. *Let* $(\mathsf{P}, \leq)$ *be a lattice satisfying the ascending chain property. Let* ∇_1, ∇_2 *be two pair-widening operators on* P. *Then, the binary operators* $\nabla_\sqcap, \nabla_\sqcup$ *defined by*

$$\begin{aligned} \mathsf{x} \nabla_\sqcap \mathsf{y} &= (\mathsf{x} \nabla_1 \mathsf{y}) \sqcap (\mathsf{x} \nabla_2 \mathsf{y}) \\ \mathsf{x} \nabla_\sqcup \mathsf{y} &= (\mathsf{x} \nabla_1 \mathsf{y}) \sqcup (\mathsf{x} \nabla_2 \mathsf{y}) \end{aligned}$$

are pair-widening operators.

This result may apply for instance to widening operators defined on the (infinite) domain of congruences [73], where prime factorization is an issue, in order to tune performance vs. accuracy of the analysis. In fact, $\nabla_\sqcup$ may gain in efficiency with respect to both ∇_1 and ∇_2, while $\nabla_\sqcap$ may better keep accuracy, thus returning a more accurate result.
A corresponding result can be obtained with narrowing operators.

Theorem 3.13. *Let* $(\mathsf{P}, \leq)$ *be a lattice satisfying the descending chain property. Let* Δ_1, Δ_2 *be two pair-narrowing operators on* P. *Then, the binary operators* $\Delta_\sqcap, \Delta_\sqcup$ *defined by*

$$\begin{aligned} \mathsf{x} \Delta_\sqcap \mathsf{y} &= (\mathsf{x} \Delta_1 \mathsf{y}) \sqcap (\mathsf{x} \Delta_2 \mathsf{y}) \\ \mathsf{x} \Delta_\sqcup \mathsf{y} &= (\mathsf{x} \Delta_1 \mathsf{y}) \sqcup (\mathsf{x} \Delta_2 \mathsf{y}) \end{aligned}$$

are pair-narrowing operators.

Similar results can be easily derived by similar proofs also for set-widening and set-narrowing operators.

Theorem 3.14. *Let* $(\mathsf{P}, \leq)$ *be a lattice satisfying the ascending chain property. Let* ∇_{*1}, ∇_{*2} *be two set-widening operators on* P. *Then, the operators* $\nabla_{*\sqcap}, \nabla_{*\sqcup}$ *defined by*

$$\begin{aligned} \nabla_{*\sqcap}(\{\mathsf{S}\}) &= (\nabla_{*1}(\{\mathsf{S}\})) \sqcap (\nabla_{*2}(\{\mathsf{S}\})) \\ \nabla_{*\sqcup}(\{\mathsf{S}\}) &= (\nabla_{*1}(\{\mathsf{S}\})) \sqcup (\nabla_{*2}(\{\mathsf{S}\})) \end{aligned}$$

are set-widening operators.

Theorem 3.15. *Let* $(\mathsf{P}, \leq)$ *be a lattice satisfying the descending chain property. Let* Δ_{*1}, Δ_{*2} *be two set-widening operators on* P. *Then, the operators* $\Delta_{*\sqcap}, \Delta_{*\sqcup}$ *defined by*

$$\begin{aligned} \Delta_{*\sqcap}(\{\mathsf{S}\}) &= (\Delta_{*1}(\{\mathsf{S}\})) \sqcap (\Delta_{*2}(\{\mathsf{S}\})) \\ \Delta_{*\sqcup}(\{\mathsf{S}\}) &= (\Delta_{*1}(\{\mathsf{S}\})) \sqcup (\Delta_{*2}(\{\mathsf{S}\})) \end{aligned}$$

are set-narrowing operators.

Strong Widening and Narrowing Operators

For numerical domains like polyhedra, where the abstract elements computed at each iteration of the analysis are not necessarily ordered, stronger notions of widening and narrowing are used for forcing the termination of the analysis. This is the case, for instance, of the trace partitioning abstract domain of Astrée, an abstract interpretation-based analyzer aiming at proving automatically the absence of run time errors in programs written in the C programming language, which has been applied with success to large safety critical real-time software for avionics [24, 45].

Definition 3.10 (strong pair-widening [122]). *Let* $(\mathsf{P}, \leq)$ *be a poset. A strong pair-widening operator is a binary operator* $\nabla : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *such that*

(i) Covering: $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x} \leq \mathsf{x} \nabla \mathsf{y}$, *and* $\mathsf{y} \leq \mathsf{x} \nabla \mathsf{y}$.

(ii) Termination: For every sequence $\{\mathsf{x}_i\}_{i \geq 0}$, *the ascending chain defined as* $\mathsf{y}_0 = \mathsf{x}_0$, $\mathsf{y}_{i+1} = \mathsf{y}_i \nabla \mathsf{x}_{i+1}$ *stabilizes after a finite number of terms.*

Observe that this definition is strictly stronger than Definition 3.6, as termination is required starting from every (not necessarily increasing) sequence.

Example 7. *The octagon domain [106, 108] is based on invariants of the form* $\pm \mathsf{x} \pm \mathsf{y} \leq \mathsf{c}$, *where* x *and* y *are numerical variables and* c *is a numeric constant. Sets described by such invariants are special kind of polyhedra called octagons because they feature at most eight edges in dimension 2. These constraints are expressed through Difference Bound Matrices, which are adjacency matrices of weighted graphs. The widening operator defined on this domain consists on removing unstable constraints. In this case, termination has to be guaranteed for the chain of widened elements starting from a sequence of elements possibly incomparable. This is why the strongest notion of pair widening has to be used.*
Notice however that as an alternative to the strong pair-widening, Bagnara et.al. [11] introduced a different representation of the octagons to ensure that the standard pair-widening can be applied. This approach is applied in [11] to several weakly-relational domains, but it can be generalized to other domains.

The two notions of Pair-widening and Strong pair-widening are equivalent for a lattice P, under associativity conditions, as shown in Theorem 3.16. In order to prove it, we introduce the following auxiliary Lemma.

Lemma 3.1. *Let* ∇ *be a pair-widening operator on a lattice* $(\mathsf{P}, \leq)$, *such that for every finite set* $\{\mathsf{x}_i\}_{0 \leq i \leq n}$ *and for every* $\mathsf{y} \in \mathsf{P}$, $(((\mathsf{x}_0 \nabla \mathsf{x}_1) \nabla \dots) \nabla \mathsf{x}_n) \nabla (\mathsf{x}_0 \sqcup \mathsf{x}_1 \sqcup \dots \sqcup \mathsf{x}_n \sqcup \mathsf{y}) = (((\mathsf{x}_0 \nabla \mathsf{x}_1) \nabla \dots) \nabla \mathsf{x}_n) \nabla \mathsf{y}$, *then* ∇ *is a strong pair-widening operator.*

Theorem 3.16. *Let* ∇ *be an associative pair-widening operator on a lattice* $(\mathsf{P}, \leq)$, *such that for* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x} \nabla \mathsf{y} = \mathsf{x} \nabla (\mathsf{x} \sqcup \mathsf{y})$, *then* ∇ *is a strong pair-widening operator.*

Example 8. *The pair-widening operator on intervals obtained from the set-widening of Example 1 following the construction of Theorem 3.7, satisfies the condition of Theorem 3.16, and it is in fact a strong pair widening operator.*
However, not every pair-widening operator is also a strong one. On the same lattice of intervals, consider for instance the pair-widening ∇ *defined by:*

$$\bot \nabla \mathsf{x} = \mathsf{x} \qquad and \qquad \mathsf{x} \nabla \bot = \mathsf{x}$$

$$[\ell_0, u_0]\nabla[\ell_1, u_1] = \quad = \begin{cases} [-\infty, +\infty] \\ \qquad if\ [\ell_0, u_0] \leq [\ell_1, u_1]\ or\ [\ell_1, u_1] \leq [\ell_0, u_0] \\ \\ [min(\ell_0, \ell_1), max(u_0, u_1)] \\ \qquad otherwise \end{cases}$$

On increasing sequences, the widened sequence terminates immediately, whereas if we consider for instance the sequence $\{[i, i+1]\}_{i \geq 0}$, ∇ yields to the ascending sequence $\{[0, i]\}_{i \geq 1}$, which does not terminate.

Definition 3.11 (strong pair-narrowing)**.** *Let $(\mathsf{P}, \leq)$ be a poset. A strong pair-narrowing operator is a binary operator $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ such that*

(i) Bounding: $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : (\mathsf{x} \leq \mathsf{y}) \implies (\mathsf{x} \leq (\mathsf{y}\Delta\mathsf{x}) \leq \mathsf{y})$.

(ii) Termination: For every sequence $\{\mathsf{x}_i\}_{i \geq 0}$, the decreasing chain defined as

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i \Delta \mathsf{x}_{i+1}$$

stabilizes after a finite number of terms.

Example 9. *The following strong narrowing operator has been introduced in [47].*

$$\begin{array}{rcl} \mathsf{x}\Delta\bot & = & \bot \\ [\ell_0, u_0]\Delta[\ell_1, u_1] & = & [if\ \ell_0 = -\infty\ then\ \ell_1\ else\ min(\ell_0, \ell_1), \\ & & \ if\ u_0 = +\infty\ then\ u_1\ else\ max(u_0, u_1)] \end{array}$$

Δ is a pair-narrowing operator, as it satisfies both bounding and termination requirements of Def.3.11. For instance:

$$\begin{array}{rlcl} [-\infty, +\infty] & \Delta\ [-\infty, 101] & = & [-\infty, 101] \\ [-\infty, 101] & \Delta[0 \quad , 100] & = & [0, 101] \\ [0, 100] & \Delta[0 \quad , 99] & = & [0, 100] \end{array}$$

The two notions of pair-narrowing (Definition 3.9) and strong pair-narrowing (Definition 3.11) are equivalent for a lattice P, under associativity conditions, as shown in Theorem 3.17. In order to prove it, we introduce the following auxiliary Lemma.

Lemma 3.2. *Let Δ be a pair-narrowing operator on a lattice $(\mathsf{P}, \leq)$, such that for every finite set $\{\mathsf{x}_i\}_{0 \leq i \leq n}$ and for every $\mathsf{y} \in \mathsf{P}$, $(((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta\ldots)\Delta\mathsf{x}_n)\ \Delta\ (\mathsf{x}_0 \sqcap \mathsf{x}_1 \sqcap \cdots \sqcap \mathsf{x}_n \sqcap \mathsf{y}) = (((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta\ldots)\Delta\mathsf{x}_n)\Delta\mathsf{y}$, then Δ is a strong pair-narrowing operator.*

Theorem 3.17. *Let Δ be an associative pair-narrowing operator on a lattice $(\mathsf{P}, \leq)$, such that for $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x}\Delta\mathsf{y} = \mathsf{x}\Delta(\mathsf{x}\sqcap\mathsf{y})$, then Δ is a strong pair-narrowing operator.*

Lower Bound Pair-Narrowing

When considering narrowing operators for numerical domains other slightly different notions of narrowing have been introduced in the literature, where different bounding constraints are considered: $\mathsf{x}\Delta\mathsf{y}$ is bound to be greater than $\mathsf{x} \sqcap \mathsf{y}$ and lower than x.

Definition 3.12 (lower-bound pair-narrowing [107])**.** *Let* $(\mathsf{P}, \leq)$ *be a meet-semi-lattice. A lower-bound pair-narrowing operator is a binary operator* $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *such that*

(i) Bounding: $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : (\mathsf{x} \sqcap \mathsf{y}) \leq (\mathsf{x}\Delta\mathsf{y}) \leq \mathsf{x}$.

(ii) Termination: For every decreasing chain $\mathsf{x}_0 \geq \mathsf{x}_1 \geq \ldots$*, the decreasing chain defined as*

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i\Delta\mathsf{x}_{i+1}$$

stabilizes after a finite number of terms.

Observe that not every pair-narrowing operator is also a lower-bound pair-narrowing. For example, the pair-narrowing of Example 5 doesn't satisfy the above condition. When modifying the termination constraints in Definition 3.12, we get:

Definition 3.13 (strong lower-bound pair-narrowing [107])**.** *Let* $(\mathsf{P}, \leq)$ *be a poset. A strong lower-bound pair narrowing operator is a binary operator* $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *such that*

(i) Bounding: $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : (\mathsf{x} \sqcap \mathsf{y}) \leq (\mathsf{x}\Delta\mathsf{y}) \leq \mathsf{x}$.

(ii) Termination: For every sequence $\mathsf{x}_0 \geq \mathsf{x}_1 \geq \ldots$*, the decreasing chain defined as*

$$\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i\Delta\mathsf{x}_{i+1}$$

stabilizes after a finite number of terms.

Example 10. *This notion of narrowing operator is introduced, for the octagon domain, in [106, 108], with the strong widening operator defined in Definition 3.10.*

Under particular conditions, the two notions of *pair-narrowing* and *strong lower-bound pair-narrowing* are equivalent.

Theorem 3.18. *Let* $(\mathsf{P}, \leq)$ *be a meet-semi-lattice (the greatest lower bound* $\mathsf{x} \sqcap \mathsf{y}$ *exist for all* $\mathsf{x.y} \in \mathsf{L}$*) satisfying the descending chain condition (no strictly decreasing chain in* L *can be infinite). Let* $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *be a pair-narrowing operator such that* $\mathsf{x}\Delta\mathsf{y} = \mathsf{x} \sqcap \mathsf{y}$*. Then* Δ *is a strong lower-bound pair-narrowing.*

We can bind pair-narrowing and lower-bound pair-narrowing throught next two theorems.

Theorem 3.19. *Let* $(\mathsf{P}, \leq)$ *be a poset and* Δ *be a pair-narrowing (Definition 3.9). If* $\forall \mathsf{v}, \mathsf{w} : \ \mathsf{v}\Delta(\mathsf{v} \sqcap \mathsf{w}) = \mathsf{v}\Delta\mathsf{w}$*, then* Δ *is a lower bound pair-narrowing (Definition 3.12).*

Theorem 3.20. *Let* $(\mathsf{P}, \leq)$ *be a poset and* Δ *be a lower-bound pair-narrowing (Definition 3.12).*
Consider $\mathsf{x}\Delta\mathsf{y}$*, it's simple to prove that* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{y} \leq \mathsf{x}$ *than* Δ *is a pair-narrowing (Definition 3.9).*

Widening and Narrowing Operators wrt Galois Insertions

Widening operators have already been used in order to derive abstract domains [147]. The next results show how to derive Galois insertions by introducing an abstraction function built on top of a widening operator. In order to do that, additional requirements have to be assumed on the widening operator, like idempotence and order-preservation on pairs/singletons.

Theorem 3.21. *Let* ∇ *be a pair-widening operator on a complete lattice* $(\mathsf{L}, \leq)$ *such that* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{L} : \mathsf{x} \leq \mathsf{y} \Rightarrow \mathsf{x}\nabla\mathsf{x} \leq \mathsf{y}\nabla\mathsf{y}$*. Let* A *be the set* $\{\mathsf{x}\nabla\mathsf{x} \mid \mathsf{x} \in \mathsf{L}\}$*. Then* $\alpha_{\mathsf{LA}}(\mathsf{x}) = \mathsf{x}\nabla\mathsf{x}$ *is the lower adjoint of a Galois insertion between* L *and* A*, with the upper adjoint being the identity function.*

A corresponding result can be obtained also for set-widening operators.

Theorem 3.22. *Let* $\nabla_\star$ *be a set-widening operator on a complete lattice* $(\mathsf{L}, \leq)$ *such that* $\nabla_\star(\{\mathsf{x}\})$ *is defined for each* x *in* L*, and such that* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{L} : \mathsf{x} \leq \mathsf{y} \Rightarrow \nabla_\star(\{\mathsf{x}\}) \leq \nabla_\star(\{\mathsf{y}\})$*. Let* A *be the set* $\{\nabla_\star(\{\mathsf{x}\}) \mid \mathsf{x} \in \mathsf{L}\}$*. Consider the function* $\alpha_{\mathsf{LA}} : \mathsf{L} \rightarrow \mathsf{A}$ *defined by* $\alpha_{\mathsf{LA}}(\mathsf{x}) = \nabla_\star(\{\mathsf{x}\})$*. Then,* α_{LA} *is the lower adjoint of a Galois insertion between* L *and* A*, with the upper adjoint being the identity function.*

Widening, Narrowing and Abstraction

The following theorem shows that pair widening is preserved through abstraction.

Theorem 3.23. *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* ∇_{C} *be a pair-widening on* C*. The binary operator* ∇_{D} *defined by* $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D}$, $\mathsf{d}_1\nabla_{\mathsf{D}}\mathsf{d}_2 = \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)\nabla_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_2))$ *is a pair-widening operator on* D*.*

A corresponding result can be obtained also for set-widening operators.

Theorem 3.24. *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* $\nabla_{*\mathsf{C}}$ *be a set-widening on* C*. The operator* $\nabla_{*\mathsf{D}}$ *defined by* $\forall \mathsf{S} \in \mathsf{D}$, $\nabla_{*\mathsf{D}}(\mathsf{S}) = \alpha_{\mathsf{CD}}(\nabla_{*\mathsf{C}}(\gamma_{\mathsf{DC}}(\mathsf{S}))$ *is a set-widening operator on* D*.*

As a corollary of Theorem 3.23, we can prove that pair-widening operators are preserved also when projecting a cartesian product of lattices on one of its components.

Corollary 3.1. *Let* A *and* D *be complete lattices, and let* ∇ *be a pair-widening operator over the cartesian product* $\mathsf{A} \times \mathsf{D}$*. Let* π_1 *be the projection on the first argument. The binary operator* $\nabla_{\mathsf{A}} : \mathsf{A} \times \mathsf{A} \to \mathsf{A}$ *defined by*

$$\mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}' = \pi_1(\langle \mathsf{a}, \top\rangle \nabla \langle \mathsf{a}', \top\rangle)$$

is a pair-widening operator.

Similarly, also, we can prove that narrowing operators are preserved by abstraction.

Theorem 3.25. *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \overset{\gamma_{\mathsf{DC}}}{\underset{\alpha_{\mathsf{CD}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* Δ_{C} *be a pair-narrowing on* C*. The binary operator* Δ_{D} *defined by* $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D}$, $\mathsf{d}_1 \Delta_{\mathsf{D}} \mathsf{d}_2 = \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)\Delta_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_2))$ *is a pair-narrowing operator on* D*.*

A corresponding result holds also for set-narrowing operators.

Theorem 3.26. *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \overset{\gamma_{\mathsf{DC}}}{\underset{\alpha_{\mathsf{CD}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* $\Delta_{*\mathsf{C}}$ *be a set-narrowing on* C*. The operator* $\Delta_{*\mathsf{D}}$ *defined by* $\forall \mathsf{S} \in \mathsf{D}$, $\Delta_{*\mathsf{D}}(\mathsf{S}) = \alpha_{\mathsf{CD}}(\Delta_{*\mathsf{C}}(\gamma_{\mathsf{DC}}(\mathsf{S}))$ *is a set-narrowing operator on* D*.*

As for widening operator, we can prove that pair-narrowing operators are preserved also when projecting a cartesian product of lattices on one of its components.

Corollary 3.2. *Let* A *and* D *be complete lattices, and let* Δ *be a pair-narrowing operator over the cartesian product* $\mathsf{A} \times \mathsf{D}$*. Let* π_1 *be the projection on the first argument. The binary operator* $\Delta_{\mathsf{A}} : \mathsf{A} \times \mathsf{A} \to \mathsf{A}$ *defined by*

$$\mathsf{a}\Delta_{\mathsf{A}}\mathsf{a}' = \pi_1(\langle \mathsf{a}, \top\rangle \Delta \langle \mathsf{a}', \top\rangle)$$

is a pair-narrowing operator.

Widening, Narrowing and Reduced Product

A very important operator for combining abstract domains in Abstract Interpretation, is the *reduced product* [50]. We have already seen in Theorem 3.10 and in Theorem 3.11 that the pair-widening and pair-narrowing operators can be combined when considering the cartesian product of two posets. Unfortunately, this result cannot be fully extended to the reduced product, due to the fact that pair-widening and pair-narrowing operators in general are not required to be monotone. However, getting results relating widening and narrowing operators in case of reduced product may have great impact on abstract domains used for the analysis of critical software. For instance, the octagon domain [108] can be seen as the reduced product of $2n^2$ abstract domains, each one of them focusing on an invariant of the form $\pm\mathsf{x}\pm\mathsf{y} \leq \mathsf{c}$.

Definition 3.14. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{CD}}{\overset{\gamma_{DC}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{CA}}{\overset{\gamma_{AC}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Consider the function reduce:$\mathsf{A} \times \mathsf{D} \to \mathsf{A} \times \mathsf{D}$ *defined by reduce*$(\langle \mathsf{a}, \mathsf{d} \rangle) = \sqcap\{\langle \mathsf{a}', \mathsf{d}' \rangle \mid \gamma_{AC}(\mathsf{a}) \sqcap \gamma_{DC}(\mathsf{d}) = \gamma_{AC}(\mathsf{a}') \sqcap \gamma_{DC}(\mathsf{d}')\}$
The reduced product $\mathsf{A} \sqcap \mathsf{D}$ *is defined as follows:*

$$\mathsf{A} \sqcap \mathsf{D} = \{\mathit{reduce}(\langle \mathsf{a}, \mathsf{d} \rangle) \mid \mathsf{a} \in \mathsf{A}, \mathsf{d} \in \mathsf{D}\}.$$

Moreover, the function $\gamma : \mathsf{A} \sqcap \mathsf{D} \to \mathsf{C}$ *defined by* $\gamma(\langle \mathsf{a}, \mathsf{d} \rangle) = \gamma_{AC}(\mathsf{a}) \sqcap \gamma_{DC}(\mathsf{d})$ *is the upper adjoint of a Galois insertion between* $\mathsf{A} \sqcap \mathsf{D}$ *and the domain* C*.*

We can prove (Lemma 3.4) that by combining two pair-widening operators in the reduced product at least covering is preserved, i.e. we can obtain an extrapolation operator (which does not necessarily terminate on ascending sequences, see for instance the domain of octagons [108]). The following auxiliary Lemma says that *reduce* behaves well with respect to the ordering in the reduced product $\mathsf{A} \sqcap \mathsf{D}$.

Lemma 3.3. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{CD}}{\overset{\gamma_{DC}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{CA}}{\overset{\gamma_{AC}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions. For* $\hat{\mathsf{a}} \in \mathsf{A}, \hat{\mathsf{d}} \in \mathsf{D}$, $\langle \mathsf{a}, \mathsf{d} \rangle \in \mathsf{A} \sqcap \mathsf{D}$, *if* $\mathsf{a} \leq \hat{\mathsf{a}}$ *and* $\mathsf{d} \leq \hat{\mathsf{d}}$*, then* $\langle \mathsf{a}, \mathsf{d} \rangle \leq \mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}} \rangle)$*.*

Lemma 3.4. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{CD}}{\overset{\gamma_{DC}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{CA}}{\overset{\gamma_{AC}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let ∇_{A} *and* ∇_{D} *be pair-widening operators defined on the lattice* A *and* D*, respectively.*
The binary operator $\bullet : (\mathsf{A} \sqcap \mathsf{D}) \times (\mathsf{A} \sqcap \mathsf{D}) \to (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D}$: $\langle \mathsf{a}, \mathsf{d} \rangle \bullet \langle \mathsf{a}', \mathsf{d}' \rangle = \mathit{reduce}(\langle \mathsf{a} \nabla_{\mathsf{A}} \mathsf{a}', \mathsf{d} \nabla_{\mathsf{D}} \mathsf{d}' \rangle)$ *is an extrapolator operator.*

The last Theorem shows that if the pairwise application of the pair-widening operators is always an element of the reduced product, the extrapolator of Lemma 3.4 enjoys also the termination property, thus resulting into a pair-widening operator too.

Theorem 3.27. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{CD}}{\overset{\gamma_{DC}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{CA}}{\overset{\gamma_{AC}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let ∇_{A} *and* ∇_{D} *be pair-widening operators defined on the lattice* A *and* D*, respectively, such that* $\forall \langle \mathsf{a}, \mathsf{d} \rangle \in \mathsf{A} \sqcap \mathsf{D}$, $\forall \mathsf{a}' \in A$, $\forall \mathsf{d}' \in D$: $\langle \mathsf{a} \nabla_{\mathsf{A}} \mathsf{a}', \mathsf{d} \nabla_{\mathsf{D}} \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the binary operator $\nabla : (\mathsf{A} \sqcap \mathsf{D}) \times (\mathsf{A} \sqcap \mathsf{D}) \to (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D}$: $\langle \mathsf{a}, \mathsf{d} \rangle \nabla \langle \mathsf{a}', \mathsf{d}' \rangle = \mathit{reduce}(\langle \mathsf{a} \nabla_{\mathsf{A}} \mathsf{a}', \mathsf{d} \nabla_{\mathsf{D}} \mathsf{d}' \rangle)$ *is a pair-widening operator.*

For narrowing operators, we can define a theorem corresponding to theorem 3.27. Also in this case we need some auxiliary lemma.

Lemma 3.5. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{CD}}{\overset{\gamma_{DC}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{CA}}{\overset{\gamma_{AC}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions. For* $\mathsf{a} \in \mathsf{A}$, $\mathsf{d} \in \mathsf{D}$, $\mathit{reduce}(\langle \mathsf{a}, \mathsf{d} \rangle) \leq \langle \mathsf{a}, \mathsf{d} \rangle$*.*

Lemma 3.6. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions. For* $\hat{\mathsf{a}} \in A, \hat{\mathsf{d}} \in D$, $\langle \mathsf{a}, \mathsf{d} \rangle \in \mathsf{A} \sqcap \mathsf{D}$, *if* $\hat{\mathsf{a}} \leq \mathsf{a}$ *and* $\hat{\mathsf{d}} \leq \mathsf{d}$, *then* *reduce*$(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}} \rangle) \leq \langle \mathsf{a}, \mathsf{d} \rangle$.

Theorem 3.28. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}}$ A *be Galois insertions.*
Let Δ_{A} *and* Δ_{D} *be pair-narrowing operators defined on the lattice* A *and* D, *respectively, such that* $\forall \langle \mathsf{a}, \mathsf{d} \rangle \in \mathsf{A} \sqcap \mathsf{D}$, $\forall \mathsf{a}' \in \mathsf{A}$, $\forall \mathsf{d}' \in \mathsf{D} : \langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the binary operator $\Delta : (\mathsf{A} \sqcap \mathsf{D}) \times (\mathsf{A} \sqcap \mathsf{D}) \to (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D} : \langle \mathsf{a}, \mathsf{d} \rangle \Delta \langle \mathsf{a}', \mathsf{d}' \rangle =$ *reduce*$(\langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle)$ *is a pair-narrowing operator.*

We can also obtain the corresponding results for set-widening and set-narrowing operators.

Theorem 3.29. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}}$ A *be Galois insertions.*
Let $\nabla_{*\mathsf{A}}$ *and* $\nabla_{*\mathsf{D}}$ *be set-widening operators defined on the lattice* A *and* D, *respectively, such that* $\forall S \subseteq \mathsf{A} \sqcap \mathsf{D}$, $\langle \nabla_{*\mathsf{A}}(\{a_i \mid \langle \mathsf{a}_i, d_i \rangle \in S\}), \nabla_{\mathsf{D}}(\{d_i \mid \langle \mathsf{a}_i, d_i \rangle \in S\}) \rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the operator $\nabla_* : \wp(\mathsf{A} \sqcap \mathsf{D}) \nrightarrow (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall S \subseteq \mathsf{A} \sqcap \mathsf{D} : \nabla_*(\{S\}) =$ *reduce*$(\langle \nabla_{*\mathsf{A}}(\{a_i \mid \langle \mathsf{a}_i, d_i \rangle \in S\}), \nabla_{*\mathsf{D}}(\{d_i \mid \langle \mathsf{a}_i, d_i \rangle \in S\}) \rangle)$ *is a set-widening operator.*

Theorem 3.30. *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}}$ A *be Galois insertions.*
Let $\Delta_{*\mathsf{A}}$ *and* $\Delta_{*\mathsf{D}}$ *be set-narrowing operators defined on the lattice* A *and* D, *respectively, such that* $\forall \mathsf{S} \subseteq \mathsf{A} \sqcap \mathsf{D}$, $\langle \Delta_{*\mathsf{A}}(\{\mathsf{a}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i \rangle \in \mathsf{S}\}), \Delta_{*\mathsf{D}}(\{\mathsf{d}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i \rangle \in \mathsf{S}\}) \rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the operator $\Delta_* : \wp(\mathsf{A} \sqcap \mathsf{D}) \nrightarrow (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall \mathsf{S} \subseteq \mathsf{A} \sqcap \mathsf{D} : \Delta_*(\{\mathsf{S}\})$ $=$ *reduce*$(\langle \Delta_{*\mathsf{A}}(\{\mathsf{a}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i \rangle \in \mathsf{S}\}), \Delta_{*\mathsf{D}}(\{\mathsf{d}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i \rangle \in \mathsf{S}\}) \rangle)$ *is a set-narrowing operator.*

3.5 Conclusions

In this chapter, we recalled some basic elements of Abstract Interpretation theory and the main methods to increase the efficiency and the accuracy of the analyses. On the one hand we have the combination of different domains, whereas on the other one we can combine widening and narrowing operators to improve fix-point computation. In particular, the combination of abstract domains will permit us to refine the results will be obtained by the dependency analysis which will be presented in Chapter 4.

4

Information Flow Analysis by Abstract Interpretation

4.1 Introduction

In this chapter, we introduce a new proposal of dependency analysis by abstract interpretation based on [151]. A simple imperative language, defined in Section 4.2, permits us to define the analysis and prove its correctness easily. We introduce the positive propositional formulae, we present the new the abstract domain Pos and finally, the Galois insertion between the concrete and abstract domains.

4.2 The Concrete Domain

4.2.1 The Language

For the sake of simplicity we consider a simple imperative language where programs are written by labelled commands (similar to [46]) and the syntax is defined in Table 4.1[1]. We use this language to define the analysis and prove its correctness. Afterwards, in Chapter 6, we will present the experimental results based on mainstream object-oriented languages like Java and Scala.

Let $in : \mathsf{C} \rightarrow \mathsf{L}$ and $f : \mathsf{C} \rightarrow \mathsf{L}$ be two function. By $in[\![\mathsf{c}]\!]$ and $f[\![\mathsf{c}]\!]$ we denote, respectively, the *initial* and *final label* of command c, respectively. The two function are formally defined in Table 4.2 and 4.3.

At each command corresponds one or more *actions*. The set of actions, denoted by A, consists in $\{{}^{\ell}\mathsf{skip},\ {}^{\ell}\mathsf{v} := \mathsf{exp},\ {}^{\ell}\mathsf{b},\ {}^{\ell}\mathsf{not\ b},\ {}^{\ell}\mathsf{endif},\ {}^{\ell}\mathsf{done}\}$. Let $a : \mathsf{C} \rightarrow \wp(\mathsf{A})$ be the function that, given a commands, returns the set of actions. The function a is defined in Table 4.4.

The variables appearing in a program are implicitly declared. By the notation $\mathsf{V}(\mathcal{P})$ we denote the set of variables in the program $\mathcal{P}$ and, similarly, by $\mathsf{V}(\mathsf{exp})$ and $\mathsf{V}(\mathsf{b})$

[1]In the following of the thesis, we will omit the initial and final labels of statements when we will not need them.

Table 4.1 Definition of the syntax for simple language

Variables:			
v	$\in$	V	
v	$::=$	$\mathsf{x} \mid \mathsf{y} \mid \ldots$	
Expressions			
exp	$\in$	E	
exp	$::=$	n	where $n \in \mathbb{N}$
	$\mid$	v	
	$\mid$	$\mathsf{exp}_1 \oplus \mathsf{exp}_2$	where $\oplus = \{+, -, *, /\}$
Conditions			
b	$\in$	B	
b	$::=$	true	
	$\mid$	false	
	$\mid$	$\mathsf{b}_1 \otimes \mathsf{b}_2$	where $\otimes = \{and,\ or\}$
	$\mid$	$\neg\mathsf{b}$	
	$\mid$	$\mathsf{b}_1 \oslash \mathsf{b}_2$	where $\oslash = \{\leq, >, =\}$
Labeled command			
ℓ	$\in$	L	the set of labels
c	$\in$	C	
c	$::=$	${}^{\ell}\mathsf{skip}$	
	$\mid$	${}^{\ell}\mathsf{v} := \mathsf{exp}$	
	$\mid$	$\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}$	
	$\mid$	$\mathsf{c}_1; \mathsf{c}_2$	
	$\mid$	$\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}$	
$\mathcal{P}$	$::=$	c^{ℓ}	is a program which end with label ℓ

the variables contained in expression exp and conditions b. The definition of these sets is in Table 4.5.

In order to better understand the above definitions consider Example 11.

Example 11. *Let $\mathcal{P}$ be the labelled program defined as follows.*

```
⁰x := 1;
while ¹x > 10 do
        ²x := x + 1;
³done⁴
```

The initial and final label of $\mathcal{P}$ are $in[\![\mathcal{P}]\!] = 0$ and $f[\![\mathcal{P}]\!] = 4$, respectively. Whereas the set of actions A *contains* $\{{}^{0}\mathsf{x} := 1,\ {}^{1}\mathsf{x} > 10,\ {}^{2}\mathsf{x} := \mathsf{x} + 1,\ {}^{3}\mathsf{done}\}$.

Table 4.2 Definition of initial label function

Initial label function		
$in[\![{}^{\ell}\mathsf{skip}]\!]$	$\overset{\text{def}}{=}$	ℓ
$in[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!]$	$\overset{\text{def}}{=}$	ℓ
$in[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!]$	$\overset{\text{def}}{=}$	ℓ
$in[\![\mathsf{c}_1; \mathsf{c}_2]\!]$	$\overset{\text{def}}{=}$	$in[\![\mathsf{c}_1]\!]$
$in[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!]$	$\overset{\text{def}}{=}$	ℓ

Table 4.3 Definition of final label function

Final label function					
$\mathcal{P}$	$::=$	c^{ℓ}	$f[\![\mathcal{P}]\!]$	$\equiv$	ℓ
			$f[\![\mathsf{c}]\!]$	$\equiv$	$f[\![P]\!]$
c	$::=$	${}^{\ell}\mathsf{skip}$	$f[\![{}^{\ell}\mathsf{skip}]\!]$	$\equiv$	$f[\![\mathsf{c}]\!]$
	$\mid$	${}^{\ell}\mathsf{v} := \mathsf{exp}$	$f[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!]$	$\equiv$	$f[\![\mathsf{c}]\!]$
	$\mid$	$\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}$	$f[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!]$	$\equiv$	$f[\![\mathsf{c}]\!]$
			$f[\![\mathsf{c}_1]\!]$	$\equiv$	ℓ'
			$f[\![\mathsf{c}_2]\!]$	$\equiv$	ℓ'
	$\mid$	$\mathsf{c}_1; \mathsf{c}_2$	$f[\![\mathsf{c}_1; \mathsf{c}_2]\!]$	$\equiv$	$f[\![\mathsf{c}]\!]$
			$f[\![\mathsf{c}_1]\!]$	$\equiv$	$in[\![\mathsf{c}_2]\!]$
			$f[\![\mathsf{c}_2]\!]$	$\equiv$	$f[\![\mathsf{c}]\!]$
	$\mid$	$\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}$	$f[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ \mathsf{done}]\!]$	$\equiv$	$f[\![\mathsf{c}]\!]$
			$f[\![\mathsf{c}]\!]$	$\equiv$	ℓ

4.2.2 Concrete Domain

An *environment* $\rho \in \mathcal{E}$ is a function $\rho : \mathsf{V} \longrightarrow \mathbb{N}$ which assign to each variable the respective value. A *state* $\sigma \in \Sigma \equiv (\mathsf{L} \times \mathcal{E})$ is a pair $\langle \ell, \rho \rangle$ where the program label ℓ is the label of the statement to be executed, and the environment ρ defines the current values of program variables.

In Example 11, the states are[2]: $\langle 0, \varepsilon \rangle$, $\langle 1, \{\mathsf{x} \mapsto 1\} \rangle$, $\langle 3, \{\mathsf{x} \mapsto 1\} \rangle$, $\langle 2, \{\mathsf{x} \mapsto 1\} \rangle$, $\langle 1, \{\mathsf{x} \mapsto 2\} \rangle$, $\langle 2, \{\mathsf{x} \mapsto 2\} \rangle$, $\cdots$, $\langle 3, \{\mathsf{x} \mapsto n\} \rangle$, $\langle 4, \{\mathsf{x} \mapsto n\} \rangle$, $\langle 4, \{\mathsf{x} \mapsto 1\} \rangle$.

We denote by $\mathrm{E}[\![\mathsf{exp}]\!]\rho$ and $\mathrm{B}[\![\mathsf{b}]\!]\rho$ the expression and the condition evaluation of $\mathsf{exp} \in \mathsf{E}$ and $\mathsf{b} \in \mathsf{B}$, respectively. Their definition are reported by Table 4.6 and 4.7, respectively.

At the beginning of the execution, variables could be related to any value. Therefore, the set I of *possible initial states* of a program $\mathcal{P}$ is $\mathsf{I}[\![\mathcal{P}]\!] \equiv \{\langle\ in[\![\mathcal{P}]\!], \rho \rangle \mid \rho \in \mathcal{E}\}$. In the same way, we can define $\mathsf{F}[\![\mathcal{P}]\!]$ as the set of *possible final states* of $\mathcal{P}$: $\mathsf{F}[\![\mathcal{P}]\!] \equiv \{\langle\ f[\![\mathcal{P}]\!], \rho \rangle \mid \rho \in \mathcal{E}\}$.

Consider again Example 11: $\mathsf{I}[\![\mathcal{P}]\!] = \{\langle 0, \varepsilon \rangle\}$ and $\mathsf{F}[\![\mathcal{P}]\!] = \{\langle 4, \{\mathsf{x} \mapsto 1\} \rangle, \langle 4, \{\mathsf{x} \mapsto 2\} \rangle, \cdots, \langle 4, \{\mathsf{x} \mapsto n\} \rangle\}$.

[2]The symbol ε indicates an empty environment

Table 4.4 Definition of action function

Action function		
$a\llbracket {}^{\ell}\mathsf{skip}\rrbracket$	$\stackrel{\text{def}}{=}$	$\{{}^{\ell}skip\}$
$a\llbracket {}^{\ell}\mathsf{v} := \mathsf{exp}\rrbracket$	$\stackrel{\text{def}}{=}$	$\{{}^{\ell}\mathsf{v} := \mathsf{exp}\}$
$a\llbracket \mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}\rrbracket$	$\stackrel{\text{def}}{=}$	$\{{}^{\ell}\mathsf{b},\ {}^{\ell}\mathsf{not\ b},{}^{\ell'}\ \mathsf{endif}\} \cup a\llbracket \mathsf{c}_1\rrbracket \cup a\llbracket \mathsf{c}_2\rrbracket$
$a\llbracket \mathsf{C}_1; \mathsf{C}_2\rrbracket$	$\stackrel{\text{def}}{=}$	$a\llbracket \mathsf{c}_1\rrbracket \cup a\llbracket \mathsf{c}_2\rrbracket$
$a\llbracket \mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}\rrbracket$	$\stackrel{\text{def}}{=}$	$\{{}^{\ell}\mathsf{b},{}^{\ell}\ \mathsf{not\ b},{}^{\ell'}\ \mathsf{done}\} \cup a\llbracket \mathsf{c}\rrbracket$

Table 4.5 Definition of variables functions

$\mathsf{V}\llbracket n\rrbracket$	$\stackrel{\text{def}}{=}$	$\emptyset$
$\mathsf{V}\llbracket \mathsf{v}\rrbracket$	$\stackrel{\text{def}}{=}$	$\{\mathsf{v}\}$
$\mathsf{V}\llbracket \mathsf{exp}_1 \oplus \mathsf{exp}_2\rrbracket$	$\stackrel{\text{def}}{=}$	$\mathsf{V}\llbracket \mathsf{exp}_1\rrbracket \cup \mathsf{V}\llbracket \mathsf{exp}_2\rrbracket$
$\mathsf{V}\llbracket \mathsf{true}\rrbracket$	$\stackrel{\text{def}}{=}$	$\emptyset$
$\mathsf{V}\llbracket \mathsf{false}\rrbracket$	$\stackrel{\text{def}}{=}$	$\emptyset$
$\mathsf{V}\llbracket \mathsf{b}_1 \otimes \mathsf{b}_2\rrbracket$	$\stackrel{\text{def}}{=}$	$\mathsf{V}\llbracket \mathsf{b}_1\rrbracket \cup \mathsf{V}\llbracket \mathsf{b}_2\rrbracket$
$\mathsf{V}\llbracket \mathsf{exp}_1 \oslash \mathsf{exp}_2\rrbracket$	$\stackrel{\text{def}}{=}$	$\mathsf{V}\llbracket \mathsf{exp}_1\rrbracket \cup \mathsf{V}\llbracket \mathsf{exp}_2\rrbracket$
$\mathsf{V}\llbracket \mathsf{skip}\rrbracket$	$\stackrel{\text{def}}{=}$	$\emptyset$
$\mathsf{V}\llbracket \mathsf{v} := \mathsf{exp}\rrbracket$	$\stackrel{\text{def}}{=}$	$\{\mathsf{v}\} \cup \mathsf{V}\llbracket \mathsf{exp}\rrbracket$
$\mathsf{V}\llbracket \mathsf{if\ b\ then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ \mathsf{endif}\rrbracket$	$\stackrel{\text{def}}{=}$	$\mathsf{V}\llbracket \mathsf{b}\rrbracket \cup \mathsf{V}\llbracket \mathsf{c}_1\rrbracket \cup \mathsf{V}\llbracket \mathsf{c}_2\rrbracket$
$\mathsf{V}\llbracket \mathsf{c}_1; \mathsf{c}_2\rrbracket$	$\stackrel{\text{def}}{=}$	$\mathsf{V}\llbracket \mathsf{c}_1\rrbracket \cup \mathsf{V}\llbracket \mathsf{c}_2\rrbracket$
$\mathsf{V}\llbracket \mathsf{while\ b\ do\ c\ done}\rrbracket$	$\stackrel{\text{def}}{=}$	$\mathsf{V}\llbracket \mathsf{b}\rrbracket \cup \mathsf{V}\llbracket \mathsf{c}\rrbracket$

The *labelled transition semantics* $\mathsf{T}\llbracket \mathsf{c}\rrbracket$ of a command c in a program $\mathcal{P}$ is a set of transitions $\langle \sigma_1, \mathsf{a}, \sigma_2\rangle$ between a state σ_1 and its next states σ_2 by an action a. The triples $\langle \sigma_1, \mathsf{a}, \sigma_2\rangle$ of states could be denoted by $\sigma_1 \xrightarrow{\mathsf{a}} \sigma_2$. The transition function $\mathsf{T} : \mathsf{C} \rightarrow \wp(\Sigma \times \mathsf{A} \times \Sigma)$, aimed at tracking all reachable states, is defined in Figure 4.1. For instance, a transition based on Example 11 is $\langle 0, \varepsilon\rangle \xrightarrow{\mathsf{x:=1}} \langle 1, \{\mathsf{x} \mapsto 1\}\rangle$.

A *labelled transition system* is a tuple $\langle \Sigma, \mathsf{I}, \mathsf{F}, \mathsf{A}, \mathsf{T}\rangle$, where Σ is a nonempty set of states, $\mathsf{I} \subseteq \Sigma$ is a nonempty set of initial states, $\mathsf{F} \subseteq \Sigma$ is a set of final states, A is a nonempty set of actions and $\mathsf{T} \in \wp(\Sigma \times \mathsf{A} \times \Sigma)$ is the labelled transition relation.

We define the *partial trace semantics* of a transition system as the set of all possible traces, of element in Σ and denoted byΣ *, recording the observation of an execution during a finite time, starting from an initial state and possibly reaching a final state.

$$\Sigma^\star \in \wp(\Sigma \times \mathsf{A} \times \Sigma)$$

$$\Sigma^\star = \{\sigma_0 \xrightarrow{\mathsf{a}_0} \ldots \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \mid n \geq 1 \land \sigma_0 \in \mathsf{I} \land \forall i \in [0, n-1] : \sigma_i \xrightarrow{\mathsf{a}_i} \sigma_{i+1} \in \mathsf{T}\}$$

Table 4.6 Evaluation of expression

E	$\in$	$\mathsf{E} \to (\mathcal{E} \to \wp(\mathbb{N}))$
$\mathrm{E}[\![n]\!]\rho$	$\overset{\text{def}}{=}$	$\{n\}$
$\mathrm{E}[\![\mathsf{v}]\!]\rho$	$\overset{\text{def}}{=}$	$\{\rho(\mathsf{v})\}$
$\mathrm{E}[\![\mathsf{exp}_1 \oplus \mathsf{exp}_2]\!]\rho$	$\equiv$	$\{v_1 \oplus v_2 \mid v_1 \in \mathrm{E}[\![\mathsf{exp}_1]\!]\rho \wedge v_2 \in \mathrm{E}[\![\mathsf{exp}_2]\!]\rho\}$

Table 4.7 Evaluated of boolean conditions

B	$\in$	$\mathsf{B} \to (\mathcal{E} \to \wp(\mathbb{N}))$
$\mathrm{B}[\![\mathsf{true}]\!]\rho$	$\overset{\text{def}}{=}$	$\{\mathsf{true}\}$
$\mathrm{B}[\![\mathsf{false}]\!]\rho$	$\overset{\text{def}}{=}$	$\{\mathsf{false}\}$
$\mathrm{B}[\![\mathsf{b}_1 \otimes \mathsf{b}_2]\!]\rho$	$\overset{\text{def}}{=}$	$\{b_1 \otimes b_2 \mid b_1 \in \mathrm{B}[\![\mathsf{b}_1]\!]\rho \wedge b_2 \in \mathrm{B}[\![\mathsf{b}_2]\!]\rho\}$
$\mathrm{B}[\![\mathsf{exp}_1 \oslash \mathsf{exp}_2]\!]\rho$	$\overset{\text{def}}{=}$	$\{\mathsf{true} \mid \exists v_1 \in \mathrm{E}[\![\mathsf{exp}_1]\!]\rho : v_2 \in \mathrm{E}[\![\mathsf{exp}_2]\!]\rho : v_1 \oslash v_2\} \cup$ $\{\mathsf{false} \mid \exists v_1 \in \mathrm{E}[\![\mathsf{exp}_1]\!]\rho : v_2 \in \mathrm{E}[\![\mathsf{exp}_2]\!]\rho : \mathsf{not}(v_1 \oslash v_2)\}$

Let $\pi_0, \pi_1 \in \Sigma^\star$ be two partial traces. We define the following lattice operators:

- $\pi_0 \preceq \pi_1$ if and only if π_0 is a subtrace of π_1
- $\pi_0 \curlywedge \pi_1 = \pi$ such that $(\pi \preceq \pi_1) \wedge (\pi \preceq \pi_2)$ and $(\forall \pi' : (\pi' \preceq \pi_1) \wedge (\pi' \preceq \pi_2)).\pi' \preceq \pi$.

$\Sigma^\star$ equipped with the order relation "$\preceq$" and meet operator "$\curlywedge$" forms the meet semi lattice $\langle \Sigma^\star, \preceq, \curlywedge \rangle$.
This partial trace semantics can be expressed also in fixpoint form.

$$\begin{aligned} &\Sigma^\star = \mathrm{lfp}^{\subseteq} F \text{ where} \\ &F \in \wp(\Sigma \times \mathsf{A} \times \Sigma) \to \wp(\Sigma \times \mathsf{A} \times \Sigma) \end{aligned}$$

Where

$$\begin{aligned} &F(\mathsf{X}) \overset{\text{def}}{=} \{\sigma \xrightarrow{\mathsf{a}'} \sigma' \in \mathsf{T} \mid \sigma \in \mathsf{I}\} \cup \\ &\{\sigma_0 \xrightarrow{\mathsf{a}_0} \ldots \xrightarrow{\mathsf{a}_{n-2}} \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \mid \sigma_0 \xrightarrow{\mathsf{a}_0} \ldots \xrightarrow{\mathsf{a}_{n-2}} \sigma_{n-1} \in \mathsf{X} \wedge \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \in \mathsf{T}\} \end{aligned}$$

Let $\langle \wp(\Sigma^\star), \subseteq, \emptyset, \Sigma^\star, \cap, \cup \rangle$ be a complete lattice of partial execution traces, where "$\subseteq$" is the classical subset relation, "$\cup$" is the set union and "$\cap$" the set intersection.

4.3 Abstract Domain of Propositional Formulae

4.3.1 Propositional Formulae

Among all the abstract domains which are used in abstract interpretation of logic programs, two that have received considerable attention are Pos and Sharing. The

Figure 4.1 Definition of the concrete transition semantics

$$\mathsf{T}[\![{}^{\ell}\mathsf{skip}]\!] = \{\langle \ell,\rho\rangle \xrightarrow{{}^{\ell}\mathsf{skip}} \langle f[\![{}^{\ell}\mathsf{skip}]\!],\rho\rangle \mid \rho \in \mathcal{E}\}$$

$$\mathsf{T}[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!] = \{\langle \ell,\rho\rangle \xrightarrow{{}^{\ell}\mathsf{v:=exp}} \langle f[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!],\rho[\mathsf{v} \leftarrow v]\rangle \mid \rho \in \mathcal{E} \wedge v \in \mathrm{E}[\![\mathsf{exp}]\!]\rho\}$$

$$\mathsf{T}[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!] = \mathsf{T}[\![\mathsf{c}_1]\!] \cup \mathsf{T}[\![\mathsf{c}_2]\!] \cup$$

$$\{\langle \ell,\rho\rangle \xrightarrow{{}^{\ell}\mathsf{b}} \langle in[\![\mathsf{c}_1]\!],\rho\rangle \mid \rho \in \mathcal{E} \wedge\ \mathsf{true}\ \in \mathrm{B}[\![\mathsf{b}]\!]\rho\} \cup$$

$$\{\langle \ell,\rho\rangle \xrightarrow{{}^{\ell}\mathsf{not\ b}} \langle in[\![\mathsf{c}_2]\!],\rho\rangle \mid \rho \in \mathcal{E} \wedge\ \mathsf{false}\ \in \mathrm{B}[\![\mathsf{b}]\!]\rho\} \cup$$

$$\{\langle \ell',\rho\rangle \xrightarrow{{}^{\ell'}\mathsf{endif}} \langle f[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!],\rho\rangle \mid \rho \in \mathcal{E}\}$$

$$\mathsf{T}[\![\mathsf{c}_1;\mathsf{c}_2]\!] = \mathsf{T}[\![\mathsf{c}_1]\!] \cup \mathsf{T}[\![\mathsf{c}_2]\!]$$

$$\mathsf{T}[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!] = \{\langle \ell,\rho\rangle \xrightarrow{{}^{\ell}\mathsf{not\ b}} \langle \ell',\rho\rangle \mid \rho \in \mathcal{E} \wedge \mathsf{false} \in \mathrm{B}[\![\mathsf{b}]\!]\rho\} \cup$$

$$\{\langle \ell,\rho\rangle \xrightarrow{{}^{\ell}\mathsf{b}} \langle in[\![\mathsf{c}]\!],\rho\rangle \mid \rho \in \mathcal{E} \wedge \mathsf{true} \in \mathrm{B}[\![\mathsf{b}]\!]\rho\} \cup \mathsf{T}[\![\mathsf{c}]\!] \cup$$

$$\{\langle \ell',\rho\rangle \xrightarrow{{}^{\ell'}\mathsf{done}} \langle f[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!],\rho\rangle \mid \rho \in \mathcal{E}\}$$

former, originally introduced by Marriott and Søndergaard [86] consists of the class of positive Boolean functions [6, 39, 38, 101]. Our approach involves only the Pos domain. This domain is most commonly applied to the analysis of groundness dependencies for logic programs and it is considered a clean and an intelligible abstract domain. In [34], the authors showed that Sharing and Pos are isomorphic, even though the interpretation of the Boolean functions differ from one to the other.

Let $\overline{\mathsf{V}} = \{\bar{\mathsf{x}}, \bar{\mathsf{y}}, \bar{\mathsf{z}}, \cdots\}$ be a countably infinite set of propositional variables and let $FP(\overline{\mathsf{V}})$ be the set of finite subset of variables of $\overline{\mathsf{V}}(FP(\overline{\mathsf{V}}) = \wp(\overline{\mathsf{V}}))$. The set of propositional formulae constructed over the variables of $\overline{\mathsf{V}}$ and the logical connectives in$\Gamma \subseteq \{\wedge, \vee, \rightarrow, \neg\}$ is denoted by $\Omega(\Gamma)$. For any $\mathsf{U} \in FP(\overline{\mathsf{V}})$,$\Omega_{\mathsf{U}}(\Gamma)$ consists of formulae using only the variables of U and the connectives of Γ.

A *truth-assignment* is a function $r : \overline{\mathsf{V}} \rightarrow \{\mathtt{T}, \mathtt{F}\}$ that assign to each propositional variable the value true ($\mathtt{T}$) or the value false ($\mathtt{F}$). Given a formula $\mathsf{f} \in \Omega(\{\wedge, \vee, \rightarrow, \neg\})$, $r \vDash \mathsf{f}$ means that r satisfies f, and $\mathsf{f}_1 \vDash \mathsf{f}_2$ is a shorthand for "$r \vDash \mathsf{f}_1$ implies $r \vDash \mathsf{f}_2$". $\Omega(\{\wedge, \vee, \rightarrow, \neg\})$ is ordered by $\mathsf{f}_1 \trianglelefteq \mathsf{f}_2$ if $\mathsf{f}_1 \vDash \mathsf{f}_2$. Two formulae f_1 and f_2 are logically equivalent, denoted $\mathsf{f}_1 \equiv \mathsf{f}_2$ if $\mathsf{f}_1 \vDash \mathsf{f}_2$ and $\mathsf{f}_2 \vDash \mathsf{f}_1$.

The *unit assignment* u is defined by $u(\mathsf{x}) = \mathtt{T}$ for all $\mathsf{x} \in \overline{\mathsf{V}}$. We define the set of positive formulae by: $\mathsf{Pos} = \{\mathsf{f} \in \Omega(\{\wedge, \vee, \rightarrow, \neg\}) \mid u \vDash \mathsf{f}\}$ Some obvious examples: $\mathtt{T}, \mathsf{x}_1 \in \mathsf{Pos}$ and $\mathtt{F}, \neg\mathsf{x}_1 \notin \mathsf{Pos}$.

We can consider the propositional formula ϕ as a conjunction of subformulae $(\zeta_0 \wedge \ldots \wedge \zeta_n)$. We denote the set of subformulas of ϕ as Sub_ϕ. Let $\triangledown$ be least upper bound operator on propositional formula, $\triangledown\{\phi_0, \ldots, \phi_n\} = \bigwedge\{Sub_{\phi_0}, \ldots, Sub_{\phi_n}\}$.

Therefore $(\mathsf{Pos}, \trianglelefteq, \triangledown)$ is a join semi lattice. Moreover, consider $\ominus : \mathsf{Pos} \times \mathsf{Pos} \rightarrow \mathsf{Pos}$: a binary operator defined as simplification between two propositional formulae: $\phi_0 \ominus \phi_1 = \bigwedge(Sub_{\phi_0} \setminus Sub_{\phi_1})$. This simplification permits us to obtain all the implication in ϕ_0 which are not contained in ϕ_1. For an introduction to the basic concepts of propositional logic, we refer the interested reader to [21].

4.3.2 Propositional Formulae Domain

The idea is to use logic formulae to represent dependency between variables (propagation of sensitive/insensitive information) and detect information leakages evaluating formulae on truth-assignment functions. The analysis of a program involves the following steps:

- For each program instruction construct a propositional formula (ϕ_i), through a fixpoint algorithm, which show an over-approximation of dependencies that occur between variables.
- Consider the public/private partitions of variables and the truth-assignment function ψ, that assigns to a propositional variable the value T (true) or the value F (false) if the corresponding variable is respectively private or public. If ψ does not satisfy ϕ_i for all program states i, there could be some information leakages.

The logic formulae, obtained from program's instructions, are in the form:

$$\bigwedge_{0 \leq i \leq n \ 0 \leq j \leq m} \{\overline{\mathsf{x}}_\mathsf{i} \rightarrow \overline{\mathsf{y}}_\mathsf{j}\}$$

which means that the values of variable $\overline{\mathsf{y}}_\mathsf{j}$ could depend on the values of variable $\overline{\mathsf{x}}_\mathsf{i}$. For instance, consider the statement $\mathsf{x} := \mathsf{y}$: we obtain the formula $\overline{\mathsf{y}} \rightarrow \overline{\mathsf{x}}$, while for the instruction $\mathsf{if}(\mathsf{x} == 0)$ then $\mathsf{y} := \mathsf{z}$ the corresponding formula is $(\overline{\mathsf{x}} \rightarrow \overline{\mathsf{y}}) \wedge (\overline{\mathsf{z}} \rightarrow \overline{\mathsf{y}})$. Notice that the propositional variable $\overline{\mathsf{u}}$ corresponds to the program variable u.

In order to better understand how our new dependency analysis works, consider the following example.

Example 12. *When the PIN reaches the issuing bank, its correspondence with the validation data (i.e. user PAN and possibly other public data, such as the card expiration date or the customer name) is checked via a verification API. Consider the case study showed in [30]. The authors consider a strict subset of the real PIN verification function named Encrypted_PIN_Verify [80].*

This function checks the equality of the actual user PIN and the trial PIN inserted at the ATM and returns the result of the verification or an error code. The former PIN is derived through the PIN derivation key pdk *and from the public data* offset, vdata, dectab*, while the latter comes encrypted under key* k *as* EPB *(Encrypted PIN Block). Note that the two keys are pre-loaded in the HSM (Hardware Security*

Figure 4.2 Dependency analysis example

```
PIN_V(PAN, EPB, len, offset, vdata, dectab) {
    $x_1 := enc_{pdk}(vdata)$;
    $x_2 := left(len, x_1)$;
    $x_3 := decimalize(dectab, x_2)$;
    $x_4 := sum\_mod10(x_3, offset)$;
    $x_5 := dec_k(EPB)$;
    $x_6 := fcheck(x_5)$;
    if ($x_6 = \bot$){
        result := "format wrong";
        return result;
    }
    if ($x_4 = x_6$)
        result := "PIN correct";
    else
        result := "PIN wrong";
    return result;
}
```

Module) and are never exposed to the untrusted external environment. The code is in Figure 4.2.

When we apply the steps defined above we obtain, at the end of the code the following formula:

$$\begin{aligned}\phi =&(\overline{\mathsf{vdata}} \to \overline{x_1}) \wedge (\overline{\mathsf{len}} \to \overline{x_2}) \wedge (\overline{x_1} \to \overline{x_2}) \wedge (\overline{\mathsf{dectab}} \to \overline{x_3}) \wedge (\overline{x_2} \to \overline{x_3}) \wedge \\ &(\overline{\mathsf{offset}} \to \overline{x_4}) \wedge (\overline{x_3} \to \overline{x_4}) \wedge (\overline{\mathsf{EPB}} \to \overline{x_5}) \wedge (\overline{x_5} \to \overline{x_6}) \wedge \\ &(\overline{x_6} \to \overline{\mathsf{result}}) \wedge (\overline{x_4} \to \overline{\mathsf{result}}) \wedge (\overline{x_6} \to \overline{\mathsf{result}})\end{aligned}$$

Let $\psi : \mathsf{V} \to \mathsf{L}, \mathsf{H}$ be a function that assign H *class to all* x_n *variables and* L *class to other variables. In this way $\overline{\psi}$, the correspondent truth-assignment function, does not satisfy ϕ. In fact, in both* if *statements, the public variable* result *depends by one or more private variables (namely* x_6 *in one case and* x_4 *and* x_6 *in the other one).*

4.3.3 Abstract Domain for Pos

An abstract state $\sigma^\sharp \in \Sigma^\sharp \overset{\text{def}}{=} \mathsf{L} \times \mathsf{Pos}$ is a pair $\langle \ell, \phi \rangle$ in which $\phi \in \mathsf{Pos}$ denotes the dependencies that occur among program variables, up to label $\ell \in \mathsf{L}$. Given a pair $\sigma^\sharp = \langle \ell, \phi \rangle$, we define $l(\sigma^\sharp) = \ell$ and $r(\sigma^\sharp) = \phi$. Let $BV(\mathsf{c})$, defined in Table 4.8, be the set of bound variables of command c.
The *abstract transition semantics* of command c is defined by $\overline{\mathsf{T}}[\![\mathsf{c}]\!]$. Similarly to the concrete domain we denote this transition by $\sigma_1^\sharp \to \sigma_2^\sharp$ and we define it in Figure 4.3.

Consider two set of abstract states S_1 and S_2 such that

$$\mathsf{S}_1 = \{\langle \ell_0^1, \phi_0^1\rangle, \dots, \langle \ell_n^1, \phi_n^1\rangle\} \; \mathsf{S}_2 = \{\langle \ell_0^2, \phi_0^2\rangle, \dots, \langle \ell_m^2, \phi_m^2\rangle\}$$

The partial ordering is define by $\mathsf{S}_1 \sqsubseteq^\sharp \mathsf{S}_2 \iff n \leq m$, $\forall i \in [0,n], \ell_i^1 = \ell_i^2$ and $\forall i \in [0,n], \phi_i^1 \trianglelefteq \phi_i^2$.
Let $\mathsf{S}_0, \dots \mathsf{S}_\mathsf{n} \in \wp(\Sigma^\sharp)$ be sets of abstract states. The join operation "$\sqcup^\sharp$" is defined by:

$$\begin{aligned}\sqcup^\sharp\{\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n}\} = &\bigcup(\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n}) \\ &\cup\{\langle\, \ell,\phi\,\rangle \mid \phi = \nabla\{\phi' \mid \langle \ell,\phi'\rangle \in \bigcup(\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n})\}\} \\ &\setminus \{\langle \ell,\phi\rangle \in \bigcup(\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n}) \mid \exists \langle \ell,\phi'\rangle \in \bigcup(\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n}) \wedge \phi \neq \phi'\}\end{aligned}$$

and the meet operation "$\sqcap^\sharp$" by:

$$\begin{aligned}\sqcap^\sharp\{\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n}\} = &\{\langle \ell,\phi\,\rangle \in \mathsf{S}' \mid \mathsf{S}' \in \{\mathsf{S}_0, \dots, \mathsf{S}_\mathsf{n}\} \wedge \\ &\forall i \in [0,n]. \exists \langle \ell,\phi_i'\rangle \in \mathsf{S}_\mathsf{i} \wedge \phi \trianglelefteq \phi_i'\}\end{aligned}$$

$\langle \wp(\Sigma^\sharp), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp\rangle$ is a complete lattice.

Let $\mathsf{I}^\sharp[\![\mathcal{P}]\!] = \{\langle in[\![\mathcal{P}]\!], \phi\rangle \mid \phi \in \mathsf{Pos}\}$ be the set of possible initial abstract state of program $\mathcal{P}$. We define the *abstract semantics* as the set of all finite sets of abstract states, denoted by$\Sigma\ ^{\star\sharp}$, reachable during one or more execution, in a finite time. For each element $\mathsf{S} \in \Sigma^{\star\sharp}$ we can denote by $\mathsf{S}^\dashv$ the set of terminal states, defined as $\mathsf{S}^\dashv = \{\sigma_0^\sharp \mid \nexists \sigma_1^\sharp \in \mathsf{S}. \sigma_0^\sharp \to \sigma_1^\sharp \in \overline{\mathsf{T}}\}$ and by $\ell(\mathsf{S})$ all labels of S. Let $\mathsf{S}_{\sigma_0^\sharp,\sigma_n^\sharp}$ denote a set of states, called *abstract sequence*, that contains a starting state $\sigma_0^\sharp$ and an ending state $\sigma_n^\sharp$ such that $\forall i \in [0, n-1], \sigma_i^\sharp \to \sigma_{i+1}^\sharp \in \overline{\mathsf{T}}$. Notice that $\mathsf{S}^\dashv_{\sigma_0^\sharp,\sigma_\mathsf{n}^\sharp} = \{\sigma_n^\sharp\}$.

We express the abstract semantics in fixpoint form.

$$\begin{aligned}&\Sigma^{\star\sharp} = lfp^{\sqsubseteq} F^\sharp \text{where} \\ &F^\sharp \in \Sigma^{\star\sharp} \to \Sigma^{\star\sharp}\end{aligned}$$

Table 4.8 Definition of BV function

$BV(^\ell\mathsf{skip})$	$=$	$\{\emptyset\}$
$BV(^\ell\mathsf{v} := \mathsf{exp})$	$=$	$\{\overline{\mathsf{x}}\}$
$BV(\mathsf{c}_0; \mathsf{c}_1)$	$=$	$BV(\mathsf{c}_0) \cup BV(\mathsf{c}_1)$
$BV(\mathsf{if}\ ^\ell\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ ^{\ell'}\mathsf{endif})$	$=$	$BV(\mathsf{c}_0) \cup BV(\mathsf{c}_1)$
$BV(\mathsf{while}\ ^\ell\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ ^{\ell'}\mathsf{done})$	$=$	$BV(\mathsf{c})$

Figure 4.3 Definition of abstract transitional semantics

$$
\begin{aligned}
&\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{skip}]\!] = \{\langle \ell,\phi \rangle \to \langle f[\![{}^{\ell}\mathsf{skip}]\!], \phi\rangle\}\\
&\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!] = \{\langle \ell,\phi \rangle \to \langle f[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!], \phi_0\rangle\}\\
&\overline{\mathsf{T}}[\![\mathsf{c}_0;\mathsf{c}_1]\!] = \overline{\mathsf{T}}[\![\mathsf{c}_0]\!] \cup \overline{\mathsf{T}}[\![\mathsf{c}_1]\!]\\
&\overline{\mathsf{T}}[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}]\!] = \overline{\mathsf{T}}[\![\mathsf{c}_0]\!] \cup \overline{\mathsf{T}}[\![\mathsf{c}_1]\!] \cup\\
&\quad \{\langle \ell,\phi \rangle \to \langle in[\![\mathsf{c}_0]\!], \phi\rangle\} \cup \{\langle \ell,\phi \rangle \to \langle in[\![\mathsf{c}_1]\!], \phi\rangle\} \cup\\
&\quad \{\langle \ell',\phi \rangle \to \langle f[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}]\!], \phi_1\rangle\}\\
&\quad \{\langle \ell',\phi \rangle \to \langle f[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}]\!], \phi_2\rangle\}\\
&\overline{\mathsf{T}}[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!] = \overline{\mathsf{T}}[\![\mathsf{c}]\!] \cup \{\langle \ell,\phi \rangle \to \langle in[\![\mathsf{c}]\!], \phi\rangle\} \cup\\
&\quad \{\langle \ell',\phi \rangle \to \langle f[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!], \phi_3\rangle\}
\end{aligned}
$$

where

$$
\begin{aligned}
\phi_0 = &\bigwedge\{\overline{\mathsf{y}} \to \overline{\mathsf{x}} \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{exp}]\!] \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\}\\
&\bigwedge\{\overline{\mathsf{z}} \to \overline{\mathsf{w}} \mid \overline{\mathsf{z}} \to \overline{\mathsf{x}}, \overline{\mathsf{x}} \to \overline{\mathsf{w}} \in \phi\} \wedge (\phi \ominus \bigwedge\{\overline{\mathsf{y}} \to \overline{\mathsf{x}} \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}} \wedge \overline{\mathsf{x}} \notin \overline{\mathsf{V}}[\![\mathsf{exp}]\!]\})\\
\phi_1 = &\bigwedge\{\overline{\mathsf{y}} \to \overline{\mathsf{x}} \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{b}]\!] \wedge x \in BV(\mathsf{c}_0) \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\} \wedge \phi\\
\phi_2 = &\bigwedge\{\overline{\mathsf{y}} \to \overline{\mathsf{x}} \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{b}]\!] \wedge x \in BV(\mathsf{c}_1) \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\} \wedge \phi\\
\phi_3 = &\bigwedge\{\overline{\mathsf{y}} \to \overline{\mathsf{x}} \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{b}]\!] \wedge \overline{\mathsf{x}} \in BV(\mathsf{c}) \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\} \wedge \phi
\end{aligned}
$$

where

$$
\begin{aligned}
F^\sharp(\mathsf{X}) \overset{\text{def}}{=} &\{\sigma^\sharp \mid \sigma^\sharp \in \mathsf{I}^\sharp\} \cup\\
&\{\mathsf{S}_{\sigma_0^\sharp,\sigma_n^\sharp} \mid n \geq 1 \wedge \sigma_0^\sharp \in \mathsf{I}^\sharp \wedge \mathsf{S}_{\sigma_0^\sharp,\sigma_{n-1}^\sharp} \in \mathsf{X} \wedge \sigma_{n-1}^\sharp \to \sigma_n^\sharp \in \overline{\mathsf{T}}\} \cup\\
&\{\sqcup^\sharp\{\mathsf{S}_{\sigma_0^\sharp,\sigma_n^\sharp} \mid \mathsf{S}_{\sigma_0^\sharp,\sigma_n^\sharp} \in \mathsf{X}\}\}
\end{aligned}
$$

We will denote by the lattice $\langle \wp(\Sigma^{\star\sharp}), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp\rangle$ the abstract states set.

4.3.4 An Instrumented Concrete Domain

In order to simplify the correctness proof, we introduce another domain, isomorphic to the concrete domain. Let $\sigma^\diamond \in \Sigma^\diamond \equiv \mathsf{L} \times \mathsf{A}$ be a pair $\langle \ell, \mathsf{a}\rangle$ where a is the action which occur at program label $\ell \in \mathsf{L}$. Consider the set$\Sigma^{\star\diamond}$ which contains all the possible sequence of $\sigma^\diamond$ that can occur during a finite computation, and the lattice $\langle \wp(\Sigma^{\star\diamond}), \subseteq, \emptyset, \Sigma^{\star\diamond}, \cap, \cup \rangle$. We can relate $\wp(\Sigma^\star)$ and $\wp(\Sigma^{\star\diamond})$ by an abstraction $\alpha^\diamond \in \wp(\Sigma^\star) \to \wp(\Sigma^{\star\diamond})$, and a concretization $\gamma^\diamond \in \wp(\Sigma^{\star\diamond}) \to \wp(\Sigma^\star)$ function.

Let $\mathsf{X} = \{\pi_0, \dots, \pi_n\} \in \wp(\Sigma^\star)$ be a set of partial trace and let $\mathsf{Y} = \{\pi_0^\diamond, \dots, \pi_n^\diamond\} \in \wp(\Sigma^{\star\diamond})$ be a set of sequences of $\sigma^\diamond$.

$$\begin{aligned}\alpha^\diamond(\mathsf{X}) \equiv& \{\langle \ell_0, \mathsf{a}_0\rangle \to \dots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \sigma_0 \xrightarrow{\ell_0 \mathsf{a}_0} \dots \xrightarrow{\ell_m \mathsf{a}_m} \sigma_{m+1} \in \mathsf{X}\}\\ \gamma^\diamond(\mathsf{Y}) \equiv& \{\pi \in \wp(\Sigma^\star) \mid \alpha^\diamond(\{\pi\}) \subseteq \mathsf{Y}\}\end{aligned}$$

Notice that $\wp(\Sigma^\star) \underset{\alpha^\diamond}{\overset{\gamma^\diamond}{\leftrightarrows}} \wp(\Sigma^{\star\diamond})$ is an isomorphism: in fact it's simple to prove that $\gamma^\diamond \circ \alpha^\diamond = \alpha^\diamond \circ \gamma^\diamond = id$, where id is the identity function (proof in Appendix A).

Now we define the relation between $\wp(\Sigma^{\star\diamond})$ and $\wp(\Sigma^{\star\sharp})$ by $\alpha^\sharp$ and $\gamma^\sharp$. $\alpha^\sharp : \wp(\Sigma^{\star\diamond}) \to \wp(\Sigma^{\star\sharp})$ is defined by $\alpha^\sharp(\mathsf{X}) = \sqcup^\sharp\{\theta(\pi^\diamond) \mid \pi^\diamond \in \mathsf{X}\}$, where $\theta : \Sigma^{\star\diamond} \to \wp(\Sigma^{\star\sharp})$ is defined as follows.

$$\begin{aligned}\theta(\mathsf{X}) =& \{\langle \ell, \phi \rangle \mid \forall \pi \in \mathsf{X}. \forall \pi' = \langle \ell_0, \mathsf{a}_0\rangle \to \langle \ell_m, \mathsf{a}_m\rangle \preceq^\diamond \pi :\\ & m \geq 0 \wedge \ell = \ell_m \wedge \phi = \mathsf{f}_0 \wedge \dots \wedge \mathsf{f}_n\}\end{aligned}$$

such that:

1. $(\forall \langle \ell, \mathsf{v} := \mathsf{exp}\rangle \in \pi' : \forall \langle \ell', \mathsf{v} := \mathsf{exp}'\rangle \in \pi'. \ell' \leq \ell). \exists \mathsf{f}_i = \bar{\mathsf{y}} \to \bar{\mathsf{v}} : \bar{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{exp}]\!]$
2. $\forall((\langle \ell_i, \mathsf{b}\rangle \to \dots \to \langle \ell_j, \mathsf{endif}\rangle) \vee (\langle \ell_i, \mathsf{not\ b}\rangle \to \dots \to \langle \ell_j, \mathsf{endif}\rangle)) \preceq^\diamond \pi^\diamond$ which represents an if statement and $\forall \langle \ell_k, \mathsf{v} := \mathsf{exp_k}\rangle : i < k < j$ exists $\mathsf{f}_h = \bar{\mathsf{y}} \to \bar{\mathsf{v}}$ such that $\bar{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{b}]\!]$.
3. $\forall((\langle \ell_i, \mathsf{b}\rangle \to \dots \to \langle \ell_j, \mathsf{done}\rangle) \vee (\langle \ell_i, \mathsf{not\ b}\rangle \to \dots \to \langle \ell_j, \mathsf{done}\rangle)) \preceq^\diamond \pi^\diamond$ which represents an while statement and $\forall \langle \ell_k, \mathsf{v} := \mathsf{exp}_k\rangle : i < k < j$ exists $\mathsf{f}_h = \bar{\mathsf{y}} \to \bar{\mathsf{v}}$ such that $\bar{\mathsf{y}} \in \overline{\mathsf{V}}[\![\mathsf{b}]\!]$.

Basically, the function θ transform each action (or sequence of actions) in one or more propositional formulae. The easiest case (1) applies when the action is an assignment statement ($\mathsf{v} := \mathsf{exp}$): we simply obtain the corresponding formula as defined in the transition semantics T. Instead, for if statements (case 2), we track all the assignment actions that are between if and endif. while statements are threated in a similar way (case 3). Notice that $\langle \ell_i, \mathsf{b}\rangle \to \dots \to \langle \ell_j, \mathsf{endif}\rangle$ (or $\langle \ell_i, \mathsf{not\ b}\rangle \to \dots \to \langle \ell_j, \mathsf{endif}\rangle$) represents an if statement if and only if $\forall(\langle \ell_p, \mathsf{b}\rangle \vee \langle \ell_p, \mathsf{not\ b}\rangle) : i < p < j. \exists(\langle \ell_q, \mathsf{endif}\rangle \vee \langle \ell_q, \mathsf{done}\rangle) : p < q < j$ and $\forall(\langle \ell_q, \mathsf{endif}\rangle \vee \langle \ell_q, \mathsf{done}\rangle) : i < q < j. \exists(\langle \ell_p, \mathsf{b}\rangle \vee \langle \ell_p, \mathsf{not\ b}\rangle) : i < p < q$. Similarly for while statement.

Informally, the pair if and endif (while and done) is an if (while) statement if and only if between these two actions, there are only assignments or other pairs if-endif or while-done which correspond to nested if and while statements. To better understand, consider the sequence $\cdots \langle \ell_0, \mathsf{b_0} \rangle \rightarrow \langle \ell_1, \mathsf{b_1} \rangle \rightarrow \langle \ell_2, \mathsf{v} := \mathsf{exp} \rangle \rightarrow \langle \ell_3, \mathsf{endif} \rangle \cdots$: the pair $\langle \ell_0, \mathsf{b_0} \rangle$ and $\langle \ell_3, \mathsf{endif} \rangle$ is not an if statement because between these two actions there is $\langle \ell_1, \mathsf{b_1} \rangle$, which does not represent an assignment action neither an if statement.

On the other hand, the concretization function $\gamma^\sharp : \wp(\Sigma^{\star\sharp}) \rightarrow \wp(\Sigma^{\star\diamond})$ is defined as follows. Let $\mathsf{Y} \in \wp(\Sigma^{\star\sharp})$:

$$\gamma^\sharp(\mathsf{Y}) = \{\pi^\diamond \in \Sigma^{\star\diamond} \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \mathsf{Y} \wedge l(\pi^{\diamond\dashv}) \in \ell(\mathsf{Y}^\dashv)\}$$

It's simple to prove that $\gamma^\sharp$ and $\alpha^\sharp$ are monotone, $\gamma^\sharp \circ \alpha^\sharp$ is extensive, $\alpha^\sharp \circ \gamma^\sharp$ is equivalent to the identity and that $\wp(\Sigma^{\star\diamond}) \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\longleftrightarrow\!\!\!\!\rightarrow}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion (Proof in Appendix A).

Finally, we can express the relation between $\wp(\Sigma^\star)$ and $\wp(\Sigma^{\star\sharp})$ by the composition of above functions, $\alpha = \alpha^\sharp \circ \alpha^\diamond$ and $\gamma = \gamma^\diamond \circ \gamma^\sharp$.
By property of function composition we can assert that $\wp(\Sigma^\star) \underset{\alpha}{\overset{\gamma}{\longleftrightarrow\!\!\!\!\rightarrow}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion.

4.4 Properties

As described in Chapter 2, the aim of information flow analysis is verify the confidentiality and the integrity of the information in computer programs. Both properties are implemented in the analysis and in the next two subsection we show how they are verified.

An information flow analysis can be carried out by considering different attacker abilities. In this context we consider two different scenarios: when the attacker can read public variables only at the beginning and at the end of the computation, and when the attacker can read public variables after each step of the computation. Note that the attacker, in both cases, knows the source code of the program.

Both the properties and the types of attacker are checked through the definition of, and the satisfiability of the propositional formulae (Pos) with respect to the truth-assignment function. Let $\overline{\Upsilon_\mathcal{P}} : \overline{\mathsf{V}} \rightarrow \{\mathtt{T}, \mathtt{F}\}$ be a truth-assignment function associated with the program $\mathcal{P}$. The security properties are modeled by the function definition, while the attacker is modeled by the the set of propositional formulae we consider for the satisfiability. For the first case, in which the attacker can read public variables only at the beginning and at the end of the computation, the set of states to consider involves only the terminal states of each sequence ($\{\mathsf{S} \in \Sigma^{\star\sharp} \mid \overline{\Upsilon_\mathcal{P}} \models r(\mathsf{S}^\dashv)\}$). Whereas in the second case, when the attacker can read public variables at each step of the computation, the set of states to consider involves all the propositional formulae in the sequence ($\{\mathsf{S} \in \Sigma^{\star\sharp} \mid \forall \sigma^\sharp \in \mathsf{S} : \overline{\Upsilon_\mathcal{P}} \models r(\sigma^\sharp)\}$).

4.4.1 Confidentiality

Confidentiality refers to limiting information access and disclosure to authorized users. For example, we require when we buy something online that our private data, e.g. credit card number, are sent to the merchant without third persons can read them during the transmission.

Let$\Upsilon_{\mathcal{P}} : \mathsf{V} \rightarrow \{\mathsf{L}, \mathsf{H}\}$ be a function which assign to each variable of program $\mathcal{P}$ a security class: public (L) or private (H). We say that program $\mathcal{P}$ respects the confidentiality property, if and only if it does not contain any information leakage with respect to the functionΥ_{P}, i.e. there is no information that moves from private to public variables. To verify this property, we define the corresponding truth-assignment function $\overline{\Upsilon_{\mathcal{P}}}$ as follows.

$$\overline{\Upsilon_{\mathcal{P}}}(\bar{\mathsf{x}}) = \begin{cases} \mathtt{T} & \text{if} \Upsilon_{\mathcal{P}}(\mathsf{x}) = \mathsf{H} \\ \mathtt{F} & \text{if} \Upsilon_{\mathcal{P}}(\mathsf{x}) = \mathsf{L} \end{cases}$$

4.4.2 Integrity

Typically, information security systems provide message integrity in addition to data confidentiality. By integrity we indicate that data cannot be modified undetectably. More precisely, unauthorized people can't modified a message when it is moving.

Consider$\Upsilon_{\mathcal{P}} : \mathsf{V} \rightarrow \{\mathsf{L}, \mathsf{H}\}$, a function which assign to each variable of program $\mathcal{P}$ a security class. The integrity property is verified if and only if public variables does not modify private variables, i.e. there is no information leakage from public variables to private variables. The corresponding truth-assignment function $\overline{\Upsilon_{\mathcal{P}}}$, to check this property, is defined as follows.

$$\overline{\Upsilon_{\mathcal{P}}}(\bar{\mathsf{x}}) = \begin{cases} \mathtt{T} & \text{if} \Upsilon_{\mathcal{P}}(\mathsf{x}) = \mathsf{L} \\ \mathtt{F} & \text{if} \Upsilon_{\mathcal{P}}(\mathsf{x}) = \mathsf{H} \end{cases}$$

Notice that is exactly the opposite of the truth-assignment function for the confidentiality property.

4.5 Complexity of the Analysis

The complexity of variables dependency analysis showed above is strictly correlated to the complexity of propositional formulae. Logical domains, in literature, are widely treated and generally, the logical equivalence of two boolean expression is a co-NP-complete problem. However, this complexity issue may not matter much in practice because the size of the set of variables appearing in the program is reasonably small [77]. Hence, on the one hand, work with propositional formulae requires the solving of a co-NP-complete problem, while on the other hand, in many frameworks (included our system), Pos only deal with the variables appearing in the programs, reducing in this way the complexity.

Moreover, it is possible to increase the efficiency of the computation using the *binary decision diagrams* (BDDs) for the implementation of propositional formulae. In fact, the binary decision diagrams are a data structure that is used to represent boolean functions in a compressed way: the idea consists to represent them as a rooted, directed and acyclic graph, which consists of decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each decision node is labeled by a propositional variable and has two child called low and high child, respectively. The edge from a node to a low (high) child represents an assignment of the variable to `F` (`T`). Normally, a BBD is called *ordered* if different variables appear in the same order on all paths from the root. For more information about binary decision diagrams see [4, 104, 3].

4.6 Conclusions

In this Chapter we presented a dependency analysis by abstract interpretation. The main limit of this analysis is that it does not track the variables values, and this might cause many false alarms. In order to get over this limit, we apply the results presented in Chapter 3 and, in Chapter 5, we propose a solution based on the combination of numerical domain. The practical experiments reported in Chapter 6 will allow us to validate our approach.

5 Enhancing the Information Flow Analysis by Combining Domains

5.1 Introduction

Many abstract domains have been designed and implemented to analyze the possible values of numerical variables during the execution of a program. As described in [151], in order to refine the results obtained by the dependency analysis presented in the previous section, we can combine the propositional formulae domain with a numerical domain through a reduce product. The modular construction allows to tune efficiency and accuracy changing the domain which represents the relations among variables value. For instance, if we use the analysis proposed by Karr in [85], we may get a loss of precision with respect to polyhedra analysis, as this domain represents only linear combination of the variables, but we improve the computational cost of the analysis (which becames polynomial).

In this chapter, we provide a brief presentation of the most significant domains with their features and their limitations. Moreover, in Section 5.3 we present the reduced product of dependencies and numerical analysis [151], and in Section 5.4 we draw a novel intuition about how to obtain further refinements of the results through more complex formulae.

5.2 Numerical Domains

5.2.1 Intervals

The interval domain introduced in [47], approximates any subset of the integer number by the least single interval enclosing them. A set $\mathsf{Z} \subseteq \mathbb{Z}$ is approximated by $[\mathsf{a}, \mathsf{b}]$ where $\mathsf{a} = \mathit{min}\ \mathsf{Z}_{i \in \mathbb{N}}$ and $\mathsf{b} = \mathit{max}\ \mathsf{Z}_{i \in \mathbb{N}}$. Obviously, sometimes it is not possible to know about the upper/lower limit of a set of integer so a and b may be $-\infty$ and ∞, respectively. This domain is a lattice, with the ordering $\sqsubseteq$ such that $[\mathsf{a}, \mathsf{b}] \sqsubseteq [\mathsf{c}, \mathsf{d}]$ if and only if the whole interval $[\mathsf{a}, \mathsf{b}]$ is contained in of $[\mathsf{c}, \mathsf{d}]$ and where the top element is the interval $[-\infty, \infty]$ and the bottom element is an empty interval which contains no element. This lattice has infinite height and contains infinite chains and therefore it needs a widening operator. The analyses with this domain are fast and easy to deal with. However, the drawbacks of this domain consists in a lower precision and in some cases the approximation may be very high.

5.2.2 Karr Analysis

Michael Karr, in 1976, presented a practical approach to detect equality relationships between linear combinations of the variables of the program, by considering the problem from the viewpoint of linear algebra. The method developed in [85] is a polynomial-time algorithm to discover affine relationship ($\Sigma_k a_k x_k = c$). This algorithm can be also understood as an abstract domain of affine equalities under the framework of abstract interpretation. The main feature is that it has a finite height, thus no widening is needed to ensure termination of the analysis, which makes it suitable for particular analysis. The affine equality domain is still one of the most efficient relational numerical abstract domains.

Notice that, to permit an efficient analysis which works with rational implementation Chen et al in [31] proposed a specific implementation using floating-point numbers.

5.2.3 Polyhedra

Convex polyhedra are regions of some n-dimensional space that are bounded by a finite set of hyperplanes. A convex polyhedron in $\mathbb{R}^n$ describes a relation between n quantities. In the seminal work [54], P. Cousot and N. Halbwachs applied the theory of abstract interpretation to the static determination of linear equality and inequality relations among program variables and introduced the use of convex polyhedra as a domain of descriptions to solve a number of important data-flow analysis problems.

For $n > 0$ we denote by $\mathbf{v} = (v_0, \dots v_{n-1}) \in \mathbb{R}^n$ an n-tuple (vector) of real numbers; $\mathbf{v} \cdot \mathbf{w}$ denotes the scalar product of vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$; the vector $\mathbf{0} \in \mathbb{R}$ has all components equal to zero. Let $\mathbf{x}$ be a n-tuple of distinct variable. Then $\beta = (\mathbf{a} \cdot \mathbf{x} \bowtie b\ )$ denotes a linear equality and inequality constraint, for each vector $\mathbf{a} \in \mathbb{R}^n$, where $\mathbf{a} \neq \mathbf{0}$, each scalar $b \in \mathbb{R}$ and $\bowtie = \{=, \geq, >\}$. A linear inequality constraint β defines an affine half-space of $\mathbb{R}^n$, denoted by $con(\{\beta\})$.

A set $\mathcal{P} \in \mathbb{R}^n$ is a (convex) polyhedron if and only if $\mathcal{P}$ can be expressed as the intersection of a finite number of affine half-spaces of $\mathbb{R}^n$, i.e. as the solution $\mathcal{P} = con(\mathcal{C})$ of a finite set of linear inequality constraints $\mathcal{C}$. The set of all polyhedra on the vector space $\mathbb{R}^n$ is denoted as $\mathsf{P_n}$. Let $\langle \mathsf{P_n}, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap \rangle$ be a lattice of convex polyhedra, where "$\subseteq$" is the set inclusion, the empty set $\emptyset$ and $\mathbb{R}^n$ as the bottom and top elements, respectively; the binary meet operation, returning the greatest polyhedron smaller than or equal to the two arguments, corresponds to the set intersection and "$\uplus$" is the binary join operation and return the least polyhedron greater than or equal to the two arguments, called convex polyhedral hull.

For more details about polyhedra analysis, see [16, 12, 15] for the use of polyhedra domains and relative Galois connectino, and [14, 10, 92, 81] for their implementation.

5.2.4 Octagons

In the 2001, Antoine Miné presented in [106] a new numerical abstract domain for static analysis by abstract interpretation. The author extended a former numerical domain based on Difference-Bound Matrices ([105]) and showed practical algorithms to represent and manipulate invariants of the form ($\pm \mathsf{x} \pm \mathsf{y} \leq \mathsf{c}$), where x and y are program variables and c is a real constant. Such invariants describe sets of point that are special kind of polyhedra called *octagons* because they feature at most eight edges in dimension 2. This new domain works well for real and rational numbers. During the analysis integer variables can be assumed to be real one in order to find approximate but safe invariants.

The set of invariants which the analysis discovers can be seen as special cases of linear inequalities, but it is more efficient with respect to the one used in polyhedron domain. In fact, this domain allows to manipulate the invariants with a $\mathcal{O}(n^2)$ worst case memory cost per abstract state and a $\mathcal{O}(n^3)$ worst case time cost per abstract operation, where n is the number of variables in the program.

5.3 The Reduced Product of Pos and Numerical Domains

We combine the abstract domain $\langle \wp(\Sigma^{\star\sharp}), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ and a numerical domain $\langle \aleph, \sqsubseteq^\aleph, \bot^\aleph, \top^\aleph, \sqcup^\aleph, \sqcap^\aleph \rangle$ through a reduced product operator [50].

Let $\wp(\Sigma^\star) \underset{\gamma_0}{\overset{\alpha_0}{\longleftrightarrows}} \wp(\Sigma^{\star\sharp})$ and $\wp(\Sigma^\star) \underset{\gamma_1}{\overset{\alpha_1}{\longleftrightarrows}} \aleph$ be two Galois connections and let $\varrho : \wp(\Sigma^{\star\sharp}) \times \aleph \to \wp(\Sigma^{\star\sharp}) \times \aleph$ be a reduce operator defined as follows: let $\mathsf{X} \in \wp(\Sigma^{\star\sharp})$ be a set of partial traces and let $\mathfrak{N} \in \aleph$ be an element of a numerical domain, i.e, a set of intervals, an octagon or a polyhedron. Notice that whatever domain you choose, $\mathfrak{N}$ can be seen as a set of relations among variables' values.

$$\varrho(\langle \mathsf{X}, \mathfrak{N} \rangle) = \langle \mathsf{X}', \mathfrak{N} \rangle$$

such that

$$\begin{aligned} \mathsf{X}' = &\{ {\sigma^\sharp}_{new} \mid \forall \sigma^\sharp \in \mathsf{X}. l({\sigma^\sharp}_{new}) = l(\sigma^\sharp) \wedge \\ &\wedge r({\sigma^\sharp}_{new}) = (r(\sigma^\sharp) \ominus \{ \bar{\mathsf{x}} \to \bar{\mathsf{y}} \mid \mathsf{y} = \mathsf{z} \in \mathfrak{N}, \mathsf{z} \in \overline{\mathbb{V}} \cup \mathbb{Z} \wedge \mathsf{z} \neq \mathsf{x} \}) \} \end{aligned}$$

The reduce operator is aimed at excluding pointless dependencies for all variables which have the same value during the execution, without loosing purposeful relations, using the condition "$\mathsf{x} \neq \mathsf{z}$". Basically, the reduce operator removes from propositional formulae, contained in X, the implications which have at the right side

Figure 5.1 Reduce product example

```
$foo()\{$
    $^0n = 0;\ ^1x = 1;\ ^2i = 0;\ ^3y = x - 1;\ ^4sum = p;$
    $while(^5i \leq k)\ do$
        $if(^6n\%2 == 0)\ then$
            $^7sum = y + p;\ ^8n = n + 1;$
        $else$
            $^9sum = x + (p - 1);\ ^{10}n = n + 3;$
        $^{11}endif$
        $^{12}i = i + 1;$
    $^{13}done$
$\}^{14}$
```

a variable that has a constant value. In fact if the variable has a constant value, it cannot depend on other variables.

Then, the reduced product $\mathsf{D}^\natural$ is defined as follows:

$$\mathsf{D}^\natural = \{\varrho(\langle \mathsf{X}, \mathfrak{N}\rangle) \mid \mathsf{X} \in \wp(\Sigma^{\star\sharp}), \mathfrak{N} \in \aleph\ \}$$

Consider $\mathsf{X}_0, \mathsf{X}_1 \in \wp(\Sigma^{\star\sharp}), \mathfrak{N}_0, \mathfrak{N}_1 \in \aleph$ and $\langle \mathsf{X}_0, \mathfrak{N}_0\rangle, \langle \mathsf{X}_1, \mathfrak{N}_1\rangle \in \mathsf{D}^\natural$. Then $\langle \mathsf{X}_0, \mathfrak{N}_0\rangle \sqsubseteq^\natural \langle \mathsf{X}_1, \mathfrak{N}_1\rangle$ if and only if $\mathsf{X}_0 \sqsubseteq^\sharp \mathsf{X}_1$ and $\mathfrak{N}_0 \subseteq \mathfrak{N}_1$. We define the least upper bound and greatest lower bound operator by $\langle X_0, \mathfrak{N}_0\rangle \sqcup^\natural \langle \mathsf{X}_1, \mathfrak{N}_1\rangle = \langle \mathsf{X}_0 \sqcup^\sharp \mathsf{X}_1, \mathfrak{N}_0 \uplus \mathfrak{N}_1\rangle$ and $\langle \mathsf{X}_0, \mathfrak{N}_0\rangle \sqcap^\natural \langle \mathsf{X}_1, \mathfrak{N}_1\rangle = \langle \mathsf{X}_0 \sqcap^\natural \mathsf{X}_1, \mathfrak{N}_0 \cap \mathfrak{N}_1\rangle$, respectively.
$\langle \mathsf{D}^\natural, \sqsubseteq^\natural, \emptyset, \varrho(\langle \Sigma^{\star\sharp}, \mathbb{R}^n\rangle), \sqcup^\natural, \sqcap^\natural\rangle$ forms a complete lattice.

In order to better understand the improvements yielded by the combination of the two domains consider the following example.

Example 13. *Consider the code in Figure 5.1. For the sake of simplicity, we show a partial representations through propositional formula and polyhedra of the variables dependency. In particular we take in account the labels* 4, 5, 8, 10, 12 *and* 14.

	Polyhedra
4	$n = 0; x - 1 = 0; i = 0; y = 0$
5	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; 3i - n \geq 0;$
8	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; -i + k \geq 0; 3i - n \geq 0;$
10	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; -i + k \geq 0; 3i - n \geq 0;$
12	$-p + sum = 0; y = 0; x - 1 = 0; -i + n - 1 \geq 0; -i + k \geq 0;$ $i \geq 0; 3i - n + 3 \geq 0;$
14	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; -i + k - 1 \geq 0; 3i - n \geq 0;$

	Propositional formula
4	$x \rightarrow y$
5	$p \rightarrow sum$
8	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (y \rightarrow sum)$
10	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (x \rightarrow sum)$
12	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (x \rightarrow sum) \wedge (y \rightarrow sum)$
14	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (x \rightarrow sum) \wedge (y \rightarrow sum) \wedge (n \rightarrow sum) \wedge$ $(i \rightarrow sum) \wedge (i \rightarrow n) \wedge (k \rightarrow sum) \wedge (k \rightarrow n)$

When we apply the reduce operator defined above we obtain the following propositional formulae:

4	$\mathtt{T}$
5	$p \rightarrow sum$
8	$p \rightarrow sum$
10	$p \rightarrow sum$
12	$p \rightarrow sum$
14	$(p \rightarrow sum) \wedge (i \rightarrow n) \wedge (k \rightarrow n)$

By using the reduce operator we simplified the propositional formulae, removing some implication which could in fact generate false alarms when using the direct product of the domains instead of the reduced product.

5.3.1 Efficiency and Accuracy: Complexity of the Analysis

In order to evaluate the efficiency of our analysis, we have to consider its two main components: Pos formulae and polyhedra. Pos complexity has already been considered in Section 4.5, while for polyhedra analysis, the complexity is well and completely treated in many works, e.g. [14], and heavily depends on its implementation. For example many implementations, e.g. Polylib and New Polka, use matrices of coefficients, that cannot grow dynamically, and the worst case space complexity of the methods employed is exponential. In PPL library, instead, all data structures are fully dynamic and automatically expanded (in amortized constant time) ensuring the best use of available memory. Comparing the efficiency of polyhedra libraries is not a simple task, because the pay-off depends on the targeted applications: in [14] the authors presented many test results about it.

The complexity of reduced product, and more precisely of reduction operator ϱ, is strictly connected with the complexity of the operations on the domains we combine.

5.4 Further Refinements

By the combination of the domain introduced in Chapter 4 with numerical domains we can refine the results and remove some false alarms. In some cases, unfortunately, this technique is useless. For example, consider the following program.

$$\mathsf{l = 0;\ if(l == 1)\ then\ l = h;}$$

The analysis track the information flow from h to l of the assignment, but it does not detect that this assignment is true if and only if the condition $(\mathsf{l} == 1)$ is satisfied. In this program the condition will never be satisfied so the information flow from private to public will never be happened.

The idea consists in bind the propositional formulae to a set of condition. The set of condition has already been defined in 4.2.1 and it is represented by B. Consider

$$\mathsf{x = 0;\ if(x == 0)\ then\ y = 0;\ else\ z = 1;}$$

by these new formulae we can obtain at the end of the analysis the following relation: $((\mathsf{x} == 0) \Rightarrow (\bar{\mathsf{x}} \rightarrow \bar{\mathsf{y}})) \wedge ((\mathsf{x}! = 0) \Rightarrow (\bar{\mathsf{x}} \rightarrow \bar{\mathsf{z}}))$. In this way, by the combination with numerical domain through reduced product, we can refine our result and say that only relation we have consist in $\bar{\mathsf{x}} \rightarrow \bar{\mathsf{y}}$. We denote the set of these new formulae by $\mathsf{Pos}^\sharp$.

More formally, by abuse of notation, we define the abstract state definition as $\sigma^\sharp \in \Sigma^\sharp \overset{\text{def}}{=} \mathsf{L} \times \wp(\mathsf{B}) \times \mathsf{Pos}^\sharp$, where L is the set of label, $\wp(\mathsf{B})$ is the powerset of condition, implemented by an ordered list, and represents the condition in which the next command will be executed and $\mathsf{Pos}^\sharp$ is the set of new formulae. The empty list of condition will be denoted by $[\varepsilon]$.
We define the new transitional semantics in Figure 5.2.

The boolean expressions before the implications indicate under which condition the relations are verified. It is easily to extend the definition of $\wp(\Sigma^{\star\sharp})$ given in Section 4.3 to this context. Hence we can consider the lattice $\langle \wp(\Sigma^{\star\sharp}), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ as an abstract domain. Moreover, in the same way, we can assert that $\wp(\Sigma^\star) \underset{\gamma}{\overset{\alpha}{\longleftrightarrow}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion between concrete and abstract domain.

5.4.1 Subformulae Elimination

Given the new abstract domain which uses more complex propositional formulae defined above, $\langle \wp(\Sigma^{\star\sharp}), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$, and a numerical domain $\langle \aleph, \sqsubseteq^\aleph, \perp^\aleph, \top^\aleph, \sqcup^\aleph, \sqcap^\aleph \rangle$, we can define a new reduce operator, denoted by $\varrho^\sharp$, which combines the two domains and increases the precision of results. Let $\wp(\Sigma^\star) \underset{\gamma_0}{\overset{\alpha_0}{\longleftrightarrow}} \wp(\Sigma^{\star\sharp})$ and $\wp(\Sigma^\star) \underset{\gamma_1}{\overset{\alpha_1}{\longleftrightarrow}} \aleph$ be two Galois connection and let $\varrho^\sharp : \wp(\Sigma^{\star\sharp}) \times \aleph \rightarrow \wp(\Sigma^{\star\sharp}) \times \aleph$ be a reduction operator. We can observe that, since the numerical analysis tracks the relation among values of variables, we can consider an element of $\aleph$ as a set of boolean conditions.

Then, we can use a new function *simplify* : $\mathsf{Pos}^\sharp \times \aleph \rightarrow \mathsf{Pos}^\sharp$ that removes from the propositional formula the sub-formulae that do not respect the conditions in $\aleph$. More formally,

$$\mathit{simplify}(\phi^\sharp, \mathsf{B}) = \{\varphi = [\mathsf{b}_0, \cdots, \mathsf{b}_n] \Rightarrow \phi \mid \varphi \in \mathit{Sub}_{\phi^\sharp} \wedge \mathsf{B} \sqsubseteq^\aleph [\mathsf{b}_0, \cdots, \mathsf{b}_n]\}$$

Let $\mathsf{X} \in \wp(\Sigma^{\star\sharp})$ and $\mathsf{B} \in \aleph$ be a set of partial traces and a set of boolean conditions, respectively. The reduced product is defined as $\varrho^{\sharp}(\langle \mathsf{X}, \mathsf{B} \rangle) = \langle \mathsf{X}', \mathsf{B} \rangle$ where

$$\mathsf{X}' = \{ {\sigma^{\sharp}}_{new} \mid \forall \sigma^{\sharp} = \langle \ell, [\mathsf{b}_0 \cdots \mathsf{b}_\mathsf{n}], \phi^{\sharp} \rangle \in \mathsf{X}. {\sigma^{\sharp}}_{new} = \langle \ell, [\mathsf{b}_0 \cdots \mathsf{b}_\mathsf{n}], \mathit{simplify}(\phi^{\sharp}, \mathsf{B}) \rangle \}$$

The product $\mathsf{D}^{\natural}_{new}$ is defined as $\mathsf{D}^{\natural}_{new} = \{ \varrho(\langle \mathsf{X}, \mathsf{B} \rangle) \mid \mathsf{X} \in \wp(\Sigma^{\star\sharp}) \wedge \mathsf{B} \in \aleph \}$ As presented in Section 5.3, we can define a new equivalence relation ($\sqsubseteq^{\natural}_{new}$), a least upper bound ($\sqcup^{\natural}_{new}$) and a greatest lower bound ($\sqcap^{\natural}_{new}$), a bottom ($\langle \emptyset, \perp^{\aleph} \rangle$) and a top ($\langle \Sigma^{\sharp}, \top^{\aleph} \rangle$). In this way $\langle \mathsf{D}^{\natural}_{new}, \sqsubseteq^{\natural}_{new}, \langle \emptyset, \perp^{\aleph} \rangle, \langle \Sigma^{\sharp}, \top^{\aleph} \rangle, \sqcup^{\natural}_{new}, \sqcap^{\natural}_{new} \rangle$ is a complete lattice.

The new product over-approximates, like the previous one, the reduce product: it has the advantage of excluding pointless dependencies, providing more precise results through an more efficient computation.

5.5 Conclusions

In this chapter we presented the combination of the dependency analysis (described in Chapter 4) and numerical analyses, through reduced product. As introduced in Chapter 3, we refine the results, obtained from the first analysis, adding the tracking of relations among the value of variables. Moreover, in Section 5.4, we proposed some preliminary results about further refinement using more complex formulae.

Figure 5.2 Definition of transitional semantics for complex formulae

$$\mathsf{T}^\sharp[\![{}^\ell skip]\!] = \{\langle \ell, [\overline{\mathsf{b}}], \phi^\sharp\rangle \rightarrow \langle f[\![{}^\ell skip]\!], [\overline{\mathsf{b}}], \phi^\sharp\rangle\}$$
$$\mathsf{T}^\sharp[\![{}^\ell x := E]\!] = \{\langle \ell, [\overline{\mathsf{b}}], \phi^\sharp\rangle \rightarrow \langle f[\![{}^\ell x := E]\!], [\overline{\mathsf{b}}], \phi^\sharp_0\rangle\}$$
$$\mathsf{T}^\sharp[\![C_0; C_1]\!] = \mathsf{T}^\sharp[\![C_0]\!] \cup \mathsf{T}^\sharp[\![C_1]\!]$$
$$\mathsf{T}^\sharp[\![if\ {}^\ell b'\ then\ C_0\ else\ C_1\ {}^{\ell'}endif]\!] = \mathsf{T}^\sharp[\![C_0]\!] \cup \mathsf{T}^\sharp[\![C_1]\!] \cup$$
$$\{\langle \ell, [\overline{\mathsf{b}}], \phi^\sharp\rangle \rightarrow \langle in[\![C_0]\!], [\overline{\mathsf{b}}, b'], \phi^\sharp\rangle\} \cup \{\langle\ \ell, [\overline{\mathsf{b}}], \phi^\sharp\rangle \rightarrow \langle in[\![C_1]\!], [\overline{\mathsf{b}}, \neg b'], \phi^\sharp\rangle\} \cup$$
$$\{\langle \ell', [\overline{\mathsf{b}}, b'], \phi^\sharp\rangle \rightarrow \langle f[\![if\ {}^\ell b'\ then\ C_0\ else\ C_1\ {}^{\ell'}endif]\!], [\overline{\mathsf{b}}], \phi^\sharp_1\rangle\}$$
$$\{\langle \ell', [\overline{\mathsf{b}}, \neg b'], \phi^\sharp\rangle \rightarrow \langle f[\![if\ {}^\ell b'\ then\ C_0\ else\ C_1\ {}^{\ell'}endif]\!], [\overline{\mathsf{b}}], \phi^\sharp_2\rangle\}$$
$$\mathsf{T}^\sharp[\![while\ {}^\ell b'\ do\ C\ {}^{\ell'}done]\!] = \mathsf{T}^\sharp[\![C]\!] \cup \{\langle\ \ell, [\overline{\mathsf{b}}], \phi^\sharp\rangle \rightarrow \langle in[\![C]\!], [\overline{\mathsf{b}}, b'], \phi^\sharp\rangle\} \cup$$
$$\{\langle \ell, [\overline{\mathsf{b}}], \phi^\sharp\rangle \rightarrow \langle f[\![while\ {}^\ell b'\ do\ C\ {}^{\ell'}done]\!], [\overline{\mathsf{b}}], \phi^\sharp\rangle\} \cup$$
$$\{\langle \ell', [\overline{\mathsf{b}}, b'], \phi^\sharp\rangle \rightarrow \langle f[\![while\ {}^\ell b'\ do\ C\ {}^{\ell'}done]\!], [\overline{\mathsf{b}}], \phi^\sharp_3\rangle\}$$

where:

$$\phi^\sharp_0 = \bigwedge\{([\overline{\mathsf{b}}] \Rightarrow (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{x}})) \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![E]\!]\}$$
$$\bigwedge\{([\overline{\mathsf{b}}, \overline{\mathsf{b_0}}, \overline{\mathsf{b_1}}] \Rightarrow (\overline{\mathsf{z}} \rightarrow \overline{\mathsf{w}})) \mid ([\overline{\mathsf{b_0}}] \Rightarrow (\overline{\mathsf{z}} \rightarrow \overline{\mathsf{x}})), ([\overline{\mathsf{b_1}}] \Rightarrow (\overline{\mathsf{x}} \rightarrow \overline{\mathsf{w}})) \in \phi^\sharp\}$$

$$\wedge\ (\phi^\sharp \ominus \bigwedge\{([\overline{\mathsf{b}}] \Rightarrow (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{x}})) \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}} \wedge \overline{\mathsf{x}} \notin \overline{\mathsf{V}}[\![E]\!]\})$$
$$\phi^\sharp_1 = \bigwedge\{([\overline{\mathsf{b}}, b'] \Rightarrow (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{x}})) \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![b']\!] \wedge \overline{\mathsf{x}} \in BV(C_0) \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\} \wedge \phi^\sharp$$
$$\phi^\sharp_2 = \bigwedge\{([\overline{\mathsf{b}}, \neg b'] \Rightarrow (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{x}})) \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![\neg b']\!] \wedge \overline{\mathsf{x}} \in BV(C_1) \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\} \wedge \phi^\sharp$$
$$\phi^\sharp_3 = \bigwedge\{([\overline{\mathsf{b}}, b'] \Rightarrow (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{x}})) \mid \overline{\mathsf{y}} \in \overline{\mathsf{V}}[\![b']\!] \wedge \overline{\mathsf{x}} \in BV(C) \wedge \overline{\mathsf{y}} \neq \overline{\mathsf{x}}\} \wedge \phi^\sharp$$

6 Experimental Results: SAILS

6.1 Introduction

Language-based information flow security has been longly studied during the last decades [148, 140]. Proving that a program enforces noninterference has been the goal of several static analyses [66, 68]. Nevertheless, despite this deep and extensive work, its practical applications have been relatively poor. Usually these approaches work on an ad-hoc programming language [9], and they do not support mainstream languages. This means that one should completely rewrite a program in order to apply them to some existing code.

Generally, works on information flow fall into two categories: dynamic, instrumentation based approaches such as tainting, and static, language-based approaches such as type systems. The disadvantage of the dynamic approaches is that they typically incur significant run-time overhead [33, 91]. The disadvantage of the static approaches is that they typically require some changes to the language and the runtime environment, as well as non-trivial type annotations [126], making the adoption of these approaches difficult in practice.

We refer to a new generic static analyzer (Sample) based on abstract interpretation. The overall idea of this analyzer is to split and combine the abstraction of the heap and the approximation of other semantic information, e.g., string [42] and type [65] abstractions. In this chapter, we present Sails (Static Analysis of Information Leakage with Sample), an extension of Sample[1] to information leakage analysis. We slightly modify the theoretical approach to information flow analysis presented in Chapter 4, in order to analyse object oriented programs using different heap abstractions (e.g., shape analysis [130, 67]), and some of the most powerful numerical abstract domains [82]. Unlike other works, our tool provides an information flow analysis without any changes to the language, since it tracks information flows between variables and heap locations over programs written in mainstream object-oriented languages like Java and Scala. We tested Sails over a set of web applications established as security and performance benchmarks. The experimental results show that the analysis is fast and effective in most of the code we analyzed.

[1]`http://www.pm.inf.ethz.ch/research/semper/Sample`

The results presented in this chapter are published in [152].

6.2 Presentation of Sample

6.2.1 Scope of the Analyzer

Sample (Static Analyzer of Multiple Programming LanguagEs) is a novel generic analyzer based on the abstract interpretation theory. Relying on compositional analyses [44], Sample can be plugged with different heap abstractions, approximations of other semantic information (e.g., numeric domains or information flow), properties of interest, and languages. Several heap analyses [67, 63], semantic [42, 65] and numerical [82] domains have been already plugged. The analyzer works on an intermediate language called Simple. Up to now, Sample supports the compilation of Scala and Java bytecode [131] to Simple.

6.2.2 Architecture

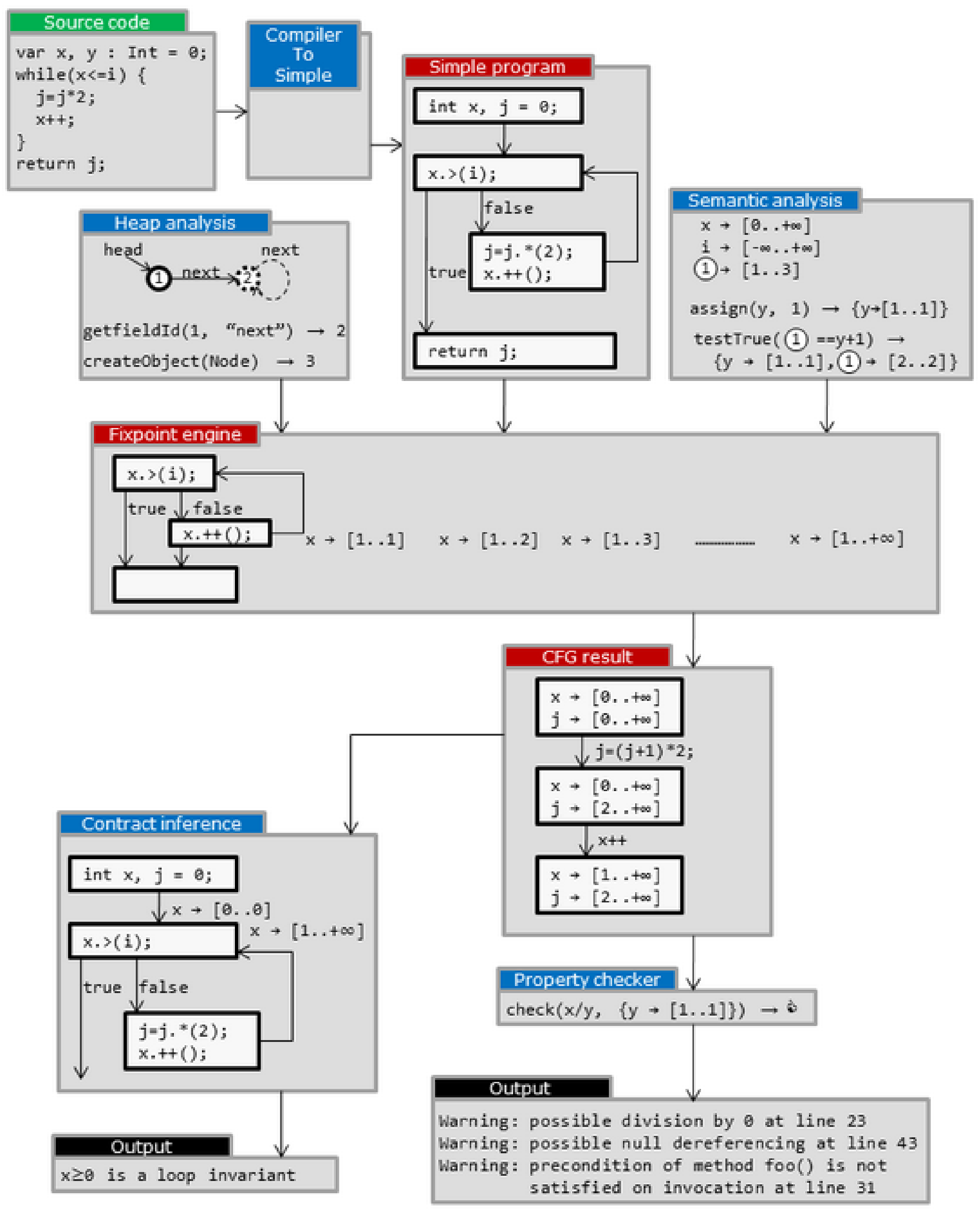
Picture 6.1 depicts the overall structure of Sample. Source code programs are compiled to Simple. A fixpoint engine receives a heap analysis, a semantic domain, and a program, and it produces an abstract result over a control flow graph for each method. This result is passed to a property checker that produces some output (e.g., warnings) to the user. The integration of an analysis in Sample allows one to take advantage of all aspects not strictly related to the analysis but that can improve its final precision (e.g., heap or numerical abstractions). For instance, Sample is interfaced with the Apron library [82] and contains an heap analysis based on TVLA [130].

The Simple language, a fixpoint engine that computes an overapproximation of all possible executions, and the representation of the results of the analysis on a control flow graph are the core of Sample. Differently to these components, which can neither be modified nor extended, other parts permits more flexibility. In particular: the compiler can be extended by the support for other languages, the semantic analysis can be extended by new properties, the heap analysis permits the implementation of specific abstractions and it is possible to develop the inference of specific contracts and checks of new properties. Moreover, the analyzer provides warnings (if the analysis is not able to prove that all executions respect a given property) or inferred contracts.

6.3 Integrating the Information Flow Analysis

In this section, we present the main issues we have to deal with in order to combine the information leakage analysis with Sample.

Figure 6.1 The structure of Sample



```
var x, y : Int = 0;
while(x<=i) {
  j=j*2;
  x++;
}
return j;
```

```
int x, j = 0;
x.>(i);
j=j.*(2);
x.++();
return j;
```

```
getfieldId(1, "next") → 2
createObject(Node)  → 3
```

```
x → [0..+∞]
i → [-∞..+∞]
1 → [1..3]
assign(y, 1) → {y→[1..1]}
testTrue(1 ==y+1) →
  {y → [1..1], 1 → [2..2]}
```

```
x → [0..+∞]
j → [0..+∞]

x → [0..+∞]
j → [2..+∞]

x → [1..+∞]
j → [2..+∞]
```

```
check(x/y, {y → [1..1]}) →
```

```
Warning: possible division by 0 at line 23
Warning: possible null dereferencing at line 43
Warning: precondition of method foo() is not
         satisfied on invocation at line 31
```

```
x≥0 is a loop invariant
```

6.3.1 Analysis extension for Object Oriented Languages

First of all, we need to extend our domains and define a new semantics for object oriented languages. Several authors proposed different approaches to the semantics of object oriented languages: through types [29], object calculi [79, 1], Abstract State Machine [141] and denotational semantics [84]. For the sake of simplicity, we based our work on the definition given by Francesco Logozzo in [60] and we tried to minimize the number of difference with the domain presented in Chapter 4.

Sintax

According to [60], an environment, e, is a map from variables to memory address, and a store, s, is a map from addresses to memory elements. In its turn, a memory can be a primitive value of an environment. The object environment is stored in a certain memory location whose address represents the object identity. In such a setting, two distinct variables are aliases for an object if they reference the same object, i.e. the same memory address. Let Var be a set of variables and let $\mathsf{Addr} \subseteq \mathbb{N}$ be a set of address. Then the set of environments is $\mathsf{Env} : \mathsf{Var} \to \mathsf{Addr}$. Let Val be a set of values such that $\mathsf{Env} \subseteq \mathsf{Val}$. Then the set of store is $\mathsf{Store} : \mathsf{Addr} \to \mathsf{Val}$. The new syntax is formally described in table 6.1.
The definitions of the functions in and f are the same we have already presented in Section 4.2.1, with the addition of new command (i.e. the function call). Moreover, the set of actions is updated with the new element ${}^{\ell}fun(\mathsf{v_0}, \cdots, \mathsf{v_n})$.

Semantics

A state $\sigma \equiv \langle \ell, \mathsf{e}, \mathsf{s} \rangle \in \Sigma$ is a triple of label, ℓ, environments, e, and store, s. Observe that we require Env to be included in the possible memory values: this permits us to treat an object as a memory address and access to its environment through the store. The semantics of a program $\mathcal{P}$ is a set of traces that represents the executions of the program starting from a set of initial state and, as defined in Section 4.2.2, it can be expressed it in fixpoint form:

$$F(\mathsf{X}) \equiv \{\sigma \xrightarrow{\mathsf{a}'} \sigma' \in \mathsf{T} \mid \sigma \in \mathsf{I}\} \cup$$
$$\{\sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-2}} \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \mid \sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-2}} \sigma_{n-1} \in \mathsf{X} \wedge \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \in \mathsf{T}\}$$

Where the transition relation, T, is defined in Figure 6.2. Observe that the values of e' and s' are related to the contracts defined, by the user, for the function $\mathsf{fun}(\mathsf{v_0}, \cdots, \mathsf{v_n})$.

In fact, according to Sample implementation, the interprocedural semantics relies on contracts. In this way, we can simplify the semantics and some issues like the dynamic dispatch problem.

Figure 6.2 Definition of transition semantics for object oriented languages

$$\mathsf{T}[\![{}^{\ell}\mathsf{skip}]\!] \equiv \{\langle\, \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{skip}} \langle f[\![{}^{\ell}\mathsf{skip}]\!], \mathsf{e}, \mathsf{s}\rangle\}$$

$$\mathsf{T}[\![{}^{\ell}\mathsf{x} := \mathsf{exp}]\!] \equiv \{\langle\, \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{x:=exp}} \langle f[\![{}^{\ell}\mathsf{x} := \mathsf{exp}]\!], \mathsf{e}[\mathsf{x} \mapsto a_0], \mathsf{s}[a_0 \mapsto \mathrm{E}(\mathsf{exp})]\rangle \mid \mathsf{exp} \notin \mathsf{Addr}\}$$

$$\{\langle \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{x:=exp}} \langle f[\![{}^{\ell}\mathsf{x} := \mathsf{exp}]\!], \mathsf{e}[\mathsf{x} \mapsto \mathrm{E}(\mathsf{exp})], \mathsf{s}\rangle \mid \mathsf{exp} \in \mathsf{Addr}\}$$

$$\mathsf{T}[\![{}^{\ell}\mathsf{if\ b\ then\ c_0\ else\ c_1}\ {}^{\ell'}\mathsf{endif}]\!] \equiv \{\langle\, \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{b}} \langle in[\![\mathsf{c_0}]\!], \mathsf{e}, \mathsf{s}\rangle \mid \mathsf{true} \in \mathrm{B}[\![\mathsf{b}]\!]\mathsf{e}\} \cup$$

$$\{\langle \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{not\ b}} \langle in[\![\mathsf{c_1}]\!], \mathsf{e}, \mathsf{s}\rangle \mid \mathsf{false} \in \mathrm{B}[\![\mathsf{b}]\!]\mathsf{e}\} \cup \mathsf{T}[\![\mathsf{c_0}]\!] \cup \mathsf{T}[\![\mathsf{c_1}]\!] \cup$$

$$\{\langle \ell', \mathsf{e}, \mathsf{s}\rangle \xrightarrow{\ell'} \langle f[\![{}^{\ell}\mathsf{if\ b\ then\ c_0\ else\ c_1}\ {}^{\ell'}\mathsf{endif}]\!], \mathsf{e}, \mathsf{s}\rangle\}$$

$$\mathsf{T}[\![{}^{\ell}\mathsf{c_0}; \mathsf{c_1}]\!] \equiv \mathsf{T}[\![\mathsf{c_0}]\!] \cup \mathsf{T}[\![\mathsf{c_1}]\!] \cup$$

$$\mathsf{T}[\![{}^{\ell}\mathsf{while}\ {}^{\ell}\mathsf{b\ do\ c}\ {}^{\ell'}\mathsf{done}]\!] \equiv \{\langle\, \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{b}} \langle in[\![\mathsf{c}]\!], \mathsf{e}, \mathsf{s}\rangle \mid \mathsf{true} \in \mathrm{B}[\![\mathsf{b}]\!]\mathsf{e}\} \cup$$

$$\{\langle \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{not\ b}} \langle \ell', \mathsf{e}, \mathsf{s}\rangle \mid \mathsf{false} \in \mathrm{B}[\![\mathsf{b}]\!]]\mathsf{e}\} \cup \mathsf{T}[\![\mathsf{c}]\!]$$

$$\mathsf{T}[\![{}^{\ell}\mathsf{fun}(\mathsf{v_0}, \cdots, \mathsf{v_n})]\!] \equiv \{\langle\, \ell, \mathsf{e}, \mathsf{s}\rangle \xrightarrow{{}^{\ell}\mathsf{fun}(\mathsf{v_0}, \cdots, \mathsf{v_n})} \langle f[\![{}^{\ell}\mathsf{fun}(\mathsf{v_0}, \cdots, \mathsf{v_n})]\!], \mathsf{e}', \mathsf{s}'\rangle\}$$

Abstract Domain

Despite the abstract domain presented in 4.3.3, the dependency relations do not involve the variables but they are associated to the set of addresses (Addr). Each information is univocally related to an address (i.e. where it is stored): if two variables, $\mathsf{obj_0}$ and $\mathsf{obj_1}$, point to the same address, $\mathsf{e}(\mathsf{obj_0}) = \mathsf{e}(\mathsf{obj_1})$, then they are the same information.

The new transition relation is defined in Figure 6.3 and permits us to keep the same abstract domain, $\langle \wp(\Sigma^\sharp), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$, and to use, in practice, the same Galois insertion presented in Chapter 4.

6.3.2 Implementation Choices

Representing Propositional Formulae

To work with object oriented languages entailed to introduce some slight modifications on the domain for information leakage analysis described in Chapter 4. We can consider a propositional formula ϕ as a conjunction of subformulae $(\zeta_0 \wedge \ldots \wedge \zeta_n)$. In the implementation, each subformula is an implication between two identifiers (an identifier is a variable abstraction, see next paragraph). Then we represent a subformula as a pair of identifiers and a formula as a set of subformulae. Consider

the command $\mathsf{if}(\mathsf{x} > 0)\ \mathsf{y} = \mathsf{z};$. The formula obtained after the analysis consists in two pairs: $(\bar{\mathsf{y}}, \bar{\mathsf{z}})$ and $(\bar{\mathsf{x}}, \bar{\mathsf{y}})$, where by $\bar{\mathsf{u}}$ we denote the identifier of the variable u. The order relation "$\trianglelefteq$" is defined by: let ϕ_0 and ϕ_1 be propositional formulae, $\phi_0 \trianglelefteq \phi_1$ is equivalent to $\phi_0 \subseteq \phi_1$, where "$\subseteq$" is the classical subset relation.
Consequently, in the new abstract domain, the set of propositional variables $\overline{\mathsf{V}}$ consists in the set of identifier Id a single propositional formula is represented by $\wp(\mathsf{Id} \times \mathsf{Id})$ and an abstract state $\sigma^\sharp \in \Sigma^\sharp$ is a conjunction of propositional formulae represented by $\wp(\wp(\mathsf{Id} \times \mathsf{Id}))$.

Heap Abstraction

In Sample heap locations are approximated by abstract heap identifiers. While the identifiers of program variables are fixed and represents exactly one concrete variable, the abstract heap identifiers may represent several concrete heap locations (e.g., if they summarize a potentially unbounded list), and they can be merged and split during the analysis. In particular we have to support (i) assignments on summary heap identifiers, and (ii) renaming of identifiers.

In order to preserve the soundness of Sails, we have to perform weak assignments to summary heap identifiers. Since a summary abstract identifier may represent several concrete heap locations and only one of them would be assigned in one particular execution, we have to take the upper bound between the assigned value, and the old one.

Any heap abstraction requires to rename, summarize or split existing identifiers. This information is passed through a replacement function $rep : \wp(\mathsf{Id}) \rightarrow \wp(\mathsf{Id})$. In TVLA, [130] two abstract nodes represented by identifiers a_1 and a_2 may be merged to a summary node a_3, or a summary abstract node b_1 may be splitted to b_2 and b_3. Our heap analysis will pass $\{a_1, a_2\} \mapsto \{a_3\}$ and $\{b_1\} \mapsto \{b_2, b_3\}$ to Sails in these cases respectively. Given a single replacement $\mathsf{S}_1 \mapsto \mathsf{S}_2$, Sails removes all subformulae dealing with some of the variables in S_1, and for each removed subformula s it inserts a new subformula s' in the resulting state renaming each of the variables in S_1 to with each of the variables in S_2. Formally:

$$
\begin{array}{l}
\mathit{rename} : (\mathsf{Pos} \times (\wp(\mathsf{Id}) \rightarrow \wp(\mathsf{Id}))) \rightarrow \mathsf{Pos} \\
\mathit{rename}(\sigma^\sharp, \mathit{rep}) = \{(\mathsf{i'}_1, \mathsf{i'}_2) : (\mathsf{i}_1, \mathsf{i}_2) \in \sigma^\sharp \& \\
\quad \mathsf{i'}_1 = \begin{cases} \mathsf{i}_1 & \text{if } \nexists \mathsf{R}_1 \in \mathit{dom}(\mathit{rep}) : \mathsf{i}_1 \in \mathsf{R}_1 \\ \mathsf{k}_1 & \text{if } \exists \mathsf{R}_1 \in \mathit{dom}(\mathit{rep}) : \mathsf{i}_1 \in \mathsf{R}_1 \ \&\ \mathsf{k}_1 \in \mathit{rep}(\mathsf{R}_1) \end{cases}, \\
\quad \mathsf{i'}_2 = \begin{cases} \mathsf{i}_2 & \text{if } \nexists \mathsf{R}_2 \in \mathit{dom}(\mathit{rep}) : \mathsf{i}_2 \in \mathsf{R}_2 \\ \mathsf{k}_2 & \text{if } \exists \mathsf{R}_2 \in \mathit{dom}(\mathit{rep}) : \mathsf{i}_2 \in \mathsf{R}_2 \ \&\ \mathsf{k}_2 \in \mathit{rep}(\mathsf{R}_2) \end{cases} \}
\end{array}
$$

Implicit Flow Detection

An implicit information flow occurs when there is an information leakage from a variable in a condition to a variable assigned inside a block dependent on that condition. For instance, in $\mathsf{if}(\mathsf{x} > 0)\ \mathsf{y} = \mathsf{z};$ there is an explicit flow from z to y, and an implicit flow from x to y. To record these relations we relate the variables in the conditions to the variables that have been assigned in the block. When we join two blocks coming from the same condition, we discharge all implicit flows on the abstract state.

On the other hand, Sample programs are represented by control flow graphs (cfg), and therefore we could have conditions that do not join in a well-defined point. For instance, in the cfg of Figure 6.4 is not clear if the condition of block 1 is joined at block 4 or 6. For this reason, Sails does not support all cfgs that can be represented in Sample but only the ones coming from structured programs, i.e., that corresponds to programs with if and while statements and not with arbitrary jumps like goto.

Property

An information flow analysis can be carried out by considering different attacker abilities. We implemented two scenarios: when the attacker can read public variables only at the beginning and at the end of the computation, and when the attacker can read public variables after each step of the computation[2]. Moreover, to each attacker we implemented two security properties: secrecy (i.e., information leakage analysis) and integrity.
The verification of these properties happens after the computation of the analysis and the declaration of private variables (at run time, by a text files writing the variables name or by a graphical user interface selecting the variables in a list).

Numerical Analyses

The information flow analysis is based on the reduced product of a dependency and a numerical analysis. Thanks to the structure of Sample, we can naturally plug Sails with different numerical domains. In particular, Sample supports the Apron library [82]. In this way, we can combine Sails with all numerical domains contained in Apron (namely, Polka, the Parma Polyhedra Library, Octagons, and a deep implementation of Intervals).

In addition, we can apply different heap abstractions to the analysis of a program without changing Sails. For instance, if we are not interested to the heap structure, we can use a less accurate domain that approximates all heap locations with one unique summary node, as in Section 6.3.4. Instead, if we look at a precise abstraction of the heap structure, we can adopt more precise approximations, as illustrated in the next section.

6.3.3 Example

Consider the Java code in Figure 6.5. Class ListWorkers models a list of workers of an enterprise. Each node contains the salary earned by the worker, and some other information (e.g., name and surname of the person). Method updateSalaries is defined as well. It receives a list of employees and a list of managers. These two lists are supposed to be disjoint. First method updateSalaries computes the maximal salary of an employee. Then it traverses the list of managers updating their salary to the maximal salary of employees if manager's salary is smaller than that.

Usually managers would like not to leak information about their salary to employees. This property could be expressed in Sails specifying that we do not want to have a flow of information *from* managers *to* employees. More precisely, we want to prove the absence of information leakage from the content of field salary of any node reachable from managers to any node reachable from employees.

We combine Sails with a heap analysis that approximates all objects created by a program point with a single abstract node[63]. We start the analysis of method updateSalaries with an abstract heap in which lists managers and employees are abstracted with a summary node and they are disjoint. Figure 6.6 depicts the initial state, where n2 and n4 contains the salary values of the ListWorkers n1 and n3, respectively. In the graphic representation we adopt dotted circles to represent summary nodes, rectangles to represent local variables, and edges between nodes to represent what is pointed by fields of objects. Note that the structure of these two lists does not change during the analysis of the program, since method updateSalaries does not modify the heap structure.

Sails infers that, after the first while loop at line 15, there is a flow of information from n2 to maxSalary. This happens because variable it points to n1 before the loop (because of the assignment at line 9), and it iterates following field next (obtaining always the summary node n1) eventually assigning the content of it.salary (that is, node n2) to maxSalary. Therefore, at line 15 we have the propositional formula $\mathsf{n2} \rightarrow \mathsf{maxSalary}$.

Then updateSalaries traverses list managers. For each node, it could potentially assign maxSalary to it.salary. Similarly to what happened in the previous loop, variable it points to n3 before and inside the loop, since field next always points to the summary node n3. Therefore the assignment at line 18 could potentially affects only node n4. For this reason, Sails discovers a flow of information from maxSalary to n4, represented by the propositional formula $\mathsf{maxSalary} \rightarrow \mathsf{n4}$.

At the end of the analysis, Sails soundly computes that $(\mathsf{n2} \rightarrow \mathsf{maxSalary}) \wedge (\mathsf{maxSalary} \rightarrow \mathsf{n4})$. By the transitive property, we know that there could be a flow of information from n2 to n4, that is, from employees to managers. This flow is allowed by our security policy. On the other hand, we also discovered that there is no information leakage from list managers to list employees, since Sails does not

[2]Notice that, as in [151], we assume that the attacker, in both cases, knows the source code of the program.

contain any propositional formula containing this flow. Therefore Sails proves that this program is safe.

Notice that, almost 10 years ago, Sabelfeld and Myers stated: "Noninterference of programs essentially means that a *variable* of confidential (high) input does not cause a variation of public (low) output"[126]. Thanks to the combination between a heap abstraction and an abstract domain tracking information flow, Sails deals directly with the structure of the heap, extending the concept of noninterference from variables to portions of the heap represented by abstract nodes. This opens a new scenario since we can prove that a whole data structure does not interfere with another one, as we have done in this example. As far as we know, Sails is the only tool that performs a noninterference analysis over a heap abstraction, and therefore it can prove properties like "there is no information flow from the nodes reachable from v_1 to the nodes reachable from v_2".

6.3.4 Analysis Results

Sails implements the analysis introduced in Chapter 4, therefore the violations which are detected by it correspond to respond the possible information leakages through either implicit and explicit flows. A well-established way of studying the precision and the efficiency of information flow analyses is the SecuriBench-micro suite [144]. We applied Sails to this test suite; the description and the results of these benchmarks are reported in Table 6.2. Column `fa` reports if the analysis did not produce any false alarm. We combined Sails with a really rough heap abstraction that approximates all concrete heap locations with one abstract node. Sails detected all information leakages in all tests, but in three cases (`Pred1`, `Pred6` and `Pred7`) it produced false alarms. This happens because Sails abstracts away the information produced when testing to true or false boolean conditions in if or while statements. Using more complex formulae defined in Section 5.4, we could avoid this kind of false alarms and obtain better results.

Since these benchmarks cover only problems with explicit flows, we performed further experiments using some Jif [113] case studies. The results are reported in Table 6.3: we discovered all flows without producing any false alarm. These results allow us to conclude that Sails is precise, since in 90% of the cases (28 out of 31 programs) it does not produce any false alarm.

About the performances, the analysis of all case studies takes 1.092 seconds (0.035 sec per method in average) without combining it with a numerical domain. When we combine it with Intervals it takes 3.015 seconds, whereas it takes 6.130 seconds in combination with Polka. All tests are performed by a MacBook Pro, Intel Core 2 Duo 2.53 GHz, 4 GB Memory. Therefore the experimental results underline the efficiency of Sails as well.

The modular construction of this tool permits further refinements using more precise domains. For instance, a plugin to interface TVLA with Sample has been developed recently [67]. A more sophisticated shape analysis that avoids the summarization of nodes with different level of confidentiality may in fact enhance the precision of the Sails analysis.

6.4 Current Limits of the Analyzer

In previous session we present some preliminary results based on SecuriBench-micro. Unfortunately, at the moment we cannot apply Sails over large programs or wide-scale tests like, e.g., the whole SecuriBench. In fact, Sails is not a complete analyzer, but it is an extension of the generic analyzer Sample the latter being still under development.

The main lacks in this version of the Sample analyzer concern two key features: contracts and data structures. Sample interprocedural semantics relies on contracts: generally for each function call developers define pre and post condition and through them the analyzer know the state at the end of the function. But, at this point of the development, Sample does not yet fully support contracts, in particular it does not support contracts dealing with levels of confidentiality. For this reason, Sails can analyze only code without function calls (intraprocedural analysis). Currently, we are working on to define an annotation language to introduce contracts like "x.f is confidential".

The second weakness point of Sample consists in its limits about data structures: the compiler cannot analyze arrays or more complex structures, yet. This limitation restricts the complexity of the programs that might be currently analyzed.

SecuriBench, unlike SecuriBench-micro, is a set of web applications which use some external libraries and professional frameworks (e.g., Struts[3] and Log4j[4]). The complexity of this set of software is one of the main features and makes impossible a simplification of the code without trivializing it. Moreover, SecuriBench treats strings and not integers: then an analysis based on numerical domains becomes pointless. For this purpose, we are planning to combine positive formulae domain with a domain for strings, like [42], to refine the results. In this way, we should able to detect all the cases. For instance, on the simple code in Figure 6.7 we should keep the fact that the value of public does not depend on secret, because the functions computed in both branches give the same results.

6.5 Comparison with other Tools

The approach adopted in Sails is quite different from existing tools that deal with information flow analysis. The most known tool in this field is Jif [9]. It is a security-typed programming language that extends Java with support for information flow and access control, enforced at compile time. Jif is an ad hoc analysis that requires

[3]http://struts.apache.org/

[4]http://logging.apache.org/log4j/

to annotate the code with some type information. Obviously, it is more efficient than Sails, but on the other hand our analysis does not require to annotate the program and take all advantages of compositional analyzers.

Other security-typed languages emerged over the years to prevent insecure information flows. The possibility of regulating the propagation of sensitive information by security type systems in realistic languages came from [18, 110, 120, 137] and their implementations.

"*Despite this rather large (and growing!) body of work on language based information flow security, there has been relatively little adoption of the proposed techniques*" [90]. According to Li and Zdancewic, one of the reasons that limited the application of these systems is that they require to re-write the whole system in the new language. In addition, usually only a small part of the system deals with critical information. Therefore developers choose the programming language that best fits the primary functionality of the system.

Our approach does not require to change the programming language, since it infers the flow of information directly on the original program, and it asks what are the private data that have not to be leaked to the user during the analysis execution.

6.5.1 JIF - Java Information Flow

In [110], A. Myers introduced *JFlow*, an extension to the Java language [72] that adds statically-checked information flow annotations. JFlow provides several new features that make information flow checking more flexible and convenient than in previous models. Moreover it supports many language features that have never been integrated successfully with static information flow control, including objects, subclassing, dynamic type tests, access control and exceptions. JFlow treats static checking of flow annotations as an extended form of type checking. Programs written in this language can be statically checked by the JFlow compiler, which prevents information leaks through storage channels [87]. JFlow is intended to support the writing of secure servers and applets that manipulate sensitive data. The JFlow compiler is structured as a source-to-source translator, so its output is a standard Java program that can be compiled by any Java compiler.

Jif [9] extends Java by adding label that express restrictions on how information may be used (technically, it use JFlow) and it provides static information flow checking via a type system, based on the Decentralized Label Model [111]. The programmer must annotate variables, methods, and class declaration with a label. Jif can also infers labels not explicitly declared, and sets them to be as restrictive as possible. A label specifies who owns data and who can read it. For example, the label $\{\mathsf{Alice}:\}$ means that Alice owns the data and only she can read it, whereas $\{\mathsf{Alice}:\ \mathsf{Bob}\}$ means that Alice owns the data and Bob can read it too. The entities that own and read data are called principals. Principals are defined by the programmer and they could be related to each other by the acts-for relation, i.e. a principal Alice may delegate authority to another principal Bob. The acts-for relation is reflexive and transitive and it can be used to model groups and roles conveniently. Through the labels it is possible to express security policies. For instance, consider int {Alice $\rightarrow$ Bob} x;. In this case the security policy says that the information in x is controlled by the principal Alice, and that Alice permits this information to ben seen by the principal Bob. In addition, Jif supports a declassify operation, which enables the owner of a piece of data to give it a less restrictive label in certain circumstances.

In conclusion, Jif and Sails have different architecture and different target: the first one is an ad-hoc tool that helps the programmer to write secure code, whereas the second one is a component of a general purpose analyzer and provides a tool dedicated to verification of security properties on already written code. If on the one hand we have, in Jif, more complex security properties (and relative options, e.g., declassification) and more efficiency, on the other one the definition and the verification of the properties are strongly static, causing the decreasing the flexibility of the tool.

6.5.2 Julia - Software Verification for Java and Android

Recently, a new tool has been developed: *Julia*[5], a static analyzer for Java or Android programs, written in Java. Its goal consists in to identifier in a fully automatic way bugs without any help on the part of the programmer. It uses the abstract interpretation technique and it is specialized to detect the following bugs:

- dead-code, methods or constructors that are not called;
- incorrect casts, a typical programming error, when the programmer assumes that some data has a different nature than it actually has;
- null-pointer errors, a typical programming error, when the programmer accesses data that is actually missing;
- non-terminating loops or recursion, they are a waste of computing resources or an actual bug;
- bad style, inappropriate class or method name, access of static data from a non-static context, etc.;
- unused fileds, fields that are only read or only written are useless or can be actual bug;
- wrong redefinitions, inappropriate types when a method is redefined, or inappropriate calls to super();
- bad equalities, use of equlas() instead of == or viceversa;

[5] `http://www.juliasoft.com/`

- bad definitions of equals/hashCode, missing hashCode() or equals();
- bad comparison, unsafe aproximated comparisons between float numbers.

The tool analyze only the compiled bytecode and it is possible to use Julia as a command-line tool, run by any standard Java Virtual Machine or it can queried as a web service, run inside a standard Tomcat container through SOAP protocol.

More details about the implementation are not specified. Julia is a software tool produced by researchers from the University of Verona which have decided to offer a service of software analysis only through a web site.

In this case the comparison is not possible. The architecture of this tool are very similar to Sample (hence Sails too) but, the aim is completely different. Indeed, as far as we know, Julia does not implement any information flow analysis yet.

Nowadays smartphones and their applications are very widespread and Julia, as far as we know, is the first static analyzer for android programs. Therefore, the implementation of our dependency analysis in Julia can lead interesting results. Adding information flow analysis permits to verify the security aspects of applications before their use and it may be crucial to protect the emerging data communication devices.

6.6 Conclusions

In this chapter we introduced Sails that applies and implements the information flow analysis on object-oriented programs. Sails is an extension of Sample, therefore it is modular with respect to the heap abstraction, and it can verify noninterference over recursive data structures using simple and efficient heap analyses.

Table 6.1 Syntax for Object Oriented Languages

Variables:			
v	$\in$	Var	
v	::=	x|y| . . .	
Arithmetic Expression:			
aexp	$\in$	Aexp	
aexp	::=	$\mathsf{n} \in \mathbb{N}$	
	|	V	
	|	Aexp $\oplus$ Aexp	where $\oplus = \{+, -, *, /\}$
Addresses:			
addr	$\in$	Addr	
addr	::=	$\mathsf{n} \subseteq \mathbb{N}$	
	|	Addr $\oplus$ n	where $\oplus = \{+, -, *, /\}$
Expression:			
exp	$\in$	Exp $\equiv$ Addr $\cup$ Aexp	
Conditions:			
b	$\in$	B	
b	::=	true | false	
	|	$b_0 \otimes b_1$	where $\otimes = \{\wedge, \vee\}$
	|	$\neg b$	
	|	Exp $\oslash$ Exp	where $\oslash = \{\leq, >, =\}$
Labeled commands:			
ℓ	$\in$	L	
c	$\in$	C	
c	::=	$^{\ell}$skip	
	|	$^{\ell}$v := exp	
	|	if $^{\ell}$b then c_0 else c_1 $^{\ell'}$endif	
	|	$\mathsf{c}_0; \mathsf{c}_1$	
	|	while $^{\ell}$b do c $^{\ell'}$endif	
	|	$^{\ell}$fun$(\mathsf{v}_0, \cdots, \mathsf{v}_n)$	where fun could be a class constructor
Programs:			
p	$\in$	$\mathcal{P}$	
p	::=	c	

Figure 6.3 Definition of abstract transition semantics for object oriented languages

$$
\begin{aligned}
&\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{skip}]\!] = \{\langle \ell,\phi \rangle \rightarrow \langle f[\![{}^{\ell}\mathsf{skip}]\!], \phi\rangle\}\\
&\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{x} := \mathsf{exp}]\!] = \{\langle \ell,\phi \rangle \rightarrow \langle f[\![{}^{\ell}\mathsf{x} := \mathsf{exp}]\!], \phi_0\rangle \mid \mathsf{exp} \notin \mathsf{Addr}\}\cup\\
&\quad \{\langle \ell,\phi \rangle \rightarrow \langle f[\![{}^{\ell}\mathsf{x} := \mathsf{exp}]\!], \phi\rangle \mid \mathsf{exp} \in \mathsf{Addr}\}\\
&\overline{\mathsf{T}}[\![\mathsf{c}_0;\mathsf{c}_1]\!] = \overline{\mathsf{T}}[\![\mathsf{c}_0]\!] \cup \overline{\mathsf{T}}[\![\mathsf{c}_1]\!]\\
&\overline{\mathsf{T}}[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}]\!] = \overline{\mathsf{T}}[\![\mathsf{c}_0]\!] \cup \overline{\mathsf{T}}[\![\mathsf{c}_1]\!]\cup\\
&\quad \{\langle \ell,\phi \rangle \rightarrow \langle in[\![\mathsf{c}_0]\!], \phi\rangle\} \cup \{\langle \ell,\phi \rangle \rightarrow \langle in[\![\mathsf{c}_1]\!], \phi\rangle\}\cup\\
&\quad \{\langle \ell', \phi\rangle \rightarrow \langle f[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}]\!], \phi_1\rangle\}\\
&\quad \{\langle \ell', \phi\rangle \rightarrow \langle f[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}]\!], \phi_2\rangle\}\\
&\overline{\mathsf{T}}[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!] = \overline{\mathsf{T}}[\![\mathsf{c}]\!] \cup\{\langle \ell,\phi \rangle \rightarrow \langle in[\![\mathsf{c}]\!], \phi\rangle\}\cup\\
&\quad \{\langle \ell', \phi\rangle \rightarrow \langle f[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!], \phi_3\rangle\}\\
&\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{fun}(\mathsf{v}_0,\cdots,\mathsf{v}_n)]\!] = \{\langle \ell,\phi \rangle \rightarrow \langle f[\![{}^{\ell}\mathsf{fun}(\mathsf{v}_0,\cdots,\mathsf{v}_n)]\!], \phi_{fun}\rangle\}
\end{aligned}
$$

where, given a contract for the command $\mathsf{fun}(\mathsf{v}_0,\cdots,\mathsf{v}_n)$ (denoted by Cont) which contain the relation created inside the function call,

$$
\begin{aligned}
\phi_0 = &\bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(y)}} \in \overline{\mathsf{V}}[\![\mathsf{exp}]\!] \wedge \overline{\mathsf{e(y)}} \neq \overline{\mathsf{e(x)}}\}\\
&\bigwedge\{\overline{\mathsf{e(z)}} \rightarrow \overline{\mathsf{e(w)}} \mid \overline{\mathsf{e(z)}} \rightarrow \overline{\mathsf{e(x)}}, \overline{\mathsf{e(x)}} \rightarrow \overline{\mathsf{e(w)}} \in \phi\}\\
&\wedge (\phi \ominus \bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(y)}} \in \overline{\mathsf{V}} \wedge \overline{\mathsf{e(x)}} \notin \overline{\mathsf{V}}[\![\mathsf{exp}]\!]\})\\
\phi_1 = &\bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(y)}} \in \overline{\mathsf{V}}[\![B]\!] \wedge \overline{\mathsf{e(x)}} \in BV(C_0) \wedge \overline{\mathsf{e(y)}} \neq \overline{\mathsf{e(x)}}\} \wedge \phi\\
\phi_2 = &\bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(y)}} \in \overline{\mathsf{V}}[\![B]\!] \wedge \overline{\mathsf{e(x)}} \in BV(C_1) \wedge \overline{\mathsf{e(y)}} \neq \overline{\mathsf{e(x)}}\} \wedge \phi\\
\phi_3 = &\bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(y)}} \in \overline{\mathsf{V}}[\![B]\!] \wedge \overline{\mathsf{e(x)}} \in BV(C) \wedge \overline{\mathsf{e(y)}} \neq \overline{\mathsf{e(x)}}\} \wedge \phi\\
\phi_{fun} = &\bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \in \mathsf{Cont}\}\\
&\bigwedge\{\overline{\mathsf{e(z)}} \rightarrow \overline{\mathsf{e(w)}} \mid \overline{\mathsf{e(z)}} \rightarrow \overline{\mathsf{e(x)}}, \overline{\mathsf{e(x)}} \rightarrow \overline{\mathsf{e(w)}} \in \phi \wedge \exists \overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \in \mathsf{Cont}\}\\
&\wedge (\phi \ominus \bigwedge\{\overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(x)}} \mid \overline{\mathsf{e(x)}} \in \overline{\mathsf{V}} \wedge \exists \overline{\mathsf{e(y)}} \rightarrow \overline{\mathsf{e(z)}} \in \mathsf{Cont}.\overline{\mathsf{e(z)}} \neq \overline{\mathsf{e(x)}}\})
\end{aligned}
$$

Figure 6.4 A CFG not supported by Sails

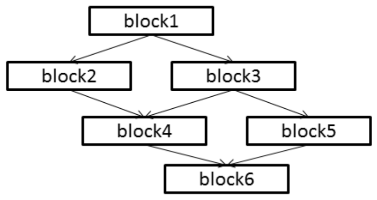


Figure 6.5 A motivating example

```
class ListWorkers {
  int salary;
  ListWorkers next;
   ...
}

public void updateSalaries
(ListWorkers employees, ListWorkers managers){
  int maxSalary = 0;
  ListWorkers it=employees;
  while(it!=null) {
    if(it.salary>maxSalary)
      maxSalary=it.salary;
    it=it.next;
  }
  it=managers;
  while(it!=null) {
    if(it.salary < maxSalary)
      it.salary=maxSalary;
    it=it.next;
  }
}
```

Figure 6.6 The initial state of the heap abstraction

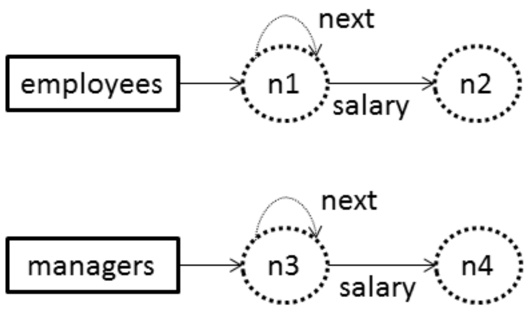


Table 6.2 SecuriBench-micro suite

Name	Description	fa
Aliasing1	Simple aliasing	✓
Aliasing2	Aliasing false positive	✓
Basic1	Very simple XSS	✓
Basic2	XSS combined with a conditional	✓
Basic3	Simple derived integer test	✓
Basic5	Test of derived integer	✓
Basic6	Complex test of derived integer	✓
Basic8	Test of complex conditionals	✓
Basic9	Chains of value assignments	✓
Basic10	Chains of value assignments	✓
Basic11	A simple false positive	✓
Basic12	A simple conditional	✓
Basic18	Protect agains simple loop unrolling	✓
Basic28	Complicated control flow	✓
Pred1	Simple if(false) test	✗
Pred2	Simple correlated tests	✓
Pred3	Simple correlated tests	✓
Pred4	Test with an integer variable	✓
Pred5	Test with a complex conditional	✓
Pred6	Test with addition	✗
Pred7	Test with multiple variables	✗

Table 6.3 Jif case studies

Name	Description	`fa`
`A`	Simple explicit flow test	✓
`Account`	Simple explicit flow test	✓
`ConditionalLeak`	Explicit flow in if statement	✓
`Do`	Implicit flow in the loop	✓
`Do2`	Implicit flow if and loop	✓
`Do3`	Implicit flow loop and if	✓
`Do4`	Implicit flow loop and if	✓
`Do5`	Implicit flow loop and if	✓
`If1`	Simple implicit flow	✓
`Implicit`	Simple implicit flow	✓

Figure 6.7 Simple strings example

```
...
public = “STRING”;
if(secret)
        public.trim();
else
        public.upperCase();
endif
...
```

7 Future Works

In this chapter we present some research directions on which we are currently investigating.

7.1 Information Flow Analysis of JavaScript Code

Static techniques have benefits of reducing runtime overhead and dynamic techniques have the benefits of permissiveness, which is of particular importance in dynamic applications, where freshly generated code is evaluated. This setting is becoming increasingly more important with the growing use of web browsers as application platforms, since the client side language, JavaScript, is a highly dynamic language. The main critical aspect, to perform a static analysis on JavaScript program, are three. First, JavaScript is dynamically typed, therefore the type checking can be performed only at runtime. Second, JavaScript allows the redefinition of functions, methods and prototypes, both user defined and built-in; this means , for instance, that a program might run in not standard environment. And third, JavaScript contains the function eval, which evaluates the code dynamically. Typically, the strings to be evaluated are not known at the time of analysis. Hence, to provide an analysis for this language is an hard challenge. In this section we present some preliminary results of analysis of JavaScript code developed by the collaboration with Sergio Maffeis.

JavaScript is widely used in Web programming and it is implemented in every major browser. Moreover, many contemporary web sites incorporate untrusted content, for example, it serve third-party advertisements allow users to post comments that are then served to others, or allow users to add their own applications to the site. For this reason, recently, many different work about security problems of JavaScript languages have been appeared [98, 97, 99, 124].

Notice that JavaScript is quite different from the languages treated in previous chapter: JavaScript commands and programs are expressions which can be evaluated in different environment. In order to apply the dependency analysis presented in Chapter 4, we started from the operational semantics for JavaScript showed in [96] and the concept presented in Section 5.4 to define an abstract representation to track the information flows within the programs. Moreover, for the sake of simplicity, we

Table 7.1 JavaScript expressions

$\mathsf{e}, \mathsf{e}_0, \mathsf{e}_1, \mathsf{e}_2$	$\in$	$\mathfrak{E}$	
$\mathsf{e}, \mathsf{e}_0, \mathsf{e}_1, \mathsf{e}_2$	$::=$	n	$\mathsf{n} \in \mathbb{N}$
	$\mid$	$\mathsf{x}, \mathsf{y}, \mathsf{z}, \cdots$	Variables
	$\mid$	$\mathsf{e}_0 = \mathsf{e}_1$	Assignments
	$\mid$	$\mathsf{if}(\mathsf{e}_0)\{\mathsf{e}_1\}\{\mathsf{e}_2\}$	Conditional
	$\mid$	$\mathsf{e}_0; \mathsf{e}_1$	Sequential Composition
	$\mid$	$\mathsf{while}(\mathsf{e}_0)\{\mathsf{e}_1\}$	Loop
	$\mid$	$\mathsf{e}_0 \oplus \mathsf{e}_1$	Arithmetic Binary Operation
	$\mid$	$\mathsf{e}_0 \& \mathsf{e}_1$	Boolean Operation (and)
	$\mid$	$\mathsf{e}_0 \mid\mid \mathsf{e}_1$	Boolean Operation (or)
	$\mid$	$\mathsf{e.x}$	Member Access
	$\mid$	$\lambda\mathsf{x.e}$	Function
	$\mid$	$\mathsf{e}_0(\mathsf{e}_1)$	Function Call
	$\mid$	$\{\mathsf{x}_1 : \mathsf{e}_1, \cdots, \mathsf{x}_n : \mathsf{e}_n\}$	Literal Object

consider only a subset of the JavaScript expressions (denoted by $\mathfrak{E}$ and showed in Table 7.1).

In this new representation, each expression is abstracted by an identifier ($[\![\mathsf{e}]\!]$ is the abstraction of the expression e), generally by a natural number. Let Γ, ϕ, S and P be a description of an expression, a propositional formula, an advanced-substitution and a predicate, respectively. A predicate is an expression (or negated expression) which have a boolean meaning[1] and an advanced-substitution consists, roughly speaking, in a multiple substitution. For instance, consider ${}^{[\![\mathsf{e}_0]\!],[\![\mathsf{e}_1]\!]}/_{[\![\mathsf{e}_2]\!]}$ and the propositional formula $\phi = [\![\mathsf{e}_2]\!] \rightarrow [\![\mathsf{e}_3]\!]$. The substitution will replace all occurrences of $[\![\mathsf{e}_2]\!]$ in the propositional formula ϕ with both $[\![\mathsf{e}_0]\!]$ and $[\![\mathsf{e}_1]\!]$. This substitution will duplicate each implication that involves $[\![\mathsf{e}_2]\!]$ and it will connect them by the "and" ($\wedge$) operator; in this case we will obtain $\phi_{new} = [\![\mathsf{e}_0]\!] \rightarrow [\![\mathsf{e}_3]\!] \wedge [\![\mathsf{e}_1]\!] \rightarrow [\![\mathsf{e}_3]\!]$. A propositional formula, differently from the version of the previous chapters, is based on the identifier. Formally, Γ is defined as follows.

$$\begin{aligned}\Gamma :&= (R, S)\\ &\mid ((P) \Rightarrow \Gamma)\\ &\mid \Gamma \wedge \Gamma\\ &\mid \varepsilon\end{aligned}$$

And the abstract semantics to obtain the abstract representation, Γ, from a program is defined in Figures 7.1 and 7.2. Notice that, when we have a function $\mathsf{e} = \lambda\mathsf{x.e}'$, we denote by $arg([\![\mathsf{e}]\!])$ the formal parameter x and by $body([\![\mathsf{e}]\!])$ the body of the function, e'. In order to better understand, consider the following example.

[1]Notice that each expression could be evaluated as a boolean value.

Table 7.2 Table expression - identifier

x	0
$\mathsf{x = 0}$	1
$\mathsf{0}$	2
$\mathsf{x = 0; while(y + x \leq 10)\{x = x + 1; w = z + x\}}$	3
$\mathsf{while(y + x \leq 10)\{x = x + 1; w = z + x; z = z - 1\}}$	4
y	5
$\mathsf{y + x}$	6
x	7
$\mathsf{y + x \leq 10}$	8
$\mathsf{10}$	9
x	10
$\mathsf{x = x + 1}$	11
x	12
$\mathsf{x + 1}$	13
$\mathsf{1}$	14
$\mathsf{x = x + 1; w = z + x}$	15
w	16
$\mathsf{w = z + x}$	17
z	18
$\mathsf{z + x}$	19
x	20

Example 14. *Consider the expression:*

$$\mathsf{x = 0; while(y + x \leq 10)\{x = x + 1; w = z + x; \}}.$$

First of all we assign to each sub-expression an identifier (a natural number), Table 7.2. Then we can apply the rules of abstract semantics to the program. The steps are following.

#	*Evaluated expression*	*Obtained state*
1	x	ε
2	$\mathsf{0}$	ε
3	$\mathsf{x = 0}$	ε
4	y	ε
5	x	ε
6	$\mathsf{y + x}$	$\langle \mathtt{T}, {}^{(5,7)}/_{6} \rangle$
7	$\mathsf{10}$	$\langle \mathtt{T}, {}^{(5,7)}/_{6} \rangle$
8	$\mathsf{y + x \leq 10}$	$\langle \mathtt{T}, {}^{(5,7)}/_{6} \rangle \wedge \langle \mathtt{T}, {}^{(6,9)}/_{8} \rangle$
9	x	$\langle \mathtt{T}, {}^{(5,7)}/_{6} \rangle \wedge \langle \mathtt{T}, {}^{(6,9)}/_{8} \rangle$
10	x	$\langle \mathtt{T}, {}^{(5,7)}/_{6} \rangle \wedge \langle \mathtt{T}, {}^{(6,9)}/_{8} \rangle$
11	$\mathsf{1}$	$\langle \mathtt{T}, {}^{(5,7)}/_{6} \rangle \wedge \langle \mathtt{T}, {}^{(6,9)}/_{8} \rangle$

12	x + 1	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle$
13	x = x + 1	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$
14	w	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$
15	z	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$
16	x	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$
17	z + x	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$ $\langle \mathtt{T},^{(18,20)}/_{19}\rangle$
18	w = z + x	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$ $\langle \mathtt{T},^{(18,20)}/_{19}\rangle \wedge$ $\langle 19 \rightarrow 16,^{16}/_{17}\rangle$
19	x = x + 1; w = z + x	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$ $\langle \mathtt{T},^{(18,20)}/_{19}\rangle \wedge$ $\langle 19 \rightarrow 16,^{16}/_{17}\rangle \wedge$ $\langle \mathtt{T},^{(11,17)}/_{15}\rangle$
20	while(y + x ≤ 10){x = x + 1; w = z + x; }	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$ $\langle \mathtt{T},^{(18,20)}/_{19}\rangle \wedge$ $\langle 19 \rightarrow 16,^{16}/_{17}\rangle \wedge$ $\langle \mathtt{T},^{(11,17)}/_{15}\rangle \wedge$ $(y + x \leq 10) \Rightarrow \langle 8 \rightarrow 15,^{15}/_4\rangle$
21	x = 0; while(y + x ≤ 10){x = x + 1; w = z + x; }	$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge$ $\langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle$ $\langle \mathtt{T},^{(18,20)}/_{19}\rangle \wedge$ $\langle 19 \rightarrow 16,^{16}/_{17}\rangle \wedge$ $\langle \mathtt{T},^{(11,17)}/_{15}\rangle \wedge$ $(y + x \leq 10) \Rightarrow \langle 8 \rightarrow 15,^{15}/_4\rangle$ $\wedge\langle \mathtt{T},^{(1,4)}/_3\rangle$

At the end of the computation we obtain, as abstract representation:

$$\langle \mathtt{T},^{(5,7)}/_6\rangle \wedge \langle \mathtt{T},^{(6,9)}/_8\rangle \wedge \langle \mathtt{T},^{(12,14)}/_{13}\rangle \wedge \langle \mathtt{T},^{10}/_{11}\rangle\langle \mathtt{T},^{(18,20)}/_{19}\rangle \wedge \langle 19 \rightarrow 16,^{16}/_{17}\rangle \wedge$$
$$\langle \mathtt{T},^{(11,17)}/_{15}\rangle \wedge (y + x \leq 10) \Rightarrow \langle 8 \rightarrow 15,^{15}/_4\rangle \wedge \langle \mathtt{T},^{(1,4)}/_3\rangle$$

At this point we split the propositional formulae and the substitutions:

$$\begin{aligned}\phi =&(19 \rightarrow 16) \wedge (\mathsf{y} + \mathsf{x} \leq \mathsf{10}) \Rightarrow (8 \rightarrow 15)\\ Subs =&^{(1,4)}/_3;^{15}/_4;^{(11,17)}/_{15};^{16}/_{17};^{(18,20)}/_{19};^{10}/_{11};^{(12,14)}/_{13};^{(6,9)}/_8;^{(5,7)}/_6;\end{aligned}$$

And, finally, we apply the advanced substitutions on the propositional formula:

$\phi = 18 \rightarrow 16 \wedge 20 \rightarrow 16 \wedge$
$(\mathsf{y} + \mathsf{x} \leq \mathsf{10}) \Rightarrow [(5 \rightarrow 10) \wedge (7 \rightarrow 10) \wedge (9 \rightarrow 10) \wedge (5 \rightarrow 16) \wedge (7 \rightarrow 16) \wedge (9 \rightarrow 16)]$

In this way, we track all the information flow between identifiers.

Currently, we are working on to define more formally all steps presented inside the example and, at the same time, to remove false alarms ($9 \rightarrow 10$ and $9 \rightarrow 16$ are not information flow). The aim of this approach consists to track statically the information flow and then use some dynamic analyses to refine the results.

7.2 Declassification

As described in Chapter 2, for many applications a complete separation between secret and public is too restrictive. Consider the code in Example 12, to avoid the warning and to pass the analysis we have to introduce a note which claims that the information flow from x_6 and x_4 to results is necessary for the user authentication.

The declassification technique is based on the lowering security classification of selected information. Reasoning on toy languages, like the one presented in Chapter 4, the implementation of declassification seems to be easy: through a renaming of propositional variables. Considering the same example code (Figure 4.2, Example 12), by a renaming of some variables, more precisely $\overline{\mathsf{x}_6}$ and $\overline{\mathsf{x}_4}$ in the if statements, with some new propositional variable, which have the right security classification, the analysis can assert that there is not information leakage.

One of the issue that are introduced by the classical approach consists in the adding of annotations, inside the code, to select the variables which have to be declassified. The modularity of our analysis should permits us to avoid the annotation inside the code and to select the variables between the analysis of the code and the properties verification.

Currently, we are investigating to formalize the declassification in the theoretical analysis definition and to implement it in Sails.

7.3 Multithreading

Multithreading appears to be the most common way to build parallel applications in commercial programming languages. Parallelism cannot be avoided it is the only immediate and native way in order to take advantage from multicore architectures, that represent the most important trend of CPU market today [64]. Multithreading consists in partitioning a large application into many different subtasks, each of them possibly running in parallel. Threads can communicate through shared memory and they can synchronize each other through monitors, semaphores, etc. The arbitrary interleaving occurring during the parallel execution of different threads may lead to nondeterministic behaviors and it is often difficult to reproduce. Even if researchers have worked on parallel computing during almost the last 30 years, there are still important shortcomings in formal methods and static analysis with respect to multithreading. Hence, we can assert that it is hard to debug multithreaded programs.

The intuition about the extension of our analysis to multithreaded programs seems, like for the declassification, to be easy. Consider the analysis which involves the attackers that observe each steps of computation, in this case our analysis detects all possible information flow in the program. When we execute a program in multithreaded way the information flows are, intuitively, always the same. Currently, we are studying the literature to formalize and verify this intuition.

7.4 Conclusions

The section above described our current work, but there are a lot of different ways to extends the analysis. For example it could be possible to apply information flow analysis to perform program slicing [150], to analyze security policy (RBAC, [62]) or to add to the dependency between variable the probability.

Figure 7.1 Abstract semantics for JavaScript expressions - Part 1

Variable or Constant:

$$\Gamma, \mathsf{x} \to \Gamma$$

Assignment:

$$\frac{\dfrac{\nabla}{\Gamma, \mathsf{e}_0 \to \Gamma^1} \qquad \dfrac{\nabla}{\Gamma^1, \mathsf{e}_1 \to \Gamma^2} \qquad \Gamma' = \Gamma^2 \wedge \langle [\![\mathsf{e}_1]\!] \to [\![\mathsf{e}_0]\!], {}^{[\![\mathsf{e}_0]\!]} /_{[\![\mathsf{e}_0 := \mathsf{e}_1]\!]} \rangle}{\Gamma, \mathsf{e}_0 := \mathsf{e}_1 \to \Gamma'}$$

Conditional:

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \to \Gamma^1} \qquad \dfrac{\nabla}{\Gamma^1, e_1 \to \Gamma^2} \qquad \dfrac{\nabla}{\Gamma^1, e_2 \to \Gamma^3}}{\Gamma, if(e_0)\{e_1\}\{e_2\} \to \Gamma'}$$

where$\Gamma\ ' = (e_0 \Rightarrow \langle [\![\mathsf{e}_0]\!] \to [\![\mathsf{e}_1]\!], {}^{e1} /_{if(e_0)\{e_1\}\{e_2\}} \rangle \wedge \Gamma^2) \wedge (\neg e_0 \Rightarrow \langle [\![\mathsf{e}_0]\!] \to [\![\mathsf{e}_2]\!], {}^{e1} /_{if(e_0)\{e_1\}\{e_2\}} \rangle \wedge \Gamma^3)$

Sequential Composition:

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \to \Gamma^1} \qquad \dfrac{\nabla}{\Gamma^1, e_1 \to \Gamma^2} \qquad \Gamma' = \Gamma^2 \wedge \langle \emptyset, {}^{([\![\mathsf{e}_0]\!], [\![\mathsf{e}_1]\!])} /_{e_0; e_1} \rangle}{\Gamma, e_0; e_1 \to \Gamma'}$$

Loop

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \to \Gamma^1} \qquad \dfrac{\nabla}{\Gamma, e_1 \to \Gamma^2} \qquad \Gamma' = \Gamma^1 \wedge (e_0 \Rightarrow \langle [\![\mathsf{e}_0]\!] \to [\![\mathsf{e}_1]\!], {}^{e_1} /_{while(e_0)\{e_1\}} \rangle \wedge \Gamma_2)}{\Gamma, while(e_0)\{e_1\} \to \Gamma'}$$

Binary Operation - Arithmetic and Boolean (except for & and ||)

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \to \Gamma^1} \qquad \dfrac{\nabla}{\Gamma^1, e_1 \to \Gamma^2} \qquad \Gamma' = \Gamma^2 \wedge \langle \emptyset, {}^{([\![\mathsf{e}_0]\!], [\![\mathsf{e}_1]\!])} /_{e_0 \oplus e_1} \rangle}{\Gamma, e_0 \oplus e_1 \to \Gamma'}$$

Figure 7.2 Abstract semantics for JavaScript expressions - Part 2

Binary Operation (Boolean - and)

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \rightarrow \Gamma^1} \qquad \dfrac{\nabla}{\Gamma, e_1 \rightarrow \Gamma^2} \qquad \Gamma' = \Gamma^1 \wedge (e_0 \Rightarrow \langle [\![\mathsf{e_0}]\!] \rightarrow [\![\mathsf{e_1}]\!], ^{([\![\mathsf{e_0}]\!],[\![\mathsf{e_1}]\!])} /_{e_0 \& e_1} \rangle \wedge \Gamma^2)}{\Gamma, e_0 \& e_1 \rightarrow \Gamma'}$$

Binary Operation (Boolean - or)

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \rightarrow \Gamma^1} \qquad \dfrac{\nabla}{\Gamma, e_1 \rightarrow \Gamma^2} \qquad \Gamma' = \Gamma^1 \wedge (\neg e_0 \Rightarrow \langle [\![\mathsf{e_0}]\!] \rightarrow [\![\mathsf{e_1}]\!], ^{([\![\mathsf{e_0}]\!],[\![\mathsf{e_1}]\!])} /_{e_0 || e_1} \rangle \wedge \Gamma^2)}{\Gamma, e_0 \mid\mid e_1 \rightarrow \Gamma'}$$

Member Access:

$$\frac{\dfrac{\nabla}{\Gamma, e \rightarrow \Gamma^1} \qquad \Gamma' = \Gamma^1 \wedge \langle \emptyset, ^{([\![\mathsf{e}]\!] \cdot [\![\mathsf{x}]\!])} /_{e.x} \rangle}{\Gamma, e.x \rightarrow \Gamma'}$$

Function

$$\frac{\dfrac{\nabla}{\Gamma, e \rightarrow \Gamma^1} \qquad \Gamma' = \Gamma^1}{\Gamma, \lambda x.e \rightarrow \Gamma'}$$

Function Call:

$$\frac{\dfrac{\nabla}{\Gamma, e_0 \rightarrow \Gamma^1} \qquad \dfrac{\nabla}{\Gamma^1, e_1 \rightarrow \Gamma^2} \qquad \Gamma' = \Gamma^2 \wedge \langle \emptyset, ^{(body([\![\mathsf{e_0}]\!]))} /_{e_0(e_1)} \rangle}{\Gamma, e_0(e_1) \rightarrow \Gamma'}$$

Literal Object:

$$\frac{\dfrac{\nabla}{\Gamma, e_1 \rightarrow \Gamma^1} \qquad \ldots \qquad \dfrac{\nabla}{\Gamma, e_n \rightarrow \Gamma^n}}{\Gamma, \{x_1 : e_1, \ldots, x_n : e_n\} \rightarrow \Gamma'}$$

where$\Gamma\ ' = \Gamma^n \wedge \langle [\![\mathsf{e_0}]\!] \rightarrow [\![\mathsf{x_0}]\!] \wedge \ldots \wedge [\![\mathsf{e_0}]\!] \rightarrow [\![\mathsf{x_0}]\!], \emptyset \rangle$

8 Conclusions

Nowadays, the major challenge for secure information flow analysis is to develop a good formalism for specifying useful information flow policies. This must be general enough for a wide variety of applications, but not too complicated for users to understand. In this thesis, we presented a static analysis tool, apt to verify secure information flows, with a flexible construction and usable without adding any kind of notation. In particular, thanks to its modular construction, we can deal with the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators.

As a first contribution, we provided new results about the combination of different domains and the use of widening and narrowing operators. We provided a formal definition of the widening and narrowing operations already introduced in the literature, we proved that the widening and narrowing operators are preserved by abstraction and we indicated a method to construct widening operators for a product domain such as the reduced and cartesian products.

A second contribution of this thesis is the development of a variables' dependencies analysis based on a propositional formulae domain. Despite the analysis is equivalent to the certification mechanism presented by Denning and Denning [57] (and then it results to be equivalent to others analyses), we used the Pos domain, originally design for the analysis of groundness dependencies for logic program, as a powerful way to tracking the information flows among program's variables.

Through the combination of the variables dependency analysis and the numerical analyses, we obtained a strictly more accurate analysis and then more precise results. The main goal of this refinement consists in the detection of some kind of false alarms, which permits, in this way, to make applicable our analysis to practical cases. Moreover, we afforded a modular construction which allows to deal with the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators. Notice that further refinement are possible by combination of our analysis with other domains.

Despite the number of works about information flow analysis, the implementations are very few. According to Li and Zdancewic [90], one of the reasons that limited the application of information flow analysis in real cases is that they require to re-write the whole system in the new language. The fourth contribution of this thesis is in fact Sails: a new tool which aims an information flow analysis. Differently from the other tools in literature, Sails does not require to re-write the whole program because it works with mainstream languages like Java. It does not require to add any manual annotation, since it infers the flow of information directly on the original program, and it asks what are the private data that have not to be leaked to the user during the analysis execution. In order to evaluate Sails we applied the SecuriBench-micro suite on it. The preliminary results gave us the confirmation of theoretical results about efficiency and accuracy.

A

Dependency Analysis Formal Proofs

Theorem A.1. *$\theta : \Sigma^{\star\diamond} \rightarrow \wp(\Sigma^{\star\sharp})$ is monotonic:* $\mathsf{x} \preceq^{\diamond} \mathsf{y} \Rightarrow \theta(\mathsf{x}) \sqsubseteq^{\sharp} \theta(\mathsf{y})$

Proof. Let $\mathsf{x}_0 = \{\sigma_0 \rightarrow \ldots \rightarrow \sigma_n\}$ and $\mathsf{x}_1 = \{\sigma'_0 \rightarrow \ldots \rightarrow \sigma'_m\}$ be two elements of$\Sigma^{\star\diamond}$ such that $\mathsf{x}_0 \preceq^{\diamond} \mathsf{x}_1$ and consider $\theta(\mathsf{x}_0) = \{{\sigma^{\sharp}}_0, \ldots, {\sigma^{\sharp}}_n\}$ and $\theta(\mathsf{x}_1) = \{{\sigma^{\sharp}}'_0, \ldots, {\sigma^{\sharp}}'_m\}$. By the definition of "$\preceq^{\diamond}$" we know that $n \leq m$, $\forall i \in [0, n].\sigma_i = \sigma'_i$. Therefore, by the definition of θ, we have that $\forall i \in [0, n].{\sigma^{\sharp}}_i = {\sigma^{\sharp}}'_i$. Then, by definition of "$\sqsubseteq^{\sharp}$", $\theta(\mathsf{x}_0) \sqsubseteq^{\sharp} \theta(\mathsf{x}_1)$. □

Theorem A.2. *$\alpha^{\sharp} : \wp(\Sigma^{\star\diamond}) \rightarrow \wp(\Sigma^{\star\sharp})$ is monotonic:* $\mathsf{X} \subseteq \mathsf{Y} \Rightarrow \alpha^{\sharp}(\mathsf{X}) \sqsubseteq^{\sharp} \alpha^{\sharp}(\mathsf{Y})$

Proof. Consider $\mathsf{X}_0, \mathsf{X}_1 \in \wp(\Sigma^{\star\diamond})$ such that $\mathsf{X}_0 \subseteq \mathsf{X}_1$, $\alpha^{\sharp}(\mathsf{X}_0) = \sqcup^{\sharp}\{\theta(\pi^{\diamond}) \mid \pi^{\diamond} \in \mathsf{X}_0\}$ and $\alpha^{\sharp}(\mathsf{X}_1) = \sqcup^{\sharp}\{\theta(\pi^{\diamond}) \mid \pi^{\diamond} \in \mathsf{X}_1\}$. By definition of "$\subseteq$", $\forall \pi_0^{\diamond} \in \mathsf{X}_0.\exists \pi_1^{\diamond} \in \mathsf{X}_1$ such that $\pi_0^{\diamond} \preceq^{\diamond} \pi_1^{\diamond}$. By Theorem A.1, $\theta(\pi_0^{\diamond}) \sqsubseteq^{\sharp} \theta(\pi_1^{\diamond})$ for all $\pi_0^{\diamond} \in \mathsf{X}_0$ and $\pi_1^{\diamond} \in \mathsf{X}_1$. Then we have $\alpha^{\sharp}(\mathsf{X}_0) \sqsubseteq^{\sharp} \alpha^{\sharp}(\mathsf{X}_1)$ □

Theorem A.3. *$\gamma^{\sharp} : \wp(\Sigma^{\star\sharp}) \rightarrow \wp(\Sigma^{\star\diamond})$ is monotonic:* $\mathsf{X} \sqsubseteq^{\sharp} \mathsf{Y} \Rightarrow \gamma^{\sharp}(\mathsf{X}) \subseteq \gamma^{\sharp}(\mathsf{Y})$

Proof. Consider $\mathsf{X}_0, \mathsf{X}_1 \in \wp(\Sigma^{\star\sharp})$ such that $\mathsf{X}_0 \sqsubseteq^{\sharp} \mathsf{X}_1$, $\gamma^{\sharp}(\mathsf{X}_0) = \{\pi^{\diamond} \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^{\diamond}) \sqsubseteq^{\sharp} \mathsf{X}_0 \wedge l(\pi^{\diamond}) \in \ell(\mathsf{X}_0^{\dashv})\}$ and $\gamma^{\sharp}(\mathsf{X}_1) = \{\pi^{\diamond} \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^{\diamond}) \sqsubseteq^{\sharp} \mathsf{X}_1 \wedge l(\pi^{\diamond}) \in \ell(\mathsf{X}_1^{\dashv})\}$. By definition of "$\sqsubseteq^{\sharp}$" and by Theorem A.1, for all $\pi_0^{\diamond} \in \gamma^{\sharp}(\mathsf{X}_0)$ exists $\pi_1^{\diamond} \in \gamma^{\sharp}(\mathsf{X}_1)$. Therefore $\gamma^{\sharp}(\mathsf{X}_0) \sqsubseteq^{\sharp} \gamma^{\sharp}(\mathsf{X}_1)$. □

Theorem A.4. *$\alpha^{\sharp} \circ \gamma^{\sharp}$ is the identity:* $\alpha^{\sharp}(\gamma^{\sharp}(\mathsf{X})) = \mathsf{X}$

Proof. Let X be an element of $\wp(\Sigma^{\star\sharp})$. By definition of $\alpha^{\sharp}$, $\alpha^{\sharp}(\gamma^{\sharp}(\mathsf{X})) = \sqcup^{\sharp}\{\theta(\pi^{\diamond}) \mid \pi^{\diamond} \in \gamma^{\sharp}(\mathsf{X})\}$. By definition of $\gamma^{\sharp}$, $\alpha^{\sharp}(\gamma^{\sharp}(\mathsf{X})) = \sqcup^{\sharp}\{\theta(\pi^{\diamond}) \mid \theta(\pi^{\diamond}) \sqsubseteq^{\sharp} \mathsf{X} \wedge l(\pi^{\diamond}) \in \ell(\mathsf{X}^{\dashv})\}$. Then, by definition of "$\sqcup^{\sharp}$" and "$\sqsubseteq^{\sharp}$", $\alpha^{\sharp}(\gamma^{\sharp}(\mathsf{X})) = \mathsf{X}$. □

Theorem A.5. *$\gamma^{\sharp} \circ \alpha^{\sharp}$ is extensive:* $\mathsf{X} \sqsubseteq^{\sharp} \gamma^{\sharp}(\alpha^{\sharp}(\mathsf{X}))$

Proof. Consider $\mathsf{X} \in \wp(\Sigma^{\star\diamond})$. By definition of $\gamma^{\sharp}$, $\gamma^{\sharp}(\alpha^{\sharp}(\mathsf{X})) = \{\pi^{\diamond} \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^{\diamond}) \sqsubseteq^{\sharp} \alpha^{\sharp}(\mathsf{X}) \wedge l(\pi^{\diamond}) \in \ell(\alpha^{\sharp}(\mathsf{X})^{\dashv})\}$. By definition of $\alpha^{\sharp}$, $\gamma^{\sharp}(\alpha^{\sharp}(\mathsf{X})) = \{\pi^{\diamond} \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^{\diamond}) \sqsubseteq^{\sharp} \sqcup^{\sharp}\{\theta(\pi^{\diamond}) \mid \pi^{\diamond} \in \mathsf{X}\} \wedge l(\pi^{\diamond}) \in \ell(\alpha^{\sharp}(\mathsf{X})^{\dashv})\}$. By definition of "$\sqcup^{\sharp}$", "$\sqsubseteq^{\sharp}$" and by Theorem A.1, $\mathsf{X} \sqsubseteq^{\sharp} \gamma^{\sharp}(\alpha^{\sharp}(\mathsf{X}))$. □

Theorem A.6. $\wp(\Sigma^{\star\diamond}) \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \wp(\Sigma^{\star\sharp})$ *is a Galois insertion.*

Proof. Notice that $\wp(\Sigma^{\star\diamond})$ and $\wp(\Sigma^{\star\sharp})$ are two complete lattices, $\gamma^\sharp$ and $\alpha^\sharp$ are monotonic (Theorems A.2 and A.3), $\alpha^\sharp \circ \gamma^\sharp$ is the identity (Theorem A.4) and $\gamma^\sharp \circ \alpha^\sharp$ is extensive (Theorem A.5). Therefore $\wp(\Sigma^{\star\diamond}) \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion. □

Theorem A.7. $\wp(\Sigma^{\star}) \underset{\alpha^\diamond}{\overset{\gamma^\diamond}{\leftrightarrows}} \wp(\Sigma^{\star\diamond})$*is an isomorphism.*

Proof. We have to prove that $\gamma^\diamond \circ \alpha^\diamond = \alpha^\diamond \circ \gamma^\diamond = id$, where id is the identity function. Let X and Y be an element of $\wp(\Sigma^{\star\diamond})$ and an element of $\wp(\Sigma^{\star})$, respectively.

$$\begin{array}{rcl}
\alpha^\diamond(\gamma^\diamond(\mathsf{X})) & = & \{\langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \sigma_0 \xrightarrow{\ell_0\,\mathsf{a}_0} \ldots \xrightarrow{\ell_m\,\mathsf{a}_m} \sigma_{m+1} \in \gamma^\diamond(\mathsf{X})\} \\
& & \text{by definition of } \alpha^\diamond \\
& = & \{\langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \sigma_0 \xrightarrow{\ell_0\,\mathsf{a}_0} \ldots \xrightarrow{\ell_m\,\mathsf{a}_m} \sigma_{m+1} \in \{\pi \mid \alpha^\diamond(\{\pi\}) \subseteq X\}\} \\
& & \text{by definition of } \gamma^\diamond \\
& = & \{\langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \in \mathsf{X}\} \\
& = & \mathsf{X}
\end{array}$$

$$\begin{array}{rcl}
\gamma^\diamond(\alpha^\diamond(\mathsf{Y})) & = & \{\pi \in \wp(\Sigma^\star) \mid \alpha^\diamond(\{\pi\}) \subseteq \alpha^\diamond(\mathsf{Y})\} \\
& & \text{by definition of } \gamma^\diamond \\
& = & \{\pi \in \wp(\Sigma^\star) \mid \alpha^\diamond(\{\pi\}) \subseteq \{\alpha^\diamond(\{\pi'\}) \mid \pi' \in \mathsf{Y}\}\} \\
& & \text{by definition of } \alpha^\diamond \\
& = & \{\pi \in \wp(\Sigma^\star) \mid \pi \in \mathsf{Y}\} \\
& = & \mathsf{Y}
\end{array}$$

□

B
Widening and Narrowing Formal Proofs

Theorem (3.6)**.** *Let* $(\mathsf{P}, \leq)$ *be a poset, and let* $\nabla : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *be a pair-widening operator on* P*. Define* $\nabla_\star : \wp(\mathsf{P}) \twoheadrightarrow \mathsf{P}$ *such that:*

- *dom*$(\nabla_\star) = \mathsf{R}_1 \cup \mathsf{R}_2$*, where*
  $\mathsf{R}_1 = \{\{x, y\} \mid x, y \in \mathsf{P}\}$*, and*
  $\mathsf{R}_2 = \{\mathsf{S} \subseteq \mathsf{P} \mid \mathsf{S}$ *is a finite ascending chain*$\}$.

- $\forall\{x, y\} \in \mathsf{R}_1$,
  $$\nabla_\star(\{\mathsf{x}, \mathsf{y}\}) =_{def} \begin{cases} \mathsf{x}\nabla\mathsf{y} & \textit{if } \mathsf{x} \leq \mathsf{y} \\ \mathsf{z} \in \{\mathsf{x}\nabla\mathsf{y}, \mathsf{y}\nabla\mathsf{x}\} & \textit{randomly, otherwise.} \end{cases}$$

- $\forall \mathsf{S} = \{\mathsf{x}_i \mid \mathsf{x}_0 \leq \mathsf{x}_1 \leq \cdots \leq \mathsf{x}_j\} \in \mathsf{R}_2$,
  $\nabla_\star(\mathsf{S}) =_{def} (((\mathsf{x}_0\nabla\mathsf{x}_1)\nabla\mathsf{x}_2 \ldots)\nabla\mathsf{x}_j)$.

Then $\nabla_\star$ *is a set-widening operator.*

Proof. We have to show that both covering and termination requirements hold for $\nabla_\star$.

- *Covering.* Let $\mathsf{S} \subseteq \mathsf{P}$ such that $\nabla_\star(\mathsf{S})$ is defined. We have to show that $\forall s \in \mathsf{S} : s \leq \nabla_\star(\mathsf{S})$.
  Case $\mathsf{S} \in \mathsf{R}_1$: it follows from the definition of ∇.
  Case $\mathsf{S} \in \mathsf{R}_2$: it follows by induction on the length of the ascending chain, and by the transitivity of the partial order.

- *Termination.* Consider the ascending chain $\{\mathsf{x}_i\}_{i\geq 0}$. Consider the corresponding ascending chain $\{\hat{\mathsf{y}}_i\}_{i\geq 0}$ obtained by ∇ (see Definition 3.6), and the ascending chain $\{\mathsf{y}_i\}_{i\geq 0}$ obtained using $\nabla_\star$ (see Definition 3.5). We can prove by induction that for each index i, $\mathsf{y}_i = \hat{\mathsf{y}}_i$.
  The basis is true, as $\mathsf{y}_0 = x_0 = \hat{\mathsf{y}}_0$.

Consider the inductive step:

$$\begin{array}{rcll}
\mathsf{y}_{i+1} & = & \nabla_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i+1\}) & \text{by (ii) of Definition 3.5} \\
& = & (((\mathsf{x}_0 \nabla \mathsf{x}_1) \nabla \mathsf{x}_2 \ldots) \nabla \mathsf{x}_{i+1}) & \text{by Definition of } \nabla_\star \\
& = & \nabla_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\}) \nabla \mathsf{x}_{i+1} & \text{again by Definition of } \nabla_\star \\
& = & \hat{\mathsf{y}}_i \nabla \mathsf{x}_{i+1} & \text{by inductive hypotesis} \\
& = & \hat{\mathsf{y}}_{i+1} & \text{by (ii) of Definition 3.6}
\end{array}$$

As the sequence $\{\hat{\mathsf{y}}_i\}_{i\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_i\}_{i\geq 0}$.

□

Theorem (3.7)**.** *Let* $(\mathsf{P}, \leq)$ *be a poset, and let* $\nabla_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *be a set-widening operator on* P *such that*

- *dom*$(\nabla_\star) \supseteq \{\{\mathsf{x}, \mathsf{y}\} \mid \mathsf{x}, \mathsf{y} \in \mathsf{P}\}$, *and*
- $\forall \mathsf{S} \subseteq \mathsf{P}, \forall \mathsf{x} \in \mathsf{P}$, *if* $\mathsf{S} \cup \{\mathsf{x}\} \subseteq$ *dom*$(\nabla_\star)$ *then also* $\mathsf{S} \subseteq$ *dom*$(\nabla_\star)$
- $\forall \mathsf{S} \subseteq \mathsf{P}, \forall \mathsf{x} \in \mathsf{P}, \nabla_\star(\mathsf{S} \cup \{\mathsf{x}\}) = \nabla_\star(\{\nabla_\star(\mathsf{S}), \mathsf{x}\})$.

Then, the binary operator $\nabla : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *defined by* $\mathsf{x} \nabla \mathsf{y} = \nabla_\star(\{\mathsf{x}, \mathsf{y}\})$ *is a pair-widening operator.*

Proof. First, observe that ∇ is well defined. The covering requirement follows immediately from the definition of ∇ and the covering property of $\nabla_\star$. Now, consider an ascending chain $\{\mathsf{x}_i\}_{i\geq 0}$ in P, and the ascending chain $\mathsf{y}_0 = \mathsf{x}_0, \mathsf{y}_{i+1} = \mathsf{y}_i \nabla \mathsf{x}_i$. As $\nabla_\star$ is a set-widening, we know that the sequence $\mathsf{y}'_0 = x_0, \mathsf{y}'_i = \nabla_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\}$ stabilizes finitely. We show by induction that for each i, $\mathsf{y}_i = \mathsf{y}'_i$. The basis is true, as $\mathsf{y}_0 = \mathsf{x}_0 = \mathsf{y}'_0$. On the induction step,

$$\begin{array}{rcll}
\mathsf{y}'_{i+1} & = & \nabla_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i+1\} & \text{by point (ii) of Definition 3.5} \\
& = & \nabla_\star(\{\nabla_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\}), \mathsf{x}_{i+1}\}) & \text{by hypothesis on } \nabla_\star \\
& = & \nabla_\star(\{\mathsf{y}'_i, \mathsf{x}_{i+1}\}) & \text{by point (ii) of Definition 3.5} \\
& = & \nabla_\star(\{\mathsf{y}_i, \mathsf{x}_{i+1}\}) & \text{by inductive hypothesis} \\
& = & \mathsf{y}_i \nabla \mathsf{x}_{i+1} & \text{by Definition of } \nabla \\
& = & \mathsf{y}_{i+1} & \text{by point (ii) of Definition 3.6.}
\end{array}$$

As the sequence $\{\mathsf{y}'_i\}_{i\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_i\}_{i\geq 0}$. □

Theorem (3.8)**.** *Let* $(\mathsf{P}, \leq)$ *be a poset, and let* $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *be a pair-narrowing operator on* P. *Define* $\Delta_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ *such that:*

- *dom*$(\Delta_\star) = \mathsf{R}_1 \cup \mathsf{R}_2$, *where*
  $\mathsf{R}_1 = \{\{\mathsf{x}, \mathsf{y}\} \mid \mathsf{x}, \mathsf{y} \in \mathsf{P} : \exists\ glb(\mathsf{x}, \mathsf{y})\}$, *and*
  $\mathsf{R}_2 = \{\mathsf{S} \subseteq \mathsf{P} \mid \mathsf{S}$ *is a finite descending chain*$\}$.

- $\forall \{\mathsf{x}, \mathsf{y}\} \in \mathsf{R}_1$,
  $$\Delta_\star(\{\mathsf{x}, \mathsf{y}\}) =_{def} \begin{cases} \mathsf{y}\Delta\mathsf{x} & \textit{if}\ \mathsf{x} \leq \mathsf{y} \\ glb(\{\mathsf{x}, \mathsf{y}\}) & \textit{otherwise.} \end{cases}$$
- $\forall \mathsf{S} = \{\mathsf{x}_i \mid \mathsf{x}_0 \geq \mathsf{x}_1 \geq \cdots \geq \mathsf{x}_j\} \in \mathsf{R}_2$,
  $\Delta_\star(\mathsf{S}) =_{def} ((((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta\mathsf{x}_2)\Delta)\ldots\Delta\mathsf{x}_j)$.

Then $\Delta_\star$ is a set-narrowing operator.

Proof. We have to show that both bounding and termination requirements hold for $\Delta_\star$.

- *Bounding.* Let $\mathsf{S} \subseteq \mathsf{S}$ such that$\Delta\ _\star(\mathsf{S})$ is defined. We have to show that $glb(\mathsf{S}) \leq \Delta_\star(\mathsf{S}) \leq \mathsf{s}$.
  Case $\mathsf{S} \in \mathsf{R}_1$: it follows from the definition ofΔ.
  Case $\mathsf{S} \in \mathsf{R}_2$: it follows by induction on the length of the decreasing chain $(\mathsf{x}_0 \geq \mathsf{x}_1 \geq \ldots \geq \mathsf{x}_j)$, and by the transitivity of the partial order.

- *Termination.* Consider the decreasing chain $\{\mathsf{x}_i\}_{i\geq 0}$. Consider the corresponding decreasing chain $\{\hat{\mathsf{y}}_i\}_{i\geq 0}$ obtained by Δ (see Definition 3.9), and the decreasing chain $\{\mathsf{y}_i\}_{i\geq 0}$ obtained using$\Delta\ _\star$ (see Definition 3.8). We can prove by induction that for each index i, $\mathsf{y}_i = \hat{\mathsf{y}}_i$.
  The basis is true, as $\mathsf{y}_0 = \mathsf{x}_0 = \hat{\mathsf{y}}_0$.
  Consider the inductive step:

$$\begin{array}{rcll} \mathsf{y}_{i+1} & = & \Delta_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i+1\}) & \text{by definition of} \\ & & & \text{the sequence } \{\mathsf{y}_j\}_{j\geq 0} \\ & = & (((((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta x_2)\Delta)\ldots\Delta\mathsf{x}_i)\Delta\mathsf{x}_{i+1}) & \text{by definition of}\Delta\ _\star \\ & = & \Delta_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\})\Delta\mathsf{x}_{i+1} & \text{again by definition of}\Delta\ _\star \\ & = & \hat{\mathsf{y}}_i\Delta\mathsf{x}_{i+1} & \text{by inductive hypothesis} \\ & = & \hat{\mathsf{y}}_{i+1} & \text{by (ii) of pair-narrowing def.} \end{array}$$

  As the sequence $\{\hat{\mathsf{y}}_i\}_{i\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_i\}_{i\geq 0}$.

□

Theorem (3.9)**.** *Let $(\mathsf{P}, \leq)$ be a poset, and let $\Delta_\star : \wp(\mathsf{P}) \nrightarrow \mathsf{P}$ be a set-narrowing operator on P such that*

1. *$dom(\Delta_\star) \supseteq \{\{\mathsf{x}, \mathsf{y}\} \mid \mathsf{x}, \mathsf{y} \in \mathsf{P}\}$, and*
2. *$\forall \mathsf{S} \subseteq \mathsf{P}$, $\forall \mathsf{x} \in \mathsf{P}$, if $\mathsf{S} \cup \{\mathsf{x}\} \subseteq dom(\Delta_\star)$ then also $\mathsf{S} \subseteq dom(\Delta_\star)$*
3. *$\forall \mathsf{S} \subseteq \mathsf{P}$, $\forall \mathsf{x} \in \mathsf{P}$, $\Delta_\star(\mathsf{S} \cup \{\mathsf{x}\}) = \Delta_\star(\{\Delta_\star(\mathsf{S}), \mathsf{x}\})$.*

Then, the binary operator $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ defined $\mathsf{x}\Delta\mathsf{y} = \Delta_\star(\{\mathsf{x}, \mathsf{y}\})$ is a pair-narrowing operator.

Proof. First, observe that Δ is well defined. The bounding requirement follows immediately from the definition of Δ and the bounding property of$\Delta\ _\star$. Now, consider an descending chain $\{\mathsf{x}_i\}_{i\geq 0}$ in P, and the descending chain $\mathsf{y}_0 = \mathsf{x}_0, \mathsf{y}_{i+1} = \mathsf{y}_i \Delta \mathsf{x}_{i+1}$. As $\Delta_\star$ is a set-narrowing, we know that the sequence $\mathsf{y}'_0 = \mathsf{x}_0, \mathsf{y}'_i = \Delta_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\})$ stabilizes finitely. We show by induction that for each i, $\mathsf{y}_i = \mathsf{y}'_i$. The basis is true, as $\mathsf{y}_0 = \mathsf{x}_0 = \mathsf{y}'_0$. On the induction step,

$$\begin{array}{rcll}
\mathsf{y}_{i+1} & = & \mathsf{y}_i \Delta \mathsf{x}_{i+1} & \text{by definition of the sequence } \{\mathsf{y}_j\}_{j\geq 0} \\
 & = & \Delta_\star(\{\mathsf{y}_i, \mathsf{x}_{i+1}\}) & \text{by } \Delta \text{ definition} \\
 & = & \Delta_\star(\{\Delta_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i\}), \mathsf{x}_{i+1}\}) & \text{by induction hypothesis} \\
 & = & \Delta_\star(\{\mathsf{x}_j \mid 0 \leq j \leq i+1\}) & \text{by the property 3} \\
 & = & \mathsf{y}'_{i+1} & \text{by (ii) of set-narrowing definition}
\end{array}$$

As the sequence $\{\mathsf{y}'_i\}_{i\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_i\}_{i\geq 0}$. □

Theorem (3.10)**.** *Let* ∇_A *and* ∇_D *be pair-widening operators defined on the posets* A *and* D*, respectively.*
The binary operator $\nabla : (\mathsf{A} \times \mathsf{D}) \times (\mathsf{A} \times \mathsf{D}) \to (\mathsf{A} \times \mathsf{D})$ *defined by* $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \times \mathsf{D} : \langle \mathsf{a}, \mathsf{d} \rangle \nabla \langle \mathsf{a}', \mathsf{d}' \rangle = \langle \mathsf{a} \nabla_{\mathsf{A}} \mathsf{a}', \mathsf{d} \nabla_{\mathsf{D}} \mathsf{d}' \rangle$ *is a pair-widening operator.*

Proof.

- *Covering*

$$\begin{array}{rll}
 & \mathsf{a} \leq \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}' \text{ and } \mathsf{d} \leq \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}' & \text{by covering of } \nabla_{\mathsf{A}}, \nabla_{\mathsf{D}} \\
\Rightarrow & \langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}' \rangle & \text{by definition of } \leq \text{ on } \mathsf{A} \times \mathsf{D} \\
\Rightarrow & \langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}, \mathsf{d} \rangle \nabla \langle \mathsf{a}', \mathsf{d}' \rangle & \text{by definition of } \nabla.
\end{array}$$

- *Termination* Let $\{\langle \mathsf{a}_i, \mathsf{d}_i \rangle\}_{i\geq 0}$ be an ascending chain in the cartesian product $\mathsf{A} \times \mathsf{D}$. We have to show that the sequence $\langle \mathsf{u}_0, \mathsf{v}_0 \rangle = \langle \mathsf{a}_0, \mathsf{d}_0 \rangle$, $\langle \mathsf{u}_{i+1}, \mathsf{v}_{i+1} \rangle = \langle \mathsf{u}_i, \mathsf{v}_i \rangle \nabla \langle \mathsf{a}_i, \mathsf{d}_i \rangle$ stabilizes after a finite number of terms.

  By the termination property of ∇_{A} and ∇_{D}, both the sequence $\hat{\mathsf{a}}_0 = \mathsf{a}_0$, $\hat{\mathsf{a}}_{i+1} = \hat{\mathsf{a}}_i \nabla_{\mathsf{A}} \mathsf{a}_i$, and the sequence $\hat{\mathsf{d}}_0 = \mathsf{d}_0$, $\hat{\mathsf{d}}_{i+1} = \hat{\mathsf{d}}_i \nabla_{\mathsf{D}} \mathsf{d}_i$ stabilize finitely.

  It can be easily proved by induction that for each i, $\langle \mathsf{u}_i, \mathsf{v}_i \rangle = \langle \hat{\mathsf{a}}_i, \hat{\mathsf{d}}_i \rangle$. Therefore, the sequence $\{\langle \mathsf{u}_j, \mathsf{v}_j \rangle\}_{j\geq 0}$ stabilizes finitely too.

□

Theorem (3.11)**.** *Let* Δ_{A} *and* Δ_{D} *be pair-narrowing operators defined on the posets* A *and* D*, respectively.*
The binary operator $\Delta : (\mathsf{A} \times \mathsf{D}) \times (\mathsf{A} \times \mathsf{D}) \to (\mathsf{A} \times \mathsf{D})$ *defined by* $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \times \mathsf{D} : \langle \mathsf{a}, \mathsf{d} \rangle \Delta \langle \mathsf{a}', \mathsf{d}' \rangle = \langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle$ *is a pair-narrowing operator.*

Proof.

- *Bounding*

$$\begin{array}{ll}
& \forall \mathsf{a}, \mathsf{a}' \in \mathsf{A} : (\mathsf{a} \leq \mathsf{a}') \Longrightarrow (\mathsf{a} \leq \mathsf{a}' \Delta \mathsf{a} \leq \mathsf{a}') \text{ and} \\
& \forall \mathsf{d}, \mathsf{d}' \in \mathsf{D} : (\mathsf{d} \leq \mathsf{d}') \Longrightarrow (\mathsf{d} \leq \mathsf{d}' \Delta \mathsf{d} \leq \mathsf{d}') \\
& \quad \text{by bounding of} \Delta_{\mathsf{A}} \text{ and} \Delta_{\mathsf{B}} \\
\Rightarrow & \langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}' \Delta_{\mathsf{A}} \mathsf{a}, \mathsf{d}' \Delta_{\mathsf{D}} \mathsf{d} \rangle \leq \langle \mathsf{a}', \mathsf{d}' \rangle \\
& \quad \text{by definition of } \leq \text{ on } \mathsf{A} \times \mathsf{D} \\
\Rightarrow & \langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}', \mathsf{d}' \rangle \Delta \langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}', \mathsf{d}' \rangle \\
& \quad \text{by definition of} \Delta
\end{array}$$

- *Termination* Let $\{\langle \mathsf{a}_i, \mathsf{d}_i \rangle\}_{i \geq 0}$ be a descending chain in the cartesian product $\mathsf{A} \times \mathsf{D}$. We have to show that the sequence $\langle \mathsf{u}_0, \mathsf{v}_0 \rangle = \langle \mathsf{a}_0, \mathsf{d}_0 \rangle$, $\langle \mathsf{u}_{i+1}, \mathsf{v}_{i+1} \rangle = \langle \mathsf{u}_i, \mathsf{v}_i \rangle \Delta \langle \mathsf{a}_i, \mathsf{d}_i \rangle$ stabilizes after a finite number of terms.

  By the termination property ofΔ_{A} andΔ_{D}, both the sequence $\hat{\mathsf{a}}_0 = \mathsf{a}_0$, $\hat{\mathsf{a}}_{i+1} = \hat{\mathsf{a}}_i \Delta_{\mathsf{A}} \mathsf{a}_i$, and the sequence $\hat{\mathsf{d}}_0 = \mathsf{d}_0$, $\hat{\mathsf{d}}_{i+1} = \hat{\mathsf{d}}_i \Delta_{\mathsf{D}} \mathsf{d}_i$ stabilize finitely.

  By induction we prove that for each i, $\langle \mathsf{u}_i, \mathsf{v}_i \rangle = \langle \hat{\mathsf{a}}_i, \hat{\mathsf{d}}_i \rangle$.
  The basis is true: $\langle \mathsf{u}_0, \mathsf{v}_0 \rangle = \langle \mathsf{a}_0, \mathsf{d}_0 \rangle = \langle \hat{\mathsf{a}}_0 \hat{\mathsf{d}}_0 \rangle$. On the induction step,

$$\begin{array}{rcll}
\langle \mathsf{u}_{i+1}, \mathsf{v}_{i+1} \rangle & = & \langle \mathsf{u}_i, \mathsf{v}_i \rangle \Delta \langle \mathsf{a}_{i+1}, \mathsf{d}_{i+1} \rangle & \text{by definition of } \rangle \mathsf{u}_{i+1}, \mathsf{v}_{i+1} \rangle \\
& = & \langle \hat{\mathsf{a}}_i, \hat{\mathsf{d}}_i \rangle \Delta \langle \mathsf{a}_{i+1}, \mathsf{d}_{i+1} \rangle & \text{by induction hypothesis} \\
& = & \langle \hat{\mathsf{a}}_i \Delta_{\mathsf{A}} \mathsf{a}_{i+1}, \hat{\mathsf{d}}_i \Delta_{\mathsf{D}} \mathsf{d}_{i+1} \rangle & \text{by definition of} \Delta \\
& = & \langle \hat{\mathsf{a}}_{i+1}, \hat{\mathsf{d}}_{i+1} \rangle & \text{by definition of } \hat{\mathsf{a}}_{i+1} \text{ and } \hat{\mathsf{d}}_{i+1}
\end{array}$$

  Therefore, the sequence $\{\langle \mathsf{u}_j, \mathsf{v}_j \rangle\}_{j \geq 0}$ stabilizes finitely too.

□

Theorem (3.12)**.** *Let* $(\mathsf{P}, \leq)$ *be a lattice satisfying the ascending chain property. Let* ∇_1, ∇_2 *be two pair-widening operators on* P*. Then, the binary operators* $\nabla_{\sqcap}, \nabla_{\sqcup}$ *defined by*

$$\begin{array}{rcl}
\mathsf{x} \nabla_{\sqcap} \mathsf{y} & = & (\mathsf{x} \nabla_1 \mathsf{y}) \sqcap (\mathsf{x} \nabla_2 \mathsf{y}) \\
\mathsf{x} \nabla_{\sqcup} \mathsf{y} & = & (\mathsf{x} \nabla_1 \mathsf{y}) \sqcup (\mathsf{x} \nabla_2 \mathsf{y})
\end{array}$$

are pair-widening operators.

Proof. It follows by properties of $\sqcup$ and $\sqcap$. □

Theorem (3.13)**.** *Let* $(\mathsf{P}, \leq)$ *be a lattice satisfying the descending chain property. Let* Δ_1, Δ_2 *be two pair-narrowing operators on* P*. Then, the binary operators* $\Delta_{\sqcap}, \Delta_{\sqcup}$ *defined by*

$$\begin{array}{rcl}
\mathsf{x} \Delta_{\sqcap} \mathsf{y} & = & (\mathsf{x} \Delta_1 \mathsf{y}) \sqcap (\mathsf{x} \Delta_2 \mathsf{y}) \\
\mathsf{x} \Delta_{\sqcup} \mathsf{y} & = & (\mathsf{x} \Delta_1 \mathsf{y}) \sqcup (\mathsf{x} \Delta_2 \mathsf{y})
\end{array}$$

are pair-narrowing operators.

Proof. It follows by properties of $\sqcup$ and $\sqcap$, as for the widening operators. □

Lemma (3.1)**.** *Let* ∇ *be a pair-widening operator on a lattice* $(\mathsf{P}, \leq)$*, such that for every finite set* $\{\mathsf{x}_i\}_{0\leq i\leq n}$ *and for every* $\mathsf{y} \in \mathsf{P}$*,* $(((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_n)\ \nabla\ (\mathsf{x}_0 \sqcup \mathsf{x}_1 \sqcup \dots \sqcup \mathsf{x}_n \sqcup \mathsf{y}) = (((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_n)\nabla \mathsf{y}$*, then* ∇ *is a strong pair-widening operator.*

Proof. We need to focus only on the termination property. Consider the sequence $\{\mathsf{x}_i\}_{0\leq i\leq n}$, and the increasing sequence

$$\mathsf{z}_0 = \mathsf{x}_0,\ \mathsf{z}_{i+1} = \mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_{i+1}$$

We show by induction that the two increasing sequences $\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = y_i \nabla \mathsf{x}_{i+1}$ and $\mathsf{h}_0 = \mathsf{z}_0,\ \mathsf{h}_{i+1} = \mathsf{h}_i \nabla \mathsf{z}_{i+1}$ are such that $\forall i : \mathsf{y}_i = \mathsf{h}_i$.
The basis is trivial, as $\mathsf{y}_0 = \mathsf{x}_0 = \mathsf{z}_0 = \mathsf{h}_0$.
The induction step:

$$\begin{array}{lll}
\mathsf{h}_{i+1} & = \mathsf{h}_i \nabla \mathsf{z}_{i+1} & \text{by def. of } \{\mathsf{h}_j\}_{j\geq 0} \\
& = \mathsf{y}_i \nabla \mathsf{z}_{i+1} & \text{by inductive hypothesis} \\
& = (((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_i)\nabla \mathsf{z}_{i+1} & \text{by def. of } \{\mathsf{y}_j\}_{j\geq 0} \\
& = (((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_i)\nabla(\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_{i+1}) & \text{by def. of } \{\mathsf{z}_j\}_{j\geq 0} \\
& = (((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_i)\nabla \mathsf{x}_{i+1} & \text{by hypothesis on } \nabla \\
& = \mathsf{y}_{i+1} & \text{by def. of } \{\mathsf{y}_j\}_{j\geq 0}
\end{array}$$

As the increasing sequence $\{\mathsf{h}_j\}_{j\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_j\}_{j\geq 0}$. □

Theorem (3.16)**.** *Let* ∇ *be an associative pair-widening operator on a lattice* $(\mathsf{P}, \leq)$*, such that for* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x}\nabla\mathsf{y} = \mathsf{x}\nabla(\mathsf{x}\sqcup\mathsf{y})$*, then* ∇ *is a strong pair-widening operator.*

Proof. By Lemma 3.1, it is sufficient to prove by induction that for every finite set $\{\mathsf{x}_i\}_{0\leq i\leq n}$ and for every $\mathsf{y} \in \mathsf{P}$, $(((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_n)\ \nabla\ (\mathsf{x}_0 \sqcup \mathsf{x}_1 \sqcup \dots \sqcup \mathsf{x}_n \sqcup \mathsf{y}) = (((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_n)\nabla \mathsf{y}$.
The basis ($n = 1$) follows immediately from the hypothesis.
Induction step:

$$\begin{array}{ll}
(((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_n)\ \nabla\ (\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_n \sqcup \mathsf{y}) = & \text{by inductive hypothesis} \\
(((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla(\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_n))\ \nabla\ (\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_n \sqcup \mathsf{y}) = & \text{by associativity of} \\
& \nabla \text{ and of } \sqcup \\
((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla((\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_n)\nabla((\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_n) \sqcup \mathsf{y})) = & \text{by applying} \\
& \text{the hypothesis} \\
((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla((\mathsf{x}_0 \sqcup \dots \sqcup \mathsf{x}_n)\nabla \mathsf{y}) = & \text{by associativity of } \nabla \\
(((\mathsf{x}_0 \nabla \mathsf{x}_1)\nabla \dots)\nabla \mathsf{x}_n)\nabla \mathsf{y}. &
\end{array}$$

□

Lemma (3.2)**.** *Let* Δ *be a pair-narrowing operator on a lattice* $(\mathsf{P}, \leq)$*, such that for every finite set* $\{\mathsf{x}_i\}_{0\leq i\leq n}$ *and for every* $\mathsf{y} \in \mathsf{P}$*,* $(((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \dots)\Delta \mathsf{x}_n)\ \Delta\ (\mathsf{x}_0 \sqcap \mathsf{x}_1 \sqcap \dots \sqcap \mathsf{x}_n \sqcap \mathsf{y}) = (((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \dots)\Delta \mathsf{x}_n)\Delta \mathsf{y}$*, then* Δ *is a strong pair-narrowing operator.*

Proof. We need to focus only on the termination property. Consider the sequence $\{\mathsf{x}_i\}_{0\leq i\leq n}$, and the decreasing sequence

$$\mathsf{z}_0 = \mathsf{x}_0,\ \mathsf{z}_{i+1} = \mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_{i+1}$$

We show by induction that the two increasing sequences $\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i \Delta \mathsf{x}_{i+1}$ and $\mathsf{h}_0 = \mathsf{z}_0,\ \mathsf{h}_{i+1} = \mathsf{h}_i \Delta \mathsf{z}_{i+1}$ are such that $\forall i : \mathsf{y}_i = \mathsf{h}_i$.
The basis is trivial, as $\mathsf{y}_0 = \mathsf{x}_0 = \mathsf{z}_0 = \mathsf{h}_0$.
The induction step:

$$\begin{array}{rcll}
\mathsf{h}_{i+1} & = & \mathsf{h}_i \Delta \mathsf{z}_{i+1} & \text{by definition of } \{\mathsf{h}_j\}_{j\geq 0} \\
 & = & \mathsf{y}_i \Delta \mathsf{z}_{i+1} & \text{by inductive hypothesis} \\
 & = & (((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_i)\Delta \mathsf{z}_{i+1} & \text{by definition of } \{\mathsf{y}_j\}_{j\geq 0} \\
 & = & (((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_i)\Delta(\mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_{i+1}) & \text{by definition of } \{\mathsf{z}_j\}_{j\geq 0} \\
 & = & (((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_i)\Delta \mathsf{x}_{i+1} & \text{by hypothesis on}\Delta \\
 & = & \mathsf{y}_{i+1} & \text{by definition of } \{\mathsf{y}_j\}_{j\geq 0}
\end{array}$$

As the increasing sequence $\{\mathsf{h}_j\}_{j\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_j\}_{j\geq 0}$. □

Theorem (3.17)**.** *Let* Δ *be an associative pair-narrowing operator on a lattice* $(\mathsf{P}, \leq)$*, such that for* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{x}\Delta\mathsf{y} = \mathsf{x}\Delta(\mathsf{x} \sqcap \mathsf{y})$*, then* Δ *is a strong pair-narrowing operator.*

Proof. By Lemma 3.2, it is sufficient to prove by induction that for every finite set $\{\mathsf{x}_i\}_{0\leq i\leq n}$ and for every $\mathsf{y} \in \mathsf{P}$, $(((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_n)\ \Delta\ (\mathsf{x}_0 \sqcap \mathsf{x}_1 \sqcap \cdots \sqcap \mathsf{x}_n \sqcap \mathsf{y}) = (((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_n)\Delta \mathsf{y}$.
The basis $(n = 1)$ follows immediately from the hypothesis.
Induction step:

$$\begin{array}{ll}
(((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_n)\ \Delta\ (\mathsf{x}_0 \sqcap \cdots \sqcap \mathsf{x}_n \sqcap \mathsf{y}) \ = & \text{by inductive hypothesis} \\
(((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta(\mathsf{x}_0 \sqcap \cdots \sqcap \mathsf{x}_n))\ \Delta(\ \mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_n \sqcap \mathsf{y}) \ = & \text{by associativity of } \Delta \text{ and of } \sqcap \\
((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta((\mathsf{x}_0 \sqcap \cdots \sqcap \mathsf{x}_n)\Delta((\mathsf{x}_0 \sqcap \cdots \sqcap \mathsf{x}_n) \sqcap \mathsf{y})) \ = & \text{by applying the hypothesis} \\
((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta((\mathsf{x}_0 \sqcap \cdots \sqcap \mathsf{x}_n)\Delta \mathsf{y}) \ = & \text{by associativity of}\Delta \\
(((\mathsf{x}_0 \Delta \mathsf{x}_1)\Delta \ldots)\Delta \mathsf{x}_n)\Delta \mathsf{y}. &
\end{array}$$

□

Theorem (3.18)**.** *Let* $(\mathsf{P}, \leq)$ *be a meet-semi-lattice (the greatest lower bound* $\mathsf{x} \sqcap \mathsf{y}$ *exist for all* $\mathsf{x.y} \in \mathsf{L}$*) satisfying the descending chain condition (no strictly decreasing chain in* L *can be infinite). Let* $\Delta : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ *be a pair-narrowing operator such that* $\mathsf{x}\Delta\mathsf{y} = \mathsf{x} \sqcap \mathsf{y}$*. Then* Δ *is a strong lower-bound pair-narrowing.*

Proof. - *Bounding*: Consider $\mathsf{y} \leq \mathsf{x}$:

$$\begin{array}{lll} & \mathsf{x} \geq \mathsf{x}\Delta\mathsf{y} \geq \mathsf{y} & \text{by bounding property in Definition 3.9} \\ \Rightarrow & \mathsf{x} \geq \mathsf{x}\Delta\mathsf{y} \geq \mathsf{x} \sqcap \mathsf{y} & \text{by the relation between } \mathsf{x} \text{ and } \mathsf{y} \end{array}$$

This result is true for each $\mathsf{x},\ \mathsf{y} \in \mathsf{P} : \mathsf{x} \leq \mathsf{y}$, as request by bounding property in Definition 3.13.

- *Termination*: Consider the sequence $\{\mathsf{x}_i\}_{0 \leq i \leq n}$ and the decreasing sequence

$$\mathsf{z}_0 = \mathsf{x}_0,\ \mathsf{z}_{i+1} = \mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_{i+1}$$

We show by induction that the two increasing sequences $\mathsf{y}_0 = \mathsf{x}_0,\ \mathsf{y}_{i+1} = \mathsf{y}_i\Delta\mathsf{x}_{i+1}$ and $\mathsf{h}_0 = \mathsf{z}_0,\ \mathsf{h}_{i+1} = \mathsf{h}_i\Delta\mathsf{z}_{i+1}$ are such that $\forall i : \mathsf{y}_i = \mathsf{h}_i$.

The basis is trivial, as $\mathsf{y}_0 = \mathsf{x}_0 = \mathsf{z}_0 = \mathsf{h}_0$.

The induction step:

$$\begin{array}{rcll} \mathsf{h}_{i+1} & = & \mathsf{h}_i\Delta\mathsf{z}_{i+1} & \text{by definition of } \{\mathsf{h}_j\}_{j\geq 0} \\ & = & \mathsf{y}_i\Delta\mathsf{z}_{i+1} & \text{by inductive hypothesis} \\ & = & (((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta\ldots)\Delta\mathsf{x}_i)\Delta\mathsf{z}_{i+1} & \text{by definition of } \{\mathsf{y}_j\}_{j\geq 0} \\ & = & (((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta\ldots)\Delta\mathsf{x}_i)\Delta(\mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_{i+1}) & \text{by definition of } \{\mathsf{z}_j\}_{j\geq 0} \\ & = & (((\mathsf{x}_0 \sqcap \mathsf{x}_1) \sqcap \ldots) \sqcap \mathsf{x}_i) \sqcap (\mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_{i+1}) & \text{by definition of}\Delta \\ & = & (\mathsf{x}_0 \sqcap \ldots \sqcap \mathsf{x}_{i+1}) & \text{by properties of } \sqcap \\ & = & (((\mathsf{x}_0\Delta\mathsf{x}_1)\Delta\ldots)\Delta\mathsf{x}_i)\Delta\mathsf{x}_{i+1} & \text{by definition of}\Delta \\ & = & \mathsf{y}_{i+1} & \text{by definition of } \{\mathsf{y}_j\}_{j\geq 0} \end{array}$$

As the increasing sequence $\{\mathsf{h}_j\}_{j\geq 0}$ stabilizes after a finite number of terms, so does $\{\mathsf{y}_j\}_{j\geq 0}$.

□

Theorem (3.19)**.** *Let* $(\mathsf{P}, \leq)$ *be a poset and* Δ *be a pair-narrowing (Definition 3.9). If* $\forall \mathsf{v}, \mathsf{w} :\ \mathsf{v}\Delta(\mathsf{v} \sqcap \mathsf{w}) = \mathsf{v}\Delta\mathsf{w}$*, then* Δ *is a lower bound pair-narrowing (Definition 3.12).*

Proof. We need to focus only on the bounding property. By Definition 3.9, ofΔ, we know that

$$(\mathsf{x} \geq \mathsf{y}) \Longrightarrow (\mathsf{x} \geq (\mathsf{x}\Delta\mathsf{y}) \geq \mathsf{y})$$

We consider $\mathsf{u}, \mathsf{v} \in \mathsf{P}$, with $\mathsf{x} = \mathsf{v}$ and $\mathsf{y} = \mathsf{v} \sqcap \mathsf{w}$.
Then we have

$$\mathsf{v} \geq \mathsf{v}\Delta(\mathsf{v} \sqcap \mathsf{w}) \geq \mathsf{v} \sqcap \mathsf{w}$$

By assumption, we have that $\forall \mathsf{v}, \mathsf{w}$

$$\mathsf{v}\Delta(\mathsf{v} \sqcap \mathsf{w}) = \mathsf{v}\Delta\mathsf{w}$$

then we get

$$\mathsf{v} \geq \mathsf{v}\Delta\mathsf{w} \geq \mathsf{v} \sqcap \mathsf{w}$$

That is the bounding property of lower-bound pair-narrowing operator. □

Theorem (3.20)**.** *Let* $(\mathsf{P}, \leq)$ *be a poset and* Δ *be a lower-bound pair-narrowing (Definition 3.12).*
Consider $\mathsf{x}\Delta\mathsf{y}$*, it's simple to prove that* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{P} : \mathsf{y} \leq \mathsf{x}$ *than* Δ *is a pair-narrowing (Definition 3.9).*

Proof. We have, by Definition 3.12, ofΔ:

$$(\mathsf{x} \sqcap \mathsf{y}) \leq (\mathsf{x}\Delta\mathsf{y}) \leq \mathsf{x}$$

if we have that $\mathsf{y} \leq \mathsf{x}$ then $\mathsf{x} \sqcap \mathsf{y} = \mathsf{y}$ by definition. Therefore

$$(\mathsf{y} \leq \mathsf{x}) \Longrightarrow \mathsf{y} \leq \mathsf{x}\Delta\mathsf{y} \leq \mathsf{x}$$

as requested by Definition 3.9. □

Theorem (3.21)**.** *Let* ∇ *be a pair-widening operator on a complete lattice* $(\mathsf{L}, \leq)$ *such that* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{L} : \mathsf{x} \leq \mathsf{y} \Rightarrow \mathsf{x}\nabla\mathsf{x} \leq \mathsf{y}\nabla\mathsf{y}$*. Let* A *be the set* $\{\mathsf{x}\nabla\mathsf{x} \mid \mathsf{x} \in \mathsf{L}\}$*. Then* $\alpha_{\mathsf{LA}}(\mathsf{x}) = \mathsf{x}\nabla\mathsf{x}$ *is the lower adjoint of a Galois insertion between* L *and* A*, with the upper adjoint being the identity function.*

Proof. According to Definition 3.3, we have to show that $(\gamma_{\mathsf{AL}}, \mathsf{L}, \mathsf{A}, \alpha_{\mathsf{LA}})$ is a Galois insertion, with γ_{AL} being the identity function. Hence, it is sufficient to prove that $\forall \mathsf{x} \in \mathsf{L} : \mathsf{x} \leq \gamma_{\mathsf{AL}}(\alpha_{\mathsf{LA}}(\mathsf{x}))$, and that $\forall \mathsf{a} \in \mathsf{A} : \mathsf{a} = \alpha_{\mathsf{LA}}(\gamma_{\mathsf{AL}}(\mathsf{a}))$.

$$\begin{array}{ll} \forall \mathsf{x} \in \mathsf{L} : & \mathsf{x} \leq \mathsf{x}\nabla\mathsf{x}, \text{ by (i) of Definition 3.6} \\ \Rightarrow & \mathsf{x} \leq \alpha_{\mathsf{LA}}(\mathsf{x}), \text{ by definition of } \alpha_{\mathsf{LA}} \\ \Rightarrow & \mathsf{x} \leq \gamma_{\mathsf{AL}}(\alpha_{\mathsf{LA}}(\mathsf{x})), \text{ as } \gamma_{\mathsf{AL}} \text{ is the identity} \end{array}$$

$$\begin{array}{ll} \forall \mathsf{a} \in \mathsf{A} : & \mathsf{a} = \mathsf{a}\nabla\mathsf{a}, \text{ by definition of } \mathsf{A} \\ \Rightarrow & \mathsf{a} = (\gamma_{\mathsf{AL}}(\mathsf{a}))\nabla(\gamma_{\mathsf{AL}}(\mathsf{a})), \text{ as } \gamma_{\mathsf{AL}} \text{ is the identity} \\ \Rightarrow & \mathsf{a} = \alpha_{\mathsf{LA}}(\gamma_{\mathsf{AL}}(\mathsf{a})), \text{ by definition of } \alpha_{\mathsf{LA}} \end{array}$$

□

Theorem (3.22)**.** *Let* $\nabla_\star$ *be a set-widening operator on a complete lattice* $(\mathsf{L}, \leq)$ *such that* $\nabla_\star(\{\mathsf{x}\})$ *is defined for each* x *in* L*, and such that* $\forall \mathsf{x}, \mathsf{y} \in \mathsf{L} : \mathsf{x} \leq \mathsf{y} \Rightarrow \nabla_\star(\{\mathsf{x}\}) \leq \nabla_\star(\{\mathsf{y}\})$*. Let* A *be the set* $\{\nabla_\star(\{\mathsf{x}\}) \mid \mathsf{x} \in \mathsf{L}\}$*. Consider the function* $\alpha_{\mathsf{LA}} : \mathsf{L} \to \mathsf{A}$ *defined by* $\alpha_{\mathsf{LA}}(\mathsf{x}) = \nabla_\star(\{\mathsf{x}\})$*. Then,* α_{LA} *is the lower adjoint of a Galois insertion between* L *and* A*, with the upper adjoint being the identity function.*

Proof. The proof is similar to the proof of Theorem 3.21. □

Theorem (3.23)**.** *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* ∇_{C} *be a pair-widening on* C*. The binary operator* ∇_{D} *defined by* $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D}$, $\mathsf{d}_1\nabla_{\mathsf{D}}\mathsf{d}_2 = \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)\nabla_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_2))$ *is a pair-widening operator on* D*.*

Proof.

- *Covering.* Let us show that $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D} : \mathsf{d}_1 \leq \mathsf{d}_1 \nabla_{\mathsf{D}} \mathsf{d}_2$.

$$\begin{array}{rcll} \gamma_{\mathsf{DC}}(\mathsf{d}_1) & \leq & \gamma_{\mathsf{DC}}(\mathsf{d}_1)\nabla_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_2) & \text{by (ii) of Definition 3.6} \\ \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)) & \leq & \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)\nabla_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_2)) & \text{by monotonicity of } \alpha_{\mathsf{CD}} \\ \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)) & \leq & \mathsf{d}_1\nabla_{\mathsf{D}}\mathsf{d}_2 & \text{by definition of } \nabla_{\mathsf{D}} \\ \mathsf{d}_1 & \leq & \mathsf{d}_1\nabla_{\mathsf{D}}\mathsf{d}_2 & \text{as } \mathsf{G}_{\mathsf{CD}} \text{ is a Galois insertion.} \end{array}$$

The same way, we can also prove that $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D} : \mathsf{d}_2 \leq \mathsf{d}_1 \nabla_{\mathsf{D}} \mathsf{d}_2$.

- *Termination.* Consider the ascending chain $\{\mathsf{d}_i\}_{i\geq 0}$ in D. Consider the corresponding ascending chain $\gamma_{\mathsf{DC}}(\mathsf{d}_0) \leq \gamma_{\mathsf{DC}}(\mathsf{d}_1) \leq \dots$ in C. And consider the sequence $\mathsf{y}_0 = \gamma_{\mathsf{DC}}(\mathsf{d}_0)$, $\mathsf{y}_{i+1} = \mathsf{y}_i \nabla_{\mathsf{C}} \gamma_{\mathsf{DC}}(\mathsf{d}_{i+1})$. As ∇_{C} is a pair-widening operator, this ascending sequence stabilizes after a finite number of terms. We have to show that also the sequence $\hat{\mathsf{y}}_0 = \mathsf{d}_0$, $\hat{\mathsf{y}}_{i+1} = \hat{\mathsf{y}}_i \nabla_{\mathsf{D}} \mathsf{d}_{i+1}$ stabilizes after a finite number of terms. By induction, we prove that for each i, $\hat{\mathsf{y}}_i = \alpha_{\mathsf{CD}}(\mathsf{y}_i)$.

The basis is trivial, as $\hat{\mathsf{y}}_0 = \mathsf{d}_0 = \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_0)) = \alpha_{\mathsf{CD}}(\mathsf{y}_0)$.

Looking at the inductive step,

$$\begin{array}{rcll} \hat{\mathsf{y}}_{i+1} & = & \hat{\mathsf{y}}_i\nabla_{\mathsf{D}}\mathsf{d}_{i+1} & \text{by definition of the sequence } \{\hat{\mathsf{y}}_j\}_{j\geq 0}. \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_i)\nabla_{\mathsf{D}}\mathsf{d}_{i+1} & \text{by inductive hypotesis} \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_i)\nabla_{\mathsf{D}}\alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_{i+1})) & \text{as } \mathsf{G}_{\mathsf{CD}} \text{ is a Galois insertion} \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_i\nabla_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_{i+1})) & \text{by definition of } \nabla_{\mathsf{D}} \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_{i+1}) & \text{by definition of the sequence } \{\mathsf{y}_j\}_{j\geq 0}. \end{array}$$

□

Theorem (3.24)**.** *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* $\nabla_{*\mathsf{C}}$ *be a set-widening on* C*. The operator* $\nabla_{*\mathsf{D}}$ *defined by* $\forall \mathsf{S} \in \mathsf{D}$, $\nabla_{*\mathsf{D}}(\mathsf{S}) = \alpha_{\mathsf{CD}}(\nabla_{*\mathsf{C}}(\gamma_{\mathsf{DC}}(\mathsf{S})))$ *is a set-widening operator on* D*.*

Proof. The proof is similar to the proof of Theorem 3.23. □

Corollary (3.1)**.** *Let* A *and* D *be complete lattices, and let* ∇ *be a pair-widening operator over the cartesian product* $\mathsf{A} \times \mathsf{D}$*. Let* π_1 *be the projection on the first argument. The binary operator* $\nabla_{\mathsf{A}} : \mathsf{A} \times \mathsf{A} \to \mathsf{A}$ *defined by*

$$\mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}' = \pi_1(\langle \mathsf{a}, \top\rangle \nabla \langle \mathsf{a}', \top\rangle)$$

is a pair-widening operator.

Proof. It is sufficient to observe that the monotone functions $\alpha : \mathsf{A} \times \mathsf{D} \to \mathsf{A}$ and $\gamma : \mathsf{A} \to \mathsf{A} \times \mathsf{D}$ defined by

$$\begin{array}{l} \forall (\mathsf{a}, \mathsf{d}) \in \mathsf{A} \times \mathsf{D} : \ \alpha(\langle \mathsf{a}, \mathsf{d}\rangle) = \mathsf{a} \\ \forall \mathsf{a} \in \mathsf{A} : \ \gamma(\mathsf{a}) = \langle \mathsf{a}, \top \rangle \end{array}$$

form a Galois insertion between A and D. Therefore, by applying Theorem 3.23, the binary operator $\nabla' = \alpha(\gamma(\mathsf{a})\nabla\gamma(\mathsf{a}'))$ is a pair widening operator on A. To conclude, it is sufficient to observe that $\nabla_{\mathsf{A}} = \nabla'$. □

Theorem (3.25). *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* Δ_{C} *be a pair-narrowing on* C*. The binary operator* Δ_{D} *defined by* $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D}$, $\mathsf{d}_1 \Delta_{\mathsf{D}} \mathsf{d}_2 = \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)\Delta_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_2))$ *is a pair-narrowing operator on* D.

Proof.

- *Bounding.* Let us show that $\forall \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{D} : (\mathsf{d}_1 \leq \mathsf{d}_2) \Rightarrow (\mathsf{d}_1 \leq \mathsf{d}_2 \Delta_{\mathsf{D}} \mathsf{d}_1 \leq \mathsf{d}_2)$.

$$\begin{array}{rcccl} \gamma_{\mathsf{DC}}(\mathsf{d}_1) & \leq & \gamma_{\mathsf{DC}}(\mathsf{d}_2)\Delta_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_1) & \leq & \gamma_{\mathsf{DC}}(\mathsf{d}_2) \\ & & & & \text{by Definition 3.9} \\ \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)) & \leq & \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_2)\Delta_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_1)) & \leq & \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_2)) \\ & & & & \text{by monotonicity of } \alpha_{\mathsf{CD}} \\ \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_1)) & \leq & \mathsf{d}_2\Delta_{\mathsf{D}}\mathsf{d}_1 & \leq & \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_2)) \\ & & & & \text{by definition of} \Delta_{\mathsf{D}} \\ \mathsf{d}_1 & \leq & \mathsf{d}_2\Delta_{\mathsf{D}}\mathsf{d}_1 & \leq & \mathsf{d}_2 \\ & & & & \text{as } \mathsf{G}_{\mathsf{CD}} \text{ is a Galois insertion.} \end{array}$$

- *Termination.* Consider the decreasing chain $\{\mathsf{d}_i\}_{i\geq 0}$ in D. Consider the corresponding decreasing chain $\gamma_{\mathsf{DC}}(\mathsf{d}_0) \geq \gamma_{\mathsf{DC}}(\mathsf{d}_1) \geq \dots$ in C. And consider the sequence $\mathsf{y}_0 = \gamma_{\mathsf{DC}}(\mathsf{d}_0)$, $\mathsf{y}_{i+1} = \mathsf{y}_i\Delta_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_{i+1})$. As$\Delta_{\mathsf{C}}$ is a pair-narrowing operator, this descending sequence stabilizes after a finite number of terms. We have to show that also the sequence $\hat{\mathsf{y}}_0 = \mathsf{d}_0$, $\hat{\mathsf{y}}_{i+1} = \hat{\mathsf{y}}_i\Delta_{\mathsf{D}}\mathsf{d}_{i+1}$ stabilizes after a finite number of terms. By induction, we prove that for each i, $\hat{\mathsf{y}}_i = \alpha_{\mathsf{CD}}(\mathsf{y}_i)$.

  The basis is trivial, as $\hat{\mathsf{y}}_0 = \mathsf{d}_0 = \alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_0)) = \alpha_{\mathsf{CD}}(\mathsf{y}_0)$.

  Looking at the inductive step,

$$\begin{array}{rcll} \hat{\mathsf{y}}_{i+1} & = & \hat{\mathsf{y}}_i\Delta_{\mathsf{D}}\mathsf{d}_{i+1} & \text{by definition of the sequence } \{\hat{\mathsf{y}}_j\}_{j\geq 0}. \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_i)\Delta_{\mathsf{D}}\mathsf{d}_{i+1} & \text{by inductive hypotesis} \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_i)\Delta_{\mathsf{D}}\alpha_{\mathsf{CD}}(\gamma_{\mathsf{DC}}(\mathsf{d}_{i+1})) & \text{as } \mathsf{G}_{\mathsf{CD}} \text{ is a Galois insertion} \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_i\Delta_{\mathsf{C}}\gamma_{\mathsf{DC}}(\mathsf{d}_{i+1})) & \text{by definition of} \Delta_{\mathsf{D}} \\ & = & \alpha_{\mathsf{CD}}(\mathsf{y}_{i+1}) & \text{by definition of the sequence } \{\mathsf{y}_j\}_{j\geq 0}. \end{array}$$

□

Theorem (3.26). *Let* C *and* D *be two complete lattices, s.t.* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *is a Galois insertion. Let* $\Delta_{*\mathsf{C}}$ *be a set-narrowing on* C*. The operator* $\Delta_{*\mathsf{D}}$ *defined by* $\forall \mathsf{S} \in \mathsf{D}$, $\Delta_{*\mathsf{D}}(\mathsf{S}) = \alpha_{\mathsf{CD}}(\Delta_{*\mathsf{C}}(\gamma_{\mathsf{DC}}(\mathsf{S}))$ *is a set-narrowing operator on* D.

Proof. The proof is similar to the proof of Theorem 3.25. □

Corollary (3.2). *Let* A *and* D *be complete lattices, and let* Δ *be a pair-narrowing operator over the cartesian product* $\mathsf{A} \times \mathsf{D}$*. Let* π_1 *be the projection on the first argument. The binary operator* $\Delta_{\mathsf{A}} : \mathsf{A} \times \mathsf{A} \to \mathsf{A}$ *defined by*

$$\mathsf{a}\Delta_{\mathsf{A}}\mathsf{a}' = \pi_1(\langle \mathsf{a}, \top\rangle \Delta \langle \mathsf{a}', \top\rangle)$$

is a pair-narrowing operator.

Proof. It is sufficient to observe that the monotone functions $\alpha : \mathsf{A} \times \mathsf{D} \to \mathsf{A}$ and $\gamma : \mathsf{A} \to \mathsf{A} \times \mathsf{D}$ defined by

$$\forall (\mathsf{a},\mathsf{d}) \in \mathsf{A} \times \mathsf{D} : \ \alpha(\langle \mathsf{a},\mathsf{d}\rangle) = \mathsf{a}$$
$$\forall \mathsf{a} \in \mathsf{A} : \ \gamma(\mathsf{a}) = \langle \mathsf{a}, \top \rangle$$

form a Galois insertion between $\mathsf{A} \times \mathsf{D}$ and D. Therefore, by applying Theorem 3.25, the binary operator$\Delta\,' = \alpha(\gamma(\mathsf{a})\Delta\gamma(\mathsf{a}'))$ is a pair narrowing operator on A. To conclude, it is sufficient to observe that$\Delta_{\mathsf{A}} = \Delta'$. □

Lemma (3.3)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions. For* $\hat{\mathsf{a}} \in \mathsf{A}, \hat{\mathsf{d}} \in \mathsf{D}$, $\langle \mathsf{a},\mathsf{d}\rangle \in \mathsf{A} \sqcap \mathsf{D}$, *if* $\mathsf{a} \le \hat{\mathsf{a}}$ *and* $\mathsf{d} \le \hat{\mathsf{d}}$, *then* $\langle \mathsf{a},\mathsf{d}\rangle \le \mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle)$.

Proof. By $\sqcap$ properties and monotonicity of γ functions, $\gamma_{\mathsf{AC}}(\mathsf{a}) \sqcap \gamma_{\mathsf{DC}}(\mathsf{d}) \le \gamma_{\mathsf{AC}}(\hat{\mathsf{a}}) \sqcap \gamma_{\mathsf{DC}}(\hat{\mathsf{d}})$. Therefore, $\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle)$ is such that

$$\gamma(\langle \mathsf{a},\mathsf{d}\rangle) \le \gamma(\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle))$$

where γ is the upper adjoint of the Galois insertion $(\gamma, C, A \sqcap D, \alpha)$ as in Definition 3.14.
By applying α to both expressions, by monotonicity of α we get

$$\alpha(\gamma(\langle \mathsf{a},\mathsf{d}\rangle)) \le \alpha(\gamma(\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle)))$$

and by Galois insertion properties, as $\alpha \circ \gamma$ is the identity function, we get

$$\langle \mathsf{a},\mathsf{d}\rangle \le \mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle)$$

□

Lemma (3.4)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let ∇_{A} *and* ∇_{D} *be pair-widening operators defined on the lattice* A *and* D*, respectively.*
The binary operator $\bullet : (\mathsf{A}\sqcap\mathsf{D}) \times (\mathsf{A}\sqcap\mathsf{D}) \to (\mathsf{A}\sqcap\mathsf{D})$ *defined by* $\forall \langle \mathsf{a},\mathsf{d}\rangle, \langle \mathsf{a}',\mathsf{d}'\rangle \in \mathsf{A}\sqcap\mathsf{D}$: $\langle \mathsf{a},\mathsf{d}\rangle \bullet \langle \mathsf{a}',\mathsf{d}'\rangle = \mathit{reduce}(\langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}'\rangle)$ *is an extrapolator operator.*

Proof. Let $\langle \mathsf{a},\mathsf{d}\rangle, \langle \mathsf{a}',\mathsf{d}'\rangle \in \mathsf{A} \sqcap \mathsf{D}$. We have to prove that $\langle \mathsf{a},\mathsf{d}\rangle \le \langle \mathsf{a},\mathsf{d}\rangle \bullet \langle \mathsf{a}',\mathsf{d}'\rangle$.

$$\begin{array}{rlll} & \langle \mathsf{a},\mathsf{d}\rangle & \le & \langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}'\rangle & \text{by covering of } \nabla_{\mathsf{A}}, \nabla_{\mathsf{D}} \\ \Rightarrow & \langle \mathsf{a},\mathsf{d}\rangle & \le & \mathit{reduce}(\langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}'\rangle) & \text{by Lemma 3.3} \\ \Rightarrow & \langle \mathsf{a},\mathsf{d}\rangle & \le & \langle \mathsf{a},\mathsf{d}\rangle \bullet \langle \mathsf{a}',\mathsf{d}'\rangle & \text{by definition of } \bullet. \end{array}$$

In the same way, we can also prove that $\langle \mathsf{a}',\mathsf{d}'\rangle \le \langle \mathsf{a},\mathsf{d}\rangle \bullet \langle \mathsf{a}',\mathsf{d}'\rangle$. □

Theorem (3.27)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let ∇_{A} *and* ∇_{D} *be pair-widening operators defined on the lattice* A *and* D*, respectively, such that* $\forall\langle \mathsf{a}, \mathsf{d}\rangle \in \mathsf{A} \sqcap \mathsf{D}$, $\forall \mathsf{a}' \in A$, $\forall \mathsf{d}' \in D$: $\langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}'\rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the binary operator $\nabla : (\mathsf{A}\sqcap\mathsf{D}) \times (\mathsf{A}\sqcap\mathsf{D}) \to (\mathsf{A}\sqcap\mathsf{D})$ *defined by* $\forall\langle \mathsf{a}, \mathsf{d}\rangle, \langle \mathsf{a}', \mathsf{d}'\rangle \in \mathsf{A} \sqcap \mathsf{D}$: $\langle \mathsf{a}, \mathsf{d}\rangle\nabla\langle \mathsf{a}', \mathsf{d}'\rangle = \mathit{reduce}(\langle \mathsf{a}\nabla_{\mathsf{A}}\mathsf{a}', \mathsf{d}\nabla_{\mathsf{D}}\mathsf{d}'\rangle)$ *is a pair-widening operator.*

Proof. By Lemma 3.4, we need to focus only on the termination property.
Consider the increasing sequence $\langle \mathsf{a}_0, \mathsf{d}_0\rangle \le \langle \mathsf{a}_1, \mathsf{d}_1\rangle \dots$ in $\mathsf{A} \sqcap \mathsf{D}$. As the ordering $\le$ in $\mathsf{A} \sqcap \mathsf{D}$ is the same as in the cartesian product $\mathsf{A} \times \mathsf{D}$, we may consider the increasing sequence $\mathsf{a}_0 \le \mathsf{a}_1 \le \dots$ in A, and the increasing sequence $\mathsf{d}_0 \le \mathsf{d}_1 \le \dots$ in D. By the termination property of ∇_{A} and ∇_{D}, we know that the corresponding sequences $\hat{\mathsf{a}}_0 = \mathsf{a}_0$, $\hat{\mathsf{a}}_{i+1} = \hat{\mathsf{a}}_i\nabla_{\mathsf{A}}\mathsf{a}_{i+1}$, and $\hat{\mathsf{d}}_0 = \mathsf{d}_0$, $\hat{\mathsf{d}}_{i+1} = \hat{\mathsf{d}}_i\nabla_{\mathsf{D}}\mathsf{d}_{i+1}$ stabilize after a finite number of terms.

We show by induction that the increasing sequence $\langle \mathsf{a}'_0, \mathsf{d}'_0\rangle = \langle \mathsf{a}_0, d_0\rangle$, $\langle \mathsf{a}'_{i+1}, \mathsf{d}'_{i+1}\rangle = \langle \mathsf{a}'_i, \mathsf{d}'_i\rangle\nabla\langle \mathsf{a}_{i+1}, \mathsf{d}_{i+1}\rangle$ is such that $\forall i$: $\langle \mathsf{a}'_i, \mathsf{d}'_i\rangle = \langle \hat{\mathsf{a}}_i, \hat{\mathsf{d}}_i\rangle$.
The basis is trivial, as $\langle \mathsf{a}'_0, \mathsf{d}'_0\rangle = \langle \mathsf{a}_0, \mathsf{d}_0\rangle = \langle \hat{\mathsf{a}}_0, \hat{\mathsf{d}}_0\rangle$.
Induction step:

$$\begin{array}{rcll}
\langle \mathsf{a}'_{i+1}, \mathsf{d}'_{i+1}\rangle & = & \langle \mathsf{a}'_i, \mathsf{d}'_i\rangle\nabla\langle \mathsf{a}_{i+1}, \mathsf{d}_{i+1}\rangle & \text{by definition of } \{\langle \mathsf{a}'_j, \mathsf{d}'_j\rangle\}_{j\ge 0} \\
& = & \text{reduce}(\mathsf{a}'_i\nabla_{\mathsf{A}}\mathsf{a}_{i+1}, \mathsf{d}'_i\nabla_{\mathsf{D}}\mathsf{d}_{i+1}) & \text{by definition of } \nabla \\
& = & \langle \mathsf{a}'_i\nabla_{\mathsf{A}}\mathsf{a}_{i+1}, \mathsf{d}'_i\nabla_{\mathsf{D}}\mathsf{d}_{i+1}\rangle & \text{by the hypothesis} \\
& = & \langle \hat{\mathsf{a}}_{i+1}, \hat{\mathsf{d}}_{i+1}\rangle & \text{by definition of } \{\hat{\mathsf{a}}_j\}_{j\ge 0} \text{ and } \{\hat{\mathsf{d}}_j\}_{j\ge 0}
\end{array}$$

It follows that $\{\langle \mathsf{a}'_j, \mathsf{d}'_j\rangle\}_{j\ge 0}$ converges in a finite number of steps, namely the maximum between the termination indexes of $\{\hat{\mathsf{a}}_j\}_{j\ge 0}$ and $\{\hat{\mathsf{d}}_j\}_{j\ge 0}$. □

Lemma (3.5)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions. For* $\mathsf{a} \in \mathsf{A}$, $\mathsf{d} \in \mathsf{D}$, $\mathit{reduce}(\langle \mathsf{a}, \mathsf{d}\rangle) \le \langle \mathsf{a}, \mathsf{d}\rangle$.

Proof. By Definition 3.14: $\mathit{reduce}(\langle \mathsf{a}, \mathsf{d}\rangle) = \sqcap\mathsf{S}$, where $\mathsf{S} = \{\langle \mathsf{a}', \mathsf{d}'\rangle \mid \gamma_{\mathsf{AC}}(\mathsf{a})\sqcap\gamma_{\mathsf{DC}}(\mathsf{d}) = \gamma_{\mathsf{AC}}(\mathsf{a}')\sqcap\gamma_{\mathsf{DC}}(\mathsf{d}')\}$. We know that $\langle \mathsf{a}, \mathsf{d}\rangle \in \mathsf{S}$ and that all elements of S are comparable, than $\mathit{reduce}(\langle \mathsf{a}, \mathsf{d}\rangle) \le \langle \mathsf{a}, \mathsf{d}\rangle$. □

Lemma (3.6)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{G}_{\mathsf{CD}} = (\gamma_{\mathsf{DC}}, \mathsf{C}, \mathsf{D}, \alpha_{\mathsf{CD}})$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions. For* $\hat{\mathsf{a}} \in A, \hat{\mathsf{d}} \in D$, $\langle \mathsf{a}, \mathsf{d}\rangle \in \mathsf{A} \sqcap \mathsf{D}$*, if* $\hat{\mathsf{a}} \le \mathsf{a}$ *and* $\hat{\mathsf{d}} \le \mathsf{d}$*, then* $\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle) \le \langle \mathsf{a}, \mathsf{d}\rangle$.

Proof. By $\sqcap$ properties and monotonicity of γ functions, $\gamma_{\mathsf{AC}}(\mathsf{a}) \sqcap \gamma_{\mathsf{DC}}(\mathsf{d}) \ge \gamma_{\mathsf{AC}}(\hat{\mathsf{a}}) \sqcap \gamma_{\mathsf{DC}}(\hat{\mathsf{d}})$. Therefore, $\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle)$ is such that

$$\gamma(\langle \mathsf{a}, \mathsf{d}\rangle) \ge \gamma(\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}}\rangle)$$

where γ is the upper adjoint of the Galois insertion $(\gamma, C, \mathsf{A} \sqcap \mathsf{D}, \alpha)$ as in Definition 3.14.
By applying α to both expressions, by monotonicity of α we get

$$\alpha(\gamma(\langle \mathsf{a}, \mathsf{d} \rangle)) \geq \alpha(\gamma(\mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}} \rangle)))$$

and by Galois insertion properties, as $\alpha \circ \gamma$ is the identity function, we get

$$\langle \mathsf{a}, \mathsf{d} \rangle \geq \mathit{reduce}(\langle \hat{\mathsf{a}}, \hat{\mathsf{d}} \rangle)$$

$\square$

Theorem (3.28). *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let Δ_{A} *and* Δ_{D} *be pair-narrowing operators defined on the lattice* A *and* D*, respectively, such that* $\forall \langle \mathsf{a}, \mathsf{d} \rangle \in \mathsf{A} \sqcap \mathsf{D}$, $\forall \mathsf{a}' \in \mathsf{A}$, $\forall \mathsf{d}' \in \mathsf{D} : \langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the binary operator $\Delta : (\mathsf{A} \sqcap \mathsf{D}) \times (\mathsf{A} \sqcap \mathsf{D}) \to (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D} : \langle \mathsf{a}, \mathsf{d} \rangle \Delta \langle \mathsf{a}', \mathsf{d}' \rangle = \mathit{reduce}(\langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle)$ *is a pair-narrowing operator.*

Proof.

- *Bounding* We have to show that $\forall \langle \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{a}', \mathsf{d}' \rangle \in \mathsf{A} \sqcap \mathsf{D}$, $(\langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}', \mathsf{d}' \rangle) \Rightarrow (\langle \mathsf{a}, \mathsf{d} \rangle \leq \langle \mathsf{a}, \mathsf{d} \rangle \Delta \langle \mathsf{a}', \mathsf{d}' \rangle \leq \langle \mathsf{a}', \mathsf{d}' \rangle)$

$$\begin{array}{lclcll}
\langle \mathsf{a}, \mathsf{d} \rangle & \leq & \langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle & \leq & \langle \mathsf{a}', \mathsf{d}' \rangle & \text{by bounding of} \Delta_{\mathsf{A}} \text{ and} \Delta_{\mathsf{D}} \\
\langle \mathsf{a}, \mathsf{d} \rangle & \leq & \mathit{reduce}(\langle \mathsf{a} \Delta_{\mathsf{A}} \mathsf{a}', \mathsf{d} \Delta_{\mathsf{D}} \mathsf{d}' \rangle) & \leq & \langle \mathsf{a}', \mathsf{d}' \rangle & \text{by Lemma 3.3 and} \\
 & & & & & \text{Lemma 3.5 or Lemma 3.6} \\
\langle \mathsf{a}, \mathsf{d} \rangle & \leq & \langle \mathsf{a}, \mathsf{d} \rangle \Delta \langle \mathsf{a}', \mathsf{d}' \rangle & \leq & \langle \mathsf{a}', \mathsf{d}' \rangle & \text{by definition of} \Delta
\end{array}$$

- *Termination* Consider the increasing sequence $\langle \mathsf{a}_0, \mathsf{d}_0 \rangle \leq \langle \mathsf{a}_1, \mathsf{d}_1 \rangle \dots$ in $\mathsf{A} \sqcap \mathsf{D}$. As the ordering $\leq$ in $\mathsf{A} \sqcap \mathsf{D}$ is the same as in the cartesian product $\mathsf{A} \times \mathsf{D}$, we may consider the increasing sequence $\mathsf{a}_0 \leq \mathsf{a}_1 \leq \dots$ in A, and the increasing sequence $\mathsf{d}_0 \leq \mathsf{d}_1 \leq \dots$ in D. By the termination property ofΔ_{A} andΔ_{D}, we know that the corresponding sequences $\hat{\mathsf{a}}_0 = \mathsf{a}_0$, $\hat{\mathsf{a}}_{i+1} = \hat{\mathsf{a}}_i \Delta_{\mathsf{A}} \mathsf{a}_{i+1}$, and $\hat{\mathsf{d}}_0 = \mathsf{d}_0$, $\hat{\mathsf{d}}_{i+1} = \hat{\mathsf{d}}_i \Delta_{\mathsf{D}} \mathsf{d}_{i+1}$ stabilize after a finite number of terms.

  We show by induction that the increasing sequence

$$\langle \mathsf{a}'_0, \mathsf{d}'_0 \rangle = \langle \mathsf{a}_0, \mathsf{d}_0 \rangle, \ \langle \mathsf{a}'_{i+1}, \mathsf{d}'_{i+1} \rangle = \langle \mathsf{a}'_i, \mathsf{d}'_i \rangle \Delta \langle \mathsf{a}_{i+1}, \mathsf{d}_{i+1} \rangle$$

  is such that $\forall i : \langle \mathsf{a}'_i, \mathsf{d}'_i \rangle = \langle \hat{\mathsf{a}}_i, \hat{\mathsf{d}}_i \rangle$.

  The basis is trivial, as $\langle \mathsf{a}'_0, \mathsf{d}'_0 \rangle = \langle \mathsf{a}_0, \mathsf{d}_0 \rangle = \langle \hat{\mathsf{a}}_0, \hat{\mathsf{d}}_0 \rangle$.

  Induction step:

$$\begin{array}{lcll}
\langle \mathsf{a}'_{i+1}, \mathsf{d}'_{i+1} \rangle & = & \langle \mathsf{a}'_i, \mathsf{d}'_i \rangle \Delta \langle \mathsf{a}_{i+1}, \mathsf{d}_{i+1} \rangle & \text{by definition of } \{\langle \mathsf{a}'_j, \mathsf{d}'_j \rangle\}_{j \geq 0} \\
 & = & \mathit{reduce}(a'_i \Delta_{\mathsf{A}} \mathsf{a}_{i+1}, \mathsf{d}'_i \Delta_{\mathsf{D}} \mathsf{d}_{i+1}) & \text{by definition of} \Delta \\
 & = & \langle \hat{a}_i \Delta_{\mathsf{A}} \mathsf{a}_{i+1}, \hat{d}_i \Delta_{\mathsf{D}} \mathsf{d}_{i+1} \rangle & \text{by the hypothesis} \\
 & = & \langle \hat{\mathsf{a}}_{i+1}, \hat{\mathsf{d}}_{i+1} \rangle & \text{by definition of} \\
 & & & \{\hat{\mathsf{a}}_j\}_{j \geq 0} \text{ and } \{\hat{\mathsf{d}}_j\}_{j \geq 0}
\end{array}$$

It follows that $\{\langle \mathsf{a}'_j, \mathsf{d}'_j\rangle\}_{j\geq 0}$ converges in a finite number of steps, namely the maximum between the termination indexes of $\{\hat{\mathsf{a}}_j\}_{j\geq 0}$ and $\{\hat{\mathsf{d}}_j\}_{j\geq 0}$

□

Theorem (3.29)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let $\nabla_{*\mathsf{A}}$ *and* $\nabla_{*\mathsf{D}}$ *be set-widening operators defined on the lattice* A *and* D*, respectively, such that* $\forall S \subseteq \mathsf{A} \sqcap \mathsf{D}$, $\langle \nabla_{*\mathsf{A}}(\{\mathsf{a}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in S\}), \nabla_{\mathsf{D}}(\{\mathsf{d}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in S\})\rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the operator $\nabla_* : \wp(\mathsf{A} \sqcap \mathsf{D}) \nrightarrow (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall S \subseteq \mathsf{A} \sqcap \mathsf{D} : \nabla_*(\{S\}) =$ *reduce*$(\langle \nabla_{*\mathsf{A}}(\{\mathsf{a}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in S\}), \nabla_{*\mathsf{D}}(\{\mathsf{d}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in S\})\rangle)$ *is a set-widening operator.*

Proof. The proofs of this Theorems is similar to Theorem 3.27. □

Theorem (3.30)**.** *Let* $\mathsf{C}, \mathsf{A}, \mathsf{D}$ *be complete lattices, and let* $\mathsf{C} \underset{\alpha_{\mathsf{CD}}}{\overset{\gamma_{\mathsf{DC}}}{\leftrightarrows}} \mathsf{D}$ *and* $\mathsf{C} \underset{\alpha_{\mathsf{CA}}}{\overset{\gamma_{\mathsf{AC}}}{\leftrightarrows}} \mathsf{A}$ *be Galois insertions.*
Let $\Delta_{*\mathsf{A}}$ *and* $\Delta_{*\mathsf{D}}$ *be set-narrowing operators defined on the lattice* A *and* D*, respectively, such that* $\forall \mathsf{S} \subseteq \mathsf{A} \sqcap \mathsf{D}$, $\langle \Delta_{*\mathsf{A}}(\{\mathsf{a}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in \mathsf{S}\}), \Delta_{*\mathsf{D}}(\{\mathsf{d}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in \mathsf{S}\})\rangle \in \mathsf{A} \sqcap \mathsf{D}$.
Then the operator $\Delta_* : \wp(\mathsf{A} \sqcap \mathsf{D}) \nrightarrow (\mathsf{A} \sqcap \mathsf{D})$ *defined by* $\forall \mathsf{S} \subseteq \mathsf{A} \sqcap \mathsf{D} : \Delta_*(\{\mathsf{S}\})$ $=$ *reduce*$(\langle \Delta_{*\mathsf{A}}(\{\mathsf{a}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in \mathsf{S}\}), \Delta_{*\mathsf{D}}(\{\mathsf{d}_i \mid \langle \mathsf{a}_i, \mathsf{d}_i\rangle \in \mathsf{S}\})\rangle)$ *is a set-narrowing operator.*

Proof. The proofs of this Theorems is similar to Theorem 3.28. □

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 40–53, New York, NY, USA, 2000. ACM.

[3] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27:509–516, June 1978.

[4] Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, Course Notes on the WWW, 1997.

[5] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2:56–76, January 1980.

[6] Tania Armstrong, Kim Marriott, Peter Schachte, and Harald Søndergaard. Two classes of boolean functions for dependency analysis. *Sci. Comput. Program.*, 31:3–45, May 1998.

[7] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402:82–101, July 2008.

[8] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 64–84. Springer, 2010.

[9] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *ESORICS*, Lecture Notes in Computer Science, pages 197–221, 2005.

[10] Roberto Bagnara, Sara Bonini, Patricia M. Hill, Andrea Pescetti, Elisa Ricci, Angela Stazzone, Enea Zaffanella, and Tatiana Zolo. The parma polyhedra library user's manual, 2002.

[11] Roberto Bagnara, Patricia M. Hill, Elena Mazzi, and Enea Zaffanella. Widening operators for weakly-relational numeric abstractions. In C. Hankin and

I. Siveroni, editors, *Static Analysis: Proceedings of the 12th International Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 3–18, London, UK, 2005. Springer-Verlag, Berlin.

[12] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58:28–56, October 2005.

[13] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *Software Tools for Technology Transfer*, 8(4/5):449–466, 2006.

[14] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72:3–21, June 2008.

[15] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Applications of polyhedral computations to the analysis and verification of hardware and software systems. *Theor. Comput. Sci.*, 410:4672–4691, November 2009.

[16] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 213–229, London, UK, 2002. Springer-Verlag.

[17] Anindya Banerjee, Roberto Giacobazzi, and Isabella Mastroeni. What you lose is what you leak: Information leakage in declassification policies. *Electron. Notes Theor. Comput. Sci.*, 173:47–66, April 2007.

[18] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE workshop on Computer Security Foundations*, CSFW '02, pages 253–, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Gilles Barthe and Tamara Rezk. Non-interference for a jvm-like language. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 103–112, New York, NY, USA, 2005. ACM.

[20] Elliot David Bell and Leonard J. La Padula. Secure computer system: Unified exposition and multics interpretation, 1976.

[21] John L. Bell and Moshe Machover. *A Course on Mathematical Logic (Universitext).* Springer-Verlag TELOS, Santa Clara, CA, USA, 1 edition, 2008.

[22] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 04 1977.

[23] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying facets of information integrity. In *Proceedings of the 6th international conference on Information systems security*, ICISS'10, pages 48–65, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *SIGPLAN Not.*, 38:196–207, May 2003.

[25] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proceedings of the 6th International Conference on Parallel Computing Technologies*, PaCT '01, pages 27–41, London, UK, 2001. Springer-Verlag.

[26] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281:109–130, June 2002.

[27] Chiara Braghin and Agostino Cortesi. Flow-sensitive leakage analysis in mobile ambients. *Electron. Notes Theor. Comput. Sci.*, 128:17–25, May 2005.

[28] Chiara Braghin, Agostino Cortesi, and Riccardo Focardi. Information flow security in boundary ambients. *Inf. Comput.*, 206:460–489, February 2008.

[29] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[30] Matteo Centenaro, Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Type-based analysis of pin processing apis. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 53–68, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. An abstract domain to discover interval linear equalities. In G. Barthe and M. Hermenegildo, editors, *11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*, volume 5944 of *LNCS*, pages 112–128, Madrid,Spain, January 2010. Springer.

[32] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 198–209, New York, NY, USA, 2004. ACM.

[33] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[34] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Trans. Program. Lang. Syst.*, 21:948–976, September 1999.

[35] Agostino Cortesi. Widening operators for abstract interpretation. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40, Washington, DC, USA, 2008. IEEE Computer Society.

[36] Agostino Cortesi, Gilberto Filé, Francesco Ranzato, Roberto Giacobazzi, and Catuscia Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19:7–47, January 1997.

[37] Agostino Cortesi, Gilberto Filé, and William Winsborough. The quotient of an abstract interpretation. *Theor. Comput. Sci.*, 202:163–192, July 1998.

[38] Agostino Cortesi, Gilberto Filé, and William H. Winsborough. Prop revisited: Propositional formula as abstract domain for groundness analysis. In *LICS*, pages 322–327, 1991.

[39] Agostino Cortesi, Gilberto Filé, and William H. Winsborough. Optimal groundness analysis using propositional logic. *J. Log. Program.*, 27(2):137–167, 1996.

[40] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Sci. Comput. Program.*, 38:27–71, August 2000.

[41] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Comput. Lang. Syst. Struct.*, 37:24–42, April 2011.

[42] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *ICFEM*, Lecture Notes in Computer Science. Springer, 2011.

[43] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French).* Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.

[44] Patrick Cousot. The Calculational Design of a Generic Abstract Interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design.* NATO ASI Series F. IOS Press, Amsterdam, 1999.

[45] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 7–9, New York, NY, USA, 2007. ACM.

[46] Patrick Cousot. Lecture notes of course: "interpétation abstraite: application à la vérification et à l'analyse statique", 2009.

[47] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Poaris, France, 1976.

[48] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[49] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[50] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[51] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[52] Patrick Cousot and Rahida Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13:103–179, July 1992.

[53] Patrick Cousot and Rardhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[54] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[55] Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of information flow security with mutable state. *J. Funct. Program.*, 15:249–291, March 2005.

[56] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[57] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[58] Vijay D'Silva. Widening for automata. In *PhD thesis, Institut fur Informatik, Universitaat Zurich*, 2006.

[59] Vijay D'Silva, Mitra Purandare, and Daniel Kroening. Approximation refinement for interpolation-based model checking. In *Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation*, VMCAI'08, pages 68–82, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] Logozzo F. *Modular static analysis of object-oriented languages.* PhD thesis, École Polytechnique, Paris, France, 15 June 2004.

[61] Jérôme Feret. Static analysis of digital filters. In David A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[62] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control.* Artech House, Inc., Norwood, MA, USA, 2003.

[63] Pietro Ferrara. A fast and precise alias analysis for data race detection. In *Proceedings of the Third Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, April 2008.

[64] Pietro Ferrara. *Static analysis via abstract interpretation of multithreaded programs.* PhD thesis, Ecole Polytechnique of Paris (France) and University "Ca' Foscari" of Venice (Italy), May 2009.

[65] Pietro Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 6117 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 2010.

[66] Riccardo Focardi and Matteo Centenaro. Information flow security of multi-threaded distributed programs. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 113–124, New York, NY, USA, 2008. ACM.

[67] Raphael Fuchs. Interfacing tvla and sample. Bachelor thesis, August 2011.

[68] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 186–197, New York, NY, USA, 2004. ACM.

[69] Roberto Giacobazzi and Isabella Mastroeni. Adjoining declassification and attack models by abstract interpretation. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 2005.

[70] Roberto Giacobazzi and Francesco Ranzato. The reduced relative power operation on abstract domains. *Theor. Comput. Sci.*, 216:159–211, March 1999.

[71] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[72] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[73] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[74] Philippe Granger. Improving the results of static analyses programs by local decreasing iteration. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 68–79, London, UK, 1992. Springer-Verlag.

[75] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. *SIGPLAN Not.*, 41:376–386, June 2006.

[76] Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM.

[77] Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Evaluation of the domain *prop*. *J. Log. Program.*, 23(3):237–278, 1995.

[78] Tae hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung goo Doh. A practical string analyzer by the widening approach. In *Asian Symposium on Programming Languages and Systems*, pages 374–388, 2006.

[79] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23:396–450, May 2001.

[80] IBM Inc. CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors. Technical report, 2006. Relases 2.53–3.27.

[81] Bertrand Jeannet. Convex polyhedra library, March 2002. Documentation of the “New Polka” library available at `http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html`.

[82] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, June 2009.

[83] Rajeev Joshi, K. Leino, and M. Rustan. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37:113–138, May 2000.

[84] Samuel N. Kamin and Uday S. Reddy. *Two semantic models of object-oriented languages*, pages 463–495. MIT Press, Cambridge, MA, USA, 1994.

[85] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[86] Herald Søndergaard Kim Marriott. Notes for a tutorial on abstract interpretation of logic programs. In *Foundations of Logic Programming*, Berlin, 1989. Springer-Verlag.

[87] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16:613–615, October 1973.

[88] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP ’01, pages 77–91, London, UK, 2001. Springer-Verlag.

[89] Peng Li and Steve Zdancewic. Unifying Confidentiality and Integrity in Downgrading Policies. In *Proc. of Foundations of Computer Security Workshop (FCS)*, 2005.

[90] Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411:1974–1994, April 2010.

[91] Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society.

[92] Vincent Loechner, Bd. S. Brant, and F-Illkirch. Polylib: A library for manipulating parameterized polyhedra, 1999.

[93] Francesco Logozzo and Manuel Fahndrich. A weakly relational domain for the efficient validation of array accesses. In *23th ACM Symposium on Applied Computing (SAC 2008), Fortaleza, Brazil*, 2008.

[94] Gavin Lowe. Quantifying information flow. In *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*, pages 18–31. IEEE Computer Society, 2002.

[95] Alexander Lux and Heiko Mantel. Formal aspects in security and trust. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, chapter Who Can Declassify?, pages 35–49. Springer-Verlag, Berlin, Heidelberg, 2009.

[96] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[97] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc of ESORICS'09*. Lecture Notes in Computer Science, 2009.

[98] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.

[99] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[100] H. Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 129–145. Springer-Verlag, November 2004.

[101] Kim Marriott and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Lett. Program. Lang. Syst.*, 2:181–196, March 1993.

[102] Isabella Mastroeni and Roberto Giacobazzi. An abstract interpretation-based model for safety semantics. *Int. J. Comput. Math.*, 88(4):665–694, 2011.

[103] Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[104] Jose S. Metos and John V. Oldfield. Binary decision diagrams: From abstract representations to physical implementations. In *Proceedings of the 20th Design Automation Conference*, DAC '83, pages 567–570, Piscataway, NJ, USA, 1983. IEEE Press.

[105] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the Second Symposium on Programs as Data Objects*, PADO '01, pages 155–172, London, UK, 2001. Springer-Verlag.

[106] Antoine Miné. The octagon abstract domain. In *Proc. of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, IEEE, pages 310–319. IEEE CS Press, October 2001. `http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf`.

[107] Antoine Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[108] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, March 2006.

[109] Masaaki Mizuno and David A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Asp. Comput.*, 4:727–754, 1992.

[110] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[111] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31:129–142, October 1997.

[112] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14:157–196, April 2006.

[113] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nate Nystrom. Jif: Java information flow. software release., July 2001-2004.

[114] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[115] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[116] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[117] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing, 1992.

[118] Peter Ørbæk. Can you trust your data? In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 575–589, London, UK, 1995. Springer-Verlag.

[119] Jens Palsberg and Peter Ørbæk. Trust in the lambda-calculus. In *Proceedings of the Second International Symposium on Static Analysis*, SAS '95, pages 314–329, London, UK, 1995. Springer-Verlag.

[120] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25:117–158, January 2003.

[121] Viswanath Ramachandran, Pascal Van Hentenryck, and Agostino Cortesi. Abstract domains for reordering clp(rlin) programs. *J. Log. Program.*, 42(3):217–256, 2000.

[122] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[123] Dorothy Elizabeth Robling Denning. *Cryptography and data security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[124] Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 92–106, 2009.

[125] Leino Rustan and Logozzo Francesco. Using widenings to infer loop invariants inside an smt solver, or: A theorem prover as abstract domain. 2007.

[126] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[127] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin / Heidelberg, 2004.

[128] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[129] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.

[130] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, May 2002.

[131] Roman Scheidegger. Translating java bytecode to simple. Bachelor thesis, June 2010.

[132] David A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In *Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science, pages 1–18, 1995.

[133] David A. Schmidt. Abstract interpretation of small-step semantics. In *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, volume 1192 of *Lecture Notes in Computer Science*, pages 76–99, 1996.

[134] David A. Schmidt. Programming language semantics. In *The Computer Science and Engineering Handbook*, pages 2237–2254. CRC Press, 1997.

[135] David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp Symb. Comput.*, 10:237–271, May 1998.

[136] David A. Schmidt. Abstract interpretation from a denotational-semantics perspective. *Electronic Notes of Theoretical Computer Science*, 249:19–37, August 2009.

[137] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003. ©INRIA.

[138] Christian Skalka and Scott Smith. Static enforcement of security with types. *SIGPLAN Not.*, 35:34–45, September 2000.

[139] Geoffrey Smith. Principles of Secure Information Flow Analysis. In *Malware Detection*, pages 297–307, 2007.

[140] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.

[141] Robert F. Stark, E. Borger, and Joachim Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[142] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.

[143] Terkel K. Tolstrup, Flemming Nielson, and H. Riis Nielson. Information flow analysis for VHDL. In *Parallel Computing Technoligies*, Lecture Notes in Computer Science. Springer, 2005.

[144] Stanford University. Stanford SecuriBench Micro. http://suif.stanford.edu/~livshits/work/securibench-micro/.

[145] Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type analysis of prolog using type graphs. *SIGPLAN Not.*, 29:337–348, June 1994.

[146] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* Elsevier and MIT Press, 1990.

[147] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of the Third International Symposium on Static Analysis*, pages 366–382, London, UK, 1996. Springer-Verlag.

[148] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[149] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 268–276, New York, NY, USA, 2000. ACM.

[150] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[151] Matteo Zanioli and Agostino Cortesi. Information leakage analysis by abstract interpretation. In *Proceedings of the 37th international conference on Current trends in theory and practice of computer science (SOFSEM)*, volume 6543 of *Lecture Notes in Computer Science*, pages 545–557. Springer-Verlag, 2011.

[152] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. Sails: Static analysis of information leakage with sample. In *Proceedings of the 2012 ACM symposium on Applied Computing*, SAC '12, New York, NY, USA, 2012. ACM. To appear.

[153] Mirko Zanotti. Security typings by abstract interpretation. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 360–375, London, UK, 2002. Springer-Verlag.

[154] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20:283–328, August 2002.