PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure

Static Analysis by Abstract Interpretation of Quantitative Program Properties

Analyse Statique par Interprétation Abstraite de Propriétés Quantitatives de Programmes

Soutenue par
Denis MAZZUCATO
Le 10 décembre 2024

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Thomas JENSEN INRIA	*Rapporteur*
Isabella MASTROENI University of Verona	*Rapporteuse*
Bor-Yuh Evan CHANG University of Colorado Boulder & Amazon	*Examinateur*
Bernadette CHARRON-BOST CNRS & École Normale Supérieure | PSL	*Examinatrice*
Matthieu MARTEL Université de Perpignan	*Examinateur*
Antoine MINÉ Sorbonne Université	*Examinateur*
Corina S. PĂSĂREANU Carnegie Mellon University & NASA Ames	*Examinatrice*
Caterina URBAN INRIA & École Normale Supérieure | PSL	*Directrice de thèse*

ENS | PSL

Abstract

The aim of this thesis is to develop mathematically sound and practically efficient methods for improving the reliability of software systems. Our approach is based on abstract interpretation, a formal framework to systematically design approximate program behaviors. We focus on the development of a quantitative framework for measuring the impact of input variables on the program behavior. Advances in the understanding of input data usage in software systems produce a more correct, performant, and secure software, particularly in safety-critical systems where these factors are non-negotiable.

To achieve this goal, we propose a novel quantitative input usage framework to discriminate between input variables based on their influence on the program behavior. This framework is flexible, parametrized in the notion of impact to suit various needs, thereby providing a method for both certification of intended behaviors and identification of potential flaws. By employing abstract interpretation, our results are guaranteed to be sound, meaning that the quantified bounds on the impact of input variables are always greater (or always lower depending on the underlying over- or under-approximation) than the actual impact. The major challenge, however, is to ensure that the quantification is precise enough to be useful in practice, while still being computationally efficient.

The quantitative framework is applied to verify both extensional and intensional properties of software. Extensional properties are based on the input-output behavior of programs, while intensional properties concern how the computations are performed. Specifically, the extensional property measures how variations in input data influence program output, whereas the intensional property assesses the impact of input data on the number of iterations within the program's loops during execution. We quantify the impact of input variables by analyzing input-output dependencies in the program under evaluation.

To demonstrate the practical applicability of this framework, we implemented it into three distinct tools: Libra, Impatto, and TimeSec, each designed for a specific type of quantitative analysis. Libra focuses on fairness of neural networks, Impatto is tailored for general-purpose software reliability analysis, and TimeSec specializes in assessing side-channel vulnerabilities. The effectiveness of these tools was validated through extensive experimental evaluations across diverse use cases, from the quantification of biased space in neural networks to the detection of timing-based side-channel attacks in cryptographic libraries. Libra's application in neural network analysis has provided insights into how input features influence model behavior, contributing to the development of more reliable machine learning systems. TimeSec has been shown to effectively quantify the impact of input variables in the Amazon Web Services s2n-bignum library, enabling the identification of critical variables that may affect system stability or security.

Overall, the quantitative framework proposed in this thesis advances the understanding of input data usage in software systems, providing theoretical framework based on abstract interpretation and practical tools for quantifying the impact of input data. The experimental results highlight the utility of our quantitative framework, demonstrating its potential.

Résumé

The aim of this thesis is to develop mathematically sound and practically efficient methods for improving the reliability of software systems. Our approach is based on abstract interpretation, a formal framework to systematically design approximate program behaviors. We focus on the development of a quantitative framework for measuring the impact of input variables on the program behavior. Advances in the understanding of input data usage in software systems produce a more correct, performant, and secure software, particularly in safety-critical systems where these factors are non-negotiable.

To achieve this goal, we propose a novel quantitative input usage framework to discriminate between input variables based on their influence on the program behavior. This framework is flexible, parametrized in the notion of impact to suit various needs, thereby providing a method for both certification of intended behaviors and identification of potential flaws. By employing abstract interpretation, our results are guaranteed to be sound, meaning that the quantified bounds on the impact of input variables are always greater (or always lower depending on the underlying over- or under-approximation) than the actual impact. The major challenge, however, is to ensure that the quantification is precise enough to be useful in practice, while still being computationally efficient.

The quantitative framework is applied to verify both extensional and intensional properties of software. Extensional properties are based on the input-output behavior of programs, while intensional properties concern how the computations are performed. Specifically, the extensional property measures how variations in input data influence program output, whereas the intensional property assesses the impact of input data on the number of iterations within the program's loops during execution. We quantify the impact of input variables by analyzing input-output dependencies in the program under evaluation.

To demonstrate the practical applicability of this framework, we implemented it into three distinct tools: Libra, Impatto, and TimeSec, each designed for a specific type of quantitative analysis. Libra focuses on fairness of neural networks, Impatto is tailored for general-purpose software reliability analysis, and TimeSec specializes in assessing side-channel vulnerabilities. The effectiveness of these tools was validated through extensive experimental evaluations across diverse use cases, from the quantification of biased space in neural networks to the detection of timing-based side-channel attacks in cryptographic libraries. Libra's application in neural network analysis has provided insights into how input features influence model behavior, contributing to the development of more reliable machine learning systems. TimeSec has been shown to effectively quantify the impact of input variables in the Amazon Web Services s2n-bignum library, enabling the identification of critical variables that may affect system stability or security.

Overall, the quantitative framework proposed in this thesis advances the understanding of input data usage in software systems, providing theoretical framework based on abstract interpretation and practical tools for quantifying the impact of input data. The experimental results highlight the utility of our quantitative framework, demonstrating its potential.

Contents

List of Figures

List of Tables

A Template with a Wide Margin

For this manuscript, we have used the Kaobook class,[1] a template featuring a wide margin specifically designed for writing books and graduate-level theses, based on Ken Arroyo Ohori's doctoral thesis[2] and on the Tufte-LaTeX class,[3] offering a layout that is both elegant and practical.

Why the narrow main text column? It is all about the wide margin: an expansive playground for notes, code snippets, figures, tables, citations, and all those additional details that enhance the narrative, as Figure 1 shows. This layout lets the main narrative take the spotlight while all the extra bits are placed in the margin, where they can be easily accessed without disrupting the flow of the text. It is like having a personal assistant handing you the information right when you need it. Larger tables and figures that would not fit in the margin are placed in the main text column. Captions follow in the margin and comply with the usual convention: above for tables and below for figures. Even larger elements may span the entire page along with their captions.

With over 200 references in this manuscript, the use of side references becomes a lifesaver. The hassle of flipping back and forth between the text and the bibliography to remember which citation matches [42] is a thing you will not endure in this thesis. Speaking of citations, as every computer scientist knows, "42" is the answer of Life, the Universe and Everything, according to Douglas Adams' *The Hitchhiker's Guide to the Galaxy*.[4] Right on point, the 42nd citation (in citation order) is where this thesis began. Written by my supervisor Caterina Urban in 2018, [42] is the foundational work for the property of input data usage, unifying existing literature on information flow analyses within the formal framework of abstract interpretation [36]. In essence, [42] inspired my academic journey and the last four years of my life.

The template also allows for short restatements of definitions and theorems, linked to their original appearances. Readers will not need to flip through the pages of this manuscript – once a concept is introduced, it conveniently pops up in the margin whenever relevant again. Sometimes, due to multiple elements in the margin, the side content may be not perfectly aligned with its use in the text, but thanks to the fluid layout, the margin content is automatically positioned to best fit the space available. Recalling what Definition 2.1.4 was has never been easier, just a look in the margin. For the PDF version of this thesis, you can also click on the title of the margin definition to jump to its original appearance in the text. This keeps the discussion clear and accessible, which is especially valuable in a complex and detailed work like a PhD thesis. In short:

this wide-margin template makes navigating through these pages a bit easier, and hopefully, a bit more enjoyable too.

1: github.com/fmarotta/kaobook

2: 3d.bk.tudelft.nl/ken/en/thesis

3: ctan.org/pkg/tufte-latex

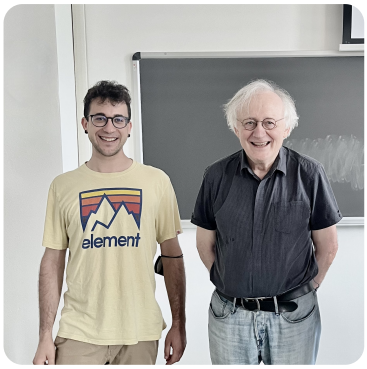


Figure 1: Me (left) with Patrick Cousot (right) co-inventor of abstract interpretation [36].

4: en.wikipedia.org/wiki/Phrases_from_The_Hitchhiker%27s_Guide_to_the_Galaxy#The_Answer_to_the_Ultimate_Question_of_Life,_the_Universe,_and_Everything_is_42

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

[36]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'

Def. 2.1.4 (Complete Lattice) *A complete lattice* $\langle X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ *is a lattice where for all subsets* $X' \subseteq X$*, the least upper bound* $\sqcup X'$ *and the greatest lower bound* $\sqcap X'$ *exist.*

Introduction

1 Introduction

In today's world, software rules safety-critical systems such as nuclear power plants [1], car engines [2], airplane control systems [3], and medical devices [4]. The reliability of these systems is based on their correctness. Any failure or vulnerability in safety-critical software poses significant risks, potentially causing financial losses or threatening human safety. Recent advances in software development have led to increasingly complex software systems. As software grows in complexity, the likelihood of bugs also increases. The effort and cost of fixing these bugs escalate with late detection, as illustrated in Table 1.1 [5], which shows the relative costs of fixing bugs at different development stages. For safety-critical systems, detecting and addressing bugs before deployment is mandatory to ensure safety.

Beyond traditional software, machine learning-based software is increasingly being integrated into safety-critical systems, presenting new challenges to traditional assurance processes [6]. Despite these challenges, the high standards for correctness in such systems remain the same, regardless of whether the software is traditional or machine learning-based. Additionally, machine learning is also influencing societal decision-making in areas such as social welfare [7], criminal justice [8], and healthcare [9]. However, recent cases have demonstrated that machine learning software can inadvertently reproduce or even amplify biases present in the training data [7–10]. In response of these risks, the European Commission proposed the Artificial Intelligence Act [11], which aims to establish the first legal framework for machine learning software, with strict requirements to minimize the potential for discriminatory outcomes. Therefore, detecting bugs and biases in machine learning software is crucial to ensure that it behaves as intended and does not cause harm to users.

Overall, from more traditional to the latest machine learning-based software, detecting errors in safety-critical or high-stakes contexts is mandatory to ensure a correct software behavior.

1.1 Software Quality

Software quality measures how well software meets its requirements. The most common method to ensure software quality is *testing*. The correct behavior of software is tested empirically for a finite number of inputs against a set of assertions specifying the functional requirements of the code. However, testing has inherent limitations. For instance, exhaustive testing is impractical, and constraints on time and budget can further impact the process. Mostly, testing cannot guarantee the absence of bugs. [1] While in some cases deploying software with bugs and relying on patches to fix it is acceptable, cf. Table 1.1, ensuring that software is free of bugs before deployment is necessary to avoid catastrophic consequences in safety-critical systems.

[1]: Krasner (2021), 'The cost of poor software quality in the US: A 2020 report'

[2]: Finch (2009), 'Toyota sudden acceleration: a case study of the national highway traffic safety administration-recalls for change'

[3]: Briere et al. (1993), 'AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems'

[4]: Leveson et al. (1993), 'Investigation of the Therac-25 Accidents'

[5]: White et al. (2017), 'Formal verification: will the seedling ever flower?'

Table 1.1: Cost of fixing bugs at different development stages [5].

Bug Found at Stage	Cost to Fix
Requirements	1x (definition)
Architecture	3x
Design	5-10x
System Test	10x
Production	10-100x

[6]: Goodloe (2023), 'Assuring Safety-Critical Machine Learning-Enabled Systems: Challenges and Promise'

[7]: Larson et al. (2016), *How We Analyzed the COMPAS Recidivism Algorithm*

[8]: Buolamwini et al. (2018), 'Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification'

[9]: Obermeyer et al. (2019), 'Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations'

[10]: Kay et al. (2015), 'Unequal Representation and Gender Stereotypes in Image Search Results for Occupations'

[11]: European Commission (2021), *Proposal for a Regulation Laying Down Harmonised Rules on Artificial Intelligence (Artificial Intelligence Act)*

1: "Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence." – Dijkstra et al. [12]

[12]: Dijkstra et al. (1976), *A discipline of programming*

In contrast to testing, *formal methods* provide rigorous mathematical guarantees about software correctness. The idea of formally verifying software dates back to the late 1960s with program proofs and invariants from Floyd [13] and Hoare [14]. Even earlier, formal methods can be traced to the work of Church [15] and Turing [16] on the foundations of computation. According to software engineering practices, formal methods should be introduced early in the development lifecycle, enabling the verification of software properties at the design stage.

Why should formal methods complement other well-known, widely accepted, and user-friendly techniques such as testing?

To answer this question, consider the following examples where testing failed:

- The Ariane 5 rocket failure in 1996, caused by an integer overflow bug, resulted in a loss of $370 million.[2]
- The Therac-25 radiation therapy machine malfunctioned due to software bugs, resulting in patient deaths and serious injuries [4].
- The Toyota unintended acceleration case, where a stack overflow resulted in the death of 89 people and a lawsuit of $1.2 billion.[3]
- A round-off error in the Patriot missile system caused the death of 28 people during the Gulf War.[4]

These cases could have been avoided with formal methods.

Unlike testing, formal methods enable exhaustive search and detection of bugs that testing may miss. System requirements are translated into formal specifications, which are mathematically verified to ensure the system's behavior aligns with real-world scenarios. However, Rice's undecidability theorem [17] states that all non-trivial program properties [5] are undecidable, meaning there is no always terminating algorithm that can decide whether any program satisfies a non-trivial program property. Consequently, formal methods either sacrifice: *automation*, *soundness/termination*, or *completeness*. Based on this, formal methods are classified into three categories [18] (cf. Figure 1.1): *theorem proving*, *model checking/symbolic execution*, and *static analysis*.

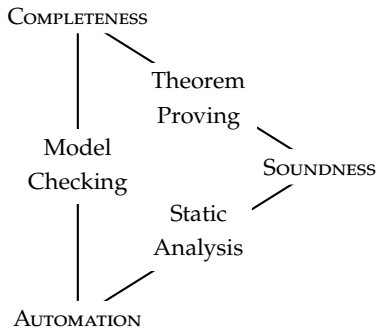


Figure 1.1: Trade-offs in formal methods.

Theorem Proving. Theorem provers [19] produce proofs of correctness using interactive tools, also called proof assistants, such as Coq [20], HOL Light [21], Lean [22], Isabelle [23], and Agda [24]. Theorem provers are complete and sound but not fully automated. Indeed, user interaction is ultimately required to guide the proof search. There are some attempts to automate theorem proving [25] based on proof search strategies where the prover automatically search for the proof. Nevertheless, the user interaction is still required whenever the proof search fails. In this category of complete and sound methods also fall proof-oriented programming languages, such as Dafny [26], F* [27], and Why3 [28].

Model Checking. Formal methods based on model checking [29, 30] automatically explore the state space of a program's model to verify whether undesirable states are reachable. Modern software model checkers trade termination for automation and completeness, as state-of-the-art

[13]: Floyd (1967), 'Assigning Meanings to Programs'

[14]: Hoare (1969), 'An Axiomatic Basis for Computer Programming'

[15]: Church (1936), 'A Note on the Entscheidungsproblem'

[16]: Turing (1937), 'On computable numbers, with an application to the Entscheidungsproblem'

2: esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[4]: Leveson et al. (1993), 'Investigation of the Therac-25 Accidents'

3: www.embeddedrelated.com/showarticle/1574.php

4: www.ima.umn.edu/~arnold/disasters/patriot.html

[17]: Rice (1953), 'Classes of recursively enumerable sets and their decision problems'

5: A property is trivial if it is true for all programs or false for all programs.

[18]: Cousot et al. (2010), 'A gentle introduction to formal verification of computer systems by abstract interpretation'

[19]: Nawaz et al. (2019), 'A Survey on Theorem Provers in Formal Methods'

[20]: Bertot et al. (2004), *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*

[21]: Harrison (2009), 'HOL Light: An Overview'

[22]: Moura et al. (2021), 'The Lean 4 Theorem Prover and Programming Language'

[23]: Wenzel et al. (2008), 'The Isabelle Framework'

[24]: Bove et al. (2009), 'A Brief Overview of Agda - A Functional Language with Dependent Types'

[25]: Sutcliffe et al. (2001), 'Evaluating general purpose automated theorem proving systems'

[26]: Leino (2010), 'Dafny: An Automatic Program Verifier for Functional Correctness'

software model checkers may not terminate. When the answer is *unknown*, indicating that the verification process halted on a timeout, the property may still be satisfied or violated, but the model checker was not able to determine it as the search space was too large. However, model checking is complete because when the model checker answers that the property has been violated, a counterexample is provided. Clarke, Kroening, and Lerda [31] applied model checking to prove the correctness of ANSI-C programs.

[27]: Swamy et al. (2016), 'Dependent types and multi-monadic effects in F'

[28]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[29]: Clarke et al. (1981), 'Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic'

[30]: Queille et al. (1982), 'Specification and verification of concurrent systems in CESAR'

[31]: Clarke et al. (2004), 'A Tool for Checking ANSI-C Programs'

[32]: Baldoni et al. (2018), 'A Survey of Symbolic Execution Techniques'

Symbolic Execution. Symbolic execution [32] approaches perform abstract execution of programs to collect constraints on the program execution. During the abstract computation, variables with unknown values are represented symbolically and propagated through the program. The collected constraints are then solved to verify whether an assertion is violated or not. As model checking, symbolic execution trades soundness for automation and completeness. Tools like KLEE [33], BinSec [34], and PathFinder [35] are based on symbolic execution.

[33]: Cadar et al. (2008), 'KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs'

[34]: David et al. (2016), 'BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis'

[35]: Pasareanu et al. (2010), 'Symbolic PathFinder: symbolic execution of Java bytecode'

Static Analysis. Static analysis verifies the program source code at some level of abstraction without user interaction. This abstraction is sound but incomplete, meaning the analysis may report *false alarms*, *i.e.*, warnings that a correct program may be incorrect. However, whenever the static analysis certifies the absence of a bug, the program is indeed bug-free. In formal methods, the most common static analysis techniques are based on *abstract interpretation*.

Abstract Interpretation. Abstract Interpretation [36] is a general theory for approximating program semantics, developed by Patrick and Radhia Cousot in the late 1970s (see [37] for an historical view and [38] for a comprehensive overview). Their framework is based on the observation that not all computational details are necessary to reason about program properties. Instead, the program's semantics can be approximated by a simpler, more abstract model that facilitates automatic reasoning.

[36]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'

[37]: Cousot (2024), 'A Personal Historical Perspective on Abstract Interpretation'

[38]: Cousot (2021), *Principles of abstract interpretation*

Over the past decade, abstract interpretation-based static analyses became part of the software development lifecycle of safety-critical systems. For instance, the Astrée static analyzer [39] is routinely used to ensure the absence of runtime errors in embedded synchronous C programs by Airbus.

[39]: Blanchet et al. (2003), 'A static analyzer for large safety-critical software'

We provide a formal introduction to abstract interpretation in Chapter 2. At the end of that chapter, we will illustrate the main results on a small idealized programming language.

1.2 Input Data Usage

Programming errors in software systems do not always result in crashes or runtime errors. Sometimes, faulty programs produce plausible yet erroneous outcomes or unsafe behaviors. Such bugs are hard to spot since they provide no clear indication that something went wrong. A potential and common source of such errors is the misuse of input data,

i.e., when an input variable has an unexpected impact on the program computation compared to the developers' expectations.

A notable example is the Reinhart and Rogoff article "Growth in a Time of Debt" [40], which claimed that economic growth is negatively correlated with public debt. This article was heavily cited to justify austerity measures worldwide in the following years. However, in 2013, Herndon, Ash, and Pollin [41] discovered that the authors had made a mistake in their Excel spreadsheet, leading to the erroneous conclusion. One of the errors was the incorrect usage of the input value relative to Norway's economic growth in 1964, with an excessive weight in the average growth rate computation.

[40]: Reinhart et al. (2010), 'Growth in a Time of Debt'

[41]: Herndon et al. (2014), 'Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff'

In data science and machine learning applications, where software involves long pipelines that filter, merge, and manipulate data, programming errors causing input variables to have more or less influence than expected are even more likely to occur. Hence, it is essential to employ techniques that enhance confidence in the usage of input variables for data-driven applications.

In Part 'Background', Chapter 3 formally introduces input data usage, a program property that captures the *qualitative* usage of input data in a program, as proposed by Urban and Müller [42]. We provide a hierarchy of semantics that precisely captures the input data usage property, abstracting unnecessary details. Finally, we report an abstract semantics that captures syntactic dependencies between variables from Urban and Müller [42, Section 10], used to approximate the input data usage property soundly.

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

Contributions In Chapter 3, we extend the original definition of the input data usage to capture abstractions of output values, similarly to abstract non-interference [43], discussing relations between the two properties. Such extension blends abstract non-interference with input data usage, providing a definition that works for non-deterministic programs natively without the need of a specialized program semantics.

[43]: Giacobazzi et al. (2018), 'Abstract Non-Interference: A Unifying Framework for Weakening Information-flow'

1.3 Quantitative Properties

Typically, program properties are qualitative: a program either satisfies a property or not. This is not always sufficient for capturing the complexity of real-world requirements [44]. For instance, in program security, one fundamental requirement is protecting sensitive information confidentiality. Secure information flow analysis questions whether a program could leak information about its secrets. Non-interference, certifying that a program reveals no information about its secrets, is a classic secure information flow approach. However, non-interference is too strict for many practical applications. For example, in a digital election protocol, individual votes should be anonymous, but the final result needs to be revealed. A password checker should not reveal the password but should indicate whether the password is correct. These cases represent deliberate violations of non-interference necessary for the program to fulfill its purpose.

[44]: Smith (2007), 'Principles of Secure Information Flow Analysis'

[45]: Smith (2009), 'On the Foundations of Quantitative Information Flow'

To address this limitation, quantitative properties are considered [45]. The key idea is to accept that a program may violate a property and compare such violation against a threshold. Programs are classified as *safe* or *unsafe* based on the degree of violation, inducing a classification among programs based on how much safety they provide.

In the Reinhart and Rogoff case, the error was not whether the value of Norway's economic growth was used in the average computation but rather how much it was used, *i.e.*, its impact was much higher than it should have been. A quantitative analysis would have revealed that the impact of Norway's economic growth was too high, allowing the authors to correct the wrong conclusion.

1.3.1 Quantitative Verification of Extensional Properties

Chapter 4 and Chapter 5 in Part 'Quantitative Verification of Extensional Properties' are based on the work published at NASA Formal Methods Symposium (NFM) 2024 [46].

[46]: Mazzucato et al. (2024), 'Quantitative Input Usage Static Analysis'

In the first part of this thesis, we propose semantics-based static analysis techniques to quantify the impact of input variables on program computation. In Chapter 4, we introduce a new formal framework based on abstract interpretation for reasoning about quantitative input data usage properties. This framework can identify variables with disproportionate impact, certifying intended behavior or revealing potential flaws by matching developers' expectations with actual results. We characterize the impact of an input variable with a notion of dependency between variables and program outcomes. Our framework is parametric in the impact definition of choice. We introduce a backward static analysis based on abstract interpretation, parametric in the impact definition, which infers a sound over-approximation of the impact of input variables. This approach allows end-users to choose the impact that fits their needs, ensuring a targeted and customizable analysis.

Chapter 5 demonstrates the quantitative framework's potential applications by evaluating our static analysis tool, Impatto,[6] on six use cases.

6: `github.com/denismazzucato/impatto`

Chapter 6 and Chapter 7 in Part 'Quantitative Verification of Extensional Properties' are based on the work published at the 28th Static Analysis Symposium (SAS) 2021 [47].

[47]: Mazzucato et al. (2021), 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'

In Chapter 6, we extend the quantitative input data usage property to neural networks. We propose two impact quantifiers for neural networks, addressing the challenges arising from the highly non-linear input space and measuring input features' fairness. We refine the backward analysis to exploit parallel computations for better performance, employing a combination of forward and backward analysis. The forward pass reduces the backward analysis's combinatorial explosion, partitioning the input space into subregions for easier parallelization. Chapter 7 demonstrates the effectiveness of our approach on neural networks, showing that our method can identify input features with a disproportionate impact on the network's output, or quantify the fairness of neural networks.

Contributions The followings are the contributions of Part 'Quantitative Verification of Extensional Properties', consisting of Chapters 4 to 7:

- In Chapter 4, we develop a theoretical framework by abstract interpretation to quantify the impact of input variables by considering three novel impact quantifiers.
- In Chapter 4, we present our static analysis and a possible abstract implementation of the impact instances.

- In Chapter 5 our tool called Impatto showcases our approach against a set of six demonstrative programs.
- An artifact of Impatto comprehensive of Impatto's source code, documentation, and evaluation benchmarks is available on Zenodo. [7]
- In Chapter 6, we present two novel impact quantifiers for neural networks.
- In Chapter 6, we propose an improved backward analysis that exploits parallel computations and achieves better precision by combining abstract domains.
- In Chapter 7, we extended Impatto and Libra [8] tools to evaluate our quantitative analysis on neural networks.
- An artifact extending Libra with the quantitative evaluation of fairness is available on Zenodo.[9]

7: doi.org/10.5281/zenodo.10392830

8: github.com/caterinaurban/libra

9: doi.org/10.5281/zenodo.4737450

1.3.2 Quantitative Verification of Intensional Properties

Part 'Quantitative Verification of Intensional Properties' is based on the work published at the 31st Static Analysis Symposium (SAS) 2024 [48].

[48]: Mazzucato et al. (2024), 'Quantitative Static Timing Analysis'

Part 'Quantitative Verification of Intensional Properties' focuses on the quantitative verification of intensional properties, *i.e.*, properties that depend on the program's internal behavior, *e.g.*, the number of iterations of a loop. This part is divided into two chapters: the first one presenting the novel quantitative timing analysis, and the second one showing the experimental evaluation.

In Chapter 8, we consider the number of program iterations as the program outcome, capturing the impact of input variables on the global number of iterations. Therefore, we are able to discover programming errors affecting iterations can degrade performance or introduce security vulnerabilities without functional errors. For instance, unexpected input impacts on runtime could reveal sensitive information [49], posing security threats. Even cryptographic programs are vulnerable to timing attacks, depending on implementation choices. Kocher [50] demonstrated that public key cryptographic algorithms, like RSA, are susceptible to timing attacks, potentially leaking secret keys.

[49]: Wong (2005), 'Timing attacks on RSA: revealing your secrets through the fourth dimension'

[50]: Kocher (1996), 'Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems'

For performance optimization, identifying input variables that significantly affect loop iterations helps developers focus on critical code segments [51]. Consequently, understanding the impact of input variables on runtime is paramount. In this second part of the thesis, we focus on quantifying input variables' impact on loop iterations as an indicator of runtime behavior.

[51]: Omar et al. (2017), 'GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms'

We leverage global loop bound analysis to derive an over-approximation of the global loop bound, encoding the quantification of input variables' impact as a linear programming problem. Our approach blends syntactic and semantic information: generating invariants as linear constraints for accuracy and combining global loop bound analysis with syntactic dependency analysis [42, Section 10] for scalability.

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

In Chapter 9, we present TimeSec,[10] a tool implementing our quantitative timing analysis. We demonstrate its effectiveness in the Amazon Web Services s2n-bignum cryptographic library,[11] certifying its immunity to timing side-channel attacks by showing no impact of input variables on loop iterations. Additionally, we evaluate TimeSec against programs from SV-Comp,[12] a benchmark suite specifically designed for software

10: github.com/denismazzucato/timesec

11: github.com/awslabs/s2n-bignum

12: sv-comp.sosy-lab.org/2024

verification tools.

Contributions The followings are the contributions of Part 'Quantitative Verification of Intensional Properties':

- In Chapter 8, we propose a static analysis, employing a linear constraint abstract domain, global loop bound analysis, and linear programming encoding, to quantify the impact of input variables on loop iterations.
- In Chapter 9, we present TimeSec,[13] a tool implementing our quantitative timing analysis, and evaluate it against the s2n-bignum cryptographic library[14] and programs from SV-Comp.[15]
- An artifact of TimeSec comprehensive of TimeSec's source code and evaluation benchmarks is available on Zenodo.[16]

13: github.com/denismazzucato/timesec

14: github.com/awslabs/s2n-bignum

15: sv-comp.sosy-lab.org/2024

16: doi.org/10.5281/zenodo.11507271

Related works are discussed at the end of each chapter, providing a comprehensive overview of the relevant literature. At the end of the thesis, in Part 'Conclusion', we conclude the thesis with a summary and perspectives on future directions.

Background

Abstract Interpretation | 2

In this chapter, we define the notations and introduce the necessary set theoretic and logic concepts to understand the rest of the thesis, we assume a basic knowledge of mathematics. Afterwards, we present an overview of Abstract Interpretation.

Dans ce chapitre, nous définissons les notations et introduisons les concepts nécessaires en théorie des ensembles et en logique pour comprendre le reste de la thèse, en supposant une connaissance de base des mathématiques. Par la suite, nous présentons un aperçu de l'interprétation abstraite.

2.1 Set Theory

Here, we briefly revisit well-known concepts to establish the logical and mathematical notation.

2.1.1 Logic Notation

We write $p \stackrel{\text{def}}{=} P$ to define the mathematical object p to be equal to the object P. Let $\mathbb{B} \stackrel{\text{def}}{=} \{\text{T}, \text{F}\}$ be the set of Boolean values, where T represents true and F represents false. We can state properties by logical predicates[1] yielding a Boolean value. Often, we say that a predicate holds whenever it evaluates to T. We combine predicates using logical connectives: $\wedge$ for conjunction, $\vee$ for disjunction, $\neg$ for negation, $\Rightarrow$ for implication (respectively, $\Leftarrow$ for reverse implication), and $\Leftrightarrow$ for if and only if, as well as $\forall$ for universal and $\exists$ for existential quantification. In the logical predicate $P \Rightarrow Q$, P is called the premise and Q the conclusion. We call P the *sufficient* condition for Q to hold and, conversely, Q the *necessary* condition for P to hold. In $P \Leftrightarrow Q$, P is a necessary and sufficient condition for Q to hold, and vice versa. $P \Rightarrow Q$ holds *vacuously* when P is false or when Q is true. We write $P \stackrel{\text{def}}{\Leftrightarrow} Q$ to denote that P is defined to hold whenever Q holds.

1: For instance, $x \leq x+1$ is a basic logical predicate

2.1.2 Sets

A set X is an unordered collection of distinct elements. We use $x \in X$ (respectively, $x \notin X$) to indicate that x is (respectively, is not) an element of the set X. A set is expressed in extension when it is uniquely determined by its elements: *e.g.*, $\{x, y\}$ represents the set containing elements x and y. The empty set is denoted by $\emptyset$ and no element belong to it, *i.e.*, $\forall x.\ x \notin \emptyset$. A singleton $\{x\}$ is a set containing only one element x. A set is defined in comprehension[2] when its elements are characterized by a common property: $\{x \in X \mid P(x)\}$ denotes the set of elements $x \in X$ for which $P(x)$ holds. The cardinality of a set X is represented by $|X|$, *e.g.*, $|\emptyset| = 0$ and $|\{x, y\}| = 2$.

2: Set comprehension notation will be, in particular, pervasively used in the rest of this thesis.

A set X is a subset of another set X', denoted $X \subseteq X'$, if every element of X is also an element of X'. The empty set $\emptyset$ is subset of any set. The power set $\wp(X)$ of a set X is defined as the set of all the subsets of X, *i.e.*, $\wp(X) \stackrel{\text{def}}{=} \{X' \mid X' \subseteq X\}$. The union of two sets X and Y, denoted $X \cup Y$, is the set containing all elements of X and all elements of Y, *i.e.*, $X \cup Y \stackrel{\text{def}}{=} \{x \mid x \in X \vee x \in Y\}$. More generally, the union of a set of sets X is denoted by $\bigcup X$, *i.e.*, $\bigcup X \stackrel{\text{def}}{=} \bigcup_{X' \in X} X' = \{x \mid \exists X' \in X.\ x \in X'\}$. The intersection $X \cap Y$ of two sets X and Y is the set of all elements that are common to both X and Y, *i.e.*, $X \cap Y \stackrel{\text{def}}{=} \{x \mid x \in X \wedge x \in Y\}$. Similarly to the set union, we generalize the intersection to a set of sets X by defining $\bigcap X$ as $\bigcap X \stackrel{\text{def}}{=} \bigcap_{X' \in X} X' = \{x \mid \forall X' \in X.\ x \in X'\}$. The relative complement of a set Y in a set X, denoted $X \setminus Y$, is the set of all elements of X that are not elements of Y, *i.e.*, $X \setminus Y \stackrel{\text{def}}{=} \{x \mid x \in X \wedge x \notin Y\}$. When $Y \subseteq X$ and the set X is clear from the context, we simply write $\neg Y$ for $X \setminus Y$ and we call it the complement of Y. A tuple is an ordered list of elements, *e.g.*, $\langle x_1, \ldots, x_n \rangle$ is a tuple of n elements, also called n-tuple. Differently from sets, the order of elements in a tuple is significant, *e.g.*, $\langle x, y \rangle \neq \langle y, x \rangle$. A pair[3] is a tuple of two elements, *e.g.*, $\langle x, y \rangle$ is a pair where the first element is x and the second y. The Cartesian product of two sets X and Y, denoted $X \times Y$, is the set of all pairs where the first component is an element of X and the second component is an element of Y, *i.e.*, $X \times Y \stackrel{\text{def}}{=} \{(x, y) \mid x \in X \wedge y \in Y\}$. More generally, $X_1 \times \cdots \times X_n \stackrel{\text{def}}{=} \{(x_1, \ldots, x_n) \mid x_1 \in X_1 \wedge \cdots \wedge x_n \in X_n\}$ denotes the set of all n-tuples from elements of $X_1, \ldots, X_n$ (we write X^n when $X_1 = \cdots = X_n = X$). The selection of the i-th element of a tuple $\langle x_1, \ldots, x_n \rangle$ is denoted by the subscript notation, *i.e.*, $\langle x_1, \ldots, x_n \rangle_i = x_i$ for $1 \leq i \leq n$.

3: Formally, a pair $\langle x, y \rangle$ can be defined as the set $\{\{x\}, \{x, y\}\}$ [52]. Thus, the first coordinate $\langle x, y \rangle_1$ is z such that $\forall X \in \langle x, y \rangle$ it holds that $z \in X$. The second coordinate $\langle x, y \rangle_2$ is z such that $\exists X \in \langle x, y \rangle$ such that $z \in X$ and that $\forall X, X' \in \langle x, y \rangle$ it holds that if $X \neq X'$ then $z \notin X$ or $z \notin X'$. We have that, $\langle x, y \rangle_1 = x$ and $\langle x, y \rangle_2 = y$.

[52]: Kuratowski (1921), 'Sur la notion de l'ordre dans la Théorie des Ensembles'

A covering of a set X is a set Z of non-empty subsets of X such that every element $x \in X$ belongs to a set in Z, *i.e.*, $X = \bigcup Z$. A partition of a set X is a covering Z such that any two sets in Z are disjoint, *i.e.*, every element $x \in X$ belongs to a unique set in Z, *i.e.*, $\forall X, Y \in Z : X \neq Y \Rightarrow X \cap Y = \emptyset$.

2.1.3 Numbers

Let $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ be the set of all naturals, integers, and reals, respectively. In general, whenever the precise numerical type is not known or required, we write $\mathbb{V} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ denoting any possible set of numbers. We write $\mathbb{V}^{\pm\infty}$ to denote $\mathbb{V}$ extended with the symbols $-\infty$ and $+\infty$. The set $\mathbb{V}_{\geq 0}$ denotes the set of non-negative numbers, *i.e.*, $\mathbb{V}_{\geq 0} \stackrel{\text{def}}{=} \{n \in \mathbb{V} \mid n \geq 0\}$. Similarly, we can use this notation with other predicates, for instance, $\mathbb{V}_{\leq m} \stackrel{\text{def}}{=} \{n \in \mathbb{V} \mid n \leq m\}$ denotes the set of numbers less than or equal to m. For example, $\mathbb{N}_{\geq 0}^{+\infty}$ denotes the set of natural numbers including zero and $+\infty$, on the other hand, $\mathbb{N}_{>0}$ denotes the set of strictly positive natural numbers.

Given two bounds $l, u \in \mathbb{V}$, we define the interval $[l, u]$ as the set of numbers between l and u, *i.e.*, $[l, u] \stackrel{\text{def}}{=} \{n \in \mathbb{V} \mid l \leq n \leq u\}$. Hence, $[l, u] = \emptyset$ when $l > u$. For the sake of compactness, we may write both the logical predicate and the numerical type in the same notation, *e.g.*, to define two natural numbers x, y that are not equal, we could write $x \neq y \in \mathbb{N}$ instead of $x, y \in \mathbb{N} \wedge x \neq y$.

2.1.4 Relations

A binary relation R between two sets X and Y is a subset of the cartesian product $X \times Y$. The following are some important properties which may hold for a binary relation R over a set S:

Reflexivity : $\forall x \in S : \langle x, x \rangle \in R$
Irreflexivity : $\forall x \in S : \langle x, x \rangle \notin R$
Symmetry : $\forall x, y \in S : \langle x, y \rangle \in R \Rightarrow \langle y, x \rangle \in R$
Antisymmetry : $\forall x, y \in S : \langle x, y \rangle \in R \wedge \langle x, y \rangle \in R \Rightarrow x = y$
Transitivity : $\forall x, y, z \in S : \langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \Rightarrow \langle x, z \rangle \in R$
Totality : $\forall x, y \in S : \langle x, y \rangle \in R \vee \langle x, y \rangle \in R$

An *equivalence* relation is a binary relation which is reflexive, symmetric, and transitive. A binary relation which is reflexive (resp. irreflexive), antisymmetric, and transitive is called a *partial order* (resp. *strict partial order*). A *preorder* is reflexive and transitive, but not necessarily antisymmetric. A *total order* is a (strict) partial order which is total, *i.e.*, any two elements of the given set are comparable (cf. totality).

2.1.5 Ordered Sets

A partially ordered set (poset in short) is defined as sets equipped with a partial order relation.

Definition 2.1.1 (Partially Ordered Set) *A* partially ordered set *(poset)* $\langle X, \sqsubseteq \rangle$ *is a set* X *equipped with a partial order relation* $\sqsubseteq \in \wp(X \times X)$*, that is reflexive* ($\forall x \in X.\ x \sqsubseteq x$)*, transitive* ($\forall x, y, z \in X.\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$)*, and antisymmetric* ($\forall x, y \in X.\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$)*.*

A finite partially ordered set $\langle D, \sqsubseteq \rangle$ can be represented by a Hasse diagram such as the one in Figure 2.1: each element $x \in X$ is uniquely represented by a node of the diagram, and there is an edge from a node $x \in X$ to a node $y \in X$ if y covers x, that is, $x \sqsubseteq y$ and there exists no $z \in X$ such that $x \sqsubseteq z \sqsubseteq y$. Hasse diagrams are usually drawn placing the elements higher than the elements they cover.

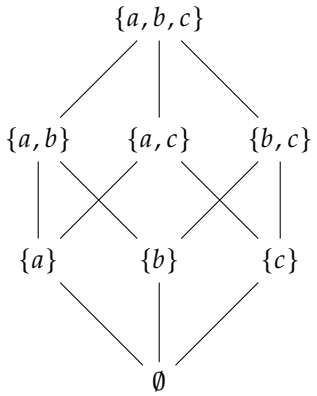


Figure 2.1: Hasse diagram for the partially ordered set $\langle \wp(\{a, b, c\}), \subseteq \rangle$.

Remark 2.1.1 Note that, for *any* set of elements X, $\langle X, \subseteq \rangle$ is a poset ordered by set inclusion $\subseteq$.

Example 2.1.1 To draw a few examples: Figure 2.2 represents the poset $\langle \wp(a, b, c), \{(a, b), (a, c)\} \rangle$; and Figure 2.3 represents the poset $\langle \wp(a, b, c, d), \{(a, b), (c, d)\} \rangle$.

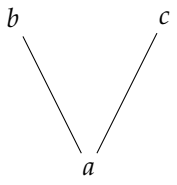


Figure 2.2: Hasse diagram for the poset $\langle \wp(a, b, c), \{(a, b), (a, c)\} \rangle$.

Next we introduce some important concepts related to partially ordered sets. Let $\langle X, \sqsubseteq \rangle$ be a partially ordered set and X' be a subset of X.

Greatest Element When it exists, the greatest element of X (or top) is denoted by $\top$: $\forall x \in X.\ x \sqsubseteq \top$.

Least Element When it exists, the least element of X (or bottom) is denoted by $\bot$: $\forall x \in X.\ \bot \sqsubseteq x$.

Maximal Element An element $x \in X'$ is maximal in X' if, $\forall y \in X'.\ x \sqsubseteq y$ implies $x = y$.

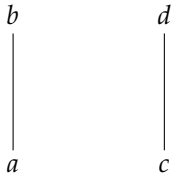


Figure 2.3: Hasse diagram for the poset $\langle \wp(a, b, c, d), \{(a, b), (c, d)\} \rangle$.

Note that, any partially ordered set can always be equipped with a least (resp. greatest) element by adding a new element that is smaller (resp. greater) than every other element.

Maximum A maximal element $x \in X'$ is a maximum of X' if it is unique.
Minimal Element Dually, an element $x \in X'$ is minimal in X' if, $\forall y \in X'.\ y \sqsubseteq x$ implies $x = y$.
Minimum A minimal element $x \in X'$ is a minimum of X' if it is unique.
Upper Bound An element $x \in X$ is an upper bound of X' if, $\forall y \in X'.\ y \sqsubseteq x$.
Lower Bound Dually, an element $x \in X$ is a lower bound of X' if, $\forall y \in X'.\ x \sqsubseteq y$.
Least Upper Bound When it exists, the least upper bound (lub) of X' is denoted by $\sqcup X'$: it is an upper bound $x \in X$ of X' such that, for every upper bound $x' \in X$ of X', $x \sqsubseteq x'$.
Greatest Lower Bound When it exists, the greatest lower bound (glb) of X' is denoted by $\sqcap X'$: it is a lower bound $x \in X$ of X' such that, for every lower bound $x' \in X$ of X', $x' \sqsubseteq x$.

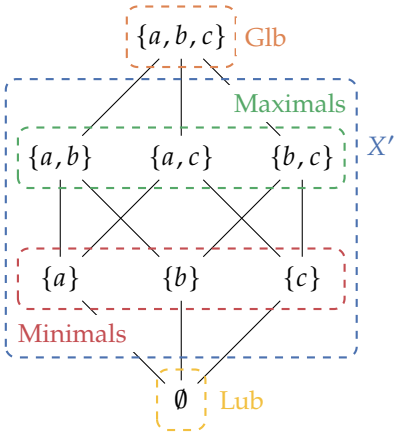


Figure 2.4: Maximal and minimal elements of the subset X', as well as its least upper and greatest lower bounds.

Example 2.1.2 Let us consider the partially ordered set $\langle \wp(\{a, b, c\}), \subseteq \rangle$ and the subset $X' \stackrel{\text{def}}{=} \{\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}\}$, represented in Figure 2.4. The maximal elements of X' are $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$, and are not comparable with each other. Hence, the maximum of X' does not exist, however, its least upper bound is $\sqcup X' = \{a, b, c\}$. The minimal elements of X' are $\{a\}$, $\{b\}$, and $\{c\}$. Again the minimum of X' does not exist and its greatest lower bound is $\sqcap X' = \emptyset$.

A set equipped with a total order is a *totally ordered set*. A *chain* is a totally ordered subset Y of a poset $\langle X, \sqsubseteq \rangle$. A chain is *ascending* if the sequence of elements is increasing, *i.e.*, $\forall i \in \{1, \ldots, n-1\}.\ y_i \sqsubseteq y_{i+1}$, and *descending* if the sequence of elements is decreasing, *i.e.*, $\forall i \in \{1, \ldots, n-1\}.\ y_{i+1} \sqsubseteq y_i$. A partially ordered set $\langle X, \sqsubseteq \rangle$ satisfies the *ascending chain condition* if and only if any infinite ascending chain $y_1 \sqsubseteq y_2 \sqsubseteq \ldots$ has an upper bound in X. Dually, it satisfies the *descending chain condition* if and only if any infinite descending chain $y_1 \sqsupseteq y_2 \sqsupseteq \ldots$ has a lower bound in X. A partially ordered set $\langle X, \sqsubseteq \rangle$ is *well-founded* if and only if there are no infinite descending chains without a lower bound in X.

Example 2.1.3 Let $x!$ be the factorial of x, defined as $x! = x \times (x-1) \times \ldots \times 1$ for $x \in \mathbb{N}$. The sequence $(x!)_{x \in \mathbb{N}}$ is an infinite ascending chain in the poset $\langle \mathbb{N}, \leq \rangle$:

$$1 \leq 2 \leq 6 \leq 24 \leq 120 \leq \ldots$$

Hence, the poset $\langle \mathbb{N}, \leq \rangle$ does not satisfy the ascending chain condition. On the other hand, any possible decreasing chain in $\langle \mathbb{N}, \leq \rangle$ has a lower bound, at least 0, hence it satisfies the descending chain condition and is well-founded.

Note that, since the empty set is a chain, a complete partial order has a least element, *i.e.*, $\bot = \sqcup\emptyset$.

Definition 2.1.2 (Complete Partial Order) *A complete partial order (CPO) is a partially ordered set $\langle X, \subseteq \rangle$ such that every chain Y has a least upper bound $\sqcup Y$ in X.*

As a consequence, any partial ordered set that satisfies the ascending chain condition and is equipped with a least element is a complete partial order.

2.1.6 Lattices

Lattices are posets that require every nonempty finite subset to have both a least upper bound and a greatest lower bound.

Any totally ordered set is a lattice.

Definition 2.1.3 (Lattice) *A* lattice $\langle X, \sqsubseteq, \sqcup, \sqcap \rangle$ *is a poset where* $\forall x, x' \in X$ *the least upper bound* $\sqcup\{x, x'\}$ *and the greatest lower bound* $\sqcap\{x, x'\}$ *exist.*

Complete lattices are lattices that additionally require every (possibly infinite) subset to have both a least upper bound and a greatest lower bound.

Definition 2.1.4 (Complete Lattice) *A* complete lattice $\langle X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ *is a lattice where* $\forall X' \subseteq X$ *the least upper bound* $\sqcup X'$ *and the greatest lower bound* $\sqcap X'$ *exist. Complete lattices have both a least element* $\bot \stackrel{\text{def}}{=} \sqcap X$ *and a greatest element* $\top \stackrel{\text{def}}{=} \sqcup X$.

Example 2.1.4 The posets of Figure 2.2 and Figure 2.3 are not lattices, as they do not have a least upper bound for every pair of elements, *e.g.*, $\sqcup\{a, b\}$ does not exist in Figure 2.2 and $\sqcup\{a, c\}$ does not exist in Figure 2.3. Figure 2.1 represents a complete lattice with a finite number of elements. The poset $\langle \mathbb{N}, \leq \rangle$ is a lattice, as every pair of natural numbers has a least upper bound and a greatest lower bound, but it is not a complete lattice, as $\sqcup\mathbb{N}$ does not exist. Lastly, $\langle \mathbb{N}^{+\infty}, \leq, \max, \min, 0, +\infty \rangle$ where $+\infty$ is greater than any natural number is a complete lattice. Figure 2.5 shows the Hasse diagram for $\langle \mathbb{N}^{+\infty}, \leq \rangle$ and Figure 2.6 the one for $\langle \mathbb{N}^{+\infty}, \leq \rangle$.

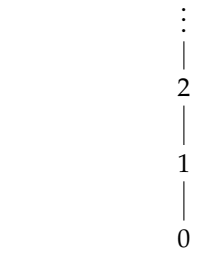


Figure 2.5: Hasse diagram for the lattice $\langle \mathbb{N}, \leq, \max, \min \rangle$.

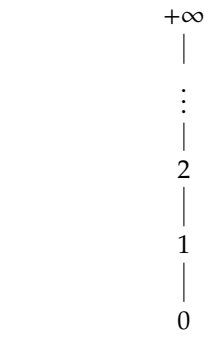


Figure 2.6: Hasse diagram for the complete lattice $\langle \mathbb{N}^{+\infty}, \leq, \max, \min, 0, +\infty \rangle$.

Remark 2.1.2 Given any set of elements X, the power set $\langle \wp(X), \subseteq, \cup, \cap, \emptyset, X \rangle$ is a complete lattice.

2.1.7 Functions

A *function* is a relation $f \in \wp(X \times Y)$ between two sets X and Y such that each element $x \in X$ is in relation with at most one element $y \in Y$, *i.e.*, $\forall x \in X.\ \exists! y \in Y.\ (x, y) \in f$. We write $f \in X \to Y$ to denote that f is a function from X to Y, and $f(x) = y$ if and only if $\langle x, y \rangle \in f$. We sometimes denote functions using the *lambda notation* $\lambda x \in X.\ f(x)$, or more concisely $\lambda x.\ f(x)$ when the domain is clear from the context. The *domain* of a function f is the set of elements $x \in X$ such that $f(x)$ is defined, *i.e.*, $\text{dom}(f) \stackrel{\text{def}}{=} \wp(x \in X \mid \exists y \in Y.\ f(x) = y)$. Dually, the *codomain* of a function f is the set of elements $y \in Y$ such that there exists $x \in X$ such that $f(x) = y$, *i.e.*, $\text{codom}(f) \stackrel{\text{def}}{=} \wp(y \in Y \mid \exists x \in X.\ f(x) = y)$. A function f is *total* when $\text{dom}(f) = X$, otherwise *partial* when $\text{dom}(f) \subseteq X$. A function $f \in X \to X$ is called an *operator* when the domain and codomain coincide.

The composition of two functions $f \in X \to Y$ and $g \in Y \to Z$ is the function $g \circ f \in X \to Z \stackrel{\text{def}}{=} \lambda x.\ g(f(x))$. The iterates of an operator

$f \in X \to X$ are defined as $f^0 \stackrel{\text{def}}{=} \text{id}$ and $f^{n+1} \stackrel{\text{def}}{=} f \circ f^n$ for $n \in \mathbb{N}$. We define the identity function $\text{id} \in X \to X$ as $\text{id} \stackrel{\text{def}}{=} \lambda x.\, x$. Let $f \in X \to Y$ be a function, the following properties may hold for f:

Injectivity $\forall x, y \in X.\, f(x) = f(y) \Rightarrow x = y$.
Surjectivity $\forall y \in Y.\, \exists x \in X.\, f(x) = y$.
Bijectivity A function that is both injective and surjective is called bijective, or isomorphism.

Two sets X and Y are *isomorphic* if there exists a bijective function $f \in X \to Y$. Next, we introduce some properties of functions that are relevant in the context of partially ordered sets.

Definition 2.1.5 (Monotonicity) *Let $\langle X, \sqsubseteq_X \rangle$ and $\langle Y, \sqsubseteq_Y \rangle$ be two partially ordered sets, and $f \in X \to Y$ be a function. The function f is* monotonic *whenever, for all $x, x' \in X$, it holds that $x \sqsubseteq_X x' \Rightarrow f(x) \sqsubseteq_Y f(x') \vee f(x) \sqsupseteq_Y f(x')$.*

We refer to a function as *increasing* monotone when $x \sqsubseteq_X x' \Rightarrow f(x) \sqsubseteq_Y f(x')$ and *decreasing* monotone when the inequality is reversed, *i.e.*, $x \sqsubseteq_X x' \Rightarrow f(x') \sqsupseteq_Y f(x)$. Note that, a decreasing monotone function is increasing monotone over $\langle X, \sqsubseteq_Y \rangle$ and $\langle Y, \sqsupseteq_Y \rangle$, where $\sqsupseteq_Y$ is the reverse order of $\sqsubseteq_Y$.

Definition 2.1.6 (Scott Continuity) *Let $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X \rangle$ and $\langle Y, \sqsubseteq_Y, \sqcup_Y, \sqcap_Y \rangle$ be two partially ordered sets, and $f \in X \to Y$ be a function. The function f is* Scott-continuous *whenever, for all chains $C \subseteq X$, if $\sqcup_X C$ exists then the upper bound is preserved, i.e., $f(\sqcup_X C) = \sqcup_Y \wp(f(x) \mid x \in C)$.*

Dually, a function is *co-continuous* whenever, for all chains $C \subseteq X$, if $\sqcap_X C$ exists then $f(\sqcap_X C) = \sqcap_Y \wp(f(x) \mid x \in C)$. A *complete $\sqcup$-morphism* (resp. *complete $\sqcap$-morphism*) preserves existing least upper bounds (resp. greatest lower bounds) of arbitrary non-empty sets.

An operator is *idempotent* when $f(f(x)) = f(x)$ for all $x \in X$, and *extensive* when $x \sqsubseteq f(x)$ for all $x \in X$.

Dually, the *lower closure operator* is monotone, idempotent, and *reductive*, *i.e.*, $f(x) \sqsubseteq x$ for all $x \in X$.

Definition 2.1.7 (Upper Closure Operator) *An* upper closure operator *on a partially ordered set $\langle X, \subseteq \rangle$ is an operator $\rho \in X \to X$ which is monotone, idempotent, and extensive.*

A *family* $F \in \Delta \to X$ of elements of X indexed by a set Δ (called the *domain* or *indexed set* of F, which may be infinite) is a function from Δ to X. Such family defines a set $\{F(i) \mid i \in \Delta\}$ (where $F(i)$ is often denoted as F_i with an index $i \in \Delta$) of elements of X indexed by Δ. It also defines the sequence $(F_i)_{i \in \Delta}$ of elements of X indexed by Δ.

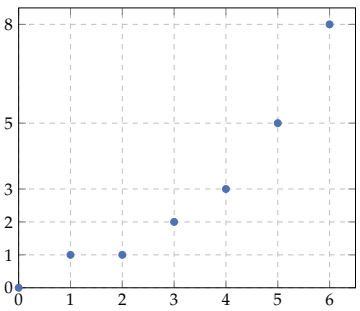


Figure 2.7: The fibonacci function.

Example 2.1.5 Let $\text{FIB}(n)$ be the n-th Fibonacci number, defined as:

$$\text{FIB}(n) \stackrel{\text{def}}{=} \begin{cases} n & \text{if } n \leq 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2) & \text{otherwise} \end{cases}$$

The sequence $(\textsc{fib}(n))_{n\in\mathbb{N}}$ is a family of elements of $\mathbb{N}$ indexed by $\mathbb{N}$:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Figure 2.7 depicts the first 6 elements of the Fibonacci sequence over the natural numbers.

Next, we show how to lift an ordered set via *pointwise lifting* to define a new ordered set.

Definition 2.1.8 (Pointwise Lifting) *Given a complete lattice* $\langle X, \sqsubseteq, \sqcup, \sqcap, \bot, \top\rangle$ *(respectively a lattice, a cpo, a poset) and a set* Y*, the set* $Y \to X$ *of all functions from* Y *to* X *forms a complete lattice* $\langle Y \to X, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top}\rangle$ *(respectively a lattice, a cpo, a poset) where the operators are defined by* pointwise lifting*:*

$$\begin{aligned} f \dot{\sqsubseteq} g &\triangleq \forall y \in Y.\ f(y) \sqsubseteq g(y) \\ \dot{\sqcup} X &\triangleq \lambda y.\ \sqcup \{f(y) \mid f \in X\} \\ \dot{\sqcap} X &\triangleq \lambda y.\ \sqcap \{f(y) \mid f \in X\} \\ \dot{\bot} &\triangleq \lambda y.\ \bot \\ \dot{\top} &\triangleq \lambda y.\ \top \end{aligned}$$

2.1.8 Properties

Properties (*e.g.*, "to be an even natural number" or "to be a number of the sequence of Fibonacci") by extension are the sets of elements that satisfy the property (*e.g.*, $2\mathbb{N} \stackrel{\text{def}}{=} \{x \in \mathbb{N} \mid \exists k \in \mathbb{N}.\ x = 2k\}$ or $B \stackrel{\text{def}}{=} \{\textsc{fib}(n) \mid n \in \mathbb{N}\}$). Therefore, if Q is a property, then $x \in Q$ means that x satisfies the property Q (*e.g.*, $2 \in 2\mathbb{N}$ or $12 \notin B$).

By considering properties as sets, we have that logical implication is subset inclusion. For instance, the property "to be an even natural number" implies "to be a multiple of 4," that is $2\mathbb{N} \subseteq 4\mathbb{N}$, where $4\mathbb{N} \stackrel{\text{def}}{=} \{x \in \mathbb{N} \mid \exists k \in \mathbb{N}.\ x = 4k\}$. Whenever P implies Q ($P \subseteq Q$), we say that P is *stronger* than Q and Q is *weaker* than P. Stronger properties are satisfied by fewer elements, while weaker properties are satisfied by more. Given an element $x \in X$, the strongest property satisfied by x is $\{x\}$, and the weakest property is X.

2.1.9 Fixpoints

Fixpoints are elements of a poset that are invariant under a function. In the following, let $\langle X, \subseteq\rangle$ be a partially ordered set and $f : X \to X$ be an operator on X.

Fixpoint An element $x \in X$ is a *fixpoint* of f whenever $f(x) = x$.
Prefixpoint An element $x \in X$ is a *prefixpoint* of f whenever $x \sqsubseteq f(x)$.
Postfixpoint An element $x \in X$ is a *postfixpoint* of f whenever $f(x) \sqsubseteq x$.
Set of Fixpoints The set of fixpoints of f is denoted by $\text{fix}(f) \stackrel{\text{def}}{=} \{x \in X \mid f(x) = x\}$.

Least Fixpoint The *least fixpoint* of f, denoted $\mathrm{lfp}\, f$, is a fixpoint of f such that, for every fixpoint $x \in X$ of f, $\mathrm{lfp}\, f \sqsubseteq x$. We write $\mathrm{lfp}_{x_0} f$ for the least fixpoint of f which is greater than or equal to $x_0 \in X$.

Greatest Fixpoint The *greatest fixpoint* of f, denoted $\mathrm{gfp}\, f$, is a fixpoint of f such that, for every fixpoint $x \in X$ of f, $x \sqsubseteq \mathrm{gfp}\, f$. We write $\mathrm{gfp}_{x_0} f$ for the greatest fixpoint of f which is smaller than or equal to $x_0 \in X$.

When the order $\sqsubseteq$ is not clear from the context, we explicitly write $\mathrm{lfp}^{\sqsubseteq} f$ and $\mathrm{gfp}^{\sqsubseteq} f$. Note that, in general, a function may have no fixpoint at all or multiple ones.

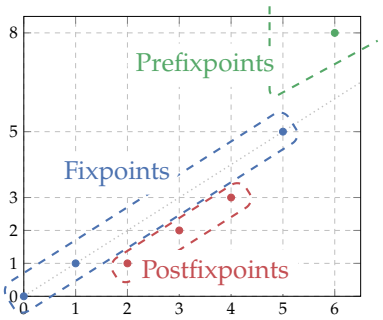


Figure 2.8: Prefixpoint, fixpoints, and postfixpoints of the fibonacci function.

Example 2.1.6 Let us consider again the fibonacci function $\textsc{fib} \in \mathbb{N} \to \mathbb{N}$. The fixpoints are shown in Figure 2.8, the set of fixpoints is $\mathrm{fix}(\textsc{fib}) = \{0, 1, 5\}$, the elements 2, 3, and 4 are prefixpoints are their values are smaller than the values of their images, and the elements greater than or equal to 6 are postfixpoints. The least fixpoint of $\textsc{fib}$ is $\mathrm{lfp}\, \textsc{fib} = 0$, as 0 is the smallest element of $\mathrm{fix}(\textsc{fib})$. On the other hand, the greatest fixpoint of $\textsc{fib}$ is $\mathrm{gfp}\, \textsc{fib} = 5$.

Instead, if we consider another operator, for example the $\textsc{cons}(x) \stackrel{\text{def}}{=} x+1$ operator, then there is no fixpoint, as $\textsc{cons}(x) \neq x$ for all $x \in \mathbb{N}$. instead, all the domain is composed of prefixpoints, as $x \leq \textsc{cons}(x) = x + 1$ for all $x \in \mathbb{N}$. Figure 2.9 shows graphically this example.

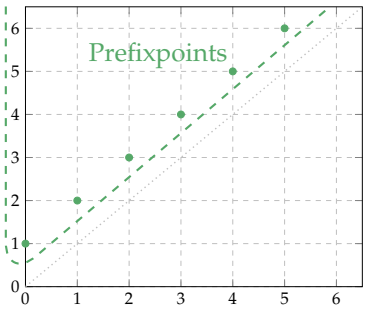


Figure 2.9: Prefixpoints of the construct function, *i.e.*, $\textsc{cons}(x) \stackrel{\text{def}}{=} x + 1$.

Next, we recall a fundamental theorem of Alfred Tarski [53] that characterizes fixpoints of monotonic functions over complete lattices.

Theorem 2.1.1 (Tarski's Fixpoint Theorem) *Let $\langle X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ be a complete lattice and $f \in X \to X$ be a monotonic function. The set of fixpoints $\mathrm{fix}(f)$ is a non-empty complete lattice.*

Proof. See Tarski [53]. □

Theorem 2.1.1 guarantees that f has a least fixpoint $\mathrm{lfp}\, f = \sqcap\{x \in X \mid f(x) \sqsubseteq x\}$ and a greatest fixpoint $\mathrm{gfp}\, f = \sqcup\{x \in X \mid x \sqsubseteq f(x)\}$ as $fixpoints(f)$ is a non-empty complete lattice.

However, such characterization is not constructive. An alternative that expresses the least fixpoint of a function as the limit of iterates is Kleene's Fixpoint Theorem [54].

Theorem 2.1.2 (Kleene's Fixpoint Theorem) *Let $\langle X, \sqsubseteq \rangle$ be a complete partial order and $f \in X \to X$ be a Scott-continuous function. The least fixpoint $\mathrm{lfp}\, f$ exists and is the least upper bound of the iterates f^n starting from $\bot$:*

$$\mathrm{lfp}\, f = \sqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$$

In the following, the partial order between the fixpoint iterates is called the *computational order*, in order to distinguish it from the *approximation order* defined in the next section.

[53]: Tarski (1955), 'A Lattice-Theoretical Fixpoint Theorem and its Applications'

[54]: Kleene (1952), *Introduction to Metamathematics*
In case of monotonic but not continuous functions, a theorem by Patrick and Radhia Cousot [55] expresses fixpoints as the limit of possibly transfinite iterations.

[55]: Cousot et al. (1979), 'Constructive Versions of Tarski's Fixed Point Theorems'

2.2 Abstract Interpretation

Abstract Interpretation [36, 56] is a general theory for approximating the semantics of programs as a unifying framework for static program analysis, co-invented in the late 1970s by Patrick Cousot and Radhia Cousot. Various books are available for an in-depth study of the topic [38, 57]. In the following, we introduce the notations, main definitions, and results that will be used throughout the rest of this thesis.

[36]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'
[56]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'
[38]: Cousot (2021), *Principles of abstract interpretation*
[57]: Rival et al. (2020), *Introduction to Static Analysis: An Abstract Interpretation Perspective*

2.2.1 Transition System

The *semantics* of a program is a mathematical characterization of its behavior for all possible inputs. In order to be independent of a particular choice of programming language, we formalize programs as transition systems, where the behavior of a program is described by a set of states and a set of transitions between states. In the next section, we instantiate the transition system model for a small imperative language.

Definition 2.2.1 (Transition System) *A* transition system *is a pair* $\langle \Sigma, \tau \rangle$ *where* Σ *is a (potentially infinite) set of states and* $\tau \in \Sigma \times \Sigma$ *is the transition relation, which describes the possible transitions between states.*

This model allows expressing programs with (possibly unbounded) non-determinism. In the following, we write $s \longrightarrow s'$ to denote a transition from state s to state s', *i.e.*, $\langle s, s' \rangle \in \tau$. The set $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma.\ \langle s, s' \rangle \notin \tau\}$ represents the *final states* (or blocking states) of the transition system.

2.2.2 Maximal Trace Semantics

In the following, we formally define the maximal trace semantics over a transition system, which is the set of all possible traces generated by the transition system.

Sequences. Given a set of elements X, the set of all sequences of exactly n elements of X is denoted by $X^n \stackrel{\text{def}}{=} \{s_0 \dots s_{n-1} \mid \forall i < n.\ s_i \in X\}$. The symbol ϵ denotes the empty sequence, hence $X^0 = \{\epsilon\}$.

We define:

Finite Sequences $X^\star$: The set of all finite sequences of elements of X is defined as $X^\star \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} X^n$.
Non-Empty Finite Sequences X^+: The set of all non-empty finite sequences of elements of X is $X^+ \stackrel{\text{def}}{=} X^\star \setminus X^0 = \bigcup_{n \in \mathbb{N}_{>0}} X^n$.
Infinite Sequences X^∞: The set of all infinite sequences of elements of X is defined as $X^\infty \stackrel{\text{def}}{=} \{s_0 \dots \mid \forall i \in \mathbb{N}.\ s_i \in X\}$.
Non-Empty Finite or Infinite Sequences $X^{+\infty}$: The set of all non-empty finite or infinite sequences of elements of X is $X^{+\infty} \stackrel{\text{def}}{=} X^+ \cup X^\infty$.
Finite or Infinite Sequences $X^{\star\infty}$: The set of all finite or infinite sequences of elements of X is $X^{\star\infty} \stackrel{\text{def}}{=} X^\star \cup X^\infty$.

Let $s \in X$ be an element, we often refer to just s as the sequence of length 1 containing only s, *i.e.*, $\{s\} \in X^1$. To *concatenate* two sequences $\sigma, \sigma' \in X^{+\infty}$, we write $\sigma \cdot \sigma'$ for the sequence obtained by appending σ' to σ. It holds that $\sigma \cdot \epsilon = \epsilon \cdot \sigma = \sigma$ and $\sigma \cdot \sigma' = \sigma$ whenever $\sigma \in X^{\infty}$. To *merge* two set of sequences $Y \subseteq X^+$ and $Y' \subseteq X^{+\infty}$, we write $Y \,;\, Y' \stackrel{\text{def}}{=} \{\sigma \cdot s \cdot \sigma' \mid s \in X \wedge \sigma \cdot s \in Y \wedge s \cdot \sigma' \in Y'\}$ when a finite sequence in Y ends with the initial state of a sequence in Y'.

Traces. Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states Σ that respects the transition relation τ. That is, for every pair of consecutive states $s, s' \in \Sigma$, it holds that $\langle s, s' \rangle \in \tau$.

Maximal Trace Semantics The *maximal trace semantics* $\Lambda \in \wp(\Sigma^{+\infty})$ of a transition system $\langle \Sigma, \tau \rangle$ is the set of all traces that are terminating in the final states Ω (*i.e.*, of finite length) and all non-terminating traces (*i.e.*, of infinite length) generated by $\langle \Sigma, \tau \rangle$.

Definition 2.2.2 (Maximal Trace Semantics) *Let $\langle \Sigma, \tau \rangle$ be a transition system. The* maximal trace semantics $\Lambda \in \wp(\Sigma^{+\infty})$ *generated by $\langle \Sigma, \tau \rangle$ is defined as:*

$$\Lambda \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}_{>0}} \{s_0 \dots s_{n-1} \in \Sigma^n \mid \forall i < n-1.\ \langle s_i, s_{i+1} \rangle \in \tau \wedge s_{n-1} \in \Omega\} \cup \{s_0 \dots \in \Sigma^{\infty} \mid \forall i \in \mathbb{N}.\ \langle s_i, s_{i+1} \rangle \in \tau\}$$

The maximal trace semantics lattice $\langle \wp(\Sigma^{+\infty}), \subseteq, \cup, \cap, \Sigma^{\infty}, \Sigma^{+} \rangle$ forms a complete lattice.

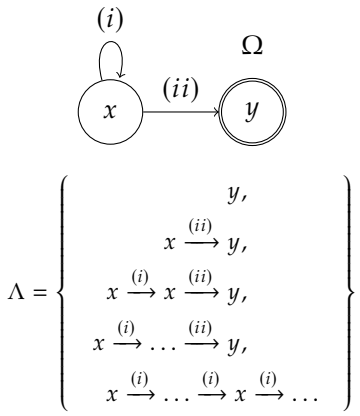


Figure 2.10: Maximal trace semantics of the transition system presented in Example 2.2.1.

Example 2.2.1 Let us consider the transition system $\langle \Sigma, \tau \rangle$ with states $\Sigma = \{x, y\}$, transitions $\tau = \{x \xrightarrow{(i)}, x \xrightarrow{(ii)} y\}$ with final states $\Omega = \{y\}$. The maximal trace semantics Λ is the set of all traces that are terminating in the final state y and all non-terminating traces from the transition $\langle x, x \rangle$. Figure 2.10 depicts graphically the transition system and the traces generated by the maximal trace semantics. Note that, the maximal trace semantics does not represent partial computations that are not terminating in the final states, such as the trace $x \xrightarrow{(i)} x \xrightarrow{(i)} x$.

Remark 2.2.1 Observe that not all program semantics can be generated by a transition system. For instance, in case the program semantics is the set of traces starting with an arbitrary finite number of x where y is eventually reached, the transition relation $\tau = \{\langle x, x \rangle, \langle x, y \rangle\}$ generates also infinite sequences of x that are not part of the semantics.

The maximal trace semantics fully describes the behavior of programs. However, reasoning about a particular property of a program is often facilitated by the design of a semantics that abstracts away from irrelevant details about program computations. In the following, to facilitate reading, we call the maximal trace semantics simply the *trace semantics*.

Remark 2.2.2 (Non-determinism) We say that a transition system $\langle \Sigma, \tau \rangle$ is *non-deterministic* whenever the maximal trace semantics generates (at least) two different traces that share the same prefix. Formally:

$$\langle \Sigma, \tau \rangle \text{ is non-deterministic}$$
$$\Leftrightarrow$$
$$\exists \sigma \in \Sigma^+, \sigma', \sigma'' \in \Sigma^{+\infty}.\ \sigma' \neq \sigma'' \wedge \sigma \cdot \sigma' \in \Lambda \wedge \sigma \cdot \sigma'' \in \Lambda$$

In fact, the transition system of Example 2.2.1 is non-deterministic, as these two traces belongs to the maximal trace semantics:

$$x \xrightarrow{(i)} x \xrightarrow{(ii)} y$$
$$x \xrightarrow{(i)} x \xrightarrow{(i)} x \xrightarrow{(ii)} y$$

Both traces share the same prefix, *i.e.*, $x \xrightarrow{(i)} x$.

2.2.3 Collecting Semantics

As seen in Section 2.1.8, a *property* is specified by its extension, that is, the set of elements that manifest such property. In this thesis, we consider *program properties* with respect to their trace semantics. By program property we mean properties of the semantics of programs. In our settings, since the program semantics is the trace semantics in $\wp(\Sigma^{+\infty})$, then a program property is a set of sets of traces in $\wp(\wp(\Sigma^{+\infty}))$. The strongest property of the trace semantics is the set containing the trace semantics and nothing else, this property is called the *collecting semantics*.

Program properties are also referred to as *hyperproperties* [58]. In this thesis, we prefer the term *program properties* to emphasize that we consider properties of the semantics of programs.

[58]: Clarkson et al. (2010), 'Hyperproperties'

Definition 2.2.3 (Collecting Semantics) *Let* $\Lambda \in \wp(\Sigma^{+\infty})$ *be the trace semantics of a transition system* $\langle \Sigma, \tau \rangle$. *The* collecting semantics $\Lambda^{\mathbb{C}} \in \wp(\wp(\Sigma^{+\infty}))$ *is defined as:*

$$\Lambda^{\mathbb{C}} \stackrel{\text{def}}{=} \{ \Lambda \}$$

The collecting semantics $\Lambda^{\mathbb{C}}$ is satisfied only and exactly by the trace semantics Λ. We write $\Lambda[\![P]\!]$ to denote the trace semantics of a particular program P, that is, the maximal trace semantics generated by the transition system induced by the program P, see later in Section 2.3. The same notation convention applies to all the semantics we study in this thesis. Hence, the collecting semantics of a program P is denoted by $\Lambda^{\mathbb{C}}[\![P]\!]$.

Property Validation. A program P with semantics $\Lambda[\![P]\!]$ satisfies a property $H \in \wp(\wp(\Sigma^{+\infty}))$ if and only if the property H is implied by the collecting semantics $\Lambda^{\mathbb{C}}[\![P]\!] \in \wp(\wp(\Sigma^{+\infty}))$.

Definition 2.2.4 (Validation of Program Properties) *Let P be a program and H be a property of programs. We say that P* satisfies *H whenever:*

$$P \models H \;\Leftrightarrow\; \Lambda^{\mathbb{C}}[\![P]\!] \subseteq H$$

Traditionally in the literature, properties are often trace properties, and thus can be expressed as sets of traces in $\wp(\Sigma^{+\infty})$. In this case, it is often used a simplified form of property validation: $P \models F \;\Leftrightarrow\; \Lambda[\![P]\!] \subseteq F$ for a property $F \in \wp(\Sigma^{+\infty})$. Note that, not all program properties can be expressed as trace properties. As an example, the unused property [42] (cf. Chapter 3) is not a trace property, because whether a program utilizes a variable or not depends on the whole set of traces generated by the program.

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

Hierarchy of Semantics To reason about a particular program property, it is not necessary to consider all the details of the trace semantics. In fact, reasoning is facilitated by the design of an abstract semantics that forgets irrelevant details about program computations. Therefore, there is no general purpose program semantics that is suitable for reasoning about all program properties. Instead, there exist a wide range of program semantics dedicated to a specific class of program properties, each of them abstracting away from the computational details differently.

Abstract interpretation provides a systematic and uniform way to relate semantics described as fixpoints of monotonic functions over ordered structures. These semantics are organized in a hierarchy, where the collecting semantics is the most precise semantics and all other semantics interrelate at different levels [59, 60].

[59]: Cousot (2002), 'Constructive design of a hierarchy of semantics of a transition system by abstract interpretation'
[60]: Cousot (2024), 'Calculational Design of [In]Correctness Transformational Program Logics by Abstract Interpretation'

2.2.4 Galois Connection

Galois connections[61] are a fundamental concept in abstract interpretation that relate two partially ordered sets. They formalize the correspondence between concrete and abstract properties, and provide a systematic way to derive abstract semantics from concrete semantics.

[61]: Schmidt (1953), 'Beiträge zur Filtertheorie. II'

Definition 2.2.5 (Galois Connection) *Let $\langle X, \sqsubseteq_X\rangle$ and $\langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ be two partially ordered sets. The pair $\langle \alpha, \gamma\rangle$ of functions $\alpha \in X \to X^\natural$ and $\gamma \in X^\natural \to X$ forms a* Galois connection, *denoted by $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$, if and only if, for all $x \in X$ and $x^\natural \in X^\natural$, it holds that:*

$$x \sqsubseteq_X \gamma(x^\natural) \;\Leftrightarrow\; \alpha(x) \sqsubseteq_{X^\natural} x^\natural$$

The definition of Galois connection can also be defined for preorders, most of the results presented hold for preorders as well.

The poset $\langle X, \sqsubseteq_X\rangle$ is called the *concrete domain*, and $\langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ is called the *abstract domain*. The function α is called the *abstraction* and γ is called the *concretization* function, or together they are called right-left adjoints. The intuition behind Galois connections is that properties in a concrete domain X are approximated by abstract properties in $X^\natural$. The concretization function γ provides the concrete meaning $\gamma(x^\natural)$ of an abstract property $x^\natural \in X^\natural$, that is, $\gamma(x^\natural)$ is the least precise element of X that is overapproximated by the abstract property $x^\natural$. As a consequence,

any concrete property $x \in X$ below $\gamma(x^\natural)$, *i.e.*, $x \sqsubseteq_X \gamma(x^\natural)$, is approximated by $x^\natural$. Whenever $x \sqsubseteq_X \gamma(x^\natural)$, we say that the abstract property $x^\natural$ is a *sound* overapproximation of the concrete property x. That is, x is a stronger property than $\gamma(x^\natural)$, the stronger the abstract property, the more precise is the abstraction.

The implication $\alpha(x) \sqsubseteq_{X^\natural} x^\natural \Rightarrow x \sqsubseteq_X \gamma(x^\natural)$ holds when $x^\natural = \alpha(x)$ so that $x \sqsubseteq_X \gamma(\alpha(x))$, meaning that $\alpha(x)$ is a sound overapproximation of x. The other implication $x \sqsubseteq_X \gamma(x^\natural) \Rightarrow \alpha(x) \sqsubseteq_{X^\natural} x^\natural$ means that for any sound overapproximation $x^\natural$ of x, $\alpha(x)$ is more precise than $x^\natural$. It follows that $\alpha(x)$ is the most precise sound overapproximation of x. We say that the abstraction of a concrete property is *exact* whenever $\gamma(\alpha(x)) = x^\natural$. In such case, the abstraction loses no information about the concrete property.

We write $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ when α is surjective (called a *Galois insertion*), $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ when γ is surjective (called a *Galois retraction*), and $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\!\twoheadrightarrow}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ for a bijection (called a *Galois correspondence*).

In Galois connections, the abstraction α and concretization γ are monotonic functions, that is, for all $x, x' \in X$ and $x^\natural_1, x^\natural_2 \in X^\natural$, it holds that $x \sqsubseteq_X x' \Rightarrow \alpha(x) \sqsubseteq_{X^\natural} \alpha(x')$ and $x^\natural_1 \sqsubseteq_{X^\natural} x^\natural_2 \Rightarrow \gamma(x^\natural_1) \sqsubseteq_X \gamma(x^\natural_2)$. The composition $\gamma \circ \alpha \in X \to X$ is extensive, *i.e.*, $x \sqsubseteq_X \gamma(\alpha(x))$ for all $x \in X$, meaning that the loss of information in the abstraction is sound. The composition $\alpha \circ \gamma \in X^\natural \to X^\natural$ is reductive, *i.e.*, $x^\natural \sqsubseteq_{X^\natural} \alpha(\gamma(x^\natural))$ for all $x^\natural \in X^\natural$, meaning that the concretization introduces no loss of information. The abstraction α preservers existing lubs, and the concretization γ preservers existing glbs.

Remark 2.2.3 In a Galois Connection, one adjoint uniquely determines the other:

$$\alpha(x) \overset{\text{def}}{=} \bigsqcap\{x^\natural \in X^\natural \mid x \sqsubseteq_X \gamma(x^\natural)\}$$
$$\gamma(x^\natural) \overset{\text{def}}{=} \bigsqcup\{x \in X \mid \alpha(x) \sqsubseteq_{X^\natural} x^\natural\}$$

It follows that if $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ and $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma'}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ are two Galois connections, then $\gamma = \gamma'$. If $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ and $\langle X, \sqsubseteq_X\rangle \underset{\alpha'}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural}\rangle$ are two Galois connections, then $\alpha = \alpha'$.[4] Galois connections also compose, that is, given two Galois connections $\langle X, \sqsubseteq_X\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Y, \sqsubseteq_Y\rangle$ and $\langle Y, \sqsubseteq_Y\rangle \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \langle Z, \sqsubseteq_Z\rangle$, then $\langle X, \sqsubseteq_X\rangle \underset{\alpha'\circ\alpha}{\overset{\gamma\circ\gamma'}{\leftrightarrows}} \langle Z, \sqsubseteq_Z\rangle$ is a Galois connection.

4: It is important to note that the two Galois connections must have the same concrete and abstract partial orders $\sqsubseteq_X$ and $\sqsubseteq_{X^\natural}$, otherwise the abstractions or concretizations may no longer be equal.

Based on Galois connections, abstract interpretation provides a *constructive* way to derive program semantics by successive abstractions of the trace semantics.

Example 2.2.2 (Forward Reachability State Semantics) The *forward reachability state semantics* $\Lambda^r \in \wp(\Sigma)$ collects the states that are reachable from a given set of initial states $\mathbb{I} \in \wp(\Sigma)$ by abstraction of the maximal

trace semantics $\Lambda \in \wp(\Sigma^{+\infty})$. The two form a Galois connection:

$$\langle \wp(\Sigma^{+\infty}), \subseteq \rangle \underset{\alpha^{\mathrm{r}}}{\overset{\gamma^{\mathrm{r}}}{\leftrightarrows}} \langle \wp(\Sigma), \subseteq \rangle$$

where the abstraction α^{r} collects all the states that are reachable from the initial states:

$$\begin{aligned} \alpha^{\mathrm{r}}(T) &\overset{\text{def}}{=} \{s \in \Sigma \mid s' \in \mathbb{I} \wedge \sigma \in \Sigma^{+} \wedge s' \cdot \sigma \cdot s \in T\} \\ \gamma^{\mathrm{r}}(S) &\overset{\text{def}}{=} \{s' \cdot \sigma \cdot s \mid s' \in \mathbb{I} \wedge \sigma \in \Sigma^{+\infty} \wedge s \in S\} \end{aligned}$$

Thus, we obtain the following definition for the forward reachability state semantics:

$$\Lambda^{r} \overset{\text{def}}{=} \alpha^{\mathrm{r}}(\Lambda)$$

Note that, the reachability semantics presented in this example is strictly less expressive than the maximal trace semantics. For instance, the reachability semantics does not capture non-terminating traces.

Example 2.2.3 (Backward Co-Reachability State Semantics) Dually, the *backward co-reachability state semantics* $\Lambda^{\overline{r}} \in \wp(\Sigma)$ collects the states that can reach a given set of final states $\Omega \in \wp(\Sigma)$ by abstraction of the maximal trace semantics $\Lambda \in \wp(\Sigma^{+\infty})$. The two form a Galois connection:

$$\langle \wp(\Sigma^{+\infty}), \subseteq \rangle \underset{\alpha^{\overline{\mathrm{r}}}}{\overset{\gamma^{\overline{\mathrm{r}}}}{\leftrightarrows}} \langle \wp(\Sigma), \subseteq \rangle$$

where the abstraction $\alpha^{\overline{\mathrm{r}}}$ collects all the states that can reach the final states:

$$\begin{aligned} \alpha^{\overline{\mathrm{r}}}(T) &\overset{\text{def}}{=} \{s \in \Sigma \mid s' \in \Omega \wedge \sigma \in \Sigma^{+} \wedge s \cdot \sigma \cdot s' \in T\} \\ \gamma^{\overline{\mathrm{r}}}(S) &\overset{\text{def}}{=} \{s \cdot \sigma \cdot s' \mid s \in S \wedge \sigma \in \Sigma^{+\infty} \wedge s' \in \Omega\} \end{aligned}$$

Thus, we obtain the following definition for the backward co-reachability state semantics:

$$\Lambda^{\overline{r}} \overset{\text{def}}{=} \alpha^{\overline{\mathrm{r}}}(\Lambda)$$

Remark 2.2.4 (Absence of Galois Connection) The use of Galois connections is not necessary to define program semantics. Instead, it represents an ideal situation where there exists the best way to approximate concrete properties. Cousot and Cousot [62] illustrates how to relax the Galois connection requirement in order only to work with concretization functions (or dually, with abstraction functions). In practice, concretization-based abstractions are much more common.

[62]: Cousot et al. (1992), 'Abstract Interpretation Frameworks'

2.3 A Small Imperative Language

The formal treatment introduced in the previous section is language independent. Here, we consider a simple imperative programming language to instantiate the transition system model and define the semantics of programs.

2.3.1 Syntax

Our programming language features the classic iterative constructs such as while loops and assignments, and the basic control flow constructs such as conditionals and sequential compositions. To keep the language simple, we do not consider function calls, recursion, or pointers. The set $\mathbb{V}\mathtt{ar}$ is the finite set of (statically allocated) program variables, the data type is $\mathbb{V} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ for arithmetic and $\mathbb{B}$ for boolean conditions. The set $\Delta \subseteq \mathbb{V}\mathtt{ar}$ contains the input variables. Figure 2.11 shows the syntax of our small imperative language, where $\mathtt{x} \in \mathbb{V}\mathtt{ar}$ is a variable, $v \in \mathbb{V}$ is a constant value, e is an arithmetic expression, b is a boolean condition, and st is a program statement.

The entry point of a program P is a statement st, denoted by $\mathtt{entry}\ st$, followed by a unique program location $l \in \mathbb{L}$. Another unique program location appears within each statement. Since we rarely need to explicitly specify program locations, most of the time we do not report them to lighten the notation. A statement st is either a skip statement $\mathtt{skip}$, an assignment statement $\mathtt{x} := e$, a conditional statement $\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st'$, a while statement $\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}$, or a sequential composition statement $st; st'$.

Arithmetic expressions e are either a constant value $v \in \mathbb{V}$, variable $\mathtt{x} \in \mathbb{V}\mathtt{ar}$, an addition $e + e$, a subtraction $e - e$, or a non-deterministic choice $\mathtt{rand}(l, u)$ that returns a random integer between $l \in \mathbb{Z}^{\pm\infty}$ and $u \in \mathbb{Z}^{\pm\infty}$.[5] The non-deterministic choice is useful to model user input, to approximate function calls, arithmetic expressions that cannot be represented exactly in our language, or – generally – uncertainty in the program. Boolean expressions b are either a comparison between arithmetic values $e \leq v$ and $e = v$, a conjunction $b \wedge b$, or a negation $\neg b$. The operator Vars takes as input an arithmetic expression or boolean condition and returns the set of variables that appear in the expression or condition. Formally:

$$
\begin{aligned}
\textsc{Vars}(v) &\stackrel{\text{def}}{=} \emptyset \\
\textsc{Vars}(\mathtt{x}) &\stackrel{\text{def}}{=} \{\mathtt{x}\} \\
\textsc{Vars}(e + e') &\stackrel{\text{def}}{=} \textsc{Vars}(e) \cup \textsc{Vars}(e') \\
\textsc{Vars}(e - e') &\stackrel{\text{def}}{=} \textsc{Vars}(e) \cup \textsc{Vars}(e') \\
\textsc{Vars}(\mathtt{rand}(l, u)) &\stackrel{\text{def}}{=} \emptyset
\end{aligned}
\qquad
\begin{aligned}
\textsc{Vars}(e \leq v) &\stackrel{\text{def}}{=} \textsc{Vars}(e) \\
\textsc{Vars}(e = v) &\stackrel{\text{def}}{=} \textsc{Vars}(e) \\
\textsc{Vars}(b \wedge b') &\stackrel{\text{def}}{=} \textsc{Vars}(b) \cup \textsc{Vars}(b') \\
\textsc{Vars}(\neg b) &\stackrel{\text{def}}{=} \textsc{Vars}(b)
\end{aligned}
$$

Remark 2.3.1 (Syntactic Sugar) The syntax depicted in Figure 2.11 is minimalistic. Through *syntactic sugar*, we can extend the language with additional constructs that are not part of the core language but can be expressed in terms of the core constructs. For instance, we can define a shorthand version of the conditional statement as just $\mathtt{if}\ b\ \mathtt{then}\ st$ without the else branch. Such syntactic sugar expands to $\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ \mathtt{skip}$.

the set of boolean values is $\mathbb{B} \stackrel{\text{def}}{=} \{\mathtt{T}, \mathtt{F}\}$ where T is true and F is false.

$$
\begin{aligned}
e &::= v \\
&\mid \mathtt{x} \\
&\mid e + e \\
&\mid e - e \\
&\mid \mathtt{rand}(l, u) \qquad (l \in \mathbb{Z}^{\pm\infty} \wedge l < u) \\
b &::= e \leq v \\
&\mid e = v \\
&\mid b \wedge b \\
&\mid \neg b \\
st &::= {}^{l}\mathtt{skip} \\
&\mid {}^{l}\mathtt{x} := e \\
&\mid \mathtt{if}\ {}^{l}b\ \mathtt{then}\ st\ \mathtt{else}\ st \\
&\mid \mathtt{while}\ {}^{l}b\ \mathtt{do}\ st\ \mathtt{done} \\
&\mid st; st \\
P &::= \mathtt{entry}\ st^{l} \qquad (l \in \mathbb{L})
\end{aligned}
$$

Figure 2.11: Syntax of the small imperative language.

5: Note that, the non-deterministic choice cannot be instantiated as $\mathtt{rand}(+\infty, +\infty)$ as we require the two bounds to be not equal.

Similarly, we can reconstruct boolean conditions as well: $b \vee b'$ is expanded to $\neg(\neg b \wedge \neg b')$ by De Morgan's laws.

We could have been ever more minimalistic, *e.g.*, all arithmetic expressions could have been defined from 1, $-$, and iteration, or all boolean conditions with $<$ and `nand`. The syntax in Figure 2.11 is a trade-off between minimalism, simplicity, and expressiveness.

$$\mathbb{A}[\![e]\!] \in \mathbb{E} \to \wp(\mathbb{V})$$
$$\mathbb{A}[\![v]\!]\mu \stackrel{\text{def}}{=} \{v\}$$
$$\mathbb{A}[\![\mathtt{x}]\!]\mu \stackrel{\text{def}}{=} \{\mu(\mathtt{x})\}$$
$$\mathbb{A}[\![e + e']\!]\mu \stackrel{\text{def}}{=} \left\{ v + v' \;\middle|\; \begin{array}{l} v \in \mathbb{A}[\![e]\!]\mu \wedge \\ v' \in \mathbb{A}[\![e']\!]\mu \end{array} \right\}$$
$$\mathbb{A}[\![e - e']\!]\mu \stackrel{\text{def}}{=} \left\{ v - v' \;\middle|\; \begin{array}{l} v \in \mathbb{A}[\![e]\!]\mu \wedge \\ v' \in \mathbb{A}[\![e']\!]\mu \end{array} \right\}$$
$$\mathbb{A}[\![\mathtt{rand}(l,u)]\!]\mu \stackrel{\text{def}}{=} \{v \in \mathbb{V} \mid l \le v \le u\}$$

Figure 2.12: Semantics of arithmetic expressions.

$$\mathbb{B}[\![b]\!] \in \mathbb{E} \to \mathbb{B}$$
$$\mathbb{B}[\![e \le v]\!]\mu \stackrel{\text{def}}{\Leftrightarrow} \mu(e) \le v$$
$$\mathbb{B}[\![e = v]\!]\mu \stackrel{\text{def}}{\Leftrightarrow} \mu(e) = v$$
$$\mathbb{B}[\![b \wedge b']\!]\mu \stackrel{\text{def}}{\Leftrightarrow} \mathbb{B}[\![b]\!]\mu \wedge \mathbb{B}[\![b']\!]\mu$$
$$\mathbb{B}[\![\neg b]\!]\mu \stackrel{\text{def}}{\Leftrightarrow} \neg\mathbb{B}[\![b]\!]\mu$$

Figure 2.13: Semantics of boolean conditions.

2.3.2 Maximal Trace Semantics

Here, we instantiate the general definition of transition system and maximal trace semantics, cf. Definition 2.2.2, for our small imperative language.

Expression Semantics An *environment* $\mu \in \mathbb{E} \stackrel{\text{def}}{=} \mathbb{V}\mathtt{ar} \to \mathbb{V}$ is a mapping from a variable $\mathtt{x} \in \mathbb{V}\mathtt{ar}$ to its value in memory $\mu(\mathtt{x}) \in \mathbb{V}$. The semantics of an arithmetic expression e is defined as the function $\mathbb{A}[\![e]\!] \in \mathbb{E} \to \wp(\mathbb{V})$ that maps an environment μ to the set of values $\mathbb{A}[\![e]\!]M$ that the expression e can evaluate to under the environments in M. Note that, expressions may evaluate to multiple values due to non-deterministic choices. Figure 2.12 shows the definition of $\mathbb{A}$.

Similarly, we define the semantics of boolean conditions b as the function $\mathbb{B}[\![b]\!] \in \mathbb{E} \to \mathbb{B}$ that holds whenever the condition b holds in the environment μ. The semantics of boolean conditions is defined in Figure 2.13. In the following, we write `true` and `false` for the boolean condition that always evaluates to true and false, respectively.

The update of an environment μ with a variable $\mathtt{x}$ to a value v is denoted by $\mu[\mathtt{x} \leftarrow v] \stackrel{\text{def}}{=} \lambda\mathtt{x}'.\mu(\mathtt{x}')$ if $\mathtt{x} \neq \mathtt{x}'$ and v otherwise. The environment update is lifted pointwise to sets of environments as $M[\mathtt{x} \leftarrow v] \stackrel{\text{def}}{=} \{\mu[\mathtt{x} \leftarrow v] \mid \mu \in M\}$.

$(st = \mathtt{skip})$
$\quad \mathrm{F}[\![{}^{l}\mathtt{skip}]\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$

$(st = \mathtt{x} := e)$
$\quad \mathrm{F}[\![{}^{l}\mathtt{x} := e]\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$

$(st = \mathtt{if}\ b\ \mathtt{then}\ st'\ \mathtt{else}\ st'')$
$\quad \mathrm{F}[\![\mathtt{if}\ {}^{l}b\ \mathtt{then}\ st'\ \mathtt{else}\ st'']\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$
$\quad \mathrm{F}[\![st']\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$
$\quad \mathrm{F}[\![st'']\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$

$(st = \mathtt{while}\ b\ \mathtt{do}\ st'\ \mathtt{done})$
$\quad \mathrm{F}[\![\mathtt{while}\ {}^{l}b\ \mathtt{do}\ st']\!]\ \mathtt{done} \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$
$\quad \mathrm{F}[\![st']\!] \stackrel{\text{def}}{=} l$

$(st = st';\ st'')$
$\quad \mathrm{F}[\![st;\ st']\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st'']\!]$
$\quad \mathrm{F}[\![st']\!] \stackrel{\text{def}}{=} \mathrm{I}[\![st'']\!]$
$\quad \mathrm{F}[\![st'']\!] \stackrel{\text{def}}{=} \mathrm{F}[\![st]\!]$

$(st = \mathtt{entry}\ st[st])$
$\quad \mathrm{F}[\![\mathtt{entry}\ st^{l}]\!] \stackrel{\text{def}}{=} l$

Figure 2.15: Final control points.

Transition System Program states are pairs of a program location and an environment, $s \in \Sigma \stackrel{\text{def}}{=} \mathbb{L} \times \mathbb{E}$. The initial control point of a program (or a statement) defines where the execution of the program (statement) starts, namely $\text{I}[\![P]\!]$ and $\text{I}[\![st]\!]$ respectively. Similarly, the final control point of a program (or a statement) defines where the execution of the program (statement) ends, namely $\text{F}[\![P]\!]$ and $\text{F}[\![st]\!]$ respectively. The formal definitions of initial and final control points are given in Figure 2.14 on the side and Figure 2.15 above.

$$\text{I}[\![{}^{l}\texttt{skip}]\!] \stackrel{\text{def}}{=} l$$
$$\text{I}[\![{}^{l}\texttt{x} := e]\!] \stackrel{\text{def}}{=} l$$
$$\text{I}[\![\texttt{if } {}^{l}b \texttt{ then } st \texttt{ else } st']\!] \stackrel{\text{def}}{=} l$$
$$\text{I}[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!] \stackrel{\text{def}}{=} l$$
$$\text{I}[\![st;\ st']\!] \stackrel{\text{def}}{=} \text{I}[\![st]\!]$$
$$\text{I}[\![\texttt{entry } st^{l}]\!] \stackrel{\text{def}}{=} \text{I}[\![st]\!]$$

Figure 2.14: Initial control points.

A program P starts the execution at its initial program location $\text{I}[\![P]\!]$ with any value for the program variables. The set of initial states of a program P is $\mathbb{I} \stackrel{\text{def}}{=} \{\langle \text{I}[\![P]\!], \mu \in \mathbb{E}\rangle \mid \mu \in \mathbb{E}\}$.

The transition relation $\tau \in \Sigma \times \Sigma$ applied to a program P is defined as the transition semantics $\tau[\![P]\!] \in \Sigma \times \Sigma$. Next, we define τ for each program construct.

$$\tau[\![{}^{l}\texttt{skip}]\!] \stackrel{\text{def}}{=} \{(l, \mu) \to (\text{F}[\![{}^{l}\texttt{skip}]\!], \mu) \mid \mu \in \mathbb{E}\}$$

($^{l}\texttt{skip}$) The skip statement does not modify the environment, and control moves from the initial location l to the final location $\text{F}[\![{}^{l}\texttt{skip}]\!]$.

$$\tau[\![{}^{l}\texttt{x} := e]\!] \stackrel{\text{def}}{=} \left\{ (l, \mu) \to (\text{F}[\![{}^{l}\texttt{x} := e]\!], \mu[\texttt{x} \leftarrow v]) \;\middle|\; \mu \in \mathbb{E} \wedge v \in \mathbb{A}[\![e]\!]\mu \right\}$$

($^{l}\texttt{x} := e$) The assignment statement updates the value of the variable x to any possible value of the arithmetic expression e, and control moves from the initial location l to the final location $\text{F}[\![{}^{l}\texttt{x} := e]\!]$.

$$\tau[\![\texttt{if } {}^{l}b \texttt{ then } st \texttt{ else } st']\!] \stackrel{\text{def}}{=} \left\{ (l, \mu) \to (\text{I}[\![st]\!], \mu) \;\middle|\; \mu \in \mathbb{E} \wedge \mathbb{B}[\![b]\!]\mu \right\} \cup \tau[\![st]\!] \cup \left\{ (l, \mu) \to (\text{I}[\![st']\!], \mu) \;\middle|\; \mu \in \mathbb{E} \wedge \neg\mathbb{B}[\![b]\!]\mu \right\} \cup \tau[\![st']\!]$$

($\texttt{if } {}^{l}b \texttt{ then } st \texttt{ else } st'$) The conditional statement moves control from the initial location l to the first statement st for environments that satisfy the boolean condition b, and to the beginning of the second statement st' for environments that do not satisfy the boolean condition; then, st and st' are executed independently.

$$\tau[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!] \stackrel{\text{def}}{=} \left\{ (l, \mu) \to (\text{I}[\![st]\!], \mu) \;\middle|\; \mu \in \mathbb{E} \wedge \mathbb{B}[\![b]\!]\mu \right\} \cup \tau[\![st]\!] \cup \left\{ (l, \mu) \to (\text{F}[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!], \mu) \;\middle|\; \mu \in \mathbb{E} \wedge \neg\mathbb{B}[\![b]\!]\mu \right\}$$

($\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}$) The while statement moves control from the initial location l to the beginning of the statement st for environments that satisfy the boolean condition b, and to the final location $\text{F}[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!]$ for environments that do not satisfy the boolean condition; then, st is executed.

$$\tau[\![st;\ st']\!] \stackrel{\text{def}}{=} \tau[\![st]\!] \cup \tau[\![st']\!]$$

($st;\ st'$) The composition statement executes the first and second statements, and join the transitions. Note that, control moves from the end of st to the beginning of st' because the final label of st is the initial label of st', cf. Figure 2.14 and Figure 2.15.

($\texttt{entry } st^{l}$) The transition relation of a program is the transition relation of its statement.

Definition 2.3.1 (Transition Semantics) *The transition semantics of a program P is the set of all possible transitions that the program can generate:*

$$\tau[\![P]\!] \stackrel{\text{def}}{=} \tau[\![\texttt{entry } st^{l}]\!] \stackrel{\text{def}}{=} \tau[\![st]\!]$$

The initial states of a program P are defined as $\mathbb{I} \stackrel{\text{def}}{=} \{\langle \text{I}[\![P]\!], \mu \in \mathbb{E}\rangle \mid \mu \in \mathbb{E}\}$. The final states are states where the program execution ends without any further transition possible, *i. e.*, $\Omega \stackrel{\text{def}}{=} \{\langle \text{F}[\![P]\!], \mu\rangle \mid \mu \in \mathbb{E}\}$.

Maximal Trace Semantics In this section, we provide the *structural* definition of the maximal trace semantics by induction on the structure of the program. We recall that the maximal trace semantics is the set of all possible traces that a program can generate, a program trace is a non-empty sequence of program states determined by the transition relation. We only consider maximal *well-formed* traces, that is, traces that start from initial states $\mathbb{I}$ of a program P, and either end in a final state Ω or never end. Accordingly, we define the maximal trace semantics

$\Lambda[\![P]\!] \in \wp(\Sigma^{+\infty})$ for programs (and $\Lambda[\![st]\!] \in \wp(\Sigma^{+\infty}) \to \wp(\Sigma^{+\infty})$ for statements). Note that, the function $\Lambda[\![st]\!]$ takes as input a set of traces starting with the final location of the statement st and returns a set of traces starting with the initial location of the statement st. Specifically, according to the transition relation semantics, cf. Definition 2.3.1, given a set of traces $T \in \wp(\Sigma^{+\infty})$ starting with the final location of the statement under evaluation:

$$\Lambda[\![{}^{l}\texttt{skip}]\!]T \stackrel{\text{def}}{=} \left\{ (l,\mu)(\text{F}[\![{}^{l}\texttt{skip}]\!],\mu)\cdot\sigma \;\middle|\; \mu \in \mathbb{E} \wedge \sigma \in \Sigma^{\star\infty} \wedge (\text{F}[\![{}^{l}\texttt{skip}]\!],\mu)\cdot\sigma \in T \right\}$$

(${}^{l}\texttt{skip}$) The trace semantics of a skip statement prepends to the traces of T that starts from the final location of the statement $\text{F}[\![{}^{l}\texttt{skip}]\!]$ (*i.e.*, the traces $(\text{F}[\![{}^{l}\texttt{skip}]\!],\mu)\cdot\sigma \in T$) a single program state with the same environment μ paired with the initial location l.

$$\Lambda[\![{}^{l}\texttt{x} := e]\!]T \stackrel{\text{def}}{=} \left\{ (l,\mu)(\text{F}[\![{}^{l}\texttt{x} := e]\!],\mu[\texttt{x} \leftarrow v])\cdot\sigma \;\middle|\; \mu \in \mathbb{E} \wedge v \in \mathbb{A}[\![e]\!]\mu \wedge \sigma \in \Sigma^{\star\infty} \wedge (\text{F}[\![{}^{l}\texttt{x} := e]\!],\mu[\texttt{x} \leftarrow v])\cdot\sigma \in T \right\}$$

(${}^{l}\texttt{x} := e$) For an assignment statement, to the traces of T that starts from the final location $\text{F}[\![{}^{l}\texttt{x} := e]\!]$ and an updated environment $\mu[\texttt{x} \leftarrow v]$ with the effect of the assignment statement (where v is a value of the arithmetic expression e under the environment μ), the trace semantics prepends a program state with the initial location l and the environment μ *before* update. Note that, traces in maximal trace semantics Λ are built backwards, thus the environment μ is prepended to the beginning of the trace, which is required starting from the updated environment $\mu[\texttt{x} \leftarrow v]$ at the final location $\text{F}[\![{}^{l}\texttt{x} := e]\!]$.

$$\Lambda[\![\texttt{if } {}^{l}b \texttt{ then } st \texttt{ else } st']\!]T \stackrel{\text{def}}{=} \left\{ (l,\mu)(\text{I}[\![st]\!],\mu)\cdot\sigma \;\middle|\; \mu \in \mathbb{E} \wedge \mathbb{B}[\![b]\!]\mu \wedge \sigma \in \Sigma^{\star\infty} \wedge (\text{I}[\![st]\!],\mu)\cdot\sigma \in \Lambda[\![st]\!]T \right\} \cup \left\{ (l,\mu)(\text{I}[\![st']\!],\mu)\cdot\sigma \;\middle|\; \mu \in \mathbb{E} \wedge \mathbb{B}[\![\neg b]\!]\mu \wedge \sigma \in \Sigma^{\star\infty} \wedge (\text{I}[\![st']\!],\mu)\cdot\sigma \in \Lambda[\![st']\!]T \right\}$$

(`if` ${}^{l}b$ `then` st `else` st') The trace semantics of a conditional statement is the union of the traces from the then and the else branch. For the then branch, the trace semantics prepends a program state with the initial location l and the environment μ to the traces of $\Lambda[\![st]\!]$ that start from the initial location of the then branch, *i.e.*, $\text{I}[\![st]\!]$, and satisfies the boolean condition b. Similarly, for the else branch, the trace semantics prepends a program state with the initial location l and the environment μ to the traces of $\Lambda[\![st']\!]$ that start from the initial location of the else branch, *i.e.*, $\text{I}[\![st']\!]$, and do not satisfy the boolean condition b.

$$\Lambda[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!]T \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq}\,\text{F}$$
$$\text{F}(X) \stackrel{\text{def}}{=} \left\{ (l,\mu)(\text{I}[\![st]\!],\mu)\cdot\sigma \;\middle|\; \mu \in \mathbb{E} \wedge \mathbb{B}[\![b]\!]\mu \wedge \sigma \in \Sigma^{\star\infty} \wedge (\text{I}[\![st]\!],\mu)\cdot\sigma \in \Lambda[\![st]\!]X \right\} \cup \left\{ (l,\mu)(\text{F}[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!],\mu)\cdot\sigma \;\middle|\; \mu \in \mathbb{E} \wedge \mathbb{B}[\![\neg b]\!]\mu \wedge \sigma \in \Sigma^{\star\infty} \wedge (\text{F}[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!],\mu)\cdot\sigma \in T \right\}$$

(`while` ${}^{l}b$ `do` st `done`) The trace semantics of a while statement is the least fixpoint of a transformer $\text{F} \in \wp(\Sigma^{+\infty}) \to \wp(\Sigma^{+\infty})$ over the complete lattice $\langle \wp(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^{\infty}, \Sigma^{+} \rangle$. Note that, this is not the complete lattice of traces, but it employs the so-called *computational order*:

$$T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} T_1^{+} \subseteq T_2^{+} \wedge T_1^{\infty} \supseteq T_2^{\infty}$$

The transformer F iteratively prepends the initial location l and the environment μ to the traces provided as argument that start from the initial location of the loop statement and satisfy the boolean condition b. In particular, the transformer F takes as input a set $X \in \wp(\Sigma^{+\infty})$ of traces. Initially, X is the set of all infinite sequences Σ^{∞} (*i.e.*, bottom element of the complete lattice of traces). At each iteration, the transformer joins (i) the traces result from a single iteration of traces in X with (ii) the traces in T that do not satisfy the boolean condition b. Respectively, these two cases are defined as follows:

(i) F prepends the initial location l and the environment μ to the traces of the loop body st, *i.e.*, $\Lambda[\![st]\!]$, that start from the initial location $\text{I}[\![st]\!]$ and satisfy the boolean condition b.

(ii) F prepends the initial location l and the environment μ to the traces in T that start from the final location of the loop statement $\mathrm{F}[\![\texttt{while } {}^{l}b \texttt{ do } st \texttt{ done}]\!]$ and do not satisfy the boolean condition b.

The invariant after the i-th iteration contains the program *traces* starting at the initial location of the loop statement whose prefix consist of 0 up to $i-1$ iterations of the loop and whose suffix is a trace in T, and the *sequences*[6] whose prefix are program traces which consist of $i-1$ iterations. At the limit, the set of traces generated by the transformer F corresponds to executing the loop an arbitrary number of times, until the boolean condition b is false, and then exiting the loop.

(st; st') The trace semantics of a sequential composition of two statements is the (function) composition of the trace semantics of the two statements, backwards starting from st' to st.

($\texttt{entry } st^{l}$) Finally, the trace semantics $\Lambda[\![P]\!] \in \wp(\Sigma^{+\infty})$ of a program P is the trace semantics of the entry statement restricted to the traces starting from the program initial states $\mathbb{I}$, defined from the set of program final states Ω.

6: Note the difference between traces and sequences: a program *trace* is a *sequence* of program states that respects the transition relation.

$$\Lambda[\![st;\ st]\!]T \stackrel{\text{def}}{=} \Lambda[\![st]\!](\Lambda[\![st']\!]T)$$

The initial states of a program P are defined as:

$$\mathbb{I} \stackrel{\text{def}}{=} \{\langle \mathrm{I}[\![P]\!], \mu \in \mathbb{E}\rangle \mid \mu \in \mathbb{E}\}$$

The final states are states where the program execution ends without any further transition possible, *i.e.*,

$$\Omega \stackrel{\text{def}}{=} \{\langle \mathrm{F}[\![P]\!], \mu\rangle \mid \mu \in \mathbb{E}\}$$

Definition 2.3.2 (Maximal Trace Semantics of Programs) *The* maximal trace semantics $\Lambda[\![P]\!] \in \wp(\Sigma^{+\infty})$ *of a program P is defined as:*

$$\Lambda[\![P]\!] = \Lambda[\![\texttt{entry } st^{l}]\!] \stackrel{\text{def}}{=} \left\{ (\mathrm{I}[\![st]\!], \mu) \cdot \sigma \in \Lambda[\![st]\!]\Omega \;\middle|\; \mu \in \mathbb{I} \wedge \sigma \in \Sigma^{\star\infty} \right\}$$

Next, we present the forward and backward reachability semantics of programs. These two semantics will be used to define the abstract semantics and to introduce the abstract domains.

2.3.3 Forward Reachability State Semantics

The forward reachability state semantics $\Lambda^r \in \wp(\Sigma)$ introduced in Example 2.2.2 abstracts the maximal trace semantics $\Lambda \in \wp(\Sigma^{+\infty})$ by collecting reachable states from a given set of initial states. Here, we present the *structural* definition of the forward reachability state semantics by induction on the structure of the program. Given a set of states $S \in \wp(\Sigma)$, for each program statement st, the forward reachability state semantics $\Lambda^r[\![st]\!]$ is defined as:

($\texttt{skip}^{l}$) The reachability semantics of a skip statement maintains the set of environments of S that are paired to program locations starting at the initial location $l = \mathrm{I}[\![\texttt{skip}^{l}]\!]$.Program locations are updated to the final location $\mathrm{F}[\![\texttt{skip}^{l}]\!]$.

$$\Lambda^r[\![\texttt{skip}^{l}]\!]S \stackrel{\text{def}}{=} \{(\mathrm{F}[\![\texttt{skip}^{l}]\!], \mu) \mid (l, \mu) \in S\}$$

($\texttt{x} := e^{l}$) The reachability semantics of an assignment statement updates the environments in S with any value from the arithmetic expression e. Program locations are updated to the final location $\mathrm{F}[\![\texttt{x} := e^{l}]\!]$.

$$\Lambda^r[\![\texttt{x} := e^{l}]\!]S \stackrel{\text{def}}{=} \left\{ (\mathrm{F}[\![\texttt{x} := e^{l}]\!], \mu[\texttt{x} \leftarrow v]) \;\middle|\; (l, \mu) \in S \wedge v \in \mathbb{A}[\![e]\!]\mu \right\}$$

($\texttt{if } {}^{l}b \texttt{ then } st \texttt{ else } st'$) The reachability semantics of a conditional statement is the union of the reachability semantics of the then and the else branch. For the then branch, $\mathbb{B}[\![b]\!]S$ keeps the environments

$$\Lambda^r[\![\texttt{if } {}^{l}b \texttt{ then } st \texttt{ else } st']\!]S \stackrel{\text{def}}{=} \Lambda^r[\![st]\!]\left\{ (\mathrm{I}[\![st]\!], \mu) \;\middle|\; \begin{matrix}(l,\mu) \in S \wedge \\ \mathbb{B}[\![b]\!]\mu\end{matrix} \right\} \cup \Lambda^r[\![st']\!]\left\{ (\mathrm{I}[\![st']\!], \mu) \;\middle|\; \begin{matrix}(l,\mu) \in S \wedge \\ \mathbb{B}[\![\neg b]\!]\mu\end{matrix} \right\}$$

that satisfy the boolean condition b and update them to the initial location $\text{I}[\![st]\!]$, then the reachability semantics of the then branch $\Lambda^r[\![st]\!]$ is computed. Note that, the reachability semantics of the then branch expects control locations at the final location of the then branch (*i.e.*, $\text{F}[\![st]\!]$) which is also the final location of the conditional statement, cf. Figure 2.15, hence the program locations are not updated after the branch evaluation. Similarly, for the else branch, the reachability semantics of the else branch $\Lambda^r[\![st']\!]$ is computed for the environments that do not satisfy the boolean condition b.

$$\Lambda^r[\![\texttt{while } {}^l b \texttt{ do } st \texttt{ done}]\!]S \stackrel{\text{def}}{=} \left\{ (\text{F}[\![\texttt{while } {}^l b \texttt{ do } st \texttt{ done}]\!], \mu) \;\middle|\; (l, \mu) \in \text{lfp}^{\subseteq} \text{F}^r \wedge \mathbb{B}[\![\neg b]\!]\mu \right\}$$

$$\text{F}^r(X) \stackrel{\text{def}}{=} S \cup \Lambda^r[\![st]\!] \left\{ (\text{I}[\![st]\!], \mu) \;\middle|\; \begin{array}{l} (l, \mu) \in X \wedge \\ \mathbb{B}[\![b]\!]\mu \end{array} \right\}$$

(`while` ${}^l b$ `do` st `done`) The reachability semantics of a while statement is the set of states with program locations updated to the final location $\text{F}[\![\texttt{while } {}^l b \texttt{ do } st \texttt{ done}]\!]$ that do not satisfy the boolean condition b from the least fixpoint of a transformer F^r. Program locations are updated to the final location $\text{F}[\![\texttt{while } {}^l b \texttt{ do } st \texttt{ done}]\!]$ because the final location of the loop statement is l, not the final location of the loop body $\text{F}[\![st]\!]$, cf. Figure 2.15. The transformer F^r takes as input a set $X \in \wp(\Sigma)$ of states. Initially, X is the empty set $\emptyset$. At each iteration, the transformer joins S with the reachability semantics of the loop body $\Lambda^r[\![st]\!]$ for the environments in X updated to the initial location $\text{I}[\![st]\!]$ that satisfy the boolean condition b. Program locations are correctly updated by $\Lambda^r[\![st]\!]$ to l, ready for another iteration or the exit of the loop. The least fixpoint of the transformer F^r collects all the states that can reach the final location of the loop statement $\text{F}[\![\texttt{while } {}^l b \texttt{ do } st \texttt{ done}]\!]$ in an arbitrary number of iterations.

$$\Lambda^r[\![st;\ st']\!]S \stackrel{\text{def}}{=} \Lambda^r[\![st']\!](\Lambda^r[\![st]\!]S)$$

(`while` ${}^l b$ `do` st `done`) The reachability semantics of a composition of two statements is the (function) composition of the reachability semantics of the two statements, forward starting from st to st'. Program locations are updated correctly from $\Lambda^r[\![st]\!]$ and $\Lambda^r[\![st']\!]$.

The initial states of a program P are defined as:

$$\mathbb{I} \stackrel{\text{def}}{=} \{\langle \text{I}[\![P]\!], \mu \in \mathbb{E}\rangle \mid \mu \in \mathbb{E}\}$$

(`entry` st^l) Finally, the reachability semantics $\Lambda^r[\![P]\!]$ of a program P is the reachability semantics from the set of initial states $\mathbb{I}$.

$$\Lambda^r[\![\texttt{skip}]\!]M \stackrel{\text{def}}{=} M$$

$$\Lambda^r[\![\texttt{x} := e]\!]M \stackrel{\text{def}}{=} \{M[\texttt{x} \leftarrow v] \mid v \in \mathbb{A}[\![e]\!]M\}$$

$$\Lambda^r[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]M \stackrel{\text{def}}{=} \Lambda^r[\![st]\!](\mathbb{B}[\![b]\!]M) \cup \Lambda^r[\![st']\!](\mathbb{B}[\![\neg b]\!]M)$$

$$\Lambda^r[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]M \stackrel{\text{def}}{=} \mathbb{B}[\![\neg b]\!](\text{lfp}^{\subseteq} \text{F}^r)$$
$$\text{F}^r(X) \stackrel{\text{def}}{=} M \cup \Lambda^r[\![st]\!](\mathbb{B}[\![b]\!]X)$$

$$\Lambda^r[\![st;\ st']\!]M \stackrel{\text{def}}{=} \Lambda^r[\![st']\!](\Lambda^r[\![st]\!]M)$$

$$\Lambda^r[\![\texttt{entry } st]\!] \stackrel{\text{def}}{=} \Lambda^r[\![st]\!]\{\mu \in \mathbb{E} \mid l \in \mathbb{L} \wedge (l, \mu) \in \mathbb{I}\}$$

Figure 2.16: Forward reachability state semantics without program locations.

Definition 2.3.3 (Forward Reachability State Semantics of Programs) *The* forward reachability state semantics $\Lambda^r[\![P]\!] \in \wp(\Sigma)$ *of a program P is defined as:*

$$\Lambda^r[\![P]\!] = \Lambda^r[\![\texttt{entry } st]\!] \stackrel{\text{def}}{=} \Lambda^r[\![st]\!]\mathbb{I}$$

Note that, the reachability semantics presented in this section is strictly less expressive than the maximal trace semantics. For instance, it does not capture non-terminating traces.

As a further abstraction, Figure 2.16 shows the forward reachability state semantics without explicitly handling program locations, in such case, the program states are just the environments, *i.e.*, $\Sigma = \mathbb{E}$. This semantics is often called the *invariance semantics* [63, Section 3.3]. To lighten the notation, we lift to set of environments the arithmetic expression and boolean condition evaluation. Formally, for $M \in \wp(\mathbb{E})$, we define:

$$\mathbb{A}[\![e]\!]M \stackrel{\text{def}}{=} \{v \in \mathbb{V} \mid v \in \mathbb{A}[\![e]\!]\mu \wedge \mu \in M\}$$
$$\mathbb{B}[\![b]\!]M \stackrel{\text{def}}{=} \{\mu \in M \mid \mathbb{B}[\![b]\!]\mu\}$$

[63]: Urban (2015), 'Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes)'

2.3.4 Backward Co-Reachability State Semantics

The backward co-reachability state semantics $\Lambda^{\overline{r}} \in \wp(\Sigma)$ introduced in Example 2.2.3 abstracts the maximal trace semantics $\Lambda \in \wp(\Sigma^{+\infty})$ by collecting the program states that can reach a given set of final states. In this section, we present the *structural* definition of the backward co-reachability state semantics by induction on the structure of the program:

(^{l}`skip`) The backward co-reachability semantics of a skip statement maintains the set of environments of S with program locations updated to the initial location of the skip statement, *i.e.*, the program location l.

($^{l}\mathtt{x} := e$) The backward co-reachability semantics of an assignment statement is the set of states (l, μ) such that the updated environment $\mu[\mathtt{x} \leftarrow v]$ is in S with program location $\mathrm{F}[\![^{l}\mathtt{x} := e]\!]$. The value v is evaluated from the arithmetic expression e under the environment μ, before the assignment.

(`if` ^{l}b `then` st `else` st') Similarly to the forward reachability semantics for the conditional statement, the backward co-reachability semantics is the union of the semantics applied to the branches. Slightly differently, the boolean condition is evaluated *after* the branch evaluation. Program locations are updated to the initial locations of the conditional statement $\mathrm{I}[\![\mathtt{if}\ ^{l}b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]$.

(`while` ^{l}b `do` st `done`) The backward co-reachability semantics of a while statement is the least fixpoint of the transformer $\mathrm{F}^{\overline{r}}$. The transformer joins the state that satisfy the boolean condition b from the backward co-reachability semantics of the loop body $\Lambda^{\overline{r}}[\![st]\!]$ applied to the input of the transformer X, with the set of states S that do not satisfy the boolean condition b. Note that, we update the program locations of the states in S as otherwise when input of the next transformer iterate they would not be considered correct for the loop body as $\mathrm{F}[\![\mathtt{while}\ ^{l}b\ \mathtt{do}\ st\ \mathtt{done}]\!] \neq \mathrm{F}[\![st]\!]$.

(st; st') Similarly to the previous semantics, the backward co-reachability semantics of a sequential composition of two statements is their (backwards) composition.

(`entry` st^{l}) Finally, the backward co-reachability semantics $\Lambda^{\overline{r}}[\![P]\!]$ of a program P is the reachability semantics from the set of final states Ω.

Definition 2.3.4 (Backward Co-Reachability State Semantics of Programs) *The* backward co-reachability state semantics $\Lambda^{\overline{r}}[\![P]\!] \in \wp(\Sigma)$ *of a program P is defined as:*

$$\Lambda^{\overline{r}}[\![P]\!] = \Lambda^{\overline{r}}[\![\mathtt{entry}\ st]\!] \stackrel{\text{def}}{=} \Lambda^{\overline{r}}[\![st]\!]\Omega$$

As done for the forward reachability semantics, Figure 2.17 shows the backward reachability state semantics without explicitly handling program locations, in such case, the program states are just the environments, *i.e.*, $\Sigma = \mathbb{E}$.

$$\Lambda^{\overline{r}}[\![^{l}\mathtt{skip}]\!]S \stackrel{\text{def}}{=} \left\{(l,\mu) \;\middle|\; \begin{array}{l}\mu \in \mathbb{E} \wedge \\ (\mathrm{F}[\![^{l}\mathtt{skip}]\!],\mu) \in S\end{array}\right\}$$

$$\Lambda^{\overline{r}}[\![^{l}\mathtt{x} := e]\!]S \stackrel{\text{def}}{=} \left\{(l,\mu) \;\middle|\; \begin{array}{l}\mu \in \mathbb{E} \wedge v \in \mathbb{A}[\![e]\!]\mu \wedge \\ (\mathrm{F}[\![^{l}\mathtt{x} := e]\!],\mu[\mathtt{x} \leftarrow v]) \in S\end{array}\right\}$$

$$\Lambda^{\overline{r}}[\![\mathtt{if}\ ^{l}b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]S \stackrel{\text{def}}{=} \left\{(l,\mu) \;\middle|\; \begin{array}{l}(\mathrm{I}[\![st]\!],\mu) \in \Lambda^{\overline{r}}[\![st]\!]S \\ \wedge\ \mathbb{B}[\![b]\!]\mu\end{array}\right\} \cup \left\{(l,\mu) \;\middle|\; \begin{array}{l}(\mathrm{I}[\![st']\!],\mu) \in \Lambda^{\overline{r}}[\![st']\!]S \\ \wedge\ \mathbb{B}[\![\neg b]\!]\mu\end{array}\right\}$$

$$\Lambda^{\overline{r}}[\![\mathtt{while}\ ^{l}b\ \mathtt{do}\ st\ \mathtt{done}]\!]S \stackrel{\text{def}}{=} \mathrm{lfp}^{\subseteq}\,\mathrm{F}^{\overline{r}}$$

$$\mathrm{F}^{\overline{r}}(X) \stackrel{\text{def}}{=} \left\{(l,\mu) \;\middle|\; \begin{array}{l}(\mathrm{I}[\![st]\!],\mu) \in \Lambda^{\overline{r}}[\![st]\!]X \\ \wedge\ \mathbb{B}[\![b]\!]\mu\end{array}\right\} \cup \left\{(l,\mu) \;\middle|\; \begin{array}{l}(\mathrm{F}[\![\mathtt{while}\ ^{l}b\ \mathtt{do}\ st\ \mathtt{done}]\!],\mu) \in S \\ \wedge\ \mathbb{B}[\![\neg b]\!]\mu\end{array}\right\}$$

$$\Lambda^{\overline{r}}[\![st;\ st']\!]S \stackrel{\text{def}}{=} \Lambda^{\overline{r}}[\![st]\!](\Lambda^{\overline{r}}[\![st']\!]S)$$

The final states of a program P are defined as:

$$\Omega \stackrel{\text{def}}{=} \{\langle \mathrm{F}[\![P]\!], \mu \in \mathbb{E}\rangle \mid \mu \in \mathbb{E}\}$$

$$\Lambda^{\overline{r}}[\![\mathtt{skip}]\!]M \stackrel{\text{def}}{=} M$$

$$\Lambda^{\overline{r}}[\![\mathtt{x} := e]\!]M \stackrel{\text{def}}{=} \left\{\mu \in \mathbb{E} \;\middle|\; \begin{array}{l}v \in \mathbb{A}[\![e]\!]\mu \wedge \\ \mu[\mathtt{x} \leftarrow v] \in M\end{array}\right\}$$

$$\Lambda^{\overline{r}}[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]M \stackrel{\text{def}}{=} \mathbb{B}[\![b]\!](\Lambda^{\overline{r}}[\![st]\!]M) \cup \mathbb{B}[\![\neg b]\!](\Lambda^{\overline{r}}[\![st']\!]M)$$

$$\Lambda^{\overline{r}}[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]M \stackrel{\text{def}}{=} \mathrm{lfp}^{\subseteq}\,\mathrm{F}^{\overline{r}}$$

$$\mathrm{F}^{\overline{r}}(X) \stackrel{\text{def}}{=} \mathbb{B}[\![\neg b]\!]M \cup \mathbb{B}[\![b]\!](\Lambda^{\overline{r}}[\![st]\!]X)$$

$$\Lambda^{\overline{r}}[\![st;\ st']\!]M \stackrel{\text{def}}{=} \Lambda^{\overline{r}}[\![st]\!](\Lambda^{\overline{r}}[\![st']\!]M)$$

$$\Lambda^{\overline{r}}[\![\mathtt{entry}\ st]\!] \stackrel{\text{def}}{=} \Lambda^{\overline{r}}[\![st]\!]\{\mu \in \mathbb{E} \mid l \in \mathbb{L} \wedge (l,\mu) \in \Omega\}$$

Figure 2.17: Backward reachability state semantics without program location.

2.3.5 Numerical Abstract Domains

In this section, we introduce one of the fundamental concepts of static analysis by abstract interpretation: abstract semantics and abstract domain. The abstract semantics are *computable* semantics that overapproximate the concrete semantics defined in the previous section. The abstract semantics yield an effective algorithm to compute an overapproximation of the program fixpoint, and for this reason the terms abstract semantics and analysis are often used interchangeably. The abstract domains are machine-representable structures that capture some detail of the program behavior. They come with computable operations that allow the abstract analysis to be performed efficiently.

We consider concretization-based abstractions of the form:

$$\langle \wp(\mathbb{E}), \subseteq \rangle \overset{\gamma}{\leftarrow} \langle \mathbb{D}, \sqsubseteq \rangle$$

which provide a sound and computable over-approximation of the concrete semantics. We introduce the concept of abstract domains with a simple example of the integer interval abstract domain [64, 65], called Boxes, w.r.t. the concrete domain of integers, *i.e.*, $\mathbb{V} = \mathbb{Z}$. Hence, concrete environments are mappings from variables to integers, *i.e.*, $\mathbb{E} = \mathbb{V}\mathtt{ar} \to \mathbb{Z}$.

[64]: Cousot et al. (1976), 'Static Determination of Dynamic Properties of Programs'
[65]: Hickey et al. (2001), 'Interval arithmetic: From principles to implementation'

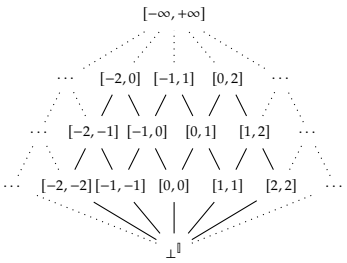


Figure 2.18: The interval lattice.

Integer Interval Abstract Domain This domain abstracts the values of each variable with an interval. First we define interval values and operations on them, then we present the abstract domain. The interval values are defined as:

$$\mathbb{I} \overset{\text{def}}{=} \{[l, u] \mid l \in \mathbb{Z}^{-\infty} \wedge u \in \mathbb{Z}^{+\infty} \wedge l \leq u\} \cup \{\bot^{\mathbb{I}}\}$$

Note that, the interval values are either a closed interval $[l, u]$ with l and u integers such that $l \leq u$, or the bottom element $\bot^{\mathbb{I}}$ that represents the empty set of integers. Therefore, we can easily store the interval values in a machine, we say that $\mathbb{I}$ is *machine-representable*. The partial order $\sqsubseteq^{\mathbb{I}} \in \mathbb{I} \times \mathbb{I}$ checks if an interval is included in another. Formally, for all $x, y \in \mathbb{I}$:

$$x = \bot^{\mathbb{I}} \ \vee \ (x = [l, u] \wedge y = [l', u'] \ \Leftrightarrow \ l' \leq l \wedge u \leq u')$$

Figure 2.18 shows the lattice of interval values. The concretization function $\gamma^{\mathbb{I}} \in \mathbb{I} \to \wp(\mathbb{Z})$ is defined as:

$$\gamma^{\mathbb{I}}(x) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } x = \bot^{\mathbb{I}} \\ \{v \mid l \leq v \leq u\} & \text{if } x = [l, u] \end{cases}$$

concretizing the set of integers between the two bounds of the interval, or the empty set for the bottom element $\bot^{\mathbb{I}}$. Note that, while interval values have a machine-representable form and the ordering relation is effectively computable, the concretization function is not. Next, we proceed with the description of the interval abstract domain.

Abstract Elements The abstract elements of the interval abstract domain are mappings from variables $\mathbb{V}\mathtt{ar}$ to intervals $\mathbb{I}$:

The finite set $\mathbb{V}\mathtt{ar}$ contains the program variables.

$$\mathbb{D}^{\mathbb{I}} \overset{\text{def}}{=} \mathbb{V}\mathtt{ar} \to (\mathbb{I} \setminus \{\bot^{\mathbb{I}}\}) \cup \{\bot^{\mathbb{I}}\}$$

where we enforce a single representation for the bottom element $\bot^{\mathbb{I}}$, that is, if the interval associated to a variable is empty, then abstract domain contains no value for all the variables. As the amount of variables is finite, and the interval values are machine-representable, in turns, the elements of the interval abstract domain are machine-representable.

Partial Order The partial order $\sqsubseteq^{\mathbb{D}^\natural}$ is defined as follows, for all $d_1^\natural, d_2^\natural \in \mathbb{D}^\natural$:

$$d_1^\natural = \bot^{\mathbb{D}^\natural} \vee (d_1^\natural \neq \bot^{\mathbb{D}^\natural} \wedge d_2^\natural \neq \bot^{\mathbb{D}^\natural} \wedge \forall \mathtt{x} \in \mathbb{V}\mathtt{ar}.\ d_1^\natural(\mathtt{x}) \sqsubseteq^{\mathbb{I}} d_2^\natural(\mathtt{x}))$$

An effective algorithm can check the ordering relation between two abstract elements just by comparing the intervals associated to each variable.

Concretization Function The concretization function $\gamma^{\mathbb{D}^\natural} \in \mathbb{D}^\natural \to \wp(\mathbb{E})$ is defined as all the environments that satisfy the interval constraints:

$$\gamma^{\mathbb{D}^\natural}(d^\natural) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } d^\natural = \bot^{\mathbb{D}^\natural} \\ \{\mu \in \mathbb{E} \mid \forall \mathtt{x} \in \mathbb{V}\mathtt{ar}.\ \mu(\mathtt{x}) \in \gamma^{\mathbb{I}}(d^\natural(\mathtt{x}))\} & \text{otherwise} \end{cases}$$

As seen in Section 2.2.4, *soundness* is the foundation of abstract interpretation. An abstract element $d^\natural$ is a sound abstraction of the set of environments M when the concretization of $d^\natural$ contains all the environments in M. Intuitively, the abstract element carries all the information of the concrete states it abstracts.

Definition 2.3.5 (Sound Abstraction) *Given a concrete domain $\langle X, \sqsubseteq_X \rangle$ and an abstract domain $\langle X^\natural, \sqsubseteq_{X^\natural} \rangle$, then $x \in X$ is a* sound abstraction *of $x^\natural \in X^\natural$, if and only if:*

$$x \sqsubseteq_X \gamma(x^\natural)$$

For instance, let the program variables be $\mathtt{x}, \mathtt{y} \in \mathbb{V}\mathtt{ar}$, the concrete domain be $\langle \mathbb{E}, \subseteq \rangle$, and the abstract domain be $\langle \mathbb{D}^\natural, \sqsubseteq^{\mathbb{D}^\natural} \rangle$. Then, the interval $d^\natural = \langle \mathtt{x} \mapsto [1,2], \mathtt{y} \mapsto [0,3] \rangle$ is a sound abstraction of the set of environments $\{\langle \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2 \rangle, \langle \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 3 \rangle\}$ as the concretization of the interval contains both environments. Instead, $d^\natural$ is not a sound abstraction for the environment $\langle \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 4 \rangle$ as the value for the variable $\mathtt{y}$ is not in the interval $[0,3]$.
The notion of soundness extend to operators as well.

Definition 2.3.6 (Sound Operator) *Given a concrete domain $\langle X, \sqsubseteq_X \rangle$ and an abstract domain $\langle X^\natural, \sqsubseteq_{X^\natural} \rangle$, then an abstract operator $f^\natural \in X^\natural \to X^\natural$ is a* sound abstraction *of a concrete operator $f \in X \to X$, if and only if:*

$$\forall x^\natural \in X^\natural.\ f(\gamma(x^\natural)) \sqsubseteq_X \gamma(f^\natural(x^\natural))$$

Abstraction Function Fortunately, the integer[7] interval domain enjoys a Galois connection with the following abstraction function $\alpha^{\mathbb{D}^\natural} \in$

7: For instance, the rational intervals abstract domain does not enjoy a Galois connection when the underlying concrete domain are real numbers, as the concrete interval $[0, \sqrt{2}]$ does not have a best approximation in the rational numbers.

$\wp(\mathbb{E}) \to \mathbb{D}^\natural$:

$$\alpha^{\mathbb{D}^\natural}(M) \stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{D}^\natural} & \text{if } M = \emptyset \\ \lambda \mathsf{x}.[\min X, \max X] & \text{otherwise} \\ \quad \text{where } X = \{\mu(\mathsf{x}) \mid \mu \in M\} & \end{cases}$$

It holds that:

$$\langle \wp(\mathbb{E}), \subseteq \rangle \underset{\alpha^{\mathbb{D}^\natural}}{\overset{\gamma^{\mathbb{D}^\natural}}{\leftrightarrows}} \langle \mathbb{D}^\natural, \sqsubseteq^{\mathbb{D}^\natural} \rangle$$

Galois connections are useful because they give a way to express the most precise abstraction of concrete values.

Remark 2.3.2 Let $\langle X, \sqsubseteq_X \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^natural} \rangle$ be a Galois connection. Then, for any $x \in X$, $\alpha(x)$ is the most precise abstraction of x in $X^\natural$, *i.e.*, the $\sqsubseteq_{X^\natural}$-smallest sound abstraction of x.

This best abstraction also extends to operators.

Definition 2.3.7 (Best Operator Abstraction) *Let* $\langle X, \sqsubseteq_X \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural} \rangle$ *be a Galois connection, and let* $f \in X \to X$ *be a concrete operator. Then, the best abstraction of* f *is* $\alpha \circ f \circ \gamma$.

Interestingly, Definition 2.3.7 above provides a constructive way to define the best abstraction of concrete operators, without requiring a priori definitions and soundness proofs. For instance, the successor operator on environments of integer variables:

$$\begin{aligned} \textsc{Succ} &\in \wp(\mathbb{E}) \to \wp(\mathbb{E}) \\ \textsc{Succ}(M) &\stackrel{\text{def}}{=} \{\mu[\mathsf{x} \leftarrow \mu(\mathsf{x}) + 1] \mid \mathsf{x} \in \mathbb{V}\mathsf{ar} \wedge \mu \in M\} \end{aligned}$$

enjoys the following best abstraction:

$$\begin{aligned} \mathsf{Succ}^\natural &\in \mathbb{D}^\natural \to \mathbb{D}^\natural \\ \mathsf{Succ}^\natural(d^\natural) &\stackrel{\text{def}}{=} \alpha^{\mathbb{D}^\natural}(\textsc{Succ}(\gamma^{\mathbb{D}^\natural}(d^\natural))) \\ &= \begin{cases} \alpha^{\mathbb{D}^\natural}(\textsc{Succ}(\emptyset)) & \text{if } d^\natural = \bot^{\mathbb{D}^\natural} \\ \alpha^{\mathbb{D}^\natural}\left(\left\{ \mu[\mathsf{x} \leftarrow \mu(\mathsf{x}) + 1] \;\middle|\; \mathsf{x} \in \mathbb{V}\mathsf{ar} \wedge \mu \in \gamma^{\mathbb{D}^\natural}(d^\natural) \right\}\right) & \text{otherwise} \end{cases} \\ &= \begin{cases} \bot^{\mathbb{D}^\natural} & \text{if } d^\natural = \bot^{\mathbb{D}^\natural} \\ \lambda \mathsf{x}.[\min_{l \in d^\natural(\mathsf{x})} l + 1, \max_{u \in d^\natural(\mathsf{x})} u + 1] & \text{otherwise} \end{cases} \\ &= \begin{cases} \bot^{\mathbb{D}^\natural} & \text{if } d^\natural = \bot^{\mathbb{D}^\natural} \\ \lambda \mathsf{x}.[l + 1, u + 1] & \text{where } [l, u] = d^\natural(\mathsf{x}) \end{cases} \end{aligned}$$

Unfortunately, often the Galois connection is not available, thus we consider the most general case and do not require a Galois connection for the abstract domain.

Bottom and Top Elements The bottom element $\bot^{\mathbb{D}^\natural} \in \mathbb{D}^\natural$ represents the empty interval for all the variables, and it is concretized to the

empty set of environments:

$$\gamma^{\mathbb{D}^\natural}(\bot^{\mathbb{D}^\natural}) = \emptyset$$

The top element $\top^{\mathbb{D}^\natural} \in \mathbb{D}^\natural$ represents the interval $[-\infty, +\infty]$ for all the variables, and it is concretized to the set of all the environments:

$$\gamma^{\mathbb{D}^\natural}(\top^{\mathbb{D}^\natural}) = \mathbb{E}$$

Join and Meet Operators We derive the best abstraction for the join $\sqcup^{\mathbb{D}^\natural}$ and meet $\sqcap^{\mathbb{D}^\natural}$ operators:

$$\begin{aligned}
\bot^{\mathbb{D}^\natural} \sqcup^{\mathbb{D}^\natural} d^\natural &\stackrel{\text{def}}{=} d^\natural \\
d^\natural \sqcup^{\mathbb{D}^\natural} \bot^{\mathbb{D}^\natural} &\stackrel{\text{def}}{=} d^\natural \\
d_1^\natural \sqcup^{\mathbb{D}^\natural} d_2^\natural &\stackrel{\text{def}}{=} \lambda \mathsf{x}.\alpha^{\mathbb{D}^\natural}(\gamma^{\mathbb{D}^\natural}(d_1^\natural(\mathsf{x})) \cup \gamma^{\mathbb{D}^\natural}(d_2^\natural(\mathsf{x}))) \\
&= \lambda \mathsf{x}.\alpha^{\mathbb{D}^\natural}\left(\left\{ x \in \mathbb{Z} \;\middle|\; \begin{array}{l} [l,u] = d_1^\natural(\mathsf{x}) \wedge [l',u'] = d_2^\natural(\mathsf{x}) \\ \wedge\, l \le x \le u \vee l' \le x \le u' \end{array} \right\}\right) \\
&= \lambda \mathsf{x}.[\min(l,l'), \max(u,u')] \\
&\qquad \text{where } [l,u] = d_1^\natural(\mathsf{x}) \wedge [l',u'] = d_2^\natural(\mathsf{x})
\end{aligned}$$

$$\begin{aligned}
\bot^{\mathbb{D}^\natural} \sqcap^{\mathbb{D}^\natural} d^\natural &\stackrel{\text{def}}{=} \bot^{\mathbb{D}^\natural} \\
d^\natural \sqcap^{\mathbb{D}^\natural} \bot^{\mathbb{D}^\natural} &\stackrel{\text{def}}{=} \bot^{\mathbb{D}^\natural} \\
d_1^\natural \sqcap^{\mathbb{D}^\natural} d_2^\natural &\stackrel{\text{def}}{=} \lambda \mathsf{x}.\alpha^{\mathbb{D}^\natural}(\gamma^{\mathbb{D}^\natural}(d_1^\natural(\mathsf{x})) \cap \gamma^{\mathbb{D}^\natural}(d_2^\natural(\mathsf{x}))) \\
&= \lambda \mathsf{x}.\alpha^{\mathbb{D}^\natural}\left(\left\{ x \in \mathbb{Z} \;\middle|\; \begin{array}{l} [l,u] = d_1^\natural(\mathsf{x}) \wedge [l',u'] = d_2^\natural(\mathsf{x}) \\ \wedge\, l \le x \le u \wedge l' \le x \le u' \end{array} \right\}\right) \\
&= \lambda \mathsf{x}.\begin{cases} [\max(l,l'), \min(u,u')] & \\ & \text{if } \max(l,l') \le \min(u,u') \\ \bot^\natural & \text{otherwise} \end{cases}
\end{aligned}$$

The join operator computes the interval that contains all the values of the two input intervals, while the meet operator computes the interval that contains only the values that are in both input intervals. These are sound operators as the concretization of the result is a sound abstraction of the set of environments that are the join or meet of the concretization of the input abstract elements.

Definition 2.3.8 (Soundness of Join and Meet Operators) *Given two abstract elements $d_1^\natural, d_2^\natural \in \mathbb{D}$, the join $\sqcup \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ and meet $\sqcap \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ operators are sound whenever it holds that:*

$$\gamma(d_1^\natural) \cup \gamma(d_2^\natural) \subseteq \gamma(d_1^\natural \sqcup d_2^\natural)$$
$$\gamma(d_1^\natural) \cap \gamma(d_2^\natural) \subseteq \gamma(d_1^\natural \sqcap d_2^\natural)$$

Note that, Definition 2.3.8 is always satisfied for best abstractions operators, cf. Definition 2.3.7.

Forward and Backward Assignments The abstract semantics of an assignment statement $\mathsf{x} := e$ for the interval abstract domain is defined as:

Def. 2.3.7 (Best Operator Abstraction) *Let $\langle X, \sqsubseteq_X \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle X^\natural, \sqsubseteq_{X^\natural} \rangle$ be a Galois connection, and let $f \in X \to X$ be a concrete operator. Then, the best abstraction of f is $\alpha \circ f \circ \gamma$.*

To update the value of a variable in the interval abstract domain we use:

$$d^\natural[\mathsf{x} \leftarrow [l,u]] \stackrel{\text{def}}{=} \lambda \mathsf{x}'. \begin{cases} [l,u] & \text{if } \mathsf{x}' = \mathsf{x} \\ d^\natural(\mathsf{x}') & \text{otherwise} \end{cases}$$

$$\mathtt{Assign}^{\mathbb{D}^\natural}[\![\mathtt{x} := e]\!]\bot^{\mathbb{D}^\natural} \stackrel{\text{def}}{=} \bot^{\mathbb{D}^\natural}$$
$$\mathtt{Assign}^{\mathbb{D}^\natural}[\![\mathtt{x} := e]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow \mathbb{A}^\natural[\![e]\!]]$$

where arithmetic expressions are evaluated by $\mathbb{A}^\natural \in (\mathbb{D}^\natural \setminus \bot^{\mathbb{D}^\natural}) \to \mathbb{I}$ as follows:

$$\mathbb{A}^\natural[\![v]\!]d^\natural \stackrel{\text{def}}{=} [v, v]$$
$$\mathbb{A}^\natural[\![\mathtt{x}]\!]d^\natural \stackrel{\text{def}}{=} d^\natural(\mathtt{x})$$
$$\mathbb{A}^\natural[\![e + e']\!]d^\natural \stackrel{\text{def}}{=} [l + l', u + u'] \quad \text{where } [l,u] = \mathbb{A}^\natural[\![e]\!]d^\natural \text{ and } [l',u'] = \mathbb{A}^\natural[\![e']\!]d^\natural$$
$$\mathbb{A}^\natural[\![e - e']\!]d^\natural \stackrel{\text{def}}{=} [l - u', u - l'] \quad \text{where } [l,u] = \mathbb{A}^\natural[\![e]\!]d^\natural \text{ and } [l',u'] = \mathbb{A}^\natural[\![e']\!]d^\natural$$
$$\mathbb{A}^\natural[\![\mathtt{rand}(l,u)]\!]d^\natural \stackrel{\text{def}}{=} [l, u]$$

$e ::= v$
$\quad \mid \mathtt{x}$
$\quad \mid e + e$
$\quad \mid e - e$
$\quad \mid \mathtt{rand}(l,u) \quad (l \in \mathbb{Z}^{\pm\infty} \wedge l < u)$

where $v \in \mathbb{Z}$ is an integer constant. In case of the backward evaluation of an assignment statement, we call substitution the operation of replacing the value of a variable with the value of an arithmetic expression that yields before the execution of the assignment. The substitution needs to be sound, *i.e.*, the interval of an assigned variable should overapproximate the precondition for the assignment to be executed resulting in the given postcondition. For instance, the evaluation of a constant assignment $\mathtt{x} := v$ is $[-\infty, +\infty]$ as any possible value could have been assigned to the variable $\mathtt{x}$ before the execution of the assignment, similarly for the assignment of a non-deterministic choice, or another variable different from the assigned one. A similar reasoning applies to the rest of the cases, for the evaluation of complex expressions we need to *reverse* the assignment [66], *i.e.*, we need to find the possible values of the variable before the assignment. Formally, a possible semantics for the substitution is defined as:

[66]: Aman et al. (2020), 'Foundations of Reversible Computation'

$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow e]\!]\bot^{\mathbb{D}^\natural} \stackrel{\text{def}}{=} \bot^{\mathbb{D}^\natural}$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow v]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [-\infty, +\infty]]$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow \mathtt{rand}(l,u)]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [-\infty, +\infty]]$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow \mathtt{x}']\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [-\infty, +\infty]] \quad \text{where } \mathtt{x} \neq \mathtt{x}'$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow \mathtt{x}]\!]d^\natural \stackrel{\text{def}}{=} d^\natural$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow \mathtt{x} + v]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [l - v, u - v]] \quad \text{where } [l,u] = d^\natural(\mathtt{x})$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow v + \mathtt{x}]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [l - v, u - v]] \quad \text{where } [l,u] = d^\natural(\mathtt{x})$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow \mathtt{x} - v]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [l + v, u + v]] \quad \text{where } [l,u] = d^\natural(\mathtt{x})$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow v - \mathtt{x}]\!]d^\natural \stackrel{\text{def}}{=} d^\natural[\mathtt{x} \leftarrow [v - l, v - u]] \quad \text{where } [l,u] = d^\natural(\mathtt{x})$$
$$\mathtt{Subs}^{\mathbb{D}^\natural}[\![\mathtt{x} \leftarrow e]\!]d^\natural \stackrel{\text{def}}{=} \top^{\mathbb{D}^\natural}$$

Note that, we could even further add semantics rules to improve the precision of the domain, for instance, by considering the case where the arithmetic expression is more complex.
The soundness condition of an assignment abstractly interpreted either forward or backward is standard, it is based on the overapproximation of pre/postconditions.

$$\Lambda^r[\![^l \mathtt{x} := e]\!]S \stackrel{\text{def}}{=} \left\{ (\mathrm{F}[\![^l \mathtt{x} := e]\!], \mu[\mathtt{x} \leftarrow v]) \;\middle|\; (l,\mu) \in S \wedge v \in \mathbb{A}[\![e]\!]\mu \right\}$$

Definition 2.3.9 (Soundness of Assignments) *Given an assignment statement* $\mathtt{x} := e$*, the forward* $\mathtt{Assign}[\![\mathtt{x} := e]\!] \in \mathbb{D} \to \mathbb{D}$ *operator is a* sound abstraction *of the forward evaluation of the assignment whenever it holds that:*

$$\forall d^\sharp \in \mathbb{D}.\ \Lambda^r[\![\mathtt{x} := e]\!](\gamma^{\mathbb{D}^\sharp}(d^\sharp)) \ \subseteq\ \gamma^{\mathbb{D}^\sharp}(\mathtt{Assign}[\![\mathtt{x} := e]\!]d^\sharp)$$

$$\Lambda^{\overline{r}}[\![^l \mathtt{x} := e]\!]S \stackrel{\text{def}}{=} \left\{ (l,\mu) \;\middle|\; \mu \in \mathbb{E} \wedge v \in \mathbb{A}[\![e]\!]\mu \wedge (\mathrm{F}[\![^l \mathtt{x} := e]\!], \mu[\mathtt{x} \leftarrow v]) \in S \right\}$$

Definition 2.3.10 (Soundness of Substitution) *Given an assignment statement* $\mathtt{x} := e$*, the backward* $\mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!] \in \mathbb{D} \to \mathbb{D}$ *operator is a* sound abstraction *of the backward evaluation of the assignment whenever it holds that:*

$$\forall d^\sharp \in \mathbb{D}.\ \Lambda^{\overline{r}}[\![\mathtt{x} := e]\!](\gamma^{\mathbb{D}^\sharp}(d^\sharp)) \ \subseteq\ \gamma^{\mathbb{D}^\sharp}(\mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^\sharp)$$

Boolean Expressions Boolean expressions do not need special handle for the forward and backward cases separately, as deciding whether a boolean condition holds or not is independent of the direction of the analysis. Therefore, the evaluation of boolean conditions in the interval abstract domain is defined as:

$$b ::= e \le v \mid e = v \mid b \wedge b \mid \neg b$$

$$\mathtt{Filter}^{\mathbb{D}^\sharp}[\![b]\!]\bot^{\mathbb{D}^\sharp} \stackrel{\text{def}}{=} \bot^{\mathbb{D}^\sharp}$$

$$\mathtt{Filter}^{\mathbb{D}^\sharp}[\![e \le v]\!]d^\sharp \stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{D}^\sharp} & \text{if } l > v \\ [l, \min(u,v)] & \text{otherwise} \end{cases} \quad \text{where } [l,u] = \mathbb{A}^\sharp[\![e]\!]d^\sharp$$

$$\mathtt{Filter}^{\mathbb{D}^\sharp}[\![e = v]\!]d^\sharp \stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{D}^\sharp} & \text{if } l > v \vee u < v \\ [v,v] & \text{otherwise} \end{cases} \quad \text{where } [l,u] = \mathbb{A}^\sharp[\![e]\!]d^\sharp$$

$$\mathtt{Filter}^{\mathbb{D}^\sharp}[\![b \wedge b']\!]d^\sharp \stackrel{\text{def}}{=} \mathtt{Filter}^{\mathbb{D}^\sharp}[\![b]\!]d^\sharp \sqcap^{\mathbb{D}^\sharp} \mathtt{Filter}^{\mathbb{D}^\sharp}[\![b']\!]d^\sharp$$

We can always solve the negation of boolean condition via De Morgan's laws. Therefore, we do not handle them explicitly, instead we first simplify the condition to the base cases, then we add specific rules for the negation of the simple cases. The following

De Morgan's laws:

$$\neg(b \wedge b') = \neg b \vee \neg b'$$
$$\neg(b \vee b') = \neg b \wedge \neg b'$$

are the additional rules:

$$\texttt{Filter}^{\mathbb{D}^\natural}[\![\neg(e \leq v)]\!]d^\natural \stackrel{\text{def}}{=} \texttt{Filter}^{\mathbb{D}^\natural}[\![e > v]\!]d^\natural$$

$$= \begin{cases} \bot^{\mathbb{D}^\natural} & \text{if } u < v \\ [\max(l, v), u] & \text{otherwise} \end{cases} \quad \text{where } [l, u] = \mathbb{A}^\natural[\![e]\!]d^\natural$$

$$\texttt{Filter}^{\mathbb{D}^\natural}[\![\neg(e = v)]\!]d^\natural \stackrel{\text{def}}{=} \texttt{Filter}^{\mathbb{D}^\natural}[\![e \neq v]\!]d^\natural = d^\natural$$

$$\texttt{Filter}^{\mathbb{D}^\natural}[\![b \vee b']\!]d^\natural \stackrel{\text{def}}{=} \texttt{Filter}^{\mathbb{D}^\natural}[\![b]\!]d^\natural \sqcup^{\mathbb{D}^\natural} \texttt{Filter}^{\mathbb{D}^\natural}[\![b']\!]d^\natural$$

The soundness of the boolean conditions is standard, it is defined as the overapproximation of the set of environments that satisfy the boolean condition.

Definition 2.3.11 (Soundness of Boolean Expressions) *Given a boolean condition b, the filter $\texttt{Filter}[\![b]\!] \in \mathbb{D} \to \mathbb{D}$ operator is a* sound abstraction *of the evaluation of the boolean condition whenever it holds that:*

$$\forall d^\natural \in \mathbb{D}.\ \mathbb{B}[\![b]\!](\gamma(d^\natural)) \subseteq \gamma(\texttt{Filter}[\![b]\!]d^\natural)$$

Widening When dealing with loops, the widening operator ensures the termination of the fixpoint computation by over-approximating the result of the iteration. This step is recommended even for concrete domains that does not have infinite ascending chains, as the convergence of the fixpoint computation can be slow. The main idea is that widening enforce fast termination by pushing unstable sequences of iterations to infinite:

$$d^\natural \nabla^{\mathbb{D}^\natural} a^\natural \stackrel{\text{def}}{=} \lambda \mathrm{x}. \begin{cases} [l, u] & \text{if } d^\natural(\mathrm{x}) \sqsubseteq^{\mathbb{I}} a^\natural(\mathrm{x}) \\ [-\infty, u] & \text{if } l' \leq l \\ [l, +\infty] & \text{if } u \leq u' \\ [-\infty, +\infty] & \text{otherwise} \end{cases}$$

where $[l, u] = d^\natural(\mathrm{x})$ and $[l', u'] = a^\natural(\mathrm{x})$

It holds that the widening domain computes an upper bound of the two input intervals:

$$\forall d^\natural, a^\natural \in \mathbb{D}^\natural.\ d^\natural \sqsubseteq^{\mathbb{D}^\natural} d^\natural \nabla^{\mathbb{D}^\natural} a^\natural \wedge a^\natural \sqsubseteq^{\mathbb{D}^\natural} d^\natural \nabla^{\mathbb{D}^\natural} a^\natural$$

and it enforces termination: for any infinite sequence of abstract elements $d_1^\natural, d_2^\natural, \cdots \in \mathbb{D}^\natural$, the sequence $a_1^\natural, a_2^\natural, \ldots$ computed as $a_1^\natural = d_1^\natural$ and $a_{i+1}^\natural = a_i^\natural \nabla^{\mathbb{D}^\natural} d_{i+1}^\natural$ stabilizes after a finite number of iterations, *i.e.*, there exists an $n \in \mathbb{N}_{\geq 0}$ such that $a_n^\natural = a_{n+1}^\natural$.

Definition 2.3.12 (Widening) *Let $\langle \mathbb{D}, \sqsubseteq \rangle$ be an abstract domain. A binary operator $\nabla \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is a* widening *operator whenever it holds that:*

- ∇ *computes an upper bound:*

$$\forall d^\sharp, a^\sharp \in \mathbb{D}.\ d^\sharp \sqsubseteq d^\sharp \nabla a^\sharp \wedge a^\sharp \sqsubseteq d^\sharp \nabla a^\sharp$$

- ∇ *enforces termination: for any infinite sequence of abstract elements* $d_1^\sharp, d_2^\sharp, \cdots \in \mathbb{D}$*, the sequence* $a_1^\sharp, a_2^\sharp, \ldots$ *computed as* $a_1^\sharp = d_1^\sharp$ *and* $a_{i+1}^\sharp = a_i^\sharp \nabla d_{i+1}^\sharp$ *stabilizes after a finite number of iterations, i.e., there exists an* $n \in \mathbb{N}_{\geq 0}$ *such that* $a_n^\sharp = a_{n+1}^\sharp$.

It holds that widenings can indeed be used to approximate least fixpoints in the abstract.

Theorem 2.3.1 *Let* $\langle X, \subseteq \rangle$ *be the concrete and* $\langle \mathbb{D}, \sqsubseteq \rangle$ *be the abstract domain. The function* $f \in X \to X$ *is a monotonic operator and* $g \in \mathbb{D} \to \mathbb{D}$ *is a sound abstraction of* f*, the following iteration:*

$$\begin{aligned} x^0 &\stackrel{\text{def}}{=} \bot \\ x^{i+1} &\stackrel{\text{def}}{=} x^i \nabla g(x^i) \end{aligned}$$

converges in finite time, and its limit x *is a sound abstractino of the least fixpoint* $\mathrm{lfp}\, f$*, i.e.,* $\mathrm{lfp}\, f \subseteq \gamma(x)$.

Narrowing The narrowing operator is the dual of the widening operator and helps to refine the over-approximation of the fixpoint computation after the application of the widening. The narrowing operator for the interval abstract domain is defined as:

$$d^\sharp \Delta^{\mathbb{D}^\sharp} a^\sharp \stackrel{\text{def}}{=} \lambda \mathrm{x}. \begin{cases} [l', u'] & \text{if } l = -\infty \wedge u = +\infty \\ [l', u] & \text{if } l = -\infty \wedge u' \neq +\infty \\ [l, u'] & \text{if } l' \neq -\infty \wedge u = +\infty \\ [l, u] & \text{otherwise} \end{cases}$$
$$\text{where } [l, u] = d^\sharp(\mathrm{x}) \text{ and } [l', u'] = a^\sharp(\mathrm{x})$$

The following states the conditions for the narrowing operator:

Definition 2.3.13 (Narrowing) *Let* $\langle \mathbb{D}, \sqsubseteq \rangle$ *be an abstract domain. A binary operator* $\Delta \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *is a* narrowing *operator whenever it holds that:*

- Δ *computes a lower bound:*

$$\forall d^\sharp, a^\sharp \in \mathbb{D}.\ d^\sharp \Delta a^\sharp \sqsubseteq d^\sharp \wedge d^\sharp \Delta a^\sharp \sqsubseteq a^\sharp$$

- Δ *enforces termination: for any infinite sequence of abstract elements* $d_1^\sharp, d_2^\sharp, \cdots \in \mathbb{D}$*, the sequence* $a_1^\sharp, a_2^\sharp, \ldots$ *computed as* $a_1^\sharp = d_1^\sharp$ *and* $a_{i+1}^\sharp = a_i^\sharp \Delta d_{i+1}^\sharp$ *stabilizes after a finite number of iterations, i.e., there exists an* $n \in \mathbb{N}_{\geq 0}$ *such that* $a_n^\sharp = a_{n+1}^\sharp$.

The narrowing ensures convergence to a better approximation of the fixpoint than only by using the widening. Generally, the total number of abstract iterations is much smaller than the number of concrete iterations.

Above we presented all the characteristics of a numerical (non-relational) abstract domain. Formally, we define an abstract domain as follows:

Definition 2.3.14 (Numerical Abstract Domain) *A* numerical abstract domain $\langle \mathbb{D}, \sqsubseteq \rangle$ *is characterized by:*

- *elements in the abstract domain* $\mathbb{D}$ *are computable-representable;*
- *a partial order* $\sqsubseteq$ *with an effective algorithm to check the ordering;*
- *a concretization function* $\gamma \in \mathbb{D} \to \wp(\mathbb{E})$*, or when possible a Galois connection*[8] $\langle \wp(\mathbb{E}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{D}, \sqsubseteq \rangle$*;*
- *a bottom element* $\bot \in \mathbb{D}$ *such that* $\gamma(\bot) = \emptyset$*;*
- *a top element* $\top \in \mathbb{D}$ *such that* $\gamma(\top) = \mathbb{E}$*;*
- *a sound join* $\sqcup \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *and meet* $\sqcap \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *operator, cf. Definition 2.3.8;*
- *a sound forward assignment* $\texttt{Assign}[\![\texttt{x} := e]\!] \in \mathbb{D} \to \mathbb{D}$ *operator, cf. Definition 2.3.9;*
- *a sound backward substitution* $\texttt{Subs}[\![\texttt{x} \leftarrow e]\!] \in \mathbb{D} \to \mathbb{D}$ *operator, cf. Definition 2.3.10;*
- *a sound filter* $\texttt{Filter}[\![b]\!] \in \to \mathbb{D} \to \mathbb{D}$ *operator, cf. Definition 2.3.11;*
- *a sound widening* $\nabla \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *operator, cf. Definition 2.3.12; and*
- *a sound narrowing* $\Delta \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *operator, cf. Definition 2.3.13.*

8: See Remark 2.2.4 for the case where a Galois connection does not exist.

As we will see in Chapter 4, the operators required for the abstract domain may vary depending on the analysis we are performing. For instance, we may require an operator to remove the information about a variable.

Remark 2.3.3 (Relational Abstract Domains) It holds that the integer interval abstract domain is a numerical abstract domain. Specifically, as it relates each variable singularly to an interval value, we say that such domain is a *non-relational* abstract domain. On the other hand, a *relational* abstract domain yields constraints that relate multiple variables together.

In the following, we report some of the most common numerical abstract domains used in the literature:

Signs [36] The sign domain infers constraints about the sign of the variables: positive, negative, zero, anything, or nothing. The domain is–potentially–the fastest available as simply associate a sign to each variable. Unfortunately, rarely useful in practice.

Powerset [67] The powerset domain associate each variable with a finite set of possible values. This domain is quite precise but becomes quickly intractable as the number as the number of possible values grows.

Excluded Powerset [68] The excluded powerset domain is a variant of the powerset domain that keeps track of the values that a variable cannot definitely take. This domain is able to compute disjunctions precisely but suffers from even higher scalability issues than the precedent.

Intervals [64, 65] The interval domain has been presented throughly above. It offers a good trade-off between precision and scalability.

[36]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'

[67]: Miné (2017), 'Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation'

[68]: Monat et al. (2024), 'Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)'

[64, 65]: Cousot et al., Hickey et al. (1976, 2001), 'Static Determination of Dynamic Properties of Programs', 'Interval arithmetic: From principles to implementation'

[69]: Granger (1991), 'Static Analysis of Linear Congruence Equalities among Variables of a Program'

Congruences [69] The congruence domain infers constraints about the remainders of the variables when divided by a constant. This domain is particularly useful when dealing with modular arithmetic and pointer offset.

These domains above are non-relational as they associate an abstract element to each variable singularly. Next, we present some of the most common relational abstract domains:

Linear Equalities [70] The linear equalities domain infers constraints about the linear relationships between variables.

Polyhedra [56] The polyhedra domain infers constraints about the linear relationships between variables and the convex combinations of the variables. Thanks to its expressiveness, the polyhedra domain is usually the most precise domain available for numerical analyses.

Octagons [71] The octagons domain infers constraints about the relationships between variables of the form $x - y \leq c$. Potentially, one of the best trade-offs between precision and scalability among the relational domains.

[70]: Karr (1976), 'Affine Relationships Among Variables of a Program'

[56]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'

[71]: Miné (2006), 'The octagon abstract domain'

Different abstract domains represent complementary information about the program variables. Therefore, it is common to combine multiple abstract domains to leverage the strengths of each domain. The combination of abstract domains is known as the *reduced product* domain [72].[9] The reduced product domain is defined as the Cartesian product of the abstract domains, with the ordering defined as the pointwise ordering of the abstract elements. Each abstract operator is defined to share information from the individual domains and to refine their abstract elements. The reduced product domain is thus sound by construction and is, at least, always as precise as any of the individual domains. The reduced product domain is particularly useful when the individual domains are complementary, *e.g.* when one domain is good at inferring linear relationships and another domain is good at inferring disjunctions.

[72]: Cousot et al. (1979), 'Systematic Design of Program Analysis Frameworks'

9: Specifically, this domain is called the *partially* reduced product, as the (fully) reduced product is only applicable when the underlying abstract domains feature Galois connections.

Specifically, let $\mathbb{D}_\times$ be the reduced product of two abstract domains, a reduction operator $\phi \in \mathbb{D}_\times \to \mathbb{D}_\times$ enables information sharing between the two domains. Importantly, the reduction operator needs to not modify the concretized states of the two abstract domains, but it is permitted to refine each of the two abstract elements in their respective domains.

Definition 2.3.15 (Soundness of the Reduction Operation) *The operator $\phi \in \mathbb{D}_\times \to \mathbb{D}_\times$ is a* sound reduction *operator between the two domains $\mathbb{D}_1$ and $\mathbb{D}_2$ whenever it holds that, for any $\langle d_1^\natural, d_2^\natural \rangle, \langle a_1^\natural, a_2^\natural \rangle \in \mathbb{D}_\times$:*

$$\begin{aligned}\langle d_1^\natural, d_2^\natural \rangle = \phi(\langle a_1^\natural, a_2^\natural \rangle) \implies & \gamma_\times(\langle d_1^\natural, d_2^\natural \rangle) = \gamma_\times(\langle a_1^\natural, a_2^\natural \rangle) \wedge \\ & \gamma_1(d_1^\natural) \sqsubseteq_1 \gamma_1(a_1^\natural) \wedge \\ & \gamma_2(d_2^\natural) \sqsubseteq_2 \gamma_2(a_2^\natural)\end{aligned}$$

The reduced product is defined as:

Definition 2.3.16 (Reduced Product) *Let $\langle \mathbb{D}_1, \sqsubseteq_1 \rangle$ and $\langle \mathbb{D}_2, \sqsubseteq_2 \rangle$ be two abstract domains, and $\phi \in \mathbb{D}_\times \to \mathbb{D}_\times$ a sound reduction operator, cf. Definition 2.3.15. The* reduced product *domain $\mathbb{D}_\times$ is defined as:*

- $\mathbb{D}_\times = \mathbb{D}_1 \times \mathbb{D}_2$;
- *for $d_1^\natural, d_2^\natural \in \mathbb{D}_1$ and $a_1^\natural, a_2^\natural \in \mathbb{D}_2$, the partial order $\langle d_1^\natural, a_1^\natural \rangle \sqsubseteq_\times \langle d_2^\natural, a_2^\natural \rangle$ holds if and only if $d_1^\natural \sqsubseteq_1 d_2^\natural$ and $a_1^\natural \sqsubseteq_2 a_2^\natural$;*
- *the concretization function $\gamma_\times(\langle d^\natural, a^\natural \rangle) \stackrel{\text{def}}{=} \gamma_1(d^\natural) \cap \gamma_2(a^\natural)$;*
- *if α_1 and α_2 exist, we have an abstraction $\alpha_\times \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ that forms a Galois connection $\langle \wp(\mathbb{E}), \subseteq \rangle \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \langle \mathbb{D}_\times, \sqsubseteq_\times \rangle$. The abstraction is defined as follows: $\alpha_\times(\mu) \stackrel{\text{def}}{=} \langle \alpha_1(\mu), \alpha_2(\mu) \rangle$;*
- *the bottom element $\bot_\times \stackrel{\text{def}}{=} \langle \bot_1, \bot_2 \rangle$;*
- *the top element $\top_\times \stackrel{\text{def}}{=} \langle \top_1, \top_2 \rangle$;*
- *the sound join $\langle d_1^\natural, d_2^\natural \rangle \sqcup_\times \langle a_1^\natural, a_2^\natural \rangle \stackrel{\text{def}}{=} \langle d_1^\natural \sqcup_1 a_1^\natural, d_2^\natural \sqcup_2 a_2^\natural \rangle$;*
- *the sound meet $\langle d_1^\natural, d_2^\natural \rangle \sqcap_\times \langle a_1^\natural, a_2^\natural \rangle \stackrel{\text{def}}{=} \langle d_1^\natural \sqcap_1 a_1^\natural, d_2^\natural \sqcap_2 a_2^\natural \rangle$;*
- *the sound forward assignment:*

$$\begin{aligned} &\texttt{Assign}_\times[\![\texttt{x} := e]\!]\langle d_1^\natural, d_2^\natural \rangle \stackrel{\text{def}}{=} \\ &\quad \langle \texttt{Assign}[\![\texttt{x} := e]\!] d_1^\natural, \texttt{Assign}[\![\texttt{x} := e]\!] d_2^\natural \rangle \end{aligned}$$

- *the sound backward substitution:*

$$\begin{aligned} &\texttt{Subs}\times[\![\texttt{x} \leftarrow e]\!](\langle d_1^\natural, d_2^\natural \rangle) \stackrel{\text{def}}{=} \\ &\quad \langle \texttt{Subs}[\![\texttt{x} \leftarrow e]\!] d_1^\natural, \texttt{Subs}[\![\texttt{x} \leftarrow e]\!] d_2^\natural \rangle \end{aligned}$$

- *the sound filter:*

$$\begin{aligned} &\texttt{Filter}_\times[\![b]\!](\langle d_1^\natural, d_2^\natural \rangle) \stackrel{\text{def}}{=} \\ &\quad \langle \texttt{Filter}[\![b]\!] d_1^\natural, \texttt{Filter}[\![b]\!] d_2^\natural \rangle \end{aligned}$$

- *the widening $\langle d_1^\natural, d_2^\natural \rangle \nabla_\times \langle a_1^\natural, a_2^\natural \rangle \stackrel{\text{def}}{=} \langle d_1^\natural \nabla_1 a_1^\natural, d_2^\natural \nabla_2 a_2^\natural \rangle$.*

Note that, the reduced product defined above applies the reduction operator ϕ after joins and meets but refrains from applying it after a widening. The reason is that the widening operator ensures termination on each individual domain, this is not any more guaranteed when applying the reduction operator between widening steps. The problem is that the reduction operator might refine the abstract elements in such a way that the widening operator does not terminate anymore. For instance, if the reduction would provide finite interval bounds, the widening operator of the interval abstract domain may push these bounds to infinity indefinitely, without ever reaching a fixpoint.

Next, we show how the (sound) operators of the numerical abstract domains can be used to define the abstract semantics of programs. We present both the forward and backward reachability state abstract semantics for a generic numerical abstract domain $\langle \mathbb{D}, \sqsubseteq \rangle$, cf. Definition 2.3.14.

Forward Reachability State Abstract Semantics The forward reachability state abstract semantics $\Lambda^\rightarrow[\![P]\!] \in \mathbb{D} \to \mathbb{D}$ is defined by induction on the structure of the program, as follows:

$\Lambda^\rightarrow[\![\texttt{skip}]\!] d^\natural \stackrel{\text{def}}{=} d^\natural$

(`skip`) The abstract semantics of a skip statement is the identity function on the abstract element.

$\Lambda^{\rightarrow}[\![\texttt{skip}]\!]d^\natural \stackrel{\text{def}}{=} d^\natural$

($^l\texttt{x} := e$) The abstract semantics of an assignment statement is the abstract forward assignment.

$\Lambda^{\rightarrow}[\![\texttt{x} := e]\!]d^\natural \stackrel{\text{def}}{=} \texttt{Assign}[\![\texttt{x} := e]\!]d^\natural$

(`if` b `then` st `else` st') The abstract semantics of a conditional statement is the join of the abstract semantics applied to the two branches.

$\Lambda^{\rightarrow}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]d^\natural \stackrel{\text{def}}{=} \Lambda^{\rightarrow}[\![st]\!](\texttt{Filter}[\![b]\!]d^\natural) \sqcup \Lambda^{\rightarrow}[\![st']\!](\texttt{Filter}[\![\neg b]\!]d^\natural)$

(`while` b `do` st `done`) The while statement represents the most interesting construct. A naïve approach would be to compute an over-approximation of the fixpoint of the loop body, filtered by the loop condition. However, this approach is not guaranteed to terminate in finite time. Instead, thanks to the widening operator, we can compute the abstract iterates least fixpoint in the abstract domain by applying the widening operator to ensure convergence. The least fixpoint computation is then guaranteed to terminate, hopefully in a small number of abstract iterations. The abstract iterates are computed by the limit operator of the abstract transformer $\lim \mathrm{F}^{\mathbb{D}}$, it is defined as follows:

$$\begin{aligned} \Lambda^{\rightarrow}[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]d^\natural &\stackrel{\text{def}}{=} \texttt{Filter}[\![\neg b]\!](\lim \mathrm{F}^{\mathbb{D}}) \\ \mathrm{F}^{\mathbb{D}}(a^\natural) &\stackrel{\text{def}}{=} d^\natural \sqcup \Lambda^{\rightarrow}[\![st]\!](\texttt{Filter}[\![b]\!]a^\natural) \end{aligned}$$

$$\begin{aligned} \lim \mathrm{F}^{\mathbb{D}} &\stackrel{\text{def}}{=} x^k \text{ such that } k \in \mathbb{N}_{\geq 0} \text{ and } x^k \sqsubseteq \mathrm{F}^{\mathbb{D}}(x^k) \\ \text{where } x^0 &\stackrel{\text{def}}{=} \bot \\ x^{i+1} &\stackrel{\text{def}}{=} x^i \nabla \mathrm{F}^{\mathbb{D}}(x^i) \end{aligned}$$

(st; st') The abstract semantics of a composition statement is the composition of the abstract semantics of the two statements, from st to st'.

$\Lambda^{\rightarrow}[\![st;\ st']\!]d^\natural \stackrel{\text{def}}{=} \Lambda^{\rightarrow}[\![st']\!](\Lambda^{\rightarrow}[\![st]\!]d^\natural)$

(`entry` st) The abstract semantics of a program is the abstract semantics of the entry statement starting from the abstract element representing all the states, *i.e.*, the top element $\top$.

Definition 2.3.17 (Forward Reachability State Abstract Semantics) *The* forward reachability state abstract semantics $\Lambda^{\rightarrow}[\![P]\!] \in \mathbb{D} \rightarrow \mathbb{D}$ *for a program* P *is defined as:*

$$\Lambda^{\rightarrow}[\![P]\!] \stackrel{\text{def}}{=} \Lambda^{\rightarrow}[\![\texttt{entry } st]\!] = \Lambda^{\rightarrow}[\![st]\!]\top$$

Def. 2.3.3 (Forward Reachability State Semantics of Programs)

$\Lambda^r[\![\texttt{skip}]\!]M \stackrel{\text{def}}{=} M$

$\Lambda^r[\![\texttt{x} := e]\!]M \stackrel{\text{def}}{=} \{M[\texttt{x} \leftarrow v] \mid v \in \mathbb{A}[\![e]\!]M\}$

$\Lambda^r[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]M \stackrel{\text{def}}{=} \Lambda^r[\![st]\!](\mathbb{B}[\![b]\!]M) \cup \Lambda^r[\![st']\!](\mathbb{B}[\![\neg b]\!]M)$

$\Lambda^r[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]M \stackrel{\text{def}}{=} \mathbb{B}[\![\neg b]\!](\mathrm{lfp}^{\subseteq} \mathrm{F}^r)$, $\mathrm{F}^r(X) \stackrel{\text{def}}{=} M \cup \Lambda^r[\![st]\!](\mathbb{B}[\![b]\!]X)$

$\Lambda^r[\![st;\ st']\!]M \stackrel{\text{def}}{=} \Lambda^r[\![st']\!](\Lambda^r[\![st]\!]M)$

$\Lambda^r[\![\texttt{entry } st]\!] \stackrel{\text{def}}{=} \Lambda^r[\![st]\!]\{\mu \in \mathbb{E} \mid l \in \mathbb{L} \wedge (l, \mu) \in \mathbb{I}\}$

The following result provides the soundness of $\Lambda^{\rightarrow}[\![P]\!]$ with respect to the concrete semantics $\Lambda^r[\![P]\!]$, cf. Definition 2.3.3.

Theorem 2.3.2 *Given a program* P*, the forward reachability state abstract semantics* $\Lambda^{\rightarrow}[\![P]\!]$ *is sound with respect to the forward reachability state semantics* $\Lambda^r[\![P]\!]$ *whenever it holds that:*

$$\Lambda^r[\![P]\!] \subseteq \gamma(\Lambda^{\rightarrow}[\![P]\!])$$

Backward Co-Reachability State Abstract Semantics The *backward* reachability state abstract semantics $\Lambda^{\leftarrow}[\![P]\!] \in \mathbb{D} \rightarrow \mathbb{D}$ is defined by induction on the structure of the program, as follows:

$$\Lambda^{\leftarrow}[\![\mathtt{x} := e]\!]d^\natural \stackrel{\text{def}}{=} \mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^\natural$$

$$\Lambda^{\leftarrow}[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]d^\natural \stackrel{\text{def}}{=} \mathtt{Filter}[\![b]\!](\Lambda^{\leftarrow}[\![st]\!]d^\natural) \sqcup \mathtt{Filter}[\![\neg b]\!](\Lambda^{\leftarrow}[\![st']\!]d^\natural)$$

$$\Lambda^{\leftarrow}[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]d^\natural \stackrel{\text{def}}{=} \lim \mathrm{F}^{\mathbb{D}}$$
$$\mathrm{F}^{\mathbb{D}}(a^\natural) \stackrel{\text{def}}{=} \begin{pmatrix}\mathtt{Filter}[\![\neg b]\!](d^\natural) \sqcup \\ \mathtt{Filter}[\![b]\!](\Lambda^{\leftarrow}[\![st]\!](a^\natural))\end{pmatrix}$$

$$\Lambda^{\leftarrow}[\![st;\ st']\!]d^\natural \stackrel{\text{def}}{=} \Lambda^{\leftarrow}[\![st]\!]\left(\Lambda^{\leftarrow}[\![st']\!]d^\natural\right)$$

(`skip`) The backward semantics of a skip statement is the identity function on the abstract element.

($^{l}\mathtt{x} := e$) The backward semantics of an assignment statement is the substitution of the value of an expression e into the variable `x`.

(`if` b `then` st `else` st') The backward semantics of a conditional statement is the backward semantics of the two branches, then the filter from the condition b is applied and the results of the two branches are joined.

(`while` b `do` st `done`) The backward semantics of a while statement is the limit of the abstract iterates. The negated condition is applied first as $d^\natural$ is the postcondition outside the loop, then the condition is applied to every iterate of the loop body, and the results are joined. The limit is computed by exploiting the widening operator to ensure convergence.

Note that, we could further improve precision of the while statement (for both the forward and backward abstract analysis) by considering the narrowing operator as follows:

$$\begin{aligned}\lim \mathrm{F}^{\mathbb{D}} &\stackrel{\text{def}}{=} y^k \text{ such that } k \in \mathbb{N}_{\geq 0} \text{ and } y^k \sqsubseteq \mathrm{F}^{\mathbb{D}}(y^k)\\ \text{where } y^0 &\stackrel{\text{def}}{=} x\\ y^{i+1} &\stackrel{\text{def}}{=} y^i \nabla \mathrm{F}^{\mathbb{D}}(y^i)\end{aligned}$$

where x is the limit of the abstract iterates using the widening operator. Further improvements can be achieved by delaying the application of the widening with the join operator, or by unrolling the loop body.

(st; st') The backward semantics of a composition statement is the composition of the abstract semantics of the two statements.

(`entry` st) The backward semantics of a program is the abstract semantics of the exit statement starting from the abstract element representing all the states, *i.e.*, the top element $\top$.

Definition 2.3.18 (Backward Co-Reachability State Abstract Semantics) *The* backward co-reachability state abstract semantics $\Lambda^{\leftarrow}[\![P]\!] \in \mathbb{D} \to \mathbb{D}$ *for a program P is defined as:*

$$\Lambda^{\leftarrow}[\![P]\!] \stackrel{\text{def}}{=} \Lambda^{\leftarrow}[\![\mathtt{entry}\ st]\!] = \Lambda^{\leftarrow}[\![st]\!]\top$$

The following result provides the soundness of $\Lambda^{\leftarrow}[\![P]\!]$ with respect to the concrete semantics $\Lambda^{\overline{r}}[\![P]\!]$, cf. Definition 2.3.4.

Theorem 2.3.3 *Given a program P, the backward reachability state abstract semantics $\Lambda^{\leftarrow}[\![P]\!]$ is sound with respect to the backward reachability state semantics $\Lambda^{\overline{r}}[\![P]\!]$ whenever it holds that:*

$$\Lambda^{\overline{r}}[\![P]\!] \subseteq \gamma(\Lambda^{\leftarrow}[\![P]\!])$$

Def. 2.3.4 (Backward Co-Reachability State Semantics of Programs)

$$\Lambda^{\overline{r}}[\![\mathtt{skip}]\!]M \stackrel{\text{def}}{=} M$$
$$\Lambda^{\overline{r}}[\![\mathtt{x} := e]\!]M \stackrel{\text{def}}{=} \{\mu \in \mathbb{E} \mid v \in \mathbb{A}[\![e]\!]\mu \wedge \mu[\mathtt{x} \leftarrow v] \in M\}$$
$$\Lambda^{\overline{r}}[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]M \stackrel{\text{def}}{=} \mathbb{B}[\![b]\!](\Lambda^{\overline{r}}[\![st]\!]M) \cup \mathbb{B}[\![\neg b]\!](\Lambda^{\overline{r}}[\![st']\!]M)$$
$$\Lambda^{\overline{r}}[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]M \stackrel{\text{def}}{=} \mathrm{lfp}^{\subseteq}\, \mathrm{F}^{\overline{r}}$$
$$\mathrm{F}^{\overline{r}}(X) \stackrel{\text{def}}{=} \mathbb{B}[\![\neg b]\!]M \cup \mathbb{B}[\![b]\!](\Lambda^{\overline{r}}[\![st]\!]X)$$
$$\Lambda^{\overline{r}}[\![st;\ st']\!]M \stackrel{\text{def}}{=} \Lambda^{\overline{r}}[\![st]\!](\Lambda^{\overline{r}}[\![st']\!]M)$$
$$\Lambda^{\overline{r}}[\![\mathtt{entry}\ st]\!] \stackrel{\text{def}}{=} \Lambda^{\overline{r}}[\![st]\!]\{\mu \in \mathbb{E} \mid l \in \mathbb{L} \wedge (l, \mu) \in \Omega\}$$

3 Input Data Usage

In this chapter, we introduce the notion of input data usage as proposed by [42]. In particular, we include output abstractions in the definition of an unused input variable, obtaining a more general definition. Through this chapter, we study relations among abstract non-interference and the unused predicate and their use in the context of non-deterministic programs. Then, we present a *sound and complete* hierarchy of semantics that contains only, and exactly, the information needed to reason about the usage of input variables. Finally, we show a *sound* abstraction of the unused property collecting syntactic dependencies among variables. Later in the next chapters, we will define a quantitative measure of usage of input variables, extending the qualitative notion presented in this chapter.

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

Dans ce chapitre, nous introduisons la notion d'utilisation des données d'entrée telle que proposée par Urban and Müller [42]. En particulier, nous incluons des abstractions de sortie dans la définition d'une variable d'entrée non utilisée, obtenant ainsi une définition plus générale. Tout au long de ce chapitre, nous étudions les relations entre la non-interférence abstraite et le prédicat de non-utilisation, ainsi que leur utilisation dans le contexte des programmes non déterministes. Ensuite, nous présentons une hiérarchie de sémantiques correcte et complète *qui contient uniquement, et exactement, les informations nécessaires pour raisonner sur l'utilisation des variables d'entrée. Enfin, nous montrons une abstraction* correcte *de la propriété de non-utilisation en collectant les dépendances syntaxiques entre les variables. Dans les prochains chapitres, nous définirons une mesure quantitative de l'utilisation des variables d'entrée, en étendant la notion qualitative présentée dans ce chapitre.*

3.1 Input Data Usage

Originally introduced by Urban and Müller [42], the notion of input data usage consists in the predicate $\textsc{Unused}_{\mathtt{W}}$ to determine whether the input variables $\mathtt{W} \in \wp(\Delta)$ are used in the computation of the program's output. Given the semantics of a program P, the predicate $\textsc{Unused}_{\mathtt{W}}$ holds whenever all the input variables in $\mathtt{W}$ are not used by the program P.

Definition 3.1.1 (Unused) *Given a program P and the set of input variables of interest $\mathtt{W} \in \wp(\Delta)$, the variables $\mathtt{W}$ are* unused *if and only if the following predicate holds:*

$$
\begin{aligned}
&\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!]) \overset{def}{\Leftrightarrow} \\
&\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \implies \\
&\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta \setminus \mathtt{W}} \sigma_0 \ \wedge\ \sigma'_0(\mathtt{W}) = v \ \wedge\ \sigma_\omega = \sigma'_\omega
\end{aligned}
$$

The set Δ contains the input variables of the program P, cf. Section 2.3.1.

Given a program P, its semantics is defined by the trace semantics $\Lambda[\![P]\!] \in \wp(\Sigma^{+\infty})$, that is, the set of finite and infinite traces of the program P, cf. Section 2.3.2.

Note that, program states Σ are tuples of locations $\mathbb{L}$ and values of variables, cf. Section 2.3.1.

Let $\sigma \in \Sigma^{+\infty}$ be a trace, σ_0 refers to the variable's values of the initial state and $\sigma_\omega \in \Sigma^{\perp}$ to the final ones if σ is of finite length, otherwise $\perp$ for non-terminating traces.

Given a set of variables $\mathtt{W} \in \wp(\Delta)$, the notation $\sigma_0(\mathtt{W}) \in \mathbb{V}^{|\mathtt{W}|}$ refers to the initial values of the variables in $\mathtt{W}$ in the trace σ, where $\mathbb{V}^{|\mathtt{W}|}$ is the vector of values for $|\mathtt{W}|$ variables.

From two states $s, s' \in \Sigma$, the notation $s =_{\Delta \setminus \mathtt{W}} s'$ denotes that the values of the variables in $\Delta \setminus \mathtt{W}$ are equal in both states, *i.e.*, $\forall \mathtt{x} \in \Delta \setminus \mathtt{W}$ it holds that $s(\mathtt{x}) = s'(\mathtt{x})$. Locations are not considered in the equality between states.

Intuitively, an input variable $\texttt{i} \in \texttt{W}$ is unused if all feasible outcomes σ_ω are feasible from all possible initial values of $\texttt{i}$. That is, for all possible initial values v that differ from the initial value of $\texttt{i}$ in the trace σ, there exists another trace σ' with initial value v for $\texttt{i}$ that leads to the same output σ_ω.

Prog. 3.1: Program to check if a student passed the school year.

```
1 is_passed(eng, math, sci, plus):
2   passing = True;
3   if not eng:
4     eng = False;
5   if not math:
6     passing = plus;
7   if not math:
8     passing = plus;
9
```

The notation

$$(l_0, \langle v_1, v_2\rangle) \longrightarrow \cdots \longrightarrow (l_\omega, \langle v_1', v_2'\rangle)$$

denotes a trace that starts with the state $\langle v_1, v_2\rangle$ at the initial location l_0, referring to the initial values of the variables `sci` and `passing` respectively, and ends with the final state $\langle v_1', v_2'\rangle$ at the last program location l_ω. For brevity, we omit the intermediate states of the trace.

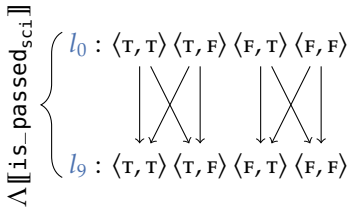


Figure 3.1: Graphical representation of the trace semantics of the Program 3.1 considering only the variables `sci` and `passing`.

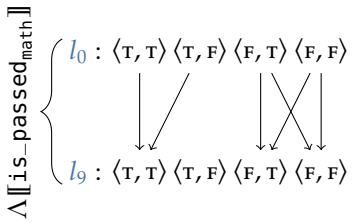


Figure 3.2: Graphical representation of the trace semantics of the Program 3.1 considering only the variables `math` and `passing`.

Prog. 3.2: Syntactic versus semantic usage of the input variable `x`.

```
1 x_plus_rand(x):
2   return x + rand();
3
```

Example 3.1.1 Let us consider the example program presented in Urban and Müller [42], cf. Program 3.1. Based on the boolean-valued input variables `eng`, `math`, `sci`, and `plus`. The program should determine whether a student has passed the year or not, and store the result in the variable `passing`. Additionally, the student is allowed a bonus, cf. the variable `plus`, to help with math and science. However, there are two mistakes in the program: the statement inside the first conditional should be `passing = False` (cf. Line 4), and the third condition should be `not sci` instead of `not math` (cf. Line 7). These two mistakes cause the variables `eng` and `sci` to be unused by Program 3.1. Let us consider the input variable `sci`. The trace semantics of the program simplified to consider only the variables `sci` and `passing` and omitting intermediate state computations is:

$$\Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!] = \left\{ \begin{array}{l} (l_0, \langle \textsc{t}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{t}, \textsc{t} \rangle), \\ (l_0, \langle \textsc{t}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{t}, \textsc{f} \rangle), \\ (l_0, \langle \textsc{f}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{f}, \textsc{t} \rangle), \\ (l_0, \langle \textsc{f}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{f}, \textsc{f} \rangle) \end{array} \right\}$$

where the symbol $*$ denotes any boolean value, the location l_0 is the initial location (Line 2), and l_9 is the last location (Line 9). Figure 3.2 shows a graphical representation of the trace semantics $\Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!]$. The input variable `sci` is unused, since each result value for `passing` is feasible from all possible initial values of `sci`.

Instead, if we consider the variable `math`, the simplified trace semantics considering only the variables `math` and `passing` is:

$$\Lambda[\![\texttt{is\_passed}_{\texttt{math}}]\!] = \left\{ \begin{array}{l} (l_0, \langle \textsc{t}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{t}, \textsc{t} \rangle), \\ (l_0, \langle \textsc{f}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{f}, \textsc{t} \rangle), \\ (l_0, \langle \textsc{f}, * \rangle) \longrightarrow \cdots \longrightarrow (l_9, \langle \textsc{f}, \textsc{f} \rangle) \end{array} \right\}$$

Figure 3.1 shows a graphical representation of the trace semantics $\Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!]$. The input variable `math` is used, since the result value for `passing` is not feasible from all possible initial values of `math`. Indeed, only the initial value $\langle \textsc{f}, * \rangle$ yields the result value $\textsc{f}$ for `passing` (in the final state $(l_9, \langle \textsc{f}, \textsc{f} \rangle)$).

Example 3.1.2 With this second example we highlight the difference between the syntactic and semantic meaning of using a variable. Indeed, a variable could be syntactically used in the program, *i.e.*, present in the code, but it could be semantically unused, *i.e.*, it does not affect the computation of the program. Consider Program 3.2, where the input variable `x` is a non-negative integer. The variable `x` is syntactically

used by the program, as it is present in the return statement. However, the variable x is semantically unused if we consider machine integers wrapped up to max_int $\in \mathbb{N}$. This difference can be clearly seen by studying the trace semantics of the program. In case of machine integers the trace semantics is, reported also in Figure 3.3:

$$\Lambda[\![\texttt{x\_plus\_rand}_{\texttt{max\_int}}]\!] = \{(l_0, \langle x \rangle) \rightarrow \cdots \rightarrow (l_3, \langle x + r \mod \texttt{max\_int} \rangle) \mid x, r \in \mathbb{N}\}$$

where $x \in \mathbb{N}$ denotes the value of the input variable x and $r \in \mathbb{N}$ a random integer from the rand() function. In this case, any outcome value $x + r \mod \texttt{max\_int}$ is feasible from any initial value of x. Even if x is bigger than a given outcome z, the result is still feasible as we can take a positive $r = \texttt{max\_int} - x + z$ to make the wrapped sum $x + r \mod \texttt{max\_int} = z$.

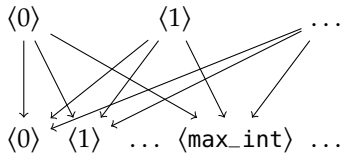


Figure 3.3: Trace semantics of the Program 3.2 bounded to machine integers.

On the other hand, if we consider unbounded positive integers, the trace semantics is:

$$\Lambda[\![\texttt{x\_plus\_rand}_{\mathbb{N}}]\!] = \{(l_0, \langle x \rangle) \rightarrow \cdots \rightarrow (l_3, \langle x + r \rangle) \mid x, r \in \mathbb{N}\}$$

Figure 3.4 shows a graphical representation of the trace semantics of the program Program 3.2 considering unbounded positive integers. In such instance, for any outcome value $x + r$, only initial values of x that are equal or smaller than $x + r$ may generate such outcome. For instance, if the outcome is 42, an initial value of 38 is feasible as it exists a random value (*e.g.*, $r = 4$) that makes the sum $38 + 4 = 42$. Instead, an initial value of 43 is not feasible as there is no positive random value r that makes the sum $43 + r = 42$.

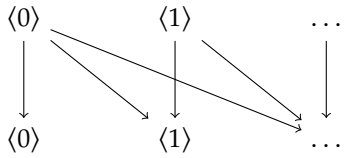


Figure 3.4: Trace semantics of the Program 3.2.

In conclusion, the variable x is syntactically used, but it may be semantically unused as it does not affect the computation of the program $\texttt{x\_plus\_rand}_{\texttt{max\_int}}$.

Definition 3.1.1 (Unused) determines whether a given input variable is used or not, even in the presence of non-deterministic programs. [1] Furthermore, the next example shows that the unused predicate is termination-aware. Indeed, termination is a possible outcome of a program, and thus a variable should be unused if it does not affect the termination as well as the result of the program.

1: In Example 3.1.1 the trace semantics where we consider subsets of input variables, cf. $\Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!]$ and $\Lambda[\![\texttt{is\_passed}_{\texttt{math}}]\!]$, are two non-deterministic sets of traces. This can be easily seen by considering the two traces:

$$(l_0, \langle \text{T}, \text{T} \rangle) \rightarrow \cdots \rightarrow (l_9, \langle \text{T}, \text{T} \rangle) \in \Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!]$$
$$(l_0, \langle \text{T}, \text{T} \rangle) \rightarrow \cdots \rightarrow (l_9, \langle \text{T}, \text{F} \rangle) \in \Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!]$$

where both start from the same initial state $\langle \text{T}, \text{T} \rangle$ and have different outcomes.

Example 3.1.3 Consider the Program 3.3 defined on the side. The program does not terminate for positive values of x (cf. Line 3). Instead, it terminates returning a random value (cf. Line 4) if x is negative or equal to 0. The trace semantics is the following:

$$\Lambda[\![\texttt{non\_termination}]\!] = \left\{ \begin{array}{l} (l_0, \langle < \rangle) \rightarrow \cdots \rightarrow (l_5, \langle * \rangle), \\ (l_0, \langle 0 \rangle) \rightarrow \cdots \rightarrow (l_5, \langle * \rangle), \\ (l_0, \langle > \rangle) \rightarrow \ldots \end{array} \right\}$$

where the symbol < denotes a negative value and > a positive one. When the program is provided a positive value for x, it does not terminate and the respective traces are all of infinite length.

Prog. 3.3: Program that does not terminate for positive values of x.

```
1 non_termination(x):
2   while x > 0:
3     skip;
4   return rand();
5
```

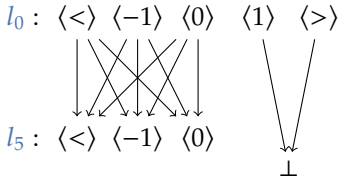


Figure 3.5: Trace semantics of the Program 3.3. The symbol ⊥ denotes non-terminating traces.

In this example we conclude that the variable x is used since the non-termination is feasible only from positive values of x. Otherwise, if we remove non-terminating traces from $\Lambda[\![\texttt{non\_termination}]\!]$, the variable x is then unused as it would be totally non-deterministic the outcome of the program.

Note that, the $\textsc{Unused}_{\mathtt{W}}$ predicate (Definition 3.1.1) is equivalent to checking whether each variable separately is unused in the program.

Lemma 3.1.1 *For all program P and set of variables $\mathtt{W} \in \wp(\Delta)$, it holds that:*

$$\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!]) \iff \forall \mathtt{i} \in \mathtt{W}.\ \textsc{Unused}_{\{\mathtt{i}\}}(\Lambda[\![P]\!])$$

Proof. Let P be a program and $\mathtt{W} \in \wp(\Delta)$ the set of variables of interest. To show ($\Rightarrow$), we assume that $\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!])$ holds. We need to prove that for all $\mathtt{i} \in \mathtt{W}$, $\sigma \in \Lambda[\![P]\!]$, and $v \in \mathbb{V}$ such that $\sigma_0(\mathtt{i}) \neq v$, there exists a trace $\sigma' \in \Lambda[\![P]\!]$ such that $\sigma'_0(\mathtt{i}) = v$, $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$, and $\sigma_\omega = \sigma'_\omega$ hold. By the definition of $\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!])$, we have that $\forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \Rightarrow \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0 \wedge \sigma'_0(\mathtt{W}) = v \wedge \sigma_\omega = \sigma'_\omega$. By expanding the definition we obtain:

Given a vector of n values $v \in \mathbb{V}^n$, we denote the j-th value as v_j.

$$\begin{aligned}
&\forall \sigma \in \Lambda[\![P]\!], v_1 \in \mathbb{V}, \ldots, v_{|\mathtt{W}|} \in \mathbb{V}.\\
&\quad \exists j \leq |\mathtt{W}|.\ \sigma_0(\mathtt{W}_j) \neq v_j \Rightarrow\\
&\qquad \exists \sigma' \in \Lambda[\![P]\!].\ \forall j \leq |\mathtt{W}|.\ \sigma'_0(\mathtt{W}_j) = v_j \wedge\\
&\qquad\quad \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0 \wedge \sigma_\omega = \sigma'_\omega
\end{aligned}$$

Thus, to prove that it exists a trace $\sigma' \in \Lambda[\![P]\!]$ such that $\sigma'_0(\mathtt{i}) = v$, $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$, and $\sigma_\omega = \sigma'_\omega$ we can take the trace σ' that satisfies the above property where the j-th variable of $\mathtt{W}$ is $\mathtt{i}$, and we pick the values $v_1 \in \mathbb{V}, \ldots, v_{|\mathtt{W}|} \in \mathbb{V}$ such that $\sigma_0(\mathtt{W}_j) \neq v_j$ but $\sigma_0((\mathtt{W})_l) = v_l$ for all the other indices l. As a consequence, from $\sigma'_0(\mathtt{W}_j) = v_j \wedge \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0$ it follows that $\sigma'_0(\mathtt{i}) = v$, $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$ since all the values of variables in $\mathtt{W} \setminus \{\mathtt{i}\}$ and $\Delta \setminus \mathtt{W}$ are equal in both σ and σ'. Hence, σ' is the trace that satisfies $\textsc{Unused}_{\{\mathtt{i}\}}$, concluding ($\Rightarrow$).

To show ($\Leftarrow$), we assume that $\forall \mathtt{i} \in \mathtt{W}.\ \textsc{Unused}_{\{\mathtt{i}\}}(\Lambda[\![P]\!])$ holds. We need to prove that for any trace $\sigma \in \Lambda[\![P]\!]$ and value $v \in \mathbb{V}^{|\mathtt{W}|}$ such that $\sigma_0(\mathtt{W}) \neq v$ it exists $\sigma' \in \Lambda[\![P]\!]$ such that $\sigma'_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma_0$, $\sigma'_0(\mathtt{W}) = v$, and $\sigma_\omega = \sigma'_\omega$ hold. This direction of the implication is less intuitive than the previous one as it requires to show that no changes to multiple variables at the same time can affect the outcome of the program. Thus, we focus on values $v \in \mathbb{V}^{|\mathtt{W}|}$ that differ from $\sigma_0(\mathtt{W})$ in more than one index, let us call this set of indices that differ as $D \in \wp(\{j \in \mathbb{N} \mid j \leq |\mathtt{W}|\})$. For each $j \in D$, by hypothesis it exists a trace $\sigma^j \in \Lambda[\![P]\!]$ such that $\sigma^j_0 =_{\Delta\setminus\{\mathtt{W}_j\}} \sigma_0$, $\sigma^j_0(\mathtt{W}_j) = v_j$, and $\sigma_\omega = \sigma^j_\omega$. Then, by considering σ^j it has to exist another trace $\sigma^{j,l} \in \Lambda[\![P]\!]$ such that $\sigma^{j,l}_0 =_{\Delta\setminus\{\mathtt{W}_j,(\mathtt{W})_l\}} \sigma_0$, $\sigma^{j,l}_0(\mathtt{W}_j, (\mathtt{W})_l) = [v_j\ v_l]$, and $\sigma_\omega = \sigma^{j,l}_\omega$ where $l \in D$. By repeating this process for all the indices in D, we obtain a trace σ' that satisfies the property of $\textsc{Unused}_{\mathtt{W}}$. Hence concluding ($\Leftarrow$). □

In the following, we prefer the formalization using sets of variables as originally proposed in Urban and Müller [42] and Urban et al. [73].

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'
[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

3.2 Abstract Input Data Usage

Inspired by the definition of abstract non-interference (ANI) [43, 74], reported in Definition 3.2.1, we introduce an abstracted version of the unused predicate. That is, a version of the unused predicate that abstracts the output states, allowing for a more general definition of the unused property. Later, we will define the quantitative measure of input data usage, and use the abstraction of output states to determine numerical values from output states.

[43]: Giacobazzi et al. (2018), 'Abstract Non-Interference: A Unifying Framework for Weakening Information-flow'
[74]: Mastroeni et al. (2023), 'Domain Precision in Galois Connection-Less Abstract Interpretation'

Def. 3.2.1 (Abstract Non-Interference) *The abstract non-interference predicate* ANI *holds if, for any two traces σ and σ':*

$$\eta(\sigma_0) = \eta(\sigma'_0) \implies \rho(\sigma_\omega) = \rho(\sigma'_\omega)$$

where η and ρ are upper closure operators on program states to abstract input and output states respectively.

We define the abstraction of output states, called output observer, as an upper closure operator over states.

Definition 3.2.2 (Output Observer) *Given a program P, an* output observer $\rho \in \Sigma^\perp \to \Sigma^\perp$ *is an upper closure operator (Definition 2.1.7) that abstracts the output states.*

Def. 2.1.7 (Upper Closure Operator) *An* upper closure operator *on a partially ordered set $\langle X, \subseteq \rangle$ is an operator $\rho \in X \to X$ which is monotone, idempotent, and extensive (i.e., $x \subseteq \rho(x)$ for all $x \in X$).*

We employ the output observer to define an abstract version of the unused predicate, called $\text{AU}_{\text{NUSED}_W}$, as follows:

Definition 3.2.3 (Abstract Unused) *Given a program P, an output observer $\rho \in \Sigma^\perp \to \Sigma^\perp$, and the input variables of interest $W \in \wp(\Delta)$, the variables W are (abstractly)* unused *if and only if the following predicate holds:*

$$\begin{aligned}&\text{AU}_{\text{NUSED}_W}(\Lambda[\![P]\!]) \overset{def}{\iff} \\ &\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|W|}.\ \sigma_0(W) \neq v \implies \\ &\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta \setminus \{i\}} \sigma_0 \ \wedge\ \sigma'_0(W) = v \ \wedge\ \rho(\sigma_\omega) = \rho(\sigma'_\omega)\end{aligned}$$

This abstract unused predicate allows for further abstractions of the output states, which is weaker than the original definition of the unused property. For instance, the output observer could identify which variable is to consider as the outcome of the program and abstract away the other variables. Another use case could be to abstract the output states into the parity of their variables, or even into the sign. In such case, a variable could be used with respect to Definition 3.1.1 (Unused) but unused with respect to Definition 3.2.3 (Abstract Unused).

Example 3.2.1 The Program 3.4 doubles the values of the input variable x. If we consider Definition 3.1.1 (Unused), the variable x is used as for any trace we consider, *e.g.*, the output 4 from an input value of $x = 2$, it does not exist a trace that starts with $x > 2$ and terminates with output 4. Instead, by considering the parity of the output value, the Program 3.4 does not use the variable x as the only possible outcome is "even."

Prog. 3.4: The variable x is used but not abstractly used.

```
1 parity(x):
2   return 2 * x;
```

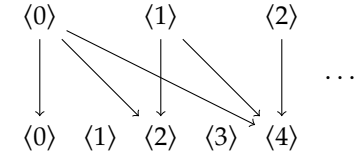


Figure 3.6: Trace semantics of Program 3.4 without abstraction.

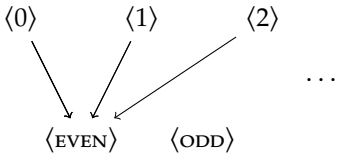


Figure 3.7: Trace semantics of Program 3.4 with parity abstraction applied to the output states.

Definition 3.2.3 (Abstract Unused) can also be seen as a potential abstract non-interference definition working with non-deterministic programs. As a drawback, we lose the input abstraction: the tread-off allows non-determinism but does not permit input state abstractions in the sense of abstract non-interference. The reason is that to take into account non-determinism, for any value of the input variables W, we need to consider all possible traces that start from a different initial value. Such

low-level detail cannot be captured by the input abstraction employed in the definition of abstract non-interference.

The next result shows that the abstract unused predicate is equivalent to the original unused when the output observer is the identity function.[2]

2: The identity function returns its parameter without any modification, *i.e.*, $\mathrm{id}(x) = x$.

Proposition 3.2.1 (Unused Equivalence) *Whenever* $\rho = \mathrm{id}$, *it holds that:*

$$\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!]) \;\Leftrightarrow\; \textsc{AUnused}_{\mathtt{W}}(\Lambda[\![P]\!])$$

Proof. The proof is straightforward by the definition of the abstract unused predicate, Definition 3.2.3, and the unused predicate, Definition 3.1.1. Indeed, the output observer $\rho = \mathrm{id}$ does not abstract the output states, and thus the two predicates are equivalent. □

Note that, $\textsc{Unused}_{\mathtt{W}}$ is the unused predicate, cf. Definition 3.1.1 (Unused), and $\textsc{AUnused}_{\mathtt{W}}$ is the abstract unused predicate, cf. Definition 3.2.3 (Abstract Unused).

Def. 3.2.3 (Abstract Unused)

$$\begin{aligned}&\textsc{AUnused}_{\mathtt{W}}(\Lambda[\![P]\!]) \overset{def}{\Leftrightarrow}\\ &\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \;\Rightarrow\\ &\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta\setminus \mathtt{W}} \sigma_0 \wedge\\ &\qquad \sigma'_0(\mathtt{W}) = v \wedge\\ &\qquad \rho(\sigma_\omega) = \rho(\sigma'_\omega)\end{aligned}$$

Def. 3.1.1 (Unused)

$$\begin{aligned}&\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!]) \overset{def}{\Leftrightarrow}\\ &\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \;\Rightarrow\\ &\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta\setminus \mathtt{W}} \sigma_0 \wedge\\ &\qquad \sigma'_0(\mathtt{W}) = v \wedge\\ &\qquad \sigma_\omega = \sigma'_\omega\end{aligned}$$

$$\Lambda[\![\mathtt{is\_passed}_{\mathtt{sci}}]\!] = \left\{\begin{array}{l}(l_0, \langle \mathrm{T}, * \rangle) \to \cdots \to (l_9, \langle \mathrm{T}, \mathrm{T} \rangle),\\ (l_0, \langle \mathrm{T}, * \rangle) \to \cdots \to (l_9, \langle \mathrm{T}, \mathrm{F} \rangle),\\ (l_0, \langle \mathrm{F}, * \rangle) \to \cdots \to (l_9, \langle \mathrm{F}, \mathrm{T} \rangle),\\ (l_0, \langle \mathrm{F}, * \rangle) \to \cdots \to (l_9, \langle \mathrm{F}, \mathrm{F} \rangle)\end{array}\right\}$$

$$= \left\{\begin{array}{llllll} l_0: & \langle \mathrm{T},\mathrm{T}\rangle & \langle \mathrm{T},\mathrm{F}\rangle & \langle \mathrm{F},\mathrm{T}\rangle & \langle \mathrm{F},\mathrm{F}\rangle & \\ & \searrow & \searrow\!\!\diagup & \searrow & \searrow\!\!\diagup & \\ l_9: & \langle \mathrm{T},\mathrm{T}\rangle & \langle \mathrm{T},\mathrm{F}\rangle & \langle \mathrm{F},\mathrm{T}\rangle & \langle \mathrm{F},\mathrm{F}\rangle & \end{array}\right.$$

With the use of the next example, we show that non-determinism may affect the abstract non-interference property: non-interfering programs may be wrongly deemed as interfering when non-deterministic statements are present.

Example 3.2.2 Let us consider the simplified trace semantics of the Program 3.1, considering only the variables `sci` and `passing`. As this set of traces is non-deterministic, the predicate ANI would discover that the variable `sci` interferes with the output states. In fact, assuming:

$$\eta_{\mathtt{sci}}(s) \;=\; \lambda \mathtt{j}.\; \begin{cases} s(\mathtt{j}) & \text{if } \mathtt{j} \neq \mathtt{sci} \\ \top & \text{otherwise} \end{cases}$$

$$\rho(s) \;=\; s$$

the predicate of abstract non-interference does not hold for the variable `sci`. This can be shown by the two traces

$$(l_0, \langle \mathrm{T}, \mathrm{T} \rangle) \to \cdots \to (l_9, \langle \mathrm{T}, \mathrm{T} \rangle)$$
$$(l_0, \langle \mathrm{T}, \mathrm{T} \rangle) \to \cdots \to (l_9, \langle \mathrm{T}, \mathrm{F} \rangle)$$

where the output states are different, but the input states are the same. Indeed, ANI does not take into account that such variation in the outcome may be due to the non-determinism and classifies the variable `sci` as an interfering variable.

We formally show that abstract non-interference matches the abstract unused predicate when the program is deterministic, assuming the input abstraction forgets the values of the variables in $\mathtt{W}$, returning $\top$ otherwise.

Proposition 3.2.2 (Abstract Non-Interference Equivalence) *If P is deterministic and the input abstraction is defined as:*

$$\eta_{\mathtt{W}}(s) \;=\; \lambda j.\; \begin{cases} s(j) & \text{if } j \notin \mathtt{W} \\ \top & \text{otherwise} \end{cases}$$

then, it holds that:

$$\text{ANI}(\Lambda[\![P]\!]) \;\Leftrightarrow\; \text{AU}_{\text{NUSED}_{\mathtt{W}}}(\Lambda[\![P]\!])$$

Def. 3.2.1 (Abstract Non-Interference) *The abstract non-interference predicate* ANI *holds if, for any two traces σ and σ':*

$$\eta(\sigma_0) = \eta(\sigma'_0) \;\Rightarrow\; \rho(\sigma_\omega) = \rho(\sigma'_\omega)$$

where η and ρ are upper closure operators on program states to abstract input and output states respectively.

Proof. To show ($\Rightarrow$), we assume ANI. Thus, for any two traces $\sigma, \sigma' \in \Lambda[\![P]\!]$, it holds that whenever $\eta_{\mathtt{W}}(\sigma_0) = \eta_{\mathtt{W}}(\sigma'_0)$, then $\rho(\sigma_\omega) = \rho(\sigma'_\omega)$. By definition of $\eta_{\mathtt{W}}$, we obtain that the output abstractions match whenever $\sigma_0(\mathtt{j}) = \sigma'_0(\mathtt{j})$ for all variables $\mathtt{j} \neq \mathtt{W}$. This hypothesis can be rewritten as $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$. To prove $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$, let $\sigma \in \Lambda[\![P]\!]$ be any trace and $v \in \mathbb{V}^{|\mathtt{W}|}$ be any value. We assume $\sigma_0(\mathtt{W}) \neq v$, otherwise the implication is vacuously true. We need to show that there exists a trace $\sigma' \in \Lambda[\![P]\!]$ such that: (i) $\sigma'_0(\mathtt{W}) = v$, (ii) $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$, and (iii) $\rho(\sigma_\omega) = \rho(\sigma'_\omega)$. The proof proceeds by absurd: we assume that for all traces σ' it holds that $\neg(i) \vee \neg(ii) \vee \neg(iii)$. By definition of the logical implication,[3] we have that $\neg(\neg(i) \vee \neg(ii)) \Rightarrow \neg(iii)$, thus $(i) \wedge (ii) \Rightarrow \neg(iii)$. Hence, for all traces $\sigma' \in \Lambda[\![P]\!]$ we assume $\sigma'_0(\mathtt{W}) = v$ and $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$, respectively (i) and (ii). As a consequence, to conclude ($\Rightarrow$) we need to show an absurd based on the hypothesis that the output abstraction does not match, *i.e.*, $\rho(\sigma_\omega) \neq \rho(\sigma'_\omega)$. Clearly, from ANI applied to (ii) we have that the outputs match, contradicting the hypothesis. Thus, $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$ holds.[4]

3: $\neg A \vee B \equiv A \Rightarrow B$.

4: Note that, to show ($\Rightarrow$) we do not need the assumption that P is deterministic.

To show ($\Leftarrow$), we assume $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$. For any trace $\sigma \in \Lambda[\![P]\!]$ and value $v \in \mathbb{V}^{|\mathtt{W}|}$, it holds that whenever (a) $\sigma_0(\mathtt{W}) \neq v$, then it exists a trace $\sigma' \in \Lambda[\![P]\!]$ such that (b) $\sigma'_0(\mathtt{W}) = v$, (c) $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$, and (d) $\rho(\sigma_\omega) = \rho(\sigma'_\omega)$ hold. To prove ANI, we assume that for any two traces $\sigma, \sigma' \in \Lambda[\![P]\!]$ it holds that $\sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma'_0$, otherwise ANI vacuously holds. We need to show that the output abstraction of the two traces match, *i.e.*, $\rho(\sigma_\omega) = \rho(\sigma'_\omega)$. Let $v \in \mathbb{V}^{|\mathtt{W}|}$ be any value.

1. If $\sigma_0(\mathtt{W}) = \sigma'_0(\mathtt{W}) = v$, then by determinism of P it follows that (d), cf. $\sigma_\omega = \sigma'_\omega$, must hold.
2. If $\sigma_0(\mathtt{W}) \neq \sigma'_0(\mathtt{W})$, we let $\sigma_0(\mathtt{W}) \neq v$ and $\sigma'_0(\mathtt{W}) = v$ without loss of generality, respectively proving (a) and (c). (b) holds by hypothesis. Hence, from $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$ it follows that (d) must hold:

$$\begin{aligned}
&\forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}. \\
&\quad (a) \Rightarrow \exists \sigma' \in \Lambda[\![P]\!].\ (b) \wedge (c) \wedge (d) && \Leftrightarrow \\
&\quad \neg(a) \vee (\exists \sigma' \in \Lambda[\![P]\!].\ (b) \wedge (c) \wedge (d)) && \Leftrightarrow \\
&\quad (\exists \sigma' \in \Lambda[\![P]\!].\ (b) \wedge (c) \wedge (d)) \vee \neg(a) && \Leftrightarrow \\
&\quad \neg(\exists \sigma' \in \Lambda[\![P]\!].\ (b) \wedge (c) \wedge (d)) \Rightarrow \neg(a) && \Leftrightarrow \\
&\quad (\forall \sigma' \in \Lambda[\![P]\!].\ \neg(b) \vee \neg(c) \vee \neg(d)) \Rightarrow \neg(a) && \Leftrightarrow \\
&\quad (\forall \sigma' \in \Lambda[\![P]\!].\ \neg(\neg(b) \vee \neg(c)) \Rightarrow \neg(d)) \Rightarrow \neg(a) && \Leftrightarrow \\
&\quad (\forall \sigma' \in \Lambda[\![P]\!].\ (b) \wedge (c) \Rightarrow \neg(d)) \Rightarrow \neg(a)
\end{aligned}$$

Thus, everything resolves around whether (d) holds or not. If (d) does not hold, then $\neg(a)$ holds, but this contradicts the hypothesis

that (a) holds, hence (d) must hold and so ANI. Otherwise, if (d) holds, then ANI holds. Concluding $(\Leftarrow)$.

□

Example 3.2.3 Interestingly, with the use of "seeds" one could also determinize a non-deterministic semantics by fixing the random observations as part of the program states. Consider the set of traces of the previous example, cf. $\Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!]$, but with the addition of a random variable that is used to determinize the outcome of the program. A possible determinization could be the set of traces:

$$\left\{\begin{array}{l}
(l_0, \langle \textsc{t}, \textsc{t}, \textsc{t}\rangle) \to \cdots \to (l_9, \langle \textsc{t}, \textsc{t}\rangle),\\
(l_0, \langle \textsc{t}, \textsc{t}, \textsc{f}\rangle) \to \cdots \to (l_9, \langle \textsc{t}, \textsc{f}\rangle),\\
(l_0, \langle \textsc{t}, \textsc{f}, \textsc{t}\rangle) \to \cdots \to (l_9, \langle \textsc{t}, \textsc{t}\rangle),\\
(l_0, \langle \textsc{t}, \textsc{f}, \textsc{f}\rangle) \to \cdots \to (l_9, \langle \textsc{t}, \textsc{f}\rangle),\\
(l_0, \langle \textsc{f}, \textsc{t}, \textsc{t}\rangle) \to \cdots \to (l_9, \langle \textsc{f}, \textsc{t}\rangle),\\
(l_0, \langle \textsc{f}, \textsc{t}, \textsc{f}\rangle) \to \cdots \to (l_9, \langle \textsc{f}, \textsc{f}\rangle)\\
(l_0, \langle \textsc{f}, \textsc{f}, \textsc{f}\rangle) \to \cdots \to (l_9, \langle \textsc{f}, \textsc{t}\rangle),\\
(l_0, \langle \textsc{f}, \textsc{f}, \textsc{f}\rangle) \to \cdots \to (l_9, \langle \textsc{f}, \textsc{f}\rangle)
\end{array}\right\}$$

where the last component of input states is the boolean seed used to determine the outcome of the program. Figure 3.8 shows the traces in a graphical representation. In such instance, the semantics is deterministic as it is not possible anymore to have two traces with the same input states and different output states. Thus, the predicate ANI and $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$ are equivalent in such instance.

$\Lambda[\![\texttt{is\_passed}_{\texttt{sci}}]\!] =$

$$\left\{\begin{array}{l}
(l_0, \langle \textsc{t}, *\rangle) \to \cdots \to (l_9, \langle \textsc{t}, \textsc{t}\rangle),\\
(l_0, \langle \textsc{t}, *\rangle) \to \cdots \to (l_9, \langle \textsc{t}, \textsc{f}\rangle),\\
(l_0, \langle \textsc{f}, *\rangle) \to \cdots \to (l_9, \langle \textsc{f}, \textsc{t}\rangle),\\
(l_0, \langle \textsc{f}, *\rangle) \to \cdots \to (l_9, \langle \textsc{f}, \textsc{f}\rangle)
\end{array}\right\}$$

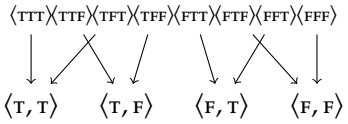


Figure 3.8: Trace semantics of Program 3.1 with determinization (third component of initial states).

Combining both non-interference and input data usage into the abstract notion of $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$ permits instances of abstract non-interference to work in the context of non-deterministic programs. For example, the *non-exploitability* property [75] is an instance of abstract non-interference where the output states are abstracted to a flag that indicates whether the program reached an error state or not. Specifically, the state maps the error flag (called ERROR in Definition 3.2.4) to a value indicating whether an error state has been reached during the execution or not. The non-exploitability property holds if, for any variation of the variables $\mathtt{W} \in \wp(\Delta)$, the program does not reach an error state (or it does so consistently across all the traces). The idea is to prove that an attacker is not able to exploit an error state based on the values of public variables under their control. When such property is plugged into $\text{AU}_{\text{NUSED}_{\mathtt{W}}}$, it convenes a non-exploitability definition for non-deterministic programs.

[75]: Parolini et al. (2024), 'Sound Abstract Nonexploitability Analysis'

Def. 3.2.4 (Non-Exploitability) *Let* $\mathtt{W} \in \wp(\Delta)$ *be the set of public input variables. The* non-exploitability *predicate* ANE *holds if, for any two traces* σ *and* σ'*:*

$$\sigma_0 =_{\Delta\setminus\mathtt{W}} \sigma'_0 \;\Rightarrow\; \sigma_\omega(\textsc{error}) = \sigma'_\omega(\textsc{error})$$

Definition 3.2.5 (Non-Exploitability for Non-Deterministic Programs) *Let* $\mathtt{W} \in \wp(\Delta)$ *be the set of public input variables. The* non-exploitability predicate for non-deterministic programs *is defined as follows:*

$$\begin{aligned}
&\text{ANE}(\Lambda[\![P]\!]) \overset{def}{\Leftrightarrow}\\
&\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \;\Rightarrow\\
&\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta\setminus\{\mathtt{i}\}} \sigma_0 \;\wedge\; \sigma'_0(\mathtt{W}) = v \;\wedge\; \sigma_\omega(\textsc{error}) = \sigma'_\omega(\textsc{error})
\end{aligned}$$

This property works as a generalization of non-exploitability for non-deterministic programs without the need of ad-hoc semantics for non-determinism, cf. Example 3.2.3.

Next, we show that whenever some variables are not used, they are also abstractly unused. In other words, the unused predicate is stronger than its abstract counterpart.

Lemma 3.2.3 (Unused Implies Abstract Unused)

$$\textsc{Unused}_{\mathtt{W}}(\Lambda[\![P]\!]) \implies \textsc{AUnused}_{\mathtt{W}}(\Lambda[\![P]\!])$$

Note that, $\textsc{Unused}_{\mathtt{W}}$ is the unused predicate, cf. Definition 3.1.1 (Unused), and $\textsc{AUnused}_{\mathtt{W}}$ is the abstract unused predicate, cf. Definition 3.2.3 (Abstract Unused).

Proof. The proof easily follows from the fact that ρ is an upper closure operator, Definition 2.1.7. Hence, it relaxes the existential condition of the trace σ' on the output states, *i.e.*, more traces are now able to satisfy $\rho(\sigma_\omega) = \rho(\sigma'_\omega)$. □

Def. 2.1.7 (Upper Closure Operator) *An* upper closure operator *on a partially ordered set $\langle X, \subseteq \rangle$ is an operator $\rho \in X \to X$ which is monotone, idempotent, and extensive (i.e., $x \subseteq \rho(x)$ for all $x \in X$).*

3.3 The Unused Property

In this section, we state the *unused property* as a property of programs. Whenever a program does not use the input variables $\mathtt{W}$, the trace semantics of the program belongs to the unused property. The unused property is defined as follows:

Given a program P, its semantics is defined by the trace semantics $\Lambda[\![P]\!] \in \wp(\Sigma^{+\infty})$, that is, the set of finite and infinite traces of the program P, cf. Section 2.3.2.

Definition 3.3.1 (Unused Property) *Given a program P, and the set of input variables of interest $\mathtt{W} \in \wp(\Delta)$, the* unused property $\mathcal{N}_{\mathtt{W}} \in \wp(\wp(\Sigma^{+\infty}))$ *is:*

$$\mathcal{N}_{\mathtt{W}} \stackrel{\text{def}}{=} \{\Lambda[\![P]\!] \in \wp(\Sigma^{+\infty}) \mid \textsc{AUnused}_{\mathtt{W}}(\Lambda[\![P]\!])\}$$

Def. 3.2.3 (Abstract Unused)

$$\begin{aligned}&\textsc{AUnused}_{\mathtt{W}}(\Lambda[\![P]\!]) \stackrel{\text{def}}{\Leftrightarrow}\\ &\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \implies\\ &\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0 \wedge\\ &\qquad \sigma'_0(\mathtt{W}) = v \wedge\\ &\qquad \rho(\sigma_\omega) = \rho(\sigma'_\omega)\end{aligned}$$

Specifically, the unused property $\mathcal{N}_{\mathtt{W}}$ is the set of all the programs (or rather, their semantics) that do not use any of the variables in $\mathtt{W}$.

Remark 3.3.1 A program P satisfies the unused property $\mathcal{N}_{\mathtt{W}}$ if and only if P does not use the set of input variables $\mathtt{W}$, formally:

$$P \models \mathcal{N}_{\mathtt{W}} \iff \Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{N}_{\mathtt{W}}$$

Note that, the unused property is extensional, *i.e.*, it does not depend on the intermediate states of the traces. Indeed, given a set of traces generated by the trace semantics of a program P, the abstract unused predicate $\textsc{AUnused}_{\mathtt{W}}$ only considers the input and output states of the given set of traces.

3.4 Dependency Semantics

Since the unused property $\mathcal{N}_{\mathtt{W}}$ is an extensional property, we can abstract away the intermediate states of traces without losing the information

about the input-output dependencies. Specifically, we abstract the collecting semantics $\Lambda^{\mathbb{C}} \in \wp(\wp(\Sigma^{+\infty}))$, cf. Definition 2.2.3, into a set of dependencies between initial and output states of finite traces and between initial and $\perp$ for infinite traces. Formally, the pair of right-left adjoints $\langle \alpha^{\rightsquigarrow}, \gamma^{\rightsquigarrow} \rangle$ is defined as follows:

The notion $\sigma_0 \in \Sigma$ refers to the initial state of a trace σ and $\sigma_\omega \in \Sigma^\perp$, where $\Sigma^\perp \stackrel{\text{def}}{=} \Sigma \cup \{\perp\}$ to the final one if σ is of finite length, otherwise $\perp$ for non-terminating traces.

Definition 3.4.1 (Right-Left Adjoints for the Dependency Semantics)

$$\begin{aligned} \alpha^{\rightsquigarrow} &\in \wp(\wp(\Sigma^{+\infty})) \to \wp(\wp(\Sigma \times \Sigma^{\perp})) \\ \alpha^{\rightsquigarrow}(S) &\stackrel{\text{def}}{=} \{\{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in T\} \mid T \in S\} \\ \gamma^{\rightsquigarrow} &\in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma^{+\infty})) \\ \gamma^{\rightsquigarrow}(W) &\stackrel{\text{def}}{=} \{T \in \wp(\Sigma^{+\infty}) \mid \{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in T\} \in W\} \end{aligned}$$

The function $\alpha^{\rightsquigarrow}$ abstracts away all intermediate states of any trace, preserving the set-structure of S. The concretization $\gamma^{\rightsquigarrow}$ yields all the semantics that share the same output observations of, at least, one of the set of semantics in W.

Theorem 3.4.1 *The two adjoints* $\langle \alpha^{\rightsquigarrow}, \gamma^{\rightsquigarrow} \rangle$ *form a* Galois insertion*:*

$$\langle \wp(\wp(\Sigma^{+\infty})), \subseteq \rangle \underset{\alpha^{\rightsquigarrow}}{\overset{\gamma^{\rightsquigarrow}}{\leftrightarrows}} \langle \wp(\wp(\Sigma \times \Sigma^{\perp})), \subseteq \rangle$$

Proof. We need to show that $\alpha^{\rightsquigarrow}(S) \subseteq W \Leftrightarrow S \subseteq \gamma^{\rightsquigarrow}(W)$. First, we show the direction ($\Rightarrow$). Assuming $\alpha^{\rightsquigarrow}(S) \subseteq W$, we have that $\gamma^{\rightsquigarrow}(W)$ contains all the possible semantics that share the same set of input-output observations of at least one of the semantics in W. Thus, $\gamma^{\rightsquigarrow}(W)$ also contains all the semantics in S, *i.e.*, $S \subseteq \gamma^{\rightsquigarrow}(W)$. To show ($\Leftarrow$), we assume $S \subseteq \gamma^{\rightsquigarrow}(W)$. It is easy to note that $\alpha^{\rightsquigarrow}(\gamma^{\rightsquigarrow}(W)) = W$ since the concretization maintains the same input-output observations and the abstraction removes only intermediate states. Hence, by monotonicity of $\alpha^{\rightsquigarrow}$, we obtain $\alpha^{\rightsquigarrow}(S) \subseteq \alpha^{\rightsquigarrow}(\gamma^{\rightsquigarrow}(W)) = W$. □

We now derive the *dependency semantics* $\Lambda^{\rightsquigarrow}$ as an abstraction of the collecting semantics.

Definition 3.4.2 (Dependency Semantics) *The* dependency semantics $\Lambda^{\rightsquigarrow} \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$ *is defined as:*

$$\Lambda^{\rightsquigarrow} \stackrel{\text{def}}{=} \alpha^{\rightsquigarrow}(\Lambda^{\mathbb{C}}) = \{\, \{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in \Lambda\} \,\}$$

The next result shows that the dependency semantics $\Lambda^{\rightsquigarrow}$ allows a sound and complete verification that a set of input variables `W` is unused by the program P, Definition 3.3.1.

Def. 3.3.1 (Unused Property)

$\mathcal{N}_{\mathtt{W}} \stackrel{\text{def}}{=} \{\Lambda \in \wp(\Sigma^{+\infty}) \mid \text{AUnused}_{\mathtt{W}}(\Lambda)\}$

Theorem 3.4.2 $\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{N}_{\mathtt{W}} \Leftrightarrow \Lambda^{\rightsquigarrow}[\![P]\!] \subseteq \alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}})$

Proof. The implication ($\Rightarrow$) follows from the monotonicity of $\alpha^{\rightsquigarrow}$, as an implication from the fact that the two adjoints $\langle \alpha^{\rightsquigarrow}, \gamma^{\rightsquigarrow} \rangle$ form a Galois

connection (cf. Theorem 3.4.1), and Definition 3.4.2 of $\Lambda^{\rightsquigarrow}$. Obtaining $\Lambda^{\mathbb{C}} \subseteq \mathcal{N}_{\mathtt{W}} \Rightarrow \alpha^{\rightsquigarrow}(\Lambda^{\mathbb{C}}) \subseteq \alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}) \Rightarrow \Lambda^{\rightsquigarrow} \subseteq \alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}})$. Regarding the other implication ($\Leftarrow$), from Definition 3.4.2 of $\Lambda^{\rightsquigarrow}$ and the property of Theorem 3.4.1, we obtain $\Lambda^{\rightsquigarrow} \subseteq \alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}) \Rightarrow \alpha^{\rightsquigarrow}(\Lambda^{\mathbb{C}}) \subseteq \alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}) \Rightarrow \Lambda^{\mathbb{C}} \subseteq \gamma^{\rightsquigarrow}(\alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}))$, which can be written as $\Lambda \in \gamma^{\rightsquigarrow}(\alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}))$ by the definition of $\Lambda^{\mathbb{C}}$. By Definition 3.4.1 of $\gamma^{\rightsquigarrow}$ it follows that $\{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in \Lambda\} \in \alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}})$. Finally, by application of Definition 3.4.1 of $\alpha^{\rightsquigarrow}$ we obtain $\Lambda \in \mathcal{N}_{\mathtt{W}}$. The conclusion $\Lambda^{\mathbb{C}} \subseteq \mathcal{N}_{\mathtt{W}}$ trivially follows from the definition of the subset inclusion ($\subseteq$). □

Example 3.4.1 We define the dependency semantics $\Lambda^{\rightsquigarrow}[\![\mathtt{is\_passed}_{\mathtt{sci}}]\!]$ as an abstraction of the collecting semantics $\{\Lambda[\![\mathtt{is\_passed}_{\mathtt{sci}}]\!]\}$:

$$\begin{aligned}\Lambda^{\rightsquigarrow}[\![\mathtt{is\_passed}_{\mathtt{sci}}]\!] &= \alpha^{\rightsquigarrow}(\{\Lambda[\![\mathtt{is\_passed}_{\mathtt{sci}}]\!]\}) \\ &= \left\{ \left\{ \begin{array}{l} \langle (l_0, \langle \mathtt{T}, * \rangle), (l_9, \langle \mathtt{T}, \mathtt{T} \rangle) \rangle, \\ \langle (l_0, \langle \mathtt{T}, * \rangle), (l_9, \langle \mathtt{T}, \mathtt{F} \rangle) \rangle, \\ \langle (l_0, \langle \mathtt{F}, * \rangle), (l_9, \langle \mathtt{F}, \mathtt{T} \rangle) \rangle, \\ \langle (l_0, \langle \mathtt{F}, * \rangle), (l_9, \langle \mathtt{F}, \mathtt{F} \rangle) \rangle \end{array} \right\} \right\}\end{aligned}$$

The dependency semantics abstracts the collecting semantics, preserving the input-output dependencies.

3.5 Output-Abstraction Semantics

We further abstract the dependency semantics $\Lambda^{\rightsquigarrow}$ into the output-abstraction semantics Λ^{ρ} as the unused property $\mathcal{N}_{\mathtt{W}}$ never considers output states without first abstracting them via the output observer ρ, Definition 3.2.2. Formally, the pair of right-left adjoints $\langle \alpha^{\rho}, \gamma^{\rho} \rangle$ is defined as:

Def. 3.2.2 (Output Observer) *Given a program P, an* output observer $\rho \in \Sigma^{\perp} \to \Sigma^{\perp}$ *is an upper closure operator that abstracts the value of output states.*

Definition 3.5.1 (Right-Left Adjoints for the Output-Abstraction Semantics)

$$\begin{aligned} \alpha^{\rho} &\in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma \times \Sigma^{\perp})) \\ \alpha^{\rho}(W) &\stackrel{\text{def}}{=} \{\{\langle s_0, \rho(s_\omega) \rangle \mid \langle s_0, s_\omega \rangle \in D\} \mid D \in W\} \\ \gamma^{\rho} &\in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma \times \Sigma^{\perp})) \\ \gamma^{\rho}(W) &\stackrel{\text{def}}{=} \{\cup\{D' \subseteq \{\langle s_0, s'_\omega \rangle \mid s_\omega = \rho(s'_\omega)\} \mid \langle s_0, s_\omega \rangle \in D\} \mid D \in W\} \end{aligned}$$

The function α^{ρ} abstracts the output states of the dependencies and γ^{ρ} concretizes the set of dependencies that share the same set of output observations. Note that, the abstraction function α^{ρ} is well-defined as it preserves arbitrary set joins, *i.e.*, $\forall D, D' \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$. $\alpha^{\rho}(D \cup D') = \alpha^{\rho}(D) \cup \alpha^{\rho}(D')$ as α^{ρ} preserves the set structure of its parameters.

Theorem 3.5.1 *The two adjoints $\langle \alpha^{\rho}, \gamma^{\rho} \rangle$ form a* Galois insertion*:*

$$\langle \wp(\wp(\Sigma \times \Sigma^{\perp})), \subseteq \rangle \underset{\alpha^{\rho}}{\overset{\gamma^{\rho}}{\leftrightarrows}} \langle \wp(\wp(\Sigma \times \Sigma^{\perp})), \subseteq \rangle$$

Proof. We need to show that $\alpha^{\rho}(S) \subseteq W \Leftrightarrow S \subseteq \gamma^{\rho}(W)$. First, we show the direction ($\Rightarrow$). Assuming $\alpha^{\rho}(S) \subseteq W$, we have that $\gamma^{\rho}(W)$ builds all the possible sets of sets of dependencies enhanced with the any amount of output states that result in an output abstraction of W'. Thus, $\gamma^{\rho}(W)$ also contains all the semantics in S, *i.e.*, $S \subseteq \gamma^{\rho}(W)$. To show ($\Leftarrow$), we assume $S \subseteq \gamma^{\rho}(W)$. We note that the concretization $\gamma^{\rho}(W)$ builds all the possible sets of dependencies that may have generated the same output observations of each set of dependencies in W. By applying the abstraction α^{ρ}, for each of the sets of dependencies concretized by γ^{ρ}, the abstraction returns the original set in W. Formally, it holds that $\alpha^{\rho}(\gamma^{\rho}(W)) = W$. Hence, by monotonicity of α^{ρ}, we obtain $\alpha^{\rho}(S) \subseteq \alpha^{\rho}(\gamma^{\rho}(W)) = W$. □

We now derive the *output-abstraction semantics* Λ^{ρ} as an abstraction of the dependency semantics.

Definition 3.5.2 (Output-Abstraction Semantics) *The* output-abstraction semantics $\Lambda^{\rho} \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$ *is defined as:*

$$\Lambda^{\rho} \stackrel{\text{def}}{=} \alpha^{\rho}(\Lambda^{\rightsquigarrow}) = \{\, \{\langle \sigma_0, \rho(\sigma_\omega)\rangle \mid \sigma \in \Lambda\} \,\}$$

The next result shows that the output-abstraction semantics Λ^{ρ} allows a sound and complete verification for proving that an input variable $\mathtt{W}$ is unused in the program P, Definition 3.3.1.

Def. 3.3.1 (Unused Property)
$\mathcal{N}_{\mathtt{W}} \stackrel{\text{def}}{=} \{\Lambda \in \wp(\Sigma^{+\infty}) \mid \text{AU}\textsc{nused}_{\mathtt{W}}(\Lambda)\}$

Theorem 3.5.2 $\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{N}_{\mathtt{W}} \Leftrightarrow \Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}))$

Proof. Similarly to the proof of Theorem 3.4.2, the implication ($\Rightarrow$) follows from the monotonicity of α^{ρ} and Definition 3.5.2 of Λ^{ρ}. The implication ($\Leftarrow$) follows from the definition of the output-abstraction semantics Λ^{ρ} and the property of Theorem 3.5.1. □

We show the unused property $\mathcal{N}_{\mathtt{W}}$ derived from the dependency and output-abstraction semantics, where the output abstraction and intermediate states are handled at a semantics level rather than in the property definition.

The set Σ contains the program states and $\Sigma^{\perp} \stackrel{\text{def}}{=} \Sigma \cup \{\perp\}$ are the states extended with $\perp$ to represent non-termination. The symbol $\mathbb{V}^{|\mathtt{W}|}$ is the vector of values for $|\mathtt{W}|$ variables.
From two states $s, s' \in \Sigma$, the notation $s =_{\Delta \setminus \mathtt{W}} s'$ denotes that the values of the variables in $\Delta \setminus \mathtt{W}$ are equal in both states, *i.e.*, $\forall \mathtt{x} \in \Delta \setminus \mathtt{W}.\ s(\mathtt{x}) = s'(\mathtt{x})$.

Remark 3.5.1 The abstraction $\alpha^{\rho} \circ \alpha^{\rightsquigarrow}$ of the unused property $\mathcal{N}_{\mathtt{W}}$ is defined as:

$$\left\{ D \in \wp(\Sigma \times \Sigma^{\perp}) \;\middle|\; \begin{array}{l} \forall \langle s_0, s_\omega\rangle \in D, v \in \mathbb{V}^{|\mathtt{W}|}.\ s_0(\mathtt{W}) \neq v \Rightarrow \\ \exists \langle s'_0, s'_\omega\rangle \in D.\ s'_0 =_{\Delta\setminus\{\mathtt{i}\}} s_0 \wedge s'_0(\mathtt{W}) = v \\ \wedge\ s_\omega = s'_\omega \end{array} \right\}$$

Note that, the property defined in the remark above is not, per se, a property of programs as by extension it should be a set of program semantics. Instead, it is a property of output-and-dependency semantics, which is sets of input-output observations. Nevertheless, such characterization is useful as, whenever $\Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{N}_{\mathtt{W}}))$ holds, the program P does not use the input variables in $\mathtt{W}$, cf. Theorem 3.5.2. In the rest of the thesis, we abuse the notation and refer to Unused for its dependency abstraction as well.

Additionally, we define the reduction of a set of sets of dependencies W to those with abstraction of a final state in X.

Definition 3.5.3 (Output Reduction) *Let $X \in \wp(\Sigma^\perp)$ be a set of output states. The* output reduction $W|_X \in \wp(\wp(\Sigma \times \Sigma^\perp))$ *of* $W \in \wp(\wp(\Sigma \times \Sigma^\perp))$ *is defined as:*

$$W|_X \stackrel{\text{def}}{=} \{\{\langle s_0, \rho(s_\omega)\rangle \in D \mid s_\omega \in X\} \mid D \in W\}$$

Such reduction is useful to analyze the dependencies that lead to a specific set of output states. Note that, in case no dependency in W ends in an output state in X, the reduction is $\{\emptyset\}$.

Similarly, we define the reduction of a set of sets of dependencies W to those that start from an *initial* state in X.

Definition 3.5.4 (Input Reduction) *Let $X \in \wp(\Sigma)$ be a set of input states. The* input reduction $W\|_X \in \wp(\wp(\Sigma \times \Sigma^\perp))$ *of* $W \in \wp(\wp(\Sigma \times \Sigma^\perp))$ *is defined as:*

$$W\|_X \stackrel{\text{def}}{=} \{\{\langle s_0, s_\omega\rangle \in D \mid s_0 \in X\} \mid D \in W\}$$

3.6 Syntactic Dependency Analysis

From Urban and Müller [42, Section 10], we report a data usage analysis based on *syntactic* dependencies between program variables. The analysis captures both implicit and explicit flows of information between variables. To do so, it tracks when a variable is used or modified in a statement also based on the nesting level. The idea is that a variable also depend on variables that are not explicitly assigned to it. For instance, if a variable is modified within a loop body, it depends on the variables of the loop condition. More formally, each variable in the program maps a value in the Usage complete lattice, Figure 3.9. The lattice contains the following elements: U (*used*) and N (*unused*) denoting respectively that a variable may be used or is definitely not used at the current nesting level. The elements B (*below*) and W (*overwritten*) denote that a variable may be used at a lower nesting level or is modified at the current nesting level, respectively.

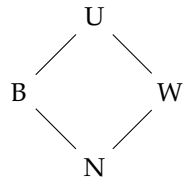


Figure 3.9: The Usage lattice.

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

Based on the syntax of the simple programming language of Figure 2.11, reported on the side, a variable is (explicitly) used if it is assigned to another variable that is used in the current (U) or lower nesting level (B). We define the operator $\textsc{Assign}[\![\mathtt{x} := e]\!]$ to handle the assignment of variables on a map $m \in \mathbb{V}\mathtt{ar} \to \textsc{Usage}$ as follows:

$$\textsc{Assign}[\![\mathtt{x} := e]\!](m) \stackrel{\text{def}}{=} \lambda \mathtt{x}'. \begin{cases} \mathrm{W} & \mathtt{x}' = \mathtt{x} \wedge \mathtt{x}' \notin \textsc{Vars}(e) \wedge m(\mathtt{x}) \in \{\mathrm{U}, \mathrm{B}\} \\ \mathrm{U} & \mathtt{x}' \in \textsc{Vars}(e) \wedge m(\mathtt{x}) \in \{\mathrm{U}, \mathrm{B}\} \\ m(\mathtt{x}') & \text{otherwise} \end{cases}$$

$$\begin{aligned} e &::= v \\ &\mid \mathtt{x} \\ &\mid e + e \\ &\mid e - e \\ &\mid \mathtt{rand}(l, u) \qquad (l \in \mathbb{Z}^{\pm\infty} \wedge l < u) \\ b &::= e \leq v \\ &\mid e = v \\ &\mid b \wedge b \\ &\mid \neg b \\ st &::= {}^{l}\mathtt{skip} \\ &\mid {}^{l}\mathtt{x} := e \\ &\mid \mathtt{if}\ {}^{l}b\ \mathtt{then}\ st\ \mathtt{else}\ st \\ &\mid \mathtt{while}\ {}^{l}b\ \mathtt{do}\ st\ \mathtt{done} \\ &\mid st; st \\ P &::= \mathtt{entry}\ st^{l} \qquad (l \in \mathbb{L}) \end{aligned}$$

The assigned variable is overwritten (W) unless it is used in the expression e. Another reason for a variable to be (implicitly) used is if it appears in the boolean condition of a statement that uses another variable (U) or modifies another used variable (W). We define the operator $\textsc{Filter}[\![b]\!]$ to handle boolean conditions as follows:

$$\textsc{Filter}[\![b]\!](m) \stackrel{\text{def}}{=} \lambda \mathsf{x}'. \begin{cases} \mathrm{U} & \mathsf{x}' \in \textsc{Vars}(b) \wedge \exists \mathsf{x} \in \mathbb{V}\mathtt{ar}.\ m(\mathsf{x}) \in \{\mathrm{U}, \mathrm{W}\} \\ m(\mathsf{x}') & \text{otherwise} \end{cases}$$

During the dependency analysis, to take into account implicit flows of information between variables we propagate a *stack* that grows and shrinks based on the nesting level of the statements. The stack is a tuple $\langle m_0, m_1, \ldots m_n \rangle$ of mutable length n, where each element $m_j \in \mathbb{V}\mathtt{ar} \to \textsc{Usage}$ maps variables to their usage status, for $j \leq n$. We denote the set of all stacks by $\textsc{Stack}$, we define the assignment and filter operators on stacks as well:

$$\begin{aligned} \textsc{Assign}[\![\mathsf{x} := e]\!](\langle m_0, m_1, \ldots m_n \rangle) &\stackrel{\text{def}}{=} \langle \textsc{Assign}[\![\mathsf{x} := e]\!](m_0), m_1, \ldots m_n \rangle \\ \textsc{Filter}[\![b]\!](\langle m_0, m_1, \ldots m_n \rangle) &\stackrel{\text{def}}{=} \langle \textsc{Filter}[\![\mathsf{x} := e]\!](m_0), m_1, \ldots m_n \rangle \end{aligned}$$

We define additional operators to handle statements with nested body (*e.g.*, loops). The $\textsc{Push}$ operator duplicates the map on top of the stack and modifies the duplicate accordingly with the increased nesting level: used variables (U) were used at the previous (lower) nesting level, overwritten variables (W) are now unused, the rest is propagated without further modifications. Formally:

$$\begin{aligned} \textsc{Push}(\langle m_0, m_1, \ldots m_n \rangle) &\stackrel{\text{def}}{=} \langle \textsc{Incr}(m_0), m_0, m_1, \ldots m_n \rangle \\ \textsc{Incr}(m) &\stackrel{\text{def}}{=} \lambda \mathsf{x}. \begin{cases} \mathrm{B} & m(\mathsf{x}) = \mathrm{U} \\ \mathrm{N} & m(\mathsf{x}) = \mathrm{W} \\ m(\mathsf{x}) & \text{otherwise} \end{cases} \end{aligned}$$

The dual operator is the $\textsc{Pop}$ operator that restores the information about the previous nested level and combines it with the new information computed inside the higher nested level:

$$\begin{aligned} \textsc{Pop}(\langle m_0, m_1, \ldots m_n \rangle) &\stackrel{\text{def}}{=} \langle \textsc{Decr}(m_0, m_1), \ldots m_n \rangle \\ \textsc{Decr}(m_0, m_1) &\stackrel{\text{def}}{=} \lambda \mathsf{x}. \begin{cases} m_1(\mathsf{x}) & m_0(\mathsf{x}) \in \{\mathrm{B}, \mathrm{N}\} \\ m_0(\mathsf{x}) & \text{otherwise} \end{cases} \end{aligned}$$

The syntactic dependency analysis Λ^{u} is a backward analysis on the lattice of stacks $\langle \textsc{Stack}, \sqsubseteq_{\mathrm{S}}, \sqcup_{\mathrm{S}} \rangle$. The partial order $\sqsubseteq_{\mathrm{S}}$ and least upper bound $\sqcup_{\mathrm{S}}$ are point-wise lifting of those of the $\textsc{Usage}$ lattice. The syntactic

$\Lambda^{\mathrm{u}}[\![\texttt{skip}]\!]q \stackrel{\text{def}}{=} q$

$\Lambda^{\mathrm{u}}[\![\mathsf{x} := e]\!]q \stackrel{\text{def}}{=} \textsc{Assign}[\![\mathsf{x} := e]\!]q$

$\Lambda^{\mathrm{u}}[\![\texttt{assert } b]\!]q \stackrel{\text{def}}{=} \textsc{Filter}[\![b]\!]q$

$\Lambda^{\mathrm{u}}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]q \stackrel{\text{def}}{=}$
$\quad \textsc{Pop} \circ \textsc{Filter}[\![b]\!] \circ \Lambda^{\mathrm{u}}[\![st]\!] \circ \textsc{Push}(q)$
$\quad \sqcup_{\mathrm{S}}\ \textsc{Pop} \circ \textsc{Filter}[\![b]\!] \circ \Lambda^{\mathrm{u}}[\![st']\!] \circ \textsc{Push}(q)$

$\Lambda^{\mathrm{u}}[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]q \stackrel{\text{def}}{=}$
$\quad \mathrm{lfp}\, \Lambda^{\mathrm{u}}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st]\!]q$

$\Lambda^{\mathrm{u}}[\![st; st']\!]q \stackrel{\text{def}}{=} \Lambda^{\mathrm{u}}[\![st]\!](\Lambda^{\mathrm{u}}[\![st']\!]q)$

Figure 3.10: The syntactic dependency analysis.

dependency analysis, for each statement in the small imperative language of Section 2.3, Λ^{u} is defined in Figure 3.10. The initial stack contains a single map m_0 that maps all the output variables to U (entry invariant of the analysis) and all the others to N. We exemplify the analysis below.

Prog. 3.1 (Program to check if a student passed the school year.):

```
1 is_passed(eng, math, sci, plus):
2   passing = True;
3   if not eng:
4     eng = False;
5   if not math:
6     passing = plus;
7   if not math:
8     passing = plus;
9
```

Example 3.6.1 Consider again Program 3.1. The syntactic dependency analysis $\Lambda^{\mathsf{u}}[\![\texttt{is\_passed}]\!]$ starts from the stack $\langle m_0 \rangle$ where m_0 maps the output variable `passing` to U and the rest to N. The analysis first approaches the last conditional statement (the analysis proceed backwards) at Line 7. A modified copy m_1 of m_0 (the map on top of the stack) is pushed on the stack: as `passing` is used in m_0, in m_1 it is marked as B, meaning that `passing` is used in a lower nesting level; all the other variables are propagated as unused. Then, the assignment on Line 8, `passing` is overwritten (cf. W) and `plus` is used (cf. U). The analysis continues with the head of the conditional statement. Since the body of the conditional statement modified a used variable, the variables belonging to the condition are marked as used as well, *i.e.*, `math` maps to U. Finally, exiting the conditional statement of Line 7, the map on top of the stack is popped and the variables that were used in the previous nesting level are marked as used in the current nesting level. Therefore, after analyzing the last conditional statement, the stack contains a single map where `math`, `plus`, and `passing` are marked as used.

The stack remains the same for the next conditional statement, Line 5, as identical operations are performed. The first conditional statement, cf. Line 3, does not modify any used variable, thus the stack does not change. Finally, at Line 2, the variable `passing` is modified and maps to W, while `math` and `plus` are still used. The analysis is precise enough to capture that the variable `eng` and `sci` are *definitely* unused in the computation of the variable `passing`.

The concretization maps syntactic dependencies to set of (state) dependencies in $\wp(\wp(\Sigma \times \Sigma^{\perp}))$ that satisfy the unused predicate $\textsc{Unused}_{\mathtt{W}}$, cf. Definition 3.1.1.

The set $\mathbb{V}\mathtt{ar}$ contains the program variables, while $\Delta \subseteq \mathbb{V}\mathtt{ar}$ is the subset of input variables. Thus, $\{\mathtt{x} \in \mathbb{V}\mathtt{ar} \mid m_0(\mathtt{x}) = \mathrm{N}\}$ contains the variables that are unused in the map m_0.

Definition 3.6.1 *The concretization* $\gamma^{\mathsf{u}} \in \textsc{Stack} \to \wp(\wp(\Sigma \times \Sigma^{\perp}))$ *of the syntactic dependency analysis is defined as:*

$$\gamma^{\mathsf{u}}(\langle m_0, m_1, \dots m_n \rangle) \stackrel{\text{def}}{=} \{D \in \wp(\Sigma \times \Sigma^{\perp}) \mid \textsc{Unused}_{\{\mathtt{x} \in \Delta \mid m_0(\mathtt{x}) = \mathrm{N}\}}(D)\}$$

Finally, we show that the syntactic dependency analysis $\Lambda^{\mathsf{u}}[\![P]\!]$ is sound for proving that a program does not use a subset of its input variables.

Theorem 3.6.1 *A program P does not use the set of input variables* `W` *only if the concretization of the syntactic dependency analysis Λ^{u} is included in the unused property $\mathcal{N}_{\mathtt{W}}$.*

$$\gamma^{\rightsquigarrow} \circ \gamma^{\mathsf{u}}(\Lambda^{\mathsf{u}}[\![P]\!]) \subseteq \mathcal{N}_{\mathtt{W}} \implies P \models \mathcal{N}_{\mathtt{W}}$$

Proof. Assuming that $\gamma^{\rightsquigarrow} \circ \gamma^{\mathsf{u}}(\Lambda^{\mathsf{u}}[\![P]\!]) \subseteq \mathcal{N}_{\mathtt{W}}$, we have that $\Lambda^{\rightsquigarrow}[\![P]\!] \subseteq \gamma^{\mathsf{u}}(\Lambda^{\mathsf{u}}[\![P]\!])$ by Definition 3.6.1 of the concretization γ^{u}. By monotonicity of

$\gamma^{\rightsquigarrow}$ (cf. Definition 3.4.1), it holds that $\gamma^{\rightsquigarrow}(\Lambda^{\rightsquigarrow}[\![P]\!]) \subseteq \gamma^{\rightsquigarrow}(\gamma^{u}(\Lambda^{u}[\![P]\!])) \subseteq \mathcal{N}_{W}$. Thus, we conclude that $P \models \mathcal{N}_{W}$ from Theorem 3.4.2. □

Note that, the syntactic dependency analysis does not consider non-termination. For instance, if a variable solely determine the termination of a loop but is not used in the output computation, the analysis will not capture such dependency. In order to take termination into account, a naïve approach could be to map each variable appearing in the guard of a loop to U. A more sophisticated one could be to run a termination analysis alongside the data usage analysis. Only variables that are used in the guard of non-terminating loops would be marked as U, thus refining the set of used variables compared to the previous approach.

3.7 Summary

In this chapter, we introduced the notion of input data usage and defined an abstract version inspired by the definition of abstract non-interference. We presented the hierarchy of semantics that allows reasoning about the usage of input variables of a program. We showed a computable semantics collecting syntactic dependencies among variables. In the next chapter, we will define a quantitative counterpart of input data usage, able to measure the impact of variations in the input data on the outcome of a program. Notably, we will exploit the output abstraction to obtain numerical values from the output states.

Dans ce chapitre, nous avons introduit la notion d'utilisation des données d'entrée et défini une version abstraite inspirée de la définition de la non-interférence abstraite. Nous avons présenté la hiérarchie des sémantiques permettant de raisonner sur l'utilisation des variables d'entrée d'un programme. Nous avons montré une sémantique calculable qui collecte les dépendances syntaxiques entre les variables. Dans le prochain chapitre, nous définirons une contrepartie quantitative de l'utilisation des données d'entrée, capable de mesurer l'impact des variations des données d'entrée sur le résultat d'un programme. Notamment, nous exploiterons l'abstraction de sortie pour obtenir des valeurs numériques à partir des états de sortie.

Quantitative Verification of Extensional Properties

4 Quantitative Input Data Usage

In this chapter, we present a quantitative notion of input data usage, extending the definition introduced in the previous chapter. We define three different quantifiers for the impact of the input variables on the program outcome, and we formalize a static analysis for verifying the resulting quantitative impact property. At the end, we present the abstract versions of these quantifiers and show how they can be used to verify the corresponding quantitative property. This chapter is based on work published at NASA Formal Methods Symposium (NFM) 2024 [46]. Later in this thesis, we will apply our quantitative framework in the context of neural networks and timing side-channel attacks.

[46]: Mazzucato et al. (2024), 'Quantitative Input Usage Static Analysis'

Dans ce chapitre, nous présentons une notion quantitative de l'utilisation des données d'entrée, en étendant la définition introduite dans le chapitre précédent. Nous définissons trois quantificateurs différents pour mesurer l'impact des variables d'entrée sur le résultat du programme, et nous formalisons une analyse statique pour vérifier la propriété d'impact quantitatif qui en résulte. Enfin, nous présentons les versions abstraites de ces quantificateurs et montrons comment ils peuvent être utilisés pour vérifier la propriété quantitative correspondante. Ce chapitre est basé sur des travaux publiés au NASA Formal Methods Symposium (NFM) 2024 [46]. Plus tard dans cette thèse, nous appliquerons notre cadre quantitatif dans le contexte des réseaux neuronaux et des attaques par canaux auxiliaires temporels.

4.1 The k-Bounded Impact Property

Quantitative input data usage should be understood as quantifying how much the input data influences the program outcome. To this end, we define the *k-bounded impact property* $\mathcal{B}^{\otimes k}_{\textsc{Impact}_{\mathtt{W}}}$ as the set of program semantics such that the input variables $\mathtt{W} \in \wp(\Delta)$ have an impact on the program outcome bounded by $k \in \mathbb{V}^{+\infty}_{\geq 0}$. Depending on the comparison operator $\otimes \in \{\leq, \geq\}$, the impact quantity is bounded by k either from above or below. We employ the *impact quantifier*:

$$\textsc{Impact}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{V}^{+\infty}_{\geq 0}$$

to quantify the impact of the input variables $\mathtt{W}$ on the outcome of program computations.

The set $\wp(\Delta)$ contains the input variables of a program.

The set $\mathbb{V}^{+\infty}_{\geq 0}$ contains non-negative numerical values as well as with $+\infty$.

Definition 4.1.1 (k-Bounded Impact Property) *Let $\mathtt{W} \in \wp(\Delta)$ be the set of input variables of interest, $\textsc{Impact}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{V}^{+\infty}_{\geq 0}$ an impact quantifier, and $k \in \mathbb{V}^{+\infty}_{\geq 0}$ the threshold. The* k-bounded impact property *$\mathcal{B}^{\otimes k}_{\textsc{Impact}_{\mathtt{W}}}$ is defined as follows:*

$$\mathcal{B}^{\otimes k}_{\textsc{Impact}_{\mathtt{W}}} \stackrel{\text{def}}{=} \{\Lambda \in \wp(\Sigma^{+\infty}) \mid \textsc{Impact}_{\mathtt{W}}(\Lambda) \otimes k\}$$

As a comparison, here is the definition of the qualitative counterpart $\mathcal{N}_{\mathtt{W}}$:

Definition 3.3.1 (Unused Property)

$\mathcal{N}_{\mathtt{W}} \stackrel{\text{def}}{=} \{\Lambda \in \wp(\Sigma^{+\infty}) \mid \textsc{AUnused}_{\mathtt{W}}(\Lambda)\}$

where the comparison operator $\otimes$ is either $\leq$ or $\geq$.

We allow the measured impact to be either less than or equal to ($\leq$) or greater than or equal to ($\geq$) the threshold k. With such a definition, we allow abstractions of program semantics to return an always greater or, respectively, an always smaller impact than the actual one and thus provide a sound over-approximation of the impact property.

Remark 4.1.1 A program P satisfies the k-bounded impact property $\mathscr{B}^{\otimes k}_{\textsc{Impact}_W}$ if and only if the collecting semantics of P implies $\mathscr{B}^{\otimes k}_{\textsc{Impact}_W}$, formally:

$$P \models \mathscr{B}^{\otimes k}_{\textsc{Impact}_W} \;\Leftrightarrow\; \Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathscr{B}^{\otimes k}_{\textsc{Impact}_W}$$

As a consequence of the k-bounded impact property, cf. Definition 4.1.1, we notice that by enlarging the threshold k, the set of program semantics that satisfy the property also increases.

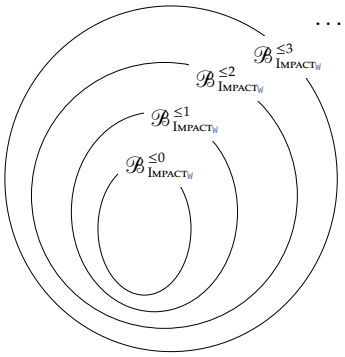


Figure 4.1: Illustration of the k-bounded impact property $\mathscr{B}^{\leq k}_{\textsc{Impact}_W}$ for different values of k.

Lemma 4.1.1 (Monotonicity of the k-Bounded Impact Property) *For all $k, k' \in \mathbb{V}^{+\infty}_{\geq 0}$ such that $k \otimes k'$, it holds that:*

$$\mathscr{B}^{\otimes k}_{\textsc{Impact}_W} \subseteq \mathscr{B}^{\otimes k'}_{\textsc{Impact}_W}$$

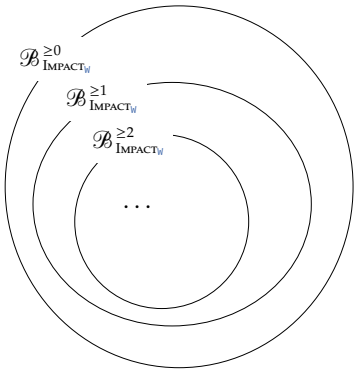


Figure 4.2: Illustration of the k-bounded impact property $\mathscr{B}^{\geq k}_{\textsc{Impact}^{\natural}_W}$ for different values of k.

Proof. The proof is based on the fact that, for all $k, k' \in \mathbb{V}^{+\infty}_{\geq 0}$ such that $k \otimes k'$, it holds that $\textsc{Impact}_W(T) \otimes k \Rightarrow \textsc{Impact}_W(T) \otimes k'$ for any set of traces $T \in \wp(\Sigma^{+\infty})$. Consequently, we have that $\mathscr{B}^{\otimes k}_{\textsc{Impact}_W} \subseteq \mathscr{B}^{\otimes k'}_{\textsc{Impact}_W}$. □

Note that, in both cases whether $\otimes$ is $\leq$ or $\geq$, the k-bounded impact property $\mathscr{B}^{\otimes k}_{\textsc{Impact}_W}$ is always a subset of $\mathscr{B}^{\otimes k'}_{\textsc{Impact}_W}$. Figure 4.1 compares the property $\mathscr{B}^{\leq k}_{\textsc{Impact}_W}$ for different values of k, the smallest being $k = 0$ containing the program semantics whose input variables have no impact. By increasing k, the set of program semantics that satisfy the property also increases. Regarding the other operator $\geq$, Figure 4.2 shows that $\mathscr{B}^{\geq 0}_{\textsc{Impact}_W}$ is the property containing all the program semantics, *i.e.*, $\mathscr{B}^{\geq 0}_{\textsc{Impact}_W} = \wp(\Sigma^{+\infty})$. By decreasing k, the set of program semantics that satisfy the property also increases. Therefore, the k-bounded impact property $\mathscr{B}^{\leq k}_{\textsc{Impact}_W}$ can be used to verify that the input variables W have an impact of *at least* k on the program outcome. Whereas, the property $\mathscr{B}^{\geq k}_{\textsc{Impact}^{\natural}_W}$ can be used to verify that the impact is of *at most* k.

Def. 3.2.2 (Output Observer) *Given a program P, an* output observer $\rho \in \Sigma^{\perp} \to \Sigma^{\perp}$ *is an upper closure operator that abstracts the value of output states.*

Prog. 4.1: Landing alarm system

```
1  landing_coeff =
2    abs(angle) + speed
3  if landing_coeff < 2:
4    risk = 0
5  else if landing_coeff > 5:
6    risk = 3
7  else:
8    risk =
9      floor(landing_coeff) - 2
10
```

Next, we instantiate Impact with the impact quantifiers of Outcomes, Range, and QUsed. Each of these quantifiers differently characterizes the impact of the input variables: Outcomes counts the number of different outcomes, Range measures the range of outcome values, and QUsed counts the number of input values that are *not* used to produce the outcomes. To this end, these quantifiers employ the concept of output observer, cf. Definition 3.2.2.

Example 4.1.1 Consider the Program 4.1, called for simplicity program L in the following. The goal of program L is to inform the pilot about the level of risk associated with the landing approach. It takes two

input variables, denoted as `angle` and `speed`, for the aircraft-airstrip alignment angle and the aircraft speed, respectively. A value of 1 represents a good alignment while -4 a non-aligned angle, whereas 1, 2, 3 denote low, medium, and high speed.[1] A safer approach is indicated by lower speed. The landing risk coefficient combines the absolute landing angle and speed. The output variable `risk` is the danger level with possible values $\{0, 1, 2, 3\}$, where 0 represents low danger and 3 high danger. Figure 4.3 shows the input space composition of this system, where the label near each input represents the degree of risk assigned to the corresponding input configuration. It is easy to note that a nonaligned angle of approach corresponds to a considerably higher level of risk, whereas the risk with a correct angle depends mostly on the aircraft speed.

1: We initially focus on discrete values to simplify the example and convey the concept. We expand to continuous inputs in Chapter 5.

Program states are $\Sigma = \{\langle a, b, c, d\rangle \mid a \in \{-4, 1\} \wedge b \in \{1, 2, 3\} \wedge c \in \mathbb{N} \wedge d \in \{0, 1, 2, 3\}\}$, where a is the value of angle, b of `speed`, c of `landing_coeff`, and d of `risk`. Here, we abuse the notation and write states in Σ as tuples of variable values, without considering program locations. The output observer is instantiated with

$$\rho(s) \stackrel{\text{def}}{=} \begin{cases} \langle \top, \top, \top, d\rangle & \text{if } s = \langle a, b, c, d\rangle \\ \langle \top, \top, \top, \top\rangle & \text{otherwise} \end{cases}$$

As a result, the output observer ρ focuses on the value of the variable `risk` when the last state of the trace is $\langle a, b, c, d\rangle$. Otherwise, in case of an infinite-length trace, the output observer maps to the state with all variables at $\top$. In other words, the output observer abstracts away the value of all variables but `risk`, the output variable. Note that, even though Program 4.1 always terminates, we need to take into account that the input parameter of the output observer ρ is either the final state or $\perp$. Figure 4.4 depicts the traces generated by Program 4.1 where the output values are abstracted by ρ.

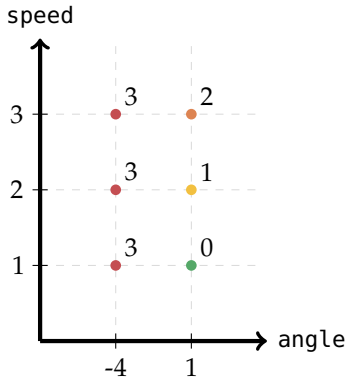


Figure 4.3: Input space composition of program L.

Example 4.1.2 Originally introduced by Giacobazzi and Mastroeni [43], the output observer ρ should deal with any possible abstraction an end-user of the framework might want to use. For example, one could be interested in the parity of the output value. In this case, the output observer ρ could be defined as:

$$\rho(s) \stackrel{\text{def}}{=} \begin{cases} \langle \top, \top, \top, d \mod 2\rangle & \text{if } s = \langle a, b, c, d\rangle \\ \langle \top, \top, \top, \top\rangle & \text{otherwise} \end{cases}$$

The parity of the value of `risk` is mapped to the value 0 if even and 1 if odd. As a result, in our framework we would quantify the impact of the input variables on the parity of the output values. Figure 4.5 shows the traces generated by Program 4.1 where the output values are abstracted by their parity.

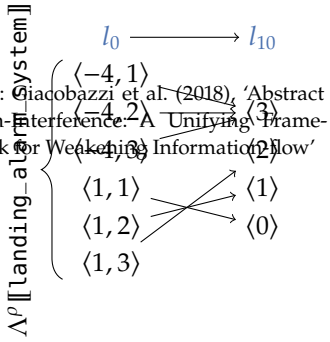


Figure 4.4: Graphical representation of the trace semantics of the Program 4.1.

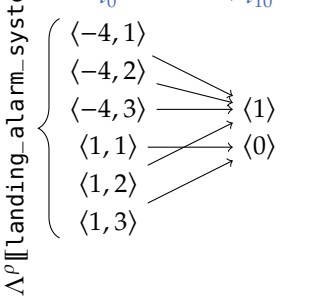


Figure 4.5: Graphical representation of the trace semantics of the Program 4.1 with the parity abstraction.

Next, we define the impact quantifiers of Outcomes, Range, and QUsed.

[43]: Giacobazzi et al. (2018), 'Abstract Non-Interference: A Unifying Framework for Weakening Information-flow'

4.1.1 The Outcomes Impact Quantifier

Formally $\text{OUTCOMES}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{N}^{+\infty}$ counts the number of different output values reachable by varying the input variables $\mathtt{W} \in \wp(\Delta)$.

$\text{OUTCOMES}_{\mathtt{W}}$ collects, for any possible input configuration $s_0 \in \Sigma|_\Delta$, all the traces that are starting from an input configuration that is a variation of s_0 on the input variable $\mathtt{W}$, *i.e.*, $\{\sigma \in T \mid \sigma_0 =_{\Delta\setminus\mathtt{W}} s_0\}$, where $T \in \wp(\Sigma^{+\infty})$. Then, it gathers the output values of this set of traces by means of the output observer ρ, *i.e.*, $\{\rho(\sigma_\omega) \mid \sigma \in T \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} s_0\}$. Specifically, this set contains all the output abstractions performed by ρ. Afterwards, it yields the number of these output values via the cardinality operator $|\cdot|$. Finally, through the supremum operator: sup, it returns the maximum value to ensure that the greatest impact is preserved.

Formally, the Outcomes impact quantifier is defined as follows:

Definition 4.1.2 (Outcomes) *Given an input variable* $\mathtt{W} \in \wp(\Delta)$ *and output observer* $\rho \in \Sigma^\perp \to \Sigma^\perp$*, the impact quantifier* $\text{OUTCOMES}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{N}^{+\infty}$ *is defined as:*

$$\text{OUTCOMES}_{\mathtt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma} |\{\rho(\sigma_\omega) \mid \sigma \in T \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} s_0\}|$$

Example 4.1.3 Let us revisit the example of the landing alarm system, cf. Program 4.1, with program states $\Sigma = \{\langle a, b, c, d\rangle \mid a \in \{-4, 1\} \wedge b \in \{1, 2, 3\} \wedge c \in \mathbb{N} \wedge d \in \mathbb{N}_{\leq 3}\}$. For brevity, we refer to such program as program L. The input variables are $\Delta = \{\mathtt{angle}, \mathtt{speed}\}$, consequently the input space is the set of program states reduced to the input variables, *i.e.* $\Sigma|_\Delta = \{\langle -4, 1\rangle, \langle -4, 2\rangle, \langle -4, 3\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle\}$. We begin by considering $\mathtt{W} = \{\mathtt{angle}\}$ and $\langle -4, 1\rangle$ as the first input configuration to be explored. Hence, we collect all traces that are starting from an input configuration that is a variation of $\langle -4, 1\rangle$, *i.e.*, $\{\sigma \in \Lambda[\![L]\!] \mid \sigma_0 =_{\Delta\setminus\{\mathtt{angle}\}} \langle -4, 1\rangle\}$, where $\Delta \setminus \{\mathtt{angle}\} = \{\mathtt{speed}\}$ and consequently $\sigma_0 =_{\{\mathtt{speed}\}} \langle -4, 1\rangle$ holds whenever the initial state of σ has $\mathtt{speed} = 1$. A possible trace (with the intermediate states and the value of all the 4 variables) of this set is $\langle 1, 1, 0, 0\rangle \to \langle 1, 1, 2, 0\rangle \to \langle 1, 1, 2, 0\rangle$ where, at the beginning, we randomly assign $\mathtt{landing\_risk} = 0$ and $\mathtt{risk} = 0$, respectively the third and fourth component of the initial state. We collect the output values of this set of traces, $\{\sigma_\omega(\mathtt{risk}) \mid \sigma \in \Lambda[\![L]\!] \wedge \sigma_0(\mathtt{speed}) = 1\}$. As a result, we obtain the set of output values $\{0, 3\}$. For instance, the output value 0 is the result of the trace we exhibited previously, where the last state is $\langle 1, 1, 2, 0\rangle$ and thus the $\mathtt{risk}$ variable of this trace is the last component with value 0. Finally, the cardinality operator returns the value 2, $|\{0, 3\}| = 2$. By doing so for all possible input configurations in $\Sigma|_\Delta$, we obtain $\text{OUTCOMES}_{\{\mathtt{angle}\}}(\Lambda[\![L]\!]) = 2$. Table 4.1 illustrate the steps for $\text{OUTCOMES}_{\{\mathtt{angle}\}}(\Lambda[\![L]\!])$ and Table 4.2 for $\text{OUTCOMES}_{\{\mathtt{speed}\}}(\Lambda[\![L]\!])$.

The Outcomes quantifier above is monotone in the amount of traces in T.

Prog. 4.1 (Landing alarm system):

```
landing_coeff =
  abs(angle) + speed
if landing_coeff < 2:
  risk = 0
else if landing_coeff > 5:
  risk = 3
else:
  risk =
   floor(landing_coeff) - 2
```

The supremum operator sup yields the least upper bound of the given set of elements X:

$$\sup X \stackrel{\text{def}}{=} x \text{ such that } \forall y \in X.\, x \geq y$$

Table 4.1: Overview of the Outcomes impact quantifier for the variable `angle` of Program 4.1 (Landing alarm system).

Input	Traces	Outcomes
$\langle -4, 1\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle 1, 1\rangle \to \langle 0\rangle$	2
$\langle -4, 2\rangle$	$\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$	2
$\langle -4, 3\rangle$	$\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$, $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	2
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$	2
$\langle 1, 2\rangle$	$\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$	2
$\langle 1, 3\rangle$	$\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$, $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	2

Table 4.2: Overview of the Outcomes impact quantifier for the variable `speed` of Program 4.1 (Landing alarm system).

Input	Traces	Outcomes
$\langle -4, 1\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$ $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	1
$\langle -4, 2\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$ $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	1
$\langle -4, 3\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$ $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	1
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$ $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	3
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$ $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	3
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$ $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	3

Lemma 4.1.2 (Outcomes is Monotonic) *For all set of traces* $T, T' \in \wp(\Sigma^{+\infty})$*, it holds that:*

$$T \subseteq T' \implies \textsc{Outcomes}_W(T) \leq \textsc{Outcomes}_W(T')$$

Proof. The proof is based on the observation that, fixed an input state $s_0 \in \Sigma$, the set of output values $\{\rho(\sigma_\omega) \mid \sigma \in T \wedge \sigma_0 =_{\Delta\setminus W} s_0\}$ is a subset of $\{\rho(\sigma_\omega) \mid \sigma \in T' \wedge \sigma_0 =_{\Delta\setminus W} s_0\}$ since by hypothesis we have that $T \subseteq T'$. Hence, the cardinality of the first set is less than or equal to the cardinality of the second set. We conclude that $\textsc{Outcomes}_W(T) \leq \textsc{Outcomes}_W(T')$. □

Let $\mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W}$ be the k-bounded impact property when instantiated with the Outcomes impact quantifier. The output semantics Λ^ρ is sound and complete for validating $\mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W}$.

Lemma 4.1.3 ($\mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W}$ Validation)

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W} \iff \Lambda^\rho[\![P]\!] \subseteq \alpha^\rho(\alpha^{\rightsquigarrow}(\mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W}))$$

Proof (Sketch). Let us consider the $\alpha^\rho \circ \alpha^{\rightsquigarrow}$ abstraction of the k-bounded impact property $\mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W}$:

$$\alpha^\rho(\alpha^{\rightsquigarrow}(\mathcal{B}^{\circledast k}_{\textsc{Outcomes}_W})) = \{D \in \wp(\Sigma \times \Sigma^\perp) \mid \sup_{s_0 \in \Sigma} |\{z \mid \langle s_0, z\rangle \in D \wedge \sigma_0 =_{\Delta\setminus W} s_0\}| \circledast k\}$$

Note that, all the intermediate states from the trace semantics are abstracted to input-output dependencies. The Outcomes impact quantifier does not consider the intermediate states, thus the abstraction to dependencies does not affect the validation of the property. Furthermore, even handling the output abstraction at the semantic level, by abstracting output states to abstract output states, does not affect the validation of the property as the Outcomes impact quantifier already abstracts the output values before counting them. □

For the Outcomes quantifier, an impact quantity of zero implies that the input variables of interest have no impact on the program outcomes, *i.e.*, the set of input variables is unused, provided that the given program semantics *is deterministic*. On the other hand, non-deterministic statements in the program may mislead Outcomes to account for differences in the program output as a result of variations in the input variables.

Example 4.1.4 Consider the Program 3.2 used in Section 3.1 to illustrate the differences between syntactic and semantic input data usage. Notably: it contains a non-deterministic statement. Assuming input values are $\mathbb{N}$, the $\textsc{Outcomes}_x$ impact quantifier (for the variable x) is $+\infty$ as for any possible input value, the program can output any value in $\mathbb{N}$. Due to the non-deterministic statement, the variable x has an infinite impact on the output values, even though the program does not use the input variable x, cf. Example 3.1.2.

Def. 3.5.2 (Output-Abstraction Semantics)

$$\Lambda^\rho \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \rho(\sigma_\omega)\rangle \mid \sigma \in \Lambda\}\}$$

Def. 3.4.1 (Right-Left Adjoints for the Dependency Semantics)

$$\alpha^{\rightsquigarrow} \in \wp(\wp(\Sigma^{+\infty})) \to \wp(\wp(\Sigma \times \Sigma^\perp))$$
$$\alpha^{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \sigma_\omega\rangle \mid \sigma \in T\} \mid T \in S\}$$

Def. 3.5.1 (Right-Left Adjoints for the Output-Abstraction Semantics)

$$\alpha^\rho \in \wp(\wp(\Sigma \times \Sigma^\perp)) \to \wp(\wp(\Sigma \times \Sigma^\perp))$$
$$\alpha^\rho(W) \stackrel{\text{def}}{=} \left\{ \{\langle s_0, \rho(s_\omega)\rangle \mid \langle s_0, s_\omega\rangle \in D\} \;\middle|\; D \in W \right\}$$

Prog. 3.2 (Syntactic versus semantic usage of the input variable x.)**:**

```
1 x_plus_rand(x):
2   return x + rand();
3
```

Notably, the property $\mathcal{B}^{\leq k}_{\textsc{Outcomes}_W}$ (with the operator $\leq$) is subset-closed, *i.e.*, if a program semantics satisfies the property, then all its subsets also satisfy the property.

Lemma 4.1.4 ($\mathcal{B}^{\leq k}_{\textsc{Outcomes}_W}$ is subset-closed) *For any two sets of traces $T, T' \in \wp(\Sigma^{+\infty})$, it holds that:*

$$T \subseteq T' \wedge T' \in \mathcal{B}^{\leq k}_{\textsc{Outcomes}_W} \Rightarrow T \in \mathcal{B}^{\leq k}_{\textsc{Outcomes}_W}$$

Proof. This proof follows directly from the monotonicity of the $\textsc{Outcomes}_W$ impact quantifier, cf. Lemma 4.1.2. □

Conversely, the property $\mathcal{B}^{\geq k}_{\textsc{Outcomes}_W}$ (with the operator $\geq$) is superset-closed, *i.e.*, if a program semantics satisfies $\mathcal{B}^{\geq k}_{\textsc{Outcomes}_W}$, then all its supersets also satisfy $\mathcal{B}^{\geq k}_{\textsc{Outcomes}_W}$.

Lemma 4.1.5 ($\mathcal{B}^{\geq k}_{\textsc{Outcomes}_W}$ is superset-closed) *For any two sets of traces $T, T' \in \wp(\Sigma^{+\infty})$, it holds that:*

$$T \subseteq T' \wedge T \in \mathcal{B}^{\geq k}_{\textsc{Outcomes}_W} \Rightarrow T' \in \mathcal{B}^{\geq k}_{\textsc{Outcomes}_W}$$

Proof. Similarly to Lemma 4.1.4, this proof follows directly from the monotonicity of the $\textsc{Outcomes}_W$ impact quantifier, cf. Lemma 4.1.2. □

4.1.2 The Range Impact Quantifier

The quantity $\textsc{Range}_W \in \wp(\Sigma^{+\infty}) \to \mathbb{R}^{+\infty}_{\geq 0}$ determines the size of the range of the output values for all the possible variations in the input variable $W \in \wp(\Delta)$. Let Z be the set of output variables on which the impact is measured. We are interested in the difference between the extreme values of the variables Z. The difference of extreme values is computed by a distance measure Distance computing the maximum distance between the set of given values. For instance, Distance could be the Euclidean distance.

Example 4.1.5 We revisit again the example of the landing alarm system, program L. Assuming $W = \{\texttt{angle}\}$, $\langle -4, 1\rangle$ is the first input configuration to be explored, we collect all traces that are starting from an input configuration that is a variation of $\langle -4, 1\rangle$. As before, we obtain $\{\sigma_\omega(\texttt{risk}) \mid \sigma \in T \wedge \sigma_0(\texttt{speed}) = 1\} = \{0, 3\}$ as the set of output values for the output variable `risk`. Here, we are interested in the difference between the extreme values of the outcomes. Hence, we apply the size function: $\textsc{Distance}(\{0, 3\}) = 3$. By doing so for all possible input configurations $\Sigma|_\Delta$, we obtain $\textsc{Range}_{\{\texttt{angle}\}}(\Lambda[\![L]\!]) = 3$. Respectively, Table 4.3 illustrates the steps for $\textsc{Range}_{\{\texttt{angle}\}}(\Lambda[\![L]\!])$ and Table 4.4 for $\textsc{Range}_{\{\texttt{speed}\}}(\Lambda[\![L]\!])$.

Formally, the Range impact quantifier is defined as follows:

Table 4.3: Overview of the Range impact quantifier for the variable `angle` of Program 4.1 (Landing alarm system).

Input	Traces	Range
$\langle -4, 1\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$	3
$\langle -4, 2\rangle$	$\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$	2
$\langle -4, 3\rangle$	$\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$, $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	1
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$	3
$\langle 1, 2\rangle$	$\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$	2
$\langle 1, 3\rangle$	$\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$, $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	1

Table 4.4: Overview of the Range impact quantifier for the variable `speed` of Program 4.1 (Landing alarm system).

Input	Traces	Range
$\langle -4, 1\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$ $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	0
$\langle -4, 2\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$ $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	0
$\langle -4, 3\rangle$	$\langle -4, 1\rangle \to \cdots \to \langle 3\rangle$, $\langle -4, 2\rangle \to \cdots \to \langle 3\rangle$ $\langle -4, 3\rangle \to \cdots \to \langle 3\rangle$	0
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$ $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	2
$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \cdots \to \langle 0\rangle$, $\langle 1, 2\rangle \to \cdots \to \langle 1\rangle$ $\langle 1, 3\rangle \to \cdots \to \langle 2\rangle$	2

Definition 4.1.3 (Range) *Given the set of input variables of interest* $\mathtt{W} \in \wp(\Delta)$ *and the output variables* $\mathtt{Z}$*, the impact quantifier* $\textsc{Range}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{R}^{+\infty}_{\geq 0}$ *is defined as*

$$\textsc{Range}_{\mathtt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}(\{\rho(\sigma_\omega)(\mathtt{Z}) \mid \sigma \in T \wedge \sigma_0 =_{\Delta \setminus \mathtt{W}} s_0\})$$

The above quantifier employs the auxiliary function Distance. For instance, such distance could be instantiated as the Euclidean distance: $\textsc{Distance}(X) \stackrel{\text{def}}{=} \max_{x,x' \in X} \sqrt{\sum_k (x - x')^2}$ if X is non-empty, otherwise $\textsc{Distance}(X) \stackrel{\text{def}}{=} 0$. From here, we consider the Euclidean distance as the default distance measure.

We note that also Range is monotonic in the amount of traces in T.

Lemma 4.1.6 (Range is Monotonic) *For all set of traces* $T, T' \in \wp(\Sigma^{+\infty})$*, it holds that:*

$$T \subseteq T' \implies \textsc{Range}_{\mathtt{W}}(T) \leq \textsc{Range}_{\mathtt{W}}(T')$$

Proof. Similarly to Lemma 4.1.2, the proof is based on the observation that the set of output values $\{\rho(\sigma_\omega)(\mathtt{Z}) \mid \sigma \in T \wedge \sigma_0 =_{\Delta \setminus \mathtt{W}} s_0\}$ is a subset of the set of output values $\{\rho(\sigma_\omega)(\mathtt{Z}) \mid \sigma \in T' \wedge \sigma_0 =_{\Delta \setminus \mathtt{W}} s_0\}$ since by hypothesis we have that $T \subseteq T'$. Hence, the size of the first set is less than or equal to the cardinality of the second set. We conclude that $\textsc{Range}_{\mathtt{W}}(T) \leq \textsc{Range}_{\mathtt{W}}(T')$. □

Let $\mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}}$ be the k-bounded impact property when instantiated with the Range impact quantifier. The output semantics Λ^ρ is sound and complete for validating $\mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}}$.

Lemma 4.1.7 ($\mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}}$ Validation)

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}} \iff \Lambda^\rho[\![P]\!] \subseteq \alpha^\rho(\alpha^{\rightsquigarrow}(\mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}}))$$

Proof (Sketch). Let us consider the (double) abstraction of the k-bounded impact property $\mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}}$:

$$\begin{aligned}&\alpha^\rho(\alpha^{\rightsquigarrow}(\mathcal{B}^{\circledcirc k}_{\textsc{Range}_{\mathtt{W}}})) = \\ &\quad \{D \in \wp(\Sigma \times \Sigma^\perp) \mid \\ &\qquad \sup_{s_0 \in \Sigma} \textsc{Distance}(\{s_\omega(\mathtt{Z}) \mid \langle s_0, s_\omega \rangle \in D \wedge \sigma_0 =_{\Delta \setminus \mathtt{W}} s_0\}) \circledcirc k\}\end{aligned}$$

As noticed before, the Range impact quantifier does not consider the intermediate states and abstracts the output states before applying the size function. Hence, neither the abstraction to dependencies nor the abstraction to output values affect the validation of the property. □

Lemma 4.1.6 and Lemma 4.1.7 show that the Range impact quantifier can be used to certify that a program has impact of *at most* k. In the present of deterministic systems, an impact quantity of zero implies that the input

Def. 3.5.2 (Output-Abstraction Semantics)

$$\Lambda^\rho \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \rho(\sigma_\omega) \rangle \mid \sigma \in \Lambda\}\}$$

Def. 3.4.1 (Right-Left Adjoints for the Dependency Semantics)

$$\alpha^{\rightsquigarrow} \in \wp(\wp(\Sigma^{+\infty})) \to \wp(\wp(\Sigma \times \Sigma^\perp))$$
$$\alpha^{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in T\} \mid T \in S\}$$

Def. 3.5.1 (Right-Left Adjoints for the Output-Abstraction Semantics)

$$\alpha^\rho \in \wp(\wp(\Sigma \times \Sigma^\perp)) \to \wp(\wp(\Sigma \times \Sigma^\perp))$$
$$\alpha^\rho(W) \stackrel{\text{def}}{=} \left\{ \{\langle s_0, \rho(s_\omega) \rangle \mid \langle s_0, s_\omega \rangle \in D\} \;\middle|\; D \in W \right\}$$

variable is unused. Otherwise, similarly to RANGE, the input variable may have an impact on the program computation even if not used.[2]

2: See Example 4.1.4 for a similar discussion of the OUTCOMES impact quantifier when applied to programs containing nondeterministic statements.

Notably, the property $\mathscr{B}^{\leq k}_{\text{RANGE}_W}$ (with the operator $\leq$) is subset-closed, *i.e.*, if a program semantics satisfies the property, then all its subsets also satisfy the property.

Lemma 4.1.8 ($\mathscr{B}^{\leq k}_{\text{RANGE}_W}$ is subset-closed) *For any two sets of traces* $T, T' \in \wp(\Sigma^{+\infty})$*, it holds that:*

$$T \subseteq T' \wedge T' \in \mathscr{B}^{\leq k}_{\text{RANGE}_W} \implies T \in \mathscr{B}^{\leq k}_{\text{RANGE}_W}$$

Proof. This proof follows directly from the monotonicity of the RANGE_W impact quantifier, cf. Lemma 4.1.6. □

Conversely, the property $\mathscr{B}^{\geq k}_{\text{RANGE}_W}$ (with the operator $\geq$) is superset-closed, *i.e.*, if a program semantics satisfies $\mathscr{B}^{\geq k}_{\text{RANGE}_W}$, then all its supersets also satisfy $\mathscr{B}^{\geq k}_{\text{RANGE}_W}$.

Lemma 4.1.9 ($\mathscr{B}^{\geq k}_{\text{RANGE}_W}$ is superset-closed) *For any two sets of traces* $T, T' \in \wp(\Sigma^{+\infty})$*, it holds that:*

$$T \subseteq T' \wedge T \in \mathscr{B}^{\geq k}_{\text{RANGE}_W} \implies T' \in \mathscr{B}^{\geq k}_{\text{RANGE}_W}$$

Proof. Similarly to Lemma 4.1.8, this proof follows directly from the monotonicity of the RANGE_W impact quantifier, cf. Lemma 4.1.6. □

4.1.3 The QUSED Impact Quantifier

The quantifier $\text{QUSED}_W \in \wp(\Sigma^{+\infty}) \to \mathbb{N}^{+\infty}$ counts the number of input states for the variables in $W \in \wp(\Delta)$ that are *not* used to produce the program outcomes. Unlike OUTCOMES and RANGE, even in the presence of non-deterministic statements, the impact of an input variable is zero w.r.t. QUSED if and only if the input variable is not used to produce any outcome.

Given a set of traces T, for each possible outcome of a trace $\sigma \in T$, QUSED collects the traces $\sigma' \in T$ that share the same output abstraction, *i.e.*, $\rho(\sigma'_\omega) = \rho(s_\omega)$. Then, it gathers the set of input states of the variables in W that correspond to these traces, called $I_\sigma(T)$. The number of missing input states is obtained by subtracting the set $I_\sigma(T)$, from the set of input states that can produce the outcome s_ω by variations of the variables in W, called $Q_W(I_\sigma(T))$. Finally, QUSED_W yields the maximum number of input values that are not used to produce the outcomes.

Formally, the QUSED impact quantifier is defined as follows:

Definition 4.1.4 (QUSED) *Given an input variable* $\mathtt{W} \in \wp(\Delta)$*, the impact quantifier* $\text{QUSED}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{N}^{+\infty}$ *is defined as*

$$\begin{aligned} \text{QUSED}_{\mathtt{W}}(T) &\stackrel{\text{def}}{=} \sup_{\sigma \in T} | \, Q_{\mathtt{W}}(I_\sigma(T)) \setminus I_\sigma(T) \, | \\ I_\sigma(T) &= \{\sigma'_0 \mid \sigma' \in T \wedge \rho(\sigma_\omega) = \rho(\sigma'_\omega)\} \\ Q_{\mathtt{W}}(S) &= \{s' \in \Sigma \mid s \in S \wedge s' =_{\Delta \setminus \mathtt{W}} s\} \end{aligned}$$

where $I_\sigma(T)$ *is the set of input states of traces in* T *that share the same output abstraction as* σ *and* $Q_{\mathtt{W}}(S)$ *is the set of states that are variations from states in* S *only in the variables* $\mathtt{W}$*.*

Example 4.1.6 To demonstrate the QUSED impact quantifier, we employ four simple programs. The first program we consider, Program 4.2, returns the value of the only input variable x without any modification. The second program, Program 4.3, returns a random value. The last two, Program 4.4 and Program 4.5, combine together non-deterministic and deterministic statements. For simplicity, variables are assumed to range in $\{0, 1, 2\}$.

The trace semantics of Program 4.2 is:

$$\Lambda[\![\texttt{id}]\!] = \left\{ \begin{array}{l} (l_0, \langle 0 \rangle) \to \cdots \to (l_3, \langle 0 \rangle), \\ (l_0, \langle 1 \rangle) \to \cdots \to (l_3, \langle 1 \rangle), \\ (l_0, \langle 2 \rangle) \to \cdots \to (l_3, \langle 2 \rangle) \end{array} \right\}$$

As explained above, for any $\sigma \in \Lambda[\![\texttt{id}]\!]$ we collect the set of input states that belong to traces $\sigma' \in \Lambda[\![\texttt{id}]\!]$ that share the same output abstraction as σ, cf. $I_\sigma(\Lambda[\![\texttt{id}]\!])$. In this case, the set of input states that can produce the outcome 0 is $\{(l_0, \langle 0 \rangle)\}$, for the outcome 1 is $\{(l_0, \langle 1 \rangle)\}$, and for the outcome 2 is $\{(l_0, \langle 2 \rangle)\}$, all singletons. Hence, the set of states that are variations, cf. $Q_{\{\mathtt{x}\}}(I_\sigma(\Lambda[\![\texttt{id}]\!]))$, is the complement of each singleton. Hence, for the three set of input states, we obtain the values 2, 2, and 2, respectively. Thus, $\text{QUSED}_{\{\mathtt{x}\}}(\Lambda[\![\texttt{id}]\!]) = 2$, meaning that the input variable x is used.

The trace semantics of Program 4.3 is:

$$\Lambda[\![\texttt{random}]\!] = \left\{ \begin{array}{l} (l_0, \langle 0 \rangle) \to \cdots \to (l_3, \langle * \rangle), \\ (l_0, \langle 1 \rangle) \to \cdots \to (l_3, \langle * \rangle), \\ (l_0, \langle 2 \rangle) \to \cdots \to (l_3, \langle * \rangle) \end{array} \right\}$$

In this case, any outcome can be produced by any input value, thus the set of input states is $\{(l_0, \langle 0 \rangle), (l_0, \langle 1 \rangle), (l_0, \langle 2 \rangle)\}$ for all outcomes. Since all possible permutations of the variable x are already in each set of input states from outcomes, we obtain the value 0. Thus, $\text{QUSED}_{\{\mathtt{x}\}}(\Lambda[\![\texttt{random}]\!]) = 0$, meaning that the input variable x is not used.

Prog. 4.2: The identity program.

```
1 id(x):
2   return x;
3
```

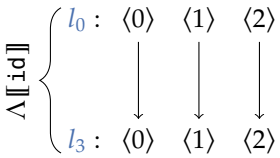


Figure 4.6: Graphical representation of the trace semantics of Program 4.2.

Prog. 4.3: The random program.

```
1 random(x):
2   return rand();
3
```

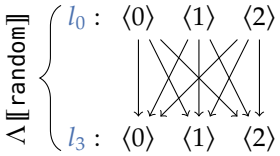


Figure 4.7: Graphical representation of the trace semantics of Program 4.3.

Prog. 4.4: First example combining random and constant value.

```
1 mix1(x):
2   if x > 0:
3     return rand();
4   else:
5     return 0;
6
```

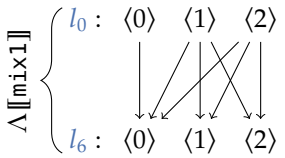


Figure 4.8: Graphical representation of the trace semantics of Program 4.4.

Prog. 4.5: Second example combining random and constant value.

```
1 mix2(x, y):
2   if (not x) and y:
3     return rand()
4   return y
5
```

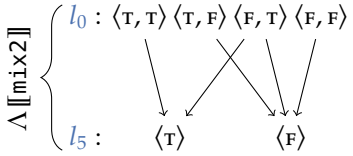


Figure 4.9: Graphical representation of the trace semantics of Program 4.5.

Def. 3.5.2 (Output-Abstraction Semantics)

$$\Lambda^{\rho} \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \rho(\sigma_\omega)\rangle \mid \sigma \in \Lambda\}\}$$

Def. 3.4.1 (Right-Left Adjoints for the Dependency Semantics)

$$\alpha^{\rightsquigarrow} \in \wp(\wp(\Sigma^{+\infty})) \to \wp(\wp(\Sigma \times \Sigma^{\perp}))$$
$$\alpha^{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \sigma_\omega\rangle \mid \sigma \in T\} \mid T \in S\}$$

Def. 3.5.1 (Right-Left Adjoints for the Output-Abstraction Semantics)

$$\alpha^{\rho} \in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma \times \Sigma^{\perp}))$$
$$\alpha^{\rho}(W) \stackrel{\text{def}}{=} \left\{ \{\langle s_0, \rho(s_\omega)\rangle \mid \langle s_0, s_\omega\rangle \in D\} \;\middle|\; D \in W \right\}$$

The trace semantics of Program 4.4 is:

$$\Lambda[\![\texttt{mix1}]\!] = \left\{ \begin{array}{l} (l_0, \langle 0\rangle) \to \cdots \to (l_6, \langle 0\rangle), \\ (l_0, \langle 1\rangle) \to \cdots \to (l_6, \langle *\rangle), \\ (l_0, \langle 2\rangle) \to \cdots \to (l_6, \langle *\rangle) \end{array} \right\}$$

In this case, the set of input states that can produce the outcome 0 is $\{(l_0, \langle 0\rangle), (l_0, \langle 1\rangle), (l_0, \langle 2\rangle)\}$, for the outcome 1 is $\{(l_0, \langle 1\rangle), (l_0, \langle 2\rangle)\}$, and for the outcome 2 is $\{(l_0, \langle 1\rangle), (l_0, \langle 2\rangle)\}$. Respectively, the input state $(l_0, \langle 0\rangle)$ is missing from both sets of input states that can produce the outcomes 1 and 2. Thus, we yield the highest value of missing states, obtaining $\text{QUsed}_{\{x\}}(\Lambda[\![\texttt{mix1}]\!]) = 1$.

For this last program we introduce another variable y, we simplify even further the variable to boolean values $\{\top, \bot\}$. The trace semantics of Program 4.5 is:

$$\Lambda[\![\texttt{mix2}]\!] = \left\{ \begin{array}{l} (l_0, \langle \top, \top\rangle) \to \cdots \to (l_5, \langle \top\rangle), \\ (l_0, \langle \top, \bot\rangle) \to \cdots \to (l_5, \langle \bot\rangle), \\ (l_0, \langle \bot, \top\rangle) \to \cdots \to (l_5, \langle \top\rangle), \\ (l_0, \langle \bot, \top\rangle) \to \cdots \to (l_5, \langle \bot\rangle), \\ (l_0, \langle \bot, \bot\rangle) \to \cdots \to (l_5, \langle \bot\rangle) \end{array} \right\}$$

In this case, the set of input states that can produce the outcome $\top$ is $\{(l_0, \langle \top, \top\rangle), (l_0, \langle \bot, \top\rangle)\}$, for the other outcome $\bot$ the set of input states is $\{(l_0, \langle \top, \bot\rangle), (l_0, \langle \bot, \top\rangle), (l_0, \langle \bot, \bot\rangle)\}$. In case $W = \{x\}$, the state $(l_0, \langle \top, \top\rangle)$ is missing from the set of input states that can produce the outcome $\bot$, obtained by permuting the variable x on the input state $(l_0, \langle \bot, \top\rangle)$. Instead, the set of states $\{(l_0, \langle \top, \top\rangle), (l_0, \langle \bot, \top\rangle)\}$ already contains all the possible permutations of the variable x. Thus, we yield the highest value of missing states, obtaining $\text{QUsed}_{\{x\}}(\Lambda[\![\texttt{mix2}]\!]) = 1$. On the other hand, if $W = \{y\}$, the states $(l_0, \langle \bot, \bot\rangle)$ and $(l_0, \langle \top, \bot\rangle)$ are missing from the set of input states that can produce the outcome $\top$. No state is missing from the set of input states that can produce the outcome $\bot$. Thus, we yield the highest value of missing states, obtaining $\text{QUsed}_{\{y\}}(\Lambda[\![\texttt{mix2}]\!]) = 2$.

From these examples we notice that the QUsed impact quantifier is able to capture that a variable is not used in a program, even in the presence of non-deterministic statements, cf. Program 4.3. Moreover, QUsed also discriminates between different amount of impact. Indeed, Program 4.2 is, intuitively, the program where the input variable has the highest impact possible as the output is exactly its value, while Program 4.3 the lowest as the output is random.

Next, we show the validation of the k-bounded impact property when instantiated with the QUsed impact quantifier.

Lemma 4.1.10 ($\mathcal{B}^{\otimes k}_{\text{QUsed}_W}$ Validation)

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{B}^{\otimes k}_{\text{QUsed}_W} \iff \Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{B}^{\otimes k}_{\text{QUsed}_W}))$$

Proof (Sketch). Let us consider the $\alpha^{\rho} \circ \alpha^{\rightsquigarrow}$ abstraction of the k-bounded impact property $\mathscr{B}^{\circledast k}_{\text{QUSED}_W}$:

$$\alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathscr{B}^{\circledast k}_{\text{QUSED}_W})) = \{D \in \wp(\Sigma \times \Sigma^{\perp}) \mid \sup_{\langle s_0, s_\omega\rangle \in D} |Q_W(I_{s_\omega}(D)) \setminus I_{s_\omega}(D)| \circledast k\}$$

As noticed before, the QUSED impact quantifier does not consider the intermediate states and abstracts the output states before any computation. Hence, neither the abstraction to dependencies nor the abstraction to output values affect the validation of the property. □

$$I_{s_\omega}(D) = \{s'_0 \mid \langle s'_0, s'_\omega\rangle \in D \wedge \rho(s_\omega) = \rho(s'_\omega)\}$$
$$Q_W(S) = \{s' \in \Sigma \mid s \in S \wedge s' =_{\Delta\setminus W} s\}$$

The next example shows that QUSED is neither monotonic nor anti-monotonic in the amount of traces, meaning that, by adding or removing traces from a given set $T \in \wp(\Sigma^{+\infty})$ the quantity $\text{QUSED}_W(T)$ may increase or decrease. Notably, the major consequence is that $\mathscr{B}^{\circledast k}_{\text{QUSED}_W}$ is neither subset- nor superset-closed. Thus, not all the subsets, respectively supersets, of a potential over-approximation, respectively under-approximation, of the trace semantics satisfy the $\mathscr{B}^{\circledast k}_{\text{QUSED}_W}$ property.

Example 4.1.7 In this example, we show that the QUSED impact quantifier is neither monotonic nor anti-monotonic in the amount of traces. Specifically, we consider two input variables with boolean values $\{\text{T}, \text{F}\}$, the Program 4.6 returns the second component of the input without any modification.

Prog. 4.6: Identity function on the second component.

```
id2(x, y):
  return y
```

The trace semantics of Program 4.6 is:

$$\Lambda[\![\texttt{id2}]\!] = \left\{ \begin{array}{l} (l_0, \langle \text{T}, \text{T}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{T}\rangle), \\ (l_0, \langle \text{T}, \text{F}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F}\rangle), \\ (l_0, \langle \text{F}, \text{T}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{T}\rangle), \\ (l_0, \langle \text{F}, \text{F}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F}\rangle) \end{array} \right\}$$

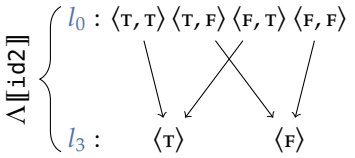


Figure 4.10: Graphical representation of the trace semantics of Program 4.6.

For the set of traces $\Lambda[\![\texttt{id2}]\!]$, we have that $\text{QUSED}_{\{\text{x}\}}(\Lambda[\![\texttt{id2}]\!]) = 0$ as no input state is missing from both the sets that can produce the outcomes T and F. In fact, the variable x is *not used* in the program. Let us consider the bigger set of traces $T \supseteq \Lambda[\![\texttt{id2}]\!]$:

$$T = \left\{ \begin{array}{l} (l_0, \langle \text{T}, \text{T}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{T}\rangle), \\ (l_0, \langle \text{T}, \text{F}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F}\rangle), \\ (l_0, \langle \text{F}, \text{T}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{T}\rangle), \\ (l_0, \langle \text{F}, \text{T}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F}\rangle), \\ (l_0, \langle \text{F}, \text{F}\rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F}\rangle) \end{array} \right\}$$

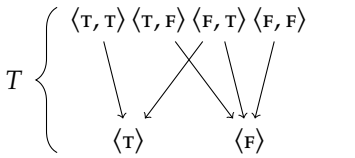


Figure 4.11: Graphical representation of T.

This set of traces could be obtained by the Program 4.5. In this case, $\text{QUSED}_{\{\text{x}\}}(T) = 1$ as the input value $(l_0, \langle \text{T}, \text{T}\rangle)$ is missing from the set of input values that can produce the outcome F, cf., the set $\{(l_0, \langle \text{T}, \text{F}\rangle), (l_0, \langle \text{F}, \text{T}\rangle), (l_0, \langle \text{F}, \text{F}\rangle)\}$. Note that, in this case, the variable x is *used* in the program. Lastly, let us enlarge again the set of traces to

Prog. 4.5 (Second example combining random and constant value.)**:**

```
mix2(x, y):
  if (not x) and y:
    return rand()
  return y
```

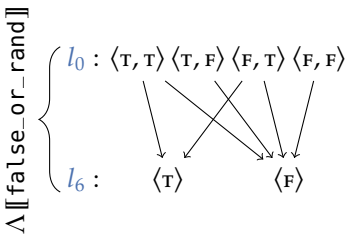


Figure 4.12: Graphical representation of the trace semantics of Program 4.7.

Prog. 4.7: False or random program.

```
false_or_rand(x, y):
  if not y:
    return False
  else:
    return rand()
```

Def. 3.2.3 (Abstract Unused)

$$\text{AUnused}_{\mathtt{w}}(\Lambda[\![P]\!]) \overset{def}{\Leftrightarrow} \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{w}|}.\ \sigma_0(\mathtt{w}) \neq v \Rightarrow \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta \setminus \mathtt{w}} \sigma_0 \wedge \sigma'_0(\mathtt{w}) = v \wedge \rho(\sigma_\omega) = \rho(\sigma'_\omega)$$

Def. 4.1.4 (QUsed)

$$\text{QUsed}_{\mathtt{w}}(T) \overset{\text{def}}{=} \sup_{\sigma \in T} |\, Q_{\mathtt{w}}(I_\sigma(T)) \setminus I_\sigma(T) \,|$$

$$I_\sigma(T) = \{\sigma'_0 \mid \sigma' \in T \wedge \rho(\sigma_\omega) = \rho(\sigma'_\omega)\}$$

$$Q_{\mathtt{w}}(S) = \{s' \in \Sigma \mid s \in S \wedge s' =_{\Delta \setminus \mathtt{w}} s\}$$

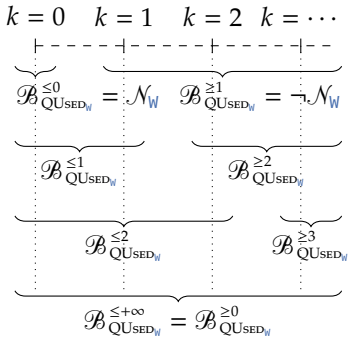


Figure 4.13: Graphical comparison between the unused $\mathcal{N}_{\mathtt{w}}$ and the k-bounded impact property $\mathcal{B}^{\otimes k}_{\text{QUsed}_{\mathtt{w}}}$.

Def. 3.3.1 (Unused Property)

$$\mathcal{N}_{\mathtt{w}} \overset{\text{def}}{=} \{\Lambda \in \wp(\Sigma^{+\infty}) \mid \text{AUnused}_{\mathtt{w}}(\Lambda)\}$$

$T' \supseteq T$:

$$T' = \left\{ \begin{array}{l} (l_0, \langle \text{T}, \text{T} \rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{T} \rangle), \\ (l_0, \langle \text{T}, \text{T} \rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F} \rangle), \\ (l_0, \langle \text{T}, \text{F} \rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F} \rangle), \\ (l_0, \langle \text{F}, \text{T} \rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{T} \rangle), \\ (l_0, \langle \text{F}, \text{T} \rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F} \rangle), \\ (l_0, \langle \text{F}, \text{F} \rangle) \longrightarrow \cdots \longrightarrow (l_3, \langle \text{F} \rangle) \end{array} \right\}$$

In this case, $\text{QUsed}_{\{\mathtt{x}\}}(T') = 0$ as the input $(l_0, \langle \text{T}, \text{T} \rangle)$ is not missing anymore in the set of input values that can produce the outcomes F. For instance, a program that may generate such a set of traces is the Program 4.7.

This example shows that

$$\text{QUsed}_{\{\mathtt{x}\}}(\Lambda[\![\texttt{id2}]\!]) \leq \text{QUsed}_{\{\mathtt{x}\}}(T) \geq \text{QUsed}_{\{\mathtt{x}\}}(T')$$

where $\Lambda[\![\texttt{id2}]\!] \subseteq T \subseteq T'$. Thus, the QUsed impact quantifier is neither monotonic nor anti-monotonic in the amount of traces, as the quantifier QUsed may increase or decrease by adding or removing traces from a given set.

Unlike for the previous impact quantifiers, it holds that, whenever $\text{QUsed}_{\mathtt{w}}(\Lambda[\![P]\!])$ is 0, the program P does not use the input variable $\mathtt{w}$, even in the presence of non-determinism. Thus, we can establish the following equivalence:

Lemma 4.1.11 (Unused Equivalence) *For all program P, it holds that:*

$$\text{AUnused}_{\mathtt{w}}(\Lambda[\![P]\!]) \Leftrightarrow \text{QUsed}_{\mathtt{w}}(\Lambda[\![P]\!]) = 0$$

Proof. To show ($\Rightarrow$) we assume that $\text{AUnused}_{\mathtt{w}}(\Lambda[\![P]\!])$. As a consequence, we have that any outcome either is not reachable or is reachable from every input value of the variables $\mathtt{w}$. Thus, for any $\sigma \in T$, the set of input values that can produce that outcome is the set of all input values, *i.e.*, $I_\sigma(T) = \mathbb{V}$. Since we remove from $Q_{\mathtt{w}}(I_\sigma(T))$ the set of input values, we obtain a value of 0 for all traces, hence $\text{QUsed}_{\mathtt{w}}(\Lambda[\![P]\!]) = 0$.

To show ($\Leftarrow$) we assume that $\text{QUsed}_{\mathtt{w}}(\Lambda) = 0$. By definition, for all s_ω, the set of input values that can produce the outcome s_ω is the set of all input values, otherwise $\text{QUsed}_{\mathtt{w}}(\Lambda)$ would not be 0. Thus, for all traces $\sigma \in \Lambda[\![P]\!]$ and input values $v \in \mathbb{V}$ such that $\sigma_0(\mathtt{w}) \neq v$, we have that it exists a trace σ' such that $\rho(\sigma'_\omega) = \rho(\sigma_\omega)$ and $\sigma'_0(\mathtt{w}) = v$, otherwise it would not be possible that the set of input values that can produce any outcome is the set of all input values. Hence, $\text{AUnused}_{\mathtt{w}}(\Lambda[\![P]\!])$. □

As a consequence of Lemma 4.1.11, we obtain an equivalence between the (un)used property (cf. Definition 3.3.1) and the k-bounded impact property (cf. Definition 4.1.1) instantiated with the QUsed quantifier:

$$\mathcal{B}^{\leq 0}_{\text{QUsed}_{\mathtt{w}}} = \mathcal{N}_{\mathtt{w}} \qquad \text{if } \otimes = \leq$$

$$\mathcal{B}^{\geq 1}_{\text{QUsed}_{\mathtt{w}}} = \neg\mathcal{N}_{\mathtt{w}} \qquad \text{if } \otimes = \geq$$

meaning that if a program semantics does not utilize the variables $\mathtt{W}$, then it has an impact of *at most* 0 (and vice versa). Conversely, if a program semantics utilizes the variables $\mathtt{W}$, then it has an impact of *at least* 1 (and vice versa). Figure 4.13 depicts an overview of the relationship between the unused property $\mathcal{N}_{\mathtt{W}}$ and the various k-bounded impact properties $\mathcal{B}^{\otimes k}_{\text{QUsed}_{\mathtt{W}}}$ instantiated with different thresholds and comparison operators.

The k-bounded impact property instantiated with the QUsed quantifier:

$$\mathcal{B}^{\otimes k}_{\text{QUsed}_{\mathtt{W}}} \stackrel{\text{def}}{=} \{\Lambda \mid \text{QUsed}_{\mathtt{W}}(\Lambda) \otimes k\}$$

In the rest of this chapter, we present the static analysis to verify the k-bounded impact property of a program and show the abstract version of the quantifiers introduced in this chapter.

4.2 Abstract Quantitative Input Data Usage

In this section, we introduce a sound and computable static analysis to verify the k-bounded impact property of a program. The soundness of the approach leverages two elements: (1) a backward abstract semantics, and (2) sound and computable implementations of the quantifiers Outcomes, Range, and QUsed (written as $\mathtt{Outcomes}^\sharp$, $\mathtt{Range}^\sharp$, and $\mathtt{QUsed}^\sharp$ respectively).

To quantify the usage of the input variables $\mathtt{W}$, we need to determine the input configurations leading to specific output values. As our impact quantifiers measure over the reachable output values (more precisely, over their abstraction by ρ) our underlying abstract semantics will be a *backward* (co-)reachability semantics starting from *disjoint* abstract post-conditions, over-approximating the (concrete) output values of the dependency semantics. Specifically, we abstract the concrete output values with an indexed set $B \in \mathbb{D}^n$ of n disjoint *output buckets*, where $\langle \mathbb{D}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is an abstract state domain with concretization function $\gamma \in \mathbb{D} \to \wp(\Sigma^\bot)$. The choice of these output buckets is a parameter of the analysis and a *good* choice is essential for obtaining a precise and meaningful analysis result.

An indexed set $F \in X^n$ where $n \in \mathbb{N}$, is a map from the set of indices $N_{\leq n}$ to the set of elements X.

3: A family of semantics is a set of program semantics parametrized by an initialization.

For each output bucket $B_j \in \mathbb{D}$, where $1 \leq j \leq n$, our analysis computes an over-approximation of the dependency semantics restricted to the input configurations leading to $\gamma(B_j)$. Our static analysis is parametrized by an underlying backward abstract family[3] of semantics $\Lambda^\leftarrow \in \mathbb{D} \to \mathbb{D}$ which computes the backward semantics $\Lambda^\leftarrow(B_j)$ from a given output bucket $B_j \in \mathbb{D}$. For instance, we could use the backward semantics of Definition 2.3.18 to compute $\Lambda^\leftarrow(B_j)$. The concretization function $\gamma^\leftarrow \in (\mathbb{D} \to \mathbb{D}) \to \mathbb{D} \to \wp(\wp(\Sigma \times \Sigma^\bot))$ employs the concretization of the abstract domain γ to restore all possible dependencies from input states to output values.

Def. 2.3.18 (Backward Co-Reachability State Abstract Semantics)

$$\Lambda^\leftarrow[\![\mathtt{skip}]\!]d^\sharp \stackrel{\text{def}}{=} d^\sharp$$
$$\Lambda^\leftarrow[\![\mathtt{x} := e]\!]d^\sharp \stackrel{\text{def}}{=} \mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^\sharp$$
$$\Lambda^\leftarrow[\![\mathtt{assert}\ b]\!]d^\sharp \stackrel{\text{def}}{=} \mathtt{Filter}[\![b]\!]d^\sharp$$
$$\Lambda^\leftarrow[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]d^\sharp \stackrel{\text{def}}{=} \mathtt{Filter}[\![b]\!](\Lambda^\leftarrow[\![st]\!]d^\sharp) \sqcup \mathtt{Filter}[\![\neg b]\!](\Lambda^\leftarrow[\![st']\!]d^\sharp)$$
$$\Lambda^\leftarrow[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]d^\sharp \stackrel{\text{def}}{=} \operatorname{lim} \mathrm{F}^{\mathbb{D}}$$
$$\mathrm{F}^{\mathbb{D}}(a^\sharp) \stackrel{\text{def}}{=} \begin{pmatrix} \mathtt{Filter}[\![\neg b]\!](d^\sharp) \sqcup \\ \mathtt{Filter}[\![b]\!](\Lambda^\leftarrow[\![st]\!](a^\sharp)) \end{pmatrix}$$
$$\Lambda^\leftarrow[\![st; st']\!]d^\sharp \stackrel{\text{def}}{=} \Lambda^\leftarrow[\![st]\!](\Lambda^\leftarrow[\![st']\!]d^\sharp)$$

Definition 4.2.1 (Concretization $\gamma^\leftarrow$) *Given a backward semantics $\Lambda^\leftarrow \in \mathbb{D} \to \mathbb{D}$ and n output buckets B, the* concretization $\gamma^\leftarrow$ *is defined as:*

$$\gamma^\leftarrow \in (\mathbb{D} \to \mathbb{D}) \to \mathbb{D} \to \wp(\wp(\Sigma \times \Sigma^\bot))$$
$$\gamma^\leftarrow(\Lambda^\leftarrow)B_j \stackrel{\text{def}}{=} \wp(\{\langle s_0, s_\omega \rangle \mid s_0 \in \gamma(\Lambda^\leftarrow(B_j)) \wedge s_\omega \in \gamma(B_j)\})$$

In words, $\gamma^\leftarrow$ collects all the input-output dependencies from the application of the backward semantics to each output bucket B_j. Since the

abstract semantics may discover more dependencies than the concrete ones, the concretization returns all possible subsets of dependencies. In such a way, we ensure that the (restriction of the) output-abstraction semantics is a subset of the abstract one. We can thus define the soundness condition for the backward semantics with respect to the reduction of the output-abstraction semantics.

Def. 3.5.3 (Output Reduction)

$$W|_X \stackrel{\text{def}}{=} \left\{ \{\langle s_0, \rho(s_\omega)\rangle \in D \mid s_\omega \in X\} \;\middle|\; D \in W \right\}$$

Definition 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$) *For all programs P, and output bucket $B_j \in \mathbb{D}$, the family of semantics $\Lambda^{\leftarrow}$ is a* sound over-approximation *of the output-abstraction semantics Λ^{ρ} reduced with $\gamma(B_j)$, when it holds that:*

$$\Lambda^{\rho}[\![P]\!]|_{\gamma(B_j)} \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}[\![P]\!])B_j$$

We define $\Lambda^{\times} \in \mathbb{D}^n \to \mathbb{D}^n$ as the backward semantics repeated on a set of output buckets $B \in \mathbb{D}^n$ as follows:

Definition 4.2.3 (Multi-Bucket Semantics $\Lambda^{\times}$) *We define $\Lambda^{\times} \in \mathbb{D}^n \to \mathbb{D}^n$ as the backward semantics repeated on a set of output buckets $B \in \mathbb{D}^n$, that is:*

$$\Lambda^{\times}[\![P]\!]B \stackrel{\text{def}}{=} (\Lambda^{\leftarrow}[\![P]\!]B_j)_{j \le n}$$

The concretization function $\gamma^{\times}$ maps the multi-bucket semantics $\Lambda^{\times}$ from a given set of output buckets B to a set of sets of dependencies in $\wp(\wp(\Sigma \times \Sigma^{\perp}))$. Specifically, for each output bucket $B_j \in B$, $\gamma^{\times}$ concretizes the output from $\gamma(B_j)$ and the input states from the application of the backward semantics to B_j, *i.e.*, $\gamma((\Lambda^{\times}[\![P]\!]B)_j)$. Hence, all the possible input-output dependencies are collected for all buckets.

Definition 4.2.4 (Multi-Bucket Concretization $\gamma^{\times}$) *We define the concretization function $\gamma^{\times} \in (\mathbb{D}^n \to \mathbb{D}^n) \to \mathbb{D}^n \to \wp(\wp(\Sigma \times \Sigma^{\perp}))$ as:*

$$\gamma^{\times}(\Lambda^{\times}[\![P]\!])B \stackrel{\text{def}}{=} \bigcup_{j \le n} \gamma^{\leftarrow}(\Lambda^{\leftarrow}[\![P]\!])B_j$$

Next, we show the soundness of the multi-bucket semantics with respect to the output-abstraction semantics.

Lemma 4.2.1 (Sound Over-Approximation for $\Lambda^{\times}$) *For all programs P, output buckets $B \in \mathbb{D}^n$, and a family of semantics $\Lambda^{\leftarrow}$, the semantics $\Lambda^{\times}$ is a* sound over-approximation *of the output-abstraction semantics Λ^{ρ} when reduced to $\bigcup_{j \le n} \gamma(B_j)$:*

$$\Lambda^{\rho}[\![P]\!]|_{\bigcup_{j \le n} \gamma(B_j)} \subseteq \gamma^{\times}(\Lambda^{\times}[\![P]\!])B$$

Proof. From Definition 4.2.3 we have that $\gamma^{\times}(\Lambda^{\times})B = \bigcup_{j \le n} \gamma^{\leftarrow}(\Lambda^{\leftarrow}(B_j))$. Then, from Definition 4.2.2, we obtain that $\forall j \le n.\, \Lambda^{\rho}|_{\gamma(B_j)} \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}(B_j))$. Thus, by monotonicity of the union operator over set inclusion, it holds

that $\bigcup_{j\leq n} \Lambda^\rho|_{\gamma(B_j)} \subseteq \bigcup_{j\leq n} \gamma^{\leftarrow}(\Lambda^{\leftarrow}(B_j))$. We conclude by:

$$\begin{aligned}
\bigcup_{j\leq n} \Lambda^\rho|_{\gamma(B_j)} &= \bigcup_{j\leq n} \{\langle s_0, s_\omega\rangle \in \Lambda^\rho \mid s_\omega \in \gamma(B_j)\} && \text{by } \Lambda^\rho|_X \\
&= \{\langle s_0, s_\omega\rangle \in \Lambda^\rho \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\} && \text{by set definition} \\
&= \Lambda^\rho|_{\bigcup_{j\leq n} \gamma(B_j)} && \text{by } \Lambda^\rho|_X
\end{aligned}$$

□

We introduce the concept of *covering* for output buckets to ensure that no potential final state is missed from the analysis.

Definition 4.2.5 (Covering) *Let $\rho \in \Sigma^\perp \longrightarrow \Sigma^\perp$ be the output observer, we say that the output buckets $B \in \mathbb{D}^n$ cover the subset of potential outcomes whenever it holds that:*

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \;\subseteq\; \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$

Whenever the output buckets *cover* the subset of outcomes abstracted by ρ, $\Lambda^\times$ is a sound over-approximation of Λ^ρ.

Lemma 4.2.2 *Let $B \in \mathbb{D}^n$ be the set of output buckets such that it covers the subset of potential outcomes of a program P.Then, it holds that the semantics $\Lambda^\times$ is a* sound over-approximation *of the output-abstraction semantics Λ^ρ:*

$$\Lambda^\rho[\![P]\!] \;\subseteq\; \gamma^\times(\Lambda^\times[\![P]\!])B$$

Proof. Follows directly from Lemma 4.2.1 and Definition 4.2.5. The covering of the output buckets ensures that no potential final state is missed and thus let the reduction of the output-abstraction semantics unnecessary, as it holds that $\Lambda^\rho[\![P]\!] = \Lambda^\rho[\![P]\!]|_{\bigcup_{j\leq n} \gamma(B_j)}$. □

So far, we presented the semantics $\Lambda^\times$ as an abstraction of the output-abstraction semantics Λ^ρ via Lemma 4.2.2. Next, this section shows the concept of sound implementation of the impact quantifiers. We expect a sound implementation $\texttt{Impact}^\natural_{\mathrm{W}} \in \mathbb{D}^n \times \mathbb{D}^n \longrightarrow \mathbb{V}^{\pm\infty}$ to return a bound on the impact which is always higher (or always lower depending on the ordering operator $\circledcirc$ of $\mathcal{B}^{\circledcirc k}_{\textsc{Impact}_{\mathrm{W}}}$) than the concrete counterpart $\textsc{Impact}_{\mathrm{W}}$.

Definition 4.2.6 (Sound Implementation) *For all output buckets B and family of semantics $\Lambda^\times$, whenever,*

(i) $\Lambda^{\leftarrow}$ is sound with respect to Λ^ρ, cf. Definition 4.2.2, and
(ii) B covers the subset of potential outcomes, cf. Definition 4.2.5,

we say that $\texttt{Impact}^\natural_{\mathrm{W}}$ is a sound implementation *of $\textsc{Impact}_{\mathrm{W}}$ if and only if:*

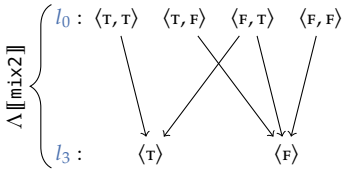
$$\textsc{Impact}_{\mathrm{W}}(\Lambda[\![P]\!]) \;\circledcirc\; \texttt{Impact}^\natural_{\mathrm{W}}(\Lambda^\times[\![P]\!]B, B)$$

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^\rho|_{\gamma(B_j)} \;\subseteq\; \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$



Prog. 4.5 (Second example combining random and constant value.)**:**

```
mix2(x, y):
  if (not x) and y:
    return rand()
  return y
```

Definition 4.2.6 ensures that the soundness of the static analysis is preserved when computing the quantity k. Then, we define $\gamma^{\text{Impact}_W} \in \mathbb{V}^{\pm\infty} \to \wp(\wp(\Sigma^{+\infty}))$, *i.e.*, the concretization of the bound measured by the abstract impact $\texttt{Impact}^{\natural}_W(\Lambda^\times[\![P]\!]B, B)$. The idea behind the concretization γ^{Impact_W} is to filter the set of sets of traces concretized by $\gamma^\times$ that yield an impact lower, respectively higher, than the bound computed in the abstract, cf. $\texttt{Impact}^{\natural}_W(\Lambda^\times[\![P]\!]B, B)$. Before defining the concretization, we show an example where not all subsets of $\gamma^\times$ satisfy the property $\mathcal{B}^{\otimes k}_{\text{Impact}_W}$.

Example 4.2.1 We assume we are interested in the validation of the property $\mathcal{B}^{\geq k}_{\text{QUsed}_W}$, where the impact quantifier is QUsed and the comparison operator for quantity bounds is $\geq$. We consider the trace semantics of the program Program 4.5, *i.e.*, $\Lambda[\![\texttt{mix2}]\!]$, depicted on the side. Consider an abstract semantics $\Lambda^\times[\![\texttt{mix2}]\!]$ that concretize to the following sets of dependencies:

$$\gamma^\times(\Lambda^\times[\![\texttt{mix2}]\!])B \;=\; \wp\left(\left\{\begin{array}{l}\langle(l_0,\langle \text{T},\text{T}\rangle),(l_3,\langle \text{T}\rangle)\rangle,\\ \langle(l_0,\langle \text{T},\text{F}\rangle),(l_3,\langle \text{F}\rangle)\rangle,\\ \langle(l_0,\langle \text{F},\text{T}\rangle),(l_3,\langle *\rangle)\rangle,\\ \langle(l_0,\langle \text{F},\text{F}\rangle),(l_3,\langle \text{F}\rangle)\rangle\end{array}\right\}\right)$$

As noted in Example 4.1.7, the $\text{QUsed}_{\{\text{x}\}}$ applied to the biggest set of the concretized dependencies above is 1. If we assume the abstract impact returns the bound 1, *i.e.*, $\texttt{Impact}^{\natural}_W(\Lambda^\times[\![\texttt{mix2}]\!], B) = 1$, then γ^{Impact_W} should filter from $\gamma^\times(\Lambda^\times[\![P]\!])B$ the sets of dependencies that yield an impact lower than the bound 1 as the validation of the property $\mathcal{B}^{\geq k}_{\text{QUsed}_W}$ employs the $\geq$ operator in this example. In particular, the following set of dependencies is removed:

$$T \;=\; \left\{\begin{array}{l}\langle(l_0,\langle \text{T},\text{T}\rangle),(l_3,\langle \text{T}\rangle)\rangle,\\ \langle(l_0,\langle \text{T},\text{F}\rangle),(l_3,\langle \text{F}\rangle)\rangle,\\ \langle(l_0,\langle \text{F},\text{T}\rangle),(l_3,\langle \text{T}\rangle)\rangle,\\ \langle(l_0,\langle \text{F},\text{F}\rangle),(l_3,\langle \text{F}\rangle)\rangle\end{array}\right\}$$

as $\text{QUsed}_{\{\text{x}\}}(T)$ is 0 in such case (cf. Example 4.1.7).

Def. 3.4.1 (Right-Left Adjoints for the Dependency Semantics)

$$\gamma^{\rightsquigarrow} \in \wp(\wp(\Sigma \times \Sigma^\perp)) \to \wp(\wp(\Sigma^{+\infty}))$$
$$\gamma^{\rightsquigarrow}(W) \stackrel{\text{def}}{=} \left\{ T \in \wp(\Sigma^{+\infty}) \;\middle|\; \{\langle\sigma_0, \sigma_\omega\rangle \mid \sigma \in T\} \in W \right\}$$

Def. 3.5.1 (Right-Left Adjoints for the Output-Abstraction Semantics)

$$\gamma^\rho \in \wp(\wp(\Sigma \times \Sigma^\perp)) \to \wp(\wp(\Sigma \times \Sigma^\perp))$$
$$\gamma^\rho(W) \stackrel{\text{def}}{=} \left\{ \bigcup \left\{ D' \subseteq S_{\langle s_0, s_\omega\rangle} \;\middle|\; \langle s_0, s_\omega\rangle \in D \right\} \;\middle|\; D \in W \right\}$$

where

$$S_{\langle s_0, s_\omega\rangle} = \{\langle s_0, s'_\omega\rangle \mid s_\omega = \rho(s'_\omega)\}$$

Definition 4.2.7 (Concretization of the Measured Quantity) *The concretization of a measured quantity k is $\gamma^{\text{Impact}_W} \in \mathbb{V}^{\pm\infty} \to \wp(\wp(\Sigma^{+\infty}))$, defined as:*

$$\gamma^{\text{Impact}_W}(\texttt{Impact}^{\natural}_W(\Lambda^\times[\![P]\!]B, B)) \stackrel{\text{def}}{=} \{T \in \gamma^{\rightsquigarrow} \circ \gamma^\rho \circ \gamma^\times(\Lambda^\times[\![P]\!])B \mid \text{Impact}_W(T) \otimes \texttt{Impact}^{\natural}_W(\Lambda^\times[\![P]\!]B, B)\}$$

The concretization γ^{Impact_W} maintains the set of traces concretized by $\gamma^\times$, γ^ρ (cf. Definition 3.4.1), and $\gamma^{\rightsquigarrow}$ (cf. Definition 3.5.1) that yield an impact lower, respectively higher, than the bound computed in the abstract, cf., $\texttt{Impact}^{\natural}_W(\Lambda^\times[\![P]\!]B, B)$.

Remark 4.2.1 Based on the impact quantifier Impact, the k-bounded impact property $\mathcal{B}^{\otimes k}_{\text{Impact}_W}$ can be subset-closed, meaning that whenever

a set of traces satisfies the property, all its subsets also satisfy the property. In this case, it would hold that

$$\gamma^{\times}(\Lambda^{\times}[\![P]\!])B \;\subseteq\; \gamma^{\textsc{Impact}_W}(\texttt{Impact}^{\sharp}_W(\Lambda^{\times}[\![P]\!]B, B))$$

However, this is not always the case. Indeed, some subsets of traces may not satisfy the property $\mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$.

The concretization $\gamma^{\textsc{Impact}_W}$ is used to prove that the abstract semantics $\Lambda^{\times}$ is sound for proving the property $\mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$.

Lemma 4.2.3 *A program P has an impact of at most k (respectively, at least k depending on the comparison operator $\otimes$) only if the set of dependencies concretized from the quantity $\textsc{Impact}_W(\Lambda[\![P]\!])$ are a subset of the semantics in $\mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$.*

$$\gamma^{\textsc{Impact}_W}(\textsc{Impact}_W(\Lambda[\![P]\!])) \;\subseteq\; \mathcal{B}^{\otimes k}_{\textsc{Impact}_W} \;\Rightarrow\; P \models \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$$

Proof. By hypothesis, we assume that $\gamma^{\rightsquigarrow} \circ \gamma^{\rho} \circ \gamma^{\textsc{Impact}_W}(k') \subseteq \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$ where $k' = \textsc{Impact}_W(\Lambda[\![P]\!])$. We have that $\Lambda^{\rho}[\![P]\!] \subseteq \gamma^{\textsc{Impact}_W}(k')$ by definition of $\gamma^{\textsc{Impact}_W}$, cf. Definition 4.2.7. Then, by monotonicity of γ^{ρ} and $\gamma^{\rightsquigarrow}$, cf. Definition 3.5.1 and Definition 3.4.1 respectively, we have that $\gamma^{\rightsquigarrow}(\gamma^{\rho}(\Lambda^{\rho}[\![P]\!])) \subseteq \gamma^{\rightsquigarrow}(\gamma^{\textsc{Impact}_W}(k'))$. Thus, since $\gamma^{\rightsquigarrow} \circ \gamma^{\rho} \circ \gamma^{\textsc{Impact}_W}(k') \subseteq \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$, we have that $\gamma^{\rightsquigarrow}(\gamma^{\rho}(\Lambda^{\rho}[\![P]\!])) \subseteq \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$. Depending on the instantiation of the impact quantifier, we conclude the proof by Lemma 4.1.3 for the Outcomes impact quantifier, Lemma 4.1.7 for the Range impact quantifier, and Lemma 4.1.10 for the QUsed impact quantifier. □

Finally, the next result shows that our static analysis is sound when employed to verify the property of interest $\mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$ for the program P. That is, if $\texttt{Impact}^{\sharp}_W$ returns the bound k', and $k' \otimes k$, then the program P satisfies the property $\mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$, cf. $P \models \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$.

Theorem 4.2.4 (Soundness) *Let $\mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$ be the property of interest we want to verify for the program P and the input variable $W \in \wp(\Delta)$. Whenever,*

(i) $\Lambda^{\leftarrow}$ is sound with respect to Λ^{ρ}, cf. Definition 4.2.2,
(ii) B covers the subset of potential outcomes, cf. Definition 4.2.5, and
(iii) $\texttt{Impact}^{\sharp}_W$ is a sound implementation of $\textsc{Impact}_W$, cf. Definition 4.2.6,

the following implication holds:

$$\texttt{Impact}^{\sharp}_W(\Lambda^{\times}[\![P]\!]B, B) = k' \wedge k' \otimes k \;\Rightarrow\; P \models \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$$

Proof. From (i), (ii), and (iii), we have that:

$$\textsc{Impact}_W(\Lambda[\![P]\!]) \otimes \texttt{Impact}^{\sharp}_W(\Lambda^{\times}[\![P]\!]B, B) \otimes k'$$

From the hypothesis $k' \otimes k$, we obtain $\textsc{Impact}_W(\Lambda[\![P]\!]) \otimes k$. By Definition 4.2.7, we have that $\gamma^{\textsc{Impact}_W}(\textsc{Impact}_W(\Lambda[\![P]\!])) \subseteq \mathcal{B}^{\otimes k}_{\textsc{Impact}_W}$. We conclude by Lemma 4.2.3. □

Def. 4.2.7 (Concretization of the Measured Quantity)

$$\gamma^{\textsc{Impact}_W}(k) \stackrel{\text{def}}{=} \left\{ D \in \gamma^{\times}(\Lambda^{\times}[\![P]\!])B \;\middle|\; \textsc{Impact}_W(T) \otimes k \right\}$$

Def. 4.1.1 (k-Bounded Impact Property)

$$\mathcal{B}^{\otimes k}_{\textsc{Impact}_W} \stackrel{\text{def}}{=} \{\Lambda \mid \textsc{Impact}_W(\Lambda) \otimes k\}$$

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^{\rho}|_{\gamma(B_j)} \;\subseteq\; \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \le n} \gamma(B_j)\}$$

Def. 4.2.6 (Sound Implementation)

$$\textsc{Impact}_W(\Lambda[\![P]\!]) \otimes \texttt{Impact}^{\sharp}_W(\Lambda^{\times}[\![P]\!]B, B)$$

Next, we define $\texttt{Range}^\natural$, $\texttt{Outcomes}^\natural$, and $\texttt{QUsed}^\natural$ as possible implementations for Range, Outcomes, and QUsed, respectively.

We assume the underlying abstract state domain $\mathbb{D}$ is equipped with an operator $\texttt{Proj}_W \in \mathbb{D} \to \mathbb{D}$ to project away the input variables W, that is, the reduction in dimensionality of the abstract element by projecting its image to the reduced variable space.

Example 4.2.2 In the context of the interval domain, where each input variable is related to a possibly unbounded lower and upper bound, $\texttt{Proj}_{\{i\}}(\langle i \mapsto [1,3], j \mapsto [2,4]\rangle) = \langle i \mapsto [-\infty,+\infty], j \mapsto [2,4]\rangle$ removes the constraints related to the variables in $\{i\}$. Often, whenever a variable maps to $\top$, or equivalently in the interval domain to $[-\infty,+\infty]$, we hide it from the abstract element, *e.g.*, instead of $\langle i \mapsto [-\infty,+\infty], j \mapsto [2,4]\rangle$ we often write $\langle j \mapsto [2,4]\rangle$.

We define the condition a sound projection operator $\texttt{Proj}_W$ must satisfy to ensure that no concrete state is missed by the projection.

The projection operator is sound whenever the projection of the variables W from the abstract value $s^\natural$ contains all the concrete states that can be obtained by perturbing the variables W from a concrete state represented by $s^\natural$. In fact, we could define the *concrete* project operator $\textsc{Proj} \in \wp(\Sigma) \to \wp(\Sigma)$ as it follows:

$$\textsc{Proj}(S) \stackrel{\text{def}}{=} \{s' \mid s \in S \wedge s =_W s'\}$$

It follows the classic definition of soundness.

Definition 4.2.8 (Sound Projection Operator) *For all abstract states $s^\natural$, the projection of the abstract state $\texttt{Proj}_W(s^\natural)$ is sound whenever it holds that:*

$$\textsc{Proj}(\gamma(s^\natural)) \subseteq \texttt{Proj}_W(s^\natural)$$

4.2.1 Abstract Implementation $\texttt{Outcomes}^\natural_W$

The definition of $\texttt{Outcomes}^\natural_W$ first projects away the input variables W from all the given abstract values ($\texttt{Proj}_W$ in the definition below), then it collects the set of indices of abstract values resulting from the meet operation over any two resulting abstract domain elements ($\texttt{InterAll}$ in the definition below). These intersections represent concrete input configurations where variations on the values of W *may* lead to changes of program outcome, from a bucket to another. We return the maximum number of abstract values that intersect after projections:

Definition 4.2.9 ($\texttt{Outcomes}^\natural_W$) *Let $W \in \wp(\Delta)$ be the input variables, $n \in \mathbb{N}$ be the number of output buckets, and $B \in \mathbb{D}^n$ the set of output buckets. We define $\texttt{Outcomes}^\natural_W \in \mathbb{D}^n \times \mathbb{D}^n \to \mathbb{V}^{\pm\infty}$ as:*

$$\texttt{Outcomes}^\natural_W(X, B) \stackrel{\text{def}}{=} \max\{|J| \mid J \in \texttt{InterAll}((\texttt{Proj}_W(X_j))_{j \le n})\}$$

where $X \in \mathbb{D}^n$ is the result of the abstract semantics $\Lambda^\times$ and $|J|$ is the cardinality of the set of indices J.

Note the use of max instead of sup as in the concrete counterpart (Definition 4.1.2) since the number of intersecting abstract values is finite and bounded by n, cf. the number of output buckets. The function `InterAll` takes as input an indexed set of abstract values and returns the set of indices of abstract values that intersect together, defined as follows:

Definition 4.2.10 (`InterAll`) *Given a vector of n abstract elements $X \in \mathbb{D}^n$, we define* $\texttt{InterAll} \in \mathbb{D}^n \to \wp(\mathbb{N})$ *as:*

$$\texttt{InterAll}(X) \stackrel{\text{def}}{=} \{J \subseteq \mathbb{N} \mid \forall j \leq n, p \leq n.\ j \in J \land p \in J \ \land\ X_j \sqcap X_p\}$$

Finding all the indices of intersecting abstract values is equivalent to find cliques in a graph, where each node represents an abstract value and an edge exists between two nodes if and only if the corresponding abstract values intersect. Therefore, `InterAll` can be efficiently implemented based on the graph algorithm by Bron and Kerbosch [76]. All together, the abstract implementation $\texttt{Outcomes}^\sharp_w$ returns the value of the maximum among all the sizes of the maximum cliques for each projection.

Def. 4.1.2 (Outcomes)

$$\textsc{Outcomes}_w(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \left| \left\{ \rho(\sigma_\omega) \;\middle|\; \begin{array}{l} \sigma \in T \land \\ \sigma_0 =_{\Delta \setminus \{i\}} s_0 \end{array} \right\} \right|$$

[76]: Bron et al. (1973), 'Finding All Cliques of an Undirected Graph (Algorithm 457)'

Prog. 4.1 (Landing alarm system):

```
1  landing_coeff =
2    abs(angle) + speed
3  if landing_coeff < 2:
4    risk = 0
5  else if landing_coeff > 5:
6    risk = 3
7  else:
8    risk =
9     floor(landing_coeff) - 2
10
```

Example 4.2.3 Consider Program 4.1 with program states defined as $\Sigma = \{\langle a, b, c, d\rangle \mid a \in \{-4, 1\} \land b \in \{1, 2, 3\} \land c \in \mathbb{N} \land d \in \mathbb{N}_{\leq 3}\}$. The output buckets represent the four levels of landing risk: low, medium-low, medium-high, and high. Using the interval domain: $B \in \mathbb{D}^4$ such that $B_{\text{low}} = \langle \texttt{risk} \mapsto 0\rangle$, $B_{\text{mid-low}} = \langle \texttt{risk} \mapsto 1\rangle$, $B_{\text{mid-high}} = \langle \texttt{risk} \mapsto 2\rangle$, and $B_{\text{high}} = \langle \texttt{risk} \mapsto 3\rangle$. Definition 4.2.5 (Covering) is satisfied as the output buckets cover all the possible outcomes of the program for the variable `risk`. We assume the backward analysis to return the following intervals:

$$\begin{aligned}
\Lambda^{\leftarrow}(B_{\text{low}}) &= X_{\text{low}} = \langle \texttt{angle} \mapsto 1, \texttt{speed} \mapsto [1, 3]\rangle \\
\Lambda^{\leftarrow}(B_{\text{mid-low}}) &= X_{\text{mid-low}} = \langle \texttt{angle} \mapsto 1, \texttt{speed} \mapsto [2, 3]\rangle \\
\Lambda^{\leftarrow}(B_{\text{mid-high}}) &= X_{\text{mid-high}} = \langle \texttt{angle} \mapsto 1, \texttt{speed} \mapsto [2, 3]\rangle \\
\Lambda^{\leftarrow}(B_{\text{high}}) &= X_{\text{high}} = \langle \texttt{angle} \mapsto -4, \texttt{speed} \mapsto [1, 3]\rangle
\end{aligned}$$

Note that, the backward analysis over-approximates the concrete semantics as, for example, the input configuration `angle` = 1 and `speed` = 3 does not lead to a low risk of approach. We are interested in the input variable `angle`, thus the projections are:

$$\begin{aligned}
\texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{low}}) &= \langle \texttt{speed} \mapsto [1, 3]\rangle \\
\texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{mid-low}}) &= \langle \texttt{speed} \mapsto [2, 3]\rangle \\
\texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{mid-high}}) &= \langle \texttt{speed} \mapsto [2, 3]\rangle \\
\texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{high}}) &= \langle \texttt{speed} \mapsto [1, 3]\rangle
\end{aligned}$$

It is easy to see that the abstract elements resulting from the projection intersect between each other. Therefore, we obtain that the set of indices of buckets that intersect is:

$$\begin{aligned}&\texttt{InterAll}((\texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{low}}), \ldots, \texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{high}}))) = \\ &\quad \wp(\{\text{low}, \text{mid-low}, \text{mid-high}, \text{high}\})\end{aligned}$$

The maximum size is 4 at $|\{\text{low}, \text{mid-low}, \text{mid-high}, \text{high}\}|$. Therefore, $\texttt{Outcomes}^{\natural}_{\{\texttt{angle}\}}(X, B) = 4$, meaning that variations of the input variable `angle` may lead to changes of program outcome from four different buckets. On the other hand, projecting away the variable `speed` from the abstract intervals results in:

$$\begin{aligned}\texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{low}}) &= \langle \texttt{angle} \mapsto 1 \rangle \\ \texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{mid-low}}) &= \langle \texttt{angle} \mapsto 1 \rangle \\ \texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{mid-high}}) &= \langle \texttt{angle} \mapsto 1 \rangle \\ \texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{high}}) &= \langle \texttt{angle} \mapsto -4 \rangle\end{aligned}$$

In this case, only a few abstract elements intersect, specifically:

$$\begin{aligned}&\texttt{InterAll}((\texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{low}}), \ldots, \texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{high}}))) = \\ &\left\{ \begin{array}{l} \{\text{low}\}, \{\text{mid-low}\}, \{\text{mid-high}\}, \{\text{high}\}, \\ \{\text{low}, \text{mid-low}\}, \{\text{low}, \text{mid-high}\}, \{\text{mid-low}, \text{mid-high}\} \\ \{\text{low}, \text{mid-low}, \text{mid-high}\} \end{array} \right\}\end{aligned}$$

The maximum clique is of size 3, cf. $|\{\text{low}, \text{mid-low}, \text{mid-high}\}|$, hence $\texttt{Outcomes}^{\natural}_{\{\texttt{speed}\}}(X, B) = 3$. This means that variations of the input variable `speed` may lead to changes of program outcome in three different buckets.

In order to prove that the abstract impact $\texttt{Outcomes}^{\natural}$ is a sound over-approximation of the concrete impact Outcomes, we require the output buckets B to be *compatible* with the output observer ρ. That is, any two output states s_ω, s'_ω that produce different output readings, *i.e.*, $\rho(s_\omega) \neq \rho(s'_\omega)$, belong to different buckets, *i.e.*, $s_\omega \in \gamma(B_j)$, $s'_\omega \in \gamma(B_p)$, and $B_j \neq B_p$. Intuitively, compatibility ensures that the counting intersecting buckets in the abstract does not miss any concrete outcome.

Definition 4.2.11 (Compatibility) *Given the output buckets $B \in \mathbb{D}^n$ and the output observer $\rho \in \Sigma^\perp \to \Sigma^\perp$, we say that B is* compatible *with ρ, whenever it holds:*

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p).\ \rho(s_j) \neq \rho(s_p) \implies B_j \neq B_p$$

Note that, $\textsc{Outcomes}_{\mathsf{W}}$ is bounded by the number of buckets when the conditions of covering and compatibility hold for the output buckets.

Lemma 4.2.5 ($\textsc{Outcomes}_{\mathsf{W}}$ Upper Bound) *When the n output buckets $B \in \mathbb{D}^n$ are compatible, cf. Definition 4.2.11, and cover the subset of potential outcomes, cf. Definition 4.2.5, it holds that $\textsc{Outcomes}_{\mathsf{W}}(\Lambda) \leq n$.*

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \leq n} \gamma(B_j)\}$$

Proof. We notice that $\textsc{Outcomes}_{\mathtt{W}}(\Lambda) \leq |\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\}|$ as the set of outputs for the trace semantics is always bigger than any set of outputs. It is easy to note that the cardinality of $\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\}$ is bounded from above by n since any two output states s_ω, s'_ω that produce different outputs, *i.e.*, $\rho(s_\omega) \neq \rho(s'_\omega)$, belong to different buckets, *i.e.*, $s_\omega \in \gamma(B_j) \wedge s'_\omega \in \gamma(B_p) \wedge B_j \neq B_p$ (by compatibility, cf. Definition 4.2.11). Where the existence of the two buckets is guaranteed by covering (cf. Definition 4.2.5). Therefore, there are at most n different outputs. □

We designed $\mathtt{Outcomes}^{\natural}$ to over-approximates the concrete quantifier $\textsc{Outcomes}$. Hence, the next result shows that the abstract impact $\mathtt{Outcomes}^{\natural}_{\mathtt{W}}$ is a sound over-approximation of the concrete $\textsc{Outcomes}_{\mathtt{W}}$ w.r.t. the $\leq$ operator.

Lemma 4.2.6 ($\mathtt{Outcomes}^{\natural}_{\mathtt{W}}$ is a Sound Implementation of $\textsc{Outcomes}_{\mathtt{W}}$) *Let $\mathtt{W} \in \wp(\Delta)$ be the input variable of interest, $\mathbb{D}$ the abstract domain, $\Lambda^{\leftarrow}$ the family of semantics, and $B \in \mathbb{D}^n$ the starting output buckets. Whenever the following conditions hold:*

(i) B covers the subset of potential outcomes, cf. Definition 4.2.5,
(ii) B is compatible with ρ, cf. Definition 4.2.11, and
(iii) $\mathtt{Proj}$ is sound, cf. Definition 4.2.8;

then, $\mathtt{Outcomes}^{\natural}_{\mathtt{W}}$ is a sound implementation of $\textsc{Outcomes}_{\mathtt{W}}$ w.r.t. the $\leq$ operator:

$$\textsc{Outcomes}_{\mathtt{W}}(\Lambda) \;\leq\; \mathtt{Outcomes}^{\natural}_{\mathtt{W}}(\gamma^{\times}(\Lambda^{\times})B, B)$$

Proof (Sketch). We need to prove that

$$\textsc{Outcomes}_{\mathtt{W}}(\Lambda) = \sup_{s_0 \in \Sigma|_{\Delta}} |\{\rho(\sigma_\omega) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} s_0\}| \leq \mathtt{Outcomes}^{\natural}_{\mathtt{W}}(\gamma^{\times}(\Lambda^{\times})B, B) = \max\{|J| \mid J \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(\gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j))_{j\leq n})\}$$

Let $\overline{s_0} \in \Sigma|_{\Delta}$ be such that:

$$\sup_{s_0 \in \Sigma|_{\Delta}} |\{\rho(\sigma_\omega) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} s_0\}| = |\{\rho(\sigma_\omega) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} \overline{s_0}\}|$$

Let $\overline{J} \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(\gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j))_{j\leq n})$ be such that:

$$\max\{|J| \mid J \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(\gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j))_{j\leq n})\} = |\overline{J}|$$

We need to show that $|\{\rho(\sigma_\omega) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} \overline{s_0}\}| \leq |\overline{J}|$. By contradiction, we assume that $|\{\rho(\sigma_\omega) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus\mathtt{W}} \overline{s_0}\}| > |\overline{J}|$, meaning that the number of distinct output abstraction from states in Λ from perturbation of the input s_0 is higher than the number of indices in the abstract. Hence, either:

(a) there exist output abstractions that are not matched by any output bucket,

Def. 4.1.2 ($\textsc{Outcomes}$)

$$\textsc{Outcomes}_{\mathtt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_{\Delta}} \left|\left\{\rho(\sigma_\omega) \;\middle|\; \begin{array}{l}\sigma \in T \wedge \\ \sigma_0 =_{\Delta\setminus\{\mathtt{i}\}} s_0\end{array}\right\}\right|$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$

Def. 4.2.11 (Compatibility)

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p). \; \rho(s_j) \neq \rho(s_p) \;\Rightarrow\; B_j \neq B_p$$

Def. 4.2.8 (Sound Projection Operator)

$$\{s' \in \Sigma^{\perp} \mid s \in \gamma(s^{\natural}) \wedge s =_{\Delta\setminus\mathtt{W}} s'\} \subseteq \gamma(\mathtt{Proj}_{\mathtt{W}}(s^{\natural}))$$

(b) some output abstractions are matched by the same output bucket, or
(c) there are output abstractions in Λ from perturbation of the input s_0 that are not considered in the abstract.

By (ii), the first case (a) is not possible as covering, cf. Definition 4.2.5, ensures that all the output abstractions are matched by the output buckets. By (i), the second case (b) is not possible as compatibility, cf. Definition 4.2.11, ensures that any two output abstractions that produce different outputs belong to different output buckets. The third case (c) is not possible by (iii) as the soundness of the projection operator ensures that all the concrete states result of perturbations on the variable `W` from a state represented by an abstract value $s^\natural$ are considered in the abstract. Therefore, the assumption is false, and we conclude that $\texttt{Outcomes}^\natural_{\texttt{W}}$ is a sound implementation of $\textsc{Outcomes}_{\texttt{W}}$. □

Prog. 4.1 (Landing alarm system):

```
landing_coeff =
  abs(angle) + speed
if landing_coeff < 2:
  risk = 0
else if landing_coeff > 5:
  risk = 3
else:
  risk =
   floor(landing_coeff) - 2
```

$$\rho(s) \stackrel{\text{def}}{=} \begin{cases} \langle \top, \top, \top, d \rangle \\ \quad \text{if } s = \langle a, b, c, d \rangle \\ \langle \top, \top, \top, \top \rangle \\ \quad \text{otherwise} \end{cases}$$

$$\begin{aligned} B_{\text{low}} &= \langle \texttt{risk} \mapsto 0 \rangle \\ B_{\text{mid-low}} &= \langle \texttt{risk} \mapsto 1 \rangle \\ B_{\text{mid-high}} &= \langle \texttt{risk} \mapsto 2 \rangle \\ B_{\text{high}} &= \langle \texttt{risk} \mapsto 3 \rangle \end{aligned}$$

Example 4.2.4 Regarding the Program 4.1, we show that the result of the abstract implementation $\texttt{Outcomes}^\natural_{\texttt{W}}$ in Example 4.2.3 is a sound over-approximation of the concrete implementation $\textsc{Outcomes}_{\texttt{W}}$. We assume the output observer ρ is defined as in Example 4.1.1, where the fourth component of the state represents the value of the output variable `risk`. Indeed, the output buckets of Example 4.2.3 are compatible with the output observer ρ as any two output states are mapped to different output buckets. As expected by Theorem 4.2.4, the bound computed by the abstract implementation $\texttt{Outcomes}^\natural_{\texttt{W}}$ is always greater or equal to the concrete one:

$$\begin{aligned} &\textsc{Outcomes}_{\{\texttt{angle}\}}(\Lambda[\![L]\!]) = 2 \\ &\quad \leq \texttt{Outcomes}^\natural_{\{\texttt{angle}\}}(X, B) = 4 \end{aligned}$$

$$\begin{aligned} &\textsc{Outcomes}_{\{\texttt{speed}\}}(\Lambda[\![L]\!]) = 3 \\ &\quad \leq \texttt{Outcomes}^\natural_{\{\texttt{speed}\}}(X, B) = 3 \end{aligned}$$

where L is the program of the landing alarm system, cf. Program 4.1.

Thm. 4.2.4 (Soundness)

$$\texttt{Impact}^\natural_{\texttt{W}}(\Lambda^\times[\![P]\!]B, B) \otimes k \implies P \models \mathscr{B}^{\otimes k}_{\textsc{Impact}_{\texttt{W}}}$$

Next, we show Theorem 4.2.4 (Soundness) instantiated with the abstract implementation $\texttt{Outcomes}^\natural_{\texttt{W}}$ and the comparison operator $\leq$.

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^\rho|_{\gamma(B_j)} \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \leq n} \gamma(B_j)\}$$

Theorem 4.2.7 (Soundness of $\mathscr{B}^{\leq k}_{\textsc{Outcomes}_{\texttt{W}}}$) *Let $\mathscr{B}^{\leq k}_{\textsc{Outcomes}_{\texttt{W}}}$ be the property of interest we want to verify for the program P and the input variable $\texttt{W} \in \wp(\Delta)$. Whenever,*

(i) $\Lambda^{\leftarrow}$ is sound with respect to Λ^ρ, cf. Definition 4.2.2, and
(ii) B covers the subset of potential outcomes, cf. Definition 4.2.5,

the following implication holds:

$$\texttt{Outcomes}^\natural_{\texttt{W}}(\Lambda^\times[\![P]\!]B, B) = k' \wedge k' \leq k \implies P \models \mathscr{B}^{\leq k}_{\textsc{Outcomes}_{\texttt{W}}}$$

Proof. Lemma 4.2.6 shows that $\texttt{Outcomes}^\natural_{\texttt{W}}$ is a sound implementation of $\textsc{Outcomes}_{\texttt{W}}$, the proof follows directly by application of the Theorem 4.2.4

instantiated with the abstract implementation $\texttt{Outcomes}^\sharp_W$ and comparison operator $\leq$. □

4.2.2 Abstract Implementation $\texttt{Range}^\sharp_W$

We define $\texttt{Range}^\sharp_W$ as the maximum distance of the range of the extreme values of the buckets represented by intersecting abstract values after projections. In such case, we assume $\mathbb{D}$ is equipped with an additional abstract operator $\texttt{Distance} \in \mathbb{D} \to \mathbb{V}^{+\infty}_{\geq 0}$, which returns the size of the given abstract element, otherwise $+\infty$ if the abstract element is unbounded or represents multiple variables.

Example 4.2.5 In the context of the interval domain, where each input variable is related to a (possibly infinite) lower and upper bound, $\texttt{Distance}(\langle \texttt{x} \mapsto [2,4] \rangle) = 2$. On the other hand, whenever the input abstract element is unbounded, the size is $+\infty$, *e.g.*, $\texttt{Distance}(\langle \texttt{x} \mapsto [0,+\infty] \rangle) = +\infty$.

Let W be the set of input variables of interest and Z be the set of output variables. The abstract range $\texttt{Range}^\sharp_W$ first projects away the input variables W from all the given abstract values (cf. $\texttt{Proj}_W$). Then, it collects the indices of all possible intersections among the projected abstract values (cf. `InterAll`). These intersections represent concrete input configurations where variations on the values of W *may* lead to changes of program outcome, from a bucket to another. All the buckets of corresponding intersections are joined together. Thus, all but the output variables (cf. $\texttt{Proj}_{\Delta \setminus Z}$) are projected away to find the maximum range of the extreme values of the buckets.

Definition 4.2.12 ($\texttt{Range}^\sharp_W$) *Let $W \in \wp(\Delta)$ be the input variables, $n \in \mathbb{N}$ be the number of output buckets, and $B \in \mathbb{D}^n$ the set of output buckets. Let Z be the output variables of interest. We define $\texttt{Range}^\sharp_W \in \mathbb{D}^n \times \mathbb{D}^n \to \mathbb{V}^{+\infty}_{\geq 0}$ as:*

$$\begin{aligned} \texttt{Range}^\sharp_W(X, B) &\stackrel{\text{def}}{=} \max \{\texttt{Distance}(\texttt{Proj}_{\Delta \setminus Z}(K)) \mid K \in I\} \\ \text{where } I &= \{\sqcup\{B_j \mid j \in J\} \mid J \in \texttt{InterAll}((\texttt{Proj}_W(X_j))_{j \leq n})\} \end{aligned}$$

where $X \in \mathbb{D}^n$ is the result of the abstract semantics $\Lambda^\times$.

Def. 4.1.3 (Range)

$$\textsc{Range}_W(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}\left(\left\{\rho(\sigma_\omega)(Z) \;\middle|\; \begin{matrix} \sigma \in T \wedge \\ \sigma_0 =_{\Delta \setminus W} s_0 \end{matrix}\right\}\right)$$

Example 4.2.6 Similarly to Example 4.2.3, we show the computation of $\texttt{Range}^\sharp_W$ for the program Program 4.1. The computation of $\texttt{Range}^\sharp_W$ works as for $\texttt{Outcomes}^\sharp_W$ until the computation of the intersections. Hence, we obtain the same projections for the variable `angle`:

$$\begin{aligned} &\texttt{InterAll}((\texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{low}}), \ldots, \texttt{Proj}_{\{\texttt{angle}\}}(X_{\text{high}}))) = \\ &\quad \wp(\{\text{low}, \text{mid-low}, \text{mid-high}, \text{high}\}) \end{aligned}$$

Next, $\texttt{Range}^\sharp_{\{\texttt{angle}\}}$ computes the size of the output buckets joined together for any combination of bucket indices. The maximum is achieved whenever both the low and high buckets, respectively buckets B_{low} and B_{high}, are joined together, as they yield the minimum and

Prog. 4.1 (Landing alarm system):

```
1  landing_coeff =
2    abs(angle) + speed
3  if landing_coeff < 2:
4    risk = 0
5  else if landing_coeff > 5:
6    risk = 3
7  else:
8    risk =
9     floor(landing_coeff) - 2
10
```

maximum value for the variable `risk`. The size of bucket B_{low} joined with B_{high} is 3, *i.e.*, $\texttt{Distance}(B_{\text{low}} \sqcup B_{\text{high}}) = \texttt{Distance}(\langle \texttt{risk} \mapsto [0,3] \rangle) = 3$. Therefore, $\texttt{Range}^\sharp_{\{\texttt{angle}\}}(X, B) = 3$. On the other hand, projecting away the variable `speed` from the abstract intervals results in:

$$\texttt{InterAll}((\texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{low}}), \ldots, \texttt{Proj}_{\{\texttt{speed}\}}(X_{\text{high}}))) = \left\{ \begin{array}{l} \{\text{low}\}, \{\text{mid-low}\}, \{\text{mid-high}\}, \{\text{high}\}, \\ \{\text{low}, \text{mid-low}\}, \{\text{low}, \text{mid-high}\}, \{\text{mid-low}, \text{mid-high}\} \\ \{\text{low}, \text{mid-low}, \text{mid-high}\} \end{array} \right\}$$

In this case, we obtain the following sizes for the combination of buckets:

$$\begin{aligned}
\texttt{Distance}(B_{\text{low}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto 0 \rangle) = 0 \\
\texttt{Distance}(B_{\text{mid-low}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto 1 \rangle) = 0 \\
\texttt{Distance}(B_{\text{mid-high}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto 2 \rangle) = 0 \\
\texttt{Distance}(B_{\text{high}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto 3 \rangle) = 0 \\
\texttt{Distance}(B_{\text{low}} \sqcup B_{\text{mid-low}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto [0,1] \rangle) \\
&= 1 \\
\texttt{Distance}(B_{\text{low}} \sqcup B_{\text{mid-high}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto [0,2] \rangle) \\
&= 2 \\
\texttt{Distance}(B_{\text{mid-low}} \sqcup B_{\text{mid-high}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto [1,2] \rangle) \\
&= 1 \\
\texttt{Distance}(B_{\text{low}} \sqcup B_{\text{mid-low}} \sqcup B_{\text{mid-high}}) &= \texttt{Distance}(\langle \texttt{risk} \mapsto [0,2] \rangle) \\
&= 2
\end{aligned}$$

Therefore, $\texttt{Range}^\sharp_{\{\texttt{speed}\}}(X, B) = 2$.

To prove that $\texttt{Range}^\sharp_{\texttt{W}}$ is a sound implementation of $\textsc{Range}_{\texttt{W}}$, we require the classic soundness condition on the abstract operator `Distance`, ensuring that the abstract distance is always greater than the concrete one.

Definition 4.2.13 (Soundness of `Distance`) *Given an abstract value $s^\sharp \in \mathbb{D}$ and the output variables of interest $\texttt{Z}$, it holds that:*

$$\textsc{Distance}(\{\rho(s)(\texttt{Z}) \mid s \in \gamma(s^\sharp)\}) \leq \texttt{Distance}(s^\sharp)$$

We designed $\texttt{Range}^\sharp$ to over-approximates the concrete quantifier Range. Hence, the next result shows that the abstract impact $\texttt{Range}^\sharp$ is a sound over-approximation of the concrete impact Range, cf. Definition 4.1.3, w.r.t. the $\leq$ operator.

Lemma 4.2.8 ($\texttt{Range}^\sharp_{\texttt{W}}$ is a Sound Implementation of $\textsc{Range}_{\texttt{W}}$) *Let $\texttt{W} \in \wp(\Delta)$ be the input variable of interest, $\mathbb{D}$ the abstract domain, $\Lambda^{\leftarrow}$ the family of semantics, and $B \in \mathbb{D}^n$ the starting output buckets. Whenever the following conditions hold:*

(i) B covers the subset of potential outcomes, cf. Definition 4.2.5,
(ii) B is compatible with ρ, cf. Definition 4.2.11,

Def. 4.1.3 (Range)

$$\textsc{Range}_{\texttt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}\left(\left\{\rho(\sigma_\omega)(\texttt{Z}) \;\middle|\; \begin{array}{l} \sigma \in T \wedge \\ \sigma_0 =_{\Delta \setminus \texttt{W}} s_0 \end{array}\right\}\right)$$

Def. 4.1.3 (Range)

$$\textsc{Range}_{\texttt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}\left(\left\{\rho(\sigma_\omega)(\texttt{Z}) \;\middle|\; \begin{array}{l} \sigma \in T \wedge \\ \sigma_0 =_{\Delta \setminus \texttt{W}} s_0 \end{array}\right\}\right)$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \leq n} \gamma(B_j)\}$$

Def. 4.2.11 (Compatibility)

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p). \; \rho(s_j) \neq \rho(s_p) \Rightarrow B_j \neq B_p$$

(iii) `Proj` *is sound, cf. Definition 4.2.8, and*
(iv) `Distance` *is sound, cf. Definition 4.2.13;*

then, $\texttt{Range}^\sharp_W$ *is a sound implementation of* $\textsc{Range}_W$ *w.r.t. the* $\leq$ *operator:*

$$\textsc{Range}_W(\Lambda) \;\leq\; \texttt{Range}^\sharp_W(\gamma^\times(\Lambda^\times)B, B)$$

Proof (Sketch). We need to prove that

$$\textsc{Range}_W(\Lambda) = \sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}(\{\rho(\sigma_\omega)(\mathtt{z}) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus W} s_0\}) \;\leq$$
$$\texttt{Range}^\sharp_W(\gamma^\times(\Lambda^\times)B, B) = \max\{\texttt{Distance}(\texttt{Proj}_{\Delta\setminus \mathtt{z}}(K)) \mid K \in I\}$$

where $I = \{\sqcup\{B_j \mid j \in J\} \mid J \in \texttt{InterAll}((\texttt{Proj}_W(\gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j))_{j\leq n})\}$. Let $\overline{s_0} \in \Sigma|_\Delta$ be such that:

$$\sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}(\{\rho(\sigma_\omega)(\mathtt{z}) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus W} s_0\}) \;=$$
$$\textsc{Distance}(\{\rho(\sigma_\omega)(\mathtt{z}) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus W} \overline{s_0}\})$$

Let $\overline{K} \in I$ be such that:

$$\max\{\texttt{Distance}(K) \mid K \in I\} \;=\; \texttt{Distance}(\overline{K})$$

where $\overline{K} = \sqcup\{B_j \mid j \in J\}$ and $J \in \texttt{InterAll}((\texttt{Proj}_W(\gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j))_{j\leq n})$. We need to show that $\textsc{Distance}(\{\rho(\sigma_\omega)(\mathtt{z}) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus W} \overline{s_0}\}) \leq \texttt{Distance}(\overline{K})$. By contradiction, we assume that $\textsc{Distance}(\{\rho(\sigma_\omega)(\mathtt{z}) \mid \sigma \in \Lambda \wedge \sigma_0 =_{\Delta\setminus W} \overline{s_0}\}) > \texttt{Distance}(\overline{K})$, meaning that the size of output abstraction over the variable `z` from states in Λ is higher than the size of $\overline{K}$ in the abstract. Hence, either:

(a) there exist output abstractions that are not matched by any output bucket,
(b) some output abstractions are matched by the same output bucket,
(c) there are output abstractions in Λ from perturbation of the input s_0 that are not considered in the abstract, or
(d) the size of the abstract element is not greater than the size of the output abstraction.

Similarly to the proof of Lemma 4.2.6, we can show that the first three cases are not possible by (i), (ii), and (iii). The fourth case (d) is not possible by (iv) as the soundness of the size operator ensures that the size of the abstract element is always greater than the size of the output abstraction. Therefore, the assumption is false, and we conclude that $\texttt{Range}^\sharp_W$ is a sound implementation of $\textsc{Range}_W$. □

Def. 4.2.8 (Sound Projection Operator)

$$\{s' \in \Sigma^\perp \mid s \in \gamma(s^\sharp) \wedge s =_{\Delta\setminus W} s'\} \subseteq \gamma(\texttt{Proj}_W(s^\sharp))$$

Def. 4.2.13 (Soundness of `Distance`)

$$\textsc{Distance}(\{\rho(s) \mid s \in \gamma(s^\sharp)\}) \leq \texttt{Distance}(s^\sharp)$$

Example 4.2.7 Let `risk` be the output variable of interest. Regarding the program Program 4.1, we show that the quantities computed by the abstract implementation $\texttt{Range}^\sharp_W$ in Example 4.2.6 are sound over-approximations of the concrete implementation $\textsc{Range}_W$.

$$\textsc{Range}_{\{\mathtt{angle}\}}(\Lambda[\![\text{landing-alarm-system}]\!]) = 3$$
$$\leq\; \texttt{Range}^\sharp_{\{\mathtt{angle}\}}(\gamma^\times(\Lambda^\times)B, B) = 3$$

Prog. 4.1 (Landing alarm system):

```
1  landing_coeff =
2    abs(angle) + speed
3  if landing_coeff < 2:
4    risk = 0
5  else if landing_coeff > 5:
6    risk = 3
7  else:
8    risk =
9     floor(landing_coeff) - 2
10
```

Thm. 4.2.4 (Soundness)

$$\mathtt{Impact}^\sharp_{\mathtt{W}}(\Lambda^\times[\![P]\!]B, B) \otimes k \;\Rightarrow\; P \models \mathscr{B}^{\otimes k}_{\textsc{Impact}_{\mathtt{W}}}$$

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^\rho|_{\gamma(B_j)} \;\subseteq\; \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$

$$\textsc{Range}_{\{\mathtt{speed}\}}(\Lambda[\![\text{landing-alarm-system}]\!]) = 2 \;\leq\; \mathtt{Range}^\sharp_{\{\mathtt{speed}\}}(X, B) = 2$$

as expected by Theorem 4.2.4.

Next, we show Theorem 4.2.4 (Soundness) instantiated with the abstract implementation $\mathtt{Range}^\sharp_{\mathtt{W}}$ and the comparison operator $\leq$.

Theorem 4.2.9 (Soundness of $\mathscr{B}^{\leq k}_{\textsc{Range}_{\mathtt{W}}}$) *Let $\mathscr{B}^{\leq k}_{\textsc{Range}_{\mathtt{W}}}$ be the property of interest we want to verify for the program P and the input variable $\mathtt{W} \in \wp(\Delta)$. Whenever,*

(i) $\Lambda^{\leftarrow}$ is sound with respect to Λ^ρ, cf. Definition 4.2.2, and
(ii) B covers the subset of potential outcomes, cf. Definition 4.2.5,

the following implication holds:

$$\mathtt{Range}^\sharp_{\mathtt{W}}(\Lambda^\times[\![P]\!]B, B) = k' \wedge k' \leq k \;\Rightarrow\; P \models \mathscr{B}^{\leq k}_{\textsc{Range}_{\mathtt{W}}}$$

Proof. Lemma 4.2.8 shows that $\mathtt{Range}^\sharp_{\mathtt{W}}$ is a sound implementation of $\textsc{Range}_{\mathtt{W}}$, the proof follows directly by application of the Theorem 4.2.4 instantiated with the abstract implementation $\mathtt{Range}^\sharp_{\mathtt{W}}$ and comparison operator $\leq$. □

4.2.3 Abstract Implementation $\mathtt{QUsed}^\sharp_{\mathtt{W}}$

We define $\mathtt{QUsed}^\sharp_{\mathtt{W}}$ as the number of input values that are missing from perturbation to the input variables $\mathtt{W}$ to reach the same outcomes. We assume the abstract domain $\mathbb{D}$ is equipped with an abstract counting operator $\mathtt{Count} \in \mathbb{D} \to \mathbb{V}^{+\infty}_{\geq 0}$ that returns the number of possible values of the given abstract element, *e.g.*, $\mathtt{Count}(\langle \mathtt{x} \mapsto \{2,3,4\}\rangle) = 3$. As well as an abstract complement operator $\mathtt{Compl}_{d^\sharp} \in \mathbb{D} \to \mathbb{D}$ that returns the complement of the given abstract element w.r.t. $d^\sharp$, which we assume to represent the whole space. For example, $\mathtt{Compl}_{\langle \mathtt{x}\mapsto\{0,1,2,3\}\rangle}(\langle \mathtt{x} \mapsto \{2,3\}\rangle) = \langle \mathtt{x} \mapsto \{0,1\}\rangle$. In general, we use top ($\top$) for the abstract element representing the whole space. This operator is useful to count missing states as those can be obtained as result from the complement of the states we do have.

Specifically, $\mathtt{QUsed}^\sharp_{\mathtt{W}}$ first computes the number (cf. $\mathtt{Count}$) of input values of the variable in $\mathtt{W}$ (cf. $\mathtt{Proj}_{\Delta\setminus\mathtt{W}}$) that are missing (cf. $\mathtt{Compl}_\top$) coming from the same output bucket. These values represent missing partial input configurations that the concrete impact quantifier $\textsc{QUsed}_{\mathtt{W}}$ may count. To compute the number of missing (total) input configurations, we multiply the number of missing input values for the number of possible values of the other input variables (cf. $\prod_{\mathtt{x}\in\mathtt{W}} l_{\mathtt{x}}$). This number of possible values of the other input variables is computed by projecting away all the variables but one from the abstract element (cf. $\mathtt{Proj}_{\Delta\setminus\{\mathtt{x}\}}$) and then applying the abstract counting operator $\mathtt{Count}$ from the whole input space $\top \in \mathbb{D}$. The result is a measure of the influence of the input variables on the output of the program, *i.e.*, the number of input configurations that *may* lead to different outcomes.

The symbol $\top$ refers to "top," *i.e.*, the abstract element representing the whole space.

Definition 4.2.14 ($\mathtt{QUsed}^{\natural}_{W}$) *Let* $W \in \wp(\Delta)$ *be the input variables,* $n \in \mathbb{N}$ *be the number of output buckets, and* $B \in \mathbb{D}^n$ *the set of output buckets. We define* $\mathtt{QUsed}^{\natural}_{W} \in \mathbb{D}^n \times \mathbb{D}^n \to \mathbb{V}^{+\infty}_{\geq 0}$ *as:*

$$\mathtt{QUsed}^{\natural}_{W}(X, B) \stackrel{\text{def}}{=} \max_{j \leq n} \mathtt{Count}(\mathtt{Proj}_{\Delta \setminus W}(\mathtt{Compl}_{\top}(X_j))) \cdot \prod_{x \in W} l_x$$

where $X \in \mathbb{D}^n$ *is the result of the abstract semantics* $\Lambda^{\times}$ *and* l *is the total number of input values a variable takes:*

$$l_x \stackrel{\text{def}}{=} \mathtt{Count}(\mathtt{Proj}_{\Delta \setminus \{x\}}(\top))$$

Example 4.2.8 We show the computation of $\mathtt{QUsed}^{\natural}_{W}$ for the program Program 4.1. For the sake of the example, we consider as abstract domain the set of values for each variable, otherwise with the use of interval abstract domain, the result would be too imprecise. The backward analysis returns the following set of values:

$$\begin{aligned}
\Lambda^{\leftarrow}(B_{\text{low}}) &= X_{\text{low}} = \langle \mathtt{angle} \mapsto \{1\}, \mathtt{speed} \mapsto \{1,2,3\} \rangle \\
\Lambda^{\leftarrow}(B_{\text{mid-low}}) &= X_{\text{mid-low}} = \langle \mathtt{angle} \mapsto \{1\}, \mathtt{speed} \mapsto \{2,3\} \rangle \\
\Lambda^{\leftarrow}(B_{\text{mid-high}}) &= X_{\text{mid-high}} = \langle \mathtt{angle} \mapsto \{1\}, \mathtt{speed} \mapsto \{2,3\} \rangle \\
\Lambda^{\leftarrow}(B_{\text{high}}) &= X_{\text{high}} = \langle \mathtt{angle} \mapsto \{-4\}, \mathtt{speed} \mapsto \{1,2,3\} \rangle
\end{aligned}$$

We are interested in the impact of the input variable `angle`, hence we project away the other variables from the complement of the given abstract intervals, obtaining:

$$\begin{aligned}
&\mathtt{Proj}_{\{\mathtt{speed}\}}(\mathtt{Compl}_{\top}(X_{\text{low}})) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4,1\}, \mathtt{speed} \mapsto \{1,2,3\} \rangle \setminus X_{\text{low}}) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4\}, \mathtt{speed} \mapsto \emptyset \rangle) = \bot \\
&\mathtt{Proj}_{\{\mathtt{speed}\}}(\mathtt{Compl}_{\top}(X_{\text{mid-low}})) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4,1\}, \mathtt{speed} \mapsto \{1,2,3\} \rangle \setminus X_{\text{mid-low}}) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4\}, \mathtt{speed} \mapsto \{1\} \rangle) \\
&\quad = \langle \mathtt{angle} \mapsto \{-4\} \rangle \\
&\mathtt{Proj}_{\{\mathtt{speed}\}}(\mathtt{Compl}_{\top}(X_{\text{mid-high}})) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4,1\}, \mathtt{speed} \mapsto \{1,2,3\} \rangle \setminus X_{\text{mid-high}}) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4\}, \mathtt{speed} \mapsto \{1\} \rangle) \\
&\quad = \langle \mathtt{angle} \mapsto \{-4\} \rangle \\
&\mathtt{Proj}_{\{\mathtt{speed}\}}(\mathtt{Compl}_{\top}(X_{\text{high}})) \\
&\quad = \mathtt{Proj}_{\{\mathtt{speed}\}}(\langle \mathtt{angle} \mapsto \{-4,1\}, \mathtt{speed} \mapsto \{1,2,3\} \rangle \setminus X_{\text{high}}) = \bot
\end{aligned}$$

Then, $\mathtt{QUsed}^{\natural}_{\{\mathtt{angle}\}}$ counts the number of input configurations:

$$\begin{aligned}
\mathtt{Count}(\bot) &= 0 \\
\mathtt{Count}(\langle \mathtt{angle} \mapsto \{-4\} \rangle) &= 1
\end{aligned}$$

This value has to be multiplied by the number of possible values for the other input variable `speed`, which is 3. The total number of missing input configurations is 3 for the variable `angle`, hence

Prog. 4.1 (Landing alarm system):

```
1  landing_coeff =
2    abs(angle) + speed
3  if landing_coeff < 2:
4    risk = 0
5  else if landing_coeff > 5:
6    risk = 3
7  else:
8    risk =
9     floor(landing_coeff) - 2
10
```

$\texttt{QUsed}^\sharp_{\{\texttt{angle}\}}(X, B) = 3$. On the other hand, when interested in the variable `speed` we project away the variable `angle` from the abstract intervals, obtaining:

$$
\begin{aligned}
&\texttt{Proj}_{\{\texttt{angle}\}}(\texttt{Compl}_\top(X_{\text{low}})) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4, 1\}, \texttt{speed} \mapsto \{1, 2, 3\}\rangle \setminus X_{\text{low}}) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4\}, \texttt{speed} \mapsto \emptyset\rangle) \;=\; \bot \\
&\texttt{Proj}_{\{\texttt{angle}\}}(\texttt{Compl}_\top(X_{\text{mid-low}})) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4, 1\}, \texttt{speed} \mapsto \{1, 2, 3\}\rangle \setminus X_{\text{mid-low}}) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4\}, \texttt{speed} \mapsto \{1\}\rangle) \\
&\quad = \langle \texttt{speed} \mapsto \{1\}\rangle \\
&\texttt{Proj}_{\{\texttt{angle}\}}(\texttt{Compl}_\top(X_{\text{mid-high}})) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4, 1\}, \texttt{speed} \mapsto \{1, 2, 3\}\rangle \setminus X_{\text{mid-high}}) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4\}, \texttt{speed} \mapsto \{1\}\rangle) \\
&\quad = \langle \texttt{speed} \mapsto \{1\}\rangle \\
&\texttt{Proj}_{\{\texttt{angle}\}}(\texttt{Compl}_\top(X_{\text{high}})) \\
&\quad = \texttt{Proj}_{\{\texttt{angle}\}}(\langle \texttt{angle} \mapsto \{-4, 1\}, \texttt{speed} \mapsto \{1, 2, 3\}\rangle \setminus X_{\text{high}}) \;=\; \bot
\end{aligned}
$$

Then, $\texttt{QUsed}^\sharp_{\{\texttt{speed}\}}$ counts the number of input configurations:

$$
\begin{aligned}
\texttt{Count}(\bot) &= 0 \\
\texttt{Count}(\langle \texttt{speed} \mapsto \{1\}\rangle) &= 1
\end{aligned}
$$

As before, we multiply such value by the number of possible values for the other input variable `angle`, cf. 2. The total number of missing input configurations is then 2 for the variable `speed`, hence $\texttt{QUsed}^\sharp_{\{\texttt{speed}\}}(X, B) = 2$.

To prove that $\texttt{QUsed}^\sharp_{\mathtt{W}}$ is a sound implementation of $\text{QUsed}_{\mathtt{W}}$, we require the soundness condition on the abstract operators `Count` and `Compl`, ensuring that the abstract count of the complement of an abstract element is always lower than the concrete one. Note that, the concrete counterpart of the abstract `Count` is the cardinality operator $|\cdot|$.

Def. 4.1.4 (QUsed)

$$
\begin{aligned}
&\text{QUsed}_{\mathtt{W}}(T) \stackrel{\text{def}}{=} \\
&\quad \sup_{\sigma \in T} |\, Q_{\mathtt{W}}(I_\sigma(T)) \setminus I_\sigma(T) \,| \\
&I_\sigma(T) \;= \\
&\quad \{\sigma'_0 \mid \sigma' \in T \wedge \rho(\sigma_\omega) = \rho(\sigma'_\omega)\} \\
&Q_{\mathtt{W}}(S) \;= \\
&\quad \{s' \in \Sigma \mid s \in S \wedge s' =_{\Delta \setminus \mathtt{W}} s\}
\end{aligned}
$$

Definition 4.2.15 (Soundness of `Count`) *Given an abstract value $s^\sharp \in \mathbb{D}$ and the set of input variables of interest $\mathtt{W} \in \wp(\Delta)$, it holds that:*

$$|\, \{s_0(\mathtt{W}) \mid s \in \Sigma \wedge s \notin \gamma(s^\sharp)\} \,| \;\geq\; \texttt{Count}(\texttt{Compl}_\top(s^\sharp))$$

We designed $\texttt{QUsed}^\sharp_{\mathtt{W}}$ to under-approximate the concrete quantifier QUsed. Hence, the next result shows that the abstract impact $\texttt{QUsed}^\sharp$ is a sound over-approximation of the concrete impact QUsed, cf. Definition 4.1.4 w.r.t. the $\geq$ operator.

Lemma 4.2.10 ($\texttt{QUsed}^\sharp_{\mathtt{W}}$ is a Sound Implementation of $\text{QUsed}_{\mathtt{W}}$) *Let $\mathtt{W} \in \wp(\Delta)$ be the input variable of interest, $\mathbb{D}$ the abstract domain, $\Lambda^\leftarrow$ the family of semantics, and $B \in \mathbb{D}^n$ the starting output buckets. Whenever the following conditions hold:*

(i) B *covers the subset of potential outcomes, cf. Definition 4.2.5,*
(ii) B *is compatible with* ρ*, cf. Definition 4.2.11,*
(iii) `Proj` *is sound, cf. Definition 4.2.8, and*
(iv) `Count` *is sound, cf. Definition 4.2.15;*

then, $\texttt{QUsed}^\sharp_W$ *is a sound implementation of* $\textsc{QUsed}_W$ *w.r.t. the* $\geq$ *operator:*

$$\textsc{QUsed}_W(\Lambda) \;\geq\; \texttt{QUsed}^\sharp_W(\gamma^\times(\Lambda^\times)B, B)$$

Proof (Sketch). We need to prove that

$$\textsc{QUsed}_W(\Lambda) \stackrel{\text{def}}{=} \sup_{\sigma \in T} |\, Q_W(I_\sigma(T)) \setminus I_\sigma(T) \,|$$
$$\geq$$
$$\texttt{QUsed}^\sharp_W(\gamma^\times(\Lambda^\times)B, B) = \max_{j \leq n} \texttt{Count}(\texttt{Proj}_{\Delta\setminus W}(\texttt{Compl}_\top(X_j))) \cdot \prod_{x \in W} l_x$$

Let $\overline{\sigma} \in \Lambda$ be such that:

$$\sup_{\sigma \in T} |\, Q_W(I_\sigma(T)) \setminus I_\sigma(T) \,| \;=$$
$$|\, Q_W(I_{\overline{\sigma}}(T)) \setminus I_{\overline{\sigma}}(T) \,|$$

Let $\bar{j} \leq n$ be such that:

$$\max_{j \leq n} \texttt{Count}(\texttt{Proj}_{\Delta\setminus W}(\texttt{Compl}_\top(X_j))) \cdot \prod_{x \in W} l_x \;=$$
$$\texttt{Count}(\texttt{Proj}_{\Delta\setminus W}(\texttt{Compl}_\top(X_j)[\bar{j}])) \cdot \prod_{x \in W} l_x$$

By contradiction, we assume that

$$|\, Q_W(I_{\overline{\sigma}}(T)) \setminus I_{\overline{\sigma}}(T) \,| < \texttt{Count}(\texttt{Proj}_{\Delta\setminus W}(\texttt{Compl}_\top(X_j)[\bar{j}])) \cdot \prod_{x \in W} l_x$$

meaning that the number of missing input configurations is lower than the number of missing input values for the variable `W` coming from the same output bucket. Hence, either:

(a) there exist output abstractions that are not matched by any output bucket,
(b) some output abstractions are matched by the same output bucket,
(c) there are output abstractions in Λ from perturbation of the input s_0 that are not considered in the abstract, or
(d) counting the number of input configurations in the abstract is not greater than the number of input configurations in the concrete.

Again, we can show that the first three cases are not possible by hypothesis (i), (ii), and (iii). By hypothesis (iv), the soundness of the count operator ensures that the number of input configurations in the abstract is always greater than the number of input configurations in the concrete. Therefore, the assumption is false, and we conclude that $\texttt{QUsed}^\sharp_W$ is a sound implementation of $\textsc{QUsed}_W$. □

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\}$$
$$\subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \leq n} \gamma(B_j)\}$$

Def. 4.2.11 (Compatibility)

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p).$$
$$\rho(s_j) \neq \rho(s_p) \;\Rightarrow\; B_j \neq B_p$$

Def. 4.2.8 (Sound Projection Operator)

$$\{s' \in \Sigma^\perp \mid s \in \gamma(s^\sharp) \wedge s =_{\Delta\setminus W} s'\}$$
$$\subseteq\; \gamma(\texttt{Proj}_W(s^\sharp))$$

Def. 4.2.15 (Soundness of `Count`)

$$|\, \{s_0(W) \mid s \in \Sigma \wedge s \notin \gamma(s^\sharp)\} \,|$$
$$\geq$$
$$\texttt{Count}(\texttt{Compl}_\top(s^\sharp))$$

Prog. 4.1 (Landing alarm system):

```
1  landing_coeff =
2    abs(angle) + speed
3  if landing_coeff < 2:
4    risk = 0
5  else if landing_coeff > 5:
6    risk = 3
7  else:
8    risk =
9     floor(landing_coeff) - 2
10
```

Thm. 4.2.4 (Soundness)

$$\mathtt{Impact}^{\sharp}_{\mathtt{W}}(\Lambda^{\times}[\![P]\!]B, B) \otimes k \;\Rightarrow\; P \models \mathscr{B}^{\otimes k}_{\textsc{Impact}_{\mathtt{W}}}$$

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^{\rho}|_{\gamma(B_j)} \;\subseteq\; \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \le n} \gamma(B_j)\}$$

Example 4.2.9 Regarding the program Program 4.1, we show that the quantities computed by the abstract implementation $\mathtt{QUsed}^{\sharp}_{\mathtt{W}}$ in Example 4.2.8 are sound over-approximations of the concrete implementation $\textsc{QUsed}_{\mathtt{W}}$:

$$\begin{aligned}\textsc{QUsed}_{\{\mathtt{angle}\}}(\Lambda[\![L]\!]) &= 3\\ &\geq \mathtt{QUsed}^{\sharp}_{\{\mathtt{angle}\}}(X, B) = 3\end{aligned}$$

$$\begin{aligned}\textsc{QUsed}_{\{\mathtt{speed}\}}(\Lambda[\![L]\!]) &= 2\\ &\geq \mathtt{QUsed}^{\sharp}_{\{\mathtt{speed}\}}(X, B) = 2\end{aligned}$$

as expected by Theorem 4.2.4, where L is the program of the landing alarm system, cf. Program 4.1. In this case, the bound computed in the abstract is the same as the concrete one, as the abstract domain is precise enough to capture the exact number of missing input configurations.

Next, we show Theorem 4.2.4 (Soundness) instantiated with the abstract implementation $\mathtt{QUsed}^{\sharp}_{\mathtt{W}}$ and the comparison operator $\geq$.

Theorem 4.2.11 (Soundness of $\mathscr{B}^{\geq k}_{\textsc{QUsed}_{\mathtt{W}}}$) *Let $\mathscr{B}^{\geq k}_{\textsc{QUsed}_{\mathtt{W}}}$ be the property of interest we want to verify for the program P and the input variable $\mathtt{W} \in \wp(\Delta)$. Whenever,*

(i) $\Lambda^{\leftarrow}$ is sound with respect to Λ^{ρ}, cf. Definition 4.2.2, and
(ii) B covers the subset of potential outcomes, cf. Definition 4.2.5,

the following implication holds:

$$\mathtt{QUsed}^{\sharp}_{\mathtt{W}}(\Lambda^{\times}[\![P]\!]B, B) = k' \wedge k' \geq k \;\Rightarrow\; P \models \mathscr{B}^{\geq k}_{\textsc{QUsed}_{\mathtt{W}}}$$

Proof. Lemma 4.2.10 shows that $\mathtt{QUsed}^{\sharp}_{\mathtt{W}}$ is a sound implementation of $\textsc{QUsed}_{\mathtt{W}}$, the proof follows directly by application of the Theorem 4.2.4 instantiated with the abstract implementation $\mathtt{QUsed}^{\sharp}_{\mathtt{W}}$ and comparison operator $\leq$. □

Interestingly, the abstract implementation $\mathtt{QUsed}^{\sharp}_{\mathtt{W}}$ under-approximates the *used* variables, whereas the syntactic dependency analysis Λ^{u} (cf. Section 3.6) under-approximates the *unused* variables. Let us define the syntactic dependency analysis as an abstract implementation:

$$\textsc{SynUnused}_{\mathtt{W}}(P) \stackrel{\text{def}}{=} \begin{cases} \mathrm{T} & \text{if } \forall \mathtt{i} \in \mathtt{W}.\ \Lambda^{\mathrm{u}}[\![P]\!](\mathtt{i}) \in \{\mathrm{N}, \mathrm{W}\} \\ \mathrm{F} & \text{otherwise} \end{cases}$$

returning T if the syntactic dependency analysis Λ^{u} found that all the input variables $\mathtt{W}$ are unused, *i.e.*, unused (N) or overwritten (W), in the program P. It holds that, whenever $\mathtt{QUsed}^{\sharp}_{\mathtt{W}} = 0$ we cannot conclude anything about $\textsc{SynUnused}_{\mathtt{W}}$. Respectively, whenever $\textsc{SynUnused}_{\mathtt{W}}$ does not hold we cannot conclude anything about $\mathtt{QUsed}^{\sharp}_{\mathtt{W}}$. However, it holds that:

$$\begin{aligned}\mathtt{QUsed}^{\sharp}_{\mathtt{W}} \neq 0 \;&\Rightarrow\; \neg\textsc{SynUnused}_{\mathtt{W}}\\ \textsc{SynUnused}_{\mathtt{W}} \;&\Rightarrow\; \mathtt{QUsed}^{\sharp}_{\mathtt{W}} = 0\end{aligned}$$

In words, whenever the abstract implementation $\texttt{QUsed}^\sharp_W$ returns a strictly positive quantity, it means that the input variables W are definitely used in the program P, and thus also syntactically used.[4] Whenever the syntactic dependency analysis $\textsc{SynUnused}_W$ holds, it means that the input variables W are definitely not used in the program P, and thus also not used in the abstract implementation $\texttt{QUsed}^\sharp_W$.

4: Note the double negation in the predicate $\neg\textsc{SynUnused}$: being not syntactically unused is equivalent to be syntactically used

4.3 Related Work

This chapter draws inspiration from Barowy, Gochev, and Berger [77], where an empirical analysis explores the use of data for debugging spreadsheet applications through stochastic sampling. Urban and Müller [42] proposed a unified view of existing information flow analyses based on abstract interpretation, establishing the theoretical foundations for understanding the use of input variables. Our approach introduces a novel quantitative perspective on input data usage. In this section, we compare our work with the existing literature, including non-interference, a foundational property for information flow analysis, entropy measures of quantitative information flow, and probabilistic programming languages.

[77]: Barowy et al. (2014), 'CheckCell: data debugging for spreadsheets'

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

Abstract Non-Interference. Quantitative input data usage, which evolves from its qualitative counterpart, is inspired by work on abstract non-interference [43]. Notably, Giacobazzi and Mastroeni [78] introduced a domain-theoretic characterization of the most precise attacker who cannot violate abstract non-interference. Later, Mastroeni [79] instantiated various policies, leading to different notions of abstract non-interference for language-based security. This work paved the way for the development of a unifying framework that weakens the non-interference property into abstract non-interference [43]. More recently, Mastroeni and Pasqua [74] explored the connection between domain completeness and narrow abstract non-interference, where domain transformers can be characterized using monotonicity within the weak framework of abstract interpretation. From abstract non-interference, our work adopts the concept of state abstractions, cf. Definition 3.2.2 (Output Observer).

[43]: Giacobazzi et al. (2018), 'Abstract Non-Interference: A Unifying Framework for Weakening Information-flow'

[78]: Giacobazzi et al. (2004), 'Abstract non-interference: parameterizing non-interference by abstract interpretation'

[79]: Mastroeni (2013), 'Abstract interpretation-based approaches to Security - A Survey on Abstract Non-Interference and its Challenging Applications'

[74]: Mastroeni et al. (2023), 'Domain Precision in Galois Connection-Less Abstract Interpretation'

Quantitative Information Flow. Given the connection between *qualitative* input usage and information flow analysis [42, Chapters 8, 9, and 10], to design a quantitative input usage analysis that fits our purposes, it may come natural to look into *quantitative* information flow, formerly introduced by Denning [80] and Gray [81]. Such analyses measure information leakage about a secret through the concept of entropy, based on observations of the program's output values. Remarkably, this similarity between entropy and our notion of impact is even more evident in the work proposed by Köpf and Rybalchenko [82] which quantifies an upper bound of the entropy of a program's input variables by computing an over-approximation of the set of input-output observations, sometimes called equivalence classes. They employ Shannon entropy, min-entropy, and other entropies through the enumeration of these equivalence classes and their respective sizes. The equivalence classes are partitions of the input space in which two input assignments belong

[42]: Urban et al. (2018), 'An Abstract Interpretation Framework for Input Data Usage'

[80]: Denning (1982), *Cryptography and Data Security*

[81]: Gray (1991), 'Toward a Mathematical Foundation for Information Flow Security'

[82]: Köpf et al. (2013), 'Automation of Quantitative Information-Flow Analysis'

Prog. 4.1 (Landing alarm system):

```
landing_coeff =
  abs(angle) + speed
if landing_coeff < 2:
  risk = 0
else if landing_coeff > 5:
  risk = 3
else:
  risk =
   floor(landing_coeff) - 2

```

to the same class whenever the program produces an equivalent output. For example, the equivalence classes of the Program 4.1, referred to as program L, are $\Pi(L) = \{\{\langle x, y\rangle \mid L(x, y) = z\} \mid z \in \{0, 1, 2, 3\}\} = \{\{\langle 1, 1\rangle\}, \{\langle 1, 2\rangle\}, \{\langle 1, 3\rangle\}, \{\langle -4, 1\rangle, \langle -4, 2\rangle, \langle -4, 3\rangle\}\}$. Next, we show how our impact quantifiers arose from the adaptation of entropy measures to our needs in three successive attempts.

Initially, we notice that Shannon-entropy computes the average uncertainty of input values based on observations of the program's outcomes, while min-entropy, defined as:

$$H_\infty(P) \stackrel{\text{def}}{=} \log_2 \frac{|\textsc{Input}_P|}{|\Pi(P)|} \tag{4.1}$$

computes the worst-case uncertainty, where $\textsc{Input}_P$ is the set of all input values of a given program P. As a first attempt, we consider min-entropy as closer to our needs since our aim is to discover the worst-case impact, *i.e.*, the case in which a variable contributes the most. By computing min-entropy on the program L, we obtain $H_\infty(L) = 0.58$, indicating that the input is highly guessable. Indeed, when the risk level is 3, the potential values of input variables are `angle` $= -4$ and `speed` $\in 1, 2, 3$; for all other output values, the input values are completely determined. Unfortunately, min-entropy lacks granularity and measures the uncertainty of the input variables collectively. Instead, our aim is to quantify the individual contributions.

[83]: Köpf et al. (2010), 'Approximation and Randomization for Quantitative Information-Flow Analysis'

To address the previous issue, as a second attempt we exploit low and high labels for input variables, where the former are considered as public, available to the attacker, and the latter as secret [83]. To assess the impact of each input variable, we prioritize one high variable at a time, considering all others as low variables. Subsequently, we compute the min-entropy of the labelled program to quantify the extent of the impact. We define $L^{\texttt{angle}}(x) \stackrel{\text{def}}{=} \langle L(x, 1), L(x, 2), L(x, 3)\rangle$ which represents the sequence of programs where `angle` is high and `speed` is low. Similarly, $L^{\texttt{speed}}(y) \stackrel{\text{def}}{=} \langle L(-4, y), L(1, y)\rangle$ where `speed` is high and `angle` is low. Computing $H_\infty(L^{\texttt{angle}})$ and $H_\infty(L^{\texttt{speed}})$ yields 0 on both because all equivalence classes consist of singletons, meaning the number of inputs equals the number of outputs. Thus, there's no uncertainty in the value of `angle` given outputs of $L^{\texttt{angle}}$, or in the value of `speed` given outputs of $L^{\texttt{speed}}$. Indeed, observing the output $\langle 3, 3, 3\rangle$ from the program $L^{\texttt{angle}}$ implies that `angle` is -4, while observing $\langle 0, 1, 2\rangle$ implies that `angle` is 1. The same applies to $L^{\texttt{speed}}$ where observing $\langle 3, 0\rangle$ implies `speed` $= 1$, $\langle 3, 1\rangle$ implies `speed` $= 2$, and $\langle 3, 2\rangle$ implies `speed` $= 3$. However, this approach does not isolate the contributions of high variables; these outcomes are combined into a tuple of values through the return statement and thus evaluated together. Consequently, min-entropy cannot distinguish the contribution of each input variable independently.

An immediate solution is to develop a similar approach to the one used for the high-low variables, but instead of using min-entropy for the derived programs ($L^{\texttt{angle}}$ and $L^{\texttt{speed}}$), we count the number of outcomes of the partially-applied programs, cf. programs $L(x, 1)$, $L(x, 2)$, and $L(x, 3)$ for $L^{\texttt{angle}}$; $L(-4, y)$ and $L(1, y)$ for $L^{\texttt{speed}}$. These programs are referred to as $L^{\texttt{angle}}_y$ and $L^{\texttt{speed}}_x$ respectively. Therefore, the third

attempt defines $H_O(L^{\texttt{angle}}) \stackrel{\text{def}}{=} \max\{|\Pi(L_y^{\texttt{angle}})| \mid y \in \{1,2,3\}\}$ and $H_O(L^{\texttt{speed}}) \stackrel{\text{def}}{=} \max\{|\Pi(L_x^{\texttt{speed}})| \mid x \in \{-4,1\}\}$ retaining the maximum to obtain the worst-case scenario. As a result, $H_O(L^{\texttt{angle}}) = 2$ and $H_O(L^{\texttt{speed}}) = 3$. This means that variations of the value of `angle` result in at most 2 different outputs, while variations of the value of `speed` result in at most 3. Effectively, this is the first notion of impact derived from min-entropy capable of discriminating the contribution of each input variable on the program computation, exploiting the number of reachable outcomes from variations of the value of the input variable under consideration. Indeed, $H_O(P^W) = \textsc{Outcomes}_W(P)$ holds for a generic program P.

Overall, entropy measures and the approach proposed by Köpf and Rybalchenko [82, 83] can be adapted to our needs. Nevertheless, their analysis grows exponentially with the number of low variables, which in our adaptation corresponds to the number of inputs, minus one. To address this limitation, we leverage an over-approximation of input-output observations of the program, focusing solely on the low variables. By doing so, we obtain the set of input configurations that lead to the same output value by variation on the value of high variables. As a result, our approach performs the backward analysis only one time per output bucket, independently of the number of low variables. A similar technique could also be used to mitigate such explosion in their work.

[82]: Köpf et al. (2013), 'Automation of Quantitative Information-Flow Analysis'
[83]: Köpf et al. (2010), 'Approximation and Randomization for Quantitative Information-Flow Analysis'

In quantitative information flow, other notable works include Chothia, Kawamoto, and Novakovic [84], who proposed a statistical approach to quantify information leakage in Java programs. Phan et al. [85, 86] and Saha et al. [87] employed symbolic execution techniques and model counting to obtain sound bounds on program entropy. Other static analyses, such as those by Assaf et al. [88] and Clark, Hunt, and Malacaria [89], are based on abstract interpretation. Girol, Lacombe, and Bardin [90] introduced a quantitative version of non-exploitability called robust reachability, measuring the difficulty of triggering bugs via symbolic execution and model counting.

[84]: Chothia et al. (2014), 'LeakWatch: Estimating Information Leakage from Java Programs'
[85]: Phan et al. (2012), 'Symbolic quantitative information flow'
[86]: Phan et al. (2014), 'Quantifying information leaks using reliability analysis'
[87]: Saha et al. (2023), 'Obtaining Information Leakage Bounds via Approximate Model Counting'
[88]: Assaf et al. (2017), 'Hypercollecting semantics and its application to static analysis of information flow'
[89]: Clark et al. (2007), 'A static analysis for quantifying information flow in a simple imperative language'
[90]: Girol et al. (2024), 'Quantitative Robustness for Vulnerability Assessment'

The key difference between our framework and existing work lies in the type of information we measure. Our analysis quantifies the effect of each input variable on the program's outcome, focusing on numerical properties. In contrast, quantitative information flow typically measures the amount of information (in bits) transferred from input variables to the program's outcome, a metric more relevant to security properties. For example, Assaf et al. [88] counts how many bits of input information are used to compute the result; Smith [45] calculates the probability of guessing a private variable's value from input variables; and McCamant and Ernst [91] retrieves the channel capacity, which represents the worst-case potential for information leakage. Most of these approaches are based on entropy measures, which are orthogonal to our approach. Recent developments include Zhang and Kaminski [92], who developed a calculus based on the strongest-postcondition style for quantitative reasoning about information flow, and Henzinger, Mazzocchi, and Saraç [93], who generalized the hierarchy of safety and liveness properties to quantitative safety and liveness. Our framework, inspired by the qualitative input data usage property, diverges in that it assesses the numerical impact of input variables on program outcomes, distinguishing

[88]: Assaf et al. (2017), 'Hypercollecting semantics and its application to static analysis of information flow'
[45]: Smith (2009), 'On the Foundations of Quantitative Information Flow'
[91]: McCamant et al. (2008), 'Quantitative information flow as network flow capacity'
[92]: Zhang et al. (2022), 'Quantitative strongest post: a calculus for reasoning about the flow of quantitative information'
[93]: Henzinger et al. (2023), 'Quantitative Safety and Liveness'

it from traditional information flow approaches that focus on counting bits or probabilities.

Qualitative input data usage and non-interference also relate to *Data Leakage* [94, Chapter 4]. This concept is particularly relevant to data science software, addressing leakage issues that arise from the training and test datasets during the process of training neural networks. A series of papers by Drobnjakovic, Subotic, and Urban [95], Subotic, Bojanic, and Stojic [96], and Subotic, Milikic, and Stojic [97] focus on verifying data leakage within code notebooks and external libraries such as NumPy or pandas, presenting new challenges in formal verification. An adaptation of their work could potentially encompass a notion of quantitative data leakage, which would be a valuable direction for further research.

Hyperproperties. Clarkson and Schneider [58] defined properties of program semantics as"hyperproperties." Hyperproperties relate different sets of program executions and can express security policies or input data usage properties, as discussed in this thesis. Clarkson and Schneider [58] categorized hyperproperties into safety and liveness, respectively termed hypersafety and hyperliveness. The k-hypersafety properties are those that require sets of k traces to be disproved, a category verifiable with traditional safety property techniques by k-times self-composition [98, 99]. Note, however, that direct approaches using self-composition can be computationally expensive [100]. In our work, we prefer "program properties" to the term "hyperproperties" because a different abstract interpretation theory is not needed to verify them [101], the classical approach for trace properties [36] is sufficient. However, it is worth noting that the k-bounded impact property is a hyperproperty. More specifically, when applied to the Range quantifier, $\mathscr{B}^{\otimes k}_{\text{Range}_{\mathbb{N}}}$ is a 2-hypersafety property, as it requires 2 traces to be disproved, *i.e.*, two traces yielding extreme values for the output variable. Conversely, when applied to the Outcomes quantifier, $\mathscr{B}^{\otimes k}_{\text{Outcomes}_{\mathbb{N}}}$ is a $(k+1)$-hypersafety property, as it requires $k+1$ traces to be disproved, *i.e.*, where the $k+1$ trace all yield a different output value.

Probabilistic Programming. Quantitative analyses are typically closely associated with probabilistic programming. Probabilistic programs [102, 103] are conventional imperative (or functional) programs with additional constructs to model random variables and probabilistic choices. The problem of computing an explicit representation of the probability distribution specified by a probabilistic program is known as probabilistic inference, with probabilistic programming aiming to decouple modeling from inference. Recent advances in verifying probabilistic programs have been significant. Martingale theory certificates have played a crucial role in automating the proof of quantitative termination and safety in probabilistic programs [104–107]. Probabilistic programs have also been studied in the context of cost analyses [108, 109]. Other approaches involve the weakest pre-expectation calculus [110], a probabilistic extension of the classical weakest precondition calculus. Symbolic execution techniques have been adapted for probabilistic programs, enabling a detailed exploration of program paths under various stochastic scenarios [111, 112]. In our work, we assume non-determinism is uniformly distributed, which simplifies the settings. However, by applying probabilistic semantics [113],

[94]: Brownlee (2020), *Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python*

[95]: Drobnjakovic et al. (2024), 'An Abstract Interpretation-Based Data Leakage Static Analysis'

[96]: Subotic et al. (2022), 'Statically detecting data leakages in data science code'

[97]: Subotic et al. (2022), 'A Static Analysis Framework for Data Science Notebooks'

[58]: Clarkson et al. (2010), 'Hyperproperties'

[98]: Barthe et al. (2011), 'Secure information flow by self-composition'

[99]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

[100]: Antonopoulos et al. (2017), 'Decomposition instead of self-composition for proving the absence of timing channels'

[101]: Cousot (2019), 'Abstract Semantic Dependency'

[36]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'

[102]: Gordon et al. (2014), 'Probabilistic programming'

[103]: Meent et al. (2018), 'An Introduction to Probabilistic Programming'

[104]: Chatterjee et al. (2017), 'Stochastic invariants for probabilistic termination'

[105]: Chatterjee et al. (2022), 'Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs'

[106]: Takisaka et al. (2021), 'Ranking and Repulsing Supermartingales for Reachability in Randomized Programs'

[107]: Wang et al. (2024), 'Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving'

[108]: Chatterjee et al. (2024), 'Quantitative Bounds on Resource Usage of Probabilistic Programs'

[109]: Wang et al. (2021), 'Central moment analysis for cost accumulators in probabilistic programs'

[110]: Kaminski et al. (2016), 'Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs'

[111]: Sankaranarayanan et al. (2013), 'Static analysis for probabilistic programs: inferring whole program properties from finitely many paths'

[112]: Beutner et al. (2022), 'Guaranteed bounds for posterior inference in universal probabilistic programming'

[113]: Cousot et al. (2012), 'Probabilistic Abstract Interpretation'

we could extend our quantifiers to probabilistic measures, providing statistical bounds on the impact of input variables on the probabilistic program's output.

4.4 Summary

In this chapter, we introduced the quantifiers $\textsc{Outcomes}_{\mathbb{W}}$, $\textsc{Range}_{\mathbb{W}}$, and $\textsc{QUsed}_{\mathbb{W}}$ to measure the impact of input variables on the output of a program. We developed a theoretical framework to verify the k-bounded impact property of a program and showed the abstract version of the quantifiers. In the next chapter, we present the evaluation of the quantifiers on a set of use cases. Later, we will show how we handle the quantification of the impact of input variables in the context of neural network models.

Dans ce chapitre, nous avons introduit les quantificateurs $\textsc{Outcomes}_{\mathbb{W}}$, $\textsc{Range}_{\mathbb{W}}$, *et* $\textsc{QUsed}_{\mathbb{W}}$ *pour mesurer l'impact des variables d'entrée sur le résultat d'un programme. Nous avons développé un cadre théorique pour vérifier la propriété d'impact borné par k d'un programme et montré la version abstraite des quantificateurs. Dans le prochain chapitre, nous présenterons l'évaluation des quantificateurs sur un ensemble de cas d'utilisation. Plus tard, nous montrerons comment nous abordons la quantification de l'impact des variables d'entrée dans le contexte des modèles de réseaux neuronaux.*

5 Implementation

In this chapter, we demonstrate the potential applications of our quantitative input data usage analysis, by evaluating an automatic proof-of-concept tool of our static analysis, called Impatto,[1] on six different demonstrative programs: a simplified program from the Reinhart and Rogoff article (cf. Section 5.1), a program extracted from a recent OpenAI keynote (cf. Section 5.2), two programs from the software verification competition SV-Comp (cf. Section 5.3 and Section 5.4), a program computing a linear expression via repeated additions (cf. Section 5.5), and Program 4.1 computing the landing risk coefficient for an alarm system (cf. Section 5.6). The tool Impatto is implemented in Python 3 and uses the Interproc[2] abstract interpreter to perform the backward analysis. We exploit this tool to automatically derive a sound quantification of the input data usage of each use case. As each impact result must be interpreted with respect to what the program computes, we analyze each use case separately.

1: github.com/denismazzucato/impatto

2: github.com/jogiet/interproc

The impact quantifiers considered in this chapter are Outcomes and Range. The QUsed quantifier has not been considered in the evaluation since we consider a continuous input space in most of the use cases: in such instance QUsed would either return $+\infty$ or 0 for each variable as it would discover that either are missing an infinite amount of input values or none at all.

An artifact of Impatto, including the source code and the six use cases is available on Zenodo.[3] This chapter is based on the evaluation section of the work published at NASA Formal Methods Symposium (NFM) 2024 [46, Section 5]. In the next chapters, we will apply our quantitative framework in the context of neural networks.

3: doi.org/10.5281/zenodo.10392830

[46]: Mazzucato et al. (2024), 'Quantitative Input Usage Static Analysis'

Dans ce chapitre, nous démontrons les applications potentielles de notre analyse quantitative de l'utilisation des données d'entrée en évaluant un outil de preuve de concept automatique de notre analyse statique, appelé Impatto[1]*, sur six programmes démonstratifs différents : un programme simplifié tiré de l'article de Reinhart et Rogoff (cf. Section 5.1), un programme extrait d'une récente keynote d'OpenAI (cf. Section 5.2), deux programmes issus de la compétition de vérification logicielle SV-Comp (cf. Section 5.3 et Section 5.4), un programme calculant une expression linéaire via des additions répétées (cf. Section 5.5), et Program 4.1 calculant le coefficient de risque d'atterrissage pour un système d'alarme (cf. Section 5.6). L'outil* Impatto *est implémenté en Python 3 et utilise l'interpréteur abstrait* Interproc[2] *pour effectuer l'analyse rétrograde. Nous exploitons cet outil pour dériver automatiquement une quantification correcte de l'utilisation des données d'entrée pour chaque cas d'utilisation. Comme chaque résultat d'impact doit être interprété en fonction de ce que le programme calcule, nous analysons chaque cas d'utilisation séparément.*

Les quantificateurs d'impact considérés dans ce chapitre sont Outcomes *et* Range*. Le quantificateur* QUsed *n'a pas été pris en compte dans l'évaluation car nous considérons un espace d'entrée continu dans la plupart des cas d'utilisation : dans un tel cas,* QUsed *renverrait soit* $+\infty$*, soit* 0 *pour chaque variable,*

car il découvrirait soit un nombre infini de valeurs d'entrée manquantes, soit aucune.

Un artefact de Impatto, incluant le code source et les six cas d'utilisation, est disponible sur Zenodo.[4] *Ce chapitre est basé sur la section d'évaluation du travail publié au NASA Formal Methods Symposium (NFM) 2024 [46, Section 5]. Dans les prochains chapitres, nous appliquerons notre cadre quantitatif dans le contexte des réseaux neuronaux.*

4: doi.org/10.5281/zenodo.10392830

[46]: Mazzucato et al. (2024), 'Quantitative Input Usage Static Analysis'

5.1 Growth in a Time of Debt

Reinhart and Rogoff article "Growth in a Time of Debt" [40] proposed a correlation between high levels of public debt and low economic growth, and was heavily cited to justify austerity measures around the world. One of the several errors discovered in the article is the incorrect usage of the input value relative to Norway's economic growth in 1964. The data used in the article is publicly available but not the spreadsheet file. We reconstructed this simplified example based on the technical critique by Herndon, Ash, and Pollin [41], and an online discussion.[5] The Program 5.1 computes the cross-country mean growth for the public debt-to-GDP 60 − 90% category, key point to the article's conclusions. The input data is the average growth rate for each country within this public dept-to-GDP category. The problem with this computation is that Norway has only one observation in such category, which alone could disrupt the mean computation among all the countries. Indeed, the year that Norway appears in the 60 − 90% category achieved a growth rate of 10.2%, while the average growth rate for the other countries is 2.7%. With such high rate, the mean growth rate raised to 3.4%, altering the article's conclusions. We assume growth rate values between −20% and 20% for all countries, consequentially, the output ranges are between these bounds as well. We instrumented the output buckets to cover the full output space in buckets of size 1, *i.e.*, $\{t \leq \texttt{avg} < t + 1 \mid -20 \leq t \leq 20\}$.

[40]: Reinhart et al. (2010), 'Growth in a Time of Debt'

[41]: Herndon et al. (2014), 'Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff'

5: economics.stackexchange.com/q/18553

Prog. 5.1: Program computing the mean growth rate in the 60 − 90% category.

```
def mean_growth_rate_60_90(
    portugal1, portugal2, portugal3,
    norway1,
    uk1, uk2, uk3, uk4,
    us1, us2, us3):
  portugal_avg = (portugal1 + portugal2 + portugal3) / 3
  norway_avg = norway1
  uk_avg = (uk1 + uk2 + uk3 + uk4) / 4
  us_avg = (us1 + us2 + us3) / 3
  avg = (portugal_avg + norway_avg + uk_avg + us_avg) / 4
```

Table 5.1: Quantitative input usage for Program 5.1 from the Reinhart and Rogoff's article.

Input	Outcomes$^\natural$	Range$^\natural$
portugal1	5	5
portugal2	5	5
portugal3	5	5
norway1	10	10
uk1	2	2
uk2	2	2
uk3	2	2
uk4	2	2
us1	3	3
us2	3	3
us3	3	3

Results for both Outcomes$^\natural$ and Range$^\natural$ are shown in Table 5.1. The analysis discovers that the Norway's only observation for this category norway1 has the biggest impact on the output, as perturbations on its value are capable of reaching 10 different outcomes (cf. row norway1), while the other countries only have 5, 2, and 3, respectively for Portugal, UK, and US. The same applies to Range$^\natural$ as the output buckets have size 1 and all the input perturbations are only capable of reaching contiguous buckets. Hence, we obtain the same exact results.

Our analysis is able to discover the disproportionate impact of Norway's only observation in the mean computation, which would have prevented one of the several programming errors found in the article. From a review

of Program 5.1, it is clear that Norway's only observation has a greater contribution to the computation, as it does not need to be averaged with other observations first. However, such methodological error is less evident when dealing with a higher number of input observations (1175 observations in the original work) and the computation is hidden behind a spreadsheet.

5.2 GPT-4 Turbo

The second use case we present is drawn from Sam Altman's OpenAI keynote in September 2023,[6] where he presented the GPT-4 Turbo. This new version of the GPT-4 language model brings the ability to write and interpret code directly without the need of human interaction. Hence, as showcased in the keynote, the user could prompt multiple information to the model, such as related to the organization of a holiday trip with friends in Paris, and the model automatically generates the code to compute the share of the total cost of the trip and run it in background. In this environment, users are unable to directly view the code unless they access the backend console. This limitation makes it challenging for them to evaluate whether the function has been implemented correctly or not, assuming users have the capability to do so. From the keynote, we extracted the Program 5.2 which computes the user's share of the total cost of a holiday trip to Paris, given the total cost of the Airbnb, the flight cost, and the number of friends going on the trip.

6: www.youtube.com/live/U9mJuUkhUzk?si=HOzuH3-gr_kTdhCt&t=2330

```
def share_division(
    airbnb_total_cost_eur,
    flight_cost_usd,
    number_of_friends):
  share_airbnb = airbnb_total_cost_eur / number_of_friends
  usd_to_eur = 0.92
  flight_cost_eur = flight_cost_usd * usd_to_eur
  total_cost_eur = share_airbnb + flight_cost_eur
```

Prog. 5.2: Program computing share division for holiday planning among friends.

Regarding the input bounds, users are willing to spend between 500 and 2000 for the Airbnb, between 50 and 1000 for the flight, and travel with between 2 and 10 friends. As a result, they expect their share, variable `total_cost_eur`, to be between 90 and 1900. To compute the impact of the input variables we choose the output buckets to cover the expected output space in buckets of size 100, *i.e.*, $\{100t + 90 \leq \texttt{total\_cost\_eur} < \min\{100(t+1)+90, 1900\} \mid 0 \leq t \leq 19\}$. The findings are similar for both the `Outcomes`$^\sharp$ and `Range`$^\sharp$ analysis, see Table 5.2. The input variable `flight_cost_usd` has the biggest impact on the output, as perturbations on its value are capable of reaching 17 different output buckets (resp. a range of 1709 output values), while the other two, `airbnb_total_cost_eur` and `number_of_friends`, only reach 10 and 9 output buckets (resp. have ranges of size 1099 and 999), respectively.

Table 5.2: Quantitative input usage for Program 5.2 computing the share division among friends.

Input	`Outcomes`$^\sharp$	`Range`$^\sharp$
`airbnb_total_cost_eur`	10	1099
`flight_cost_usd`	17	1709
`number_of_friends`	9	999

These results confirm the user expectations about the proposed program from ChatGPT: the flight cost yields the biggest impact as it cannot be shared among friends.

5.3 Termination Analysis (A)

Prog. 5.3: Example program from termination analysis.

```
def example(x, y):
  counter = 0
  while x >= 0:
    if y <= 50:
      x += 1
    else
      x -= 1
    y += 1
    counter += 1
```

Program 5.4 is adapted from the termination category of the software verification competition SV-COMP.[7] Assuming both input positives, $x, y \geq 0$, this program terminates in $x+1$ iterations if $y > 50$, otherwise it terminates in $x - 2y + 103$ iterations. We define `counter` as the output variable, with output buckets defined as $\{10k \leq \texttt{counter} < 10(k+1) \mid 0 \leq k < 50\}$ and $\{\texttt{counter} \geq 500\}$. These output buckets represent cumulative ranges of iterations required for termination. The analysis results are illustrated in Table 5.3, they show that the input variable `x` has the biggest impact. Modifying the value of `x` can result in the program terminating within any of the other 50 iteration ranges. On the other hand, perturbations on `y` can only result in the program terminating within 10 different iteration ranges. Such difference is motivated by the fact that `y` is only used to determine the number of iterations in the case where `y` is greater than 50, otherwise it is not used at all. Therefore, two values of `y`, *e.g.*, y_0 and y_1, only result in two different ranges of iterations required to make the program terminate if either both of them are below 50 or $y_0 < 50 \wedge y_1 \geq 50$ or $y_0 \geq 50 \wedge y_1 < 50$, not in all the cases.

7: sv-comp.sosy-lab.org/

Prog. 5.4: Example program from termination analysis.

```
def example(x, y):
  counter = 0
  while x >= 0:
    if y <= 50:
      x += 1
    else
      x -= 1
    y += 1
    counter += 1
```

Table 5.3: Quantitative input usage for Program 5.4.

Input	Outcomes$^\natural$	Range$^\natural$
x	50	499
y	10	99

The given results can be interpreted as follows: the speed of termination of this loop is highly dependent on the value of `x`, while `y` has a much smaller impact.

5.4 Termination Analysis (B)

This use case comes from the software verification competition SV-Comp, [8] where the goal is to verify the termination of a program. Program 5.5 and Program 5.6 have originally been proposed by Chen, Flur, and Mukhopadhyay [114], respectively these are Example (2.16) and Example (2.21) of such work.

Program 5.5 returns the value of `y` whenever $x = 0$, otherwise it returns -1. We assume both input variables are positive up to 1000, $0 \leq x \leq 1000$ and $0 \leq y \leq 1000$. Regarding such a function, it is interesting to study its behaviors around 0, thus the output bucket are $\{\texttt{result} < 0\}$, $\{\texttt{result} = 0\}$, and $\{\texttt{result} > 0\}$. With the above parameters, the analysis Outcomes$^\natural$ returns 1 for both input variables. Such result is not too interesting, but by looking at the internal stages of the analysis we notice that perturbations on the value of the variable `x` may be able to produce from an output negative value to zero or a positive one (and viceversa). While perturbations on the value of the variable `y` are only able to produce from zero to positive (and viceversa).

As a second experiments, we consider the buckets from -1 to 19, $\{\texttt{result} = n \mid -1 \leq n \leq 19\}$, and we notice that the analysis Outcomes$^\natural$ returns 1 for the input variable `x` and 19 for `y`, meaning that the variable `y` is able to affect far more output values than `x`. However, combing the results of the previous experiment, only the variable `x` is able to affect the negative output values.

From the same work, we also consider Program 5.6 which returns the value of `y` whenever $x = 0$, otherwise it returns -1. Unfortunately, the backward analysis does not capture a precise loop invariant, thus both the analyses Outcomes$^\natural$ and Range$^\natural$ are inconclusive in such case. The

8: sv-comp.sosy-lab.org/

[114]: Chen et al. (2012), 'Termination Proofs for Linear Simple Loops'

Prog. 5.5: Program Ex2.16 from software verification competition SV-Comp.

```
def termination_a(x, y):
  while x > 0:
    x = y
    y = y - 1
  result = x + y
  return result
```

Prog. 5.6: Program Ex2.21 from software verification competition SV-Comp.

```
def termination_b(x, y):
  while x > 0:
    x = x + y
    y = -y - 1
  result = x + y
  return result
```

key takeaway is that our analysis is highly dependent on the precision of the underlying backward analysis.

As a conclusion, even though SV-Comp proposes challenging benchmarks for termination, reachability, and safety analyses, they are not amenable for information flow analysis. Most of the time, their examples involve loops with complex invariant, but as input-output relations, the variables involved are just zeroed out after the loop. Drawing examples from their dataset is less appealing to our work.

5.5 Linear Loops

Program 5.7 computes the linear expression $(5x + 2y)$ via repeated additions. Note that the invariant of the loop is indeed non-linear (`result = i * x` and `result = result' + i * y` respectively for the first and second loop, where `result'` is the value of `result` before entering the second loop), but the loop is executed a fixed number of times, thus the analysis is able to compute the exact output buckets through loop unrolling.

For the analysis the input bounds are $0 \leq \texttt{x} \leq 1000$ and $0 \leq \texttt{y} \leq 1000$, while the output buckets are $\{n * 100 \leq \texttt{result} < (n+1) * 100 \mid n \leq 70\}$. Both analyses, $\texttt{Outcomes}^{\natural}$ and $\texttt{Range}^{\natural}$, show that x has an impact $\frac{5}{2}$ times bigger than y on the output. Thus, the impact quantity provides insight about the termination speed. Indeed, the loop for x is executed 5 times, while the one for y only 2.

Prog. 5.7: Program computing the linear expression $(5x + 2y)$ via repeated additions.

```
def linear_expression(x, y):
  result = 0
  i = 0
  while i < 5:
    result = result + x
    i += 1
  i = 0
  while i < 2:
    result = result + y
    i += 1
```

5.6 Landing Risk System

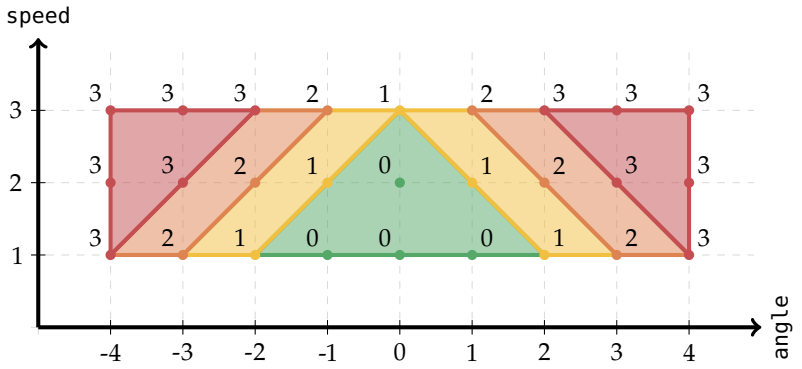


Figure 5.1: Input space composition with continuous input values.

Finally, we apply our quantitative analysis to Program 4.1 for the landing alarm system extended with the continuous input space for the aircraft angle of approach, where $(-4 \leq \texttt{angle} \leq 4) \wedge (1 \leq \texttt{speed} \leq 3)$, see Figure 5.1. In this instance, the precision of the abstraction drastically drops as convex abstract domains are not able to capture the symmetric features of the input space around 0. Indeed, the analysis result, cf. Figure 5.2, is unable to reveal any difference in the input usage of input variables as all the abstract preconditions result of the backward analysis intersect together. As a consequence, $\texttt{Outcomes}^{\natural}$ and $\texttt{Range}^{\natural}$ are unable to provide any meaningful information, first row of Table 5.4.

Prog. 4.1 (Landing alarm system)**:**

```
landing_coeff =
  abs(angle) + speed
if landing_coeff < 2:
  risk = 0
else if landing_coeff > 5:
  risk = 3
else:
  risk =
   floor(landing_coeff) - 2
```

Table 5.4: Quantitative input usage for Program 4.1.

Input Bounds		Outcomes♮		Range♮	
		angle	speed	angle	speed
$-4 \leq$ angle ≤ 4		3	3	3	3
$-4 \leq$ angle ≤ 0	$\wedge 1 \leq$ speed ≤ 3	3	2	3	2
$0 \leq$ angle ≤ 4		3	2	3	2

A possible approach to overcome the non-convexity of the input space is to split the input space into two subspaces (as a bounded set of disjunctive polyhedra), $-4 \leq$ angle ≤ 0 and $0 \leq$ angle ≤ 4, second and third row of Table 5.4. In the first subset $-4 \leq$ angle ≤ 0, we are able to perfectly captures the input regions that lead to each output bucket with our abstract analysis, cf. Figure 5.3. Therefore, we are able to recover the information that the input configurations from the bucket {risk = 3} do not intersect with the ones from the bucket {risk = 0} after projecting away the axis speed. As the end, our analysis notices that variations in the value of the input angle results in three possible output values, while variations in the speed input lead to two. Similarly, regarding the range of values, variations in the angle input cover the entire spectrum of output values, whereas to the speed input only span a range of 2 since it exists no input value such that modifications in the speed value could obtain a range of output values bigger than 2. The same reasoning applies to the other subspace with $0 \leq$ angle ≤ 4.

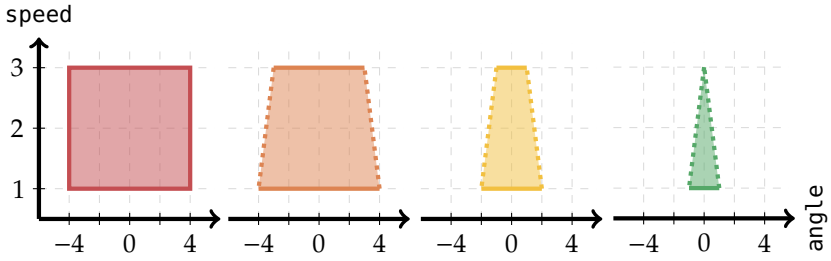


Figure 5.2: Result of the analysis with convex polyhedra.

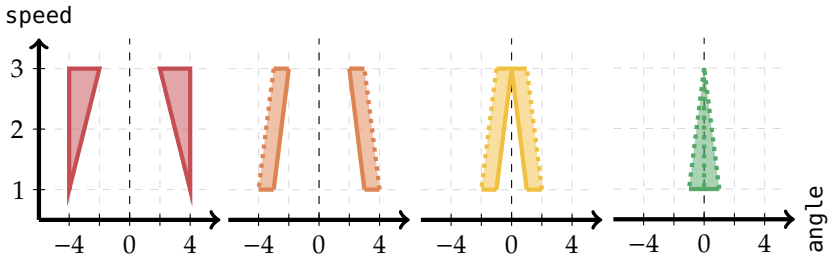


Figure 5.3: Result after splitting the input space into two subspaces around angle = 0.

5.7 Summary

This chapter evaluated our approach for the quantification of the influence of input variables on the output computation of a program. Next, we apply the quantitative framework to the context of neural networks.

Ce chapitre a évalué notre approche pour la quantification de l'influence des variables d'entrée sur le calcul du résultat d'un programme. Ensuite, nous appliquerons le cadre quantitatif au contexte des réseaux neuronaux.

Quantitative Verification for Neural Networks

6

In this chapter, we introduce feed-forward deep neural networks classifiers. We define two new quantitative impact quantifiers: the Changes impact quantifier, which targets the repetitions of changes in the outcome, and the QAUnused impact quantifier, which measures the amount of unused input space. We then present the abstract implementation of the two impact quantifiers, respectively called `Changes`$^\natural$ and `QAUnused`$^\natural$, and we show how to validate the k-bounded impact property for both quantifiers. Finally, we present the parallel implementation `QAUnused`$^\natural$ for efficient neural network verification of the QAUnused impact quantifier. This chapter is based on the work presented at the 28th Static Analysis Symposium (SAS) 2021 [47]. The next chapter will present the experimental evaluation for the quantitative verification of neural networks.

[47]: Mazzucato et al. (2021), 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'

Dans ce chapitre, nous introduisons les classificateurs de réseaux neuronaux profonds feed-forward. Nous définissons deux nouveaux quantificateurs d'impact quantitatif : le quantificateur d'impact Changes*, qui cible les répétitions de changements dans le résultat, et le quantificateur d'impact* QAUnused*, qui mesure la quantité d'espace d'entrée non utilisé. Nous présentons ensuite l'implémentation abstraite de ces deux quantificateurs d'impact, respectivement appelés* `Changes`$^\natural$ *et* `QAUnused`$^\natural$*, et nous montrons comment valider la propriété d'impact borné par k pour ces deux quantificateurs. Enfin, nous présentons l'implémentation parallèle* `QAUnused`$^\natural$ *pour une vérification efficace des réseaux neuronaux en utilisant le quantificateur d'impact* QAUnused*. Ce chapitre est basé sur les travaux présentés au 28e Symposium sur l'analyse statique (SAS) 2021 [47]. Le prochain chapitre présentera l'évaluation expérimentale pour la vérification quantitative des réseaux neuronaux.*

6.1 Neural Networks

This section introduces the computational model of a feed-forward deep neural network and how it can be used for classification purposes.

6.1.1 Feed-Forward Deep Neural Networks

A *feed-forward deep neural network* is a directed acyclic graph where each node represents a neuron, and each edge represents a connection between neurons. The nodes are organized into layers: the first layer is the input layer (L_0), the last layer is the output layer (L_N), and the layers in between are called hidden layers ($\mathrm{L}_1, \ldots, \mathrm{L}_{\mathrm{N}-1}$). Each layer L_i, for $1 \leq i \leq \mathrm{N}$, consists of $|\mathrm{L}_i|$ nodes and is connected to the previous layer L_{i-1} through a weight matrix W_i of size $|\mathrm{L}_i| \times |\mathrm{L}_{i-1}|$ and a bias vector b_i of size $|\mathrm{L}_i|$.

The entire set of nodes in the network is denoted by X, and the set of nodes in layer L_i is denoted by X_i. The j-th node in layer L_i is denoted

by $x_{i,j}$. In this thesis, we focus on feed-forward neural networks used for classification tasks, where each node in the output layer represents a class, and the output of the network is the class corresponding to the node in the output layer with the highest value. The network has $|\mathrm{L_N}|$ target classes in total.

An input value is a $|\mathrm{L_0}|$-dimensional vector X_0. For simplicity, in the rest of the chapter, we assume that the input values are normalized to the interval $[0, 1]$. The value of each node in hidden layers is computed by applying an activation function to the weighted sum of the values of the nodes in the previous layer and the bias. The activation function is usually a non-linear function; in this work, we consider the ReLU activation function, defined as $\mathrm{ReLU}(x) = \max(0, x)$. Thus, the value of node $x_{i,j}$ in layer L_i is computed as:

$$x_{i,j} = \mathrm{ReLU}\left(\sum_{k=1}^{|\mathrm{L}_{i-1}|} W^i_{j,k} x_{i-1,k} + b^i_j\right)$$

where $W^i_{j,k}$ is the weight connecting node $x_{i-1,k}$ in layer L_{i-1} to node $x_{i,j}$ in layer L_i, and b^i_j is the bias of node $x_{i,j}$ in layer L_i. Weights and biases are learned during the training phase of the network. In the following, we consider networks that have been previously trained.

For the rest of this chapter, the letter M denotes a neural network model. The trace semantics of a neural network model is denoted by $\Lambda[\![M]\!] \in \wp(\Sigma^{+\infty})$. Note that, for the models we consider in this thesis, we have that the output of the network is deterministic for each input configuration, and that all the traces terminate after a specific number of steps, which is exactly the number of nodes in the network.

6.1.2 Classification Task

In this thesis, we focus on feed-forward neural networks used for classification tasks. Given an input vector, the network classifies it into one of the target classes, each represented by a node in the output layer. The network classifies an input vector by computing the values of the nodes in the output layer and then selecting the class associated with the node having the highest value, *i.e.*, $\arg\max_j x_{\mathrm{N},j}$.

Building on the quantitative framework of Chapter 4 (Quantitative Input Data Usage), the classification task is formalized using the output observer ρ (Definition 3.2.2), which maps the output values of the network to the target classes. The output observer returns 1 if the given node (*i.e.*, a variable of the neural network) corresponds to the target class (*i.e.*, nodes in the output layer) with the highest value, and 0 otherwise. Thus, when the output state of a trace from the network computation is observed by ρ, the only non-zero value corresponds to the target class. Formally, the output observer is defined as:

Def. 3.2.2 (Output Observer) *Given a program P, an* output observer $\rho \in \Sigma^{\perp} \to \Sigma^{\perp}$ *is an upper closure operator that abstracts the value of output states.*

$X_{\mathrm{L_N}}$ denotes the set of nodes in the output layer of a neural network.

$$\rho(s) \stackrel{\text{def}}{=} \lambda \mathsf{y}. \begin{cases} 1 & \text{if } \mathsf{y} = \arg\max_{x_{i,j} \in X_{\mathrm{L_N}}} s(x_{i,j}) \\ 0 & \text{otherwise} \end{cases}$$

where y is a node (not necessarily in the output layer) of the network. This characterization shows that two different states of the network can

be distinguished by the output observer if and only if they are associated with different target classes, thus yielding different outcomes.

Example 6.1.1 Consider a simple neural network with two input and two output variables. The weights and biases connecting the input to the output layer are irrelevant for this example. Therefore, the network states are defined as $\Sigma = \{\langle x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}\rangle \mid x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1} \in [0,1]\}$. Given, for instance, the neural network state $s = \langle 0,1,0,1\rangle$, the output observer ρ returns the state $\rho(s) = \langle 0,0,0,1\rangle$ to indicate that the target class is the second one, corresponding to $x_{1,1}$. Given the network state $\langle 0.5,1,0.7,0.2\rangle$, the output observer returns the state $\rho(s) = \langle 0,0,1,0\rangle$ to indicate that the target class is the first one (the first two variables are the input variables), and so on.

Note that the output observer always maps the values of neural network nodes that do not belong to the output layer to zero.

6.1.3 Neural Network Abstract Analysis

Abstract interpretation of a neural network model involves propagating an abstract element through each neuron of the network, over-approximating the possible states the model may reach. The analysis can be performed in two directions: forward, from the input layer to the output layer, or backward, from the output layer to the input layer. The networks considered in this thesis are acyclic, meaning that the abstract iterator always reaches the fixpoint after a finite number of iterations, without the need to traverse the same node multiple times. The forward analysis starts with an abstract element representing the input values and returns an abstract element representing an over-approximation of the reachable target classes. Conversely, the backward analysis begins with an abstract element representing the target classes and returns an abstract element representing an over-approximation of the input values that may lead to one of the given target classes.

When a neural network processes an input vector, it computes the value of each hidden node via an affine transformation followed by an activation function, in our case, the ReLU function. The ReLU function sets negative values to zero and leaves positive values unchanged. A node is said to be *active* if its value is positive and *inactive* otherwise. This flag is called the node's *activation status*. An *activation pattern* fixes the activation status of all nodes in the network, inducing a *path* in the neural network from regions in the input space where the network behaves similarly.

An *abstract activation pattern* is a partial activation pattern that assigns a fixed activation status only to a subset of ReLU nodes, thus representing a set of possible activation patterns. ReLU nodes with an unknown activation status are those whose corresponding flag does not have a fixed value. (Abstract) Activation patterns are valuable for the static analysis of networks as the knowledge of a node status can prune the search space, reducing computational cost. Specifically, the ReLU function is simpler to handle when the activation status is known: it behaves as the identity function for active nodes and as a constant zero for inactive nodes. Abstract activation patterns can be discovered from a forward

analysis of the network: if the over-approximation of input values to a node is greater than zero, the node is considered active; if less than zero, inactive; otherwise, unknown.

Given an abstract domain $\mathbb{D}$ with sound forward and backward operators for affine and ReLU operations, the forward analysis can be defined as a network computation using the abstract sound counterparts of the concrete operators. Algorithm 1 illustrates a forward analysis of neural network models, parameterized by the abstract domain $\mathbb{D}$ and an abstract element $d^\sharp$ representing the input values. The abstract operator $\overrightarrow{\texttt{Affine}}^\sharp$ computes the result $d^\sharp$ of the analysis of the affine transformation performed on a given node, while $\overrightarrow{\texttt{ReLU}}^\sharp_p$ additionally handles the ReLU activation function and determines the node's activation status p.

Conversely, a backward analysis of neural networks replaces concrete operators with their abstract sound *backward* counterparts. This analysis is also parameterized by the abstract domain $\mathbb{D}$ and an abstract element $d^\sharp$ representing the target classes. Algorithm 2 shows the backward analysis of neural network models, where the abstract operator $\overleftarrow{\texttt{Affine}}^\sharp$ computes the result $d^\sharp$ of the analysis of the backward affine transformation performed on a given node, and $\overleftarrow{\texttt{ReLU}}^\sharp_p$ manages the ReLU activation function given the node's activation status p.

Both analyses are sound, meaning they over-approximate the possible states the network may reach. Before formally showing the soundness of the forward and backward analyses, we introduce the forward concretization $\gamma^{\rightarrow}$. Similar to the backward concretization $\gamma^{\leftarrow}$, cf. Definition 4.2.1, it is defined as:

$$\gamma^{\rightarrow}(\Lambda^{\rightarrow})d^\sharp \stackrel{\text{def}}{=} \wp(\{\langle s_0, s_\omega\rangle \mid s_\omega \in \gamma(\Lambda^{\rightarrow}(d^\sharp)) \wedge s_0 \in \gamma(d^\sharp)\})$$

Lemma 6.1.1 (Soundness of Forward Neural Network Analysis) *Given a neural network model M and an abstract domain $\mathbb{D}$ with sound forward operators for affine and ReLU operations, the forward analysis of the network is sound whenever it holds that:*

$$\Lambda^\rho[\![M]\!]\|_{\gamma(d^\sharp)} \subseteq \gamma^{\rightarrow}(\texttt{Forward}^\sharp_{M,\mathbb{D}})d^\sharp$$

Proof. Trivially follows from the definition of the forward concretization and the soundness of the network operators. □

The next result instead shows the soundness of the backward analysis of neural networks.

Lemma 6.1.2 (Soundness of Backward Neural Network Analysis) *Given a neural network model M and an abstract domain $\mathbb{D}$ with sound backward operators for affine and ReLU operations, the backward analysis of the network is sound whenever it holds that:*

$$\Lambda^\rho[\![M]\!]|_{\gamma(d^\sharp)} \subseteq \gamma^{\leftarrow}(\texttt{Backward}^\sharp_{M,\mathbb{D}})d^\sharp$$

Algorithm 1: Forward analysis of neural networks.

```
Forward_{M,D}(d♯):
  p = ∅
  for i = 1 up to N do
    for j = 1 up to |L_i| do
      d♯ = Affine→♯ [[x_{i,j}]] d♯
      d♯, p = ReLU→♯_p [[x_{i,j}]] d♯
  return d♯, p
```

In the algorithm above (cf. $\texttt{Forward}^\sharp_{M,\mathbb{D}}$), N is the number of layers of the neural network M, $|L_i|$ is the number of nodes in layer i, and $x_{i,j}$ is the j-th node in layer i. Even though not explicitly written, the handler for affine and activation functions (respectively, $\overrightarrow{\texttt{Affine}}^\sharp$ and $\overrightarrow{\texttt{ReLU}}^\sharp_p$) employs the abstract domain $\mathbb{D}$.

Algorithm 2: Backward analysis of neural networks.

```
Backward_{M,D,p}(d♯):
  for i = N down to 1 do
    for j = |L_i| down to 1 do
      d♯ = ReLU←♯_p [[x_{i,j}]] d♯
      d♯ = Affine←♯ [[x_{i,j}]] d♯
  return d♯
```

Similarly to the notes of the forward analysis, in the algorithm above (cf. $\texttt{Backward}^\sharp_{M,\mathbb{D}}$), N is the number of layers of the neural network M, $|L_i|$ is the number of nodes in layer i, and $x_{i,j}$ is the j-th node in layer i. Even though not explicitly written, the backward handler for affine and activation functions (respectively, $\overleftarrow{\texttt{Affine}}^\sharp$ and $\overleftarrow{\texttt{ReLU}}^\sharp_p$) employs the abstract domain $\mathbb{D}$.

Def. 4.2.1 (Concretization $\gamma^{\leftarrow}$)

$$\gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j \stackrel{\text{def}}{=} \wp\left(\left\{\langle s_0, s_\omega\rangle \;\middle|\; \begin{array}{l} s_0 \in \gamma(\Lambda^{\leftarrow}(B_j)) \\ \wedge\, s_\omega \in \gamma(B_j) \end{array}\right\}\right)$$

The symbol $\Lambda^\rho[\![M]\!]\|_{\gamma(d^\sharp)}$ denotes the reduction of the semantics $\Lambda^\rho[\![M]\!]$ to the input states $\gamma(d^\sharp)$, cf. Definition 3.5.4.

Def. 3.5.4 (Input Reduction)

$$W\|_X \stackrel{\text{def}}{=} \{\{\langle s_0, s_\omega\rangle \in D \mid s_0 \in X\} \mid D \in W\}$$

The symbol $\Lambda^\rho[\![M]\!]|_{\gamma(d^\sharp)}$ denotes the reduction of the semantics $\Lambda^\rho[\![M]\!]$ to the output states $\gamma(d^\sharp)$, cf. Definition 3.5.3.

Proof. Trivially follows from the definition of the backward concretization and the soundness of the network operators. □

Def. 3.5.3 (Output Reduction)

$$W|_X \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \{\langle s_0, \rho(s_\omega)\rangle \in D \mid s_\omega \in X\} \\ \mid D \in W \end{array} \right\}$$

6.1.4 Abstract Domains for Neural Network Analysis

Different abstract domains can be used for the forward analysis of neural networks. The choice of the abstract domain depends on the trade-off between precision and scalability. Here, we present four abstract domains: Boxes [64, 65], Symbolic [115, 116], DeepPoly [117], and Neurify [118]. Additionally, we propose a generic reduced product domain construction, called Product, to combine any of these domains together.

[64]: Cousot et al. (1976), 'Static Determination of Dynamic Properties of Programs'
[65]: Hickey et al. (2001), 'Interval arithmetic: From principles to implementation'

Boxes The Boxes domain simply uses interval arithmetic [64, 65] to compute concrete lower and upper bound estimations l and u for the value of each neuron x in the neural network.

Symbolic Constant Propagation The Symbolic domain [115] combines Boxes with symbolic constant propagation [116]: in addition to being bounded by concrete lower and upper bounds, the value of each neuron x is represented symbolically as a linear combination of the input neurons and the value of the non-fixed ReLU nodes in previous layers. Specifically, given x bounded by $l < 0$ and $u > 0$, $\text{ReLU}(x)$ is represented by a fresh symbolic variable bounded by 0 and u (cf. Figure 6.1). By retaining variable dependencies, symbolic representations yield a tighter over-approximation of the value of each neuron in the network.

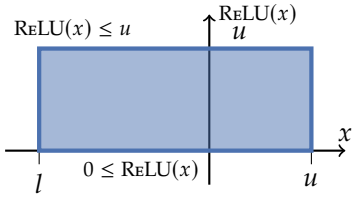


Figure 6.1: Naïve convex approximation of a ReLU activation function.

[115]: Wang et al. (2018), 'Formal Security Analysis of Neural Networks using Symbolic Intervals'
[116]: Miné (2006), 'Symbolic Methods to Enhance the Precision of Numerical Abstract Domains'

DeepPoly The DeepPoly domain [117] associates to each neuron x of a neural network concrete lower and upper bounds l and u as well as symbolic bounds expressed as linear combinations of neurons in the preceding layer of the network. The concrete bounds are computed by back-substitution of the symbolic bounds up to the input layer. Non-fixed ReLU nodes are overapproximated by partially retaining dependencies with preceding neurons using the tighter convex approximation between those shown in Figure 6.2 (*i.e.*, above when $u \leq -l$, and below otherwise).

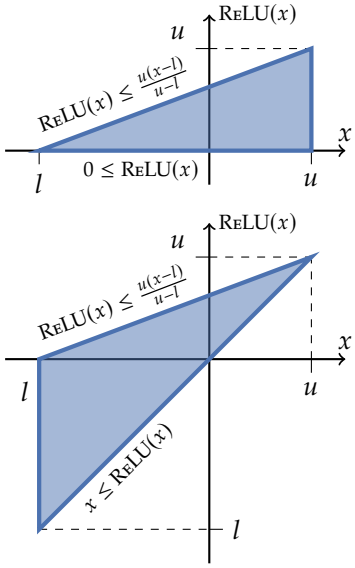


Figure 6.2: DeepPoly's convex approximation of a ReLU activation function.

Neurify The Neurify domain [118] similarly maintains symbolic lower and upper bounds low and up for each neuron x of neural network. Unlike DeepPoly, concrete lower and upper bounds are computed for *each* symbolic bound: l_{low} and u_{low} for the symbolic lower bound, and l_{up} and u_{up} for the symbolic upper bound. The over-approximation of non-fixed ReLU nodes is done *independently* for each symbolic bound, *i.e.*, for the low bound if $l_{low} < 0 < u_{low}$, and for the up bound if $l_{up} < 0 < u_{up}$. Figure 6.3 shows the approximation for $l = l_{low} = l_{up}$ and $u = u_{low} = u_{up}$. In general, the slope of the symbolic constraints will differ through successive approximation steps.

[117]: Singh et al. (2019), 'An abstract domain for certifying neural networks'

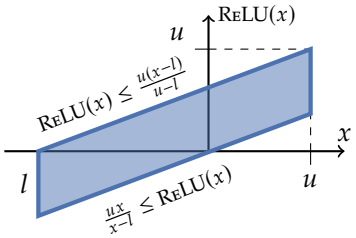


Figure 6.3: Neurify's convex approximation of a ReLU activation function.

[118]: Wang et al. (2018), 'Efficient Formal Safety Analysis of Neural Networks'

[47]: Mazzucato et al. (2021), 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'

[72]: Cousot et al. (1979), 'Systematic Design of Program Analysis Frameworks'

Reduced Product Finally, as part of the thesis contributions, the *Product Builder* [47] provides a parametric interface for constructing the product of the above domains [72]. The reduction function consists in an exchange of concrete bounds between domains. In particular, this allows determining tighter lower and upper bound estimations for each neuron in the network and thus reducing the over-approximation error introduced by the ReLU nodes. New abstract domains only need to implement the interface to share bounds information to enable their combination with other domains by the Product Builder.

6.2 Quantitative Impact Quantifiers

This section elaborates on quantifiers that can be used to measure the influence of input variables on the output of a neural network. Namely, we introduce the Changes and QAUnused impact quantifiers, which are specifically designed to handle neural network models.

6.2.1 The Changes Impact Quantifier

The Changes impact quantifier is designed to overcome the limitations of the impact quantifiers defined in the previous chapter when applied in the context of neural networks. Indeed, for neural networks, all possible input variations typically lead to all possible outcomes (*i.e.*, classifications). Therefore, the impact quantifiers defined in the previous chapter would not be useful as they measure the impact of input variables based on the number (*e.g.*, Outcomes) or the magnitude (*e.g.*, Range) of the output values; even QUsed would be meaningless in this setting as all the classification targets are often reachable by all the input values.

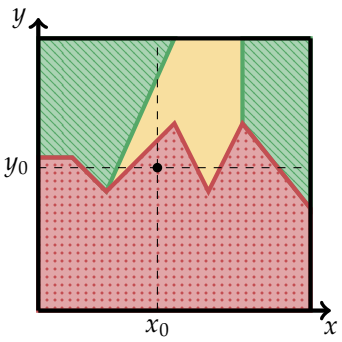


Figure 6.4: Input space with two input variables (x and y) and three possible outcomes (stripped green, plain yellow, dotted red).

Example 6.2.1 Consider a simple neural network with two input variables x and y, and three possible outcomes, denoted by the colors green, yellow, and red. A potential input space of such a model M is represented in Figure 6.4, where each outcome is reachable by the portion of the input space that is colored with the corresponding color. In this case, the Outcomes impact quantifier would not be useful as all the outcomes are reachable by perturbations of any input value. Consider for example the point (x_0, y_0) in the input space of outcome red, by applying a perturbation of one axis, the outcome can change to green or yellow, for both axes. Thus, the impact of both x and y is maximal, cf. $\text{Outcomes}_x([\![M]\!]) = \text{Outcomes}_y([\![M]\!]) = 3$. A similar reasoning applies to the Range or QUsed impact quantifiers, hence neither of them provides any insight.

On the other hand, we notice that one way to discriminate which input variable is more impactful is to consider the number of times the classification changes when the input variables are modified.

Example 6.2.2 Consider the input point (x_0, y_0) in the input space of the neural network in Figure 6.4. By applying a perturbation of the

x-axis, the outcome can change to green or yellow multiple times, a higher number of changes in the outcome is observed compared to the y-axis.

The Changes impact quantifier is designed to count how many times the network outcome changes by modification in the value of the input variables $\mathtt{W}$. In contrast with the previously defined Outcomes impact quantifier, in this case, we consider changes in the outcome *with repetitions*: if two different variations in $\mathtt{W}$ result in the same change in outcome, it counts as double change. The higher the number of changes in the outcome, the greater the influence on the program outcome. Therefore, this quantifier demonstrates its effectiveness when the same outcomes are reachable by multiple variations. Such situation often arises in the context of neural networks, where generally all the possible input variations lead to every possible outcome. The design of this quantifier builds on this observation. Thus, counting the repetitions is a potential solution to define a meaningful impact definition for neural networks.

In practice, the Changes impact definition considers variations in the value of input configurations that do not belong to the same *stable region*. In words, a stable region is a subset of the input space where the network output remains stable for all possible perturbations of the input variables. Figure 6.5 shows an example of a stable region (cf. region A) for the variable x, where the network output remains the same (cf. dotted red) for all possible variations of x. Instead, in the same figure, the region B is unstable for the variable x, where the network output changes for different variations of x (for example, both the outcomes of plain yellow and stripped green belong to this region). We first define the function Segments, which takes as input a set of traces T and an output value z. This function partitions the set of traces T into stable subsets with respect to the input variables $\mathtt{W}$. Each subset T' satisfies three conditions: (1) all the traces in T' share the same output value z, (2) for any two traces in T', there is no other trace in T with an input value between the two traces leading to a different output value, and (3) the subset is maximal, that is, there is no other trace in T that can be added to T' without violating the first two conditions. The function Segments is defined as follows:

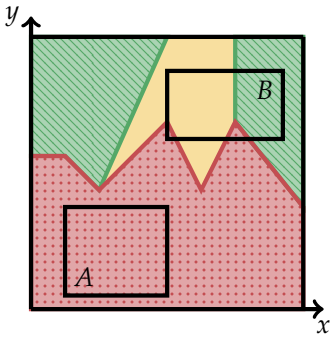


Figure 6.5: Example of stable (A) and unstable (B) regions for the variable x.

Definition 6.2.1 (Segments of Continuous Regions) *Let* $\mathtt{W} \in \wp(\Delta)$ *be the set of input variables of interest. Given a set of traces* $T \in \wp(\Sigma^{+\infty})$ *and an output value* $z \in \Sigma^{\perp}$*, the function* Segments $\in \wp(\Sigma^{+\infty}) \times \Sigma^{\perp} \to \wp(\wp(\Sigma^{+\infty}))$ *is defined as:*

$$\textsc{Segments}(T, z) \stackrel{\text{def}}{=} \{T' \in \phi(T) \mid \forall T'' \supset T'.\ T'' \notin \phi(T)\}$$

$$\text{where } \phi(T) \stackrel{\text{def}}{=} \left\{ T' \subseteq T \;\middle|\; \begin{array}{l} \forall \sigma \in T'.\ \rho(\sigma_\omega) = z \wedge \\ \forall \sigma, \sigma' \in T'.\ \exists \sigma'' \in T. \\ \quad \sigma_0(\mathtt{W}) \leq \sigma''_0(\mathtt{W}) \leq \sigma'_0(\mathtt{W}) \Rightarrow \sigma'' \in T' \end{array} \right\}$$

Note that, the auxiliary function ϕ partitions the set of traces T into subsets that satisfy the first two conditions: (1) all the traces in the subset share the same output value z, and (2) for any two traces in the subset (cf. $\rho(\sigma_\omega) = z$), there is no other trace in T with an input value between the two traces leading to a different output value (cf. $\sigma_0(\mathtt{W}) \leq \sigma''_0(\mathtt{W}) \leq \sigma'_0(\mathtt{W}) \Rightarrow \sigma'' \in T'$). The function Segments then returns the maximal subsets (3) that satisfy the first two conditions. To better

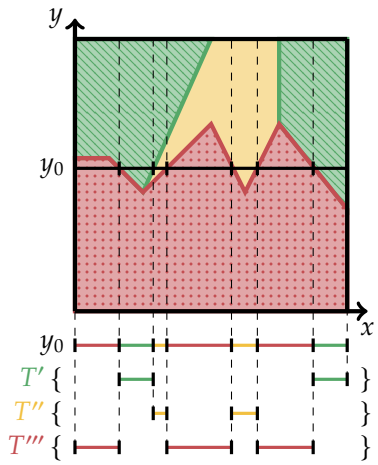


Figure 6.6: Function Segments.

illustrate how the function Segments works, consider the following example.

Example 6.2.3 Let us consider the set of traces from the neural network in Figure 6.4 with $y = y_0$, we call this set T. The function Segments partitions the set of traces T into three subsets, each containing the traces leading to the same output value. Figure 6.6 shows the three partitions graphically, respectively for the outcome green (T'), yellow (T''), and red (T''').

Formally, Changes is defined as the maximum, for each input configuration, of the number of stable regions in which the output value changes. In words, a set of input variables $\mathtt{W}$ has a high impact on the network outcome if the output value changes in many stable regions when the input variables are modified.

Definition 6.2.2 (Changes) *Given a set of input variables of interest* $\mathtt{W} \in \wp(\Delta)$*, and an output observer* ρ*, the quantity* $\textsc{Changes}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{N}^{+\infty}$ *is defined as:*

$$\textsc{Changes}_{\mathtt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \sup_{z \in \Sigma^\perp} | Q_{\mathtt{W},s_0,z} |$$

$$\text{where } Q_{\mathtt{W},s_0,z} \stackrel{\text{def}}{=} \bigcup_{z' \in \Sigma^\perp \setminus \{z\}} \textsc{Segments}(\{\sigma \in T \mid \sigma_0 =_{\Delta \setminus \mathtt{W}} s_0\}, z')$$

The auxiliary set $Q_{\mathtt{W},s_0,z}$ contains the set of stable regions leading to an output value different from z. Additionally, $\textsc{Changes}_{\mathtt{W}}$ requires the set of traces T to be deterministic, that is, that there is only one output value for each input configuration. Such a requirement is necessary to ensure that segments are well-defined. Otherwise, the Changes impact quantifier would potentially be meaningless in such a setting. Nevertheless, since we consider neural networks for classification tasks in this chapter, the output value is deterministic for each input configuration, and thus the set of traces T is deterministic.

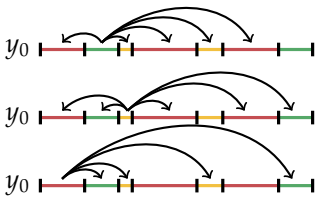


Figure 6.7: Function Changes.

Example 6.2.4 Figure 6.7 shows the Changes impact quantifier applied to the input space of the neural network in Figure 6.4. The function $\textsc{Changes}_{\mathtt{W}}$ returns the maximum number of stable regions in which the output value changes when the input variables are modified. As illustrated, for the variable x, this can only happen when $y = y_0$ as all the other values of y lead to the same or fewer changes. Consider the outcomes green, yellow, and red (graphically represented by the three figures in Figure 6.7, top to bottom), modifying the input variable x may lead to 5, 5, and 4 different outcomes, respectively. Therefore, the Changes impact quantifier for the input variable x is 5.

Lemma 6.2.1 (Changes is Monotonic) *For all set of traces* $T, T' \in \wp(\Sigma^{+\infty})$*, it holds that:*

$$T \subseteq T' \implies \textsc{Changes}_{\mathtt{W}}(T) \leq \textsc{Changes}_{\mathtt{W}}(T')$$

Proof. The proof is based on the observation that more traces in T' can only increase the number of stable regions. Hence, the number of stable regions leading to a different output (cf. $Q_{\mathtt{W},s_0,z}$) can only increase with more traces. We conclude that $\textsc{Changes}_{\mathtt{W}}(T) \leq \textsc{Changes}_{\mathtt{W}}(T')$. □

We show the validation of the k-bounded impact property when instantiated with the Changes impact quantifier, denoted $\mathscr{B}^{\otimes k}_{\textsc{Changes}_{\mathtt{W}}}$.

Lemma 6.2.2 ($\mathscr{B}^{\otimes k}_{\textsc{Changes}_{\mathtt{W}}}$ Validation)

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathscr{B}^{\otimes k}_{\textsc{Changes}_{\mathtt{W}}} \;\Leftrightarrow\; \Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathscr{B}^{\otimes k}_{\textsc{Changes}_{\mathtt{W}}}))$$

Def. 3.5.2 (Output-Abstraction Semantics)

$$\Lambda^{\rho} \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \rho(\sigma_\omega)\rangle \mid \sigma \in \Lambda\}\}$$

Proof (Sketch). The Changes impact quantifier does not consider the intermediate states, in fact, it only employs the first state in the definition of $Q_{\mathtt{W},s_0,z}$ and the last one in the definition of Segments. Thus, the abstraction to dependencies does not affect the validation of the property. Furthermore, even handling the output abstraction at the semantic level, by abstracting output states to abstract output states, does not affect the validation of the property as the Changes impact quantifier already abstracts the output values before comparing to the given output z, cf. $\rho(\sigma_\omega) = z$ in Definition 6.2.1 (Segments of Continuous Regions). □

6.2.2 The QAUnused Impact Quantifier

The second quantifier introduced in this section is the QAUnused impact quantifier, which stands for quantitative abstract unused. Notably, this quantifier is designed to quantify the amount of neural network's input space that *does not use* a given set of input features. In the context of neural networks, the QAUnused impact quantifier can be used to induce a quantification of the fair space of a model. A network is fair whenever the classification determined by a model does not depend on the "sensitive" input variables. In our setting, the sensitive input variables are represented by the set of input variables $\mathtt{W}$ and measuring the inability of the input variables $\mathtt{W}$ to affect the network outcome is equivalent to quantifying the amount of fair space. To this end, we define the QAUnused impact quantifier as the volume of input space that is not able to change the model classification by perturbation of the input variables $\mathtt{W}$. The higher the volume the higher the fairness of the model, and thus the lower the amount of space that is prone to bias.

In practice, we collect the input space (without the input variables $\mathtt{W}$ to account for any possible permutations of their input values) where the variables $\mathtt{W}$ do not influence the network outcome. To do so, we employ the $\text{AUnused}_{\mathtt{W}}$ predicate, cf. Definition 3.2.3, to check whether a subset of the given set of traces T is not able to change the network outcome. We determine the volume of this set of points by applying the standard volume operation inherent to metric spaces. Formally, the QAUnused impact quantifier is defined as follows:

Def. 3.2.3 (Abstract Unused)

$$\begin{aligned}&\text{AUnused}_{\mathtt{W}}(\Lambda[\![P]\!]) \stackrel{def}{\Leftrightarrow}\\ &\quad \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|\mathtt{W}|}.\ \sigma_0(\mathtt{W}) \neq v \;\Rightarrow\\ &\quad \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0 \wedge\\ &\qquad \sigma'_0(\mathtt{W}) = v \wedge\\ &\qquad \rho(\sigma_\omega) = \rho(\sigma'_\omega)\end{aligned}$$

Definition 6.2.3 (QAUnused) *Given a set of input variables of interest* $\mathtt{W} \in \wp(\Delta)$*, and an output descriptor* ρ*, the quantity* $\text{QAUnused}_{\mathtt{W}} \in \wp(\Sigma^{+\infty}) \to$

The input space of neural networks is assumed to be normalized in the box $[0,1]$ for each input variable. As a consequence, the maximum volume of the input space is 1.

$[0,1]$ *is defined as:*

$$\text{QAUNUSED}_{\mathtt{W}}(T) = \text{VOLUME}(\{\sigma_0(\Delta \setminus \mathtt{W}) \mid \text{AUNUSED}_{\mathtt{W}}(\{\sigma' \in T \mid \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0\})\})$$

In other words, the QAUNUSED quantifies the volume of the biggest set of traces $T' \subseteq T$ that does not contain bias, *i.e.*, where the predicate $\text{AUNUSED}_{\mathtt{W}}$ holds.

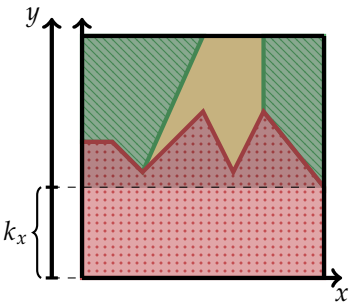


Figure 6.8: Function QAUNUSED.

Example 6.2.5 Consider the network of example presented in Figure 6.4. The QAUNUSED impact quantifier quantifies the volume of the input space where the input variable x does not influence the network outcome. In Figure 6.8, it is clear that the input variable x does not influence the network outcome when y is below a certain threshold. This volume of fair input space is denoted by k_x in Figure 6.8.

As noticed for the previous impact quantifier CHANGES, neural network models produce deterministic set of traces. Hence, we have that by adding more traces to a given (deterministic) set of traces, the volume of the fair space can only decrease as more traces are able to change the network outcome.

Lemma 6.2.3 (QAUNUSED is Anti-Monotonic) *For all deterministic set of traces* $T, T' \in \wp(\Sigma^{+\infty})$*, it holds that:*

$$T \subseteq T' \implies \text{QAUNUSED}_{\mathtt{W}}(T) \geq \text{QAUNUSED}_{\mathtt{W}}(T')$$

Proposition 3.2.2 *If P is deterministic and the input abstraction is defined as:*

$$\eta_{\mathtt{W}}(s) = \lambda j. \begin{cases} s(j) & \text{if } j \notin \mathtt{W} \\ \top & \text{otherwise} \end{cases}$$

then, it holds that:

$$\text{ANI}(\Lambda[\![P]\!]) \Leftrightarrow \text{AUNUSED}_{\mathtt{W}}(\Lambda[\![P]\!])$$

Proof. The proof is based on the observation that more traces in T' can only increase the volume of bias space as, in the worst case scenario, the added traces are the ones able to change the network outcome. In fact, Proposition 3.2.2 proves that the $\text{AUNUSED}_{\mathtt{W}}$ predicate is equivalent to the ANI predicate whenever the given set of traces is deterministic, which is anti-monotonic in the amount of traces. Hence, the volume of the set of points leading to different output values can only increase with more traces, hence the volume of fair space decreases. We conclude that $\text{QAUNUSED}_{\mathtt{W}}(T) \geq \text{QAUNUSED}_{\mathtt{W}}(T')$. □

Def. 3.5.2 (Output-Abstraction Semantics)

$$\Lambda^{\rho} \stackrel{\text{def}}{=} \{\{\langle \sigma_0, \rho(\sigma_\omega)\rangle \mid \sigma \in \Lambda\}\}$$

We show the validation of the k-bounded impact property when instantiated with the QAUNUSED impact quantifier, denoted $\mathscr{B}^{\otimes k}_{\text{QAUNUSED}_{\mathtt{W}}}$.

Lemma 6.2.4 ($\mathscr{B}^{\otimes k}_{\text{QAUNUSED}_{\mathtt{W}}}$ Validation)

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathscr{B}^{\otimes k}_{\text{QAUNUSED}_{\mathtt{W}}} \Leftrightarrow \Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathscr{B}^{\otimes k}_{\text{QAUNUSED}_{\mathtt{W}}}))$$

Thm. 3.5.2

$$\Lambda^{\mathbb{C}} \subseteq \mathscr{N}_{\mathtt{W}} \Leftrightarrow \Lambda^{\rho} \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathscr{N}_{\mathtt{W}}))$$

Proof. The QAUNUSED impact quantifier employs the $\text{AUNUSED}_{\mathtt{W}}$ predicate to determine the volume of the input space that does not contain bias. Thus, this proof directly follows from Theorem 3.5.2. □

6.3 Quantitative Analysis of Neural Networks

This section presents the abstract implementations of the Changes and QAUnused impact quantifiers, respectively called $\texttt{Changes}^\natural$ and $\texttt{QAUnused}^\natural$. We show how to validate the k-bounded impact property for both quantifiers.

6.3.1 Abstract Implementation $\texttt{Changes}^\natural_{\texttt{W}}$

In this section, we introduce $\texttt{Changes}^\natural_{\texttt{W}}$ as a sound implementation of $\textsc{Changes}_{\texttt{W}}$. First, we describe the implementation, followed by the validation of the k-bounded impact property $\mathscr{B}^{\circledS k}_{\textsc{Changes}_{\texttt{W}}}$.

As previously noted, the Changes impact quantifier is ineffective when the input is a non-deterministic set of traces. The backward analysis, starting from the output buckets, computes an over-approximation of the set of traces leading to the same output bucket. If this over-approximation overlaps with the results of the abstract backward analysis from different output buckets, it concretizes to a non-deterministic set of traces. To address this issue, we require the backward analysis to be exact for our property and underlying network model.

The core idea of the Changes impact quantifier is to group together abstract elements representing distinct stable regions. Convex abstract domains, such as polyhedra, octagons, or intervals (introduced in Section 2.3.5), are inadequate for representing multiple disjoint stable regions because they also include the points in between. Therefore, the abstract domain $\mathbb{D}$ used in the backward analysis $\Lambda^{\leftarrow}$ needs to employ disjunctive sets, which allow for the representation of distinct stable regions. We leverage the *disjunctive polyhedra abstract domain* $\mathbb{DP}$, defined as $\langle \wp_{\text{finite}}(\mathbb{P}), \subseteq^{\mathbb{DP}} \rangle$, where $\mathbb{P}$ represents the *convex polyhedra abstract domain* [56] and $\wp_{\text{finite}}(\mathbb{P})$ is the set of all finite subset of $\mathbb{P}$. From the polyhedra domain $\mathbb{P}$, the function $\texttt{Proj}_{\texttt{W}}$ uses the Fourier-Motzkin elimination algorithm [119] to project away the input variables $\texttt{W}$. Specifically, $\texttt{Proj}_{\texttt{W}}$ takes as input a set of variables $\texttt{W}$ and a polyhedron in d-dimensions where $d \geq |\texttt{W}|$, returning a polyhedron in $(d - |\texttt{W}|)$-dimensions, removing the variables in $\texttt{W}$. This domain is exact for the property of interest when applied to feed-forward ReLU-activated neural networks.

[56]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'

[119]: Dantzig et al. (1973), 'Fourier-Motzkin Elimination and Its Dual'

The function $\texttt{Changes}^\natural_{\texttt{W}}$ takes as input the variables of interest $\texttt{W}$, n output buckets $B \in \mathbb{D}^n$, and n disjunctions of polyhedra $X_j \in \wp_{\text{finite}}(\mathbb{P})^n$ obtained from the backward analysis. For clarity, we access each polyhedron and disjunctions of polyhedra via indexing as in a matrix-based structure, that is, $X_j = \{\{P_{1,1} \vee \cdots \vee P_{1,p}\}, \ldots, \{P_{n,1} \vee \cdots \vee P_{n,q}\}\}$ where $p = |X_1|$ and $q = |X_n|$. For instance, X_j refers to the disjunction of polyhedra $\{P_{j,1} \vee \cdots \vee P_{j,k} \vee \ldots\}$, for $j \leq n$, and $X_{j,k}$ refers to the polyhedron $P_{j,k} \in \mathbb{P}$. The projection of disjunctions of polyhedra applies, in turn, the polyhedron projection to each polyhedron in the disjunction, then collects the projected polyhedra in a disjunction of polyhedra, *i.e.*, $\texttt{Proj}_{\texttt{W}}(X_j \in \wp_{\text{finite}}(\mathbb{P})) \stackrel{\text{def}}{=} \bigvee_{k \leq |X_j|} \texttt{Proj}_{\texttt{W}}(X_{j,k})$.

Computationally speaking, the function $\texttt{Changes}^\natural_{\texttt{W}}$ projects away the input variables $\texttt{W}$ from each polyhedron $X_{j,k}$. The projected polyhedra represent regions where $\texttt{W}$ ranges on all possible values, considering

all potential variations of this variable. The function `InterAll` gathers the set of indexes J, also called the connected components, where the projected polyhedra intersect. The underlying idea is that each connected component corresponds to the set of stable regions reachable through variations of $\mathtt{W}$. Finally, $\mathtt{Changes}^{\natural}_{\mathtt{W}}$ determines the maximum count of changes across all connected components J and buckets B. It counts the number of indices l (cf. $|\{l \in J \mid l \neq j\}|$) in each connected component J where k is not equal to j, thus excluding the polyhedra leading to the same output bucket j.

Definition 6.3.1 ($\mathtt{Changes}^{\natural}_{\mathtt{W}}$) *We define* $\mathtt{Changes}^{\natural}_{\mathtt{W}} \in \mathbb{D}^n \times \mathbb{D}^n \to \mathbb{V}^{+\infty}_{\geq 0}$ *as:*

$$\mathtt{Changes}^{\natural}_{\mathtt{W}}(X, B) \stackrel{\text{def}}{=} \max\{\max_{j \leq n} |\{l \in J \mid l \neq j\}| \mid J \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(X_j))_{j \leq n})\}$$

Before proceeding with the soundness proof of $\mathtt{Changes}^{\natural}_{\mathtt{W}}$, we recall the requirements on the concrete impact quantifier $\textsc{Changes}_{\mathtt{W}}$, namely, the requirement on determinism. Such a requirement is necessary to ensure a meaningful quantity of changes in the outcome, otherwise the number of changes would be infinite in the presence of stable regions with non-deterministic output values. Equivalently, in the abstract, any over-approximation from the backward analysis that overlaps different preconditions X_j would lead to an infinite number of changes. To this end, we require the backward analysis to be exact: sound and *complete*. The soundness ensures that the backward analysis does not miss any possible behavior, while the completeness ensures that the backward analysis does not introduce any spurious behavior. In addition to Definition 4.2.2, we require the following completeness condition on the backward analysis:

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^{\rho}|_{\gamma(B_j)} \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Definition 6.3.2 (Complete Under-Approximation) *For all programs P, and output bucket $B_j \in \mathbb{D}$, the family of semantics $\Lambda^{\leftarrow}$ is a* complete under-approximation *of the output-abstraction semantics $\Lambda^{\rho}[\![P]\!]$ when it holds that:*

$$\Lambda^{\rho}[\![P]\!]|_{\gamma(B_j)} \supseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}[\![P]\!])B_j$$

Whenever the backward semantics $\Lambda^{\leftarrow}[\![P]\!]$ is both sound and complete, it yields an exact semantics.

Lemma 6.3.1 (Exact Backward Semantics) *Whenever the backward semantics $\Lambda^{\leftarrow}[\![P]\!]$ is sound and complete, it holds that:*

$$\Lambda^{\rho}[\![P]\!]|_{\gamma(B_j)} = \gamma^{\leftarrow}(\Lambda^{\leftarrow}[\![P]\!])B_j$$

Proof. Trivially follows from the definitions of soundness (Definition 4.2.2) and completeness (Definition 6.3.2). □

Note that, whenever the given program is deterministic, the traces concretized from the backward analysis are deterministic as well. With the exactness of the backward analysis, we show that $\mathtt{Changes}^{\natural}_{\mathtt{W}}$ is a sound

implementation of $\textsc{Changes}_{\mathtt{W}}$. The next result shows that the abstract impact $\mathtt{Changes}^{\natural}_{\mathtt{W}}$ is a sound over-approximation of the concrete $\textsc{Changes}_{\mathtt{W}}$ w.r.t. the $\leq$ operator.

Lemma 6.3.2 ($\mathtt{Changes}^{\natural}_{\mathtt{W}}$ is a Sound Implementation of $\textsc{Changes}_{\mathtt{W}}$) *Let $\mathtt{W} \in \wp(\Delta)$ be the set of input variables of interest, $\mathbb{D}$ the abstract domain, $\Lambda^{\leftarrow}$ the family of semantics, and $B \in \mathbb{D}^n$ the starting output buckets. Whenever the following conditions hold:*

(i) B covers the subset of potential outcomes, cf. Definition 4.2.5,
(ii) B is compatible with ρ, cf. Definition 4.2.11,
(iii) $\Lambda^{\leftarrow}$ is an exact backward semantics, cf. Lemma 6.3.1, and
(iv) $\mathtt{Proj}_{\mathtt{W}}$ is sound, cf. Definition 4.2.8;

then, $\mathtt{Changes}^{\natural}_{\mathtt{W}}$ is a sound implementation of $\textsc{Changes}_{\mathtt{W}}$ w.r.t. the $\geq$ operator:

$$\textsc{Changes}_{\mathtt{W}}(\Lambda) \;\leq\; \mathtt{Changes}^{\natural}_{\mathtt{W}}(\gamma^{\times}(\Lambda^{\times})B, B)$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$

Def. 4.2.11 (Compatibility)

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p). \; \rho(s_j) \neq \rho(s_p) \;\Rightarrow\; B_j \neq B_p$$

Def. 4.2.8 (Sound Projection Operator)

$$\{s' \in \Sigma^{\perp} \mid s \in \gamma(s^{\natural}) \wedge s =_{\Delta\setminus\mathtt{W}} s'\} \subseteq \gamma(\mathtt{Proj}_{\mathtt{W}}(s^{\natural}))$$

Proof (Sketch). We need to prove that

$$\textsc{Changes}_{\mathtt{W}}(\Lambda) = \sup_{s_0\in\Sigma|_{\Delta}} \sup_{z\in\Sigma^{\perp}} |\, Q_{\mathtt{W},s_0,z} \,|$$
$$\leq$$
$$\mathtt{Changes}^{\natural}_{\mathtt{W}}(\gamma^{\times}(\Lambda^{\times})B, B) =$$
$$\max\{\max_{j\leq n} |\{l \in J \mid l \neq j\}| \mid J \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(X_j))_{j\leq n})\}$$

where $Q_{\mathtt{W},s_0,z} = \cup_{z'\in\Sigma^{\perp}\setminus\{z\}}\textsc{Segments}(\{\sigma \in T \mid \sigma_0 =_{\Delta\setminus\mathtt{W}} s_0\}, z')$. Let $\overline{s_0} \in \Sigma|_{\Delta}$ and $\overline{z} \in \Sigma^{\perp}$ be such that:

$$\sup_{s_0\in\Sigma|_{\Delta}} \sup_{z\in\Sigma^{\perp}} |\, Q_{\mathtt{W},s_0,z} \,| \;=\; |\, Q_{\mathtt{W},\overline{s_0},\overline{z}} \,|$$

Let $\overline{J} \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(X_j))_{j\leq n})$ be such that:

$$\max\{\max_{j\leq n} |\{l \in J \mid l \neq j\}| \mid J \in \mathtt{InterAll}((\mathtt{Proj}_{\mathtt{W}}(X_j))_{j\leq n})\} =$$
$$\max_{j\leq n} |\{l \in \overline{J} \mid l \neq j\}|$$

thus, we need to show that $|\, Q_{\mathtt{W},\overline{s_0},\overline{z}} \,| \leq \max_{j\leq n} |\{l \in \overline{J} \mid l \neq j\}|$. Provided that B is compatible with ρ and covers the subset of potential outcomes, respectively (i) and (ii), an implication of the exactness assumption (cf. (iii)) of the backward analysis $\Lambda^{\leftarrow}$ is any stable region intersecting after projecting away the variables $\mathtt{W}$ is represented in the abstract by two indices j and k in the connected components (from (iv)). Hence, under these assumptions, the number of stable regions leading to different output buckets has to be lower the number of intersecting buckets in the abstract, *i.e.* $|\, Q_{\mathtt{W},\overline{s_0},\overline{z}} \,| \leq \max_{j\leq n} |\{l \in \overline{J} \mid l \neq j\}|$, proving that $\mathtt{Changes}^{\natural}_{\mathtt{W}}$ is a sound implementation of $\textsc{Changes}_{\mathtt{W}}$. □

Next, we show Theorem 4.2.4 (Soundness) instantiated with the abstract implementation $\mathtt{Changes}^{\natural}_{\mathtt{W}}$ and the comparison operator $\leq$.

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^\perp\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j \leq n} \gamma(B_j)\}$$

Def. 4.2.11 (Compatibility)

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p). \; \rho(s_j) \neq \rho(s_p) \implies B_j \neq B_p$$

Def. 4.2.8 (Sound Projection Operator)

$$\{s' \in \Sigma^\perp \mid s \in \gamma(s^\sharp) \wedge s =_{\Delta \setminus \mathtt{W}} s'\} \subseteq \gamma(\mathtt{Proj}_{\mathtt{W}}(s^\sharp))$$

Theorem 6.3.3 (Soundness of $\mathscr{B}^{\leq k}_{\textsc{Changes}_{\mathtt{W}}}$) *Let $\mathscr{B}^{\leq k}_{\textsc{Changes}_{\mathtt{W}}}$ be the property of interest we want to verify for the program P and the input variable $\mathtt{W} \in \wp(\Delta)$. Whenever,*

(i) B covers the subset of potential outcomes, cf. Definition 4.2.5,
(ii) B is compatible with ρ, cf. Definition 4.2.11,
(iii) $\Lambda^\leftarrow$ is an exact backward semantics, cf. Lemma 6.3.1, and
(iv) $\mathtt{Proj}_{\mathtt{W}}$ is sound, cf. Definition 4.2.8;

the following implication holds:

$$\mathtt{Changes}^\sharp_{\mathtt{W}}(\Lambda^\times[\![P]\!]B, B) = k' \wedge k' \leq k \implies P \models \mathscr{B}^{\leq k}_{\textsc{Changes}_{\mathtt{W}}}$$

Proof. Lemma 6.3.2 shows that $\mathtt{Changes}^\sharp_{\mathtt{W}}$ is a sound implementation of $\textsc{Changes}_{\mathtt{W}}$, the proof follows directly by application of the Theorem 4.2.4 instantiated with the abstract implementation $\mathtt{Changes}^\sharp_{\mathtt{W}}$ and comparison operator $\leq$. □

While this result shows that an exact backward analysis is required to ensure a sound implementation of $\textsc{Changes}_{\mathtt{W}}$, it also presents scalability challenges as the backward analysis is computationally expensive and requires exploring all the possible paths of the network. In the next section, we introduce the QA$\textsc{Unused}$ impact quantifier, which is more scalable and efficient thanks to the use or parallelization.

6.3.2 Abstract Implementation $\mathtt{QAUnused}^\sharp_{\mathtt{W}}$

In this section, we present $\mathtt{QAUnused}^\sharp_{\mathtt{W}}$ as a sound implementation of QA$\textsc{Unused}_{\mathtt{W}}$, computing an over-approximation of the volume of input space that may contain bias. Conversely, $\mathtt{QAUnused}^\sharp_{\mathtt{W}}$ computes an under-approximation of the fair input space. We first present the implementation of $\mathtt{QAUnused}^\sharp_{\mathtt{W}}$, which is too naïve to be practical, but it is still useful for building upon in the parallel implementation $\mathtt{QAUnused}^\sharp_{\mathtt{W}}$ later in this chapter.

Definition 6.3.3 ($\mathtt{QAUnused}^\sharp_{\mathtt{W}}$) *We define $\mathtt{QAUnused}^\sharp_{\mathtt{W}} \in \mathbb{D}^n \times \mathbb{D}^n \to \mathbb{V}^{+\infty}_{\geq 0}$ as:*

$$\mathtt{QAUnused}^\sharp_{\mathtt{W}}(X, B) \stackrel{\text{def}}{=} 1 - \mathtt{Volume}\left(\bigsqcup \left\{ \mathtt{Proj}_{\mathtt{W}}(X_j) \sqcap \mathtt{Proj}_{\mathtt{W}}(X_k) \;\middle|\; j, k \leq n \wedge j \neq k \right\}\right)$$

The output buckets B represent all the possible classification targets of the neural network. In this way, both covering (Definition 4.2.5) and compatibility (Definition 4.2.11) conditions are satisfied as all the target classes are covered and for any two different target classes, there exist two different output buckets representing them. The analysis proceeds backwards from each output bucket B_j via the backward semantics $\Lambda^\leftarrow$ in order to determine an over-approximation of the initial states X_j. $\mathtt{QAUnused}^\sharp_{\mathtt{W}}$ projects away the input variables $\mathtt{W}$ from the abstract states X_j to account for any possible permutations of their input values. Then, it computes the pairwise intersections among the projected abstract states to find the portion of input space leading to different output buckets only

via variations of the input variables $\mathtt{W}$. The volume of this input space is an over-approximation of the biased space, hence its complement is an under-approximation of the fair input space. We assume the classical soundness condition on the abstract operator `Volume` to ensure that the abstract volume is always bigger than the concrete one. The quantity measured by $\mathtt{QAUnused}^{\natural}_{\mathtt{W}}$, cf. Definition 6.3.3, is always lower than the concrete $\text{QAU}_{\text{NUSED}_{\mathtt{W}}}$, cf. Definition 6.2.3, as it measures the volume of the input space that contains bias.

Def. 6.2.3 (QAUnused)

$$\text{QAU}_{\text{NUSED}_{\mathtt{W}}}(T) = \text{VOLUME}\left(\left\{\sigma_0(\Delta \setminus \mathtt{W}) \;\middle|\; \text{AU}_{\text{NUSED}_{\mathtt{W}}}(T_{\sigma_0})\right\}\right)$$

where $T_{\sigma_0} = \{\sigma' \in T \mid \sigma'_0 =_{\Delta\setminus\mathtt{W}} \sigma_0\}$.

Lemma 6.3.4 ($\mathtt{QAUnused}^{\natural}_{\mathtt{W}}$ *is a Sound Implementation of* $\text{QAU}_{\text{NUSED}_{\mathtt{W}}}$) *Let* $\mathtt{W} \in \wp(\Delta)$ *be the set of input variables of interest,* $\mathbb{D}$ *the abstract domain,* $\Lambda^{\leftarrow}$ *the family of semantics, and* $B \in \mathbb{D}^n$ *the starting output buckets. Whenever the following conditions hold:*

(i) B *covers the subset of potential outcomes, cf. Definition 4.2.5,*
(ii) B *is compatible with* ρ*, cf. Definition 4.2.11,*
(iii) $\Lambda^{\leftarrow}$ *is a sound backward semantics, cf. Definition 4.2.2,*
(iv) $\mathtt{Proj}_{\mathtt{W}}$ *is sound, cf. Definition 4.2.8, and*
(v) `Volume` *is sound;*

then, $\mathtt{QAUnused}^{\natural}_{\mathtt{W}}$ *is a sound implementation of* $\text{QAU}_{\text{NUSED}_{\mathtt{W}}}$ *w.r.t. the* $\geq$ *operator:*

$$\text{QAU}_{\text{NUSED}_{\mathtt{W}}}(\Lambda) \;\geq\; \mathtt{QAUnused}^{\natural}_{\mathtt{W}}(\gamma^{\times}(\Lambda^{\times})B, B)$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$

Def. 4.2.11 (Compatibility)

$$\forall s_j \in \gamma(B_j), s_p \in \gamma(B_p).\; \rho(s_j) \neq \rho(s_p) \;\Rightarrow\; B_j \neq B_p$$

Proof (Sketch). The soundness argument to prove this lemma is based on the soundness of the operators involved in the computation of $\mathtt{QAUnused}^{\natural}_{\mathtt{W}}$, namely (iii), (iv) and (v), the fact that the buckets B cover the subset of potential outcomes (i) and are compatible with the output observations (ii). The proof follows by showing that $\mathtt{QAUnused}^{\natural}_{\mathtt{W}}$ is the complement of the volume of biased space, computed by first projecting away the input variables $\mathtt{W}$ from all the given abstract values resulting from the backward analysis, then collecting all the possible pairwise intersections among the projected abstract values to find the portion of input space leading to, at least, two different output buckets. By the fact that all the operators involved are sound over-approximations (cf. (iv) and (v)) as well as the backward analysis (cf. (iii)), the proof follows directly. As the volume computed in the abstract over-approximates the biased space, its complement under-approximates the fair input space. □

Def. 4.2.8 (Sound Projection Operator)

$$\{s' \in \Sigma^{\perp} \mid s \in \gamma(s^{\natural}) \wedge s =_{\Delta\setminus\mathtt{W}} s'\} \subseteq \gamma(\mathtt{Proj}_{\mathtt{W}}(s^{\natural}))$$

We show Theorem 4.2.4 (Soundness) instantiated with the abstract implementation $\mathtt{QAUnused}^{\natural}_{\mathtt{W}}$ and the comparison operator $\geq$.

Theorem 6.3.5 (Soundness of $\mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$) *Let* $\mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$ *be the property of interest we want to verify for the program* P *and the input variable* $\mathtt{W} \in \wp(\Delta)$*. Whenever,*

(i) $\Lambda^{\leftarrow}$ *is sound with respect to* Λ^{ρ}*, cf. Definition 4.2.2, and*
(ii) B *covers the subset of potential outcomes, cf. Definition 4.2.5,*

the following implication holds:

$$\mathtt{QAUnused}^{\natural}_{\mathtt{W}}(\Lambda^{\times}[\![P]\!]B, B) = k' \wedge k' \geq k \;\Rightarrow\; P \models \mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$$

Def. 4.2.2 (Sound Over-Approximation for $\Lambda^{\leftarrow}$)

$$\Lambda^{\rho}|_{\gamma(B_j)} \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow})B_j$$

Def. 4.2.5 (Covering)

$$\{\rho(s_\omega) \mid s_\omega \in \Sigma^{\perp}\} \subseteq \{\rho(s_\omega) \mid s_\omega \in \bigcup_{j\leq n} \gamma(B_j)\}$$

Proof. Lemma 6.3.4 shows that $\texttt{QAUnused}^{\sharp}_{\mathbb{W}}$ is a sound implementation of $\text{QAU}_{\text{NUSED}_{\mathbb{W}}}$, the proof follows directly by application of the Theorem 4.2.4 instantiated with the abstract implementation $\texttt{QAUnused}^{\sharp}_{\mathbb{W}}$ and comparison operator $\leq$. □

Although sound, the abstract implementation $\texttt{QAUnused}^{\sharp}_{\mathbb{W}}$ does not work well in presence of imprecision from the abstract domain employed during the backward analysis. In fact, the abstract implementation $\texttt{QAUnused}^{\sharp}_{\mathbb{W}}$ applied directly to the backward analysis yields the so called *naïve casual-fairness analysis* [73]. Such analysis suffers from the choice of existing abstract domains, which are rather fast but too imprecise to handle non-linear constraints, such as those arising from the activation functions in neural networks. Indeed, even using the polyhedra domain for the backward analysis, handling the ReLU activation function would over-approximate what effectively is a conditional branch, leading to a loss of precision that is reflected for each node of the neural network. On the other hand, one could use a disjunctive completion of the polyhedra domain [72], which would retain a separate polyhedron each condition. However, this analysis would be extremely slow.

[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

[72]: Cousot et al. (1979), 'Systematic Design of Program Analysis Frameworks'

6.4 Parallel Analysis for Efficient Validation of the $\mathcal{B}^{\otimes k}_{\text{QAU}_{\text{NUSED}_{\mathbb{W}}}}$ Property

To overcome the limitation of $\texttt{QAUnused}^{\sharp}_{\mathbb{W}}$ described above, we first reason at a concrete-semantics level, introducing an additional semantics: the *parallel semantics*. Intuitively, we show how partitioning the input space into *fair* partitions still allows for property validation. Thus, we could measure the amount of bias in the input space in parallel for each partition, and then aggregate the quantity. Finally, we show how to validate the k-bounded impact property for the parallel semantics. This section is based on the work presented in Urban et al. [73] and Mazzucato and Urban [47].

[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

[47]: Mazzucato et al. (2021), 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'

6.4.1 Parallel Semantics

We observe that the semantics of a program satisfying the $\mathcal{B}^{\otimes k}_{\text{QAU}_{\text{NUSED}_{\mathbb{W}}}}$ property induces a partition of the input space restricted to the input variables in $\Delta \setminus \mathbb{W}$. We call this input partition *fair*.

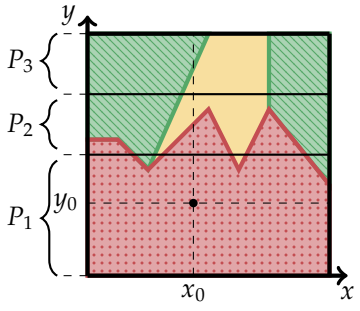


Figure 6.9: The partitions P_1, P_2, and P_3 are fair partitions for the variable y.

Definition 6.4.1 (Fair Input Partition) *An input partition* $\mathbb{I}$ *of* $\Sigma|_{\Delta}$ *is* fair *if all value choices* $\mathbb{V}$ *for the variables of interest* $\mathbb{W}$ *are possible in all the partitions:* $\forall \mathrm{I} \in \mathbb{I}, v \in \mathbb{V}.\ \exists s_0 \in \mathrm{I}.\ s_0(\mathbb{W}) = v$

Example 6.4.1 Figure 6.9 show a potential partition of the input space into P_1 and P_2 and P_3, fair for the input variables $\{y\}$. All possible permutations in the y-axis of any input value (x_0, y_0) result in a point (x_0, y') that still belongs to the same partition of (x_0, y_0).

Given a fair input partition $\mathbb{I}$ of $\Sigma|_\Delta$, we can verify whether a program P has an impact of k for each element $\mathrm{I} \in \mathbb{I}$, *independently*, and aggregate the results.

Lemma 6.4.1 *Given a fair input partition* $\mathbb{I}$*, it holds that:*

$$P \models \mathcal{B}^{\otimes k}_{\text{QAUnused}_W} \Leftrightarrow \sum_{\mathrm{I} \in \mathbb{I}} \text{QAUnused}_W(\Lambda[\![P]\!]\|_\mathrm{I}) \otimes k$$

Def. 6.2.3 (QAUnused)

$$\text{QAUnused}_W(T) = \text{Volume}\left(\left\{\sigma_0(\Delta \setminus W) \;\middle|\; \text{AUnused}_W(T_{\sigma_0})\right\}\right)$$

where $T_{\sigma_0} = \{\sigma' \in T \mid \sigma'_0 =_{\Delta\setminus W} \sigma_0\}$.

Proof. Note that, by Definition 6.2.3, $\text{QAUnused}_W(T)$ computes the volume of input space that does not use the input variables W w.r.t. to the set of traces T. It does so by partitioning the set of given traces that are perturbations of the input variables W, cf. the set T_{σ_0} on the side. By definition, these partitions are fair partitions by Definition 6.4.1. Furthermore, we could call these partitions as *minimal* in the sense that they are the smallest fair partitions that can be obtained from the input space. As a consequence, the QAUnused_W impact quantifier could be written as the sum of the impacts of the minimal fair partitions, *i.e.*, $\text{QAUnused}_W(\Lambda[\![P]\!]) = \sum_{\sigma\in\Lambda[\![P]\!]} \text{QAUnused}_W(\{\sigma' \in \Lambda[\![P]\!] \mid \sigma'_0 =_{\Delta\setminus W} \sigma_0\})$. Since any combination of minimal fair partitions is a fair partition, it holds that:

$$\text{QAUnused}_W(\Lambda[\![P]\!]) = \sum_{\mathrm{I}\in\mathbb{I}} \text{QAUnused}_W(\Lambda[\![P]\!]\|_\mathrm{I}) \tag{6.1}$$

where $\Lambda[\![P]\!]\|_\mathrm{I}$ is the set of traces restricted to the input partition I. We conclude:

$$\begin{aligned}
&P \models \mathcal{B}^{\otimes k}_{\text{QAUnused}_W} && \text{by Definition 4.1.1}\\
&\Leftrightarrow \text{QAUnused}_W(\Lambda[\![P]\!]) \otimes k && \text{Let } k' \otimes k \text{ such that}\\
& && \quad \text{QAUnused}_W(\Lambda[\![P]\!]) = k'\\
&\Leftrightarrow \text{QAUnused}_W(\Lambda[\![P]\!]) = k' && \text{by Equation 6.1}\\
&\Leftrightarrow \sum_{\mathrm{I}\in\mathbb{I}} \text{QAUnused}_W(T_\mathrm{I}) = k' && \text{by } k' \otimes k\\
&\Leftrightarrow \sum_{\mathrm{I}\in\mathbb{I}} \text{QAUnused}_W(T_\mathrm{I}) \otimes k
\end{aligned}$$

In fact, we proved an even stronger result, the fact that given a fair input partition $\mathbb{I}$, the volume of fair input space is exactly as the sum of the volumes of fair space in each fair partition. □

We exploit the above insight to further abstract the output-abstraction semantics α^ρ to the *parallel semantics* $\Lambda^\mathbb{I}$. Formally, the right adjoint[1] $\alpha^\mathbb{I}$ for the parallel semantics is defined as:

1: The left adjoint is uniquely defined by the right one.

Definition 6.4.2 (Right Adjoint for the Parallel Semantics)

$$\begin{aligned} \alpha^{\mathbb{I}} &\in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma \times \Sigma^{\perp})) \\ \alpha^{\mathbb{I}}(W) &\stackrel{\text{def}}{=} \{D^{\mathrm{I}} \mid D \in W \ \wedge \ \mathrm{I} \in \mathbb{I}\} \\ \gamma^{\mathbb{I}} &\in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma \times \Sigma^{\perp})) \\ \gamma^{\mathbb{I}}(W) &\stackrel{\text{def}}{=} \{\bigcup_{D \in G} D \mid G \in \prod_{\mathrm{I} \in \mathbb{I}} \cup_{D \in W} D^{\mathrm{I}}\} \end{aligned}$$

where D^{I} *is the set of dependencies restricted to the input partition* I*, i.e.,* $D^{\mathrm{I}} \stackrel{\text{def}}{=} \{s \mid \langle s_0, s_\omega \rangle \in D \wedge s_0 \in \mathrm{I}\}$.

The parallel abstraction $\alpha^{\mathbb{I}}$ is defined as the set of dependencies restricted to each fair input partition. Conversely, the parallel concretization $\gamma^{\mathbb{I}}$ brings back all the original set of dependencies by joining all the dependencies in each fair input partition. Potentially, the concretization adds set of dependencies that are not in the original set of dependencies as spurious combinations of different fair input partitions.

We have the following result:

Theorem 6.4.2 *The two adjoints* $\langle \alpha^{\mathbb{I}}, \gamma^{\mathbb{I}} \rangle$ *form a* Galois connection*:*

$$\langle \wp(\wp(\Sigma \times \Sigma^{\perp})), \subseteq \rangle \underset{\alpha^{\mathbb{I}}}{\overset{\gamma^{\mathbb{I}}}{\leftrightarrows}} \langle \wp(\wp(\Sigma \times \Sigma^{\perp})), \subseteq \rangle$$

Proof. We need to show that $\alpha^{\mathbb{I}}(W) \subseteq W' \Leftrightarrow W \subseteq \gamma^{\mathbb{I}}(W')$. First, we show the direction ($\Rightarrow$). Assuming $\alpha^{\mathbb{I}}(W) \subseteq W'$, we have that $\gamma^{\mathbb{I}}(W')$ contains all the possible semantics made of different combinations of input fair partitions of W'. Thus, $\gamma^{\mathbb{I}}(W')$ at least retrieves all the semantics in W, *i.e.*, $W \subseteq \gamma^{\mathbb{I}}(W')$. To show ($\Leftarrow$), we assume $W \subseteq \gamma^{\mathbb{I}}(W')$. It is easy to note that $\alpha^{\mathbb{I}}(\gamma^{\mathbb{I}}(W')) = W'$ since the concretization joins together set of dependencies from different fair input partitions and the abstraction splits them subdivided into fair input partitions, obtaining the original set. Hence, by monotonicity of $\alpha^{\mathbb{I}}$, we obtain $\alpha^{\mathbb{I}}(W) \subseteq \alpha^{\mathbb{I}}(\gamma^{\mathbb{I}}(W')) = W'$. □

We can now derive the parallel semantics as an abstraction of the output-abstraction semantics.

Definition 6.4.3 (Parallel Abstraction Semantics) *The* parallel semantics $\Lambda^{\mathbb{I}} \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$ *is defined as:*

$$\Lambda^{\mathbb{I}} \stackrel{\text{def}}{=} \alpha^{\mathbb{I}}(\Lambda^{\rho}) = \{ \{\langle \sigma_0, \rho(\sigma_\omega) \rangle \mid \sigma \in \Lambda^{\mathrm{I}}\} \mid \mathrm{I} \in \mathbb{I}\}$$

where $\Lambda^{\mathrm{I}} \stackrel{\text{def}}{=} \{s \mid \langle s_0, s_\omega \rangle \in \Lambda \wedge s_0 \in \mathrm{I}\}$.

It remains to show soundness and completeness for the parallel semantics when applied to the $\mathscr{B}^{\otimes k}_{\text{QAU}_{\text{NUSED}_W}}$ property.

Theorem 6.4.3 *Given a fair input partition* $\mathbb{I}$ *of* $\Sigma|_\Delta$*, it holds that:*

$$P \models \mathscr{B}^{\otimes k}_{\text{QAU}_{\text{NUSED}_W}} \ \Leftrightarrow \ \Lambda^{\mathbb{I}}[\![P]\!] \subseteq \alpha^{\mathbb{I}}(\alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathscr{B}^{\otimes k}_{\text{QAU}_{\text{NUSED}_W}})))$$

Proof. Let $P \models \mathcal{B}^{\otimes k}_{\text{QAUnused}_W}$. From Lemma 6.2.4, we have that $\Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{B}^{\otimes k}_{\text{QAUnused}_W}))$. Thus, from the Galois connections in Theorem 6.4.2, we have that $\alpha^{\mathbb{I}}(\Lambda^{\rho}[\![P]\!]) \subseteq \alpha^{\mathbb{I}}(\alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{B}^{\otimes k}_{\text{QAUnused}_W})))$. From Definition 6.4.3, we can conclude that $\Lambda^{\mathbb{I}}[\![P]\!] \subseteq \alpha^{\mathbb{I}}(\alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{B}^{\otimes k}_{\text{QAUnused}_W})))$. □

Lemma 6.2.4

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathcal{B}^{\otimes k}_{\text{QAUnused}_W} \Leftrightarrow \Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathcal{B}^{\otimes k}_{\text{QAUnused}_W}))$$

6.4.2 Parallel Implementation $\texttt{QAUnused}^{\mathbb{I}}_W$

In this section, we build upon the parallel semantics to design an abstract implementation $\texttt{QAUnused}^{\natural}_W$, called $\texttt{QAUnused}^{\mathbb{I}}_W$, that computes an over-approximation of the volume of input space that may be fair. This static analysis automatically find a fair partition of the input space, then computes the volume of the fair input space. The abstract implementation $\texttt{QAUnused}^{\mathbb{I}}_W$ is defined as the sum of the volumes of the fair partitions.

Algorithm 3 shows the implementation of $\texttt{QAUnused}^{\mathbb{I}}_W$, combining a forward with a backward analysis. The forward analysis (cf. Algorithm 1) uses an abstract domain $\mathbb{D}$ and builds partition $\mathbb{I}$ of the input space, while the backward analysis (cf. Algorithm 2) employs the disjunction of polyhedra abstract domain $\mathbb{DP}$. Then, we perform the quantification of the biased space.

More specifically, the forward analysis bounds the number of paths that the backward analysis has to explore. We represent each path by an abstract activation pattern.[2] The analysis receives a *budget* parameter, called TUNE in Line 2, providing an upper bound U on the number of tolerated ReLU nodes with an unknown activation status for each element $\mathrm{I} \in \mathbb{I}$, *i.e.*, an upper bound on the number of paths that are to be explored by the backward analysis in each partition I. The forward analysis starts with the trivial partition of the whole input space, *i.e.*, $\mathbb{I} = \{\Sigma|_{\Delta}\}$ in Line 4. Then, it proceeds forward for each element $\mathrm{I} \in \mathbb{I}$ by computing the abstract activation patterns that are compatible with the input partition I (cf. Line 8), starting from the empty set of activation patterns. If I leads to a unique outcome (cf. Line 9), then the partition is already fair without further analysis. Therefore, we add the partition I to the set of *completed* partitions C, (cf. Line 10). On the other hand, if abstract activation pattern p fixes the activation status of enough ReLU nodes, then we declare the partition I *feasible* and ready for the backward analysis. In this case, the pair of p and I is added into a map F from abstract activation patterns to input partitions (cf. Line 12). Otherwise, the partition I needs to be further refined, with respect to the variables in $\Delta \setminus \mathrm{W}$ (cf. Line 16). Partitioning may continue until the volume of the partition is below a certain threshold L from the budget configuration TUNE. Finally, whenever the partition is smaller than L (cf. Line 13), we exclude it from the set of partitions to be analyzed until more resources become available (cf. Line 14). Note that, the forward analysis does not need expressive and slow abstract domain since it does not need to precisely handle polyhedral constraints.

The budget configuration of the pre-analysis (*i.e.*, choices of an abstract domain, lower bound L, and upper bound U) allows trading-off between precision and scalability of the approach. Ultimately however, the optimal configuration largely depends on the analyzed neural network. For this reason, a *configuration auto-tuning mechanism* dynamically updates the lower bound and upper bound configuration according to a chosen *search heuristic h* (cf. Line 18). By default, whenever an input partition exceeds

The symbol $\Sigma|_{\Delta}$ represents the set of input states, *i.e.*, let Δ the set of input variables:

$$\Sigma|_{\Delta} \stackrel{\text{def}}{=} \{s(\Delta) \mid s \in \Sigma\}$$

2: The activation pattern determines the activation status of every ReLU operation in the network model. The abstract activation pattern is a partial activation pattern where some activation status may be unknown.

Algorithm 3: Parallel Implementation $\texttt{QAUnused}^{\mathbb{I}}_W$

```
1  QAUnused^{||}_{M,𝔻}(W, TUNE):
2    U, L, U_max, L_min, h = TUNE
3    F, E, C = ∅, ∅, ∅
4    𝕀 = {Σ|_Δ}
5    do
6      while 𝕀 ≠ ∅ do
7        I = 𝕀.pop()
8        d♮, p = Forward_{M,𝔻}(I)
9        if d♮ is unique then
10         C = C ∪ {⟨p, I⟩}
11       else if |M| − |p| ≤ U then
12         F = F ∪ {I}
13       else if |I| ≤ L then
14         E = E ∪ {I}
15       else
16         𝕀 = 𝕀 ∪ Partition_{Δ\W}(I)
17     𝕀 = E
18     U, L = h(U, L)
19   while U ≠ U_max and L ≠ L_min
20   E = 𝕀
21   B = ∅
22   for ⟨p, 𝕀⟩ in F do
23     O = ∅
24     for j = 0 up to |L_N| do
25       d♮ = Backward_{M,𝔻ℙ,p}(x_{L_N,j})
26       O = O ∪ {⟨x_{L_N,j}, d♮⟩}
27     for I in 𝕀 do
28       O' = ∅
29       for ⟨o, d♮⟩ in O do
```

the current configuration (cf. Line 17), the pre-analysis alternates between increasing the upper bound by one, up to a maximum upper bound U_{max}, and halving the lower bound, down to a minimum lower bound L_{min}. Other bound update patterns are configurable (e.g., by updating both bounds at the same time, or performing multiple increments to the upper bound before halving the lower bound, etc.). To guarantee the termination of the whole analysis, the search heuristic should return (U_{max}, L_{min}) after a finite number of iterations.

Then, the analysis proceeds backwards, independently for each abstract activation pattern p and input partition I in F (cf. Line 22). By exploiting the knowledge of the activation pattern, the backward analysis does not need to explore all the possible paths in the network model, but only those with an unknown activation status. Specifically, the backward analysis only needs to split the abstract invariant only when a node has an unknown activation status, as otherwise it is handled as either the identity of the constant function. For each input partition, the analysis computes the abstract domain representing the biased input space (cf. Line 32). Specifically, the function Check, cf. Algorithm 4, performs the pair-wise intersection among the abstract values coming from different output buckets (*i.e.*, different network classifications) and returns the set of abstract values that intersect, representing the biased input space.

Algorithm 4: Bias check.

```
1 Check(O):
2   B = ∅
3   for o1, d1♮ in O do
4     for o2 ≠ o1, d2♮ in O do
5       if d1♮ ⊓ d2♮ ≠ ⊥ then
6         B = B ∪ {d1♮ ⊓ d2♮}
7   return B
```

Finally, the abstract implementation $\texttt{QAUnused}^{\natural}_{\mathtt{W}}$ computes the sum of the volumes of the biased portion of each partition together (cf. Line 32). The complement of the biased input space is an under-approximation of the fair input space.

The next result shows that the abstract implementation $\texttt{QAUnused}^{\natural}_{\mathtt{W}}$ is a sound under-approximation of the concrete implementation $\text{QAU}_{\text{NUSED}_{\mathtt{W}}}$. Therefore, it can be used to validate the $\mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$ property with the $\geq$ operator.

Theorem 6.4.4 *Let $\mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$ be the property of interest we want to verify for the model M and the input variable $\mathtt{W} \in \wp(\Delta)$. Given an abstract domain $\mathbb{D}$, and the budget $\text{TUNE} = \langle U, L, U_{max}, L_{min}\rangle$, the following implication holds:*

$$\texttt{QAUnused}^{\natural}_{\mathtt{W}M,\mathbb{D}}(\mathtt{W}, \text{TUNE}) = k' \wedge k' \geq k \implies M \models \mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$$

Proof. $\texttt{QAUnused}^{\natural}_{\mathtt{W}M,\mathbb{D}}(\mathtt{W}, \text{TUNE})$ in Algorithm 3 first computes the abstract activation patterns that cover a fraction C of the input space in which the analysis is feasible (cf. from Line 3 to Line 12). Then, it computes an over-approximation of the regions of C that yield each target class $x_{\text{L}_N,j}$ (cf. Line 25). Thus, it actually computes an over-approximation of the parallel semantics $\Lambda^{\natural}[\![P]\!]$. In Line 32, the complement of the sum of the volumes, one for each fair partition (cf. Definition 6.4.1), is always greater than the volume of the fair input space. We conclude $M \models \mathscr{B}^{\geq k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}}$ by Lemma 6.4.1. □

Lemma 6.4.1

$$P \models \mathscr{B}^{\otimes k}_{\text{QAU}_{\text{NUSED}_{\mathtt{W}}}} \Leftrightarrow \sum_{\mathtt{I} \in \mathbb{I}} \text{QAU}_{\text{NUSED}_{\mathtt{W}}}(\Lambda[\![P]\!]\|_{\mathtt{I}}) \otimes k$$

[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

Notably, the parallel implementation $\texttt{QAUnused}^{\natural}_{\mathtt{W}}$ yields the so called *perfectly parallel fairness analysis* [73].

6.5 Related Work

In this section, we discuss related work on the verification of neural networks and the quantification of input usage in machine learning models.

Verification of Neural Networks. The interest in the verification of neural networks has significantly increased over the last decade. For an introduction to the topic, refer to Albarghouthi's tutorial [120]. The most common approach in this domain is to verify the robustness of neural networks against minimal input perturbations [121]. This research area is highly diverse and includes a variety of techniques, including SMT-based methods [122–126], mixed-integer programming [127–129], branch-and-bound algorithms [115, 130–132], over-approximation of non-linear activation functions [117, 132–134], and symbolic propagation [115, 127, 135, 136]. For a comprehensive survey of the field, see König et al. [137] and Huang et al. [138]. Notably, no single algorithm consistently outperforms others across all verification problem instances.

The literature also extensively addresses fairness and bias in machine learning models. For example, statistical approaches include Galhotra, Brun, and Meliou [139], which proposes an efficient method for fairness testing, Udeshi, Arora, and Chattopadhyay [140], which generates discriminatory inputs for machine learning models, and Tramèr et al. [141], which introduces the unwarranted-associations framework. On the other hand, Bastani, Zhang, and Solar-Lezama [142] provides probabilistic guarantees on the fairness of machine learning models, while Urban et al. [73] uses abstract interpretation to verify fairness properties in neural networks, offering formal guarantees. Additionally, Albarghouthi et al. [143] encodes the fairness problem as a probabilistic program property, verified using an SMT solver. Furthermore, Albarghouthi and Vinitsky [144] proposes a technique to repair biases in neural networks.

Recent formal developments [145–147] are expanding this research to other machine learning models, such as decision trees and support vector machines, by developing formal methods to analyze input variable usage. Quantitative analyses have the potential to enhance these techniques by leveraging a broader spectrum of possible results.

Quantitative Verification of Neural Networks. There is also a growing interest in the quantification of properties in the context of verification of neural networks, with most of the literature providing statistical guarantees. While our thesis does not address probabilistic properties, it offers formal guarantees regarding the quantification of impact properties. For example, Tran et al. [148] quantifies the probability of safety violations by propagating Gaussian-distributed inputs through neural networks. Similarly, Baluta et al. [149, 150] estimate the portion of the input space that satisfies properties such as robustness, fairness, or security, through sampling. These estimates come with statistical guarantees of being within a specified confidence interval. The framework presented by Zhang et al. [151, 152] quantifies adversarial robustness in binarized neural networks by model counting adversarial examples. Another statistical approach, [153], uses Monte Carlo methods to estimate the

[120]: Albarghouthi (2021), 'Introduction to Neural Network Verification'
[121]: Goodfellow et al. (2016), *Deep Learning*
[122]: Katz et al. (2017), 'Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks'
[123]: Katz et al. (2019), 'The Marabou Framework for Verification and Analysis of Deep Neural Networks'
[124]: Pulina et al. (2011), 'Checking Safety of Neural Networks with SMT Solvers: A Comparative Evaluation'
[125]: Pulina et al. (2011), 'NeVer: a tool for artificial neural networks verification'
[126]: Pulina et al. (2012), 'Challenging SMT solvers to verify neural networks'
[127]: Botoeva et al. (2020), 'Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis'
[128]: Lomuscio et al. (2017), 'An approach to reachability analysis for feed-forward ReLU neural networks'
[129]: Tjeng et al. (2019), 'Evaluating Robustness of Neural Networks with Mixed Integer Programming'
[115]: Wang et al. (2018), 'Formal Security Analysis of Neural Networks using Symbolic Intervals'
[130]: Bunel et al. (2018), 'A Unified View of Piecewise Linear Neural Network Verification'
[131]: Palma et al. (2021), 'Improved Branch and Bound for Neural Network Verification via Lagrangian Decomposition'
[132]: Ehlers (2017), 'Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks'
[117]: Singh et al. (2019), 'An abstract domain for certifying neural networks'
[132]: Ehlers (2017), 'Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks'
[133]: Weng et al. (2018), 'Towards Fast Computation of Certified Robustness for ReLU Networks'
[134]: Zhang et al. (2018), 'Efficient Neural Network Robustness Certification with General Activation Functions'
[115]: Wang et al. (2018), 'Formal Security Analysis of Neural Networks using Symbolic Intervals'
[127]: Botoeva et al. (2020), 'Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis'
[135]: Henriksen et al. (2020), 'Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search'
[136]: Wang et al. (2021), 'Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification'
[137]: König et al. (2024), 'Critically Assessing the State of the Art in Neural Network Verification'

likelihood of property violations, thereby providing robustness metrics. These works can be viewed as quantifications of the Outcomes quantifier for fairness checks in a weaker, statistical sense.

6.6 Summary

In this chapter, we presented the Changes and QAUnused impact quantifiers, and their abstract implementations, `Changes`$^\sharp$ and `QAUnused`$^\sharp$, respectively. We validated the k-bounded impact property for both quantifiers. To address the scalability issues of the backward analysis, we introduced the parallel semantics and the abstract implementation `QAUnused`$^\natural$. The next chapter will present the experimental results of the Changes and QAUnused impact quantifiers applied to neural networks. Afterwards, the thesis will focus on intensional properties, specifically how to quantify the influence of input variables on the number of iterations of a program.

Dans ce chapitre, nous avons présenté les quantificateurs d'impact Changes *et* QAUnused*, ainsi que leurs implémentations abstraites,* `Changes`$^\sharp$ *et* `QAUnused`$^\sharp$*, respectivement. Nous avons validé la propriété d'impact borné par k pour les deux quantificateurs. Pour résoudre les problèmes de scalabilité de l'analyse rétrograde, nous avons introduit les sémantiques parallèles et l'implémentation abstraite* `QAUnused`$^\natural$*. Le prochain chapitre présentera les résultats expérimentaux des quantificateurs d'impact* Changes *et* QAUnused *appliqués aux réseaux neuronaux. Par la suite, la thèse se concentrera sur les propriétés intentionnelles, en particulier sur la manière de quantifier l'influence des variables d'entrée sur le nombre d'itérations d'un programme.* ¡

[138]: Huang et al. (2020), 'A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability'

[139]: Galhotra et al. (2017), 'Fairness testing: testing software for discrimination'

[140]: Udeshi et al. (2018), 'Automated directed fairness testing'

[141]: Tramèr et al. (2017), 'FairTest: Discovering Unwarranted Associations in Data-Driven Applications'

[142]: Bastani et al. (2019), 'Probabilistic verification of fairness properties via concentration'

[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

[143]: Albarghouthi et al. (2017), 'FairSquare: probabilistic verification of program fairness'

[144]: Albarghouthi et al. (2019), 'Fairness-Aware Programming'

[145]: Ranzato et al. (2020), 'Abstract Interpretation of Decision Tree Ensemble Classifiers'

[146]: Ranzato et al. (2021), 'Fairness-Aware Training of Decision Trees by Abstract Interpretation'

[147]: Pal et al. (2022), 'Abstract Interpretation-Based Feature Importance for SVMs'

[148]: Tran et al. (2023), 'Quantitative Verification for Neural Networks using ProbStars'

[149]: Baluta et al. (2019), 'Quantitative Verification of Neural Networks and Its Security Applications'

[150]: Baluta et al. (2021), 'Scalable Quantitative Verification For Deep Neural Networks'

[151]: Zhang et al. (2021), 'BDD4BNN: A BDD-Based Quantitative Analysis Framework for Binarized Neural Networks'

[152]: Zhang et al. (2023), 'Precise Quantitative Analysis of Binarized Neural Networks: A BDD-based Approach'

[153]: Webb et al. (2019), 'A Statistical Approach to Assessing Neural Network Robustness'

Experimental Evaluation on Neural Networks

7

In this chapter, we investigate whether our two impact quantifiers for neural networks, cf. Changes and QAUnused, successfully quantify variations in the usage of input features. Specifically, Section 7.1 evaluates the effectiveness of Changes by comparing it to two *feature importance metrics*, which are commonly used to evaluate the relevance of input features in making predictions within a machine learning model. Section 7.2 evaluates QAUnused on neural networks trained on the Adult dataset [1] from the UCI Machine Learning Repository to discover the amount of fair input space. This chapter is partially based on the evaluation section of the work presented at the 28th Static Analysis Symposium (SAS) 2021 [47, Section 3].

Dans ce chapitre, nous examinons si nos deux quantificateurs d'impact pour les réseaux neuronaux, cf. Changes *et* QAUnused, *permettent de quantifier efficacement les variations dans l'utilisation des caractéristiques d'entrée. Plus précisément, Section 7.1 évalue l'efficacité de* Changes *en le comparant à deux* mesures d'importance des caractéristiques, *qui sont couramment utilisées pour évaluer la pertinence des caractéristiques d'entrée dans les prédictions d'un modèle d'apprentissage automatique. Section 7.2 évalue* QAUnused *sur des réseaux neuronaux entraînés sur le jeu de données Adult*[1] *de la UCI Machine Learning Repository pour découvrir la quantité d'espace d'entrée équitable. Ce chapitre est partiellement basé sur la section d'évaluation du travail présenté lors du 28e Symposium sur l'analyse statique (SAS) 2021 [47, Section 3].*

1: archive.ics.uci.edu/ml/datasets/adult

[47]: Mazzucato et al. (2021), 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'

7.1 Evaluation of Changes

For the evaluation of Changes, we implemented this evaluation prototype as part of the tool Impatto,[2] cf. Chapter 5, in Python. We extended the front-end of Impatto to handle neural networks written in Python format, and the backend to support the abstract analysis of neural networks, cf. Section 6.1.3. This prototype implements the $\texttt{Changes}^{\natural}$ abstract quantifier, which is used to quantify the impact of input features on the output of a neural network. All neural network models used in this evaluation are open source and part of Impatto. In particular we instrument Impatto to use $\texttt{Changes}^{\natural}$ starting from the output buckets representing all the target classes in the model, one for each bucket: $B = [\{\bigwedge_{j' \leq m} x_{\text{L}_N,j'} \leq x_{\text{L}_N,j}\} \mid j \leq m]$.

2: github.com/denismazzucato/impatto

For our evaluation, we used public datasets from the online community platform Kaggle[3] to train several neural network models. We focused on four datasets: "Red Wine Quality" [154], "Prima Indians Diabetes" [155], "Rain in Australia" [156], and "Cure the Princess" [157]. We pre-processed the "Rain in Australia" database and removed non-continuous input features, as we do not support discrete input features yet. To preserve data consistency, since the majority of daily weather observations were collected in Canberra, we eliminated the observations from other stations.

3: www.kaggle.com

[154]: Cortez et al. (2009), 'Modeling wine preferences by data mining from physicochemical properties'

[155]: Smith et al. (1988), 'Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus'

[156]: Young (2019), *Rain in Australia*

[157]: Unmoved (2023), *Cure The Princess*

As a result of this pre-processing step, we retained approximately 2000 entries, aligning with the sizes of other datasets. We trained about 700 networks per database by permuting the network structure and number of input features to obtain a uniform benchmark. The number of input features ranges from at least 3, to the number of attribute of the databases (after pre-processing), respectively, 10 attributes for "Red Wine Quality," 8 for "Prima Indians Diabetes", 17 for "Rain in Australia", and 13 for "Cure the Princess". The model size ranges from 2 to 6 hidden layers, each with 3 to 6 nodes, for a total of 6 to 36 nodes for each network model. This relatively small network size is chosen as a proof-of-concept evaluation to study Changes without the need for extensive computational resources. All models were trained with Keras,[4] using the Adam optimizer [158] with the default learning rate, and binary crossentropy as the loss function. Each model was trained for 150 iterations. The obtained neural network accuracy usually depends on the chosen subset of input features which is usually lower than the accuracy achieved in the literature. However, we remark that our study focuses on the impact analysis, therefore high accuracy is not needed in our benchmarks.

4: github.com/keras-team/keras

[158]: Kingma et al. (2014), 'Adam: A Method for Stochastic Optimization'

To empirically check that Impatto specialized with $\texttt{Changes}^{\natural}_{w}$, behaves similarly to Changes, we uniformly sample 1000 points in the input space of the network and then apply Changes to this set; we refer to this result as the *baseline*. This approach is not sound as we could miss changes in the outcome not exploited by unsampled points, however it is overall a close-enough approximation of Changes. We compare the result of Impatto with baseline employing four heuristics, called maximum common prefix length (mcpl), relaxed maximum common prefix length (rmcpl), Euclidean distance (ed), and Manhattan distance (md), defined below.

Given two analyses F, F' (*e.g.*, Impatto and baseline), the heuristic (mcpl) first sorts the result of the analyses F, F' applied to all the input features, by decreasing order. The heuristic then returns the corresponding indices of the sorted results. Formally, it computes $I = \arg\mathrm{sort}_{\mathtt{i}}\, F(M, \mathtt{i})$ and $J = \arg\mathrm{sort}_{\mathtt{i}}\, F'(M, \mathtt{i})$ for the two analyses respectively, where arg sort returns the corresponding indices of the sorted list (by decreasing order), *e.g.*, $\arg\mathrm{sort}\langle 30, 65, 2, 60\rangle = \langle 2, 4, 1, 3\rangle$. Afterwards, (mcpl) retrieves the length of the maximal common prefix between I and J:

$$\textsc{mcpl}(I, J) \stackrel{\text{def}}{=} \begin{cases} 1 + \textsc{mcpl}(\langle i_2, \ldots, i_m\rangle, \langle j_2, \ldots, j_m\rangle) \\ \quad \text{if } I = \langle i_1, \ldots, i_m\rangle \wedge J = \langle i_1, \ldots, j_m\rangle \\ 0 \\ \quad \text{otherwise} \end{cases} \tag{7.1}$$

For instance, assuming $I = \langle 4, 1, 2, 3, 5\rangle$ and $J = \langle 4, 1, 2, 5, 3\rangle$, $\textsc{mcpl}(I, J) = 3$ since the maximum common prefix is $\langle 4, 1, 2\rangle$ of length 3. The relaxed variation (rmcpl) allows a 10% of overlap among quantity values, *e.g.*, given $[F(M, \mathtt{i}) \mid \mathtt{i}] = \langle 0.4, 0.95, 0.6, 1\rangle$ and $[F'(M, \mathtt{i}) \mid \mathtt{i}] = \langle 30, 65, 2, 60\rangle$, we obtain $I = \arg\mathrm{sort}\langle 0.4, 0.95, 0.6, 1\rangle = \langle 4, 2, 3, 1\rangle$ and $J = \arg\mathrm{sort}\langle 30, 65, 2, 60\rangle = \langle 2, 4, 1, 3\rangle$, hence $\textsc{mcpl}(I, J)$ would return 0. However, a 10% of margin of error permits to swap the indices of values 0.95 and 1 in the first list obtaining $\langle 2, 4, 3, 1\rangle$, we say that I is equivalent to $\langle 2, 4, 3, 1\rangle$ (written $I \simeq \langle 2, 4, 3, 1\rangle$), hence $\textsc{mcpl}(\langle 2, 4, 3, 1\rangle, J) = 2$. Formally, we define rmcpl as the maximum of mcpl of all the possible

equivalences:

$$\text{RMCPL}(I, J) \stackrel{\text{def}}{=} \max\{\text{MCPL}(I', J') \mid I' \simeq I \wedge J' \simeq J\} \tag{7.2}$$

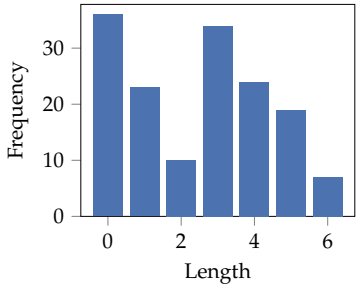
The Euclidean distance is defined as $\text{ED}(I, J) \stackrel{\text{def}}{=} \sqrt{\sum_k (I_k - J_k)^2}$, and the Manhattan distance as $\text{MD}(I, J) \stackrel{\text{def}}{=} \sum_k \mid I_k - J_k \mid$.

7.1.1 Quantifying Usage of Input Features

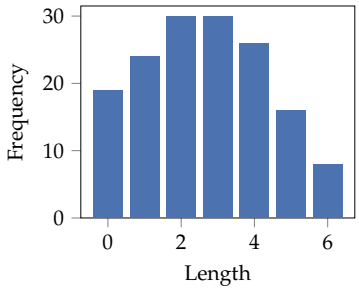
We first verify whether Impatto produces a similar impact quantification with respect to Changes. To this end, we demonstrate Impatto in comparison to baseline. This experiment uses the "Prima Indians Diabetes" dataset, full evaluation with all the datasets in later in Section 7.1.3. The evaluation, depicted in Figure 7.1, considers all the four heuristics.

The results confirm the similarity between Impatto and baseline. Figure 7.1a and Figure 7.1b show the exact and relaxed maximum common prefix length results. In this context, the result of our analysis Impatto is fairly similar to the baseline, this is even enhanced by the relaxed heuristic where close impact quantities (up to 10% difference) are softened together. In particular, Figure 7.1b shows that more than 100 test cases produce similar impact quantity to baseline up to, at least, the 3 most influent features.

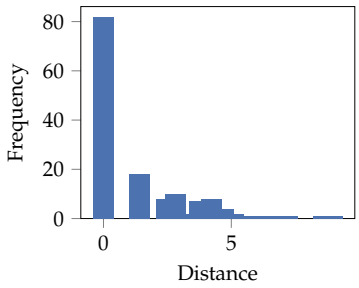
Figure 7.1c and Figure 7.1d show the result for the Euclidean and Manhattan distance respectively. For these two heuristics, a less dense graph shows higher similarity. Low distance means that the two input vectors (or list of sorted indices as in our case) are close together. Confirming our expectancy, both Figure 7.1c and Figure 7.1d show high similitude in most of the test cases. Note that, the Euclidean distance (ED) returns a real number as distance value, thus bars do not necessarily correspond to a discrete values in the graph.



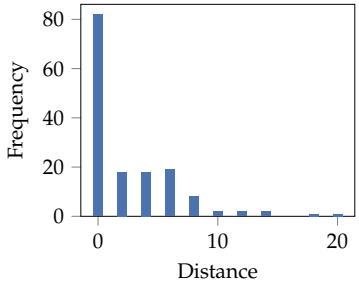
(a) Maximum common prefix length.



(b) Relaxed max. common prefix length.



(c) Euclidean distance



(d) Manhattan distance

Figure 7.1: "Prima Indians Diabetes" database, comparison between baseline and Impatto.

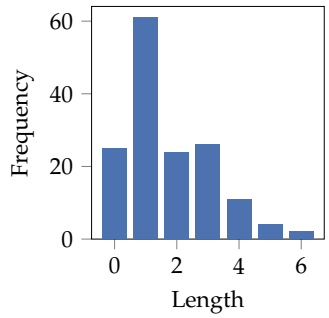
7.1.2 Comparison with Stochastic Methods

In this second experiment, we compare to the results of our analysis with stochastic quantitative measures, known as feature importance metrics, which are used to determine the influence of input variables in machine learning models. Specifically, we compare to:

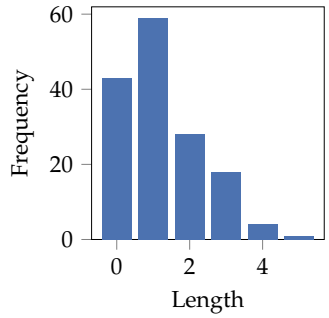
(RFE) A naïve feature importance metric that evaluates the changes in performance of a model when retrained without the feature of interest; in the following we call this metric *Retraining Feature Elimination*.

(PFI) *Permutation Feature Importance* [159], one of the most popular feature importance metrics, which monitors changes in performance when the values of the feature of interest are randomly shuffled.

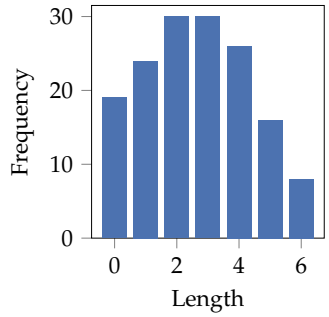
As before, we first employ the "Prima Indians Diabetes" dataset, then Section 7.1.3 presents the evaluation on all the datasets. Figure 7.2 demonstrates the comparison of baseline with, permutation feature importance (PFI), retraining feature importance (RFE), and Impatto, respectively Figure 7.2a, Figure 7.2b, and Figure 7.2c. In this evaluation, we focus on the relaxed maximum common prefix length (RMCPL) heuristic. Like the



(a) Baseline vs (PFI).



(b) Baseline vs (RFE).



(c) Baseline vs IMPATTO.

Figure 7.2: "Prima Indians Diabetes" database, comparison between baseline and (left to right) permutation feature importance, retraining feature importance, and IMPATTO.

[159]: Breiman (2001), 'Random Forests'

(MCPL) approach, it highlights the most impactful features. However, the key advantage of RMCPL is the employment of a margin opf error when computing the common prefix, which allows for a clearer distinction when two analysis results are similar. In summary, we notice that our static analyzer IMPATTO always achieves a higher similarity compared to the other two stochastic metrics considered.

In conclusion, our evaluation demonstrates the effectiveness of the prototype of our static analysis in quantifying the impact of input features with respect to a formally defined impact property. Through comparison with stochastic metrics, we consistently observe a higher degree of similarity between the results produced by IMPATTO and the baseline. The experiments conducted on the "Prima Indians Diabetes" dataset illustrate this similarity across various heuristics and distance measures. Notably, our approach stands out as the only one capable of addressing a formally defined impact property, providing a flexible framework that surpasses the hard-coded intuitions of the other two methods. Overall, these findings highlight the reliability and strength of IMPATTO in assessing the significance of input features.

7.1.3 Overview on all Datasets

This section contains the (full) overview of the experiments on all datasets. All the figures are organized in a 4 rows by 3 columns setup, we dedicated one page for each dataset. Figure Figure 7.3 refers to "Prima Indians Diabetes" dataset, Figure 7.4 to "Red Wine Quality," Figure 7.5 to "Rain in Australia," and Figure 7.6 to "Cure the Princess." Each column corresponds to a different comparison: from left to right, we show the comparison between IMPATTO vs PFI, IMPATTO vs RFE, and PFI vs RFE.

The row refers to a different similarity metric, namely, from top to bottom, maximum common prefix length (MCPL, already defined in Equation 7.1), relaxed maximum common prefix length (RMCPL), Euclidean distance (ED), and Manhattan distance (MD). Similarly to the sections above, the x-axis represents the length of the longest prefix, while the y-axis represents the frequency of test cases with a prefix of *exactly* the length of the x-axis.

Continue in the next pages.

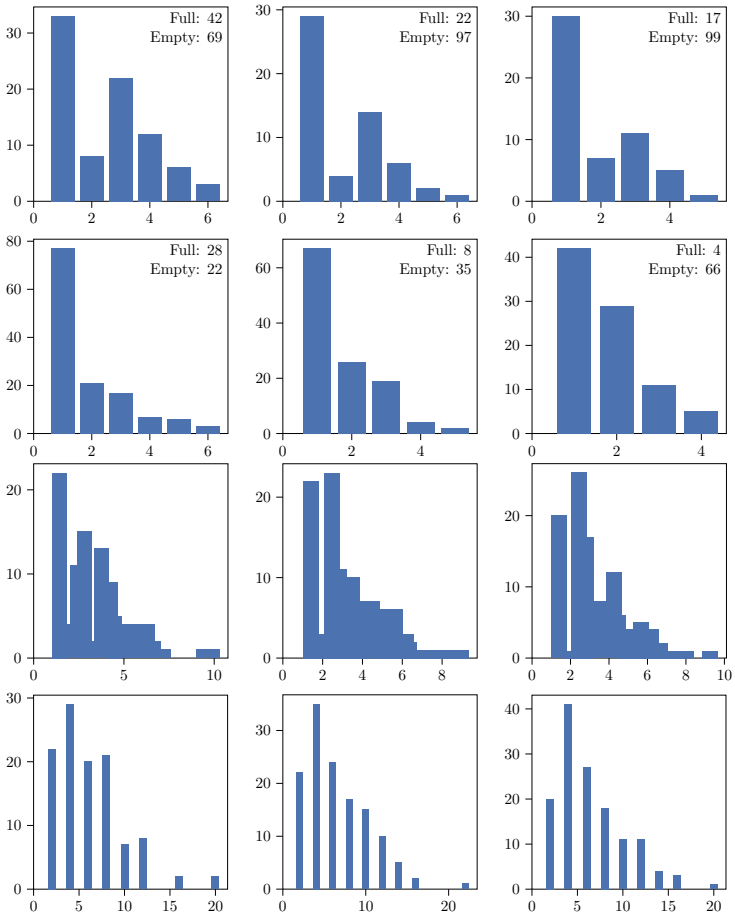


Figure 7.3: Experimental overview regarding the Prima Indians Diabetes dataset.

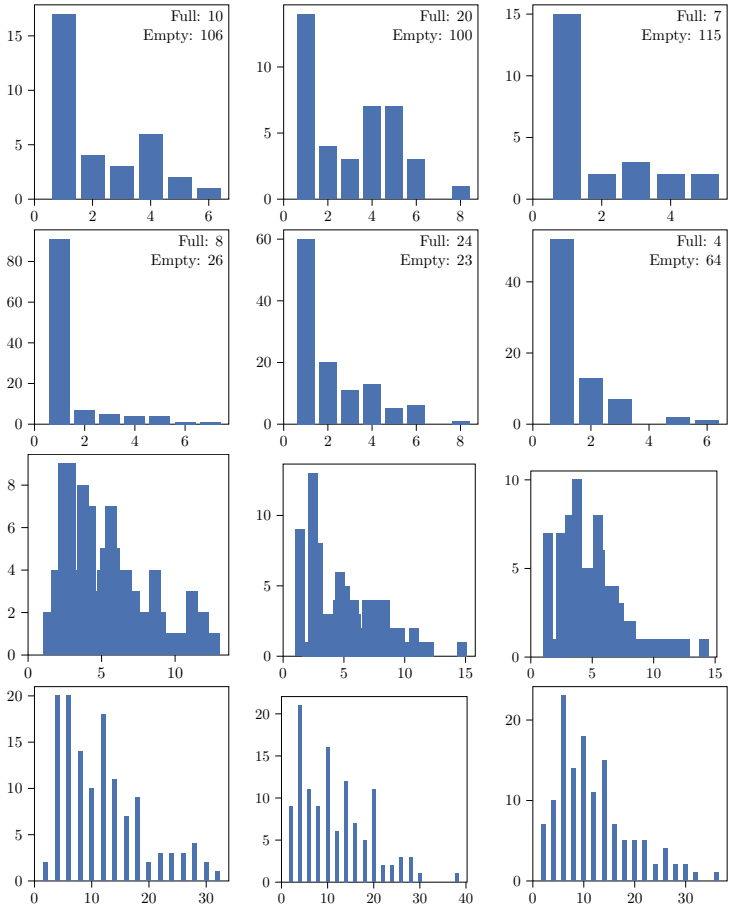


Figure 7.4: Experimental overview regarding the "Red Wine Quality" dataset.

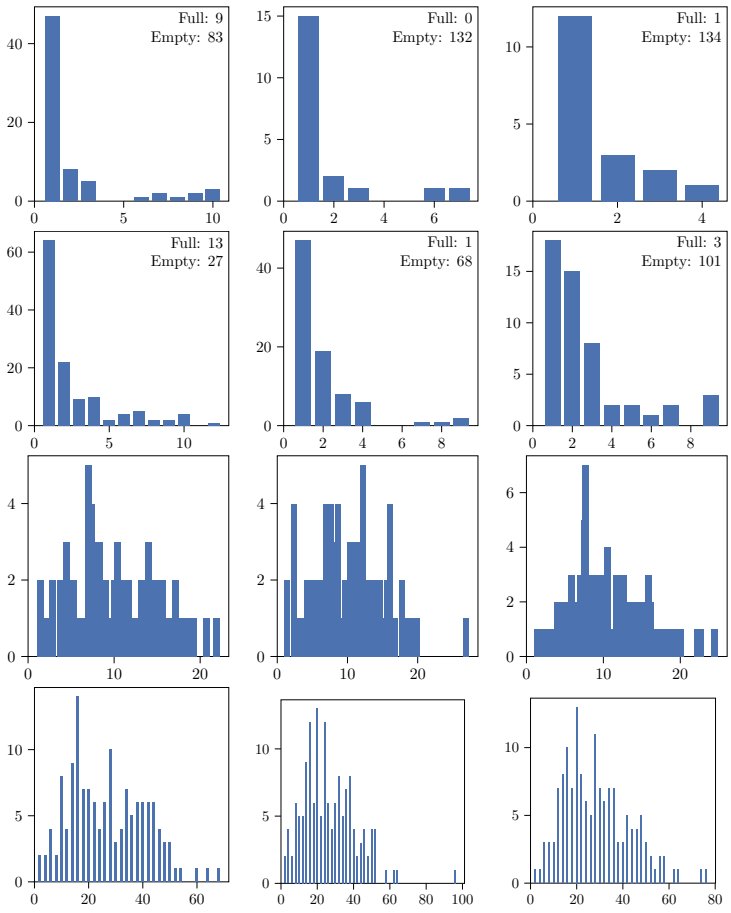


Figure 7.5: Experimental overview regarding the "Rain in Australia" dataset.

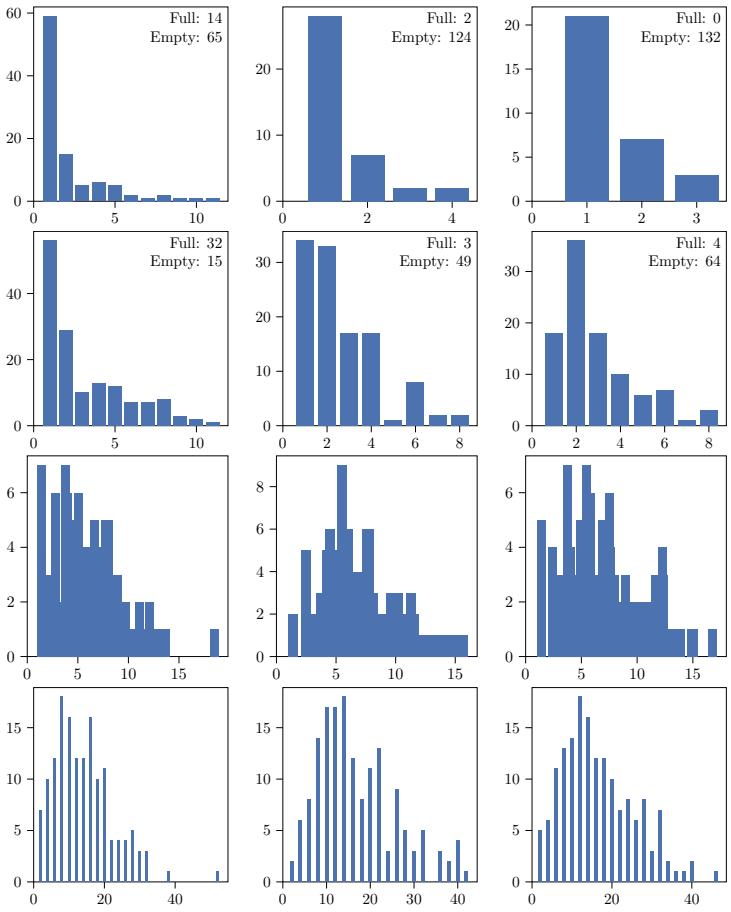


Figure 7.6: Experimental overview regarding the "Cure the Princess" dataset.

7.2 Evaluation of QAUnused

We implement QAUnused in Python as part of the tool Libra.[5] To demonstrate its effectiveness, we evaluated it on neural networks trained on the Adult dataset[6] from the UCI Machine Learning Repository. The dataset assigns to individuals a yearly income greater or smaller than $50k based on personal attributes such as education and occupation but also gender, marital status, or race. The gender is set to be the sensitive input feature. The neural networks were trained with Keras for 50 iterations, using the RMSprop optimizer with the default learning rate, and categorical cross-entropy as the loss function. All networks are open source as part of Libra.

5: github.com/caterinaurban/libra

6: archive.ics.uci.edu/ml/datasets/adult

The experiments were conducted on the Inria Paris cleps infrastructure,[7] on a machine with two 16-core Intel® Xeon® 5218 CPU @ 2.4GHz, 192GB of RAM, and running CentOS 7.7. with linux kernel 3.10.0. For each experiment, we report the average results of five executions to account for the effect of randomness of the environment.

7: paris-cluster-2019.gitlabpages.inria.fr/cleps/cleps-userguide/

7.2.1 Effect of Neural Network Structure on Precision and Scalability.

The precision and scalability of Libra's analysis depend on the analyzed neural network. Table 7.1 shows the result of running Libra on different neural networks with different choices for the abstract domain used by the pre-analysis. Column $|\mathrm{M}|$ refers to the analyzed neural network by the number of its ReLU activations. From top to bottom, the neural networks have the following number of hidden layers and nodes per layer: 2 and 5, 4 and 3, 4 and 5, 4 and 10, and 9 and 5. We configured the pre-analysis with lower bound $\mathrm{L} = 0.5$ and upper bound $\mathrm{U} = 5$. Each column shows the chosen abstract domain. We show here the results for Boxes, Symbolic, DeepPoly, Neurify, and the reduced product deeppoly+neurify+symbolic (*i.e.*, Product in the Table 7.1), which is the most precise of all possible reduced products. The input rows show the average input-space coverage, that is, the average percentage of the input space that Libra was able to analyze with the chosen pre-analysis configuration. The time rows show the average running time.

For all neural networks, Product achieves the highest input-space coverage, an improvement of up to 12.49% over the best coverage obtained with only the abstract domains available in the preliminary version of Libra [73] (*i.e.*, with respect to the DeepPoly domain for $|\mathrm{M}| = 40$). Interestingly, such an improvement comes at the cost of a very modest increase in running time (*i.e.*, just over 1 minute). Indeed, using a more precise abstract domain for the pre-analysis generally results in fewer input space partitions being passed to the backward analysis and, in turn, this reduces the overall running time.

[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

For the smallest neural networks (*i.e.*, $|\mathrm{M}| \in \{10, 12, 20\}$), the Symbolic abstract domain is the second-best choice in terms of input-space coverage. This is likely due to the convex ReLU approximations of DeepPoly and Neurify which in some case produce a negative lower bound (cf. Figure 6.2 and Figure 6.3), while Symbolic always sets the lower bound to zero (cf. Figure 6.1).

$\|M\|$	Boxes	Symbolic	DeepPoly	Neurify	Product	
10	96.81%	**98.72**%	98.37%	98.51%	**99.44**%	input
	6m 32s	**4m 52s**	3m 23s	4m 27s	**4m 40s**	time
12	69.10%	**76.70**%	66.39%	64.58%	**77.29**%	input
	4m 53s	**2m 27s**	2m 0s	1m 31s	**2m 30s**	time
20	41.01%	**56.11**%	56.10%	53.06%	**68.23**%	input
	4m 8s	**9m 7s**	3m 43s	3m 53s	**8m 9s**	time
40	0.35%	34.72%	**38.69**%	41.22%	**51.18**%	input
	1m 3s	7m 2s	**37m 16s**	10m 33s	**38m 27s**	time
45	1.74%	43.78%	51.21%	50.59%	**55.53**%	input
	50s	3m 42s	5m 14s	5m 10s	**6m 22s**	time

Table 7.1: Comparison of the amount of fair input space discovered by different abstract domains of Libra on different neural networks.

Finally, for the largest neural networks (*i.e.*, $|M| \in \{40, 45\}$), it is the structure of the network (rather than its number of ReLU activations) that impacts the precision and scalability of the analysis: for the deep but narrow network (*i.e.*, $|M| = 45$), Libra achieves a higher input-space coverage in a shorter running time than for the shallow but wide network (*i.e.*, $|M| = 40$).

7.2.2 Precision-vs-Scalability Tradeoff.

The configuration of Libra's pre-analysis allows trading-off between precision and scalability. Table 7.2 shows the average results of running Libra on the neural network with 20 ReLUs with different lower and upper bound configurations, and different choices for the abstract domain used by the pre-analysis. Columns L and U show the configured lower and upper bounds. We tried $L \in \{0.5, 0.25\}$ and $U \in \{3, 5\}$. We again show the results for the Boxes, Symbolic, DeepPoly, Neurify abstract domains, and the most precise reduced product domain deeppoly+neurify+symbolic (*i.e.*, Product in Table 7.2).

As expected, decreasing the lower bound L or increasing the upper bound U improves the input-space coverage (input rows) and increases the running time (time rows). We obtain an improvement of up to 12.44% by increasing U from 3 to 5 (with L = 0.25 and Boxes), and up to 42.05% by decreasing L from 0.5 to 0.25 (with U = 5 and Boxes). The smaller is L and the larger is U, the higher is the impact on the running time. Once again, for all lower and upper bound configurations, deeppoly+neurify+symbolic achieves the highest input-space coverage, improving up to 12.08% with respect to DeepPoly with L = 0.5 and U = 5. The improvement is more important for configurations with larger lower bounds.

Notably, Table 7.2 shows that *none among the Symbolic, DeepPoly, and Neurify abstract domains is always more precise than the others*. There are cases where even Symbolic (implemented by [115]) outperforms Neurify

[115]: Wang et al. (2018), 'Formal Security Analysis of Neural Networks using Symbolic Intervals'

L	U	boxes	symbolic	deeppoly	neurify	Product	
0.5	3	**37.88**%	48.78%	49.01%	46.49%	**59.20**%	input
		36s	42s	1m 35s	32s	1m 58s	time
	5	**41.01**%	56.11%	56.15%	53.06%	**68.23**%	input
		4m 8s	9m 10s	3m 47s	3m 57s	8m 16s	time
0.25	3	**70.62**%	83.63%	81.82%	81.40%	**87.04**%	input
		5m 49s	5m 55s	5m 20s	5m 20s	7m 12s	time
	5	**83.06**%	91.67%	91.58%	92.33%	**95.48**%	input
		26m 43s	21m 8s	22m 8s	25m 54s	21m 58s	time

Table 7.2: Comparison on how the fair input space discovered by Libra is affected by different lower and upper bound configurations.

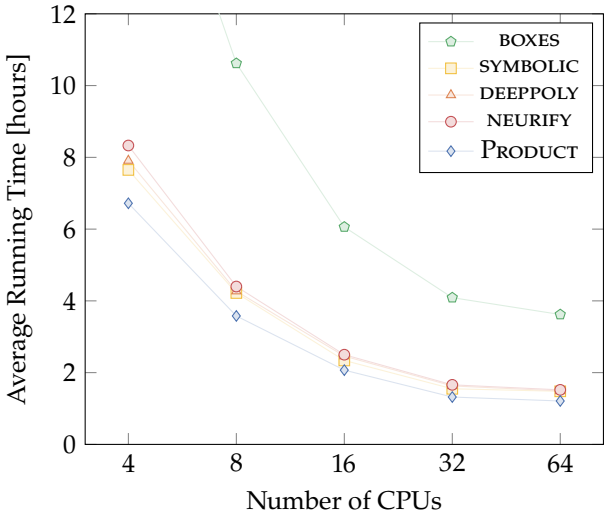


Figure 7.7: Comparison of running times for different number of CPUs.

(implemented by [118] which is the successor of [115] and is believed to be strictly superior to its predecessor), e.g., configuration L = 0.5 and U = 5. We thus argue for using reduced products of abstract domains also in other contexts beyond fairness certification, e.g., verifying local robustness [117, 160] or verifying functional properties of neural networks [122].

[118]: Wang et al. (2018), 'Efficient Formal Safety Analysis of Neural Networks'

[117]: Singh et al. (2019), 'An abstract domain for certifying neural networks'

[160]: Li et al. (2019), 'Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification'

[122]: Katz et al. (2017), 'Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks'

7.2.3 Leveraging Multiple CPUs.

The optimal pre-analysis configuration in terms of precision or scalability depends on the analyzed neural network. In order to push Libra to its limits and obtain 100% input-space coverage on the neural network with 20 ReLUs, we used the configuration auto-tuning mechanism starting with L = 1 and U = 0 (*i.e.*, the most restrictive lower and upper bound configuration) and setting $L_{min} = 0$ and $U_{max} = 20$ (*i.e.*, the most permissive configuration). For all choices of abstract domains, the pre-analysis eventually stabilizes with lower bound L = 0.015625 and upper bound U = 6.

Figure 7.7 compares the average running times for Boxes, Symbolic, DeepPoly, Neurify, and the reduced product deeppoly+neurify+symbolic (*i.e.*, Product) as a function of the number of available CPUs. With Product we obtained a running time improvement of 14.39% over Symbolic, *i.e.*, the fastest domain available in the preliminary version of Libra (a minimum improvement of 11.54% with 16 CPUs, and a maximum improvement of 18.24% with 64 vCPUs). As expected, adding more CPUs always improves Libra running time. The most limited improvement in running time that occurs between 32 CPUs and 64 vCPUs is likely due to the use of hyperthreading as context switches between processes running intense numeric computations produce more overhead.

Table 7.3 additionally shows the estimated percentage of bias detected with each abstract domain, *i.e.*, Libra is able to certify fairness for about 95% of the neural network input space. Note that, the bias estimate depends on the partitioning of the input space computed by the pre-analysis, cf. Section 6.4.2. This explains the different percentages found

|cpu|	Boxes	Symbolic	DeepPoly	Neurify	Product	
4	100%	100%	100%	100%	100%	input
	4.55%	5.23%	5.20%	5.11%	5.42%	bias
	19h 20m 0s	7h 38m 43s	7h 54m 35s	8h 19m 36s	6h 43m 28s	time
8	100%	100%	100%	100%	100%	input
	4.41%	5.16%	5.12%	5.18%	5.46%	bias
	10h 37m 28s	4h 13m 27s	4h 16m 13s	4h 24m 13s	3h 34m 38s	time
16	100%	100%	100%	100%	100%	input
	4.56%	5.19%	5.12%	5.20%	5.34%	bias
	6h 3m 23s	**2h 20m 37s**	2h 27m 31s	2h 30m 4s	**2h 4m 9s**	time
32	100%	100%	100%	100%	100%	input
	4.50%	5.11%	5.10%	5.10%	5.37%	bias
	4h 5m 16s	1h 33m 16s	1h 37m 40s	1h 39m 19s	1h 19m 23s	time
64	100%	100%	100%	100%	100%	input
	4.51%	5.11%	5.20%	5.16%	5.37%	bias
	3h 37m 9s	**1h 28m 38s**	1h 29m 26s	1h 31m 28s	**1h 12m 21s**	time

Table 7.3: Comparison of how the number of available affects the performance of Libra.

even by runs with the same abstract domain. Within the same column, the difference is at most 0.14% on average.

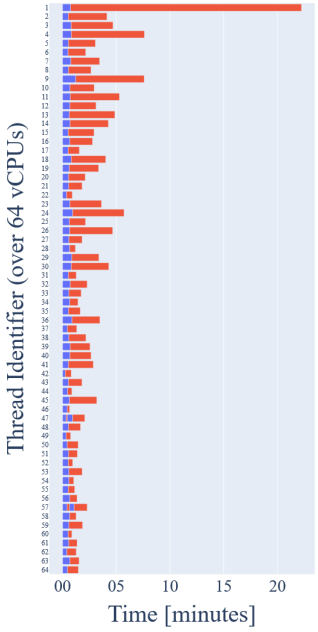


Figure 7.8: Former Task Scheduling

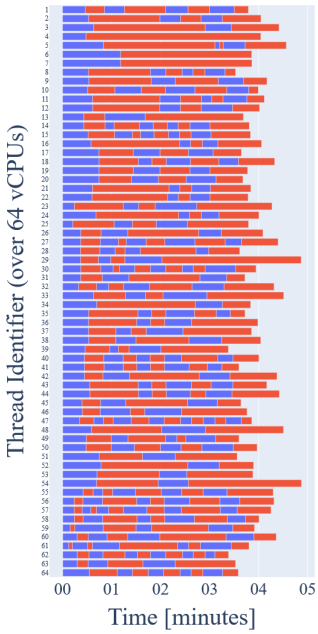


Figure 7.9: Current Task Scheduling

Finally, we remark that *the auto-tuning mechanisms is essential for scalability*. We tried repeating this experiment by directly running Libra with the configuration at which auto-tuning stabilizes, *i.e.*, L = 0.015625 and U = 6. After six days it still had not completed and we had to interrupt it.

Note that, while implementing our quantitative extension to Libra, we also improved greatly the parallelization of the tool. In the original version of Libra(originally presented in [73], without our quantitative extension), feasible partitions are first grouped by activation pattern, *i.e.*, activation patterns that fix more ReLUs are merged with those that fix fewer ReLUs. This way, in principle, the amount of work that the backward analysis has to do is reduced: it only needs to run once for each activation pattern, and can then perform all the checks for bias on each feasible partition. In practice, the abstract domains used in the original implementation prevented the parallelization of all bias checks, which were thus run sequentially. This prevented Librafrom exploiting multi-core architectures effectively. After our extension, we optimize the backward analysis in order to possibly repeat the analysis for the same activation pattern but allowing it to parallelize the bias checks. Figure 7.8 and Figure 7.9 compare the previous and current backward analysis task scheduling on the same analysis instance. Each row in the Gantt diagrams shows computations of the same thread. Blue bars stand for activation pattern computations, while red bars indicate bias check computations, these two bars alternate throughout the whole backward computation for each processor (if possible). As shown in Figure 7.8, the running time was determined almost completely by the task with the most associated bias checks, leaving all the other threads idle from the very beginning. The diagram in Figure 7.9 is more compact, meaning that threads are always running jobs uniformly. Consequently, the backward analysis running time decreases from about 22 to only 5 minutes.

7.3 Summary

This chapter, presenting the evaluation of our quantitative framework on neural networks, concludes part of quantitative verification for extensional properties. We have shown a quantitative framework to quantify the impact of input variables on the output of a program and applied it to the context of neural networks. In the next part, we will study how to extend this framework to intensional properties, which are properties that depend on the internal state of the program. We will quantify the impact of input variables on the number of iterations of a program.

Ce chapitre, qui présente l'évaluation de notre cadre quantitatif sur les réseaux neuronaux, conclut la partie dédiée à la vérification quantitative des propriétés extensionnelles. Nous avons montré un cadre quantitatif permettant de mesurer l'impact des variables d'entrée sur le résultat d'un programme et l'avons appliqué au contexte des réseaux neuronaux. Dans la prochaine partie, nous étudierons comment étendre ce cadre aux propriétés intentionnelles, qui sont des propriétés dépendant de l'état interne du programme. Nous quantifierons l'impact des variables d'entrée sur le nombre d'itérations d'un programme.

Quantitative Verification of Intensional Properties

Quantitative Static Timing Analysis

8

This chapter presents a static analysis for quantifying the impact of input variables on the number of iterations of a program. The analysis employs a semantic global loop bound analysis to derive an over-approximation of the impact quantity. First, we introduce the maximal trace semantics augmented with a global loop counter to model the number of iterations of all the loops of a program. Then, we show an abstract global loop bound analysis, followed by the impact quantification, which is employed to verify the corresponding k-impact property. This chapter is based on the work presented at the 31st Static Analysis Symposium (SAS 2024) [48]. Chapter 9 will discuss some implementation details and optimizations, as well as the evaluation of the analysis presented on this chapter.

[48]: Mazzucato et al. (2024), 'Quantitative Static Timing Analysis'

Ce chapitre présente une analyse statique pour quantifier l'impact des variables d'entrée sur le nombre d'itérations d'un programme. L'analyse utilise une analyse sémantique des bornes globales des boucles pour dériver une sur-approximation de la quantité d'impact. Tout d'abord, nous introduisons la sémantique des traces maximales augmentée d'un compteur global de boucles pour modéliser le nombre d'itérations de toutes les boucles d'un programme. Ensuite, nous présentons une analyse abstraite des bornes globales des boucles, suivie par la quantification de l'impact, qui est utilisée pour vérifier la propriété d'impact borné par k correspondante. Ce chapitre est basé sur les travaux présentés lors du 31e Symposium sur l'analyse statique (SAS 2024) [48]. Chapter 9 discutera de certains détails d'implémentation et optimisations, ainsi que de l'évaluation de l'analyse présentée dans ce chapitre.

8.1 Global Loop Bound Semantics

We present a concrete semantics extended with the global loop counter for the simple imperative language of Section 2.3, reported on the side for convenience.

We recall that the variables of the program are represented by the set $\mathbb{Var}$, where the finite set $\Delta \subseteq \mathbb{Var}$ contains the input variables.

$$
\begin{aligned}
e &::= v \mid \mathtt{x} \mid e + e \mid e - e \\
b &::= e \leq v \mid e = v \mid b \wedge b \mid \neg b \\
st &::= \mathtt{skip} \mid \mathtt{x} := e \mid \mathtt{assert}\ b \\
&\quad \mid \mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st \\
&\quad \mid \mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done} \\
&\quad \mid st; st \\
P &::= \mathtt{entry}\ st
\end{aligned}
$$

In order to accumulate the number of iterations of (all) loops in the program, we introduce a variable $\mathtt{nit} \notin \mathbb{Var}$, called the *global loop counter*. Consequently, we extend the program states to include the value of the global loop counter, *i.e.*, Σ maps from variables in $\mathbb{Var}^+ \stackrel{\text{def}}{=} \mathbb{Var} \cup \{\mathtt{nit}\}$ to values in $\mathbb{V}$. As a consequence, $\mathtt{nit}$ is not an input variable either, *i.e.*, $\mathtt{nit} \notin \Delta$. We extend the maximal trace semantics Λ to include the global loop counter as follows:

For simplicity, we omit program locations in the maximal trace semantics of Definition 8.1.1

Prog. 8.1: Program `Add`, computing the sum of two numbers `x` and `y` into `z`.

```
def Add(p, z, m, x, n, y):
    r = min(p, m)
    s = min(p, n)
    if (r < s):
      t = p - s
      q = s - r
      # i = 0
      # a = 0
      for (; r > 0; r--):
        # s = x[i]
        # w = y[i]
        # z[i] = s + w + a
        # i = i + 1
        # a = (w < a) ||
        #   (s + w < s) ||
        #   (s + w + a < s)
      do:
        # r = y[i]
        # b = (r < a) ||
        #   (r + a < r)
        # z[i] = r + a
        # i = i + 1
        q--
        # a = b
      while (q > 0)
  else:
    t = p - r
    q = r - s
    # i = 0
    # b = 0
    for (; s > 0; s--):
      # r = x[i]
      # w = y[i]
      # z[i] = r + w + b
      # i = i + 1
      # b = (w < b) ||
      #   (r + w < r) ||
      #   (r + w + b < r)
    for (; q > 0; q--):
      # r = x[i]
      # z[i] = r + b
      # i = i + 1
      # b = (r < b) ||
      #   (r + b < r)
    if (t > 0):
      # z[i] = b
      while (t > 0):
        # i = i + 1
        t--
        # if (t > 0):
          # z[i] = 0
```

Table 8.1: A few executions to show how many times the program `Add` iterates over the loops. The symbol * denotes any possible value.

`Add(p,`	`z,`	`m,`	`x,`	`n,`	`y)`	$\rightsquigarrow$	global number of iterations
`Add(0,`	`*,`	`0,`	`*,`	`0,`	`*)`	$\rightsquigarrow$	0
`Add(1,`	`*,`	`0,`	`*,`	`0,`	`*)`	$\rightsquigarrow$	1
`Add(2,`	`*,`	`0,`	`*,`	`0,`	`*)`	$\rightsquigarrow$	2
`Add(0,`	`*,`	`1,`	`*,`	`0,`	`*)`	$\rightsquigarrow$	0
`Add(1,`	`*,`	`1,`	`*,`	`0,`	`*)`	$\rightsquigarrow$	1
`Add(2,`	`*,`	`1,`	`*,`	`0,`	`*)`	$\rightsquigarrow$	2
`Add(0,`	`*,`	`0,`	`*,`	`1,`	`*)`	$\rightsquigarrow$	0
`Add(1,`	`*,`	`0,`	`*,`	`1,`	`*)`	$\rightsquigarrow$	1
`Add(2,`	`*,`	`0,`	`*,`	`1,`	`*)`	$\rightsquigarrow$	2

Definition 8.1.1 (Maximal Trace Semantics with the Global Loop Counter) *The semantics* $\Lambda_{\texttt{nit}} \in \wp(\Sigma^{+\infty}) \to \wp(\Sigma^{+\infty})$ *is defined as:*

$$\Lambda_{\texttt{nit}}[\![\texttt{skip}]\!]T \stackrel{\text{def}}{=} \{ss \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in T\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{x} := e]\!]T \stackrel{\text{def}}{=} \{ss[\texttt{x} \leftarrow v] \cdot \sigma \mid s \in \Sigma \wedge v \in \mathbb{A}[\![e]\!]s \wedge \sigma \in \Sigma^{\star\infty} \wedge s[\texttt{x} \leftarrow v] \cdot \sigma \in T\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{assert } b]\!]T \stackrel{\text{def}}{=} \{s \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in T\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]T \stackrel{\text{def}}{=} \{s \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in \Lambda_{\texttt{nit}}[\![st]\!]T\} \cup \{s \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \wedge s \cdot \sigma \in \Lambda_{\texttt{nit}}[\![st']\!]T\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]T \stackrel{\text{def}}{=} \operatorname{lfp} \mathrm{F}$$
$$\mathrm{F}(X) \stackrel{\text{def}}{=} \{s \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \wedge s \cdot \sigma \in T\} \cup \{s \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in \Lambda_{\texttt{nit}}[\![st]\!]\textsc{Decr}^{\leftarrow}_{\texttt{nit}}(X)\}$$

$$\Lambda_{\texttt{nit}}[\![st; st']\!]T \stackrel{\text{def}}{=} \Lambda_{\texttt{nit}}[\![st]\!](\Lambda_{\texttt{nit}}[\![st']\!]T)$$

$$\Lambda_{\texttt{nit}}[\![\texttt{entry } st]\!]T \stackrel{\text{def}}{=} \Lambda_{\texttt{nit}}[\![st]\!]\{s[\texttt{nit} \leftarrow 0] \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in T\}$$

The semantics $\Lambda_{\texttt{nit}}$ increments the value of the global loop counter `nit` at each iteration. Therefore, the global loop counter `nit` contains the number of iterations of all the loops in the program.

Example 8.1.1 Consider the Program 8.1, called `Add`, presented on the side. The goal of program `Add` is to compute the sum of two given numbers `x` and `y`, storing the result into `z`. The input variables `x`, `y`, and the output `z` are represented in the form of arrays, respectively of length `m`, `n`, and `p`. The program `Add` computes the column addition of the two input arrays. For instance, assuming `x` = [3 8], `y` = [4] and the size of `z` is 3, then `Add`(3, `z`, 2, [3 8], 1, [4]) computes:

$$\begin{array}{r} [3\ \ 8] + \\ \quad [4] = \\ \hline [0\ \ 4\ \ 2] \end{array}$$

where the result is stored back into `z`, available in the calling context of the function. The statements that are not relevant to the number of iterations of loops are commented out (cf. #). As we will discover in the

$\Lambda_{\texttt{nit}}[\![\texttt{Add}]\!]$

$\langle\, \texttt{p}, \texttt{m}, \texttt{n}, \ldots, \texttt{nit} \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle \texttt{p}, \texttt{m}, \texttt{n}, \ldots \texttt{nit} \rangle$
$\langle\, 0, 0, 0, \ldots, 0 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 0, 0, 0, \ldots, 0 \,\rangle$
$\langle\, 1, 0, 0, \ldots, 1 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 1, 0, 0, \ldots, 0 \,\rangle$
$\langle\, 2, 0, 0, \ldots, 2 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 2, 0, 0, \ldots, 0 \,\rangle$
$\langle\, 0, 1, 0, \ldots, 0 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 0, 1, 0, \ldots, 0 \,\rangle$
$\langle\, 1, 1, 0, \ldots, 1 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 1, 1, 0, \ldots, 0 \,\rangle$
$\langle\, 2, 1, 0, \ldots, 2 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 2, 1, 0, \ldots, 0 \,\rangle$
$\langle\, 0, 0, 1, \ldots, 0 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 0, 0, 1, \ldots, 0 \,\rangle$
$\langle\, 1, 0, 1, \ldots, 1 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 1, 0, 1, \ldots, 0 \,\rangle$
$\langle\, 2, 0, 1, \ldots, 2 \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, 2, 0, 1, \ldots, 0 \,\rangle$
$\vdots$		$\vdots$
$\langle\, \text{a}, \text{b}, \text{c}, \ldots, \text{a} \,\rangle$	$\longrightarrow \cdots \longrightarrow$	$\langle\, \text{a}, \text{b}, \text{c}, \ldots, 0 \,\rangle$

Table 8.2: Maximal traces of the program `Add`.

next chapter, these irrelevant statements are pruned by the syntactic dependency analysis. Table 8.1 shows a few executions of the program `Add`.

The semantics of the program `Add`, cf. Definition 8.1.1, is defined backwards from any state with the global loop counter `nit` initialized to 0. Thus, starting from final states where $\texttt{nit} = 0$, the semantics of the program `Add` decrements (in a backward fashion) the global loop counter `nit` at each iteration of the loops. The backward decrement is defined as $\textsc{Decr}^{\leftarrow}_{\texttt{nit}}(S) \stackrel{\text{def}}{=} \{s \in \Sigma \mid s[\texttt{nit} \leftarrow s(\texttt{nit}) - 1] \in S\}$. After the computation of the semantics of a given program, we obtain the number of iterations of all the loops in the program. Table 8.2 shows the maximal traces of the executions of the program `Add`, intermediate states are omitted for brevity.

We define the k-impact property, which quantifies the impact of input variables on the global number of iterations of a program.

Definition 8.1.2 (k-Bounded Impact Property with Loop Iterations) *Let P be a program, $\texttt{W}$ the set of input variables of interest, $\textsc{Impact}_{\texttt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{V}^{+\infty}_{\geq 0}$ an impact quantifier, and $k \in \mathbb{V}^{+\infty}_{\geq 0}$ the threshold. The k-bounded impact property with loop iterations is defined as:*

$$\mathscr{B}^{\otimes k}_{\textsc{Impact}_{\texttt{W}}} \stackrel{\text{def}}{=} \{\Lambda_{\texttt{nit}} \in \wp(\Sigma^{+\infty}) \mid \textsc{Impact}_{\texttt{W}}(\Lambda_{\texttt{nit}}) \otimes k\}$$

where $\otimes \in \{\leq, \geq\}$.

To compute the impact of input variables on the global number of iterations of a program, we need to instantiate the k-bounded impact property with an impact quantifier. As a design choice, we choose the Range quantifier among the quantifiers defined in Section 4.1 (The k-Bounded Impact Property). The range quantifier instrumented with the global loop counter `nit` as output variables and the identity function as output observer ρ is defined as follows:

Definition 8.1.3 (Range with `nit`) *Given the set of input variables of interest $\texttt{W} \in \wp(\Delta)$, the impact quantifier $\textsc{Range}_{\texttt{W}} \in \wp(\Sigma^{+\infty}) \to \mathbb{R}^{+\infty}_{\geq 0}$*

instrumented with the global loop counter `nit` *is defined as*

$$\textsc{Range}_{\mathtt{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \textsc{Distance}(\{\sigma_0(\mathtt{nit}) \mid \sigma \in T \wedge \sigma_0 =_{\Delta \setminus \mathtt{W}} s_0\})$$

Let $\mathscr{B}^{\trianglerighteq k}_{\textsc{Range}_{\mathtt{W}}}$ be the k-bounded impact property with loop iterations when instantiated with the Range impact quantifier. Formally:

$$\mathscr{B}^{\trianglerighteq k}_{\textsc{Range}_{\mathtt{W}}} \stackrel{\text{def}}{=} \{\Lambda_{\mathtt{nit}} \in \wp(\Sigma^{+\infty}) \mid \textsc{Range}_{\mathtt{W}}(\Lambda_{\mathtt{nit}}) \trianglerighteq k\}$$

Example 8.1.2 Consider the trace semantics $\Lambda_{\mathtt{nit}}[\![\mathtt{Add}]\!]$ defined in Table 8.2. For brevity, we consider states as $\Sigma = \{\langle p, m, n, nit\rangle \mid p, m, n, nit \in [0, u]\}$, where p is the value of `p`, m of `m`, n of `n`, and nit of `nit`, all ranging in the interval $[0, u]$. Assuming we are interested in the impact of the input variable `p`, $\textsc{Range}_{\{\mathtt{p}\}}(\Lambda_{\mathtt{nit}}[\![\mathtt{Add}]\!])$ considers all possible input values $s_0 \in \Sigma|_\Delta$, on the left of Table 8.2. We collect all the input states that agree with the input value s_0 on the other input variables, cf. `m` and `n`. Specifically, we collect all the input states s such that $s =_{\{\mathtt{m},\mathtt{n}\}} s_0$. For instance, for the input value $s_0 = \langle 0, 0, 0, 0\rangle$, we collect the set of states $\{\langle p, 0, 0, nit\rangle \mid p, nit \in [0, u]\}$ where everything but `p` and `nit` is fixed. From this set of states, we consider only the values of the counter `nit`. For instance, regarding the input $s_0 = \langle 0, 0, 0, 0\rangle$, we collect the values $[0, u]$. Then, we apply the operator $\textsc{Distance}([0, u]) = \sup [0, u] - \inf [0, u] = u$. For all the input values, the maximum value is taken; we obtain $\textsc{Range}_{\{\mathtt{p}\}}(\Lambda_{\mathtt{nit}}[\![\mathtt{Add}]\!]) = u$.

Let us analyze the impact of other input variables, *e.g.*, the input variable `m`. By considering the input value $s_0 = \langle 0, 0, 0, 0\rangle$, we collect the dependencies that start from states in $\{\langle 0, m, 0, nit\rangle \mid m, nit \in [0, u]\}$. As we notice from left columns of Table 8.2, the number of iterations starting from any of these states is always 0. Hence, $\textsc{Distance}(\{0\}) = \sup \{0\} - \inf \{0\} = 0$. For all the input values, we obtain: $\textsc{Range}_{\{\mathtt{m}\}}(\Lambda_{\mathtt{nit}}[\![\mathtt{Add}]\!]) = \textsc{Range}_{\{\mathtt{n}\}}(\Lambda_{\mathtt{nit}}[\![\mathtt{Add}]\!]) = 0$. We conclude that:

$$\mathtt{Add} \models \mathscr{B}^{\leq u}_{\textsc{Range}_{\{\mathtt{p}\}}}, \qquad \mathtt{Add} \models \mathscr{B}^{\leq 0}_{\textsc{Range}_{\{\mathtt{n}\}}}, \text{ and } \qquad \mathtt{Add} \models \mathscr{B}^{\leq 0}_{\textsc{Range}_{\{\mathtt{m}\}}}$$

As a consequence, we can infer that there exist two executions starting from a different value for the input variable `p` that differ in the global number of iterations by at most u. On the contrary, any variation in the input variables `m` and `n` does not affect the global number of iterations.

Importantly, our goal is to quantify the impact of input variables on the global number of iterations. As a consequence, only the initial and final states are relevant, *i.e.*, where the program loop counter contains the global number of iterations. In fact, we reduced the verification of an *intensional* property about the global number of iterations of a program to the verification of an *extensional* property by collecting the global number of iterations into input-output dependencies. Thus, the hierarchy of semantics (Sections 3.4 and 3.5) applies accordingly from the maximal trace semantics $\Lambda_{\mathtt{nit}}$. We retrieve the collecting semantics $\Lambda^{\mathbb{C}}_{\mathtt{nit}} \stackrel{\text{def}}{=} \{\Lambda_{\mathtt{nit}}\}$, cf. Definition 2.2.3, and the dependency semantics $\Lambda^{\rightsquigarrow}_{\mathtt{nit}} \stackrel{\text{def}}{=} \alpha^{\rightsquigarrow}(\Lambda^{\mathbb{C}}_{\mathtt{nit}})$, cf. Definition 3.4.2. Regarding Example 8.1.1, the left and right columns

of Table 8.2 contain, in fact, the input-output dependencies with the additional variable for the global loop counter `nit`.

Next, we show that the dependency semantics $\Lambda^{\overset{\rightsquigarrow}{\texttt{nit}}}$ is sound and complete for validating $\mathscr{B}^{\circledS k}_{\textsc{Range}_W}$.

Def. 3.4.2 (Dependency Semantics)

$$\Lambda^{\overset{\rightsquigarrow}{}} \overset{\text{def}}{=} \{\{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in \Lambda\}\}$$

Lemma 8.1.1

$$\Lambda^{\mathbb{C}}_{\texttt{nit}}[\![P]\!] \subseteq \mathscr{B}^{\circledS k}_{\textsc{Range}_W} \Leftrightarrow \Lambda^{\overset{\rightsquigarrow}{\texttt{nit}}}[\![P]\!] \subseteq \alpha^{\rightsquigarrow}(\mathscr{B}^{\circledS k}_{\textsc{Range}_W})$$

Proof. The proof is similar to the proof of Lemma 4.1.7 where $\rho = \mathrm{id}$. □

Lemma 4.1.7

$$\Lambda^{\mathbb{C}}[\![P]\!] \subseteq \mathscr{B}^{\circledS k}_{\textsc{Range}_W} \Leftrightarrow \Lambda^{\rho}[\![P]\!] \subseteq \alpha^{\rho}(\alpha^{\rightsquigarrow}(\mathscr{B}^{\circledS k}_{\textsc{Range}_W}))$$

Note that we do not consider the output observer ρ, hence the output-abstraction semantics α^{ρ} is not needed. Furthermore, we notice from the example above that as the loop counter `nit` holds the global number of iterations in the initial states of the dependency semantics, we can further abstract the dependency semantics to the backward co-reachability semantics by removing output states as not necessary for $\mathscr{B}^{\circledS k}_{\textsc{Range}_W}$.

8.2 Backward Reachability Semantics

We define the pair of right-left adjoints $\langle \alpha^{\bar{\mathrm{r}}}, \gamma^{\bar{\mathrm{r}}} \rangle$ to abstract the dependency semantics into the backward co-reachability semantics.

Definition 8.2.1 (Right-Left Adjoints for the Backward Reachability Semantics)

$$\begin{aligned}
\alpha^{\bar{\mathrm{r}}} &\in \wp(\wp(\Sigma \times \Sigma^{\perp})) \to \wp(\wp(\Sigma)) \\
\alpha^{\bar{\mathrm{r}}}(W) &\overset{\text{def}}{=} \{\{\sigma_0 \mid \langle \sigma_0, \sigma_\omega \rangle \in D\} \mid D \in W\} \\
\gamma^{\bar{\mathrm{r}}} &\in \wp(\wp(\Sigma)) \to \wp(\wp(\Sigma \times \Sigma^{\perp})) \\
\gamma^{\bar{\mathrm{r}}}(R) &\overset{\text{def}}{=} \{S \times Q \mid S \in R \wedge Q \subseteq \Sigma^{\perp}\}
\end{aligned}$$

The function $\alpha^{\bar{\mathrm{r}}}$ abstracts away output states of any dependency, preserving the set-structure of W. The concretization $\gamma^{\bar{\mathrm{r}}}$ yields all the semantics that share the same input states of, at least, one of the set of semantics in R.

Theorem 8.2.1 *The two adjoints* $\langle \alpha^{\bar{\mathrm{r}}}, \gamma^{\bar{\mathrm{r}}} \rangle$ *form a* Galois insertion*:*

$$\langle \wp(\wp(\Sigma^{+\infty})), \subseteq \rangle \underset{\alpha^{\bar{\mathrm{r}}}}{\overset{\gamma^{\bar{\mathrm{r}}}}{\leftrightarrows}} \langle \wp(\wp(\Sigma)), \subseteq \rangle$$

Proof. We need to show that $\alpha^{\bar{\mathrm{r}}}(W) \subseteq R \Leftrightarrow W \subseteq \gamma^{\bar{\mathrm{r}}}(R)$. First, we show the direction ($\Rightarrow$). Assuming $\alpha^{\bar{\mathrm{r}}}(W) \subseteq R$, we have that $\gamma^{\bar{\mathrm{r}}}(R)$ contains all the possible semantics that share the same set of input states of at least one of the semantics in R. Thus, $\gamma^{\bar{\mathrm{r}}}(R)$ also contains all the semantics in W, *i.e.*, $W \subseteq \gamma^{\bar{\mathrm{r}}}(R)$. To show ($\Leftarrow$), we assume $W \subseteq \gamma^{\bar{\mathrm{r}}}(R)$. It is easy to note that $\alpha^{\bar{\mathrm{r}}}(\gamma^{\bar{\mathrm{r}}}(R)) = R$ since the concretization maintains the same set

of input states and the abstraction removes only output states. Hence, by monotonicity of $\alpha^{\bar{r}}$, we obtain $\alpha^{\bar{r}}(W) \subseteq \alpha^{\bar{r}}(\gamma^{\bar{r}}(R)) = R$. □

We now derive the *backward co-reachability semantics* $\Lambda^{\bar{r}}_{\texttt{nit}}$ as an abstraction of the dependency semantics.

Definition 8.2.2 (Backward Co-Reachability Semantics) *The* backward co-reachability semantics $\Lambda^{\bar{r}}_{\texttt{nit}} \in \wp(\wp(\Sigma))$ *is defined as:*

$$\Lambda^{\bar{r}}_{\texttt{nit}} \stackrel{\text{def}}{=} \alpha^{\bar{r}}(\Lambda^{\rightsquigarrow}_{\texttt{nit}}) = \{\, \{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}\} \,\}$$

The next result shows that the backward co-reachability semantics $\Lambda^{\bar{r}}_{\texttt{nit}}$ allows a sound and complete verification of $\mathcal{B}^{\otimes k}_{\textsc{Range}_W}$.

Lemma 8.2.2

$$\Lambda^{\mathbb{C}}_{\texttt{nit}}[\![P]\!] \subseteq \mathcal{B}^{\otimes k}_{\textsc{Range}_W} \Leftrightarrow \Lambda^{\bar{r}}_{\texttt{nit}}[\![P]\!] \subseteq \alpha^{\bar{r}}(\alpha^{\rightsquigarrow}(\mathcal{B}^{\otimes k}_{\textsc{Range}_W}))$$

Proof. The implication ($\Rightarrow$) follows from the monotonicity of $\alpha^{\bar{r}}$, as an implication from the fact that the two adjoints $\langle \alpha^{\bar{r}}, \gamma^{\bar{r}} \rangle$ form a Galois connection (cf. Theorem 8.2.1), and Definition 8.2.2 of $\Lambda^{\bar{r}}_{\texttt{nit}}$. Obtaining $\Lambda^{\mathbb{C}} \subseteq \mathcal{N}_W \Rightarrow \alpha^{\bar{r}}(\Lambda^{\mathbb{C}}) \subseteq \alpha^{\bar{r}}(\mathcal{N}_W) \Rightarrow \Lambda^{\bar{r}}_{\texttt{nit}} \subseteq \alpha^{\bar{r}}(\mathcal{N}_W)$. Regarding the other implication ($\Leftarrow$), from Definition 8.2.2 of $\Lambda^{\bar{r}}_{\texttt{nit}}$ and the property of Theorem 8.2.1, we obtain $\Lambda^{\bar{r}}_{\texttt{nit}} \subseteq \alpha^{\bar{r}}(\mathcal{N}_W) \Rightarrow \alpha^{\bar{r}}(\Lambda^{\mathbb{C}}) \subseteq \alpha^{\bar{r}}(\mathcal{N}_W) \Rightarrow \Lambda^{\mathbb{C}} \subseteq \gamma^{\bar{r}}(\alpha^{\bar{r}}(\mathcal{N}_W))$, which can be written as $\Lambda \in \gamma^{\bar{r}}(\alpha^{\bar{r}}(\mathcal{N}_W))$ by the definition of $\Lambda^{\mathbb{C}}$. By Definition 8.2.1 of $\gamma^{\bar{r}}$ it follows that $\{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in \Lambda\} \in \alpha^{\bar{r}}(\mathcal{N}_W)$. Finally, by application of Definition 8.2.1 of $\alpha^{\bar{r}}$ we obtain $\Lambda \in \mathcal{N}_W$. The conclusion $\Lambda^{\mathbb{C}} \subseteq \mathcal{N}_W$ trivially follows from the definition of the subset inclusion ($\subseteq$). □

8.3 Static Analysis for Global Loop Bound

This section presents a sound computable static analysis to automatically compute the global number of iterations. An over-approximation of the global loop counter is computed by analyzing the program backwards. The soundness of the approach leverages: (1) an abstract domain of conjunctions of linear constraints, (2) a sound global loop bound analysis to collect the dependencies of the loop counter `nit` from the input variables, and (3) a linear programming encoding as a sound implementation of the impact Range, called $\texttt{Range}^{\texttt{nit}}$.

8.3.1 Conjunctions of Linear Constraints

We define the numerical abstract state domain used in the global loop bound analysis. In principle, our abstract domain could be any convex abstract domain subsumed by the polyhedra domain [56], such as the interval domain [56], octagon domain [71], or the polyhedra domain itself; see Section 2.3.5 for an overview of numerical abstract domains.

[56]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'

[71]: Miné (2006), 'The octagon abstract domain'

The elements of the abstract domain are conjunctions of linear constraints of the form:

$$c_1 \cdot \mathtt{x}_1 \ + \ \ldots \ + \ c_n \cdot \mathtt{x}_n \ + \ c_{n+1} \ \geq \ 0$$

where $\mathtt{x}_j \in \mathbb{V}\mathtt{ar}^+$ are variables and $c_j \in \mathbb{V}$ are constant values. For better readability, we avoid writing the constant-variable multiplication term $c_i \cdot \mathtt{x}_i$ when $c_i = 0$; and we abuse the notation, *e.g.*, for the constraint $\mathtt{x}_1 = \mathtt{x}_2$, to denote the conjunction of the two linear constraints $\mathtt{x}_1 - \mathtt{x}_2 \geq 0$ and $\mathtt{x}_2 - \mathtt{x}_1 \geq 0$. The abstract domain is a lattice $\langle \mathbb{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ equipped with a concretization function $\gamma \in \mathbb{D} \to \wp(\Sigma)$, defined as follows:

Definition 8.3.1 (Concretization of Linear Constraints) *The concretization γ of conjunctions of linear constraints $d^\natural \in \mathbb{D}$ is defined as:*

$$\gamma(d^\natural) \stackrel{\text{def}}{=} \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall\, (c_1 \cdot \mathtt{x}_1 + \ldots + c_n \cdot \mathtt{x}_n + c_{n+1} \geq 0) \in d^\natural. \\ c_1 \cdot s(\mathtt{x}_1) + \ldots + c_n \cdot s(\mathtt{x}_n) + c_{n+1} \geq 0 \end{array} \right\}$$

Additionally, in order to be effectively used in the context of the global loop bound analysis, we assume:

(i) an operator $\mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!] \in \mathbb{D} \to \mathbb{D}$ to substitute the variable $\mathtt{x} \in \mathbb{V}\mathtt{ar}^+$ with the expression e,
(ii) an operator $\mathtt{Filter}[\![b]\!] \in \mathbb{D} \to \mathbb{D}$ to handle boolean expressions b,
(iii) a widening operator $\nabla \in \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ to ensure termination of the analysis, and
(iv) a project operator $\mathtt{Proj}_{\mathtt{W}} \in \mathbb{D} \to \mathbb{D}$ to remove the input variables $\mathtt{W}$ from the given abstract state.

These requirements are satisfied by any of the commonly used numerical abstract domains [56, 71].

[56]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'
[71]: Miné (2006), 'The octagon abstract domain'

8.3.2 Global Loop Bounds

The global loop bound semantics $\Lambda^{\leftarrow}_{\mathtt{nit}} \in \mathbb{D}$ is a backward abstract semantics that generates an invariant over input variables and the global loop counter $\mathtt{nit}$. During the backward analysis, the value of $\mathtt{nit}$ increases from 0 to an over-approximation of the possible global range of iterations. As a consequence, the pre-condition invariant generated by the backward semantics $\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]$ over-approximates relations between $\mathtt{nit}$ and the initial values of variables in $\mathbb{V}\mathtt{ar}$. The concretization function $\gamma^{\leftarrow} \in \mathbb{D} \to \wp(\wp(\Sigma))$ maps an abstract state to a set of sets of input states. Its goal is to preserve the relations between input values and the global loop counter $\mathtt{nit}$. Formally:

$$\gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]) \stackrel{\text{def}}{=} \wp(\gamma(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])) \tag{8.1}$$

The concretization $\gamma^{\leftarrow}$ restores all possible input states that satisfy the relations between the input variables and the global loop counter $\mathtt{nit}$. For the backward semantics $\Lambda^{\leftarrow}_{\mathtt{nit}}$ to be sound, we require the concretization $\gamma^{\leftarrow}$ to over-approximate the backward co-reachability semantics $\Lambda^{\bar{r}}_{\mathtt{nit}}$. Formally, for all programs P, it should hold that:

$$\Lambda^{\bar{r}}_{\mathtt{nit}}[\![P]\!] \ \subseteq \ \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]) \tag{8.2}$$

$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{skip}]\!]d^{\natural} \stackrel{\text{def}}{=} d^{\natural}$$
$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{x} := e]\!]d^{\natural} \stackrel{\text{def}}{=} \mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^{\natural}$$
$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{assert}\ b]\!]d^{\natural} \stackrel{\text{def}}{=} \mathtt{Filter}[\![b]\!]d^{\natural}$$
$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]d^{\natural} \stackrel{\text{def}}{=}$$
$$\mathtt{Filter}[\![b]\!](\Lambda^{\leftarrow}_{\mathtt{nit}}[\![st]\!]d^{\natural}) \sqcup \mathtt{Filter}[\![\neg b]\!](\Lambda^{\leftarrow}_{\mathtt{nit}}[\![st']\!]d^{\natural})$$
$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]d^{\natural} \stackrel{\text{def}}{=} \lim_{n} F^{\natural}_{n}$$
$$F^{\natural}_{0} \stackrel{\text{def}}{=} d$$
$$F^{\natural}_{n+1} \stackrel{\text{def}}{=} F^{\natural}_{n} \triangledown F^{\natural}(F^{\natural}_{n})$$
$$F^{\natural}(a) \stackrel{\text{def}}{=} \mathtt{Filter}[\![\neg b]\!]d$$
$$\sqcup\ \mathtt{Filter}[\![b]\!](\Lambda^{\leftarrow}_{\mathtt{nit}}[\![st]\!](\mathtt{Subs}[\![\mathtt{nit} \leftarrow \mathtt{nit} - 1]\!]a)) \tag{8.3}$$
$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![st;\ st']\!]d^{\natural} \stackrel{\text{def}}{=} \Lambda^{\leftarrow}_{\mathtt{nit}}[\![st]\!](\Lambda^{\leftarrow}_{\mathtt{nit}}[\![st']\!]d^{\natural})$$

$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{entry}\ st]\!] \stackrel{\text{def}}{=} \Lambda^{\leftarrow}_{\mathtt{nit}}[\![st]\!](\mathtt{nit} = 0) \tag{8.4}$$

Figure 8.1: Global loop bound semantics.

The soundness condition, cf. Equation 8.2, allows any sound global loop bound analysis $\Lambda^{\leftarrow}_{\mathtt{nit}}$ to verify the k-bounded impact property $\mathscr{B}^{\Subset k}_{\text{RANGE}_M}$.

A possible candidate semantics for the global loop bound analysis $\Lambda^{\leftarrow}_{\mathtt{nit}} \in \mathbb{D}$ is defined in Figure 8.1. The semantics $\Lambda^{\leftarrow}_{\mathtt{nit}}$ is a *backward co-reachability semantics* instrumented to increment the loop counter `nit` at each loop iteration, cf. Equation 8.3. The loop counter `nit` is handled semantically in the abstract domain without loss of precision, initialized to 0 at the program exit, cf. Equation 8.4. The rest of the semantics is classical. The following result states the soundness of the backward semantics defined in Figure 8.1.

Def. 8.2.2 (Backward Co-Reachability Semantics)

$$\Lambda^{\overline{r}} \stackrel{\text{def}}{=} \{\, \{\sigma_0 \mid \sigma \in \Lambda_{\mathtt{nit}}\} \,\}$$

Def. 8.1.1 (Maximal Trace Semantics with the Global Loop Counter)

$$\Lambda_{\mathtt{nit}}[\![\mathtt{skip}]\!]T \stackrel{\text{def}}{=} \{ss \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in T\}$$
$$\Lambda_{\mathtt{nit}}[\![\mathtt{x} := e]\!]T \stackrel{\text{def}}{=} \left\{ ss[\mathtt{x} \leftarrow v] \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge v \in \mathbb{A}[\![e]\!]s \\ \wedge\, \sigma \in \Sigma^{\star\infty} \\ \wedge\, s[\mathtt{x} \leftarrow v] \cdot \sigma \in T \end{array} \right\}$$
$$\Lambda_{\mathtt{nit}}[\![\mathtt{assert}\ b]\!]T \stackrel{\text{def}}{=} \left\{ s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \\ \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in T \end{array} \right\}$$
$$\Lambda_{\mathtt{nit}}[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st']\!]T \stackrel{\text{def}}{=} \left\{ s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \\ \wedge\, s \cdot \sigma \in \Lambda_{\mathtt{nit}}[\![st]\!]T \end{array} \right\} \cup \left\{ s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \\ \wedge\, s \cdot \sigma \in \Lambda_{\mathtt{nit}}[\![st']\!]T \end{array} \right\}$$
$$\Lambda_{\mathtt{nit}}[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]T \stackrel{\text{def}}{=} \mathrm{lfp}\, \mathrm{F}$$
$$\Lambda_{\mathtt{nit}}[\![st;\ st']\!]T \stackrel{\text{def}}{=} \Lambda_{\mathtt{nit}}[\![st]\!](\Lambda_{\mathtt{nit}}[\![st']\!]T)$$
$$\Lambda_{\mathtt{nit}}[\![\mathtt{entry}\ st]\!]T \stackrel{\text{def}}{=} \Lambda_{\mathtt{nit}}[\![st]\!] \left\{ s[\mathtt{nit} \leftarrow 0] \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \\ \wedge\, \sigma \in \Sigma^{\star\infty} \\ \wedge\, s \cdot \sigma \in T \end{array} \right\}$$

Lemma 8.3.1 (Soundness of $\Lambda^{\leftarrow}_{\mathtt{nit}}$) *For all programs P, the semantics $\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!] \in \mathbb{D}$ defined in Figure 8.1 is a* sound over-approximation *of the backward co-reachability semantics $\Lambda^{\overline{r}}_{\mathtt{nit}}[\![P]\!]$:*

$$\Lambda^{\overline{r}}_{\mathtt{nit}}[\![P]\!] \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])$$

Proof. We prove by induction on the syntax of the program that the global loop bound semantics $\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]$, cf. Figure 8.1, is a sound over-approximation of the backward co-reachability semantics $\Lambda^{\overline{r}}_{\mathtt{nit}}[\![P]\!]$, cf. Definition 8.2.2. Regarding the base cases, we have:

(skip)

$$
\begin{aligned}
&\Lambda^{\overline{r}}_{\mathtt{nit}}[\![\mathtt{skip}]\!]\gamma(d^\sharp) && \text{(by Def. 8.2.2)}\\
&= \{\{\sigma_0 \mid \sigma \in \Lambda_{\mathtt{nit}}[\![\mathtt{skip}]\!]\gamma(d^\sharp)\}\} && \text{(by Def. 8.1.1)}\\
&= \{\{(ss)_0 \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in \gamma(d^\sharp)\}\} && \text{(by } \sigma = \epsilon\text{)}\\
&= \{\{(ss)_0 \mid s \in \Sigma \wedge s \in \gamma(d^\sharp)\}\} && \text{(by set property)}\\
&= \{\{(ss)_0 \mid s \in \gamma(d^\sharp)\}\} && \text{(by } (ss)_0 = s\text{)}\\
&= \{\{s \mid s \in \gamma(d^\sharp)\}\} && \text{(by set definition)}\\
&= \{\gamma(d^\sharp)\} && \text{(by Eq. (8.1))}\\
&\subseteq \wp(\gamma(d^\sharp)) && \text{(by Eq. (8.2))}\\
&= \gamma^{\leftarrow}(d^\sharp) && \text{(by Fig. 8.1)}\\
&= \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{skip}]\!]d^\sharp)
\end{aligned}
$$

(x := e)

$$
\begin{aligned}
&\Lambda^{\overline{r}}_{\mathtt{nit}}[\![\mathtt{x} := e]\!]\gamma(d^\sharp) && \text{(by Def. 8.2.2)}\\
&= \{\{\sigma_0 \mid \sigma \in \Lambda_{\mathtt{nit}}[\![\mathtt{x} := e]\!]\gamma(d^\sharp)\}\} && \text{(by Def. 8.1.1)}\\
&= \left\{\left\{(ss[\mathtt{x} \leftarrow v] \cdot \sigma)_0 \;\middle|\; \begin{array}{l} s \in \Sigma \wedge v \in \mathbb{A}[\![e]\!]s \wedge \sigma \in \Sigma^{\star\infty} \wedge \\ s[\mathtt{x} \leftarrow v] \cdot \sigma \in \gamma(d^\sharp) \end{array}\right\}\right\} && \text{(by } \sigma = \epsilon\text{)}\\
&= \{\{(ss[\mathtt{x} \leftarrow v] \cdot \sigma)_0 \mid s \in \Sigma \wedge v \in \mathbb{A}[\![e]\!]s \wedge s[\mathtt{x} \leftarrow v] \in \gamma(d^\sharp)\}\} && \text{(by } (ss[\mathtt{x} \leftarrow v] \cdot \sigma)_0 = s\text{)}\\
&= \{\{s \in \Sigma \mid v \in \mathbb{A}[\![e]\!]s \wedge s[\mathtt{x} \leftarrow v] \in \gamma(d^\sharp)\}\} && \text{(by } \mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^\sharp\text{)}\\
&\subseteq \wp(\gamma(\mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^\sharp)) && \text{(by Eq. (8.2))}\\
&= \gamma^{\leftarrow}(\mathtt{Subs}[\![\mathtt{x} \leftarrow e]\!]d^\sharp) && \text{(by Fig. 8.1)}\\
&= \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{x} := e]\!]d^\sharp)
\end{aligned}
$$

(assert b)

$$
\begin{aligned}
&\Lambda^{\overline{r}}_{\mathtt{nit}}[\![\mathtt{assert}\ b]\!]\gamma(d^\sharp) && \text{(by Def. 8.2.2)}\\
&= \{\{\sigma_0 \mid \sigma \in \Lambda_{\mathtt{nit}}[\![\mathtt{assert}\ b]\!]\gamma(d^\sharp)\}\} && \text{(by Def. 8.1.1)}\\
&= \{\{(s \cdot \sigma)_0 \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in \gamma(d^\sharp)\}\} && \text{(by } \sigma = \epsilon\text{)}\\
&= \{\{(s)_0 \mid s \in \Sigma \wedge \mathbb{B}[\![b]\!]s \wedge s \in \gamma(d^\sharp)\}\} && \text{(by set property)}\\
&= \{\{(s)_0 \mid s \in \gamma(d^\sharp) \wedge \mathbb{B}[\![b]\!]s\}\} && \text{(by } (s)_0 = s\text{)}\\
&= \{\{s \in \gamma(d^\sharp) \mid \mathbb{B}[\![b]\!]s\}\} && \text{(by Def. 8.3.1)}\\
&\subseteq \wp(\gamma(\mathtt{Filter}[\![b]\!]d^\sharp)) && \text{(by } \mathtt{Filter}[\![b]\!]d^\sharp\text{)}\\
&= \gamma^{\leftarrow}(\mathtt{Filter}[\![b]\!]d^\sharp) && \text{(by Fig. 8.1)}\\
&= \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{assert}\ b]\!]d^\sharp)
\end{aligned}
$$

Eq. (8.1)

$$\gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]) \stackrel{\text{def}}{=} \wp(\gamma(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]))$$

Eq. (8.2)

$$\Lambda^{\overline{r}}_{\mathtt{nit}}[\![P]\!] \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])$$

Def. 8.2.2 (Backward Co-Reachability Semantics)

$$\Lambda^{\overline{r}} \stackrel{\text{def}}{=} \{\{\sigma_0 \mid \sigma \in \Lambda_{\mathtt{nit}}\}\}$$

Def. 8.1.1 (Maximal Trace Semantics with the Global Loop Counter)

$$\Lambda_{\mathtt{nit}}[\![\mathtt{skip}]\!]T \stackrel{\text{def}}{=} \{ss \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in T\}$$

$$\Lambda_{\mathtt{nit}}[\![\mathtt{x} := e]\!]T \stackrel{\text{def}}{=} \left\{ss[\mathtt{x} \leftarrow v] \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge v \in \mathbb{A}[\![e]\!]s \\ \wedge\ \sigma \in \Sigma^{\star\infty} \\ \wedge\ s[\mathtt{x} \leftarrow v] \cdot \sigma \in T \end{array}\right\}$$

$$\Lambda_{\mathtt{nit}}[\![\mathtt{assert}\ b]\!]T \stackrel{\text{def}}{=} \left\{s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \\ \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in T \end{array}\right\}$$

$$\Lambda_{\mathtt{nit}}[\![\mathtt{if}\ b\ \mathtt{then}\ st\ \mathtt{else}\ st'\!]\!]T \stackrel{\text{def}}{=} \left\{s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \\ \wedge\ s \cdot \sigma \in \Lambda_{\mathtt{nit}}[\![st]\!]T \end{array}\right\} \cup \left\{s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \\ \wedge\ s \cdot \sigma \in \Lambda_{\mathtt{nit}}[\![st']\!]T \end{array}\right\}$$

$$\Lambda_{\mathtt{nit}}[\![\mathtt{while}\ b\ \mathtt{do}\ st\ \mathtt{done}]\!]T \stackrel{\text{def}}{=} \mathrm{lfp}\,\mathrm{F}$$

$$\Lambda_{\mathtt{nit}}[\![st; st']\!]T \stackrel{\text{def}}{=} \Lambda_{\mathtt{nit}}[\![st]\!](\Lambda_{\mathtt{nit}}[\![st']\!]T)$$

$$\Lambda_{\mathtt{nit}}[\![\mathtt{entry}\ st]\!]T \stackrel{\text{def}}{=} \Lambda_{\mathtt{nit}}[\![st]\!]\left\{s[\mathtt{nit} \leftarrow 0] \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \\ \wedge\ \sigma \in \Sigma^{\star\infty} \\ \wedge\ s \cdot \sigma \in T \end{array}\right\}$$

$(\texttt{entry}\ st)$

$$
\begin{aligned}
&\Lambda^{\bar{r}}_{\texttt{nit}}[\![\texttt{entry}\ st]\!]\gamma(d^\natural) && \text{(by Def. 8.2.2)}\\
&= \{\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}[\![\texttt{entry}\ st]\!]\gamma(d^\natural)\}\} && \text{(by Def. 8.1.1)}\\
&= \{\{(s[\texttt{nit} \leftarrow 0] \cdot \sigma)_0 \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in \gamma(d^\natural)\}\} && \text{(by } \sigma = \epsilon\text{)}\\
&= \{\{s[\texttt{nit} \leftarrow 0] \mid s \in \gamma(d^\natural)\}\} && \text{(by } \texttt{Subs}[\![\texttt{nit} \leftarrow 0]\!]d^\natural\text{)}\\
&\subseteq \wp(\gamma(\texttt{Subs}[\![\texttt{nit} \leftarrow 0]\!]d^\natural)) && \text{(by Eq. (8.2))}\\
&= \gamma^{\leftarrow}(\texttt{Subs}[\![\texttt{nit} \leftarrow 0]\!]d^\natural) && \text{(by Fig. 8.1)}\\
&= \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\texttt{nit}}[\![\texttt{entry}\ st]\!]d^\natural)
\end{aligned}
$$

Eq. (8.1)

$$\gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\texttt{nit}}[\![P]\!]) \stackrel{\text{def}}{=} \wp(\gamma(\Lambda^{\leftarrow}_{\texttt{nit}}[\![P]\!]))$$

Eq. (8.2)

$$\Lambda^{\bar{r}}_{\texttt{nit}}[\![P]\!] \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\texttt{nit}}[\![P]\!])$$

Def. 8.2.2 (Backward Co-Reachability Semantics)

$$\Lambda^{\bar{r}} \stackrel{\text{def}}{=} \{\,\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}\}\,\}$$

Def. 8.1.1 (Maximal Trace Semantics with the Global Loop Counter)

$$\Lambda_{\texttt{nit}}[\![\texttt{skip}]\!]T \stackrel{\text{def}}{=} \{ss \cdot \sigma \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge s \cdot \sigma \in T\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{x} := e]\!]T \stackrel{\text{def}}{=} \left\{ss[\texttt{x} \leftarrow v] \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge v \in \mathbb{A}[\![e]\!]s \\ \wedge\, \sigma \in \Sigma^{\star\infty} \\ \wedge\, s[\texttt{x} \leftarrow v] \cdot \sigma \in T \end{array}\right\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{assert}\ b]\!]T \stackrel{\text{def}}{=} \left\{s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \\ \mathbb{B}[\![b]\!]s \wedge s \cdot \sigma \in T \end{array}\right\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{if}\ b\ \texttt{then}\ st\ \texttt{else}\ st']\!]T \stackrel{\text{def}}{=} \left\{s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \\ \wedge\, s \cdot \sigma \in \Lambda_{\texttt{nit}}[\![st]\!]T \end{array}\right\} \cup \left\{s \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \\ \wedge\, s \cdot \sigma \in \Lambda_{\texttt{nit}}[\![st']\!]T \end{array}\right\}$$

$$\Lambda_{\texttt{nit}}[\![\texttt{while}\ b\ \texttt{do}\ st\ \texttt{done}]\!]T \stackrel{\text{def}}{=} \mathrm{lfp}\,\mathrm{F}$$

$$\Lambda_{\texttt{nit}}[\![st; st']\!]T \stackrel{\text{def}}{=} \Lambda_{\texttt{nit}}[\![st]\!](\Lambda_{\texttt{nit}}[\![st']\!]T)$$

$$\Lambda_{\texttt{nit}}[\![\texttt{entry}\ st]\!]T \stackrel{\text{def}}{=} \Lambda_{\texttt{nit}}[\![st]\!]\left\{s[\texttt{nit} \leftarrow 0] \cdot \sigma \;\middle|\; \begin{array}{l} s \in \Sigma \\ \wedge\, \sigma \in \Sigma^{\star\infty} \\ \wedge\, s \cdot \sigma \in T \end{array}\right\}$$

Regarding the inductive cases, we assume the following inductive hypothesis: $\Lambda^{\bar{r}}_{\texttt{nit}}[\![st]\!]\gamma(d^\natural) \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st]\!]d^\natural)$. Note that, the inductive hypothesis implies that $\Lambda_{\texttt{nit}}[\![st]\!]\gamma(d^\natural) \in \wp(\gamma(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st]\!]d^\natural))$ by Definition 8.2.2 and Equation 8.1. Therefore, we have:

$(st;\ st)$

$$
\begin{aligned}
&\Lambda^{\bar{r}}_{\texttt{nit}}[\![st;\ st]\!]\gamma(d^\natural) && \text{(by Def. 8.2.2)}\\
&= \{\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}[\![st;\ st]\!]\gamma(d^\natural)\}\} && \text{(by Def. 8.1.1)}\\
&= \{\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}[\![st]\!](\Lambda_{\texttt{nit}}[\![st']\!]\gamma(d^\natural))\}\} && \text{(by inductive hypothesis)}\\
&\subseteq \wp(\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}[\![st]\!]\gamma(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st']\!]d^\natural)\}) && \text{(by inductive hypothesis)}\\
&\subseteq \wp(\{\sigma_0 \mid \sigma \in \gamma(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st]\!]\Lambda^{\leftarrow}_{\texttt{nit}}[\![st']\!]d^\natural)\}) && \text{(by } \sigma = s\text{)}\\
&\subseteq \wp(\{s \mid s \in \gamma(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st]\!]\Lambda^{\leftarrow}_{\texttt{nit}}[\![st']\!]d^\natural)\}) && \text{(by set definition)}\\
&= \wp(\gamma(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st]\!](\Lambda^{\leftarrow}_{\texttt{nit}}[\![st']\!]d^\natural))) && \text{(by Eq. (8.1))}\\
&= \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st]\!](\Lambda^{\leftarrow}_{\texttt{nit}}[\![st']\!]d^\natural)) && \text{(by Fig. 8.1)}\\
&= \gamma^{\leftarrow}(\Lambda^{\leftarrow}_{\texttt{nit}}[\![st;\ st]\!]d^\natural)
\end{aligned}
$$

(`if` b `then` st `else` st)

$$\Lambda^{\overleftarrow{r}}_{\texttt{nit}}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]\gamma(d^\natural) \qquad \text{(by Def. 8.2.2)}$$

$$= \{\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st']\!]\gamma(d^\natural)\}\} \qquad \text{(by Def. 8.1.1)}$$

$$= \{\{(s\cdot\sigma)_0 \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \wedge s\cdot\sigma \in \Lambda_{\texttt{nit}}[\![st]\!]\gamma(d^\natural)\} \cup \{(s\cdot\sigma)_0 \mid s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \wedge s\cdot\sigma \in \Lambda_{\texttt{nit}}[\![st']\!]\gamma(d^\natural)\}\}$$

(by inductive hypothesis)

$$\subseteq \wp\left(\begin{array}{l}\left\{(s\cdot\sigma)_0 \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![b]\!]s \wedge \\ s\cdot\sigma \in \gamma(\Lambda_{\texttt{nit}}[\![st]\!]d^\natural)\end{array}\right\} \\ \cup \left\{(s\cdot\sigma)_0 \;\middle|\; \begin{array}{l} s \in \Sigma \wedge \sigma \in \Sigma^{\star\infty} \wedge \mathbb{B}[\![\neg b]\!]s \wedge \\ s\cdot\sigma \in \gamma(\Lambda_{\texttt{nit}}[\![st']\!]d^\natural)\end{array}\right\}\end{array}\right) \qquad (\text{by } \sigma = \epsilon)$$

$$= \wp\left(\begin{array}{l}\{s \in \Sigma \mid \mathbb{B}[\![b]\!]s \wedge s \in \gamma(\Lambda_{\texttt{nit}}[\![st]\!]d^\natural)\} \cup \\ \{s \in \Sigma \mid \mathbb{B}[\![\neg b]\!]s \wedge s \in \gamma(\Lambda_{\texttt{nit}}[\![st']\!]d^\natural)\}\end{array}\right) \qquad \text{(by } \texttt{Filter}[\![b]\!]d^\natural\text{)}$$

$$\subseteq \wp\left(\begin{array}{l}\gamma(\texttt{Filter}[\![b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural)) \cup \\ \{s \in \Sigma \mid \mathbb{B}[\![\neg b]\!]s \wedge s \in \gamma^{\overleftarrow{}}(\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st']\!]d^\natural)\}\end{array}\right) \qquad \text{(by } \texttt{Filter}[\![\neg b]\!]d^\natural\text{)}$$

$$\subseteq \wp\left(\begin{array}{l}\gamma(\texttt{Filter}[\![b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural)) \cup \\ \gamma(\texttt{Filter}[\![\neg b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural))\end{array}\right) \qquad \text{(by } \sqcup\text{)}$$

$$\subseteq \wp\left(\gamma\left(\texttt{Filter}[\![b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural) \sqcup \texttt{Filter}[\![\neg b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural)\right)\right) \qquad \text{(by Eq. (8.1))}$$

$$= \gamma^{\overleftarrow{}}(\texttt{Filter}[\![b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural) \sqcup \texttt{Filter}[\![\neg b]\!](\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![st]\!]d^\natural)) \qquad \text{(by Fig. 8.1)}$$

$$= \gamma^{\overleftarrow{}}(\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![\texttt{if } b \texttt{ then } st \texttt{ else } st]\!]d^\natural)$$

(`while` b `do` st `done`)

$$\Lambda^{\overleftarrow{r}}_{\texttt{nit}}[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]\gamma(d^\natural) \qquad \text{(by Def. 8.2.2)}$$

$$= \{\{\sigma_0 \mid \sigma \in \Lambda_{\texttt{nit}}[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]\gamma(d^\natural)\}\} \qquad \text{(by Def. 8.1.1)}$$

$$= \{\{\sigma_0 \mid \sigma \in \mathrm{lfp}\, \mathrm{F}\}\} \qquad (\text{by } \lim_n F^\natural_n)$$

$$\subseteq \wp(\{\sigma_0 \mid \sigma \in \gamma(\lim_n F^\natural_n)\}) \qquad (\text{by } \sigma = s)$$

$$= \wp(\{s \mid s \in \gamma(\lim_n F^\natural_n)\}) \qquad \text{(by set definition)}$$

$$= \wp(\gamma(\lim_n F^\natural_n)) \qquad \text{(by Eq. (8.1))}$$

$$= \gamma^{\overleftarrow{}}(\lim_n F^\natural_n) \qquad \text{(by Fig. 8.1)}$$

$$= \gamma^{\overleftarrow{}}(\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![\texttt{while } b \texttt{ do } st \texttt{ done}]\!]d^\natural)$$

As a consequence, the backward semantics $\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![P]\!]$ is a sound over-approximation of the backward co-reachability semantics $\Lambda^{\overleftarrow{r}}_{\texttt{nit}}[\![P]\!]$, *i.e.*, $\Lambda^{\overleftarrow{r}}_{\texttt{nit}}[\![P]\!] \subseteq \gamma^{\overleftarrow{}}(\Lambda^{\overleftarrow{}}_{\texttt{nit}}[\![P]\!])$. □

Next, we show how to quantify the impact of the input variables on the global loop counter `nit`. We employ the Range impact quantifier, instrumented with `nit` as output variable, and we design a linear programming encoding to compute the abstract implementation of the impact $\textsc{Range}_{\texttt{W}}$.

Def. 8.3.1 (Concretization of Linear Constraints)

$$\gamma(d^\natural) \stackrel{\text{def}}{=} \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall (c_1 \cdot \mathtt{x}_1 + \ldots + c_{n+1} \geq 0) \\ \quad \in d^\natural. \\ c_1 \cdot s(\mathtt{x}_1) + \ldots + c_{n+1} \geq 0 \end{array} \right\}$$

Prog. 8.1 (Program `Add`, computing the sum of two numbers `x` and `y` into `z`.):

```
1  def Add(p, z, m, x, n, y):
2    r = min(p, m)
3    s = min(p, n)
4    if (r < s):
5      t = p - s
6      q = s - r
7      # ...
8      for (; r > 0; r--):
9        # ...
10     do:
11       # ...
12       q--
13       # ...
14     while (q > 0)
15 else:
16   t = p - r
17   q = r - s
18   # ...
19   for (; s > 0; s--):
20     # ...
21   for (; q > 0; q--):
22     # ...
23 if (t > 0):
24   # ...
25   while (t > 0):
26     # ...
27     t--
28     # ...
```

Example 8.3.1 We consider the Program 8.1, where states Σ are tuples $\langle p, m, n, nit \rangle$, respectively for the variables `p`, `m`, `n`, and the global loop counter `nit`. Let us assume that the computation of the backward semantics on the program `Add` results in:

$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{Add}]\!] \;=\; (\mathtt{p} = \mathtt{nit})$$

Then, from Definition 8.3.1, the concretization of the abstract element $\mathtt{p} = \mathtt{nit}$ is:

$$\gamma(\mathtt{p} = \mathtt{nit}) \;=\; \{\langle p, m, n, p \rangle \mid p, m, n \in \mathbb{V}\}$$

where $\langle p, m, n, p \rangle$ is the concrete state in which the input variable `p` (first p in the tuple) is equal to the loop counter `nit` (last p in the tuple). The goal of the concretization $\gamma^{\leftarrow}$ is to over-approximate the backward co-reachability semantics $\Lambda^{\overline{r}}_{\mathtt{nit}}[\![\mathtt{Add}]\!]$. By concretizing any subset of $\gamma(\mathtt{p} = \mathtt{nit})$ we obtain all the possible input states such that the input variable `p` is equal to the value of the global loop counter `nit`, including the backward co-reachability semantics $\Lambda^{\overline{r}}_{\mathtt{nit}}[\![\mathtt{Add}]\!]$.

8.4 An Abstract Range Implementation via a Linear Programming Encoding

We present a linear programming encoding $\mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}} \in \mathbb{D} \to \mathbb{V}^{+\infty}_{\geq 0}$ to compute the abstract implementation of the impact $\textsc{Range}_{\mathtt{W}}$ from the result of the global loop bound analysis $\Lambda^{\leftarrow}_{\mathtt{nit}}$.

Definition 8.4.1 ($\mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}}$) *Let $\Lambda^{\leftarrow}_{\mathtt{nit}} \in \mathbb{D}$ be the global loop bound analysis and $\mathtt{W} \subseteq \Delta$ be the input variables of interest. For all programs P, $\mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])$ is defined as:*

$$\begin{aligned} &\textit{maximize}\;\; k && (8.5)\\ &\textit{subject to}\;\; \mathtt{Proj}_{\mathtt{W}}(\mathtt{Subs}[\![\mathtt{nit} \leftarrow \overline{\mathtt{nit}}]\!](\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])) && (8.6)\\ &\qquad \wedge\; \mathtt{Proj}_{\mathtt{W}}(\mathtt{Subs}[\![\mathtt{nit} \leftarrow \underline{\mathtt{nit}}]\!](\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])) && (8.7)\\ &\qquad \wedge\; 0 \leq k \leq \overline{\mathtt{nit}} - \underline{\mathtt{nit}} && (8.8) \end{aligned}$$

where $\overline{\mathtt{nit}}$, $\underline{\mathtt{nit}}$ are fresh variables.

Since k should be an integer variable, we specifically solve a mixed-integer linear programming problem. Equation 8.6 substitutes the variable `nit` with $\overline{\mathtt{nit}}$ to account for the maximal value of the global loop counter `nit`. Then, it projects away the input variables `W` to encompass any possible variation of that variables. Equation 8.7 substitutes the variable `nit` with $\underline{\mathtt{nit}}$ for the minimal value of `nit`, and again projects away the input variables `W`. Hence, the set of constraints from Equation 8.6 and Equation 8.7 only differ in the variable of the loop counter, respectively $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$. Finally, the objective function, cf. Equation 8.5, maximizes the value of the bound k, which ranges between 0 and $\overline{\mathtt{nit}} - \underline{\mathtt{nit}}$, cf. Equation 8.8. The maximum value of the bound k is the length of the range of the feasible values for the loop counter `nit`.

Example 8.4.1 We consider again the example of the Program 8.1. Let us assume that $\mathtt{p} \in [0, u]$ and the computation of the backward semantics on the program `Add` results in:

$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{Add}]\!] \;=\; (\mathtt{p} = \mathtt{nit} \;\wedge\; 0 \leq \mathtt{p} \leq u)$$

To compute the abstract range $\mathtt{Range}^{\mathtt{nit}}_{\{\mathtt{p}\}}$ for the input variable `p`, we solve the linear programming problem Equation 8.5–Equation 8.8. Where Equation 8.6 and Equation 8.7 are respectively:

$$\begin{aligned}
&\mathtt{Proj}_{\{\mathtt{p}\}}(\mathtt{Subs}[\![\mathtt{nit} \leftarrow \overline{\mathtt{nit}}]\!](\mathtt{p} = \mathtt{nit} \;\wedge\; 0 \leq \mathtt{p} \leq u)) \\
&\qquad = \mathtt{Proj}_{\{\mathtt{p}\}}(\mathtt{p} = \overline{\mathtt{nit}} \;\wedge\; 0 \leq \mathtt{p} \leq u) \;=\; 0 \leq \overline{\mathtt{nit}} \leq u \\
&\mathtt{Proj}_{\{\mathtt{p}\}}(\mathtt{Subs}[\![\mathtt{nit} \leftarrow \underline{\mathtt{nit}}]\!](\mathtt{p} = \mathtt{nit} \;\wedge\; 0 \leq \mathtt{p} \leq u)) \\
&\qquad = \mathtt{Proj}_{\{\mathtt{p}\}}(\mathtt{p} = \underline{\mathtt{nit}} \;\wedge\; 0 \leq \mathtt{p} \leq u) \;=\; 0 \leq \underline{\mathtt{nit}} \leq u
\end{aligned}$$

Therefore, the linear programming encoding for $\mathtt{Range}^{\mathtt{nit}}_{\{\mathtt{p}\}}(\mathtt{p} = \mathtt{nit} \wedge 0 \leq \mathtt{p} \leq u)$ is defined as:

$$\begin{aligned}
&\text{maximize} \;\; k \\
&\text{subject to} \;\; 0 \leq \overline{\mathtt{nit}} \leq u \\
&\qquad\qquad \wedge \;\; 0 \leq \underline{\mathtt{nit}} \leq u \\
&\qquad\qquad \wedge \;\; 0 \leq k \leq \overline{\mathtt{nit}} - \underline{\mathtt{nit}}
\end{aligned}$$

which maximizes at u. On the other hand, projecting away the other input variables leaves the invariant $\mathtt{p} = \mathtt{nit} \;\wedge\; 0 \leq \mathtt{p} \leq u$ unchanged. Thus, k maximizes at 0 as the variable `p` is equal to both $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$. Interestingly, we did not lose any precision regarding the k-bounded impact property as $\textsc{Range}_{\mathtt{W}}(\Lambda_{\mathtt{nit}}[\![\mathtt{Add}]\!]) = \mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![\mathtt{Add}]\!])$ for all input variables `W` in the program `Add`.

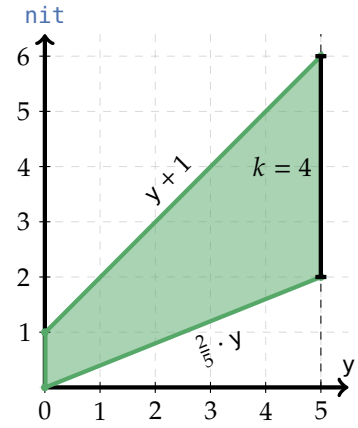


Figure 8.2: $\frac{2}{5} \cdot \mathtt{y} \;\leq\; \mathtt{nit} \;\leq\; \mathtt{y} + 1$.

Example 8.4.2 In the previous example, the projection of `p` alone produced the invariant $0 \leq \mathtt{nit} \leq u$, which gives already the bound on the impact of the input variable `p` on the global loop counter `nit`. We consider a more complex example to demonstrate the necessity of the linear programming encoding, in a scenario where the invariant computed by the global loop bound analysis does not already reveal the impact of input variables. Let us assume that the computation of the abstract dependency semantics for a program P results in:

$$\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!] \;=\; \left(\frac{2}{5} \cdot \mathtt{y} + \frac{1}{5} \cdot \mathtt{x} \;\leq\; \mathtt{nit} \;\leq\; \mathtt{y} + \frac{1}{5} \cdot \mathtt{x} \;\wedge\; \mathtt{x}, \mathtt{y} \in [0, 5]\right)$$

To compute the abstract range $\mathtt{Range}^{\natural}_{\mathtt{x}}$ for the input variable `x`, we first project away the variable `x` from the abstract state, obtaining:

$$\frac{2}{5} \cdot \mathtt{y} \;\leq\; \mathtt{nit} \;\leq\; \mathtt{y} + 1 \;\wedge\; \mathtt{y} \in [0, 5]$$

Then, we solve the linear programming problem, where we substitute the variable `nit` with $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$ to respectively maximize and

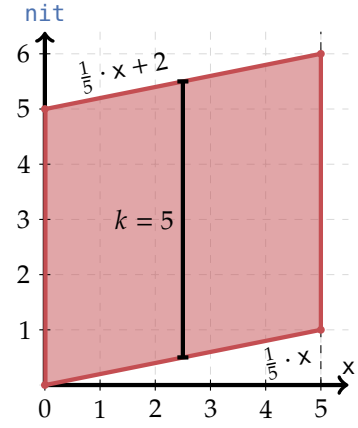


Figure 8.3: $\frac{1}{5} \cdot \mathtt{x} \;\leq\; \mathtt{nit} \;\leq\; \frac{1}{5} \cdot \mathtt{x} + 2$.

minimize the global loop counter nit:

$$\begin{aligned}&\text{maximize } \ k\\ &\text{subject to } \ \frac{2}{5}\cdot \mathsf{y} \ \leq\ \overline{\mathtt{nit}} \ \leq\ \mathsf{y}+1 \ \wedge\ \mathsf{y}\in[0,5]\\ &\qquad\wedge\ \frac{2}{5}\cdot \mathsf{y} \ \leq\ \underline{\mathtt{nit}} \ \leq\ \mathsf{y}+1 \ \wedge\ \mathsf{y}\in[0,5]\\ &\qquad\wedge\ 0 \ \leq\ k \ \leq\ \overline{\mathtt{nit}}-\underline{\mathtt{nit}}\end{aligned}$$

The linear programming encoding maximizes at $k = 4$ where $\mathsf{y} = 5$, Figure 8.2 shows graphically the feasible region and maximization point. On the other hand, to compute the abstract range $\mathsf{Range}^{\natural}_{\mathsf{y}}$ for the input variable y, we solve linear programming encoding where we project away the variable y from the abstract state, obtaining:

$$\begin{aligned}&\text{maximize } \ k\\ &\text{subject to } \ \frac{1}{5}\cdot \mathsf{x} \ \leq\ \overline{\mathtt{nit}} \ \leq\ \frac{1}{5}\cdot \mathsf{x}+2 \ \wedge\ \mathsf{x}\in[0,5]\\ &\qquad\wedge\ \frac{1}{5}\cdot \mathsf{x} \ \leq\ \underline{\mathtt{nit}} \ \leq\ \frac{1}{5}\cdot \mathsf{x}+2 \ \wedge\ \mathsf{x}\in[0,5]\\ &\qquad\wedge\ 0 \ \leq\ k \ \leq\ \overline{\mathtt{nit}}-\underline{\mathtt{nit}}\end{aligned}$$

This second linear programming problem maximizes at $k = 5$ for any value of x, Figure 8.3 shows graphically the feasible region and maximization point for this second programming problem. As a result, we obtain $\mathsf{Range}^{\natural}_{\mathsf{x}}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]) = 4$ and $\mathsf{Range}^{\natural}_{\mathsf{y}}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]) = 5$, showing that the variable y has a higher impact on the global loop counter nit than the variable x.

The next result shows that $\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}$ is a sound implementation of the impact $\textsc{Range}_{\mathsf{W}}$, by means of Definition 4.2.6 when applied to the $\leq$ operator. That is, the abstract quantity is always higher than the concrete counterpart.

Def. 4.2.6 (Sound Implementation)

$$\textsc{Impact}_{\mathsf{W}}(\Lambda[\![P]\!]) \ \circledS\ \mathtt{Impact}^{\natural}_{\mathsf{W}}(\Lambda^{\times}[\![P]\!]B, B)$$

Lemma 8.4.1 ($\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}$ is a Sound Implementation of $\textsc{Range}_{\mathsf{W}}$) *For any program P, the following holds:*

$$\textsc{Range}_{\mathsf{W}}(\Lambda_{\mathtt{nit}}[\![P]\!]) \ \leq\ \mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!])$$

Def. 8.1.3 ($\textsc{Range}$ with nit)

$$\textsc{Range}_{\mathsf{W}}(T) \stackrel{\text{def}}{=} \sup_{s_0\in\Sigma|_{\Delta}} \textsc{Distance}\left(\left\{\sigma_0(\mathtt{nit}) \;\middle|\; \begin{matrix}\sigma\in T\wedge\\ \sigma_0 =_{\Delta\setminus\mathsf{W}} s_0\end{matrix}\right\}\right)$$

Def. 8.4.1 ($\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}$)

$$\begin{aligned}&\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(d^{\natural}) \stackrel{\text{def}}{=}\\ &\quad \textit{maximize } \ k\\ &\quad \textit{subject to}\\ &\qquad \mathtt{Proj}_{\mathsf{W}}(\mathtt{Subs}[\![\mathtt{nit}\leftarrow\overline{\mathtt{nit}}]\!](d^{\natural}))\wedge\\ &\qquad \mathtt{Proj}_{\mathsf{W}}(\mathtt{Subs}[\![\mathtt{nit}\leftarrow\underline{\mathtt{nit}}]\!](d^{\natural}))\wedge\\ &\qquad 0\leq k\leq \overline{\mathtt{nit}}-\underline{\mathtt{nit}}\end{aligned}$$

Proof. We show that $\textsc{Range}_{\mathsf{W}}(\Lambda_{\mathtt{nit}}) \leq \mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}})$ by contradiction. Let us assume that $\textsc{Range}_{\mathsf{W}}(\Lambda_{\mathtt{nit}}) > \mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}})$ and $\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}}) = k$. Then, by definition of $\textsc{Range}_{\mathsf{W}}$, cf. Definition 8.1.3, there exists two concrete states, differing only in the value of W, such that the difference in the value of the global loop counter nit is greater than k. As a consequence, in the abstract implementation $\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}$, cf. Definition 8.4.1 the distance between $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$ should maximize at a value greater than k. This contradicts the assumption that $k = \mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}})$. From Lemma 8.3.1, and the fact that $\textsc{Range}_{\mathsf{W}}$ is monotonic, cf. Lemma 4.1.6, we conclude that: $\textsc{Range}_{\mathsf{W}}(\Lambda_{\mathtt{nit}}) \leq \mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}})$. □

Next, we show Theorem 4.2.4 (Soundness) instantiated with the abstract implementation $\mathsf{Range}^{\mathtt{nit}}_{\mathsf{W}}$ and the comparison operator $\leq$.

Theorem 8.4.2 (Soundness of $\mathcal{B}^{\leq k}_{\textsc{Range}_{\mathtt{W}}}$) *Let $\mathcal{B}^{\leq k}_{\textsc{Range}_{\mathtt{W}}}$ be the property of interest we want to verify for the program P and the input variable $\mathtt{W} \in \wp(\Delta)$. Whenever, $\Lambda^{\leftarrow}_{\mathtt{nit}}$ is sound, cf. Equation 8.2, the following implication holds:*

$$\mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}}(\Lambda^{\leftarrow}_{\mathtt{nit}}[\![P]\!]) = k' \wedge k' \leq k \implies P \models \mathcal{B}^{\leq k}_{\textsc{Range}_{\mathtt{W}}}$$

Proof. Lemma 8.4.1 shows that $\mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}}$ is a sound implementation of $\textsc{Range}_{\mathtt{W}}$, the proof follows directly by application of the Theorem 4.2.4 instantiated with the abstract implementation $\mathtt{Range}^{\mathtt{nit}}_{\mathtt{W}}$ and comparison operator $\leq$. □

Thm. 4.2.4 (Soundness)

$$\mathtt{Impact}^{\natural}_{\mathtt{W}}(\Lambda^{\times}[\![P]\!]B, B) \otimes k \implies P \models \mathcal{B}^{\otimes k}_{\textsc{Impact}_{\mathtt{W}}}$$

8.5 Related Work

Loop bound analyses are extensively studied in the literature, given their importance in various program analyses, including termination, Worst-Case Execution Time (WCET), and side-channel analysis. In this section, we discuss the most relevant works in these areas and compare them with our approach.

Worst-Case Execution Time. Worst-Case Execution Time (WCET) analysis aims to derive sound upper bounds on the execution time of programs, see Wilhelm et al. [161] for a survey. More specifically, determining an upper bound on the number of loop iterations is a particular instance of loop bound analysis [162, 163], which seeks to infer invariants on the iterations of individual loops. In contrast, global loop bound analyses consider the aggregate behavior of all loops within a program [164, 165]. Our work is orthogonal to the existing literature on WCET and loop bound analysis because it does not generate invariants on the global number of iterations. Instead, we focus on quantifying the impact of each input variable on the total number of loop iterations. However, we leverage an underlying global loop bound analysis to understand the input-output relationship involving the global loop counter.

Side-Channels Analysis. Side-channel attacks exploit vulnerabilities in the implementation of cryptographic systems rather than targeting computational complexity directly [50]. These attacks, including those based on power consumption [166] and speculative executions [167, 168], can leak sensitive information without physical tampering [49]. Various methods have been developed to quantify the information leaked through side channels.

The quantitative analysis of side-channel attacks, first introduced by Köpf and Basin [169], uses model counting and a greedy algorithm to manage the exponential growth of potential paths requiring examination. Phan et al. [85, 170] and Saha et al. [87] apply symbolic execution to enumerate the equivalence classes of a program's output values and use entropy measures to quantify the information leakage. Building on these works, Malacaria et al. [171] handles symbolic inputs interpreted probabilistically, producing provably under- and over-approximating bounds on the leakage (with respect to Shannon and min-entropy). Other

[161]: Wilhelm et al. (2008), 'The worst-case execution-time problem - overview of methods and survey of tools'

[162]: Cadek et al. (2018), 'Using Loop Bound Analysis For Invariant Generation'

[163]: Ermedahl et al. (2007), 'Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis'

[164]: Carbonneaux et al. (2015), 'Compositional certified resource bounds'

[165]: Sinn et al. (2017), 'Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints'

[50]: Kocher (1996), 'Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems'

[166]: Kocher et al. (1999), 'Differential Power Analysis'

[167]: Kocher et al. (2019), 'Spectre Attacks: Exploiting Speculative Execution'

[168]: Lipp et al. (2020), 'Meltdown: reading kernel memory from user space'

[49]: Wong (2005), 'Timing attacks on RSA: revealing your secrets through the fourth dimension'

[169]: Köpf et al. (2007), 'An information-theoretic model for adaptive side-channel attacks'

[85]: Phan et al. (2012), 'Symbolic quantitative information flow'

[170]: Phan et al. (2017), 'Synthesis of Adaptive Side-Channel Attacks'

[87]: Saha et al. (2023), 'Obtaining Information Leakage Bounds via Approximate Model Counting'

[171]: Malacaria et al. (2018), 'Symbolic Side-Channel Analysis for Probabilistic Programs'

static analyses, such as those by Assaf et al. [88] and Clark, Hunt, and Malacaria [89], rely on abstract interpretation. Based on string analysis, Bang et al. [172] presented an efficient technique for segmented oracles, where an attacker can explore each segment of secret data. Pasareanu, Phan, and Malacaria [173] quantifies the information leaked to an attacker who performs multiple side-channel measurements.

Our approach diverges by quantifying the impact of each input variable separately, rather than providing a comprehensive quantification for all input variables. This fine-grained analysis is essential for identifying specific variables that significantly influence program behavior. This fine-grained analysis is crucial for identifying specific variables that significantly influence program behavior. While quantitative analyses could theoretically infer similar information, their entropy measures are not designed for this purpose (cf. Section 4.3), and they typically require k-times self-composition [98, 99], which is computationally intensive [100]. Instead, our method requires only a single abstract analysis and uses variable projections to isolate the contributions of each input variable [47, 73].

Moving forward, the emergence of attacks exploiting microarchitectural features, such as out-of-order and speculative execution, diminishes the efficacy of traditional constant-time verification. Indeed, constant-time code may still vulnerable to timing attacks on processors with microarchitectural vulnerabilities [174]. Recently, OpenSSL updated its security model to explicitly exclude physical system side-channels.[1] Related work [175, 176] has proposed incorporating microarchitectural features into the analysis of timing side-channel attacks. Our approach could be extended to consider the impact of input variables in the context of out-of-order and speculative execution by an ad-hoc semantics for microarchitectural features [177].

Termination Analysis. The program termination problem can be defined as whether a program will always finish running within a finite time frame, or could execute indefinitely. It is rooted in the early 20th century mathematical logic, particularly in the Hilbert's Entscheidungsproblem.[2] Turing's proof of the undecidability of the termination problem via disjunctive arguments is a cornerstone in the field [16, 178]. Recent advancements in termination analysis have a significant impact in the literature, particularly those grounded in transition invariants as introduced in Podelski and Rybalchenko [179]. Notably, the Terminator analyzer [180] constructs iteratively these invariants to prove termination.

From the unifying work of existing approaches proposed by Cousot and Cousot [181], Caterina Urban and Antoine Miné advanced the field in a series of works [182–186] by proving termination through the synthesis of ranking functions across all program paths within the framework of abstract interpretation. Compared to other works [187, 188], their approach does not rely on counterexample-guided analysis or perform explicit checks on the well-foundedness of the termination argument. Their method, implemented in the tool FuncTion [189], handles arbitrary control flow structures, unlike earlier methods [179, 190]. Similarly, Tsitovich et al. [191] uses loop summarization to address termination, and Courant and Urban [192] refines FuncTion's scalability and precision

[88]: Assaf et al. (2017), 'Hypercollecting semantics and its application to static analysis of information flow'

[89]: Clark et al. (2007), 'A static analysis for quantifying information flow in a simple imperative language'

[172]: Bang et al. (2016), 'String analysis for side channels with segmented oracles'

[173]: Pasareanu et al. (2016), 'Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT'

[98]: Barthe et al. (2011), 'Secure information flow by self-composition'

[99]: Terauchi et al. (2005), 'Secure Information Flow as a Safety Problem'

[100]: Antonopoulos et al. (2017), 'Decomposition instead of self-composition for proving the absence of timing channels'

[47]: Mazzucato et al. (2021), 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'

[73]: Urban et al. (2020), 'Perfectly parallel fairness certification of neural networks'

[174]: Cauligi et al. (2020), 'Constant-time foundations for the new spectre era'

1: (Last Accessed: 16th August 2024) www.openssl.org/policies/secpolicy.html

[175]: Guarnieri et al. (2020), 'Spectector: Principled Detection of Speculative Information Flows'

[176]: Bard et al. (2024), 'Automatic and Incremental Repair for Speculative Information Leaks'

[177]: Dongol et al. (2024), 'Relative Security: Formally Modeling and (Dis)Proving Resilience Against Semantic Optimization Vulnerabilities'

2: In English: "decision problem."

[16]: Turing (1937), 'On computable numbers, with an application to the Entscheidungsproblem'

[178]: Turing (1949), 'Checking a Large Routine'

[180]: Cook et al. (2006), 'Terminator: Beyond Safety'

[181]: Cousot et al. (2012), 'An abstract interpretation framework for termination'

[182]: Urban (2013), 'The Abstract Domain of Segmented Ranking Functions'

[183]: Urban et al. (2014), 'An Abstract Domain to Infer Ordinal-Valued Ranking Functions'

[184]: Urban et al. (2014), 'A Decision Tree Abstract Domain for Proving Conditional Termination'

[185]: Urban et al. (2015), 'Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation'

[186]: Urban et al. (2017), 'Inference of ranking functions for proving temporal properties by abstract interpretation'

through fine-tuning and new heuristics for widening.

Many methods, such as those in [193–195], indirectly leverage transition invariants. Among them, Alias et al. [193] explores the synthesis of ranking functions via linear programming, which is complete only for programs with rational values. Chatterjee et al. [196] proposed a straightforward but effective approach to over-approximate non-terminating program states by synthesizing invariants through syntactic program reversal, handling programs with non-determinism and integer values. Additionally, the inference of preconditions for termination has been explored, with Cook et al. [197] focusing on generating necessary preconditions, while Ganty and Genaim [198] and Massé [199] derive these by complementing over-approximations of non-terminating states.

State-of-the-art tools in this area include VeryMax [200], AProVE [201], Ultimate [202, 203], and LoAT [204]. Specifically, VeryMax [200] searches for non-termination witnesses using quasi-invariants but lacks relative completeness guarantees [195]. AProVE [201] uses recurrence sets to prove non-termination in Java programs with non-determinism but has limitations with nested loops Brockschmidt et al. [205]. The method by Urban, Gurfinkel, and Kahsai [206] attempts to prove either termination or non-termination in programs with non-determinism by incrementally refining termination arguments and using a safety prover to identify non-terminating traces. Other approaches, such as those in Ultimate [202, 203], use geometric series and constraint solving to handle deterministic programs. Computational efficiency and non-determinism remain significant challenges for termination provers; see Shi et al. [207] for a comparison. Recently, graph neural networks have also been employed to tackle (non-)termination under weaker (stochastic) guarantees [208].

The work presented in this chapter does not aim to prove termination but could significantly benefit from termination analysis to infer termination-aware quantities for the global loop counter. By understanding which input data leads to terminating executions, we could refine the impact quantification. At the current stage, an input variable that influences program termination but not the number of iterations might be incorrectly deemed unused.

8.6 Summary

This chapter presented a static analysis for quantifying the impact of input variables on the number of iterations of a program. The following chapter presents the experimental evaluation of our approach and concludes the main body of this thesis.

Ce chapitre a présenté une analyse statique pour quantifier l'impact des variables d'entrée sur le nombre d'itérations d'un programme. Le chapitre suivant présente l'évaluation expérimentale de notre approche et conclut le corps principal de cette thèse.

[187]: Berdine et al. (2007), 'Variance analyses from invariance analyses'
[188]: Heizmann et al. (2013), 'Linear Ranking for Linear Lasso Programs'
[189]: Urban (2015), 'FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution)'
[179]: Podelski et al. (2004), 'A Complete Method for the Synthesis of Linear Ranking Functions'
[190]: Bradley et al. (2005), 'Linear Ranking with Reachability'
[191]: Tsitovich et al. (2011), 'Loop Summarization and Termination Analysis'
[192]: Courant et al. (2017), 'Precise Widening Operators for Proving Termination by Abstract Interpretation'
[193]: Alias et al. (2010), 'Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs'
[194]: Brockschmidt et al. (2013), 'Better Termination Proving through Cooperation'
[195]: Larraz et al. (2013), 'Proving termination of imperative programs using Max-SMT'
[196]: Chatterjee et al. (2021), 'Proving non-termination by program reversal'
[197]: Cook et al. (2008), 'Proving Conditional Termination'
[198]: Ganty et al. (2013), 'Proving Termination Starting from the End'
[199]: Massé (2014), 'Policy Iteration-Based Conditional Termination and Ranking Functions'
[200]: Borralleras et al. (2017), 'Proving Termination Through Conditional Termination'
[201]: Giesl et al. (2017), 'Analyzing Program Termination and Complexity Automatically with AProVE'
[202]: Leike et al. (2018), 'Geometric Nontermination Arguments'
[203]: Chen et al. (2018), 'Advanced automata-based algorithms for program termination checking'
[204]: Frohn et al. (2019), 'Proving Non-Termination via Loop Acceleration'
[205]: Brockschmidt et al. (2011), 'Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode'
[206]: Urban et al. (2016), 'Synthesizing Ranking Functions from Bits and Pieces'
[207]: Shi et al. (2022), 'Large-scale analysis of non-termination bugs in real-world OSS projects'
[208]: Alon et al. (2022), 'Using graph neural networks for program termination'

9 Evaluation of the Timing Analysis

This chapter presents TimeSec[1] written in about 3000 lines of Python code. The global loop bound analysis is based on the numerical library Apron [209], we instrumented the analysis with widening and narrowing operators after 2 fixpoint iterations. For the linear programming encoding, we used the Python library SciPy.[2] In this chapter, we discuss some implementation features that make the analysis scalable, precise, and able to handle real-world programs. We evaluate TimeSec on the s2n-bignum library,[3] a collection of arithmetic routines designed for cryptographic applications, and on the SV-Comp benchmarks.[4] An artifact of TimeSec, including the source code, the benchmarks, and the evaluation results, is available on Zenodo.[5] This chapter is based on the work presented at the 31st Static Analysis Symposium (SAS 2024) [48].

1: github.com/denismazzucato/timesec

[209]: Jeannet et al. (2009), 'Apron: A Library of Numerical Abstract Domains for Static Analysis'

2: scipy.org

3: github.com/awslabs/s2n-bignum

4: sv-comp.sosy-lab.org/2024

5: doi.org/10.5281/zenodo.11507271

[48]: Mazzucato et al. (2024), 'Quantitative Static Timing Analysis'

Ce chapitre présente TimeSec[1], écrit en environ 3000 lignes de code Python. L'analyse des bornes globales des boucles est basée sur la bibliothèque numérique Apron [209], et nous avons instrumenté l'analyse avec des opérateurs de renforcement et d'affaiblissement après 2 itérations du point fixe. Pour l'encodage en programmation linéaire, nous avons utilisé la bibliothèque Python SciPy[2]. Dans ce chapitre, nous discutons de certaines fonctionnalités d'implémentation qui rendent l'analyse évolutive, précise et capable de gérer des programmes réels. Nous évaluons TimeSec sur la bibliothèque s2n-bignum [4], une collection de routines arithmétiques conçues pour des applications cryptographiques, ainsi que sur les benchmarks SV-Comp [2]. Un artefact de TimeSec, incluant le code source, les benchmarks et les résultats de l'évaluation, est disponible sur Zenodo[5]. Ce chapitre est basé sur les travaux présentés lors du 31e Symposium sur l'analyse statique (SAS 2024) [48].

9.1 Implementation Discussion

The syntactic dependency analysis of Section 3.6 can be helpful to reduce the number of variables in the abstract domain during the global loop bound analysis. In fact, the syntactic dependency analysis is used to determine an over-approximation of the set of relevant variables for the global loop counter, for each program location. We employ such information to apply program slicing. As a consequence, we reduce the number of variables in the underlying abstract domain, and avoid analyzing irrelevant statements. The program under analysis can be evaluated even in presence of statements and expressions that are hard to handle, *e.g.* bitwise operations, array manipulation, and function calls to name a few, as long as they are not relevant. Indeed, excluding irrelevant statements from the analysis does not affect the global loop bounds. The excluded statements are never looping structures as they are always relevant.

Prog. 8.1 (Program `Add`, computing the sum of two numbers `x` and `y` into `z`.):

```
1  def Add(p, z, m, x, n, y):
2    r = min(p, m)
3    s = min(p, n)
4    if (r < s):
5      t = p - s
6      q = s - r
7      # ...
8      for (; r > 0; r--):
9        # ...
10     do:
11       # ...
12       q--
13       # ...
14     while (q > 0)
15 else:
16   t = p - r
17   q = r - s
18   # ...
19   for (; s > 0; s--):
20     # ...
21   for (; q > 0; q--):
22     # ...
23 if (t > 0):
24   # ...
25   while (t > 0):
26     # ...
27     t--
28     # ...
```

[36]: Cousot et al. (1977), 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'

Prog. 9.1: Simplified first loop of Program 8.1.

```
1 assert r >= 0
2 while (r > 0):
3   r = r - 1
```

We recall from Section 3.6, the syntactic dependency analysis already serves as an input data usage analysis, identifying variables with zero impact on the global loop counter.

Example 9.1.1 Regarding the Program 8.1, the syntactic dependency analysis is able to discover that most of the variables are irrelevant for the global loop bound. Therefore, we are able to exclude most of the statements, including the bitwise operations regarding the remainder (*e.g.* Line 14), the array indices (*e.g.* Line 10), the conditional for the padding at the end (cf. Line 50). Overall, we excluded 33 from the original 52 lines of code (about the 60%), and the analysis was able to handle the program with ease without any specific handling for the excluded statements. Regarding the amount of variables, we reduced the number of variables from 13 to 7 (without counting the global loop counter $\texttt{nit}$).

Moreover, we combine forward and backward phases to provide tighter invariants in the global loop bound analysis, using the abstract domain of conjunctions of linear constraints (cf. Section 8.3.1). Specifically, an initial forward reachability analysis enhances both the syntactic dependency analysis and the global loop bound analysis. Additionally, we employ a narrowing operator to refine the upper bound of the least fixpoint computed by the widening operator [36].

Example 9.1.2 Consider the first for-loop at Line 9 of the Program 8.1. For simplicity, we reported it in the form of a while loop in Program 9.1. Without a forward pre-analysis, the backward analysis would infer that the global number of iterations is always greater or equal to the initial value of `r`. The missing information is that the value of `r` is always non-negative at the beginning of the loop. However, a forward pre-analysis could easily propagate such information to the backward analysis. As a consequence, the backward analysis would infer that the global number of iterations is always equal to the initial value of `r`. The main difference of the two approaches can be observed when the result of the backward analysis is used to verify the global loop bound property. With the invariant discovered without the forward analysis, cf. $\texttt{r} \geq \texttt{nit}$, the impact quantity maximizes the linear programming problem $\texttt{Range}^{\texttt{nit}}_{\texttt{W}}(\texttt{r} \geq \texttt{nit})$ to u (cf. $0 \leq \texttt{r} \leq u$), for all input variables $\texttt{W}$, even when $\texttt{i} \neq \texttt{r}$. The linear programming problem $\texttt{Range}^{\texttt{nit}}_{\texttt{W}}(\texttt{r} \geq \texttt{nit})$ is:

$$\begin{aligned}
&\text{maximize} && k && (9.1)\\
&\text{subject to} && \texttt{r} \geq \overline{\texttt{nit}} && (9.2)\\
&&\wedge\ && \texttt{r} \geq \underline{\texttt{nit}} && (9.3)\\
&&\wedge\ && 0 \leq \texttt{r} \leq u && (9.4)\\
&&\wedge\ && 0 \leq k \leq \overline{\texttt{nit}} - \underline{\texttt{nit}} && (9.5)
\end{aligned}$$

In this case, the variable $\overline{\texttt{nit}}$ and $\underline{\texttt{nit}}$ are not constrained to be equal, thus they can be minimized and maximized independently as long as they satisfy the other constraints, resulting in $k = u$. On the other hand, the invariant discovered with the help of the forward pre-analysis

is $\mathtt{r} = \mathtt{nit}$. The impact quantity maximizes the linear programming problem $\mathtt{Range}_{\mathtt{W}}^{\mathtt{nit}}(\mathtt{r} = \mathtt{nit})$ to 0 whenever the input variable $\mathtt{i} \neq \mathtt{r}$. In such case, the linear programming problem $\mathtt{Range}_{\mathtt{W}}^{\mathtt{nit}}(\mathtt{r} = \mathtt{nit})$ is:

$$\begin{aligned}
&\text{maximize} && k && (9.6)\\
&\text{subject to} && \mathtt{r} = \overline{\mathtt{nit}} && (9.7)\\
&&\wedge\ && \mathtt{r} = \underline{\mathtt{nit}} && (9.8)\\
&&\wedge\ && 0 \leq \mathtt{r} \leq u && (9.9)\\
&&\wedge\ && 0 \leq k \leq \overline{\mathtt{nit}} - \underline{\mathtt{nit}} && (9.10)
\end{aligned}$$

Note that, both $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$ are constrained to be equal. Therefore, their maximum distance is 0.

Example 9.1.3 To show that also the syntactic dependency analysis can benefit from a forward pre-analysis (and in general, from a numerical analysis [75]), consider the program on the side. It assigns first the value of y to x and then subtracts y from x. The result is that x is zero at the end of the program execution, while y maintains its input value. Let us assume we are interested in the variables that are relevant to compute the value of x. Without a forward pass, the syntactic dependency analysis (a backward analysis) would infer that y is relevant for the value of x after handling the second assignment at Line 2. Then, the first assignment at Line 1 would add no dependency as x is overwritten. On the other hand, a forward analysis could be able to infer that at the end of the program, the value of x is zero. As a consequence, the information that x is a constant value supersedes the information that x is used (at the end of the program). Therefore, the syntactic dependency analysis would infer that y is, in fact, irrelevant.

```
1 x = y
2 x = x - y
```

[75]: Parolini et al. (2024), 'Sound Abstract Nonexploitability Analysis'

9.2 Timing Side-Channels

In this section, we showcase the potential of TimeSec on the s2n-bignum library.[6] The s2n-bignum library is a collection of arithmetic routines designed for cryptographic applications. All the routines are written in pure machine code, designed to be callable from C and other high-level languages. Each function is written in a constant-time style, to avoid leaking information through timing side-channels. Constant-time means that the execution time of an s2n-bignum operation is independent of the actual numbers involved, depending only on their nominal sizes. If a result does not fit in the provided size, it is systematically truncated modulo that size. Allocation of memory is always the caller's responsibility, the s2n-bignum interface only uses pointers to pre-existing arrays. The developers avoid the use of certain machine instructions known to be problematic for constant-time execution, such as the division instruction. Furthermore, on ARM platforms, the library sets the DIT (Data Independent Timing) bit to have hardware guaranteed constant-time execution.

6: github.com/awslabs/s2n-bignum

The library is fully verified for functional correctness in HOL Light [21], but the verification of the constant-time property is still ongoing. At present, the constant-time property is enforced by the strict compliance to the constant-time design discipline and the use of empirical testing.

[21]: Harrison (2009), 'HOL Light: An Overview'

Their empirical result[7] shows that the variation in runtime with respect to the data being manipulated is within a few percent in all the cases. Unfortunately, the empirical study is not sufficient to guarantee the constant-time property, as it is not exhaustive and does not cover all the possible inputs. On the other hand, the quantitative analysis of TimeSec provides a formal verification of the constant-time property. In particular, whenever an input variable has no impact on the global number of loop iterations, it is formally guaranteed that the number of iterations is independent of the values of that input variable. Formally, a program P is free of timing side-channels with respect to an input variable $\mathtt{i} \in \Delta$, if and only if $P \models \mathscr{B}_{\mathtt{i}}^{\leq 0}$. By Theorem 4.2.4, we know that this is implied from $\mathtt{Range}_{\mathtt{W}}^{\mathtt{nit}}(\Lambda^{\leftarrow}[\![P]\!]) \leq 0$. Therefore, the verification of *timing side-channel freedom* is sound with respect to our quantitative analysis of input variables. We partition the input variables of the s2n-bignum library into two subsets. The nominal size variables and additional parameters that may safely influence the runtime into Δ_{S}. The variables that represent the actual numerical values and additional parameters that, instead, should not influence the execution time into Δ_{N}. The s2n-bignum library is free of timing side-channels, whenever for any program P in s2n-bignum and any numerical input variable $\mathtt{i} \in \Delta_{\mathrm{N}}$, it holds that $\mathtt{Range}_{\mathtt{W}}^{\mathtt{nit}}(\Lambda^{\leftarrow}[\![P]\!]) = 0$.

7: (Last accessed: 14th May 2024) github.com/awslabs/s2n-bignum?tab=readme-ov-file#benchmarking-and-constant-time

For our setup, we consider the disassembled operations[8] of the s2n-bignum library as input programs with a few rewriting steps to fit the set of supported operations of our tool. Mostly, the rewriting steps soundly resolve the few jumps that arise from the disassembling process. Our benchmark contains a total of 72 disassembled arithmetic routines, excluding only a single operation (program `bignum_modexp`) that has function calls, which our tool does not yet support. On average, each program has about 83 lines of code, for a total of 5984 lines of code.

8: We used Ghidra (ghidra-sre.org) to disassemble the library and extract the arithmetic routines.

The library contains a total of 1172 variables, 272 of which are input variables. Table 9.1 reports the analysis findings for the input variables of the s2n-bignum library: column Maybe Dangerous reports variables which could be prone to timing side-channel attacks (namely $\mathtt{Range}_{\mathtt{W}}^{\mathtt{nit}}(\Lambda^{\leftarrow}[\![P]\!]) > 0$), column Zero Impact reports the variables with an impact quantity of zero (namely $\mathtt{Range}_{\mathtt{W}}^{\mathtt{nit}}(\Lambda^{\leftarrow}[\![P]\!]) = 0$). The property $\mathscr{B}_{\mathtt{i}}^{\leq 0}$ holds for input variables $\mathtt{i}$ that have an impact quantity of zero (column Zero Impact). Overall, we soundly verified that 187 (69%) of the input variables do not influence the global number of iterations, while 85 (31%) are maybe dangerous and maybe susceptible to timing side-channel attacks. Column Safe Δ_{S} reports the nominal size variables (called s_i), column Numerical Δ_{N} reports the numerical variables (called n_i, where i is the index of the variable as they appear in the function signature). Table 9.1 shows that no numerical variable is identified as potentially dangerous, indeed Maybe Dangerous $\cap\, \Delta_{\mathrm{N}} = \emptyset$ in all rows. We conclude that the s2n-bignum library is *free of timing side-channels*.

Program	Input Variables I		Maybe	Zero
	Safe Δ_S	Numerical Δ_N	Dangerous	Impact
Add	s_1, s_3, s_5	n_2, n_4, n_6	s_1	s_3, s_5, n_2, n_4, n_6
Amontifier	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Amontmul	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Amontredc	s_1, s_3, s_6	n_2, n_4, n_5	s_1, s_3, s_6	n_2, n_4, n_5
Amontsqr	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Bitfield	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Bitsize	s_1	n_2	s_1	n_2
Cdiv	s_1, s_3	n_2, n_4, n_5	s_1, s_3	n_2, n_4, n_5
Cdiv_exact	s_1, s_3	n_2, n_4, n_5	s_1	n_2, s_3, n_4, n_5
Cld	s_1	n_2	s_1	n_2
Clz	s_1	n_2	s_1	n_2
Cmadd	s_1, s_4	n_2, n_3, n_5	s_1, s_4	n_2, n_3, n_5
Cmnegadd	s_1, s_4	n_2, n_3, n_5	s_1, s_4	n_2, n_3, n_5
Cmod	s_1	n_2, n_3	s_1	n_2, n_3
Cmul	s_1, s_4	n_2, n_3, n_5	s_1, s_4	n_2, n_3, n_5
Coprime	s_1, s_3	n_2, n_4, n_5	s_1, s_3	n_2, n_4, n_5
Copy	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Copy_row_from_table	s_3, s_4	n_1, n_2, n_5	s_3, s_4	n_1, n_2, n_5
Copy_row_from_table_16_neon	s_3	n_1, n_2, n_4	s_3	n_1, n_2, n_4
Copy_row_from_table_32_neon	s_3	n_1, n_2, n_4	s_3	n_1, n_2, n_4
Copy_row_from_table_8n_neon	s_3, s_4	n_1, n_2, n_5	s_3, s_4	n_1, n_2, n_5
Ctd	s_1	n_2	s_1	n_2
Ctz	s_1	n_2	s_1	n_2
Demont	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Digit	s_1	n_2, n_3	s_1	n_2, n_3
Digitsize	s_1	n_2	s_1	n_2
Divmod10	s_1	n_2	s_1	n_2
Emontredc	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Eq	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Even	s_1	n_2		s_1, n_2
Ge	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Gt	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Iszero	s_1	n_2	s_1	n_2
Le	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Lt	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Madd	s_1, s_3, s_5	n_2, n_4, n_6	s_1, s_3, s_5	n_2, n_4, n_6
Modadd	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Moddouble	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Modifier	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Modinv	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Modoptneg	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Modsub	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Montifier	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Montmul	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Montredc	s_1, s_3, s_6	n_2, n_4, n_5	s_1, s_3, s_6	n_2, n_4, n_5
Montsqr	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Mul	s_1, s_3, s_5	n_2, n_4, n_6	s_1, s_3, s_5	n_2, n_4, n_6
Muladd10	s_1	n_2, n_3	s_1	n_2, n_3
Mux	s_2	n_1, n_3, n_4, n_5	s_2	n_1, n_3, n_4, n_5
Mux16	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Negmodinv	s_1	n_2, n_3	s_1	n_2, n_3
Nonzero	s_1	n_2	s_1	n_2
Normalize	s_1	n_2	s_1	n_2
Odd	s_1	n_2		s_1, n_2
Of_word	s_1	n_2, n_3	s_1	n_2, n_3
Optadd	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Optneg	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Optsub	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Optsubadd	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Pow2	s_1	n_2, n_3	s_1	n_2, n_3
Shl_small	s_1, s_3	n_2, n_4, n_5	s_1, s_3	n_2, n_4, n_5
Shr_small	s_1, s_3	n_2, n_4, n_5	s_1	s_3, n_2, n_4, n_5
Sqr	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Sub	s_1, s_3, s_5	n_2, n_4, n_6	s_1	s_3, s_5, n_2, n_4, n_6
Word_bytereverse		n_1		n_1
Word_clz		n_1		n_1
Word_ctz		n_1		n_1
Word_divstep59		n_1, n_2, n_3, n_4		n_1, n_2, n_3, n_4
Word_max		n_1, n_2		n_1, n_2
Word_min		n_1, n_2		n_1, n_2
Word_negmodinv		n_1		n_1
Word_recip		n_1		n_1
Total Variables:	93	179	85	187

Table 9.1: Input composition of the s2n-bignum library. The variables Safe Δ_S are highlighted in green, while the variables Numerical Δ_N in red. No numerical variable should be Maybe Dangerous.

Table 9.2: Ablation study of TimeSec on the s2n-bignum benchmark. How input variables are influenced by the syntactic dependency analysis and other optimizations.

	w/o	TimeSec
Maybe Dangerous	266	85
Zero Impact	6	187

Table 9.3: Ablation study of TimeSec on the s2n-bignum benchmark. How analysis time (s) is influenced by the syntactic dependency analysis and other optimizations.

	w/o	TimeSec
Dep	0.0s	51.81s
Inv	16.36s	55.83s
LP	7.79s	0.27s
Tot	26.45s ±0.81	110.48s ±4.94

Additionally, we perform an ablation study to evaluate the impact of the dependency analysis and the other optimizations on our tool. The first row "w/o" of Table 9.2 reports the analysis findings without the various analysis stages of Section 9.1, while the second row "TimeSec" shows the finding of the full TimeSec analysis. Without the dependency analysis we do not apply program slicing anymore, we handle bitwise operations and array accesses with a conservative over-approximation that may lead to false positives. Generally, we notice that the invariant inferred from the global loop bound analysis alone is not tight enough to produce a precise quantification of the impact. Therefore, we are not able to infer useful insights from our analysis as 266 input variables are maybe dangerous. In particular, the 6 input variables with zero impact belong to acyclic programs. Regarding the analysis time, column Dep refers to the time of the dependency analysis, column Inv for the global loop bound analysis, and column LP for the quantification of impact. The time is reported in seconds for the evaluation of the 72 programs in Table 9.3 The last column Tot reports the total analysis time, with the standard deviation after the symbol ±. Table 9.3 does not show the time for parsing, logging and other overheads of the tool. We notice that without the optimizations, the analysis time is about 4 times faster than the full analysis, with most time spent on the linear programming problem as more variables need to be quantified. In this case, the standard deviation of the total analysis time (after ± in the column Tot) is the lowest, meaning that the analysis time is more consistent among programs. With only the dependency analysis on, the analysis usually takes around 50 seconds and, without optimizations, the global loop bound analysis is quite fast. The full analysis is about 100 seconds in total, with an average of 1.22 seconds per program. Most of the analysis time is spent on the syntactic dependency and the global loop bound analysis. Notably, the linear programming problem to quantify the impact of input variables takes less than half a second in total for the whole library. However, the analysis time is not consistent for all the programs, in fact, the analysis time for each program ranges from 0.03 to 33.88 seconds (standard deviation of about 4 seconds). Nevertheless, the full analysis is also the most precise, as it is able to exclude the most number of maybe dangerous variables.

In conclusion, the s2n-bignum library is a good candidate for our analysis, as it is a real-world cryptographic library potentially vulnerable against timing side-channel attacks for numerical input variables. Up to the decompilation phase and the chosen abstraction of the runtime, cf. the global number of iterations, our analysis soundly verifies that no input variable containing numerical data is susceptible to timing side-channel attacks.

9.3 SV-Comp Benchmarks

9: sv-comp.sosy-lab.org/2024

The SV-Comp benchmarks[9] are a collection of programs used for verification competition. The benchmarks are divided into different categories, such as termination, memory safety, reachability. As of 2024, the SV-Comp repository hosts thousands of programs, which are written in C and annotated with assertions. In this evaluation, we conduct a comprehensive study focusing on: the effect of changes in the input space, the

analysis time, and the categorization of input variables. We focus on the categories of Termination Crafted, and Termination Crafted Lit. These categories describe programs that are crafted to be challenging for termination analysis. In total, we selected 208 programs (68 from Termination Crafted, and 140 from termination Crafted Lit), with 5705 total lines of code. An average of 27 lines of code per program.

To evaluate TimeSec against the SV-Comp benchmarks, we consider the input variables as unbounded non-negative integers. We repeat the analysis 5 times, each time with a different bound on the input variables, ranging from $[0, 10]$ to $[-\infty, +\infty]$. Table 9.4 reports, for each bound range, the average quantity of impact (column Average), the standard deviation (column Std), and the analysis time for the dependency analysis (column Dep), the global loop bound analysis (column Inv), and the quantification of the impact (column LP). We exclude to take into account quantities that are infinite, as they would disrupt the average calculation. Note that, even in presence of a bounded input space, the impact of a variable could be infinite if the global loop bound analysis is not able to infer a bound on the possible number of iterations.

From Table 9.4, we observe that the average quantity of impact increases with the bound range (column Average). This is expected, as the larger the input space, the more the variance in the values of input variables, and the more the impact on the global number of iterations. However, as soon as the input space is unbounded, the measured quantities that are not infinite are very low. In this setting, a variable often has either an impact of 0 or $+\infty$. Regarding the analysis time, as expected we notice that the syntactic dependency analysis (column Dep) is not influenced by the bound range. The reason is that the syntactic dependency analysis is not a semantics analysis and does not depend on the values of the input variables. On the contrary, the global loop bound analysis (column Inv) and the quantification of the impact (column LP) are affected. From bounded to unbounded input space, we observe a reduction in the analysis time. In fact, in the context of a bigger input space, the analysis precision drops drastically and thus propagate less information faster. The global loop bound analysis is the most time-consuming part of the analysis. Overall, the analysis time is acceptable, with an average of 0.11 seconds per program, and a total of about 126 seconds for the whole benchmark suite, cf. 208 programs for 5 different bound ranges (1055 programs in total).

Table 9.4: Quantitative results for the SV-Comp benchmarks.

Benchmark	Bound Ranges	Quantities ($< \infty$)		Analysis Time (s)			
		Average	Std	Dep	Inv	LP	Tot
Termination Crafted (68 programs)	$0 - 10$	6.12	6.16	0.51	3.54	0.32	6.19
	$0 - 100$	50.13	48.44	0.51	3.5	0.32	6.18
	$0 - 1000$	500.13	483.18	0.5	3.53	0.32	6.15
	≥ 0	0.0	0.0	0.5	2.4	0.26	5.03
	$[-\infty, +\infty]$	0.0	0.0	0.46	2.01	0.05	4.38
Termination Crafted Lit (140 programs)	$0 - 10$	435.46	1892.32	1.56	16.6	1.02	23.08
	$0 - 100$	38248.52	194557.7	1.54	16.66	1.01	23.04
	$0 - 1000$	38577853.04	192885029.79	1.55	16.66	1.01	23.03
	≥ 0	0.0	0.0	1.49	9.66	0.77	15.8
	$[-\infty, +\infty]$	0.0	0.0	1.4	8.27	0.22	13.65

Table 9.5: Analysis findings for the SV-Comp benchmarks.

Benchmark	Bound Ranges	Variables	
		May Impact	Zero Impact
Termination Crafted (68 programs)	$0-10$	99/135	13/284
	$0-100$	99/135	13/284
	$0-1000$	99/135	13/284
	≥ 0	99/135	13/284
	$[-\infty,+\infty]$	105/141	7/278
Termination Crafted Lit (140 programs)	$0-10$	275/336	36/707
	$0-100$	277/338	34/705
	$0-1000$	277/338	34/705
	≥ 0	277/338	34/705
	$[-\infty,+\infty]$	286/348	25/695

Table 9.5 shows the composition of variables of the two categories of SV-Comp benchmarks. In terms of the k-bounded impact property (cf. Definition 4.1.1), the column May Impact corresponds to $\mathscr{B}^{\leq k}_{\text{IMPACT}_w}$ for $k > 0$, while the columns Zero Impact corresponds to $\mathscr{B}^{\leq 0}_{\text{i}}$. For each bound range, we report the number of input variables / total variables (cf., local and input variables together) that fall into each category. As expected, by enlarging the input space, the number of maybe dangerous variables increases, while the number of zero used variables decreases. Overall, our analysis is able to verify that most of the input variables in the SV-Comp benchmarks influence the global number of loop iterations. This is expected, as the benchmarks are crafted to be challenging for termination analysis, thus it is not surprising that the input variables have a significant impact on the global number of iterations. Unfortunately, such programs have invariants that are, on purpose, hard to infer. Our analysis can do little to achieve a tight quantification in such case. In conclusion, we notice that by enlarging the input space, the number of variables that may impact the runtime increases as more variety in the input values leads to more impact on the global number of iterations.

9.4 Summary

This chapter concludes the main body of the thesis, presenting the evaluation of our static analysis for quantitative program properties. Next, we present the conclusion and future work.

Ce chapitre conclut le corps principal de la thèse en présentant l'évaluation de notre analyse statique pour les propriétés quantitatives des programmes. Ensuite, nous présenterons la conclusion et les perspectives de travaux futurs.

Conclusion

10 Conclusion and Future Directions

Contributions

The aim of this thesis is to develop static analyses based on abstract interpretation to soundly quantify the impact of input data. We proposed a formal quantitative framework to reason about the impact of input data, parametrized by an impact quantifier to meet various needs. The static analyses we developed are *automatic* and *sound*, meaning they guarantee an upper (or lower depending on the property comparison operator) bound on the impact of input data without requiring user intervention. We applied our approach to two primary areas: quantify the impact of input features on neural network classification task, and assessing the impact of input data on the execution time of programs. Our experiments demonstrate the practical effectiveness of these analyses.

Quantitaitve Input Usage. Our first contribution is the design of a novel quantitative framework to quantify the impact of input data based on a chosen impact quantifier. We introduced three different quantifiers: Outcomes, Range, and QUsed, each providing a unique perspective on the impact of input variables. Outcomes counts distinct outcomes, Range measures the range of outcome values, and QUsed counts the number of input values not used to produce outcomes. We developed a static analysis to automatically compute a sound bound on the impact according to an abstract implementation of the given quantifier. These results are guaranteed to be sound by abstract interpretation, formally ensuring that the computed bounds are sound w.r.t. the actual impact. A prototype implementation, Impatto,[1] demonstrated the feasibility of our approach on a set of sample programs.

1: github.com/denismazzucato/impatto

Impact of Input Features. Our second contribution extends the quantitative framework to neural networks. Recognizing that Outcomes, Range, and QUsed do not meaningfully capture the impact of input features on neural network classification, we introduced two new quantifiers: Changes and QAUnused. These quantifiers are specifically designed to address the non-linear behavior inherent from the input space of neural networks. We evaluated our method by extending two tools, Impatto and Libra.[2] Our experiments confirmed that these quantifiers effectively capture the impact of input features on neural network classifications.

2: github.com/caterinaurban/libra

Impact on Execution Time. Our final contribution applies the quantitative framework to discover the impact related to the number of loop iterations in programs, which can estimate the effect of input variables on program execution time. We implemented this analysis in a tool called TimeSec,[3] which automatically computes an upper bound on the impact of input data on loop iterations. Notably, we certified that the s2n-bignum library[4] is free from timing side-channel vulnerabilities, providing stronger guarantees than previous empirical results.

3: github.com/denismazzucato/timesec

4: github.com/awslabs/s2n-bignum

Future Research Directions

We conclude with some perspectives on future research directions we would like to study.

Abstract Domains. Throughout this thesis, we utilized various abstract domains to drive our static analyses. For instance, in Chapter 4 (Quantitative Input Data Usage), the backward analysis was parameterized by convex numerical abstract domains such as the interval domain [56], the octagon domain [71], and the polyhedra domain [56]. In Chapter 6 (Quantitative Verification for Neural Networks), the forward pre-analysis employed the Symbolic [115], DeepPoly [117], and Neurify [118] abstract domains for neural network analysis. Future work could involve developing new *relational* abstract domains to capture non-linear variable relations more precisely, thereby improving analysis accuracy. At the current stage, the quantification of the impact heavily relies on the precision of the abstract analysis. To achieve a tight quantification, we noticed that we need strong invariants on a small subset of variables, cf. Chapter 9. Tailoring abstract domains to specific program behaviors could enhance the precision of the impact quantification.

[56]: Cousot et al. (1978), 'Automatic Discovery of Linear Restraints Among Variables of a Program'

[71]: Miné (2006), 'The octagon abstract domain'

Non-Termination and Non-Determinism. Extending our approach to handle non-termination and non-determinism presents an interesting research direction. A potential solution could involve integrating termination analysis [189, 196, 210] with backward analysis to account for potential non-termination states, refining our quantitative analysis with termination-aware impact quantities. Indeed, by knowing that some executions do not terminate after a certain loop could improve the precision of the quantitative bound as we avoid considering all the successive iterations as potential executions. To address non-determinism, we could consider the sequence of all possible non-deterministic choices as a parameter in the semantics [75, 113]. In such way, the non-deterministic behavior of the program could be determinized with an oracle that provides the sequence of choices during an execution. On an orthogonal direction, the development of quantitative properties that natively handle non-determinism at the property reasoning level could erase the need of ad-hoc semantics. For instance, the unused property (cf. Definition 3.1.1) and the k-bounded impact property w.r.t. the QUsedquantifier (cf. Definition 4.1.4) handle natively the non-determinism and do not require any specific semantics.

[189]: Urban (2015), 'FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution)'
[196]: Chatterjee et al. (2021), 'Proving non-termination by program reversal'
[210]: Gonnord et al. (2015), 'Synthesis of ranking functions using extremal counterexamples'

[75]: Parolini et al. (2024), 'Sound Abstract Nonexploitability Analysis'
[113]: Cousot et al. (2012), 'Probabilistic Abstract Interpretation'

Def. 3.1.1 (Unused)

$$\textsc{Unused}_W(\Lambda[\![P]\!]) \overset{def}{\Leftrightarrow} \forall \sigma \in \Lambda[\![P]\!], v \in \mathbb{V}^{|W|}.\ \sigma_0(W) \neq v \implies \exists \sigma' \in \Lambda[\![P]\!].\ \sigma'_0 =_{\Delta \setminus W} \sigma_0 \wedge \sigma'_0(W) = v \wedge \sigma_\omega = \sigma'_\omega$$

Def. 4.1.4 (QUsed)

$$\textsc{QUsed}_W(T) \overset{def}{=} \sup_{\sigma \in T} |\, Q_W(I_\sigma(T)) \setminus I_\sigma(T) \,|$$

$$I_\sigma(T) = \{\sigma'_0 \mid \sigma' \in T \wedge \rho(\sigma_\omega) = \rho(\sigma'_\omega)\}$$

$$Q_W(S) = \{s' \in \Sigma \mid s \in S \wedge s' =_{\Delta \setminus W} s\}$$

Data Science Code. A broader comparison of our quantitative input data usage with related properties, such as quantitative data leakage, could yield valuable insights. Addressing verification challenges posed by data science code, particularly dynamic code notebooks, could also be an interesting verification direction [95, 97, 211]. Especially, by the dynamic nature of code notebooks, which makes them susceptible to programming errors that are less common in other development environments. Notebooks allow users to execute code out of order or modify variables on the fly, leading to errors and inconsistencies in the analysis process. Such issues can have significant consequences,

[95]: Drobnjakovic et al. (2024), 'An Abstract Interpretation-Based Data Leakage Static Analysis'
[97]: Subotic et al. (2022), 'A Static Analysis Framework for Data Science Notebooks'
[211]: Negrini et al. (2023), 'Static Analysis of Data Transformations in Jupyter Notebooks'

particularly in fields where data-driven decisions play a crucial role. Furthermore, data science notebooks heavily rely on external libraries like NumPy or pandas for dataset analysis and manipulation. Quantifying the impact of programs utilizing these libraries is a major challenge due to their complex and low-level implementations. Although the source code is usually available, analyzing it is a demanding task and often impractical. Therefore, employing custom abstractions of these libraries can facilitate verification by abstracting unnecessary details and focusing on the high-level logic of the libraries.

Neural Network Verification. Future work could involve supporting additional activation functions beyond ReLU and adapting the quantifiers of Libra to accommodate other fairness notions, such as individual fairness [212]. Additionally, equipping Libra with a smarter reduced product between domains, capable of exchanging symbolic and concrete bounds, would be beneficial. Extending our approach to other machine learning models, such as support vector machines [213] or decision tree ensembles [159, 214], is also a promising direction.

Quantitative Properties. Exploring new impact definitions for cyber-physical systems [215] represents another promising research direction. Additionally, analyzing the impact of abstract domains in static program analyzers using pre-metrics, as defined in [216, 217], could be valuable.

Post-Spectre Era. As discussed in Chapter 8, certifying that some sensitive input variables do not affect execution time is crucial to prevent timing side-channel attacks. However, this may no longer hold in the post-Spectre attack era [168, 218]. Microarchitectural features, such as out-of-order and speculative execution, render constant-time programs vulnerable to timing side-channel attacks [174]. Incorporating these features into our analysis, as done by [175, 176], could extend our approach to soundly quantify the impact of input variables even in the presence of out-of-order and speculative execution.

[212]: Dwork et al. (2012), 'Fairness through awareness'

[213]: Cristianini et al. (2010), *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*

[159]: Breiman (2001), 'Random Forests'

[214]: Friedman (2001), 'Greedy Function Approximation: A Gradient Boosting Machine'

[215]: Kwiatkowska (2016), 'Advances and challenges of quantitative verification and synthesis for cyber-physical systems'

[216]: Campion et al. (2022), 'Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis'

[217]: Campion et al. (2023), 'A Formal Framework to Measure the Incompleteness of Abstract Interpretations'

[168]: Lipp et al. (2020), 'Meltdown: reading kernel memory from user space'

[218]: Kocher et al. (2018), 'Spectre Attacks: Exploiting Speculative Execution'

[174]: Cauligi et al. (2020), 'Constant-time foundations for the new spectre era'

[175]: Guarnieri et al. (2020), 'Spectector: Principled Detection of Speculative Information Flows'

[176]: Bard et al. (2024), 'Automatic and Incremental Repair for Speculative Information Leaks'

Bibliography

Here are the references in citation order.

[1] Herb Krasner. 'The cost of poor software quality in the US: A 2020 report'. In: *Proc. Consortium Inf. Softw. QualityTM (CISQTM)* 2 (2021) (cited on page 3).

[2] Joel Finch. 'Toyota sudden acceleration: a case study of the national highway traffic safety administration-recalls for change'. In: *Loy. Consumer L. Rev.* 22 (2009), p. 472 (cited on page 3).

[3] D. Briere and P. Traverse. 'AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems'. In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. 1993, pp. 616–623. DOI: 10.1109/FTCS.1993.627364 (cited on page 3).

[4] Nancy G. Leveson and Clark Savage Turner. 'Investigation of the Therac-25 Accidents'. In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940 (cited on pages 3, 4).

[5] Neil White, Stuart Matthews, and Roderick Chapman. 'Formal verification: will the seedling ever flower?' In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017), p. 20150402 (cited on page 3).

[6] A. E. Goodloe. 'Assuring Safety-Critical Machine Learning-Enabled Systems: Challenges and Promise'. In: *Computer* 56.09 (2023), pp. 83–88. DOI: 10.1109/MC.2023.3266860 (cited on page 3).

[7] Jeff Larson et al. *How We Analyzed the COMPAS Recidivism Algorithm*. https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm. 2016. URL: https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm (cited on page 3).

[8] Joy Buolamwini and Timnit Gebru. 'Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification'. In: *Conference on Fairness, Accountability and Transparency, FAT 2018, 23-24 February 2018, New York, NY, USA*. Ed. by Sorelle A. Friedler and Christo Wilson. Vol. 81. Proceedings of Machine Learning Research. PMLR, 2018, pp. 77–91 (cited on page 3).

[9] Ziad Obermeyer et al. 'Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations'. In: *Science* 366 (6464 2019), pp. 447–453 (cited on page 3).

[10] Matthew Kay, Cynthia Matuszek, and Sean A. Munson. 'Unequal Representation and Gender Stereotypes in Image Search Results for Occupations'. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*. Ed. by Bo Begole et al. ACM, 2015, pp. 3819–3828. DOI: 10.1145/2702123.2702520 (cited on page 3).

[11] European Commission. *Proposal for a Regulation Laying Down Harmonised Rules on Artificial Intelligence (Artificial Intelligence Act)*. https://digital-strategy.ec.europa.eu/en/library/proposal-regulation-laying-down-harmonised-rules-artificial-intelligence-artificial-intelligence. Apr. 2021 (cited on page 3).

[12] Edsger Wybe Dijkstra et al. *A discipline of programming*. Vol. 613924118. prentice-hall Englewood Cliffs, 1976 (cited on page 3).

[13] Robert W. Floyd. 'Assigning Meanings to Programs'. In: *Proceedings of Symposia in Applied Mathematics*. Vol. 19. 1967 (cited on page 4).

[14] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on page 4).

[15] Alonzo Church. 'A Note on the Entscheidungsproblem'. In: *J. Symb. Log.* 1.1 (1936), pp. 40–41. DOI: 10.2307/2269326 (cited on page 4).

[16] Alan M. Turing. 'On computable numbers, with an application to the Entscheidungsproblem'. In: *Proc. London Math. Soc.* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/PLMS/S2-42.1.230 (cited on pages 4, 158).

[17] H. Gordon Rice. 'Classes of recursively enumerable sets and their decision problems'. In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366. DOI: 10.1090/S0002-9947-1953-0053041-6 (cited on page 4).

[18] Patrick Cousot and Radhia Cousot. 'A gentle introduction to formal verification of computer systems by abstract interpretation'. In: *Logics and Languages for Reliability and Security*. Ed. by Javier Esparza, Bernd Spanfelner, and Orna Grumberg. Vol. 25. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2010, pp. 1–29. DOI: 10.3233/978-1-60750-100-8-1 (cited on page 4).

[19] M. Saqib Nawaz et al. 'A Survey on Theorem Provers in Formal Methods'. In: *CoRR* abs/1912.03028 (2019) (cited on page 4).

[20] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004 (cited on page 4).

[21] John Harrison. 'HOL Light: An Overview'. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 60–66. DOI: 10.1007/978-3-642-03359-9_4 (cited on pages 4, 163).

[22] Leonardo de Moura and Sebastian Ullrich. 'The Lean 4 Theorem Prover and Programming Language'. In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37 (cited on page 4).

[23] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. 'The Isabelle Framework'. In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Otmane Ait Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 33–38. DOI: 10.1007/978-3-540-71067-7_7 (cited on page 4).

[24] Ana Bove, Peter Dybjer, and Ulf Norell. 'A Brief Overview of Agda - A Functional Language with Dependent Types'. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6 (cited on page 4).

[25] Geoff Sutcliffe and Christian Suttner. 'Evaluating general purpose automated theorem proving systems'. In: *Artificial Intelligence* 131.1 (2001), pp. 39–54. DOI: https://doi.org/10.1016/S0004-3702(01)00113-8 (cited on page 4).

[26] K. Rustan M. Leino. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cited on page 4).

[27] Nikhil Swamy et al. 'Dependent types and multi-monadic effects in F'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodik and Rupak Majumdar. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655 (cited on pages 4, 5).

[28] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 - Where Programs Meet Provers'. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8 (cited on pages 4, 5).

[29] Edmund M. Clarke and E. Allen Emerson. 'Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic'. In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. DOI: 10.1007/BFB0025774 (cited on pages 4, 5).

[30] Jean-Pierre Queille and Joseph Sifakis. 'Specification and verification of concurrent systems in CESAR'. In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22 (cited on pages 4, 5).

[31] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 'A Tool for Checking ANSI-C Programs'. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15 (cited on page 5).

[32] Roberto Baldoni et al. 'A Survey of Symbolic Execution Techniques'. In: *ACM Comput. Surv.* 51.3 (2018). DOI: 10.1145/3182657 (cited on page 5).

[33] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 'KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs'. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224 (cited on page 5).

[34] Robin David et al. 'BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis'. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43 (cited on page 5).

[35] Corina S. Pasareanu and Neha Rungta. 'Symbolic PathFinder: symbolic execution of Java bytecode'. In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. Ed. by Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto. ACM, 2010, pp. 179–180. DOI: 10.1145/1858996.1859035 (cited on page 5).

[36] Patrick Cousot and Radhia Cousot. 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cited on pages xv, 5, 21, 42, 98, 162).

[37] Patrick Cousot. 'A Personal Historical Perspective on Abstract Interpretation'. In: *The French School of Programming*. Ed. by Bertrand Meyer. Springer, 2024, pp. 205–239. DOI: 10.1007/978-3-031-34518-0_9 (cited on page 5).

[38] Patrick Cousot. *Principles of abstract interpretation*. MIT Press, 2021, pp. 1–819 (cited on pages 5, 21).

[39] Bruno Blanchet et al. 'A static analyzer for large safety-critical software'. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. Ed. by Ron Cytron and Rajiv Gupta. ACM, 2003, pp. 196–207. DOI: 10.1145/781131.781153 (cited on page 5).

[40] Carmen M Reinhart and Kenneth S Rogoff. 'Growth in a Time of Debt'. In: *American Economic Review* (2010). DOI: 10.1257/AER.100.2.573 (cited on pages 6, 102).

[41] Thomas Herndon, Michael Ash, and Robert Pollin. 'Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff'. In: *Cambridge Journal of Economics* 38.2 (Dec. 2014), pp. 257–279. DOI: 10.1093/cje/bet075 (cited on pages 6, 102).

[42] Caterina Urban and Peter Müller. 'An Abstract Interpretation Framework for Input Data Usage'. In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 683–710. DOI: 10.1007/978-3-319-89884-1_24 (cited on pages xv, 6, 8, 24, 47, 48, 50, 59, 95).

[43] Roberto Giacobazzi and Isabella Mastroeni. 'Abstract Non-Interference: A Unifying Framework for Weakening Information-flow'. In: *ACM Trans. Priv. Secur.* 21.2 (2018), 9:1–9:31. DOI: 10.1145/3175660 (cited on pages 6, 51, 67, 95).

[44] Geoffrey Smith. 'Principles of Secure Information Flow Analysis'. In: *Malware Detection*. Ed. by Mihai Christodorescu et al. Vol. 27. Advances in Information Security. Springer, 2007, pp. 291–307. DOI: 10.1007/978-0-387-44599-1_13 (cited on page 6).

[45] Geoffrey Smith. 'On the Foundations of Quantitative Information Flow'. In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Luca de Alfaro. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 288–302. DOI: 10.1007/978-3-642-00596-1_21 (cited on pages 6, 7, 97).

[46] Denis Mazzucato, Marco Campion, and Caterina Urban. 'Quantitative Input Usage Static Analysis'. In: *NASA Formal Methods - 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Proceedings*. Ed. by Nathaniel Benz, Divya Gopinath, and Nija Shi. Vol. 14627. Lecture Notes in Computer Science. Springer, 2024, pp. 79–98. DOI: 10.1007/978-3-031-60698-4_5 (cited on pages 7, 65, 101, 102).

[47] Denis Mazzucato and Caterina Urban. 'Reduced Products of Abstract Domains for Fairness Certification of Neural Networks'. In: *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*. Ed. by Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi. Vol. 12913. Lecture Notes in Computer Science. Springer, 2021, pp. 308–322. DOI: 10.1007/978-3-030-88806-0_15 (cited on pages 7, 107, 112, 122, 129, 158).

[48] Denis Mazzucato, Marco Campion, and Caterina Urban. 'Quantitative Static Timing Analysis'. In: *SAS* (2024) (cited on pages 8, 143, 161).

[49] Wing H. Wong. 'Timing attacks on RSA: revealing your secrets through the fourth dimension'. In: *ACM Crossroads* 11.3 (2005), p. 5. DOI: 10.1145/1144396.1144401 (cited on pages 8, 157).

[50] Paul C. Kocher. 'Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems'. In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9 (cited on pages 8, 157).

[51] Hamza Omar, Masab Ahmad, and Omer Khan. 'GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms'. In: *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*. IEEE Computer Society, 2017, pp. 201–208. DOI: 10.1109/ICCD.2017.38 (cited on page 8).

[52] Casimir Kuratowski. 'Sur la notion de l'ordre dans la Théorie des Ensembles'. fre. In: *Fundamenta Mathematicae* 2.1 (1921), pp. 161–171 (cited on page 14).

[53] Alfred Tarski. 'A Lattice-Theoretical Fixpoint Theorem and its Applications'. In: *Pacific Journal of Mathematics* 5 (1955), pp. 285–309 (cited on page 20).

[54] Stephen Cole Kleene. *Introduction to Metamathematics*. 1952 (cited on page 20).

[55] Patrick Cousot and Radhia Cousot. 'Constructive Versions of Tarski's Fixed Point Theorems'. In: 81.1 (1979), pp. 43–57 (cited on page 20).

[56] Patrick Cousot and Nicolas Halbwachs. 'Automatic Discovery of Linear Restraints Among Variables of a Program'. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 84–96. DOI: 10.1145/512760.512770 (cited on pages 21, 43, 117, 148, 149, 172).

[57] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, 2020 (cited on page 21).

[58] Michael R. Clarkson and Fred B. Schneider. 'Hyperproperties'. In: *J. Comput. Secur.* 18.6 (2010), pp. 1157–1210. DOI: 10.3233/JCS-2009-0393 (cited on pages 23, 98).

[59] Patrick Cousot. 'Constructive design of a hierarchy of semantics of a transition system by abstract interpretation'. In: *Theor. Comput. Sci.* 277.1-2 (2002), pp. 47–103. DOI: 10.1016/S0304-3975(00)00313-3 (cited on page 24).

[60] Patrick Cousot. 'Calculational Design of [In]Correctness Transformational Program Logics by Abstract Interpretation'. In: *Proc. ACM Program. Lang. 8, POPL, Article 7*. London, UK: ACM Press, New York, NY, Jan. 2024, p. 62 (cited on page 24).

[61] Jürgen Schmidt. 'Beiträge zur Filtertheorie. II'. In: *Mathematische Nachrichten*. Vol. 10. 3-4. 1953, pp. 197–232. DOI: 10.1002/mana.19530100309 (cited on page 24).

[62] Patrick Cousot and Radhia Cousot. 'Abstract Interpretation Frameworks'. In: *J. Log. Comput.* 2.4 (1992), pp. 511–547. DOI: 10.1093/LOGCOM/2.4.511 (cited on page 26).

[63] Caterina Urban. 'Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes)'. PhD thesis. École Normale Supérieure, Paris, France, 2015 (cited on page 32).

[64] Patrick Cousot and Radhia Cousot. 'Static Determination of Dynamic Properties of Programs'. In: *Second International Symposium on Programming*. 1976, pp. 106–130 (cited on pages 34, 42, 111).

[65] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. 'Interval arithmetic: From principles to implementation'. In: *J. ACM* 48.5 (2001), pp. 1038–1068. DOI: 10.1145/502102.502106 (cited on pages 34, 42, 111).

[66] Bogdan Aman et al. 'Foundations of Reversible Computation'. In: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*. Ed. by Irek Ulidowski et al. Vol. 12070. Lecture Notes in Computer Science. Springer, 2020, pp. 1–40. DOI: 10.1007/978-3-030-47361-7_1 (cited on page 38).

[67] Antoine Miné. 'Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation'. In: *Found. Trends Program. Lang.* 4.3-4 (2017), pp. 120–372. DOI: 10.1561/2500000034 (cited on page 42).

[68] Raphaël Monat et al. 'Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14572. Lecture Notes in Computer Science. Springer, 2024, pp. 387–392. DOI: 10.1007/978-3-031-57256-2_26 (cited on page 42).

[69] Philippe Granger. 'Static Analysis of Linear Congruence Equalities among Variables of a Program'. In: *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'91)*. Ed. by Samson Abramsky and T. S. E. Maibaum. Vol. 493. Lecture Notes in Computer Science. Springer, 1991, pp. 169–192. DOI: 10.1007/3-540-53982-4_10 (cited on pages 42, 43).

[70] Michael Karr. 'Affine Relationships Among Variables of a Program'. In: *Acta Informatica* 6 (1976), pp. 133–151. DOI: 10.1007/BF00268497 (cited on page 43).

[71] Antoine Miné. 'The octagon abstract domain'. In: *High. Order Symb. Comput.* 19.1 (2006), pp. 31–100. DOI: 10.1007/S10990-006-8609-1 (cited on pages 43, 148, 149, 172).

[72] Patrick Cousot and Radhia Cousot. 'Systematic Design of Program Analysis Frameworks'. In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen. ACM Press, 1979, pp. 269–282. DOI: 10.1145/567752.567778 (cited on pages 43, 112, 122).

[73] Caterina Urban et al. 'Perfectly parallel fairness certification of neural networks'. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 185:1–185:30. DOI: 10.1145/3428253 (cited on pages 50, 122, 126–128, 136, 140, 158).

[74] Isabella Mastroeni and Michele Pasqua. 'Domain Precision in Galois Connection-Less Abstract Interpretation'. In: *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*. Ed. by Manuel V. Hermenegildo and José F. Morales. Vol. 14284. Lecture Notes in Computer Science. Springer, 2023, pp. 434–459. DOI: 10.1007/978-3-031-44245-2_19 (cited on pages 51, 95).

[75] Francesco Parolini and Antoine Miné. 'Sound Abstract Nonexploitability Analysis'. In: *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*. Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Vol. 14500. Lecture Notes in Computer Science. Springer, 2024, pp. 314–337. DOI: 10.1007/978-3-031-50521-8_15 (cited on pages 54, 163, 172).

[76] Coenraad Bron and Joep Kerbosch. 'Finding All Cliques of an Undirected Graph (Algorithm 457)'. In: *Commun. ACM* 16.9 (1973), pp. 575–576 (cited on page 83).

[77] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. 'CheckCell: data debugging for spreadsheets'. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black and Todd D. Millstein. ACM, 2014, pp. 507–523. DOI: 10.1145/2660193.2660207 (cited on page 95).

[78] 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 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 186–197. DOI: 10.1145/964001.964017 (cited on page 95).

[79] Isabella Mastroeni. 'Abstract interpretation-based approaches to Security - A Survey on Abstract Non-Interference and its Challenging Applications'. In: *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*. Ed. by Anindya Banerjee et al. Vol. 129. EPTCS. 2013, pp. 41–65. DOI: 10.4204/EPTCS.129.4 (cited on page 95).

[80] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982 (cited on page 95).

[81] James W. Gray. 'Toward a Mathematical Foundation for Information Flow Security'. In: *IEEE* (1991). DOI: 10.1109/RISP.1991.130769 (cited on page 95).

[82] Boris Köpf and Andrey Rybalchenko. 'Automation of Quantitative Information-Flow Analysis'. In: *Formal Methods for Dynamical Systems - 13th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2013, Bertinoro, Italy, June 17-22, 2013. Advanced Lectures*. Ed. by Marco Bernardo et al. Vol. 7938. Lecture Notes in Computer Science. Springer, 2013, pp. 1–28. DOI: 10.1007/978-3-642-38874-3_1 (cited on pages 95, 97).

[83] Boris Köpf and Andrey Rybalchenko. 'Approximation and Randomization for Quantitative Information-Flow Analysis'. In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 2010, pp. 3–14. DOI: 10.1109/CSF.2010.8 (cited on pages 96, 97).

[84] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 'LeakWatch: Estimating Information Leakage from Java Programs'. In: *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*. Ed. by Miroslaw Kutylowski and Jaideep Vaidya. Vol. 8713. Lecture Notes in Computer Science. Springer, 2014, pp. 219–236. DOI: 10.1007/978-3-319-11212-1_13 (cited on page 97).

[85] Quoc-Sang Phan et al. 'Symbolic quantitative information flow'. In: *ACM SIGSOFT Softw. Eng. Notes* 37.6 (2012), pp. 1–5. DOI: 10.1145/2382756.2382791 (cited on pages 97, 157).

[86] Quoc-Sang Phan et al. 'Quantifying information leaks using reliability analysis'. In: *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*. Ed. by Neha Rungta and Oksana Tkachuk. ACM, 2014, pp. 105–108. DOI: 10.1145/2632362.2632367 (cited on page 97).

[87] Seemanta Saha et al. 'Obtaining Information Leakage Bounds via Approximate Model Counting'. In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1488–1509. DOI: 10.1145/3591281 (cited on pages 97, 157).

[88] Mounir Assaf et al. 'Hypercollecting semantics and its application to static analysis of information flow'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 874–887. DOI: 10.1145/3009837.3009889 (cited on pages 97, 158).

[89] David Clark, Sebastian Hunt, and Pasquale Malacaria. 'A static analysis for quantifying information flow in a simple imperative language'. In: *J. Comput. Secur.* 15.3 (2007), pp. 321–371. DOI: 10.3233/JCS-2007-15302 (cited on pages 97, 158).

[90] Guillaume Girol, Guilhem Lacombe, and Sébastien Bardin. 'Quantitative Robustness for Vulnerability Assessment'. In: *Proc. ACM Program. Lang.* 8.PLDI (2024), pp. 741–765. DOI: 10.1145/3656407 (cited on page 97).

[91] Stephen McCamant and Michael D. Ernst. 'Quantitative information flow as network flow capacity'. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 193–205. DOI: 10.1145/1375581.1375606 (cited on page 97).

[92] Linpeng Zhang and Benjamin Lucien Kaminski. 'Quantitative strongest post: a calculus for reasoning about the flow of quantitative information'. In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–29. DOI: 10.1145/3527331 (cited on page 97).

[93] Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. 'Quantitative Safety and Liveness'. In: *Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*. Ed. by Orna Kupferman and Pawel Sobocinski. Vol. 13992. Lecture Notes in Computer Science. Springer, 2023, pp. 349–370. DOI: 10.1007/978-3-031-30829-1_17 (cited on page 97).

[94] J. Brownlee. *Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python*. Machine Learning Mastery, 2020 (cited on page 98).

[95] Filip Drobnjakovic, Pavle Subotic, and Caterina Urban. 'An Abstract Interpretation-Based Data Leakage Static Analysis'. In: *Theoretical Aspects of Software Engineering - 18th International Symposium, TASE 2024, Guiyang, China, July 29 - August 1, 2024, Proceedings*. Ed. by Wei-Ngan Chin and Zhiwu Xu. Vol. 14777. Lecture Notes in Computer Science. Springer, 2024, pp. 109–126. DOI: 10.1007/978-3-031-64626-3_7 (cited on pages 98, 172).

[96] Pavle Subotic, Uros Bojanic, and Milan Stojic. 'Statically detecting data leakages in data science code'. In: *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*. Ed. by Laure Gonnord and Laura Titolo. ACM, 2022, pp. 16–22. DOI: 10.1145/3520313.3534657 (cited on page 98).

[97] Pavle Subotic, Lazar Milikic, and Milan Stojic. 'A Static Analysis Framework for Data Science Notebooks'. In: *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 13–22. DOI: 10.1109/ICSE-SEIP55303.2022.9794067 (cited on pages 98, 172).

[98] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 'Secure information flow by self-composition'. In: *Math. Struct. Comput. Sci.* 21.6 (2011), pp. 1207–1252. DOI: 10.1017/S0960129511000193 (cited on pages 98, 158).

[99] Tachio Terauchi and Alexander Aiken. 'Secure Information Flow as a Safety Problem'. In: *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. Ed. by Chris Hankin and Igor Siveroni. Vol. 3672. Lecture Notes in Computer Science. Springer, 2005, pp. 352–367. DOI: 10.1007/11547662_24 (cited on pages 98, 158).

[100] Timos Antonopoulos et al. 'Decomposition instead of self-composition for proving the absence of timing channels'. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 362–375. DOI: 10.1145/3062341.3062378 (cited on pages 98, 158).

[101] Patrick Cousot. 'Abstract Semantic Dependency'. In: *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*. Ed. by Bor-Yuh Evan Chang. Vol. 11822. Lecture Notes in Computer Science. Springer, 2019, pp. 389–410. DOI: 10.1007/978-3-030-32304-2_19 (cited on page 98).

[102] Andrew D. Gordon et al. 'Probabilistic programming'. In: *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. Ed. by James D. Herbsleb and Matthew B. Dwyer. ACM, 2014, pp. 167–181. DOI: 10.1145/2593882.2593900 (cited on page 98).

[103] Jan-Willem van de Meent et al. 'An Introduction to Probabilistic Programming'. In: *CoRR* abs/1809.10756 (2018) (cited on page 98).

[104] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 'Stochastic invariants for probabilistic termination'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 145–160. DOI: 10.1145/3009837.3009873 (cited on page 98).

[105] Krishnendu Chatterjee et al. 'Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs'. In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 55–78. DOI: 10.1007/978-3-031-13185-1_4 (cited on page 98).

[106] Toru Takisaka et al. 'Ranking and Repulsing Supermartingales for Reachability in Randomized Programs'. In: *ACM Trans. Program. Lang. Syst.* 43.2 (2021), 5:1–5:46. DOI: 10.1145/3450967 (cited on page 98).

[107] Peixin Wang et al. 'Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving'. In: *Proc. ACM Program. Lang.* 8.PLDI (2024), pp. 1361–1386. DOI: 10.1145/3656432 (cited on page 98).

[108] Krishnendu Chatterjee et al. 'Quantitative Bounds on Resource Usage of Probabilistic Programs'. In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (2024), pp. 362–391. DOI: 10.1145/3649824 (cited on page 98).

[109] Di Wang, Jan Hoffmann, and Thomas W. Reps. 'Central moment analysis for cost accumulators in probabilistic programs'. In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 559–573. DOI: 10.1145/3453483.3454062 (cited on page 98).

[110] Benjamin Lucien Kaminski et al. 'Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs'. In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389. DOI: 10.1007/978-3-662-49498-1_15 (cited on page 98).

[111] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 'Static analysis for probabilistic programs: inferring whole program properties from finitely many paths'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 447–458. DOI: 10.1145/2491956.2462179 (cited on page 98).

[112] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 'Guaranteed bounds for posterior inference in universal probabilistic programming'. In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 536–551. DOI: 10.1145/3519939.3523721 (cited on page 98).

[113] Patrick Cousot and Michael Monerau. 'Probabilistic Abstract Interpretation'. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 169–193. DOI: 10.1007/978-3-642-28869-2_9 (cited on pages 98, 172).

[114] Hong Yi Chen, Shaked Flur, and Supratik Mukhopadhyay. 'Termination Proofs for Linear Simple Loops'. In: *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. Ed. by Antoine Miné and David Schmidt. Vol. 7460. Lecture Notes in Computer Science. Springer, 2012, pp. 422–438. DOI: 10.1007/978-3-642-33125-1_28 (cited on page 104).

[115] Shiqi Wang et al. 'Formal Security Analysis of Neural Networks using Symbolic Intervals'. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 1599–1614 (cited on pages 111, 127, 137, 138, 172).

[116] Antoine Miné. 'Symbolic Methods to Enhance the Precision of Numerical Abstract Domains'. In: *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Vol. 3855. Lecture Notes in Computer Science. Springer, 2006, pp. 348–363. DOI: 10.1007/11609773_23 (cited on page 111).

[117] Gagandeep Singh et al. 'An abstract domain for certifying neural networks'. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 41:1–41:30. DOI: 10.1145/3290354 (cited on pages 111, 127, 138, 172).

[118] Shiqi Wang et al. 'Efficient Formal Safety Analysis of Neural Networks'. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 6369–6379 (cited on pages 111, 112, 138, 172).

[119] George B. Dantzig and B. Curtis Eaves. 'Fourier-Motzkin Elimination and Its Dual'. In: *J. Comb. Theory A* 14.3 (1973), pp. 288–297. DOI: 10.1016/0097-3165(73)90004-6 (cited on page 117).

[120] Aws Albarghouthi. 'Introduction to Neural Network Verification'. In: *Found. Trends Program. Lang.* 7.1-2 (2021), pp. 1–157. DOI: 10.1561/2500000051 (cited on page 127).

[121] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016 (cited on page 127).

[122] Guy Katz et al. 'Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks'. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 97–117. DOI: 10.1007/978-3-319-63387-9_5 (cited on pages 127, 138).

[123] Guy Katz et al. 'The Marabou Framework for Verification and Analysis of Deep Neural Networks'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 443–452. DOI: 10.1007/978-3-030-25540-4_26 (cited on page 127).

[124] Luca Pulina and Armando Tacchella. 'Checking Safety of Neural Networks with SMT Solvers: A Comparative Evaluation'. In: *AI*IA 2011: Artificial Intelligence Around Man and Beyond - XIIth International Conference of the Italian Association for Artificial Intelligence, Palermo, Italy, September 15-17, 2011. Proceedings*. Ed. by Roberto Pirrone and Filippo Sorbello. Vol. 6934. Lecture Notes in Computer Science. Springer, 2011, pp. 127–138. DOI: 10.1007/978-3-642-23954-0_14 (cited on page 127).

[125] Luca Pulina and Armando Tacchella. 'NeVer: a tool for artificial neural networks verification'. In: *Ann. Math. Artif. Intell.* 62.3-4 (2011), pp. 403–425. DOI: 10.1007/S10472-011-9243-0 (cited on page 127).

[126] Luca Pulina and Armando Tacchella. 'Challenging SMT solvers to verify neural networks'. In: *AI Commun.* 25.2 (2012), pp. 117–135. DOI: 10.3233/AIC-2012-0525 (cited on page 127).

[127] Elena Botoeva et al. 'Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis'. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 3291–3299. DOI: 10.1609/AAAI.V34I04.5729 (cited on page 127).

[128] Alessio Lomuscio and Lalit Maganti. 'An approach to reachability analysis for feed-forward ReLU neural networks'. In: *CoRR* abs/1706.07351 (2017) (cited on page 127).

[129] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. 'Evaluating Robustness of Neural Networks with Mixed Integer Programming'. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cited on page 127).

[130] Rudy Bunel et al. 'A Unified View of Piecewise Linear Neural Network Verification'. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 4795–4804 (cited on page 127).

[131] Alessandro De Palma et al. 'Improved Branch and Bound for Neural Network Verification via Lagrangian Decomposition'. In: *CoRR* abs/2104.06718 (2021) (cited on page 127).

[132] Rüdiger Ehlers. 'Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks'. In: *CoRR* abs/1705.01320 (2017) (cited on page 127).

[133] Tsui-Wei Weng et al. 'Towards Fast Computation of Certified Robustness for ReLU Networks'. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 5273–5282 (cited on page 127).

[134] Huan Zhang et al. 'Efficient Neural Network Robustness Certification with General Activation Functions'. In: *CoRR* abs/1811.00866 (2018) (cited on page 127).

[135] P. Henriksen and Alessio Lomuscio. 'Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search'. In: *European Conference on Artificial Intelligence*. 2020 (cited on page 127).

[136] Shiqi Wang et al. 'Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification'. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc'Aurelio Ranzato et al. 2021, pp. 29909–29921 (cited on page 127).

[137] Matthias König et al. 'Critically Assessing the State of the Art in Neural Network Verification'. In: *Journal of Machine Learning Research* 25.12 (2024), pp. 1–53 (cited on page 127).

[138] Xiaowei Huang et al. 'A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability'. In: *Comput. Sci. Rev.* 37 (2020), p. 100270. DOI: 10.1016/J.COSREV.2020.100270 (cited on pages 127, 128).

[139] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 'Fairness testing: testing software for discrimination'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. Ed. by Eric Bodden et al. ACM, 2017, pp. 498–510. DOI: 10.1145/3106237.3106277 (cited on pages 127, 128).

[140] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 'Automated directed fairness testing'. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 98–108. DOI: 10.1145/3238147.3238165 (cited on pages 127, 128).

[141] Florian Tramèr et al. 'FairTest: Discovering Unwarranted Associations in Data-Driven Applications'. In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 401–416. DOI: 10.1109/EUROSP.2017.29 (cited on pages 127, 128).

[142] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 'Probabilistic verification of fairness properties via concentration'. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 118:1–118:27. DOI: 10.1145/3360544 (cited on pages 127, 128).

[143] Aws Albarghouthi et al. 'FairSquare: probabilistic verification of program fairness'. In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 80:1–80:30. DOI: 10.1145/3133904 (cited on pages 127, 128).

[144] Aws Albarghouthi and Samuel Vinitsky. 'Fairness-Aware Programming'. In: *Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT*2019, Atlanta, GA, USA, January 29-31, 2019*. Ed. by danah boyd and Jamie H. Morgenstern. ACM, 2019, pp. 211–219. DOI: 10.1145/3287560.3287588 (cited on pages 127, 128).

[145] Francesco Ranzato and Marco Zanella. 'Abstract Interpretation of Decision Tree Ensemble Classifiers'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (04 Apr. 2020), pp. 5478–5486. DOI: 10.1609/AAAI.V34I04.5998 (cited on pages 127, 128).

[146] Francesco Ranzato, Caterina Urban, and Marco Zanella. 'Fairness-Aware Training of Decision Trees by Abstract Interpretation'. In: *International Conference on Information and Knowledge Management, Proceedings* (Oct. 2021), pp. 1508–1517. DOI: 10.1145/3459637.3482342 (cited on pages 127, 128).

[147] Abhinandan Pal et al. 'Abstract Interpretation-Based Feature Importance for SVMs'. In: (Oct. 2022) (cited on pages 127, 128).

[148] Hoang-Dung Tran et al. 'Quantitative Verification for Neural Networks using ProbStars'. In: *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2023, San Antonio, TX, USA, May 9-12, 2023*. ACM, 2023, 4:1–4:12. DOI: 10.1145/3575870.3587112 (cited on pages 127, 128).

[149] Teodora Baluta et al. 'Quantitative Verification of Neural Networks and Its Security Applications'. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro et al. ACM, 2019, pp. 1249–1264. DOI: 10.1145/3319535.3354245 (cited on pages 127, 128).

[150] Teodora Baluta et al. 'Scalable Quantitative Verification For Deep Neural Networks'. In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 312–323. DOI: 10.1109/ICSE43902.2021.00039 (cited on pages 127, 128).

[151] Yedi Zhang et al. 'BDD4BNN: A BDD-Based Quantitative Analysis Framework for Binarized Neural Networks'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 175–200. DOI: 10.1007/978-3-030-81685-8_8 (cited on pages 127, 128).

[152] Yedi Zhang et al. 'Precise Quantitative Analysis of Binarized Neural Networks: A BDD-based Approach'. In: *ACM Trans. Softw. Eng. Methodol.* 32.3 (2023), 62:1–62:51. DOI: 10.1145/3563212 (cited on pages 127, 128).

[153] Stefan Webb et al. 'A Statistical Approach to Assessing Neural Network Robustness'. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cited on pages 127, 128).

[154] Paulo Cortez et al. 'Modeling wine preferences by data mining from physicochemical properties'. In: *Decision Support Systems* 47 (4 Nov. 2009), pp. 547–553. DOI: 10.1016/j.dss.2009.05.016 (cited on page 129).

[155] Jack W Smith et al. 'Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus'. In: *Proceedings of the Annual Symposium on Computer Application in Medical Care* (Nov. 1988), p. 261 (cited on page 129).

[156] Joe Young. *Rain in Australia*. 2019. URL: https://www.kaggle.com/datasets/jsphyg/weather-dataset-rattle-package (cited on page 129).

[157] A. Unmoved. *Cure The Princess*. 2023. URL: https://www.kaggle.com/datasets/unmoved/cure-the-princess (cited on page 129).

[158] Diederik P Kingma and Jimmy Ba. 'Adam: A Method for Stochastic Optimization'. In: (Nov. 2014) (cited on page 130).

[159] Leo Breiman. 'Random Forests'. In: *Mach. Learn.* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324 (cited on pages 131, 132, 173).

[160] Jianlin Li et al. 'Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification'. In: *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*. Ed. by Bor-Yuh Evan Chang. Vol. 11822. Lecture Notes in Computer Science. Springer, 2019, pp. 296–319. DOI: 10.1007/978-3-030-32304-2_15 (cited on page 138).

[161] Reinhard Wilhelm et al. 'The worst-case execution-time problem - overview of methods and survey of tools'. In: *ACM Trans. Embed. Comput. Syst.* 7.3 (2008), 36:1–36:53. DOI: 10.1145/1347375.1347389 (cited on page 157).

[162] Pavel Cadek et al. 'Using Loop Bound Analysis For Invariant Generation'. In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by Nikolaj S. Bjørner and Arie Gurfinkel. IEEE, 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8603005 (cited on page 157).

[163] Andreas Ermedahl et al. 'Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis'. In: *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*. Ed. by Christine Rochange. Vol. 6. OASIcs. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007 (cited on page 157).

[164] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 'Compositional certified resource bounds'. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 467–478. DOI: 10.1145/2737924.2737955 (cited on page 157).

[165] Moritz Sinn, Florian Zuleger, and Helmut Veith. 'Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints'. In: *J. Autom. Reason.* 59.1 (2017), pp. 3–45. DOI: 10.1007/S10817-016-9402-4 (cited on page 157).

[166] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 'Differential Power Analysis'. In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25 (cited on page 157).

[167] Paul Kocher et al. 'Spectre Attacks: Exploiting Speculative Execution'. In: (2019). DOI: 10.1109/SP.2019.00002 (cited on page 157).

[168] Moritz Lipp et al. 'Meltdown: reading kernel memory from user space'. In: *Commun. ACM* 63.6 (2020), pp. 46–56. DOI: 10.1145/3357033 (cited on pages 157, 173).

[169] Boris Köpf and David A. Basin. 'An information-theoretic model for adaptive side-channel attacks'. In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 2007, pp. 286–296. DOI: 10.1145/1315245.1315282 (cited on page 157).

[170] Quoc-Sang Phan et al. 'Synthesis of Adaptive Side-Channel Attacks'. In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 328–342. DOI: 10.1109/CSF.2017.8 (cited on page 157).

[171] Pasquale Malacaria et al. 'Symbolic Side-Channel Analysis for Probabilistic Programs'. In: *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 2018, pp. 313–327. DOI: 10.1109/CSF.2018.00030 (cited on page 157).

[172] Lucas Bang et al. 'String analysis for side channels with segmented oracles'. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. Ed. by Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su. ACM, 2016, pp. 193–204. DOI: 10.1145/2950290.2950362 (cited on page 158).

[173] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 'Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT'. In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 387–400. DOI: 10.1109/CSF.2016.34 (cited on page 158).

[174] Sunjay Cauligi et al. 'Constant-time foundations for the new spectre era'. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 913–926. DOI: 10.1145/3385412.3385970 (cited on pages 158, 173).

[175] Marco Guarnieri et al. 'Spectector: Principled Detection of Speculative Information Flows'. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1–19. DOI: 10.1109/SP40000.2020.00011 (cited on pages 158, 173).

[176] Joachim Bard, Swen Jacobs, and Yakir Vizel. 'Automatic and Incremental Repair for Speculative Information Leaks'. In: *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*. Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Vol. 14500. Lecture Notes in Computer Science. Springer, 2024, pp. 291–313. DOI: 10.1007/978-3-031-50521-8_14 (cited on pages 158, 173).

[177] B. Dongol et al. 'Relative Security: Formally Modeling and (Dis)Proving Resilience Against Semantic Optimization Vulnerabilities'. In: *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 403–418. DOI: 10.1109/CSF61375.2024.00027 (cited on page 158).

[178] Alan M. Turing. 'Checking a Large Routine'. In: A corrected version is printed in [**Morris:1984:EPP**]. The original is reprinted in [**Williams:1989:EBC**]. 1949, pp. 67–69 (cited on page 158).

[179] Andreas Podelski and Andrey Rybalchenko. 'A Complete Method for the Synthesis of Linear Ranking Functions'. In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 239–251. DOI: 10.1007/978-3-540-24622-0_20 (cited on pages 158, 159).

[180] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 'Terminator: Beyond Safety'. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 415–418. DOI: 10.1007/11817963_37 (cited on page 158).

[181] Patrick Cousot and Radhia Cousot. 'An abstract interpretation framework for termination'. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 245–258. DOI: 10.1145/2103656.2103687 (cited on page 158).

[182] Caterina Urban. 'The Abstract Domain of Segmented Ranking Functions'. In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 43–62. DOI: 10.1007/978-3-642-38856-9_5 (cited on page 158).

[183] Caterina Urban and Antoine Miné. 'An Abstract Domain to Infer Ordinal-Valued Ranking Functions'. In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 412–431. DOI: 10.1007/978-3-642-54833-8_22 (cited on page 158).

[184] Caterina Urban and Antoine Miné. 'A Decision Tree Abstract Domain for Proving Conditional Termination'. In: *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*. Ed. by Markus Müller-Olm and Helmut Seidl. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 302–318. DOI: 10.1007/978-3-319-10936-7_19 (cited on page 158).

[185] Caterina Urban and Antoine Miné. 'Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation'. In: *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*. Ed. by Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen. Vol. 8931. Lecture Notes in Computer Science. Springer, 2015, pp. 190–208. DOI: 10.1007/978-3-662-46081-8_11 (cited on page 158).

[186] Caterina Urban and Antoine Miné. 'Inference of ranking functions for proving temporal properties by abstract interpretation'. In: *Comput. Lang. Syst. Struct.* 47 (2017), pp. 77–103. DOI: 10.1016/J.CL.2015.10.001 (cited on page 158).

[187] Josh Berdine et al. 'Variance analyses from invariance analyses'. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, 2007, pp. 211–224. DOI: 10.1145/1190216.1190249 (cited on pages 158, 159).

[188] Matthias Heizmann et al. 'Linear Ranking for Linear Lasso Programs'. In: *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*. Ed. by Dang Van Hung and Mizuhito Ogawa. Vol. 8172. Lecture Notes in Computer Science. Springer, 2013, pp. 365–380. DOI: 10.1007/978-3-319-02444-8_26 (cited on pages 158, 159).

[189] Caterina Urban. 'FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution)'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 464–466. DOI: 10.1007/978-3-662-46681-0_46 (cited on pages 158, 159, 172).

[190] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 'Linear Ranking with Reachability'. In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 491–504. DOI: 10.1007/11513988_48 (cited on pages 158, 159).

[191] Aliaksei Tsitovich et al. 'Loop Summarization and Termination Analysis'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 81–95. DOI: 10.1007/978-3-642-19835-9_9 (cited on pages 158, 159).

[192] Nathanaël Courant and Caterina Urban. 'Precise Widening Operators for Proving Termination by Abstract Interpretation'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 136–152. DOI: 10.1007/978-3-662-54577-5_8 (cited on pages 158, 159).

[193] Christophe Alias et al. 'Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs'. In: *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. Ed. by Radhia Cousot and Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 117–133. DOI: 10.1007/978-3-642-15769-1_8 (cited on page 159).

[194] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 'Better Termination Proving through Cooperation'. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 413–429. DOI: 10.1007/978-3-642-39799-8_28 (cited on page 159).

[195] Daniel Larraz et al. 'Proving termination of imperative programs using Max-SMT'. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 218–225 (cited on page 159).

[196] Krishnendu Chatterjee et al. 'Proving non-termination by program reversal'. In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 1033–1048. DOI: 10.1145/3453483.3454093 (cited on pages 159, 172).

[197] Byron Cook et al. 'Proving Conditional Termination'. In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 328–340. DOI: 10.1007/978-3-540-70545-1_32 (cited on page 159).

[198] Pierre Ganty and Samir Genaim. 'Proving Termination Starting from the End'. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 397–412. DOI: 10.1007/978-3-642-39799-8_27 (cited on page 159).

[199] Damien Massé. 'Policy Iteration-Based Conditional Termination and Ranking Functions'. In: *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. Ed. by Kenneth L. McMillan and Xavier Rival. Vol. 8318. Lecture Notes in Computer Science. Springer, 2014, pp. 453–471. DOI: 10.1007/978-3-642-54013-4_25 (cited on page 159).

[200] Cristina Borralleras et al. 'Proving Termination Through Conditional Termination'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 99–117. DOI: 10.1007/978-3-662-54577-5_6 (cited on page 159).

[201] Jürgen Giesl et al. 'Analyzing Program Termination and Complexity Automatically with AProVE'. In: *J. Autom. Reason.* 58.1 (2017), pp. 3–31. DOI: 10.1007/S10817-016-9388-Y (cited on page 159).

[202] Jan Leike and Matthias Heizmann. 'Geometric Nontermination Arguments'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. Lecture Notes in Computer Science. Springer, 2018, pp. 266–283. DOI: 10.1007/978-3-319-89963-3_16 (cited on page 159).

[203] Yu-Fang Chen et al. 'Advanced automata-based algorithms for program termination checking'. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 135–150. DOI: 10.1145/3192366.3192405 (cited on page 159).

[204] Florian Frohn and Jürgen Giesl. 'Proving Non-Termination via Loop Acceleration'. In: *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. Ed. by Clark W. Barrett and Jin Yang. IEEE, 2019, pp. 221–230. DOI: 10.23919/FMCAD.2019.8894271 (cited on page 159).

[205] Marc Brockschmidt et al. 'Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode'. In: *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Vol. 7421. Lecture Notes in Computer Science. Springer, 2011, pp. 123–141. DOI: 10.1007/978-3-642-31762-0_9 (cited on page 159).

[206] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 'Synthesizing Ranking Functions from Bits and Pieces'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 54–70. DOI: 10.1007/978-3-662-49674-9_4 (cited on page 159).

[207] Xiuhan Shi et al. 'Large-scale analysis of non-termination bugs in real-world OSS projects'. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 256–268. DOI: 10.1145/3540250.3549129 (cited on page 159).

[208] Yoav Alon and Cristina David. 'Using graph neural networks for program termination'. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 910–921. DOI: 10.1145/3540250.3549095 (cited on page 159).

[209] Bertrand Jeannet and Antoine Miné. 'Apron: A Library of Numerical Abstract Domains for Static Analysis'. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52 (cited on page 161).

[210] Laure Gonnord, David Monniaux, and Gabriel Radanne. 'Synthesis of ranking functions using extremal counterexamples'. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 608–618. DOI: 10.1145/2737924.2737976 (cited on page 172).

[211] Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 'Static Analysis of Data Transformations in Jupyter Notebooks'. In: *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023*. Ed. by Pietro Ferrara and Liana Hadarean. ACM, 2023, pp. 8–13. DOI: 10.1145/3589250.3596145 (cited on page 172).

[212] Cynthia Dwork et al. 'Fairness through awareness'. In: *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. Ed. by Shafi Goldwasser. ACM, 2012, pp. 214–226. DOI: 10.1145/2090236.2090255 (cited on page 173).

[213] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2010 (cited on page 173).

[214] Jerome H. Friedman. 'Greedy Function Approximation: A Gradient Boosting Machine'. In: *Annals of Statistics* 29.5 (Oct. 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451 (cited on page 173).

[215] Marta Kwiatkowska. 'Advances and challenges of quantitative verification and synthesis for cyber-physical systems'. In: *2016 Science of Security for Cyber-Physical Systems Workshop, SOSCYPS@CPSWeek 2016, Vienna, Austria, April 11, 2016*. IEEE Computer Society, 2016, pp. 1–5. DOI: 10.1109/SOSCYPS.2016.7579999 (cited on page 173).

[216] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 'Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis'. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498721 (cited on page 173).

[217] Marco Campion et al. 'A Formal Framework to Measure the Incompleteness of Abstract Interpretations'. In: *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*. Ed. by Manuel V. Hermenegildo and José F. Morales. Vol. 14284. Lecture Notes in Computer Science. Springer, 2023, pp. 114–138. DOI: 10.1007/978-3-031-44245-2_7 (cited on page 173).

[218] Paul Kocher et al. 'Spectre Attacks: Exploiting Speculative Execution'. In: *meltdownattack.com* (2018), pp. 1–19. DOI: 10.1109/SP.2019.00002 (cited on page 173).

Alphabetical Index

RÉSUMÉ

L'objectif de cette thèse est de développer des méthodes mathématiquement solides et pratiquement efficaces pour améliorer la fiabilité des systèmes logiciels. Ce travail repose sur la théorie de l'Interprétation Abstraite, un cadre formel pour l'approximation des comportements des programmes. En particulier, cette thèse se concentre sur la quantification de l'impact des variables d'entrée sur l'exécution des programmes, un aspect crucial pour garantir la correction, la performance et la sécurité des systèmes logiciels.

Pour ce faire, nous présentons un nouveau cadre quantitatif d'utilisation des entrées permettant de discriminer les variables d'entrée en fonction de leur impact sur le programme. Ce cadre permet d'identifier les variables qui affectent de manière disproportionnée le système et peut être utilisé pour certifier le comportement attendu ou révéler des défauts potentiels. La notion d'impact est paramétrique au cadre, offrant ainsi une flexibilité d'adaptation à différents contextes et exigences.

En particulier, nous explorons l'application de ce cadre quantitatif pour la vérification des propriétés intentionnelles et extensionnelles. Les propriétés extensionnelles concernent le comportement input-output d'un programme, tandis que les propriétés intentionnelles englobent également les états internes.

Les résultats présentés dans cette thèse ont été implémentés dans trois outils : Libra, Impatto et TimeSec. Les résultats expérimentaux montrent la quantification de l'impact dans divers scénarios, y compris l'évaluation de l'équité pour les réseaux de neurones et la détection des vulnérabilités liées aux canaux auxiliaires.

MOTS CLÉS

interprétation abstraite ⋆ analyse statique ⋆ méthodes formelles ⋆ vérification quantitative ⋆ réseaux de neurones ⋆ propriétés de sécurité

ABSTRACT

The aim of this thesis is to develop mathematically sound and practically efficient methods for improving the reliability of software systems. This work is grounded in the theory of Abstract Interpretation, a formal framework for approximating program behaviors. In particular, this thesis focuses on quantifying the impact of input variables on program execution, a critical aspect for ensuring correctness, performance, and security of software systems.

To achieve this, we present a novel quantitative input usage framework to discriminate between input variables based on their impact on the program. This framework allows the identification of variables that disproportionately affect the system, and can be used to certify intended behavior or reveal potential flaws. The notion of impact is parametric to the framework, providing flexibility to adapt to different contexts and requirements.

In particular, we explore the application of this quantitative framework for verifying both intensional and extensional properties. Extensional properties refer to the input-output behavior of a program, while intensional properties also encompass the internal states.

The results presented in this thesis have been implemented into three tools: Libra, Impatto, and TimeSec. Experimental results show the quantification of impact in a variety of scenarios, including the evaluation of fairness for neural networks and the detection of side-channel vulnerabilities.

KEYWORDS

abstract interpretation ⋆ static analysis ⋆ formal methods ⋆ quantitative verification ⋆ neural networks ⋆ security properties