Università Degli Studi di Verona

APPROXIMATIONS IN LEARNING & PROGRAM ANALYSIS

a dissertation
submitted to the department of Computer Science
and the committee on graduate studies
of University of Verona
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Vivek Notani

February 2020

Certificate

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Internal Committee:

(Roberto Giacobazzi) Principal Adviser
(University of Verona, Italy)

(Mila Dalla Preda)
((University of Verona, Italy)

(Alessandro Farinelli)
(University of Verona, Italy)

External Reviewers:

(Sébastien Bardin)
(Commissariat à l'Energie Atomique, France)

(Francesco Ranzato)
(University of Padova, Italy)

Defense Committee:

(Roberto Giacobazzi) Principal Adviser
(University of Verona, Italy)

(Roberta Gori)
(University of Pisa, Italy)

(Roberto Bruni)
(University of Pisa, Italy)

"God sometimes plays dice with whole numbers"

Gregory Chaitin

Preface

In this work we compare and contrast the approximations made in the problems of Data Compression, Program Analysis and Supervised Machine Learning.

Gödel's Incompleteness Theorem mandates that any formal system rich enough to include integers will have unprovable truths. Thus non computable problems abound, including, but not limited to, Program Analysis, Data Compression and Machine Learning. Indeed, it can be shown that there are more non-computable functions than computable. Due to non-computability, precise solutions for these problems are not feasible, and only approximate solutions may be computed.

Presently, each of these problems of Data Compression, Machine Learning and Program Analysis is studied independently. Each problem has it's own multitude of abstractions, algorithms and notions of tradeoffs among the various parameters.

It would be interesting to have a unified framework, across disciplines, that makes explicit the abstraction specifications and ensuing tradeoffs. Such a framework would promote inter-disciplinary research and develop a unified body of knowledge to tackle non-computable problems.

As a small step to that larger goal, we propose an Information Oriented Model of Computation that allows comparing the approximations used in Data Compression, Program Analysis and Machine Learning. To the best of our knowledge, this is the first work to propose a method for systematic comparison of approximations across disciplines.

The model describes computation as set reconstruction. Non-computability is then presented as inability to perfectly reconstruct sets. In an effort to compare and contrast the approximations, select algorithms for Data Compression, Machine Learning and Program Analysis are analyzed using our model.

We were able to relate the problems of Data Compression, Machine Learning and Program Analysis as specific instances of the general problem of approximate set reconstruction. We

demonstrate the use of abstract interpreters in compression schemes.

We then compare and contrast the approximations in Program Analysis and Supervised Machine Learning. We demonstrate the use of ordered structures, fixpoint equations and least fixpoint approximation computations, all characteristic of Abstract Interpretation (Program Analysis) in Machine Learning algorithms.

We also present the idea that widening, like regression, is an inductive learner. Regression generalizes known states to a hypothesis. Widening generalizes abstract states on a iteration chain to a fixpoint. While Regression usually aims to minimize the total error (sum of false positives and false negatives), Widening aims for soundness and hence errs on the side of false positives to have zero false negatives. We use this duality to derive a generic widening operator from regression on the set of abstract states.

The results of the dissertation are the first steps towards a unified approach to approximate computation. Consequently, our preliminary results lead to a lot more interesting questions, some of which we have tried to discuss in the concluding chapter.

Acknowledgments

I would like to express the deepest appreciation to Dr. Roberto Giacobazzi, Professor at the University of Verona and my advisor, for without his guidance and persistent help this work would not have been possible.

I would like to thank my committee members Dr. Mila Dalla Preda and Dr. Alessandro Farinelli for their continual support and guidance throughout this endeavor.

I would also like to thank Dr. Damiano Zanardini, Associate Professor at Universidad Politécnica de Madrid, Spain and Dr. Francesco Ranzato, Professor at the University of Padova, Italy for all the spirited discussions that helped shape some of the ideas presented here.

I would also like to extend my sincere gratitude to Dr. Arun Lakhotia, Professor at University of Louisiana at Lafayette, USA for introducing me to the world of cybersecurity and encouraging me to pursue a Ph.D.

On a personal note, I would also like to thank all my family and friends. My parents Jaya and Raj Notani and my brother Jayant Notani for being a constant source of inspiration and encouragement. My friends Leia Kagawa for the Cajun care packages and Domenico Mastronardi for introducing me to the Italian way of life. Thanks are also due to Surbhi, Merle, Anjali, Yamini, Konika, Prateek, Joachim, Vinodh, and Venky, for their presence was very important in a process that often felt tremendously solitaire.

Contents

List of Tables

List of Figures

"If I have seen further it is by standing on the shoulders of Giants"

Isaac Newton

1 Introduction

1.1 | A Brief Historical Review

In 1931, Gödel presented his famous Incompleteness Theorems and layed to rest the question that had occupied early 20^{th} century mathematicians for decades: *What constitutes a valid proof in mathematics?* and *How is such a proof to be recognized?* [8].

David Hilbert had earlier proposed to devise an artificial language in which valid proofs can be found mechanically. He viewed mathematics as manipulation of symbols, in the sense that once the rules of inference were known, deductions may be drawn using those rules, from arbitrarily given system of postulates [14, page 2]. To that end, he even proposed the axiomatization of Physics in his seminal talk at the International Congress of Mathematics, in the year 1900 [29].

It was only 30 years later that Gödel showed that no such perfect language exists. Gödel's Incompleteness Theorem stated that not all truths are provable.

Further, Nagel *et al.* [47] showed that Gödel's Incompleteness Theorem applies to all formal systems that deal with integers. This has profound implications since it implies that all these formal systems will have unprovable truths.

This discovery shook the tenets of mathematical community. Voicing his concern, Hermann Weyl in 1946, commented that "[Gödel's Theorem] is a constant drain on the enthusiasm and determination with which I pursued my research work" [8]. The mathematicians were worried they may end up devoting entire careers trying to prove unprovable results. This then led mathematicians to question: *What is provable?*.

1.2 | What is Provable ?

In 1936, Turing showed that Gödel's Incompleteness Theorem is equivalent to the Halting Problem—assertion that there can be no general method for systematically deciding if a computer program halts [10]. Thus, from the Turing perspective, provable truths are functions

computable by a Turing Machine [65].

Indeed several mathematicians were working on the problem parallelly to come up with their own definitions for computable functions. Prominent descriptions of computable functions include lambda calculus [13], $\mu-$recursive functions, Register Machines, etc. However, it was shown that all these definitions are equivalent and correspond to the same class of functions [33, 35]. Indeed this is the famous Church-Turing Thesis [34].

This brings us to the next question: *What are some undecidable problems?* Computer scientists soon realized that it is not just the property of termination, but all non-trivial extensional code properties are in general not decidable. This is also known as Rice's Theorem [53].

Thus Program Analysis—the problem of automatically analyzing behavior of computer programs regarding safety, security and/or other semantic properties, is undecidable.
Semantic properties are code extensional properties. Since one may reduce the problem of deciding an extensional property to that of deciding termination, it follows that Program Analysis is also limited by Gödel's Incompleteness Theorems. Mathematically, this limitation shows up in the incompleteness of Hoare Logic- a popular method of Program Verification in the 20^{th} century [30].

In 1977, Cousot & Cousot presented their Abstract Interpretation [17] framework that eventually changed the view of mathematicians towards Gödel's results from dismay to excitement. To quote Roberto Giacobazzi[1], "Impossibility is opportunity [to design abstractions that allow approximate computations]". Indeed approximations abound in several disciplines including Program Analysis, Machine Learning, Data Compression, etc.

Cousot's key contribution was to provide a mathematical framework for controlled loss of information, via abstractions to soundly approximate non-computable problems. Although it has been applied almost exclusively to the problem of Program Analysis, our opinion is that it should be extensively studied in the context of design of approximations for other non-computable problems.

We discuss the problem of Program Analysis in Chapter-3, and the Abstract Interpretation framework at length in Chapter-4 and Chapter-5.

[1]Dr. Roberto Giacobazzi is my mentor and Ph.D. advisor.

1.3 | Computation as a Physical Process

Around the 1950s, another approach to understanding the implications of Gödel's Incompleteness Theorem was proposed—that of Information Theory, in the works of Shannon [60], Kolmogorov [36], Chaitin [6,7], and Solomonoff, among others.

Here computation concerns communication of information from a *source* to *target* over some communication channel. The source has some object that it wants to transmit to the target, using some *description* (of the object) over a communication channel. Gödel's Incompleteness Theorem shows up here in the form of limitations that apply on what constitutes *effective descriptions*.

Consider a scenario with two people- Alice and Bob, where Bob wants Alice to send him charts on diameter and area of circles. She can share a table with thousands of entries of $\langle \text{diameter}, \text{area} \rangle$, or, share the formula to build the table: area = $\pi \left(\frac{d}{2}\right)^2$.

Now consider another example, where Bob wanted Alice to send him Indian Cricket team's run rate averaged over all games played in a year, for every year since 1990. No formula exists to predict the Indian Cricket team's performance in a given year. Hence, Alice has no choice but to send a table with all the entries.

These examples allows us to intuitively define randomness as *incompressibility*; the idea being that random sequences do not follow any pattern, while non-random sequences do have a pattern that can be used to compress them [8].

This may be understood as the fact that every bit in a random sequence adds information, while non-random sequences have redundant bits that may be discarded without loss of information. This raises the question- *What is the least number of bits that are required to describe an object?*

Shannon supplied the answer. He viewed computation as a dynamic physical processand described how Entropy may be used as information measure. Along the lines of Thermodynamic Entropy of physical processes, Shannon defined the Information Entropy such that if Alice wants to share some objects (selected via probabilistic measure p from a domain of objects) with Bob, then Shannon Entropy describes the width (number of bits needed) of the channel to communicate.

Around 1964, Kolmogorov promoted the idea that for the case of digital communication, it makes sense to consider descriptions that are Turing computable, as effective descriptions.

This combination of Computability and Information theories led to the birth of Algorithmic Information Theory (a term coined by Chaitin). Thus, Kolmogorov Complexity may be understood as Shannon Entropy with the domain of objects restricted to Turing computable sets.

Algorithmic Information Theory forms the basis of the field of Data Compression. Namely, the problem of encoding the data (set of objects) into a compressed object that has a cheaper computer representation, yet can be used to reconstruct the original set of objects.

While Kolmogorov's initial model was for objects to have a stand alone description that describes the entire object under a universally shared domain of descriptions; Chaitin, on the other hand, opened up this model for further generalization with his two-part description model. The two part code (description) views Kolmogorov descriptions as a program and an interpreter (the Turing Machine), that will execute the program to reconstruct the object. This opens the door to experiment with Abstract Interpreters in Compression Schemes.

1.4 | Our Approach

The Turing Model of Computation views computations as proofs and non-computability as impossibility of proving a truth. The Information Theory Model views computation as an exchange of information between a source and a digital target, and non-computability as the impossibility of exchanging certain information.

We aim to bridge together the ideas of Information Theory model of computation that makes explicit measures on the amount of information, with that of Abstract Interpretation framework that allows for controlled loss of information to allow for a generalized framework that permits explicit measured and controlled loss of information when approximating non-computable problems.

We present a small step in this work to serve that large goal. We start with Chaitin's Two-Part Code model as a low hanging fruit to bring Abstract Interpretation into the problem of Compression Schemes. Next, since Learnability and Compressibility have been previously studied together with interesting results [39], we study the problem of Supervised Machine Learning and Program Analysis through the context of Information Theory and Abstract Interpretation.

We use Abstract Interpretation to answer the questions: *What possible abstractions exist to attack a specific problem? Is there a standardized approach to choosing the right abstraction for the given*

problem? What are the ensuing tradeoffs? How does one reason about the ensuing tradeoffs?

Simultaneously, we use Information Theory to answer questions such as: *Is the proof problem feasible?*, or, *Is the amount of information in the assertions sufficient to arrive at the implication? If not, how much more information is required via assumptions?*

Presently, each of the above mentioned disciplines- Data Compression, Program Analysis and Supervised Machine Learning, have their own intuitions, techniques, and formalizations to answer these questions. We want a more generalized framework that allows to instantiate these problems, thereby promoting cross-disciplinary use of ideas, tools and techniques, whenever possible.

Work towards studying and comparing abstractions across disciplines has been disparate. Since the 1980s, Machine Learning and Data Compression have been studied together leading to some results relating Learnability Theory with Compressibility Theory [39]. Recently, it was shown that the SAT Solvers may be viewed as Abstract Interpreters [19]. However, we are not aware of any work comparing the approximations in Supervised Machine Learning with those in Program Analysis. To the best of our knowledge, there are no works aiming to design a method for systematic comparison of approximations across disciplines.

1.5 | Outline

In this work, we propose an *Information Oriented Model of Computation* that allows for a systematic comparison of the approximations made in various problems in computer science. This is the first work to propose such a generalized framework that bridges Computability Theory, Information Theory and Abstract Interpretation to view computation from an Information Oriented perspective.

We then use our model to study and compare the approximations in Supervised Machine Learning, Program Analysis and Data Compression problems.

Other than this Introduction, the dissertation has 8 more chapters. Chapters-2 through 6 describe the motivations and background, while Chapters-7 and 8 present our major contributions, and finally Chapter-9 is the conclusion. We list here briefly the objectives and contributions in each chapter.

The following chapter, Chapter-2, discusses the mathematical notations and necessary mathematical background required for understanding the rest of the dissertation. Specifically, we discuss Order Theory and Probability Theory.

We begin the main body of the dissertation with the question: *What is Program Analysis?* Chapter-3 presents an introduction to the problem, discusses the challenges, and presents a systematic method to organize the techniques used for Program Analysis.

We then follow up with the next logical question: *How do you analyze programs?* Chapter-4 presents the Abstract Interpretation framework for Program Analysis. Chapter-5 describes the most common abstract domains for numerical invariants. We will refer to some of these domains for examples throughout the dissertation.

Another approach to Program Analysis is that via Machine Learning techniques. Chapter-6 discusses the fundamentals of Machine Learning Theory, specifically the Supervised Machine Learning, Regression, and the PAC learning model.

We then ask the question: *How can we compare the approximations made by Program Analyzers and Machine Learners?* Chapter-7 addresses this question via our key contribution- an *Information Oriented Model of Computation* that uses ideas from Information Theory and Abstract Interpretation to allow examination of approximations made by an algorithm. We then analyze the problems of Data Compression, Program Analysis and Supervised Machine Learning within our model and compare and contrast the available input information and approximations made in each problem.

In chapter-8, we further build upon the relationship between Program Analysis and Machine Learning. We first examine a Machine Learning problem through the lens of Abstract Interpretation and demonstrate the similarities. We then examine Widening, an Abstract Interpretation construct, through the lens of Machine Learning, and once again, demonstrate the relationship between the two problems of Program Analysis and Machine Learning.

Finally, we end the dissertation in Chapter-9 with a concluding discussion on our Information Oriented Model of Computation, and avenues for future work on relating Compressibility, Obfuscation and Learnability theories.

1.6 | Contributions

We present here the original contributions of this work in the form of questions examined, solutions proposed and insights gained:

Question: What techniques are used for Program Analysis?

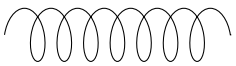
Answer: Since Program Analysis is undecidable in general, numerous tools and techniques exist for Program Analysis, each best suited for a specific (set of) application(s). The suitability of a

specific method to a specific application is determined by the tradeoffs made by the technique and the requirements of the application. In Section-3.4, we propose a novel technique to systematically study these techniques by way of arranging them on a cube such that the position in the cube is determined by the tradeoffs made. This led to an insight: *Abstract Interpretation and Machine Learning are the two base techniques used for Program Analysis, and that other Program Analyzers may be derived as some combination of abstract interpreters and learners.*

Question: How can we compare the approximations made by Program Analyzers and Machine Learners?
Answer: We investigate this in Chapter-7. We begin by proposing an *Information Oriented Model of Computation* that bridges Computability Theory, Information Theory and Abstract Interpretation into one framework that allows analysis of computation as a communication of information. Information Theory permits measurement and comparison of information content in inputs & expected output and classify problems into decidable, partially decidable and probabilistic estimations.
We then use the framework to compare and contrast approximations in Data Compression, Supervised Machine Learning and Program Analysis and demonstrate how all three techniques solve the problem of set reconstruction, albeit with differing *input quality* and *output quality*.
Further, this is the first work to relate Compressibility, Learnability and Analyzability. Though it should be noted that Compressibility and Learnability have been studied together previously beginning with [39]; we are the first to introduce Program Analysis into the mix.

Question: Do Abstract Interpretation and Machine Learning take fundamentally different approaches to solving related problems, or can we demonstrate similarity in their approach to solving these related problems?
Answer: We investigate this in chapter-8. We begin by examining Supervised Machine Learning through the lens of Abstract Interpretation. We demonstrate the presence of ordered structures in Machine Learning hypothesis classes (abstract domains), the equivalence of ideal solution to non-computable least fixpoint solution and computation of least fixpoint approximations by Learning Algorithms.
We then examine Abstract Interpretation through the lens of Supervised Machine Learning. We demonstrate that the problem addressed by widening in Abstract Interpretation Framework: least fixpoint approximation in finite steps, is fundamentally a learning theory problem.

We present a novel idea that widening, like regression, is an inductive learner. While regression generalizes known states to a hypothesis, widening generalizes abstract states on a iteration chain to a fix point. We use this to design a generic learning theory based widening algorithm.



"My lattice-theoretic arguments seemed to me so much more beautiful, and to bring out so much more vividly the essence of the considerations involved, that they were obviously the 'right' proofs to use"

Garrett Birkhoff

2 Basics

This chapter describes the mathematical notations and discusses the basic mathematical theories that lay the foundation for the work presented in the rest of the dissertation.

We describe the notations first and then discuss Order Theory and, finally, Probability Theory. Order Theory is fundamental to Abstract Interpretation way of describing abstractions as we will see in Chapter-4. Probability Theory lays the foundation for Machine Learning Theory discussed in Chapter-6.

2.1 | Mathematical Notations

We introduce notations that will be used in the rest of dissertation.

2.1.1 — Logic notations

We use standard notations from first-order logic: disjunction $\vee$, conjunction $\wedge$, logical negation $\neg$, implication $\implies$, equivalence $\iff$, as well as universal $\forall$ and existential $\exists$ quantifiers. For definitions, we use $\stackrel{\text{def}}{=}$ and $\stackrel{\text{def}}{\iff}$. We use the notation $a := b$ to denote the assignment operation where variable a is *set* to be equal to the value of variable b or evaluation of the expression b.

2.1.2 — Set Notations

We also use standard notations from set theory: the empty set $\varnothing$, set union $\cup$ and intersection $\cap$: in binary form $(S \cup T)$ and $(S \cap T)$, or over a family $\bigcup_{i \in I} S_i$ and $\bigcap_{i \in I} S_i$, for a family $(S_i)_{i \in I}$ of sets indexed by I, which may be finite or infinite.

Given two sets S and T, we denote the cartesian product of S and T with $S \times T$, set inclusion $S \subseteq T$, strict inclusion $S \subset T$, ownership $S \in T$, set difference $S \setminus T$, set complement $\overline{S}$, and cardinality (the number of elements in S) by $|S|$.

Set comprehension: $\{s \in S | P(s)\}$ is the subset of elements from S that additionally satisfy some logic predicate P. Given a set S, we denote as $\wp(S)$ its powerset, i.e., the set of all the subsets of S, including $\varnothing$ and S itself. We denote by $[S]^{fin}$ the set of all finite subsets of S. Additionally, $[S]^n$ for $n \in \mathbb{N}$ represents the set of subsets of S with cardinality n.

Given two sets S and T, we denote as $S \longrightarrow T$ the set of functions from S to T; S is called the domain of such a function, and T as its codomain.We will sometimes use the lambda notation $\lambda x.f(x)$ to denote functions concisely. Alternatively, a function can be described in extension as $[x_1 \mapsto v_1, ..., x_n \mapsto v_n]$, meaning that its domain is $x_1, ..., x_n$ and it maps any x_i to the corresponding v_i .

Given a function f , $f[x \mapsto v]$ denotes the function that maps x to v, and any y such that $x \neq y$ to $f(y)$, i.e., it updates the function at point x to value v.

Given $f : S \longrightarrow T$ and $g : T \longrightarrow Q$ we denote with $g \circ f : S \longrightarrow Q$ their composition, i.e., $g \circ f = \lambda x.g(f(x))$. Finally, f^i denotes f composed i times, and id denotes the identity function: $f^0 \stackrel{\text{def}}{=} \text{id}$ and $f^{i+1} = f^i \circ f = f \circ f^i$.

We denote respectively as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{R}_+$ the sets of natural integers, integers, rationals, reals and non-negative reals.

2.1.3 — Matrices and Vector Notations

Vectors are denoted as $\vec{v}$, matrices as $\mathbf{M}$, matrix-matrix and matrix-vector multiplications as $\times$, and the dot product of vectors is denoted as $\cdot$ (a dot).

The i-th component of a vector $\vec{v}$ is denoted as v_i . Moreover, the i-th line or i-th column of a matrix M, depending on the context, is a vector denoted as $\vec{M}_i$. The element at line i and column j of a matrix M is denoted as M_{ij} .

We order vectors element-wise: $\vec{v} \geq \vec{w}$ means $\forall i : v_i \geq w_i$. The zero vector is denoted as $\vec{0}$.

2.1.4 — Context Specific Notations

Sometimes, in special context, we will use popular notations in common use for the specific problems in a obvious way, or with explicit warning, as needed, to make the text easier to comprehend.

For instance, in the context of Machine Learning, the set elements may be indexed, with index denoted by superscript. For instance, we use $x^{(i)}$ to denote the "input" variables, also

called input features, and $y^{(i)}$ to denote the "output" variable, also called target variable, that we are trying to predict. A pair $(x^{(i)}, y^{(i)})$ is called a training example, and the dataset that we use to learn:—a list of m training examples $\{(x^{(i)}, y^{(i)}) \mid i = 1, ..., m\}$ -is called a training set. The superscript "(i)" in the notation is simply an index into the training set. We will also use X to denote the set of input values, and Y the set of output values. In this context, often $\vec{x^{(i)}}$'s and/or $\vec{y^{(i)}}$'s maybe multidimensional vectors.

For the sake of brevity, and when the context makes this obvious, we will skip the vector notation and refer to multidimensional vectors $\vec{x^{(i)}}$'s and/or $\vec{y^{(i)}}$ as simply $x^{(i)}$'s and/or $y^{(i)}$.

Further, when dealing with n features ($n \in \mathbb{N} \wedge n \geq 2$), the j^{th} element of the multi-dimensional vector $x^{(i)}$ may be referenced as $x_j^{(i)}$ for $j \in [1, n]$. Similarly, we may use X_j to refer to the set $\{x_j^{(i)} \mid i = 1, ..., m\}$.

2.2 | Order Theory

Partial orders and lattices are the fundamental algebraic structures used to describe the various abstractions in Abstract Interpretation. We describe here Order Theory sufficient for understanding and reading of this work. The definitions are standard material on the subject and the reader is referred to [48] for further details on Order Theory and Lattices. The following definitions were taken from a tutorial on basic Abstract Interpretation by Antoine Mine [45].

2.2.1 — Data Structures

Definition 2.2.1 (Partial Order, Poset)**.** A partial order $\sqsubseteq$ on a set X is a relation $\sqsubseteq \ \in X \times X$ that is:

1. reflexive: $\forall x \in X$: $x \sqsubseteq x$
2. anti-symmetric: $\forall x, y \in X$: $(x \sqsubseteq y) \wedge (y \sqsubseteq x) \implies x = y$
3. transitive: $\forall x, y, z \in X$: $(x \sqsubseteq y) \wedge (y \sqsubseteq z) \implies x \sqsubseteq z$.

The $\sqsubseteq$ describes the ordering applied to the set X. The poset is called a total order iff the ordering is defined for every pair of elements in the set: $\forall x, y \in X$, either $x \sqsubseteq y$ or $y \sqsubseteq x$.

If there exist pairs of elements in X which are not comparable (ordering is not defined), then the poset is strictly a partially ordered set, and not a totally ordered set.

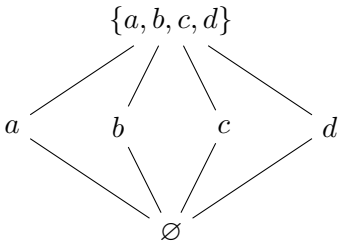
Lower and Upper Bounds

Let P be an ordered set and $S \subseteq P$. We say $x \in P$ is an upper bound of S iff $x \sqsubseteq s \ \forall s \in S$. An upper bound x need not belong in S. We say that x is the least upper bound for S if x is an upper bound for S and $x \sqsubseteq y$ for every upper bound y of S. If the least upper bound of S exists, then it is unique. Least upper bound, lub or join of two elements a and b is written as $a \sqcup b$. Likewise, the unique greatest lower bound of a and b, also called glb or meet, if it exists, is the greatest element smaller than a and b, and it is denoted as $a \sqcap b$.

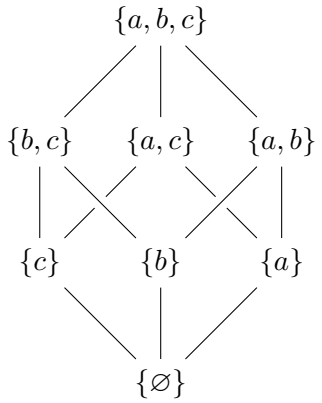
As $\sqcup$ and $\sqcap$ are associative operators, we will employ also the notations $\sqcup S$ and $\sqcap S$ to compute lubs and glbs on arbitrary (possibly infinite) sets S of elements. Finally, we denote respectively as $\bot$ and $\top$ the least element and the greatest element in the poset, if they exist.

Hasse Diagrams

We say that x *is covered by* y *in poset* P, written $x \prec y$, if $x \sqsubseteq y$ and there is no $z \in P$ such that $x \sqsubseteq z \sqsubseteq y$. It is clear that the covering relation determines the partial order in a finite ordered set P. A *Hasse diagram* of a poset P has the elements of P represented by points on a plane, with greater elements higher, and, a line is drawn from point x upto y when $x \prec y$. Figure-2.1 shows example Hasse Diagrams.



(a) Example Hasse Diagram of a Lattice



(b) Example Hasse Diagram of a Complete Lattice

Figure 2.1: Hasse Diagrams

Definition 2.2.2 (Chain)**.** A chain $(C, \sqsubseteq)$ in a poset $(X, \sqsubseteq)$ is a subset $C \subseteq X$ such that $\forall x, y \in C, \ (x \sqsubseteq y) \vee (y \sqsubseteq x)$

Definition 2.2.3 (CPO)**.** A CPO or Complete Partial Order is a poset such that every chain has a least upper bound.

Theorem 2.2.4 (Zorn's Lemma)**.** *If every chain in an ordered set X has an upper bound in X, then X contains a maximal element.*

Definition 2.2.5 (Lattice)**.** A *Lattice* $L(X, \sqsubseteq, \sqcup, \sqcap)$ is a poset such that $\forall x, y \in X, x \sqcup y$ and $x \sqcap y$ exist.

Definition 2.2.6 (Complete Lattice)**.** A *Complete Lattice* $L(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a poset such that:

1. $\forall A \subseteq X, \sqcup A$ exists.
2. $\forall A \subseteq X, \sqcap A$ exists.
3. X has a least element $\bot$.
4. X has a greatest element $\top$.

Clearly, a complete lattice is a lattice and a CPO. Less obviously, to get a complete lattice, either condition 1 or 2 is sufficient, as each one implies the other one, and both imply conditions 3-4. The proof involves Zorn's Lemma as described in theorem-2.2.4.

Definition 2.2.7 (Distributive Lattice)**.** A Lattice $L(X, \sqsubseteq, \sqcup, \sqcap)$ that satisfies equivalent definitions of the distributive law:

$$\forall x, y, z \in X, (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z) \tag{2.1}$$

$$\forall x, y, z \in X, (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \tag{2.2}$$

Definition 2.2.8 (SubLattice)**.** A Lattice $L'(X', \sqsubseteq', \sqcup', \sqcap')$ is a sublattice of $L(X, \sqsubseteq, \sqcup', \sqcap')$ if:

1. $X' \subseteq X$
2. we use the same order $\sqsubseteq$ on both X and X', and
3. X' is closed under $\sqcup$ and $\sqcap$, i.e., lubs and glbs exist in X' and coincide with those in X: $\sqcup X = \sqcup X'$ and $\sqcap X = \sqcap X'$

2.2.2 — Operators and Fixpoints

An operator is a function $f : X \longrightarrow X$ with the same domain and codomain.

Definition 2.2.9 (Fixpoints, Prefixpoints, Postfixpoints)**.** Given a Poset $(X, \sqsubseteq)$ and an operator $f : X \longrightarrow X$:

1. x is a fixpoint of f if $f(x) = x$.
   We denote as $\text{fp}\, f \stackrel{\text{def}}{=} \{x \in X \mid f(x) = x\}$ the set of fixpoints of f.
2. x is a prefixpoint of f if $x \sqsubseteq f(x)$.
3. x is a postfixpoint of f if $f(x) \sqsubseteq x$.
4. $\text{lfp}_x\, f \stackrel{\text{def}}{=} \min\{y \in \text{fp}\, f \mid x \sqsubseteq y\}$, if it exists, is the least fixpoint of f greater than x.
   $\text{lfp}\, f \stackrel{\text{def}}{=} \text{lfp}_\bot\, f$, if it exists, is the least fixpoint of f.
5. Dually, $\text{gfp}_x\, f \stackrel{\text{def}}{=} \max\{y \in \text{fp}\, f \mid y \sqsubseteq x\}$ is the greatest fixpoint of f smaller than x.
   $\text{gfp}\, f \stackrel{\text{def}}{=} \text{gfp}_\top\, f$, if it exists, is the greatest fixpoint of f.

Definition 2.2.10 (Operator Properties)**.**

1. Monotonicity: A function mapping two posets $f : (A_1, \sqsubseteq_1) \longrightarrow (A_2, \sqsubseteq_2)$ is *monotone* if $\forall x, y \in A_1 : x \sqsubseteq_1 y \Longrightarrow f(x) \sqsubseteq_2 f(y)$.
2. Continuity: A function between two CPO $f : (A_1, \sqsubseteq_1, \sqcup_1) \longrightarrow (A_2, \sqsubseteq_2, \sqcup_2)$ is continuous if for every chain $C \subseteq A_1, \{f(c) \mid c \in C\}$ is also a chain and the limits coincide: $f(\sqsubseteq_1 C) = \sqsubseteq_2 \{f(c) \mid c \in C\}$.
3. Extensivity: An operator $f : (A, \sqsubseteq) \longrightarrow (A, \sqsubseteq)$ on a poset is extensive if $\forall a \in A : a \sqsubseteq f(a)$.
4. Reductivity: An operator $f : (A, \sqsubseteq) \longrightarrow (A, \sqsubseteq)$ on a poset is reductive if $\forall a \in A : f(a) \sqsubseteq a$.

Theorem 2.2.11 (Tarski's Theorem)**.** *If $f \in X \longrightarrow X$ is a monotonic operator in the complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, then the set of fixpoints $\text{fp}\, f$ is a non-empty complete lattice. In particular, $\text{lfp}\, f$ exists. Further, $\text{lfp}\, f = \sqcap\{x \in X \mid f(x) \sqsubseteq x\}$.*

Theorem 2.2.12 (Kleene's Theorem)**.** *If $f \in X \longrightarrow X$ is a continuous operator in a CPO $(X, \sqsubseteq, \sqcup, \top)$, then $\text{lfp}\, f$ exists. Further, $\text{lfp}\, f = \sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$.*

2.2.3 — Galois Connections and Insertions

Definition 2.2.13 (Upper Closure Operator)**.** An operator $f : X \longrightarrow X$ on a poset $(X, \sqsubseteq)$ is called an Upper Closure Operator (uco) iff f is:

- extensive: $\forall x \in X : x \sqsubseteq f(x)$
- monotonic: $\forall a, b \in X : a \sqsubseteq b \Longrightarrow f(a) \sqsubseteq f(b)$
- idempotent: $\forall x \in X : f(f(x)) = f(x)$

We denote with $\mathrm{uco}(X)$ the set of all closure operators on the poset X. If L is a complete lattice, then $\langle \mathrm{uco}(L), \sqsubseteq, \sqcup, \sqcap, \mathbb{T}, \mathrm{id} \rangle$ forms a complete lattice [68] where the bottom is $\mathrm{id} = \lambda x.x$, the top is $\mathbb{T} = \lambda x.\ \top_L$, and for every $\rho, \eta \in \mathrm{uco}(L)$, ρ is *more concrete than* η iff $\rho \sqsubseteq \eta$ iff $\forall y \in L.\ \rho(y) \leq \eta(y)$ iff $\eta(L) \subseteq \rho(L)$, $(\sqcap_{i \in I} \rho_i)(x) = \wedge_{i \in I} \rho_i(x)$; $(\sqcup_{i \in I} \rho_i)(x) = x$ iff $\forall i \in I.\ \rho_i(x) = x$.

Definition 2.2.14 (Galois Connection)**.** Given two posets $(C, \sqsubseteq_1)$ and $(A, \sqsubseteq_2)$, the pair $(\alpha : C \longrightarrow A, \gamma : A \longrightarrow C)$ is a Galois connection iff:

$$\forall a \in A, c \in C : c \sqsubseteq_1 \gamma(a) \Longrightarrow \alpha(c) \sqsubseteq_2 a \tag{2.3}$$

which is denoted as $(C, \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq_2)$. α and γ are said to be *adjoint functions*, where α is the upper adjoint and γ is the lower adjoint.

Definition 2.2.15 (Galois Connection Properties)**.** A Galois connection $(C, \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq_2)$ satisfies the following properties:

1. $\alpha \circ \gamma$ and $\gamma \circ \alpha$ are idempotent
2. $\gamma \circ \alpha \circ \gamma = \gamma$ and $\alpha \circ \gamma \circ \alpha = \alpha$
3. $\forall c \in C : \alpha(c) = \sqcup_2 \{a \mid c \sqsubseteq_1 \gamma(a)\}$
4. $\forall a \in A : \gamma(a) = \sqcup_1 \{c \mid \alpha(c) \sqsubseteq_2 a\}$
5. α maps corresponding lubs: $\forall X \sqsubseteq_1 C$: if $\sqcup_1 X$ exists, then $\alpha(\sqcup_1 X) = \sqcup_2 \{\alpha(x) \mid x \in X\}$
6. γ maps corresponding glbs: $\forall X \sqsubseteq_1 A$: if $\sqcap_2 X$ exists, then $\gamma(\sqcap_2 X) = \sqcap_1 \{\gamma(x) \mid x \in X\}$

Definition 2.2.16 (Galois Insertion)**.** A Galois Insertion is a galois connection with upper adjoint α and lower γ is called a galois insertion iff $\alpha \circ \gamma = \mathrm{id}$.

Definition 2.2.17 (Galois Isomorphism)**.** A Galois Insertion is a galois connection with upper adjoint α and lower γ is called a galois insertion iff $\gamma \circ \alpha = \mathrm{id}$. Since both $\alpha \circ \gamma$ and $\gamma \circ \alpha$ are identity functions, it implies that α and γ are bijective.

2.3 | Probability Theory

Probability is a quantification of the degree of uncertainty. In this section we briefly describe the terminology used, and discuss the fundamental axioms and principles of Probability Theory that are required for understanding the text of the dissertation.

The definitions and examples are borrowed from [38] and [22]. However, the notation used might be slightly different from either sources. Interested readers may refer to [22] for quick introduction to probability.

2.3.1 — Origins of Probability

Modern probability theory emerged from the letters of correspondence between Fermat and Pascal in 1654, after the gambler Antoine Gombaud, Chevalier de Méré, reached out to Pascal regarding a chance game- a dice game requiring a "double-six" in 24 throws [1].

Emerson's notes [67] raise the question that although chance games have been around a millennia, why was probability not studied by western mathematicians and philosophers until the 17^{th} century?

Emerson [67] postulates that one reason is that since the western world viewed science and nature as deterministic, and chance as "divine intervention", it made the study of probability impious.

Another view, suggested by Hacking [26], is that western mathematics lacked the several foundational aspects needed to build probability theory, such as the lack of a simple numerical notation system in Greek mathematics.

Emerson, citing Hacking, notes that the Indian culture had a "science of dicing" as early as $400AD$ since the Indian culture had developed many mathematical aspects much before the European culture [67]. For instance, the ancient Indian text of Yajurveda already used a place value system, and gives names for numbers upto 10^{12}, which was not possible in the Greek and Roman numerals [52]. Emerson, citing Hacking, then notes that it is not surprising that Italians

were the first probabilists in Europe who worked with Arabic numerals and mathematical concepts [67].

The work of Raju [52] discusses a history of probability in Indian mathematics and points out how several foundational probability concepts were well understood by Indians much earlier, with a notion of fairness in dice game being referenced in the earliest of Hindu texts- Mahabharata, to discussion of weighted averages in the context similar to *dutch bets* in *Ganita Sara Samgraha-* 8th century text by Mahavira.

Raju [52] presents a fascinating discussion on how fundamental differences in Philosophy between the East and West influenced their respective development of Mathematics and ultimately Science.

2.3.2 — Definitions and Axioms

The calculus of Probability studies mathematical models of uncertain situations. Given an uncertain situation, namely, the *outcome* is not deterministic, we call the set of all possible outcomes as the *sample space* $\mathbb{S}$. We then describe an event a as a subset of $\mathbb{S}$.

Example 2.3.1. The tossing of two coins, gives a sample space $\mathbb{S}$ consisting of all pairs (i, j) where i is the outcome of the first coin, and j is the outcome of the second coin. If $a = \{(H,T),(H,H),(T,H)\}$, then a is the event that at least one coin gives H *heads*. If $b = \{(H,T),(T,T),(T,H)\}$, then b is the event that atleast one coins gives T *tails*.

Intuitively, the probability of an event a is the apparent limit of the relative frequency of outcomes in a in the long run in a sequence of independent repetitions of the experiment.

Note that not all $a \subseteq \mathbb{S}$ are events. If $\mathcal{A}$ is the set of all events, then $\mathcal{A}$ is a σ-algebra on $\mathbb{S}$.

Definition 2.3.2 (σ-algebra and Measurable Space)**.** A σ-algebra on a set $\mathbb{S}$ is a set of subsets of $\mathbb{S}$ $\left(\mathcal{A} \subseteq \wp(\mathbb{S})\right)$ such that $\mathcal{A}$ contains the set $\mathbb{S}$, is closed under complement and countable unions. Thus:

- $\varnothing \in \mathcal{A}$
- $a \in \mathcal{A} \implies \overline{a} \in \mathcal{A}$
- $a_1, a_2, \ldots \in \mathcal{A} \implies \bigcup_{i=1}^{\infty} a_1 \in \mathcal{A}$

The pair $(\mathbb{S}, \mathcal{A})$ is called a *Measurable Space* or *Borel Space*.

Definition 2.3.3 (Properties of Probability)**.** The probability $\mathbb{P}(a)$ for an event $a \in \mathcal{A}$ satisfies:

- $\forall a \in \mathcal{A} : \mathbb{P}(a) \geq 0$

- $\mathbb{P}(\mathbb{S}) = 1$ is the *certain* event
- $\mathbb{P}(\varnothing) = 0$ is the *impossible* event
- If a and b are disjoint sets in $\mathcal{A}$, then $\mathbb{P}(a \cup b) = \mathbb{P}(a) + \mathbb{P}(b)$. This may be extended to infinite events: if $a_1, a_2, ...$ are pairwise disjoint, then $\mathbb{P}(\bigcup_{i=1}^{\infty} a_i) = \sum_{i=1}^{\infty} \mathbb{P}(a_i)$
- For a decreasing sequence $a_1 \supset a_2 \supset a_3 ...$ of events with $\bigcap_n a_n = \varnothing$, we have $\lim_{n\to\infty} \mathbb{P}(a_n) = 0$

For systems with finite events, the last axiom clearly follows from the preceding axioms. However, for infinite events, the last axiom is independent of the preceding events.

Also note that given a set $\mathbb{S}$, $\mathcal{A} = \{\varnothing, \mathbb{S}\}$ is the smallest σ-algebra, and $\mathcal{A} = \wp(\mathbb{S})$ is the biggest σ-algebra on set $\mathbb{S}$.

We call $\mathbb{P}$ a probability distribution and say that the distribution $\mathbb{P}$, over the set of events $\mathcal{A}$, associates the *measure* $\mathbb{P}(a)$ with a. It has the following properties:

- $\forall a \in \mathcal{A} : 0 \leq \mathbb{P}(a) \leq 1$
- For any $a, b \in \mathcal{A} : a \subseteq b \implies \mathbb{P}(a) \leq \mathbb{P}(b)$
- if $\overline{a}$ is the complement set $\mathbb{S} - a$, then $\mathbb{P}(\overline{a}) = 1 - \mathbb{P}(a)$
- $\mathbb{P}(a \cup b) = \mathbb{P}(a) + \mathbb{P}(b) - \mathbb{P}(a \cap b)$.

Given two events $a, b \in \mathcal{A}$, we say a and b are *mutually independent events* iff $\mathbb{P}(a \cap b) = \mathbb{P}(a) \times \mathbb{P}(b)$. They are dependent events otherwise.

The idea of independence is that knowing that an event a has happened provides no additional information about the occurrence of event b.

For dependent events a and b, we define the conditional probability by Bayes' Rule:

$$\mathbb{P}(b|a) = \frac{\mathbb{P}(a \cap b)}{\mathbb{P}(a)} \tag{2.4}$$

The idea here is that while $\mathbb{P}(b)$ is the *prior probability* of event b, but after knowing that event a happened, we can adjust the probability of event b from $\mathbb{P}(b)$ to $\mathbb{P}(b|a)$, the *posterior probability*.

2.3.3 — Random Variables and Probability Mass Functions

A *Random Variable* is a number whose value depends upon the outcome of a random experiment. Mathematically though, a random variable is a real-valued function on the sample space $\mathbb{S}$. They are usually denoted as X, Y, Z.

Example 2.3.4 (Random Variable)**.** Some examples of Random Variables:

1. Toss a coin 10 times and let X be the number of Heads.
2. Choose a random person in a class and let X be the weight of the person, in Kg.
3. Let X be value of the MSFT stock price at the closing of the next business day.

A discrete random variable X has finitely or countably many values $x_i, i = 1, 2, \ldots$, and $p(x_i) = \mathbb{P}(X = x_i)$ with $i = 1, 2, \ldots$ is called the probability mass function of X. Sometimes X is added as the subscript of its p. m. f., $p = p_X$.

Properties of Probability Mass Function:

- For all $i, p(x_i) > 0$ (usually, we do not list values of X which occur with probability 0)
- For any interval B, $\mathbb{P}(X \in B) = \sum_{x_i \in B} p(x_i)$
- As X must have some value, $\sum_i p(x_i) = 1$

Assume that X is a discrete random variable with possible values $x_i, i = 1, 2, \ldots$, then, the expected value, also called *expectation* of X is: $\mathbb{E}(X) = \sum_i x_i p(x_i)$.

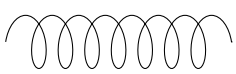
For any function For any function, $g : \mathbb{R} \longrightarrow \mathbb{R}$: $\mathbb{E}_g(X) = \sum_i g(x_i)p(x_i)$.

Definition 2.3.5 (Joint Probability and Marginal Probability)**.** : Given two random variables X, Y, the joint probability mass function $p(x, y)$ is defined as: $p(x, y) = \mathbb{P}(X = x, Y = y)$, so that:

$$\mathbb{P}\big((X, Y) \in a\big) = \sum_{(x,y)\in a} p(x, y)$$

The marginal probability of X and Y are defined as:

$$\mathbb{P}(X = x) = \sum_y \mathbb{P}(X = x, Y = y) = \sum_y p(x, y)$$
$$\mathbb{P}(Y = y) = \sum_x \mathbb{P}(X = x, Y = y) = \sum_x p(x, y)$$



"[…] it is not only the programmer's responsibility to produce a correct program but also to demonstrate its correctness in a convincing manner …"

Edsger Wybe Dijkstra

3 Program Analysis

3.1 | The Problem

Informally, Program Analysis is the problem of automatically reasoning about the behavior of computer programs. A better definition would be as the problem of automatically synthesizing program invariants. Program Invariants are assertions on program properties that hold true on all possible program executions (semantics). Thus, Program Analysis is the problem of automatically generating true assertions about program semantics.

Program Analysis is one of the methods available to solve the problem of Program Verification. Program Verification is the problem of proving that program execution is *correct* (with respect to some program specification) in all specified environments.

What do we mean by that? Consider the set of natural numbers $\mathbb{N}$. Natural numbers have numerous interesting properties- they may be even, odd, prime, or, divisible by 20, and so on. Let's say we want to *verify* that a given set of numbers $X = \{2, 4, 6, 8\}$ does not contain odd numbers. One way to verify the assertion that X *does not contain odd numbers* would be to check if each element $x \in X$ satisfies the assertion. Thus, we pick each element and check if it is odd, and raise a flag if the assertion is violated. Once we have exhausted all elements without raising the flag, we know the assertion holds true, and we have successfully *verified* the assertion. This approach could be expensive if the set X is very large.

Another alternative way would be to use the set of all odd numbers $O = \{1, 3, 5, ...\}$ and see if $X \cap O = \varnothing$. If the set intersection is empty, means X does not contain any odd numbers, and we have once again verified the assertion.

A key idea in the second approach was to use the set of all odd numbers O as a representation for the property of being odd. Indeed we can extend the idea to any property, and define a property by the set of all elements that all hold the property. Thus, the properties of natural numbers are $\{p | p \in \wp(\mathbb{N})\}$. Further, given a set of natural numbers $X \in \wp(\mathbb{N})$, we can verify the assertion that X *holds property* $p \iff X \subseteq p \iff X \cap \overline{p} = \varnothing$. Further, we can also

provably demonstrate the verification failure: X *does not hold property* $p \iff \exists x \in X$ such that $x \notin p$, or, $X \cap \overline{p} \neq \varnothing$.

The same idea can be extended to computer programs. Understand that computer programs are essentially functions on memory states $\mathcal{E}$. In the example, properties of natural numbers were described by sets in the power-set of natural numbers. Thus, just like the example, properties of behaviors of computer programs (semantic properties) may be expressed by power-set of all possible memory states $\wp(\mathcal{E})$. Thus, the semantics (computed memory states) of program P, denoted by $[\![P]\!]$, hold some property $c \in \wp(\mathcal{E})$ iff $[\![P]\!] \subseteq c$.

The problem with this, however, is that the semantics of programs in Turing Complete Language is, in general, an infinite object and not computable. One may however compute an approximation of $[\![P]\!]$ that has a cheaper representation, but loses some information about $[\![P]\!]$. And that is the problem of Program Analysis- to compute an approximation of $[\![P]\!]$, such that one can still perform the verification, or any other application for which the analysis was performed.

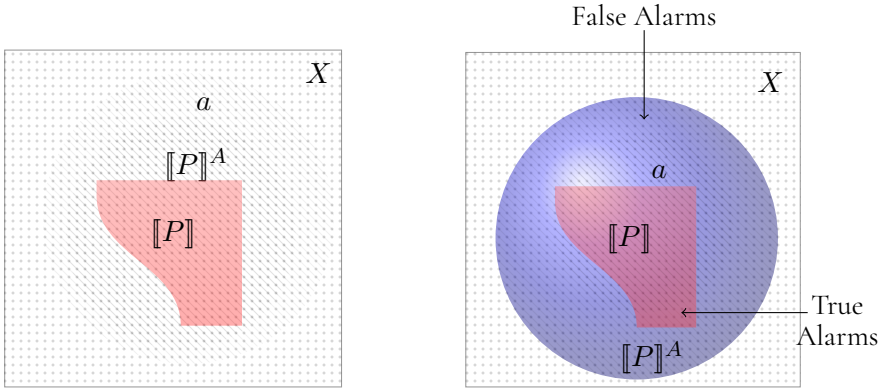
Abstract Interpretation is a mathematical framework that explains how to compute useful approximations of program semantics. Indeed other methods exist for Program Verification, including deductive methods, model checking and program typing. These methods are similar and only differ in their choices of approximations. Abstract Interpretation [17], as we show in Chapter-4, formalizes this notion of controlled loss of information via approximation in a unified framework.

3.2 | Approximations

Two types of useful approximations of program semantics are most common in Abstract Interpretation: Over-approximation and under-approximation. Over-approximation refers to computing a superset of program semantics, while under-approximation refers to computing a subset of the program semantics.

By "useful approximations" we mean that meaningful inferences drawn on these *abstract semantics*, are also valid on the *concrete semantics*—the un-abstracted real world semantics. Note that we will use the terms *abstract* and *concrete* in a relative context. This property of being able to assert inferences drawn in the abstract world in the real/concrete world is referred to as *Soundness* of program analysis.

Consider a program P with concrete semantics $[\![P]\!]$. Typically, a program analyzer would



(a) No Alarms: $[\![P]\!]^A \subseteq a$ (b) Alarms: $[\![P]\!]^A \cap \overline{a}$

Figure 3.1: Program Verification using OverApproximation: $[\![P]\!] \subseteq [\![P]\!]^A$

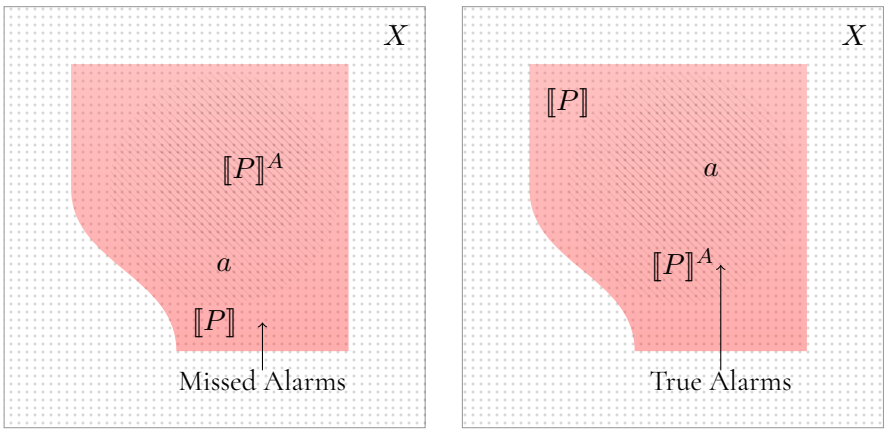
compute it's approximate (abstract) semantics $[\![P]\!]^A$, in some space of semantic objects $\wp(X)$. Then, a program verifier would check the approximate semantics against some assertion (property) $a \in \wp(X)$. The verifier would raise alarms for all $x \in [\![P]\!]^A \cap \overline{a}$. Thus, the verification is completed without raising alarms when $[\![P]\!]^A \cap \overline{a} = \varnothing \iff [\![P]\!]^A \subseteq a$.

Since approximation implies a loss of information, not all inferences will be sound, that is to say that not all inferences drawn in the abstract world will be true assertions in the concrete world.

Consider the set of concrete semantics $[\![P]\!]$, and it's over approximation $[\![P]\!]^A$ as shown in Figure-3.1. Then, $\forall x \in X,\ x \in [\![P]\!] \implies x \in [\![P]\!]^A$. Thus, if some assertion (property) $a \in \wp(X)$ holds true for $[\![P]\!]^A$ ($[\![P]\!]^A \subseteq a$), then the property also holds true for $[\![P]\!]$ ($[\![P]\!] \subseteq a$) as shown in Figure-3.1a.

However, when the verification condition does not hold for the abstract, it does not imply that the assertion fails in concrete as well. Namely, as $[\![P]\!]^A$ is an over-approximation, it contains extraneous semantic objects that do not exist in the concrete world $[\![P]\!]$. These may induce false alarms as shown in Figure-3.1b. Thus, an over-approximation may be used to verify absence of semantic objects, but not presence of semantic objects.

Dually, the under-approximation, may be used to draw sound inferences on negating assertions. As can be seen in Figure-3.2, in the case of under-approximation $\forall x \in X,\ x \in [\![P]\!]^A \implies x \in [\![P]\!]$. Thus, here, if an assertion does not hold for $[\![P]\!]^A$, it cannot be true on $[\![P]\!]$. This means that all alarms raised by verifier using under-approximation correspond to



(a) No Alarms: $[\![P]\!]^A \subseteq a$ (b) Alarms: $[\![P]\!]^A \cap \overline{a}$

Figure 3.2: Program Verification using UnderApproximation: $[\![P]\!] \supseteq [\![P]\!]^A$

property violations in the concrete world as well, as shown in Figure-3.2b.

However, when the verification condition does hold for the abstract, it does not imply that the assertion holds in concrete. Namely, as $[\![P]\!]^A$ is an under-approximation, it is missing semantic objects that do exist in the concrete world $[\![P]\!]$. These may cause the verifier to miss some alarms as shown in Figure-3.2a. Thus, an under-approximation may be used to verify absence of semantic objects, but not presence of semantic objects.

Thus, in Program Analysis, assertions are designed to describe security, correctness or robustness properties, and over-approximation is used to verify these safety properties, and under-approximation is used to provably demonstrate safety property violations.

3.3 | Challenges

Consider again the scenario of Figure-3.1 where the property a represents a desired property on the concrete semantics, and using over-approximation of concrete semantics for verification.

This approximation $[\![P]\!]^A$ loses information about the concrete world by adding additional objects that did not exist in the concrete world. This loss of information with respect to the concrete world is called loss of *Precision* or *Incompleteness of the analysis*.

Thus, the analyzer will either verify the property (as in Figure-3.1a) or raise alarms that may be false alarms (as in Figure-3.1b), in the sense that although an unsafe state was arrived in the abstract, the analyzer does not know whether the state was reachable in concrete world.

In common parlance, we often use the terms *False Positives* for these false alarms. That

stems from the labeling where Positive implies an undesired states, negative means a desired state. True and False refer to whether the label applied by analyzer in the abstract world, holds merit in the concrete world.

Similarly, the under-approximation loses precision (demonstrates incompleteness) by missing out on objects in the abstract that were present in the concrete world. This is why, here, the analysis will either provably demonstrate the presence of undesired states (see Figure-3.2b), or raise an unprovable property assertion because of the possible presence of false negatives (see Figure-3.2a).

Indeed Rice's Theorem states that program analysis is in general undecidable for all non-trivial semantic properties. Thus it makes sense that an analyzer may not be both sound and complete.

Thus, soundness, completeness and efficiency are the three main challenges in automated program analysis. The Abstract Interpretation framework guarantees soundness by design of approximation, while completeness and efficiency are traded.

3.4 | Techniques & Applications

The choice and design of approximation is fundamental to program analysis. It is heavily dictated by application, as we will see in this section.

There exist a vast variety of tools and applications for program analysis developed over the last half century, and any attempt to list them all shall be incomplete. It makes more sense to study the techniques with respect to how they tackle the challenges in program analysis- namely the Soundness, Completeness and Efficiency tradeoffs.

It should be noted, however, that Soundness and Completeness depend not just on the method, but also the application. For instance, software testing is sound when used to demonstrate a bug, but unsound when used to demonstrate the absence of bugs.

As such, we prefer to replace soundness and completeness with False Positives and False Negatives- the errors produced by program analysis. Thus we study program analysis techniques within the boundaries of a hypothetical cube of Program Analysis techniques, as shown in Figure-3.3

The axis are labeled as $1 - \mathsf{FP}$ and $1 - \mathsf{FN}$ to denote that False Positive and False Negative ratios decrease from 1 to zero as we move along the direction of the axis. Efficiency increases as we move up along that axis. We denote efficiency as time complexity w.r.t. size of program

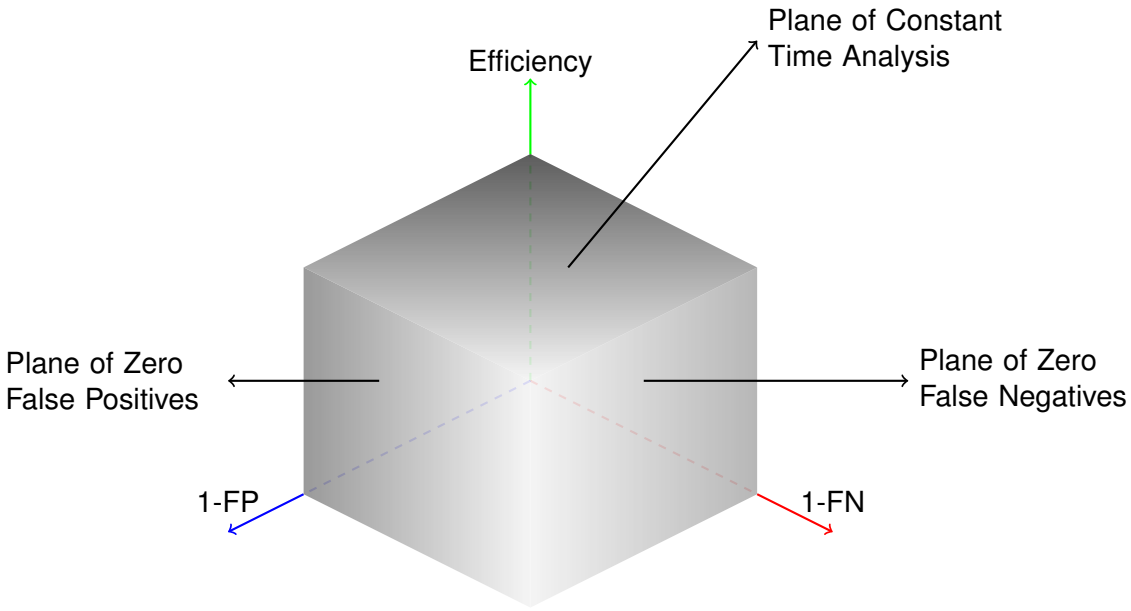


Figure 3.3: Program Analysis Techniques and Tradeoffs

semantics, in the big $\mathcal{O}$ notation.

The cube may be useful when given a set of benchmark programs, it can be used to visualize the tradeoffs made by any program analyzer available today, or in the future. The front and back planes describe the boundaries for the best and the worst possible error ratios in analyzers respectively.

The visible front planes refer to analysis that guarantee zero false positives (such as simulation and testing), or zero false negatives (such as over-approximating Abstract Interpretation), with respect to specified property. The back planes refer to False Positive and False Negative ratios of 1, indicating the highest possible error rates.

The top plane for constant time analysis refers to non-semantic analyzers such as those relying on filenames. The bottom plane is that of non-terminating analysis.

The region around the intersection of the front planes fades into white to indicate that no analyzer may exist that guarantees zero false positives and zero false negatives ratios simultaneously (in the general case).

Inside the cube, we refer to analysis that merge some of these techniques to make balance tradeoffs as suited to the respective applications. For instance, Semantic Similarity analysis, that combines static analysis with learning techniques to trade errors for efficiency. For

Instance, the work of Sharma, Aiken et. al. to combine static analysis with learning techniques [61, 62] and several others. Finally there is also work from Ernst [20] that combine static and dynamic techniques.

3.5 | Conclusion

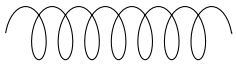
To summarize, in this chapter we have discussed the question *What is Program Analysis?*. In short, program analysis is problem of understanding program properties. It's applications include Program Verification, Malware Analysis, Malware Classification, Software Protection (obfuscation), Threat Intelligence, among others.

Program Analysis is an undecidable problem, and thus only approximate analyzers may terminate in finite time. Namely, sound and precise analyzers are non-terminating. Thus, Soundness, precision and efficiency, are three main challenges in program analysis.

Abstract Interpretation and Machine Learning based methods form the bulk of program analysis techniques. Over-Approximating abstract interpreters guarantee soundness (in the sense that they analyze all possible program behaviors), but are imprecise (generate false alarms), and, trade precision and efficiency. Under-approximating abstract interpreters guarantee precision (no false alarms), but are unsound (in the sense that they do not analyze all possible program behaviors), and trade soundness (missed alarms) for efficiency. Machine learning techniques favor efficiency over soundness and precision and thus may miss some alarms and/or generate false alarms. Other techniques may use a combination of various abstractions and machine learning techniques.

This forms a key motivation for us to search for an umbrella framework that allows to express various Abstract Interpretations and machine learning techniques under one framework. Such a framework would allow us to compare and contrast the approximations in these techniques and answer questions like- *How are approximations in Machine Learning and Abstract Interpretation similar and/or in what ways are they different?*

In the following chapters, we first describe the details of these two techniques, namely Abstract Interpretation in Chapter-4 and Chapter-5, followed by Machine Learning in Chapter-6. This is followed by Chapter-7 where we discuss our question.



"The reliance on the abstract is thus not a result of an over-estimation but rather of an insight into the limited powers of our reason"

Friedrich Hayek

4 Abstract Interpretation

4.1 | From Logic to Lattices

Recall the axiomatic approach of Hilbert discussed in the very beginning of Chapter-1. Tony Hoare applied it to computer programming in his seminal paper titled "*An axiomatic basis for computer programming*" [30], and presented Hoare Logic, in 1969, as one of the earliest methods for program verification. Quoting the paper: "This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs".

The key feature of Hoare Logic is a Hoare triple: $\langle P, S, Q \rangle$[1] where P and Q are assertions and S the program statement/command. Here, assertions P and Q are predicate logic formulae, and describe the pre and post conditions respectively, with respect to the execution of the program statement S. Thus, the idea is that given a pre-condition and a program statement, one may use the axioms and logic inference rules to derive a proof for checking the post-condition.

Gödel's incompleteness manifests here in the form of the inability of the proof system to always prove termination. This is why, a hoare triple is often read as: "*If P is the pre-condition that holds before the execution of program S, then either Q is the post-condition that holds after the execution of S or S does not terminate.*"

We are not interested in describing the entire proof system in detail. However, to give a sense of what it is about, we list some axioms and inference rules in examples:

Example 4.1.1 (Empty Statement Axiom). $\dfrac{}{\{P\}\ \mathsf{skip}\ \{P\}}$

Example 4.1.2 (Assignment Axiom). $\dfrac{}{\{P[E/x]\}\ x{:=}E\ \{P\}}$

Example-4.1.1 describes the skip (empty) statement that does not change the state of program, hence the post-condition is same as pre-condition. Example-4.1.2 describes the assignment axiom that states given assertion P in which variable x is free, then, after the assignment,

[1] Hoare used the notation $P\{C\}Q$

any predicate that was previously true for the right-hand side of the assignment now holds for the variable. Finally, Example-4.1.3 demonstrates an informal proof for a program to swap two variables without using a third variable.

Example 4.1.3 (Proof Example with Hoare Logic)**.**

Nr.	Code	Assertions
1.		$\{a = A \wedge b = B\}$
2.	$a := a + b;$	
3.		$\{a - b = A \wedge b = B\}$
4.	$b := a - b;$	
5.		$\{b = A \wedge a - b = B\}$
6.	$a := a - b$	
7.		$\{b = A \wedge a = B\}$

The key takeaway is that P and Q are predicate logic formulae, and that there exist axioms and inference rules for every possible program statement that describes the relation between pre-condition and post-condition.

Following the approach of Chapter-3 where we described properties by sets of objects that satisfy the property, we can describe the assertions here by the set of states that satisfy the assertion. Thus, $P, Q \in \wp(\mathcal{E})$ where $\mathcal{E}$ is the set of all possible program states; and the inference rules maybe viewed as operators on the complete lattice $(\wp(\mathcal{E}), \subseteq, \cup, \cap, \varnothing, \mathcal{E})$.

Indeed lattices arise naturally in Logic. Dedekind was the first to recognize the connection between algebra and Lattice Theory and remarked "There's nothing new under the sun" [48]. It is not surprising then, that several works of that time leveraged Partial Orders and Lattices for proving program properties [5, 50, 57, 58].

Consider a Hoare triple for Example–4.1.3: $\{a = A \wedge b = B\} a := a + b; b := a - b; a := a - b \{b = A \wedge a = B\}$. Notice that this semantics retains the sets of initial and final states, but loses information on the intermediate states. Additionally, Hoare Logic semantics cannot work with infinite computation since the final state is unknowable/does not exist.

We are, however, interested program invariants- assertions that hold true for all possible executions, and hence must operate in semantics that can describe invariants. Such semantics must include the initial, final and all intermediate states. Indeed *concrete collecting semantics* are such a semantics. These may be understood as abstract semantics that retain the set of initial states, final states and all the intermediate program states arrived at during the computation, but loses all information about the transitions.

Given the partial order of collecting semantics (the concrete domain), the collecting program semantics can be expressed as least fix-point solutions to fixpoint operators [40, 41]. Incompleteness manifests in the form of non-computability of the least fixpoint in the general case.

Cousot and Cousot, in 1976, first presented the idea of approximating the least fixpoint computation by using an abstract domain of intervals [16]. This approach was generalized into the Abstract Interpretation framework in the seminal paper [17] in 1977.

Indeed one can also design Hoare-like logics that operate on these abstract domains, however, the idea of abstraction on the lattices is more intuitive. This makes it easier to understand the proof using lattice arguments. Recall the quote from Garrett Birkhoff printed before Chapter-2.

4.2 | Language and Least Fixpoint Semantics

We describe the numeric toy language used by Mine in his tutorial [45]. Indeed the rest of this chapter describes Abstract Interpretation framework as discussed in the tutorial and we do not claim any original work in the following sections.

Although the toy language has limited constructs, it is Turing Complete. We first present the syntax of our language. We then present its concrete collecting semantics.

4.2.1 — Syntax

We assume a fixed, finite set $\mathbb{V}$ of program variables, with numeric values. We denote by $\mathbb{I}$ the domain of variables, and assume that $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. For simplicity, we assume these to be numeric sets and not machine-integers or floating-point numbers.

Figure-4.1 describes the syntax of our language with program statements *stat* in Figure-4.1a, Numeric expressions *expr* in Figure-4.1b, Conditions *cond* in Figure-4.1c.

The expressions include an interval construction: $[c_1, c_2]$ to denote non-deterministic inputs. We assume that $[c_1, c_2]$ is not empty, that is $c_1 \in \mathbb{I} \cup \{-\infty\}, c_2 \in \mathbb{I} \cup \{+\infty\}$, and $c_1 \leq c_2$. Note that this makes the semantics of program non-deterministic. Hence verifiers must consider the set of all possible outcomes at these intervals. This helps model unavailable code, external environment interactions, etc.

stat	::=	$V \longleftarrow$ *expr*	(assignment, $V \in \mathbb{V}$)
	|	*stat*;*stat*	(sequence)
	|	**if** *cond* **then** *stat* **else** *stat* **endif**	(conditional)
	|	**while** *cond* **do** *stat* **done**	(loop)
	|	**skip**	(no-op)
	|	**assert** *cond*	(assertion)

(a) Syntax of Programs

expr	::=	V	(variable, $V \in \mathbb{V}$)
	|	c	(numeric constant, $c \in \mathbb{I}$)
	|	$-$ *expr*	(negation)
	|	*expr* $\diamond$ *expr*	(binary operator, $\diamond \in \{+, -, \times, /\}$)
	|	$[c_1, c_2]$	(input, $c_1, c_2 \in \mathbb{I} \cup \{-\infty, +\infty\}$)

(b) Syntax of (numeric) Expressions

cond	::=	b	(boolean constant, $b \in \{b \in \mathbb{B}\}$)
	|	$\neg$*cond*	(logic negation)
	|	*cond* $\vee$ *cond*	(logic or)
	|	*cond* $\wedge$ *cond*	(logic and)
	|	*expr* $\bowtie$ *expr*	(comparison, $\bowtie \in \{\leq, \geq, <, >, =, \neq\}$)

(c) Syntax of (boolean) conditions

Figure 4.1: Syntax of Language

4.2.2 — Concrete Semantics

We are interested in inferring program invariants, i.e., properties of the memory state a program can be in at each program location. Hence, our concrete collecting semantics operates on a domain of sets of memory states. A memory state, denoted as $\rho \in \mathcal{E}$, is a function mapping each variable to its value: $\rho \stackrel{\text{def}}{=} (\mathbb{V} \longrightarrow \mathbb{I})$. The concrete domain is thus the power-set complete lattice:

$$(D, \subseteq, \cup, \cap, \emptyset, \mathcal{E}) \quad \text{where } D \stackrel{\text{def}}{=} \wp(\mathcal{E}) \tag{4.1}$$

Given a single memory state $\rho \in \mathcal{E}$, an expression $e \in$ *expr* can evaluate to one or more (due to non-determinism) values in $\mathbb{I}$. The evaluation function, denoted as $\mathbb{E}[\![e]\!] : \mathcal{E} \longrightarrow \wp(\mathbb{I})$, is defined in Table-4.1a by induction on the syntax.

Conditions, described in Table-4.1b, filter memory states. To simplify, we assume that all negations $\neg$ have been removed using DeMorgan's laws and usual arithmetic laws:$\neg(a \vee b) =$

$\mathbb{E}[\![expr]\!] : \mathcal{E} \longrightarrow \wp(\mathbb{I})$		
$\mathbb{E}[\![V]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{\rho(V)\}$
$\mathbb{E}[\![c]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{c\}$
$\mathbb{E}[\![[c_1, c_2]]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{x \in \mathbb{I} \mid c_1 \leq x \leq c_2\}$
$\mathbb{E}[\![-e]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{-v \mid v \in \mathbb{E}[\![e]\!]\rho\}$
$\mathbb{E}[\![e_1 + e_2]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{v_1 + v_2 \mid v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho\}$
$\mathbb{E}[\![e_1 - e_2]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{v_1 - v_2 \mid v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho\}$
$\mathbb{E}[\![e_1 \times e_2]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{v_1 \times v_2 \mid v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho\}$
$\mathbb{E}[\![e_1/e_2]\!]\rho$	$\stackrel{\text{def}}{=}$	$\{v_1/v_2 \mid v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho, v_2 \neq 0\}$

(a) Concrete Collecting Semantics of Expressions

$\mathbb{C}[\![cond]\!]R : D \longrightarrow D$		
$\mathbb{C}[\![\mathsf{True}]\!]R$	$\stackrel{\text{def}}{=}$	R
$\mathbb{C}[\![\mathsf{False}]\!]R$	$\stackrel{\text{def}}{=}$	$\emptyset$
$\mathbb{C}[\![c_1 \wedge c_2]\!]R$	$\stackrel{\text{def}}{=}$	$\mathbb{C}[\![c_1]\!]R \cap \mathbb{C}[\![c_2]\!]R$
$\mathbb{C}[\![c_1 \vee c_2]\!]R$	$\stackrel{\text{def}}{=}$	$\mathbb{C}[\![c_1]\!]R \cup \mathbb{C}[\![c_2]\!]R$
$\mathbb{C}[\![e_1 = e_2]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho \in R \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho : v_1 = v_2\}$
$\mathbb{C}[\![e_1 \neq e_2]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho \in R \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho : v_1 \neq v_2\}$
$\mathbb{C}[\![e_1 < e_2]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho \in R \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho : v_1 < v_2\}$
$\mathbb{C}[\![e_1 > e_2]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho \in R \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho : v_1 > v_2\}$
$\mathbb{C}[\![e_1 \leq e_2]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho \in R \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho : v_1 \leq v_2\}$
$\mathbb{C}[\![e_1 \geq e_2]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho \in R \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\rho, v_2 \in \mathbb{E}[\![e_2]\!]\rho : v_1 \geq v_2\}$

(b) Concrete Collecting Semantics of Conditions

Table 4.1: Concrete Collecting Semantics of Expressions and Conditionals

$(\neg a) \wedge (\neg b), \neg(e_1 = e_2) = (e_1 \neq e_2)$, etc.

Note that since e_1 and e_2 can evaluate to several values in ρ, an arithmetic comparison such as $e_1 = e_2$ holds on a given memory state ρ when at least one possible value of $\mathbb{E}[\![e_1]\!]\rho$ equals one possible value of $\mathbb{E}[\![e_2]\!]\rho$. Thus, a state may simultaneously satisfy a condition and it's negation. Also interesting is the *join morphism* $\mathbb{C}[\![c]\!](\cup_{i\in I} R_i) = \cup_{i\in I}(\mathbb{C}[\![c]\!]R_i)$ for arbitrary families of $(R_i)_{i\in I}$.

The *semantic transfer function*, denoted as $\mathbb{S}[\![s]\!] : D \longrightarrow D$, is an input/output relation that maps the set of states before the execution of s to the set of states after the execution. It is described in Table-4.2.

We can extend the idea to compound statements as well by induction on syntax. A sequence of statements $s_1; s_2$ is simply composition of functions.

For conditionals, consider the statement "**if** *cond* **then** s_1 **else** s_2 **endif**" as applied

$\mathbb{S}[\![stat]\!] : D \longrightarrow D$		
$\mathbb{S}[\![V \longrightarrow e]\!]R$	$\stackrel{\text{def}}{=}$	$\{\rho[V \mapsto v] \mid \rho \in R, v \in \mathbb{E}[\![e]\!]\rho\}$
$\mathbb{S}[\![\textsf{assert } cond]\!]R$	$\stackrel{\text{def}}{=}$	$\mathbb{C}[\![cond]\!]R$
$\mathbb{S}[\![\textsf{skip}]\!]R$	$\stackrel{\text{def}}{=}$	R
$\mathbb{S}[\![s_1; s_2]\!]R$	$\stackrel{\text{def}}{=}$	$(\mathbb{S}[\![s_2]\!] \circ \mathbb{S}[\![s_1]\!])R$
$\mathbb{S}[\![\textsf{if } cond \textsf{ then } s_1 \textsf{ else } s_2 \textsf{ endif}]\!]R$	$\stackrel{\text{def}}{=}$	$\mathbb{S}[\![s_1]\!](\mathbb{C}[\![cond]\!]R) \cup \mathbb{S}[\![s_2]\!](\mathbb{C}[\![\neg cond]\!]R)$
$\mathbb{S}[\![\textsf{while } cond \textsf{ do } s \textsf{ done}]\!]R$	$\stackrel{\text{def}}{=}$	$\mathbb{C}[\![\neg cond]\!]\,\text{lfp}\,F$ where $F(X) \stackrel{\text{def}}{=} R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![cond]\!]X)$

Table 4.2: Concrete Collecting Semantics of Program Statements

to a set of states R. First, the condition filters R as $R^1 = \mathbb{C}[\![cond]\!]R$ and $R^2 = \mathbb{C}[\![\neg cond]\!]R$ for the **then** and **else** branches. Then the semantic transfer functions $\mathbb{S}[\![s_1]\!]$ and $\mathbb{S}[\![s_2]\!]$ are applied to R^1 and R^2 respectively: $R^{1'} = \mathbb{S}[\![s_1]\!](R^1)$ and $R^{2'} = \mathbb{S}[\![s_2]\!](R^2)$. Finally, we return $R^{1'} \cup R^{2'}$.

For loops, consider the statement "**while** *cond* **do** s **done**" as applied to a set of states R. Also, consider loop invariant I as a set of states. Then, $\rho \in I$ can be interpreted by induction as:

- Either we have not performed any loop iteration yet, and $\rho \in R$;
- Or, we have performed one or more loop iterations and generated new state ρ from some state $\rho' \in I$. Thus, ρ' satisfies the loop condition *cond* and produces ρ after executing the loop body s, i.e., $\rho \in \mathbb{S}[\![s]\!](\mathbb{C}[\![cond]\!]\{\rho'\})$.

Thus, we have established the following equation: $I = R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![cond]\!]I)$, which can be expressed as $I = F(I)$ using our definition of F in Table-4.2.

Thus, fixpoints of F are loop invariants and the least fixpoint is the smallest, tighest loop invariant we can derive. Finally, we keep only states that will exit the loop by applying the filter $\mathbb{C}[\![\neg cond]\!]$ to lfp F.

Theorem 4.2.1 (Existence of lfp F)**.** *For any statement $s \in stat$, the semantic function $\mathbb{S}[\![s]\!] : D \longrightarrow D$ is a join morphism; hence, it is monotonic and continuous. As a consequence, the least-fixpoints used in the semantics of loops are also well-defined, through both Tarski's and Kleene's fixpoint theorems- Theorems-2.2.11 and 2.2.12.*

Thus, we have described a language with syntax and concrete collecting semantics that if computed, would infer the precise numerical invariants. Unfortunately, they cannot be computed due to three reasons:

1. Non-representable Concrete Semantics: The concrete elements live in $D \stackrel{\text{def}}{=} \wp(\mathcal{E})$, and so, are not all representable in a computer — even when restricting the domain of variables I to be finite machine-integers or floats, although D becomes finite, it remains extremely large, so that a naive representation of sets in extension is not practicable;

2. Expensive Computations: The semantic operators for atomic statements $\mathbb{S}[\![V \longleftarrow e]\!]$, $\mathbb{C}[\![cond]\!]$ and join $\cup$ are not computable — or, given a finite D, would be too costly to evaluate individually on each memory state;

3. Non-Computability of Least Fixpoints: The iterations required in the semantics of loops may not converge in finite time — or, for finite D, may require iterating on finite, yet extremely long chains.

We now show how Abstract Interpretation tackles challenge 1 and 2 by leveraging abstract domains and challenge 3 by using convergence acceleration.

4.3 | Abstract Domains

Abstract Interpretation replaces costly, expressive representations with cheaper, more simpler ones. The idea is that since the properties in the concrete domain $(C, \subseteq)$ are precise and expensive to represent, we lose some details, not relevant to the intended application, and represent properties in some simpler abstract domain $(A, \sqsubseteq)$ that allows cheaper representation of properties. The concrete and abstract world are related via a concretization function γ:

Definition 4.3.1 (Concretization Function)**.** Concretization function $\gamma \in (A, \sqsubseteq) \longrightarrow (C, \subseteq)$ is a monotonic function assigning a concrete meaning, in C, to each abstract properties in A.

Definition 4.3.2 (Soundness & Exactness of abstract properties)**.** Given a concretization function γ connecting an abstract domain of objects $(A, \sqsubseteq)$, to it's concrete domain of objects $(C, \subseteq)$, we say:

- $a \in A$ is a sound abstraction of $c \in C$ if and only if $c \subseteq \gamma(a)$.
- $a \in A$ is an exact abstraction of $c \in C$ if and only if $c = \gamma(a)$.

Additionally, we also replace expensive computations in concrete domain D with abstract transfer functions in $D^\sharp$.

Definition 4.3.3 (Soundness & Exactness of abstract transfer functions)**.** Given a concretization function γ connecting an abstract domain of objects $(A, \sqsubseteq)$, to it's concrete domain of objects $(C, \subseteq)$, a concrete operator $f : C \longrightarrow C$ and an abstract operator $g : A \longrightarrow A$, we say:

- g is a sound approximation of f if and only if $\forall a \in A, (f \circ \gamma)a \subseteq (\gamma \circ g)a$.
- g is an exact approximation of f if and only if $\forall a \in A, (f \circ \gamma)a = (\gamma \circ g)a$.

Just like concrete transfer functions, abstract transfer functions may also be composed to handle sequence of statements:

Theorem 4.3.4 (Operator Composition)**.** *Given concrete operators $f, f' : C \longrightarrow C$ and abstract operators $g, g' : A \longrightarrow A$:*

1. *if g and g' are sound approximations (definition-4.3.3) of f and f', and f is monotonic, then $g \circ g'$ is a sound abstraction of $f \circ f'$.*
2. *if g and g' are exact approximations of f and f', then $g \circ g'$ is an exact abstraction of $f \circ f'$.*

Finally, for approximating least fix-point in finite iterations, the framework introduces a widening operator (see definition-4.3.7). The idea for widening is rooted in two fixpoint approximation theorems (Theorems-4.3.5 and 4.3.5) proved by Cousot & Cousot discussed below.

The theorems state the conditions under which the limit of the abstract operator $F^\sharp$ is a sound approximation of least fixpoint of the concrete operator F. Widening, then is defined as the operator that computes that limit in the abstract, in finite steps (see Theorem-4.3.8), and hence computes a sound approximation of the least fix-point.

Theorem 4.3.5 (Kleene fixpoint Approximation)**.** *If $f : C \longrightarrow C$ is continuous in a CPO $(C, \subseteq, \cup, \bot)$, and $g : A \longrightarrow A$ is a sound — not necessarily monotonic — abstraction of f in a poset abstract domain $(A, \subseteq, \cup, \bot')$, and the sequence $g^i(\bot')$ has a limit x in A, then it is a sound approximation of* $\mathrm{lfp}\, f$ *, i.e.,* $\mathrm{lfp}\, f \subseteq \gamma(x)$.

Theorem 4.3.6 (Tarski fixpoint Approximation)**.** *Given a complete lattice concrete domain $(C, \subseteq, \cup, \cap, \bot, \top)$, a monotonic concrete function $f : C \longrightarrow C$, and a sound — not necessarily monotonic — abstraction $g : A \longrightarrow A$ of f in a poset abstract domain $(A, \sqsubseteq)$, then any postfix-point a of g, i.e., $g(a) \sqsubseteq a$, is a sound abstraction of* $\mathrm{lfp}\, f$*, i.e.,* $\mathrm{lfp}\, f \leq \gamma(a)$.

Definition 4.3.7 (Widening)**.** A binary operator $\nabla : A \times A \longrightarrow A$ in an abstract domain $(A, \sqsubseteq)$ is a widening operator if and only if:

1. it computes upper bounds: $\forall x, y \in A : x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$;

2. and it enforces convergence: for any sequence $(y^i)_{i \in \mathbb{N}}$ in A, the sequence $(x^i)_{i \in \mathbb{N}}$ computed as $x^0 \stackrel{\text{def}}{=} y^0$, $x^{i+1} \stackrel{\text{def}}{=} x^i \nabla y^{i+1}$ stabilizes in finite time: $\exists k \geq 0 : x^{k+1} = x^k$.

Theorem 4.3.8. *If f is a monotonic operator in a complete concrete lattice and g is a sound abstraction of f , then the following iteration:*

$$x^0 \stackrel{\text{def}}{=} \bot \tag{4.2}$$

$$x^{i+1} \stackrel{\text{def}}{=} x^i \nabla g(x^i) \tag{4.3}$$

converges in finite time, and it's limit x is a sound abstraction of $\text{lfp}\, f$*:* $\text{lfp}\, f \subseteq \gamma(x)$.

We now formally discuss the elements of an abstract domain. An abstract domain is given by:

1. Minimum Algebraic Structure: a set $D^\sharp$ of computer-representable abstract values; and an effective partial order $\subseteq^\sharp$ on $D^\sharp$.

2. Concretization Function: a monotonic concretization function $\gamma : D^\sharp \longrightarrow D$ (definition-4.3.1).

3. a smallest element $\bot^\sharp$ and a largest element $\top^\sharp \in D^\sharp$ that represent respectively $\emptyset$ and $\mathcal{E}$

4. (optionally) a galois connection $(D, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp, \subseteq^\sharp)$ (definition-2.2.17)

5. sound and effective abstractions (definition-4.3.3) of assignments and atomic arithmetic conditions:
   $\mathbb{S}^\sharp[\![V \longleftarrow e]\!] : D^\sharp \longrightarrow D^\sharp$
   $\mathbb{C}^\sharp[\![e_1 \bowtie e_2]\!] : D^\sharp \longrightarrow D^\sharp$ such that:
   $$\forall X^\sharp \in D^\sharp : (\mathbb{S}[\![V \longleftarrow e]\!] \circ \gamma) X^\sharp \subseteq (\gamma \circ \mathbb{S}[\![V \longleftarrow e]\!]) X^\sharp$$
   $$\forall X^\sharp \in D^\sharp : (\mathbb{S}[\![e_1 \bowtie e_2]\!] \circ \gamma) X^\sharp \subseteq (\gamma \circ \mathbb{S}[\![e_1 \bowtie e_2]\!]) X^\sharp$$

6. sound and effective abstractions (definition-4.3.3) of set union $\cup$ and set intersection $\cap$:
   $\cup^\sharp : D^\sharp \longrightarrow D^\sharp$ and $\cap^\sharp : D^\sharp \longrightarrow D^\sharp$ such that:
   $$\forall X^\sharp, Y^\sharp \in D^\sharp : \gamma(X^\sharp) \cup \gamma(Y^\sharp) \subseteq \gamma(X^\sharp \cup^\sharp Y^\sharp)$$
   $$\forall X^\sharp, Y^\sharp \in D^\sharp : \gamma(X^\sharp) \cap \gamma(Y^\sharp) \subseteq \gamma(X^\sharp \cap^\sharp Y^\sharp)$$

7. a widening operator ∇ (definition-4.3.7)

Requirement-1 describe the minimum algebraic structure needed as Posets. However, many ordered sets have a richer structure, for instance the interval domain is a complete lattice (discussed in the next chapter, section-**??**) defined as:

$$(\{[a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\bot\}, \subseteq, \sqcup, \sqcap, \bot, [-\infty, +\infty]) \quad (4.4)$$

Indeed lack of structure often results in loss of precision.

Non-distributivity of abstract domain lattice, for instance, is a common cause of precision loss. A typical analysis design tends to join information, such as the result of different control-flow paths merging at some common program location, as early as possible, favoring computations of the form $(a \sqcup b) \sqcap c$. A formulation such as $(a \sqcap c) \sqcup (b \sqcap c)$, that delays the join, requires more operators, and is thus more costly. The later is, however, always at least as precise as the former, and it is strictly more precise if the lattice is not distributive. Yet, limiting ourselves to distributive lattices would severely hinder our ability to choose the appropriate domain for each task. When precision matters, we may consider a tradeoff where the later formulation, delaying joins, is used parsimoniously. This phenomenon is well-known in the field of data-flow analysis [32], where the second formulation leads to the meet-over-all-paths algorithm, and the former leads to the least fixpoint algorithm.

Similarily, when abstract domain lattice is not a sublattice of the concrete domain lattice, this lack of a strong algebraic property (sublattice), results in a loss of precision. But, again, limiting ourselves to sublattices only would hinder our ability to use extremely useful abstractions, such as intervals which is not a sublattice of concrete domain of integers. Consequently, result of some operations, such as the join of two intervals, must be approximated to stay within the abstract world of intervals.

We can already argue that Abstract Interpretation is particularly lax when it comes to algebraic requirements, especially on the abstract world. It is required to be a poset, but not necessarily a CPO nor a lattice. Even if $D^\sharp$ is a lattice, we do not require that $\cup^\sharp$ and $\cap^\sharp$ correspond to the lub $\sqcup^\sharp$ and glb $\cap^\sharp$. It may be a key to its success, as it leaves abstractions open to many possibilities.

Requirement-2 is for a concretization function γ that connects the abstract world to concrete, to be monotone. The monotonicity simply states that coarser abstract elements represent coarser concrete elements. Also note that that γ does not need to be onto — i.e., injective. It

is possible to imagine an abstract world where several abstract elements represent the same concrete element. Consider for instance a variant of Interval Lattice from Equation-4.4, where we do not enforce $a \leq b$: $\{(a, b) \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in Z \cup \{+\infty\}\}$. Here any pair such that $a > b$ is a representation for the empty set. This is mainly useful in the case of domains where computing unique, normal forms for an abstract element is not possible or would be too time-consuming.

Requirement-4 optionally requires a galois connection between the abstract and concrete domain. Following theorem provides an alternate, but equivalent view of galois connections to definition-2.2.17.

Theorem 4.3.9 (Alternate characterization of Galois connections)**.** *We have a galois connection* $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *if and only if the function pair* (α, γ) *satisfy the following properties:*

1. γ *is monotonic.*
2. α *is monotonic.*
3. $\gamma \circ \alpha$ *is extensive i.e.,* $\forall c \in C : c \subseteq (\gamma \circ \alpha)c$.
4. $\alpha \circ \gamma$ *is reductive i.e.,* $\forall a \in A : (\alpha \circ \gamma)a \sqsubseteq a$.

Property-3 states that, going through the abstract world A and back gives a result that is either equal or less precise than staying in the concrete. The result is strictly less precise when the original concrete element has no exact representation in the abstract, so that we lose information during the conversion between the concrete and the abstract.

Property-4 states that coming from the abstract and going back to the abstract through the concrete, we may end up with a smaller abstract element. In fact, this happens only when a concrete element has several different abstract representations, in which case α will naturally choose the smallest for the abstract order $\sqsubseteq$.

Galois connection should be viewed as another strong algebraic relationship between concrete and abstract domains, which is not required, but if it exists, provides additional gains. In this case, galois connection can be used to derive optimal abstract operators. Definition-2.2.15-3 states a very important property relating Galois connections, soundness, and optimality. Indeed, $c \leq \gamma(a)$ means, by definition-4.3.2, that a is a sound abstraction of c. Hence, $\alpha(c) = \cup\{a \mid c \leq \gamma(a)\}$ is, literally, the best (i.e., smallest) sound abstraction of c. We can additionally generalize this notion of best abstraction to operators:

Definition 4.3.10 (Best operator abstraction)**.** Given a galois connection $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, an operator $f : C \longrightarrow C$, the best abstraction of f is given by $(\alpha \circ f \circ \gamma)$.

Definition-4.3.10 is a powerful tool as it allows deriving the abstract semantics systematically from the concrete one and a Galois connection. Note, however, that this is a constructive mathematical definition that cannot generally be implemented as is, as neither its components α, f, γ are likely to be computable. Rather, it is the designer's responsibility to turn this mathematical definition into an algorithm. Sometimes, it is not easy to derive such an algorithm, or the algorithm might not be sufficiently efficient. Finally, there is always the case of abstract domains without a Galois connection- example polyedra domain [18]. In those cases, the designer has to rely on intuition to invent a suitable abstract operator, several choices being possibles, and then prove its soundness through definition-4.3.3.

Theorem 4.3.11 (Non-Composability of Optimality)**.** *Given concrete operators $f, f' : C \longrightarrow C$ and their best abstract operators $g, g' : A \longrightarrow A$, $g \circ g'$ is a sound approximation of $f \circ f'$, but not necessarily the best abstract operator.*

Proof. As best abstract operators are sound, then by theorem-4.3.10 g and g' are sound approximations of f and f' respectively. Using theorem-4.3.4, composition of sound abstract operators results in sound approximation of composition of corresponding concrete operators. Thus $g \circ g'$ is a sound approximation of $f \circ f'$. By definition-4.3.10, $g = \alpha \circ f \circ \gamma$ and $g' = \alpha \circ f' \circ \gamma$. Thus, $g \circ g' = \alpha \circ f \circ \gamma \circ \alpha \circ f' \circ \gamma$, while the best abstraction for $f \circ f' = g''(say) = \alpha \circ f \circ f' \circ \gamma$. We know by property of galois connection (definition-4.3.9-3 that $\gamma \circ \alpha$ is extensive, hence $g'' \sqsubseteq g \circ g'$. □

The lack of general composability for the notion of best abstraction has rather important practical ramifications. The precision of an analysis depends on the granularity of the decomposition of the program semantics into atomic operations abstracted independently. The finer the decomposition, the larger the risk of some imprecision appearing. A coarser decomposition allows, on the other hand, optimal abstractions for larger code blocks. It may not be practicable, however, as the number of possible blocks grows in a combinatorial fashion with block size. There is generally a trade-off to achieve.

$\mathbb{S}^\sharp[\![stat]\!], \mathbb{C}^\sharp[\![cond]\!] : D^\sharp \longrightarrow D^\sharp$		
$\mathbb{S}^\sharp[\![V \longrightarrow e]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	given
$\mathbb{C}^\sharp[\![c_1 \bowtie c_2]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	given
$\mathbb{S}^\sharp[\![\texttt{assert } cond]\!]R$	$\stackrel{\text{def}}{=}$	$\mathbb{C}^\sharp[\![cond]\!]R^\sharp$
$\mathbb{S}^\sharp[\![\texttt{skip}]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$R^\sharp$
$\mathbb{S}^\sharp[\![s_1; s_2]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$(\mathbb{S}^\sharp[\![s_2]\!] \circ \mathbb{S}^\sharp[\![s_1]\!])R^\sharp$
$\mathbb{S}^\sharp[\![\texttt{if } cond \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ endif}]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$\mathbb{S}^\sharp[\![s_1]\!](\mathbb{C}^\sharp[\![cond]\!]R^\sharp) \cup^\sharp \mathbb{S}^\sharp[\![s_2]\!](\mathbb{C}^\sharp[\![\neg cond]\!]R^\sharp)$
$\mathbb{S}^\sharp[\![\texttt{while } cond \texttt{ do } s \texttt{ done}]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$\mathbb{C}^\sharp[\![\neg cond]\!](\lim F^\sharp)$ where $F^\sharp(X^\sharp) \stackrel{\text{def}}{=} X^\sharp \nabla (R^\sharp \cup^\sharp \mathbb{S}^\sharp[\![s]\!](\mathbb{C}^\sharp[\![cond]\!]X^\sharp))$
$\mathbb{C}^\sharp[\![\texttt{True}]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$R^\sharp$
$\mathbb{C}^\sharp[\![\texttt{False}]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$\bot^\sharp$
$\mathbb{C}^\sharp[\![c_1 \wedge c_2]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$\mathbb{C}^\sharp[\![c_1]\!]R^\sharp \cap^\sharp \mathbb{C}^\sharp[\![c_2]\!]R^\sharp$
$\mathbb{C}^\sharp[\![c_1 \vee c_2]\!]R^\sharp$	$\stackrel{\text{def}}{=}$	$\mathbb{C}^\sharp[\![c_1]\!]R^\sharp \cup^\sharp \mathbb{C}^\sharp[\![c_2]\!]R^\sharp$

Table 4.3: Abstract Semantics of Program Statements

4.3.1 — Abstract Semantics

Table-4.3 presents the abstract version of the concrete semantics presented in Table-4.2. The semantics now operates only in abstract domain $D^\sharp$. It uses the abstract version of assignments, tests, join, and meet operators we assume given with the domain. It composes them to construct the semantics of more complex statements and tests by induction on the syntax, and we can see that it follows very closely the concrete definition, up to the use of $\sharp$ symbols.

Another key difference is that the concrete least fixpoint $\text{lfp}\, F$ used in the semantics of loops has been replaced with $\lim F^\sharp$, which computes the limit of the iterates of $F^\sharp$ from $\bot^\sharp$:

$$\lim F^\sharp \stackrel{\text{def}}{=} F^{\sharp\delta}(\bot^\sharp)$$

where δ is the minimal value such that $F^{\sharp\delta+1}(\bot^\sharp) = F^{\sharp\delta}(\bot^\sharp)$.

The definition of the widening, definition-4.3.7, ensures that this limit is always reached after a finite number of iterations. The result of the analysis is sound: it is the composition of sound abstractions (theorem-4.3.4) and sound fixpoint abstractions with widening (theorem-4.3.8), so:

Theorem 4.3.12 (Termination and soundness)**.** $\mathbb{S}^\sharp[\![p]\!]$ *always terminates, and is sound:* $\forall p \in stat$, $\mathbb{I}^\sharp \in D^\sharp : \mathbb{S}[\![p]\!](\gamma(\mathbb{I}^\sharp)) \subseteq \gamma(\mathbb{S}^\sharp[\![p]\!](\mathbb{I}^\sharp))$.

4.4 | Summary

To summarize, Cousot & Cousot, during the late 20^{th} century, presented a theory of Abstract Interpretation [17] for reasoning about the semantics of discrete dynamic systems, e.g., programming languages, at different levels of abstraction.

We showed how Abstract Interpretation is used as a mathematical framework to soundly approximate the concrete program semantics into various abstract domains.

The framework consists of three key steps:

1. Describe the concrete and abstract semantics as interpretations on concrete and abstract (approximate) domains. The domains are specified as partial orders.

   Consider $[\![P]\!]$ is the concrete semantics mapping programs P into properties of computed states in $\mathcal{E}$, namely $[\![P]\!] \in \wp(\mathcal{E})$, where $(\wp(\mathcal{E}), \subseteq, \cup, \cap, \varnothing, \mathcal{E})$ is a complete lattice concrete semantic domain. An abstract domain $(A, \sqsubseteq)$ of approximate properties $a \in A$ such that $\gamma(a) \in \wp(\mathcal{E})$ is the concrete property represented by a, abstract semantics $[\![P]\!]^A$ is specified as an approximate interpreter for P on approximate (abstract) denotations in domain of objects A.

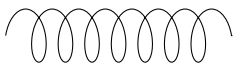
2. Concrete and abstract semantics are specified as least-fixpoint solutions to a corresponding fixpoint operators on the concrete and abstract domain.

   The concrete interpretation is usually specified as the solution of fix-point equations of the form $X = F_P(X)$, i.e., $[\![P]\!] = \mathrm{lfp}\, F_P$ of a continuous *predicate transformer* $F_P \in \wp(S) \longrightarrow \wp(S)$. It is known that $\mathrm{lfp}\, F_P = \bigcup_{n \in N} F_P^n(\varnothing)$ of the *iterates* $F_P^n(\varnothing)$ defined by $F_P^0(X) = X$ and $F_P^{n+1}(X) = F_P(F_P^n(X))$ for all $X \in \wp(S)$. Similarily, $[\![P]\!]^A$ is defined as approximate fix-point computations of an approximate operation F_P^A on A.

3. Finally, it provides a way to approximate these least fixpoint solutions in an efficient manner.

   Since the $\mathrm{lfp}\, F_P$ is not computable in the general case, it may be soundly approximated by $\lim F_P^A$ which is computable. Additionally, when the computation of $\lim F_P^A$ requires large number of iterations, Widening operations ∇ are used to accelerate convergence to the limit in A. Widening also guarantees finite convergence of the approximate iterates in A. A widening $\nabla \in A \times A \to A$ is such that: $\forall x, y \in A\colon x, y \sqsubseteq x \nabla y$ and for all increasing chains $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$ in A, the increasing chain defined by

$y^0 = x^0, \ldots, y^{i+1} = y^i \nabla x^{i+1}, \ldots$ has a finite limit. In this case the approximate iterates $\bar{a}^0 = \bot$, $\bar{a}^{i+1} = \bar{a}^i \nabla F_P^A(\bar{a}^i)$ is ultimately stationary and it's limit $\bar{a}$ is a sound upper approximation of $[\![P]\!]$. Soundness here means that $[\![P]\!] \subseteq \gamma([\![P]\!]A)$.



"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise"

Edsger Wybe Dijkstra

5 Common Numerical Domains

We have described Abstract Interpretation in the previous chapter without describing specific abstractions leveraged by Abstract Interpretation users. This chapter describes some common numerical abstract domains. Although, there exist non-numerical abstract domains as well, we focus on numerical domains because the approximations involved are easier to relate to, as we compare Abstract Interpretation with other disciplines.

We have described Sign domain- a simple numerical non-relational domain, Polyhedra Domain- a complex relational numerical domain and Template Domain- that presents a template for intervals, octagons and others.

The contents of this chapter are standard material in the study of Abstract Interpretation. We borrow the definitions and examples from Mine's tutorial on the subject [45] and do not claim any original work in this chapter.

5.1 | Sign Abstract Domain

Sign domain is one of the simplest abstract domains. It abstracts numerical values to signs, namely ≤ 0, ≥ 0 or 0. Additionally, it contains a bottom element $\bot^\sharp$ to map undefined/unreachable variable values, dead code etc., and, a top element $\top^\sharp$ that signifies a value of unknown sign (may be either positive, negative or zero).

5.1.1 — Representation

The domain has a small number of possible abstract values: $D^\sharp \stackrel{\text{def}}{=} \{\bot^\sharp,\ \leq 0,\ \geq 0,\ 0,\ \top^\sharp\}$. Consequently, an abstract value $d \in D^\sharp$ may be represented simply with above notation.

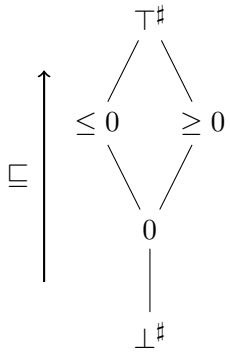


Figure 5.1: Hasse Diagram of Sign Domain $\langle D^\sharp, \sqsubseteq \rangle$

5.1.2 — Order Structure

The hasse diagram for the sign abstract domain is shown in Figure-5.1. The order $\sqsubseteq$ is defined implicitly by the diagram. Next, we define the abstraction operation $\alpha : \wp(\mathbb{Z}) \longrightarrow D^\sharp$ and concretization function $\gamma : D^\sharp \longrightarrow \wp(\mathbb{Z})$. It should be noted that $\langle D^\sharp, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \wp(\mathbb{Z}), \subseteq \rangle$.

$$\alpha(C) \overset{\text{def}}{=} \begin{cases} \bot^\sharp & \text{if } C = \varnothing \\ 0 & \text{if } C = \{0\} \\ (\leq 0) & \text{if } \forall c \in C : c \leq 0 \\ (\geq 0) & \text{if } \forall c \in C : c \geq 0 \\ \top^\sharp & \text{otherwise} \end{cases} \qquad \gamma(d) \overset{\text{def}}{=} \begin{cases} \varnothing & \text{if } d = \bot^\sharp \\ \{0\} & \text{if } d = 0 \\ \{c \in \mathbb{Z} | c \leq 0\} & \text{if } d = (\leq 0) \\ \{c \in \mathbb{Z} | c \geq 0\} & \text{if } d = (\geq 0) \\ \mathbb{Z} & \text{if } d = \top^\sharp \end{cases}$$

5.1.3 — Abstract Operators

The abstract operators for arithmetic operations are described in Table-5.1. Intuitively, one may understand the sign domain abstracts out the absolute values and only retains the sign. Thus, the arithmetic operations are described by rules of signs.

Similarly, assignment operations can be designed by following the sign of the constants or interval bounds.

The abstractions for join and meet (set union and intersection) are exact operations in the sign domain.

$+$	$\bot^\sharp$	0	≤ 0	≥ 0	$\top^\sharp$
$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$
0	$\bot^\sharp$	0	≤ 0	≥ 0	$\top^\sharp$
≤ 0	$\bot^\sharp$	≤ 0	≤ 0	$\top^\sharp$	$\top^\sharp$
≥ 0	$\bot^\sharp$	≥ 0	$\top^\sharp$	≥ 0	$\top^\sharp$
$\top^\sharp$	$\bot^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$

(a) Addition

$\times$	$\bot^\sharp$	0	≤ 0	≥ 0	$\top^\sharp$
$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$
0	$\bot^\sharp$	0	0	0	0
≤ 0	$\bot^\sharp$	0	≥ 0	≤ 0	$\top^\sharp$
≥ 0	$\bot^\sharp$	0	≤ 0	≥ 0	$\top^\sharp$
$\top^\sharp$	$\bot^\sharp$	0	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$

(b) Multiplication

$/$	$\bot^\sharp$	0	≤ 0	≥ 0	$\top^\sharp$
$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$
0	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$
≤ 0	$\bot^\sharp$	0	≥ 0	≤ 0	$\top^\sharp$
≥ 0	$\bot^\sharp$	0	≤ 0	≥ 0	$\top^\sharp$
$\top^\sharp$	$\bot^\sharp$	0	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$

(c) Division

Table 5.1: Abstract Arithmetic Operators for Sign Domain

5.1.4 — Convergence Acceleration

Since sign domain is finite, convergence is guaranteed and no special widening operators are needed.

5.2 | Polyhedra Abstract Domain

The Polyhedra domain is a relational numerical domain. Introduced in 1978 [18] to infer affine inequalities among variables:

$$\bigwedge_j \sum_{i=1}^{|\mathbb{V}|} \alpha_{ij} V_i \geq \beta_j, \qquad \alpha_{ij}, \beta_j \in \mathbb{I} \tag{5.1}$$

Algorithms for this domain are based on the field of linear programming and convex polyhedra. Reader interested in these theories may refer to [56] for proofs and details on results we use here. Since we are working with linear programming, we assume $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$.

5.2.1 — Representation

Equation-5.1 describes a convex, topologically closed polyhedra. The Polyhedron may be bounded or unbounded (eg.$X \geq 0$):

$$\mathcal{D}^\sharp \simeq \{\text{closed convex polyhedra of } \mathbb{P}\}$$

where $\mathbb{P} \stackrel{\text{def}}{=} \mathbb{I}^{|\mathbb{V}|}$.

Polyhedra have two common representations: the constraint representation and generator representation. Chernikova's algorithm [12] and it's modern versions [37] can be used to convert the representations from one form to the other.

Constraint Representation

This representation involves affine inequality constraints in matrix form: $\langle \mathbf{M}, \vec{C} \rangle$ with matrix $\mathbf{M} \in \mathbb{I}^{m \times n}$ and vector $\vec{C} \in \mathbb{I}^m$, where m is number of constraints and n is number of variables $|\mathbb{V}|$. $\langle \mathbf{M}, \vec{C} \rangle$ represents the set:

$$\gamma\Big(\langle \mathbf{M}, \vec{C} \rangle\Big) \stackrel{\text{def}}{=} \{\vec{V} \in \mathbb{P} | \mathbf{M} \times \vec{V} \geq \vec{C}\}$$

The matrix representation is equivalent to the constraint set notation $\{\sum_i \alpha_{ij} V_i \geq \beta_j | j \in [1, m]\}$, and, logical notation: $\bigwedge_{j=1}^{m} \sum_i \alpha_{ij} V_i \geq \beta_j$. We may also use dot product notation: $\{\vec{\alpha}_j . \vec{V} \geq \beta_j | j \in [1, m]\}$ and $\bigwedge_{j=1}^{m} \vec{\alpha}_j . \vec{V} \geq \beta_j$.

Generator Representation

This representation involves *generators*, that is vectors representing vertices or rays. The idea is that a polyhedron is represented by $[P, R]$ where set $P = \{\vec{P}_1, \vec{P}_2, ..., \vec{P}_p\} \subseteq \mathbb{P}$ is a finite set of vertices, and $R = \{\vec{R}_1, \vec{R}_2, ..., \vec{R}_r\} \subseteq \mathbb{P}$ is the finite set of rays. The polyhedron represented by $[P, R]$ is described as:

$$\gamma\big([P, R]\big) \stackrel{\text{def}}{=} \left\{ \left(\sum_{j=1}^{p} \alpha_j \vec{P}_j \right) + \left(\sum_{j=1}^{p} \beta_j \vec{R}_j \right) | \, \forall j : \alpha_j, \beta_j \geq 0, \sum_{j=1}^{p} \alpha_j = 1 \right\}$$

Note that neither generator representation, nor constraint representation is unique. A polyhedra has multiple representations in both constraint and generator representations. We

can also have *minimal representation* in either form by, for instance, removing redundant constraints. However, even minimal representations are not unique. For the empty set, we will always use $\bot^\sharp$.

5.2.2 — Order Structure

The ordering in the abstract $X^\sharp \sqsubseteq Y^\sharp$ corresponds to set inclusion in the concrete:$\gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$. Implementation involves verifying that each generator of $X^\sharp$ satisfies every constraint of $Y^\sharp$:

$$X^\sharp \sqsubseteq Y^\sharp \stackrel{\text{def}}{=} \begin{cases} \forall \vec{P} \in P_{X^\sharp} : \mathbf{M}_{Y^\sharp} \times \vec{P} \geq \vec{C}^\sharp \\ \forall \vec{P} \in P_{X^\sharp} : \mathbf{M}_{Y^\sharp} \times \vec{P} \geq 0 \end{cases}$$

Equality testing involves verifying double inclusion: $X^\sharp = Y^\sharp \iff \left(X^\sharp \sqsubseteq Y^\sharp \wedge Y^\sharp \sqsubseteq X^\sharp\right)$.

Although $\sqsubseteq^\sharp$ is not anti-symmetric, by identifying elements in $D^\sharp$ with the same concretization, we obtain a partial order. We may even derive a lattice $(D^\sharp, \sqsubseteq^\sharp, \cup^\sharp, \cap^\sharp)$, but it will not be complete since the infinite join of polyhedra are discs.

5.2.3 — Abstract Operators

We define the various abstract operators for polyhedra domain.

Meet: Abstract Intersection $\cap^\sharp \stackrel{\text{def}}{=} \sqcap^\sharp$ is obtained by simply combining the constraints. It is exact operation (sound and precise):

$$X^\sharp \sqcap^\sharp Y^\sharp \stackrel{\text{def}}{=} \left\langle \begin{bmatrix} \mathbf{M}_{X^\sharp} \\ \mathbf{M}_{Y^\sharp} \end{bmatrix}, \begin{bmatrix} \vec{C}_{X^\sharp} \\ \vec{C}_{Y^\sharp} \end{bmatrix} \right\rangle$$

Join: The set union of two polyhedra is not necessarily a polyhedra. Hence the abstract union $\cup^\sharp \stackrel{\text{def}}{=} \cap^\sharp$ is necessarily an approximate operation. It involves joining the vertices and rays from the generator representation:

$$X^\sharp \sqcup^\sharp Y^\sharp \stackrel{\text{def}}{=} \left[\left[P_{X^\sharp} \;\; P_{Y^\sharp}\right], \left[R_{X^\sharp} \;\; R_{Y^\sharp}\right]\right]$$

Note that both join and meet may introduce redundant generators or constraints, and hence Chernikova algorithm may be used to remove them.

Abstract Conditions: We can filter states by simply adding the condition to the constraint

representation:

$$\mathbb{C}^\sharp[\![\Sigma_i \alpha_i V_i \geq \beta]\!] X^\sharp \stackrel{\text{def}}{=} \left\langle \begin{bmatrix} \mathbf{M}_{X^\sharp} \\ \alpha_1, ..., \alpha_n \end{bmatrix}, \begin{bmatrix} \vec{C}_{X^\sharp} \\ \beta \end{bmatrix} \right\rangle$$

Abstract Assignments: We use generator representation for assignments. we apply affine transformation to the vertices and the associated linear transformation to rays.
Consider Non-deterministic assignment $V_j \longleftarrow [-\infty, +\infty]$:

$$\mathbb{S}^\sharp[\![V_j \longleftarrow [-\infty, +\infty]]\!] X^\sharp \stackrel{\text{def}}{=} [P_{X^\sharp}, [R_{X^\sharp} \ \ \vec{v}_j \ \ (-\vec{v}_j)]]$$

For Affine Assignments $\mathbb{S}^\sharp[\![V_j \longleftarrow \Sigma_i \alpha_i V_i + \beta]\!] X^\sharp$:

- $\alpha_j \neq 0$: Case of invertible assignment. In every constraint, replace V_j with $(V_j - \Sigma_{i \neq j} \alpha_i V_i - \beta)/\alpha_j$ that expresses old value of V_j as function of new value.
- $\alpha_j = 0$: Case of non-invertible assignment. We forget the value of V_j and add an equality constraint modelled as pair of inequalities:

$$\mathbb{S}^\sharp[\![V_j \longleftarrow \Sigma_i \alpha_i V_i + \beta]\!] X^\sharp \stackrel{\text{def}}{=} \mathbb{C}^\sharp[\![V_j = \Sigma_i \alpha_i V_i + \beta]\!] \circ \mathbb{S}^\sharp[\![V_j \longleftarrow [-\infty, +\infty]]\!]$$

5.2.4 — Convergence Acceleration

Widening is needed since polyhedra domain infinite strictly increasing chains. We use ideas similar to interval widening, except instead of removing bounds, we remove unstable constraints.

Naive Widening:

$$X^\sharp \nabla Y^\sharp \stackrel{\text{def}}{=} \{c \in X^\sharp | Y^\sharp \sqsubseteq \{c\}\} \tag{5.2}$$

This widening does terminate since iterations with widening necessarily as the set of constraints decreases. However, it is not as precise and also is not semantic. Namely, it depends on the choice of constraints used to represent the polyhedra and does not treat the polyhedra as the set of points (the semantics).

Semantic Widening:

$$X^\sharp \nabla Y^\sharp \stackrel{\text{def}}{=} \left\{c \in X^\sharp | Y^\sharp \sqsubseteq \{c\}\right\} \cup \left\{c \in Y^\sharp | \exists c' \in X^\sharp : X^\sharp =^\sharp \left(X^\sharp \setminus \{c'\}\right) \cup \{c\}\right\}$$
$$\text{where } X^\sharp =^\sharp Y^\sharp \text{ means } X^\sharp \sqsubseteq^\sharp Y^\sharp \wedge Y^\sharp \sqsubseteq^\sharp X^\sharp \tag{5.3}$$

This widening takes into account the constraints on both the left and right arguments. $X^\sharp \nabla Y^\sharp$ keeps stable constraints from $X^\sharp$ and also keeps constraints from $Y^\sharp$ if they can be swapped with a constraint from $X^\sharp$ without changing $\gamma(X^\sharp)$. This widening can be proven to be independent of chosen representation [2]

5.3 | Template Domain

The Template domain is a weakly relational domain, in-between in terms of cost and precision between intervals [16] and polyhedra [18]. The template domain, introduced by Sankaranarayanan et al. [55], infers conjunctions of affine inequalities, a constraint set $CS = \mathbf{M} \times \vec{V} \leq \vec{C}$, but unlike the polyhedra domain, only the right-hand side $\vec{C}$ i.e., the upper bounds are inferred. The left-hand side $\mathbf{M}$, i.e., the coefficients of the variables in the constraints, are fixed before the analysis is run, and not inferred during the analysis.

Concerning the expressive power, we can see the zone [43] and the octagon domains [44] (and even the interval domain) as special cases of the template domain, for specific $\mathbf{M}$. However. unlike those domains, the shape of the left-hand side M is not fixed by the domain, but can be configured freely by the user before the analysis.

This is key reason we choose this to be the standard domain we use for the rest of this work. Since Intervals, Zones and Octagons domains are instances of template domain, any results we derive on template domain, are automatically valid for those other domains, modulo the algorithmic cost for operations that are actually representation dependent.

This domain has two unique features. Firstly, its expressiveness is fully parameterized, so that a user can decide on a cost versus precision trade-off within the domain, and also adapt the domain to the requirements of a specific program analysis. Secondly, its algorithmic core is based on linear programming, which is a change from polyhedra based on the double description method, and from zones and octagons based on shortest path closure.

As we use general linear algebra, we assume that $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$.

5.3.1 — Representation

We assume that a matrix $\mathbf{M} \in \mathbb{I}^{m \times n}$ is fixed. $n = \mid \mathbb{V} \mid$ is the number of variables, while m is arbitrary. Intuitively, the set and number of linear expressions on the left-hand of constraints can be freely chosen. We call $\mathbf{M}$ the template. An abstract element $X^\sharp \in D^\sharp$ is given by setting the upper bound of each linear expression, which we store as a m-dimensional vector $\vec{C}$. Note,

however, that we need a way to state that a linear expression in **M** is unbounded, hence we live in $(\mathbb{I} \cup +\infty)^m$. As usual, we also add a $\bot^\sharp$ element, which is a canonical representation of the empty set, thus:

$$D^\sharp \stackrel{\text{def}}{=} \{\bot^\sharp\} \cup (\mathbb{I} \cup \{+\infty\})^m \tag{5.4}$$

and the concretization is naturally:

$$\gamma(\vec{C}) \stackrel{\text{def}}{=} \{\vec{V} \in \mathbb{P} \mid \mathbf{M} \times \vec{V} \leq \vec{C}\} \tag{5.5}$$

As stated above, the template domain generalizes the interval, zone, and octagon domains, which become special cases for a fixed template **M**. More precisely:

- for intervals, $m = 2n$: there is an affine expression V_i and an affine expression $-V_i$ for every variable $V_i \in \mathbb{V}$;
- for zones, $m = n^2 + n$: there is an affine expression $V_j - V_i$ for every $i \neq j$, in addition to the affine expressions representing intervals.

However, from an algorithmic point of view, the domains are implemented quite differently, and are much less efficient than the native interval and zone domains we presented in previous sections. The template domain is useful when **M** remains small but has a complex structure, featuring more varied expressions, out of the scope of octagons.

5.3.2 — Example: Interval Representation

Consider P : `x = 1; y = 0; while x < 10 {x++; y++}`. We use the domain of intervals [16] I to infer the invariant that defines the bounds on x and y. Intervals in two dimensions are conjunction of four lines: $a_1 \leqslant x \leqslant a_2 \wedge a_3 \leqslant y \leqslant a_4$ where integer valued $a_1, a_2, ..., a_4$.

For instance, during the second $(i = 2)$ and third $(i = 3)$ iteration inside the loop (point 2), the respective intervals as below:

$$\begin{array}{rr} point2(i=2) = & point2(i=3) = \\ 1 \leqslant x \leqslant 2 \ \wedge \ 0 \leqslant y \leqslant 1 & 1 \leqslant x \leqslant 3 \ \wedge \ 0 \leqslant y \leqslant 2 \end{array}$$

We may use our constraint system to represent these intervals as below:

$$CS_{i=2} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 2 \\ -1 \\ 1 \\ 0 \end{bmatrix} \qquad CS_{i=3} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 3 \\ -1 \\ 2 \\ 0 \end{bmatrix}$$

5.3.3 — Example: Octagon Representation

Consider P : $\texttt{x} = \texttt{1}; \texttt{y} = \texttt{0};$ $\texttt{while x} < \texttt{10}$ $\{\texttt{x}++; \texttt{y}++\}$. We must use the domain of octagons [44] O to infer the invariant that relates x and y. Octagons are conjunction of eight lines: $a_1 \leqslant x \leqslant a_2 \wedge a_3 \leqslant y \leqslant a_4 \wedge a_5 \leqslant a_6 x + y \leqslant a_7 \wedge a_8 \leqslant -a_6 x + y \leqslant a_9$ where integer valued $a_1, a_2, ..., a_9$.

For instance, during the second $(i = 2)$ and third $(i = 3)$ iteration inside the loop (point 2), our constraint system may be used to describe the respective octagons as below:

$$\begin{aligned} point2(i=2) = \\ 1 \leqslant x \leqslant 2 \ \wedge \ 0 \leqslant y \leqslant 1 \\ 1 \leqslant x + y \leqslant 3 \\ -1 \leqslant -x + y \leqslant -1 \end{aligned} \qquad \begin{aligned} point2(i=3) = \\ 1 \leqslant x \leqslant 3 \ \wedge \ 0 \leqslant y \leqslant 2 \\ 1 \leqslant x + y \leqslant 5 \\ -1 \leqslant -x + y \leqslant -1 \end{aligned}$$

We may use our constraint system to represent these octagons as below:

$$CS_{i=2} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 2 \\ -1 \\ 1 \\ 0 \\ 3 \\ -1 \\ -1 \\ 1 \end{bmatrix} \qquad CS_{i=3} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 3 \\ -1 \\ 2 \\ 0 \\ 5 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

5.3.4 — Order Structure

We extend the natural total order on bounds $(\mathbb{I} \cup \{+\infty\}, \leq)$ to vectors, pointwise, to get our partial order $\sqsubseteq^\sharp$ on $D^\sharp$. We actually get a lattice structure $(D^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$, where $\sqcup^\sharp$

is the point-wise maximum, $\sqcap^\sharp$ is the point-wise minimum, and $\top^\sharp$ maps all affine expressions to $+\infty$. The lattice is complete if $\mathbb{I} = R$, and we can define an abstraction function. It associates to each affine expression $\vec{M_i}$ at line i in $\mathbf{M}$, the tightest upper bound:

$\forall i \leq m : [\alpha(S)]_i = max\{\vec{M_i}.\vec{V} \mid \vec{V} \in S\}$ when $S \neq \emptyset$ and $\alpha(S) = \bot^\sharp$ otherwise

5.3.5 — Normalization

The concretization γ is not one-to-one: there exist different abstract elements that represent the same polyhedron. We define a normal form $\vec{C}^*$ the constraints as much as possible, until they saturate (i.e., touch) the polyhedron $\gamma(\vec{C})$. We have indeed $\vec{C}^* = \alpha(\gamma(\vec{C}))$.

The normal form can be effectively computed using linear programming LP:

$$\begin{aligned}
&\forall i \leq m : [\vec{C}^*]_i \stackrel{\text{def}}{=} LP(\langle \mathbf{M}, \vec{C} \rangle, \vec{M_i}) \\
&\text{where} \\
&LP(\langle \mathbf{M}, \vec{C} \rangle, \vec{V}) \stackrel{\text{def}}{=} \max\{\vec{P}.\vec{V} \mid \mathbf{M} \times \vec{P} \leq \vec{C}\}
\end{aligned} \tag{5.6}$$

Additionally, standard LP algorithms are able to determine whether the set of affine constraints $\mathbf{M} \times \vec{V} \leq \vec{C}$ is satisfiable and, if it is not, we return $\bot^\sharp$

5.3.6 — Abstract Operators

Using the normal form, we can decide equality and inclusion exactly:

$$\begin{aligned}
\gamma(X^\sharp) = \gamma(Y^\sharp) &\iff X^{\sharp *} = Y^{\sharp *} \\
\gamma(X^\sharp) \subseteq \gamma(Y^\sharp) &\iff X^{\sharp *} \sqsubseteq Y^{\sharp *}
\end{aligned} \tag{5.7}$$

We can model the union and the intersection by taking, respectively, for each affine expression, the loosest or the strictest of the constraints from both arguments. The intersection $\cap^\sharp \stackrel{\text{def}}{=} \sqcap^\sharp$ is exact. The union $\cup^\sharp$ is not exact and, for it to be optimal, we must use the normal form, i.e., we state $X^\sharp \cup^\sharp Y^\sharp \stackrel{\text{def}}{=} X^{\sharp *} \sqcup^\sharp Y^{\sharp *}$. The result is naturally in normal form.

Tests, conditions, assignments are not of interest to us, so we skip those for now.

5.3.7 — Convergence Acceleration

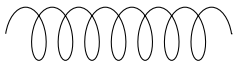
The template domain features infinite increasing and decreasing chains. It is composed of a finite number of bounds of fixed affine expressions. We can thus construct widenings and

narrowing independently on each bound. We define:

$$\forall i \leq m : [\vec{C} \nabla \vec{D}]_i \stackrel{\text{def}}{=} \begin{cases} C_i & \text{if } C_i \geq D_i \\ +\infty & \text{otherwise} \end{cases} \tag{5.8}$$

$$\forall i \leq m : [\vec{C} \triangle \vec{D}]_i \stackrel{\text{def}}{=} \begin{cases} D_i & \text{if } C_i = +\infty \\ C_i & \text{otherwise} \end{cases} \tag{5.9}$$

Finally, note that the result of the widening should not be put in normal form between two iterations, as it may jeopardize the convergence.



"When you're fundraising, it's AI / When you're hiring, it's ML / When you're implementing, it's linear regression / When you're debugging, it's printf()"

Baron Schwartz

6 Supervised Learning

This chapter discusses the fundamentals of Machine Learning. We start from the basics, leading upto formal theoretical models for learning and finally present some algorithms used commonly in practise. The goal is to build a foundation to support easier reading and understanding of the dissertation, rather than providing a full or in-depth course on the subject matter.

We follow the notations and formalizations of Shalev-Shwartz and Ben-David [59]. We borrow most of the definitions and examples from the same book. Some definitions and examples are also borrowed from the book of Bishop [4]. To the readers interested in reading more about theoretical machine learning, we recommend the book by Shalev [59]. Those interested in more practical aspects, or a more in depth look at the probabilistic approach to machine learning, we recommend the book by Bishop [4].

6.1 | Learning Fundamentals

6.1.1 — What is Machine Learning ?

Informally, Automated Learning, or Machine Learning (ML) is the process by which computers turn "experience" into "knowledge". By "experience", we refer to the input of the learner (machine learning program), and by "knowledge" we refer to another computer program that generalizes the input experience to better perform a "task".

> *"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*
>
> ——Tom M. Mitchell

For instance, consider the archtypical example where we want to learn how to predict if a choco-chip cookie is tasty or not (the task). First we decide on the *features* that our prediction will be based on: say Radius of the cookie, and choco-chip density (number of choco-chips per

unit area). The input is a sample of choco-chip cookies. We analyze for the features and then taste. We now analyze this learning task.

To be more formal, we begin by introducing some terminology as it relates to the problem:

- The Domain set $\mathcal{X}$: An arbitrary set $\mathcal{X}$ that we may wish to learn about (the choco-chip cookies). The domain *instances* are represented by a *feature* (radius and choco-chip density) vector.

- The Label set $\mathcal{Y}$: The possible labels. For our example, $\mathcal{Y} = \{0, 1\}$ where 0 implies not tasty, and 1 implies tasty.

- Training Data S: We describe $S = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ as a finite sequence of m labeled data-points in $\mathcal{X} \times \mathcal{Y}$. This is the sampled cookies that we tasted to find out if they were tasty or not. Note that despite the name "set", S is a sequence, implying the same data-point may repeat, and also some algorithms take the order of the sequence into account.

- Prediction Rule h: The learner learns $h : \mathcal{X} \longrightarrow \mathcal{Y}$. h may also be referred as a hypothesis, or a classifier. We use the notation $A(S)$ to denote the hypothesis that a learning algorithm, A, returns upon receiving the training sequence S.

- Data-generation Model: We assume that instances are generated by some arbitrary probability distribution $\mathcal{D}$ over $\mathcal{X}$. We also assume that there exists a true labeling function $f : \mathcal{X} \longrightarrow \mathcal{Y}$. Pairs in S are generated by first sampling from $\mathcal{X}$ using $\mathcal{D}$ then applying f.

- Measures of success: Success is actually measured by error, where error is the probability that the predictor makes incorrect predictions: Probability that for a random instance x drawn according to distribution $\mathcal{D}$, $h(x) \neq f(x)$.
  Thus, given some $A \subset \mathcal{X}$ (more precisely, A in some σ-algebra on $\mathcal{X}$), $\mathcal{D}(A)$ is the probability of the event A. Therefore, $A = \{x \in \mathcal{X} : \pi(x) = 1\}$ where $\pi : \mathcal{X} \longrightarrow \{0, 1\}$. We may also denote $\mathcal{D}(A)$ as $\mathbb{P}_{x\sim\mathcal{D}}[\pi(x)]$.
  Error of Prediction rule $h : \mathcal{X} \longrightarrow \mathcal{Y}$, denoted by $\mathbb{L}_{(\mathcal{D},f)}$, and sometimes also referred to as *generalization error*, *true error*, or *risk* is defined as:

$$\mathbb{L}_{(\mathcal{D},f)}(h) \stackrel{\text{def}}{=} \mathop{\mathbb{P}}_{x\sim\mathcal{D}}[h(x) \neq f(x)] \stackrel{\text{def}}{=} \mathcal{D}\{x : h(x) \neq f(x)\} \tag{6.1}$$

Thus, we can describe a learner A as a program that takes as input the domain set $\mathcal{X}$, the training data S and the label set $\mathcal{Y}$. It is blind to the distribution $\mathcal{D}$ and the true labeling function $f : \mathcal{X} \longrightarrow \mathcal{Y}$. The goal is to approximate f with a hypothesis $h_S : \mathcal{X} \longrightarrow \mathcal{Y}$ while minimizing the error $\mathbb{L}_{(\mathcal{D},f)}$. The subscript S denotes that the hypothesis was learned using training set S.

6.1.2 — Emperical Risk Minimization

As a first learning strategy, we look into solving for h that minimizes the generalization error. Since we do not know the $\mathcal{D}$ and f, the true error is not calculable.

Hence, we use another useful notion of error: the *emperical risk*, also referred to as the *emperical error* or the *training error*, described as:

$$\mathbb{L}_S(h) \overset{\text{def}}{=} \frac{|\{i \in [m] : h(x_i) \neq f(x_i)\}|}{m} \tag{6.2}$$

where $[m] = \{1, 2, ..., m\}$.

The learning strategy to derive the predictor $h : \mathcal{X} \longrightarrow \mathcal{Y}$ that minimizes the emperical risk is called *Emperical Risk Minimization*.

However, this approach suffers from the problem of overfitting. Namely, the strategy may give a predictor that fits perfectly with the training data, but performs poorly in the real world. This may be so because the predictor (h) only minimizes $\mathbb{L}_S(h)$, but not $\mathbb{L}_{(\mathcal{D},f)}(h)$.

One way to address the challenge of overfitting is to restrict the search space. Namely, instead of searching all possible predictors, we restrict the search to a finite set of hypothesis $\mathcal{H}$. That is:

$$\mathsf{ERM}_{\mathcal{H}}(S) \in \operatorname*{\mathsf{argmin}}_{h \in \mathcal{H}} \mathbb{L}_S(h)$$

where argmin filters $\mathcal{H}$ for hypothesis that achieve minimum $\mathbb{L}_S(h)$ over $\mathcal{H}$. These restrictions cause the learner to *bias* towards a particular set of hypothesis, and hence are referred to as *inductive bias*.

Intuitively, a more restrictive hypothesis class lowers the risk of overfitting, while simultaneously increases the inductive bias. Thus, there exist a tradeoff bias vs overfitting. We discuss this later.

This brings us to a fundamental question in Machine Learning: *How to restrict a hypothesis class $\mathcal{H}$ so that $\mathsf{ERM}_{\mathcal{H}}$ does not overfit ?*

It can be shown that any finite hypothesis class $\mathcal{H}$ will not overfit if the Realizability (Definition-6.1.1) and the i.i.d. (Definition-6.1.2) assumptions hold, and the training set is large enough (Theorem-6.1.3).

Definition 6.1.1 (Realizability Assumption)**.** There exists $h^\star \in \mathcal{H}$ s.t. $\mathbb{L}_{(\mathcal{D},f)}(h^\star) = 0$.

Definition 6.1.2 (The i.i.d Assumption)**.** The data-points in the training set are independently and identically distributed (i.i.d.) according to the distribution $\mathcal{D}$. Namely, $S \sim \mathcal{D}^m$.

Theorem 6.1.3. *For any labeling function, f , and for any distribution, $\mathcal{D}$, for which the realizability assumption (Definition-6.1.1) holds, then with probability of at least $(1-\delta)$, where $\big(\delta \in (0,1)\big)$, over the choice of an i.i.d. sample S of size $m \geq \frac{\log(|H|/\delta)}{\epsilon}$, we have that for every **ERM** hypothesis, h_S , it holds that:*

$$\mathbb{L}_{(\mathcal{D},f)}(h_S) \leq \epsilon \qquad \text{where } \epsilon > 0$$

The Theorem-6.1.3 says that for a sufficiently large m, the $\mathsf{ERM}_{\mathcal{H}}$ rule over a finite hypothesis class, under the realizability and iid assumptions, will be probably (with confidence $1-\delta$) approximately (up to an error of ϵ) correct.

Definition 6.1.4 (PAC Learnability)**.** A hypothesis class $\mathcal{H}$ is PAC learnable if there exist a function $m_{\mathcal{H}} : (0,1)^2 \longrightarrow \mathbb{N}$ and a learning algorithm with the following property: For every $\epsilon, \delta \in (0,1)$, for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \longrightarrow \{0,1\}$, if the realizable assumption holds with respect to $\mathcal{H}, \mathcal{D}, f$, then when running the learning algorithm on $m \geq m_{\mathcal{H}}(\epsilon,\delta)$ i.i.d. examples generated by $\mathcal{D}$ and labeled by f, the algorithm returns a hypothesis h such that, with probability of at least $1-\delta$ (over the choice of the examples), $\mathbb{L}_{(\mathcal{D},f)}(h) \leq \epsilon$.

The definition of PAC includes two approximation parameters-δ and ϵ for confidence and accuracy. The accuracy parameter ϵ describes the distance between learned hypothesis h and true labeling function f, while the confidence parameter δ indicates the probability of hypothesis h satisfying the accuracy condition.

6.1.3 — Agnostic PAC: A more general approach

In practical applications, the Realizability Assumption of Definition-6.1.1 can be too restrictive. Futhermore, the feature set may not be complete. Thus, the realizability assumption may be relaxed in favor of a probabilistic labeling function. Namely, from this point on, we consider $\mathcal{D}$ to be a joint probability distribution over $\mathcal{X} \times \mathcal{Y}$.

Consequently, we now revise the true risk of hypothesis h:

$$\mathbb{L}_{\mathcal{D}}(h) \overset{\text{def}}{=} \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}}[h(x) \neq y] \overset{\text{def}}{=} \mathcal{D}(\{(x,y) : h(x) \neq y\}) \tag{6.3}$$

Note that the emperical risk function did not change:

$$\mathbb{L}_S(h) \overset{\text{def}}{=} \frac{|\{i \in [m] : h(x_i) \neq y_i\}|}{m} \tag{6.4}$$

where $[m] = \{1, 2, ..., m\}$.

As before, the goal is to find the predictor $h : \mathcal{X} \longrightarrow \mathcal{Y}$ that probably approximately minimizes the true risk $\mathbb{L}_{\mathcal{D}}$, however, with $\mathcal{D}$ being unknown, we may only calculate the emperical risk.

Indeed, if we knew the distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, then the Bayes predictor would be the optimal predictor:

$$f_{\mathcal{D}} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } \mathbb{P}[y = 1|x] \geq 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

Clearly, no learning algorithm may find a hypothesis better than the bayes optimal predictor. As such, we try to bound the distance between learned hypothesis and the optimal predictor:

Definition 6.1.5 (Agnostic PAC Learning)**.** A hypothesis class $\mathcal{H}$ is agnostic PAC learnable if there exist a function $m_{\mathcal{H}} : (0,1)^2 \longrightarrow \mathbb{N}$ and a learning algorithm with the following property: For every $\epsilon, \delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, when running the learning algorithm on $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. examples generated by $\mathcal{D}$, the algorithm returns a hypothesis h such that, with probability of at least $1 - \delta$ (over the choice of the m training examples),

$$\mathbb{L}_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} \mathbb{L}_{\mathcal{D}}(h') + \epsilon$$

Clearly when the realizability assumption holds, the Agnostic PAC leaner gives the same guarantees as the PAC learner. The model may be generalized by further by allowing $\mathcal{Y}$ to be larger finite set or even the set of all reals $\mathbb{R}$. A larger, but finite, set $\mathcal{Y}$ is useful for Multiclass Classification problems, while the continuous set $\mathcal{Y}$ is applied in Regression problems.

We thus generalize the loss function $\ell : \mathcal{H} \times \mathbb{Z} \longrightarrow \mathbb{R}_+$ where $\mathbb{Z}$ is some domain determined by the application. Note that for prediction problems we have $\mathbb{Z} = \mathcal{X} \times \mathcal{Y}$, however,

that may not be the case for all applications.

We define two of the most commonly used loss functions here:

- 0-1 Loss: Here $\mathbb{Z} : \mathcal{X} \times \mathcal{Y}$.

$$\ell_{0-1}\big(h, (x, y)\big) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } h(x) = h(y) \\ 1 & \text{if } h(x) \neq h(y) \end{cases} \tag{6.6}$$

- Squared Loss: Once again, $\mathbb{Z} : \mathcal{X} \times \mathcal{Y}$.

$$\ell_{sq}\big(h, (x, y)\big) \stackrel{\text{def}}{=} \big(h(x) - y\big)^2 \tag{6.7}$$

Definition 6.1.6 (Uniform Convergence)**.** We say that a hypothesis class $\mathcal{H}$ has the uniform convergence property (w.r.t. a domain $\mathbb{Z}$ and a loss function ℓ) if there exists a function $m_{\mathcal{H}}^{UC} : (0, 1) \longrightarrow \mathbb{N}$ such that for every $\epsilon, \delta \in (0, 1)$ and for every probability distribution $\mathcal{D}$ over $\mathbb{Z}$, if S is a sample of $m \geq m_{\mathcal{H}}^{UC}(\epsilon, \delta)$ examples drawn i.i.d. according to $\mathcal{D}$, then, with probability of at least $1 - \delta$, S is ϵ-representative. Namely,

$$\forall h \in \mathcal{H} : |\mathbb{L}_S(h) - \mathbb{L}_{\mathcal{D}}(h)| \leq \epsilon$$

It can be shown that for finite $\mathcal{H}$, and $\ell : \mathcal{H} \times \mathbb{Z} \longrightarrow \{0, 1\}$, the uniform convergence property holds for:

$$m_{\mathcal{H}}^{UC}(\epsilon, \delta) \leq \left\lceil \frac{\log(2|\mathcal{H}|/\delta)}{2\epsilon^2} \right\rceil$$

Theorem 6.1.7 (UC Property implies Agnostic PAC Learnability)**.** *If a class $\mathcal{H}$ has the uniform convergence property with a function $m_{\mathcal{H}}^{UC}$ then the class is agnostically PAC learnable with the sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leq m_{\mathcal{H}}^{UC}(\epsilon/2, \delta)$. Furthermore, in that case, the $\textsf{ERM}_{\mathcal{H}}$ paradigm is a successful agnostic PAC learner for $\mathcal{H}$.*

6.1.4 — Bias Variance Tradeoffs

In Section-6.1.2 we discussed the need to restrict the hypothesis class $\mathcal{H}$ to avoid overfitting. The idea was that without some prior knowledge, the learner would overfit on the training set S and not generalize. Thus, we were imposing some prior belief about the distribution $\mathcal{D}$ with our choice of selected class of Hypothesis $\mathcal{H}$. However, too restrictive hypothesis class also implies an induced bias error. Hence, there exist a tradeoff. In this section we formalize these notions.

We begin by describing the No Free Lunch Theorem that states proves the impossibility of the existence of a universal learner.

Theorem 6.1.8 (No Free Lunch)**.** *Let A be any learning algorithm for the task of binary classification with respect to the $0-1$ loss over a domain $\mathcal{X}$. Let m be any number smaller than $|\mathcal{X}|/2$, representing a training set size. Then, there exists a distribution $\mathcal{D}$ over $\mathcal{X} \times \{0,1\}$ such that:*

- *There exists a function $f : \mathcal{X} \longrightarrow \{0,1\}$ with $\mathbb{L}_{\mathcal{D}}(f) = 0$.*
- *With probability of at least $1/7$ over the choice of $S \sim \mathcal{D}^m$ we have that $\mathbb{L}_{\mathcal{D}}\big(A(S)\big) \geq 1/8$.*

This theorem implies that for every learner, there exists a task on which it fails, while there exists another learner that will succeed.

No Free Lunch theorem implies that an ERM predictor chosen over an infinite hypothesis class, without prior knowledge, that is over all possible f, will fail on some learning task. Hence it is not a PAC Learnable class.

Corollary 6.1.9. Let $\mathcal{X}$ be an infinite domain set and let $\mathcal{H}$ be the set of all functions from $\mathcal{X}$ to $\{0,1\}$. Then, $\mathcal{H}$ is not PAC learnable.

Thus, we must restrict the hypothesis class. However, we also do not want to bias the learner to hypothesis class. On the other hand, we also cannot learn without any prior knowledge. Hence, we need to balance this tradeoff.

To better understand the error induced by restricting and not-restricting the hypothesis class, we begin by decomposing the learning error into *Approximation Error* and *Estimation Error*: Let h_S be an $\mathsf{ERM}_{\mathcal{H}}$ hypothesis. Then,

$$\mathbb{L}_{\mathcal{D}}(h_S) = \epsilon_{\text{app}} + \epsilon_{\text{est}} \quad \text{where } \epsilon_{\text{app}} = \min_{h \in \mathcal{H}} \mathbb{L}_{\mathcal{D}}(h),\ \epsilon_{\text{est}} = \mathbb{L}_{\mathcal{D}}(h_S) - \epsilon_{\text{app}} \tag{6.8}$$

- Approximation Error: This measures the *inductive bias*, namely the error induced as a consequence of restricting ourselves to the domain. Under the Realizability Assumption, this is 0. In general, this error increases with reduction in size and complexity of hypothesis class $\mathcal{H}$.

- Estimation Error: This is the error induced as a consequence of using the training error (emperical risk) to estimate the true risk. This error decreases with increase in the training set size $|m|$, and, increases with the increases in size and complexity of hypothesis class $\mathcal{H}$.

The equation-6.8 demonstrates the *bias-variance tradeoff*. Namely, we want to reduce the total risk, but reducing the complexity of $\mathcal{H}$ reduces the estimation error, but increases the estimation error. Hence, a balance must be achieved to avoid underfitting (bias) and overfitting (variance). A great choice for $\mathcal{H}$ would be containing only the Bayes optimal classifier, however, that classifier depends on the unknown distribution $\mathcal{D}$, which we do not know. Indeed learning would have been unnecessary if we knew the distribution $\mathcal{D}$.

6.1.5 — VC Dimension

The error of the $\mathsf{ERM}_{\mathcal{H}}$ rule depends on the choice of hypothesis class $\mathcal{H}$. Indeed the approximation error depends on how well our choice of $\mathcal{H}$ fits with the underlying distribution $\mathcal{D}$. In contrast, the definition of PAC learnability requires that estimation error be bounded uniformly over all possible $\mathcal{D}$.

The question is then which classes are learnable, and how to characterize the sample complexity of learning a given hypothesis class.

It can be shown that while finiteness of a hypothesis class is a sufficient condition for learnability, it is not a necessary condition. We describe now the VC-Dimension, a property of the hypothesis class that gives a correct characterization of it's learnability.

Definition 6.1.10 (Shattering)**.** Hypothesis class $\mathcal{H}$ shatters a finite set $C = \{x_1, ..., x_m\} \subset \mathcal{X}$ if, the set of functions from C to $\{0, 1\}$ that can be derived from $\mathcal{H}$, denoted $\mathcal{H}_C$, is the set of all functions $C \longrightarrow \{0, 1\}$. That is, $|\mathcal{H}_C| = 2^{|C|}$.

Definition 6.1.11 (VC Dimension)**.** The VC-dimension of a hypothesis class $\mathcal{H}$, denoted VCdim($\mathcal{H}$), is the maximal size of a set $C \subset \mathcal{X}$ that can be shattered by $\mathcal{H}$. If $\mathcal{H}$ can shatter sets of arbitrarily large size we say that $\mathcal{H}$ has infinite VC-dimension.

Theorem 6.1.12. *Let $\mathcal{H}$ be a class of infinite VC-dimension. Then, $\mathcal{H}$ is not PAC learnable.*

For finite hypothesis class $\mathcal{H}$ and any set C, we have $|\mathcal{H}_C| \leq |\mathcal{H}|$. Thus C cannot be shattered if $|\mathcal{H}| < 2^{|C|}$. Therefore, VCdim($\mathcal{H}$)$\leq \log_2(|\mathcal{H}|)$.

Often VCdim is equal to the number of parameters, however, this is not always the case. For instance, for domain $\mathcal{X} = \mathbb{R}$, Hypothesis class $\mathcal{H} = \{h_\theta : \theta \in \mathbb{R}\}$ where $h_\theta : \mathcal{X} \longrightarrow \{0, 1\}$ is defined by $h_\theta(x) = \lceil 0.5 sin(\theta x) \rceil$. It can be shown that VCdim($\mathcal{H}$)$= \infty$.

Theorem-6.1.12 implies that hypothesis classes with infinite VC-dim are not learnable. We present below the fundamental theorem of statistical learning that says the converse is also true, namely, finite VCdim is sufficient condition for learnability of a hypothesis class.

Theorem 6.1.13 (Fundamental Theorem of Statistical Learning)**.** *Let $\mathcal{H}$ be a hypothesis class of functions from a domain $\mathcal{X}$ to $\{0,1\}$ and let the loss function be the $0-1$ loss. Then, the following are equivalent:*

1. *$\mathcal{H}$ has the uniform convergence property*
2. *Any **ERM** rule is a successful agnostic PAC learner for $\mathcal{H}$*
3. *$\mathcal{H}$ is agnostic PAC learnable*
4. *$\mathcal{H}$ is PAC learnable*
5. *Any **ERM** rule is a successful PAC learner for $\mathcal{H}$.*
6. *$\mathcal{H}$ has a finite VC-dimension*

Theorem 6.1.14 (Fundamental Theorem of Statistical Learning - Quantitative Version)**.** *Let $\mathcal{H}$ be a hypothesis class of functions from a domain $\mathcal{X}$ to $\{0,1\}$ and let the loss function be the $0-1$ loss. Assume that VCdim($\mathcal{H}$)= $d < \infty$. Then, there are absolute constants C_1, C_2 such that:*

1. *$\mathcal{H}$ has the uniform convergence property with sample complexity:*

$$C_1 \frac{d + \log(1/\delta)}{\epsilon^2} \leq m_{\mathcal{H}}^{UC}(\epsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\epsilon^2}$$

2. *$\mathcal{H}$ is agnostic PAC learnable with sample complexity:*

$$C_1 \frac{d + \log(1/\delta)}{\epsilon^2} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\epsilon^2}$$

3. *$\mathcal{H}$ is PAC learnable with sample complexity:*

$$C_1 \frac{d + \log(1/\delta)}{\epsilon} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{d \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

6.2 | Learning in Practise

Given a training set of data: (Input,Output) pairs (x, y) where $x \in X, y \in Y$, learn a *hypothesis* function $h : X \longrightarrow Y$ such that $h(x)$ is a good predictor of *target variable* y. As previously discussed, when Y is a set of continuous values, we call the problem a *regression* problem; when the set of target values Y is discrete, we call it a *classification* problem.

6.2.1 — Linear Regression

Consider the data shown in Table-6.1 that lists the poverty rate, which is the percent of the state's population living in households with incomes below the federally defined poverty level and the corresponding birth rate per 1000 females 15 to 17 years old.

A typical supervised machine learning problem is to figure out a relationship between the poverty rate and birth rate, given the data in the Table-6.1, that can then be used to predict the birth rates in other areas, when the poverty rate is known.

Figure-6.1 shows a scatter plot of the data in Table-6.1. We must define a computer representation of the hypotheses. For simplicity, let's say we decide to approximate birth rate (y) as a linear function of poverty rate (x):

$$h_\theta(x) = \theta_0 + \theta_1 x$$

In general, the domain of Linear Hypothesis from the space of input features X to target Variable Y is parameterized by the *parameters* θ_j's. Hence the general form of the equation:

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^T x \tag{6.9}$$

where n is the number of input features, $x_0 = 1$ is the intercept term. Note that the form on the right hand side treats both θ and x as vectors.

We compute the parameters θ by minimizing the distance between $h(x)$ and y for the training data. This is achieved by defining a **cost function** $J(\theta)$, that, given a θ, computes the distance between $h(x^{(i)})$'s and corresponding $y^{(i)}$'s. Thus reducing the problem to an optimization problem, where the goal is to compute θ that minimizes the cost function (also sometimes referred to as the *loss function*) $J(\theta)$.

Location	Poverty Rate	Brth15to17
Alabama	20.1	31.5
Alaska	7.1	18.9
Arizona	16.1	35
Arkansas	14.9	31.6
California	16.7	22.6
Colorado	8.8	26.2
Connecticut	9.7	14.1
Delaware	10.3	24.7
District of Columbia	22	44.8
Florida	16.2	23.2
Georgia	12.1	31.4
Hawaii	10.3	17.7
Idaho	14.5	18.4
Illinois	12.4	23.4
Indiana	9.6	22.6
Iowa	12.2	16.4
Kansas	10.8	21.4
Kentucky	14.7	26.5
Louisiana	19.7	31.7
Maine	11.2	11.9
Maryland	10.1	20
Massachusetts	11	12.5
Michigan	12.2	18
Minnesota	9.2	14.2
Mississippi	23.5	37.6
Missouri	9.4	22.2
Montana	15.3	17.8
Nebraska	9.6	18.3
Nevada	11.1	28
New Hampshire	5.3	8.1
New Jersey	7.8	14.7
New Mexico	25.3	37.8
New York	16.5	15.7
North Carolina	12.6	28.6
North Dakota	12	11.7
Ohio	11.5	20.1
Oklahoma	17.1	30.1
Oregon	11.2	18.2
Pennsylvania	12.2	17.2
Rhode Island	10.6	19.6
South Carolina	19.9	29.2
South Dakota	14.5	17.3
Tennessee	15.5	28.2
Texas	17.4	38.2
Utah	8.4	17.8
Vermont	10.3	10.4
Virginia	10.2	19
Washington	12.5	16.8
West Virginia	16.7	21.5
Wisconsin	8.5	15.9
Wyoming	12.2	17.7

Table 6.1: Training Data for learning Birth Rates per 1000 females that are 15 to 17 year old, as a function of Poverty Rate of US States

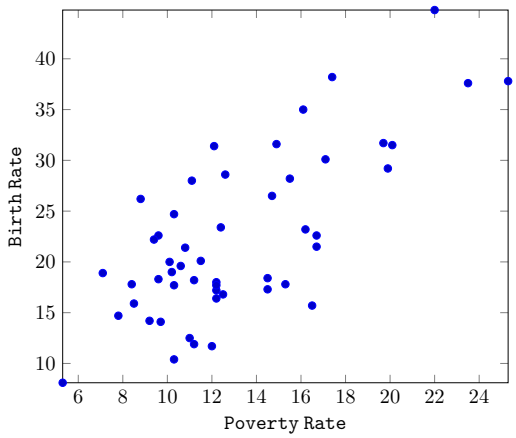


Figure 6.1: Plot of Training Data for learning Birth Rates per 1000 females that are 15 to 17 year old, as a function of Poverty Rate of US States

6.2.2 — Gradient Descent

Gradient Descent is a first order iterative optimization algorithm for finding the minimum of a function. The idea is to start with some initial θ and:

repeat until convergence: {

$$\theta_j := \theta_j - \beta \frac{\partial}{\partial \theta_j} J(\theta) \qquad \text{(for every j)}$$

} (6.10)

where β is called the **learning rate**. The algorithm basically takes repeated steps in the direction of the steepest decrease of $J(\theta)$.

A common choice for the loss function is the *least squared cost function* defined as:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{6.11}$$

Substituting this $J(\theta)$ into Equation-6.10 and expanding:

$$\theta_j := \theta_j - \beta \frac{\partial}{\partial \theta_j} \left(\frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \right)$$

$$\theta_j := \theta_j - \frac{1}{2}.2.\beta \sum_{i=1}^{m} \left(\left(h_\theta(x^{(i)}) - y^{(i)} \right) \frac{\partial \left(h_\theta(x^{(i)}) - y^{(i)} \right)}{\partial \theta_j} \right)$$

substituting the expansion for $h_\theta(x^{(i)})$ from Equation-6.9 into the above for partial derivation:

$$\theta_j := \theta_j - \beta \sum_{i=1}^{m} \left(\left(h_\theta(x^{(i)}) - y^{(i)} \right) \frac{\partial \left(\sum_{j=0}^{n} \theta_j x_j^{(i)} - y^{(i)} \right)}{\partial \theta_j} \right)$$

$$\theta_j := \theta_j - \beta \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right).x_j^{(i)}$$

$$\theta_j := \theta_j + \beta \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)}) \right).x_j^{(i)} \tag{6.12}$$

The update rule of Equation-6.12, derived by using Mean Squared Loss (Equation-6.11) in Gradient Descent (Equation-6.10) is also known as the *least mean squared* algorithm. Also, note

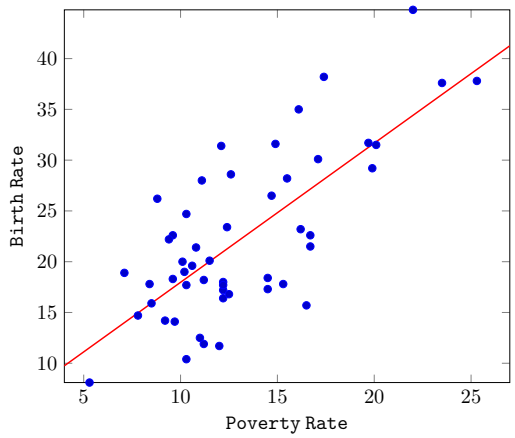


Figure 6.2: Linear Regression on Data for learning Birth Rates per 1000 females that are 15 to 17 year old, as a function of Poverty Rate of US States

that this method looks at every example in the training set on every step, hence the name **batch gradient descent**.

An alternative method is to cycle through the training data-set and for-each training example, update the parameters (θ_j's) according to the gradient of the error with respect to that one single training example only. This is known as **stochastic gradient descent** or **incremental gradient descent**:

repeat until convergence: {

for $i = 1$ to m {

$$\theta_j := \theta_j + \beta(y^{(i)} - h_\theta(x^{(i)})).x_j^{(i)} \qquad \text{(for every j)}$$

}

} (6.13)

In the general case, gradient descent is susceptible to local minima. However, the optimization problem we have here is attempting to minimize a convex quadratic function (Least Squared Cost Function), which has only one global minima and no other local optima. Thus, this gradient descent will always converge to the global minima (assuming the learning rate α is not too large).

Applying batch gradient descent to the data in Table-6.1, we get $\theta_0 = 4.27$ and $\theta_1 = 1.37$. Figure-6.2 shows the resultant plot.

"For Wiener, entropy was a measure of disorder; for Shannon, of uncertainty. Fundamentally, as they were realizing, these were the same"

James Gleick

7 Information Oriented Model of Computation

7.1 | Motivation

In Chapter-3, we presented a new way to systematize Program Analysis techniques in a cube (Figure-3.3). This allowed a new insight that Abstract Interpretation and Machine Learning are two base techniques for program analysis. And that other methods maybe derived by a combination of various abstract interpreters and learning approaches.

Next, as we described in Chapter-4 through Chapter-6, both techniques leverage abstractions to efficiently compute approximate solutions.

This brings us to the question: *Can we compare these techniques?* More specifically, the problem: *How does one compare abstractions used by Abstract Interpretation and Machine Learning?* A rather more general version of this problem would be: *Given two techniques for approximating undecidable problems, How should one approach comparing their abstractions/approximations?*

One way to make such a comparison would be to have a more generalized framework that allows to instantiate these problems. It is reasonable to expect such that such a framework would help not only in better understanding of the general problems solved by these techniques, but also with promoting cross-disciplinary use of ideas, tools and techniques, whenever possible.

We posit two key requirements for such a framework:

1. Generalization Requirement (Req-1): Notice that both Abstract Interpretation, and Machine Learning, are considered to be very generic techniques, applicable to a large body of problems. Thus, a framework that instantiates such generic techniques, needs to be abstract enough to allow instantiating any computation problem.

2. Information Comparison Requirement (Req-2): Since our stated goal with the framework is to be able to compare the approximations, the framework would need to take a more information oriented approach that permits drawing conclusions on the *need* and *level* of approximations being made.

To that end, in this chapter, we present a framework that bridges Abstract Interpretation with Information Theory to create an Information Oriented Model of Computation. Abstract Interpretation allows for controlled loss of information, namely, explicit specification of abstractions that lets control *what* information is being lost. Information Theory allows explicit measurement of the amount of information available, namely *how much* information is available.

Recall from our introductory discussion in Chapter-1, that the Turing Model of Computation views computations as proofs and non-computability as impossibility of proving a truth. The Information Theory model views computation as exchange of information between a source and a target, and non-computability as the impossibility of exchanging certain information. Additionally, Information Theory also provides a rich mathematical framework for making explicit measurements on information.

Thus by building our model of computation atop the Information Theory model, allows us to compare the amount of information in inputs and desired outputs for a problem and explain:

- if the problem is decidable,
- if the problem is only partially decidable, or,
- if only probabilistic inferences may be drawn.

We begin with a section on preliminaries describing our language agnostic approach for the rest of the chapter, followed by a brief description of Information Theory concepts needed to understand our work. This is followed by a description of our Information Oriented Model of Computation and a discussion of problems from the field of Data Compression, Program Analysis and Machine Learning as viewed under our model. We will use these problems to demonstrate the three cases listed above. Finally, the chapter ends with a conclusion that summarizes the discussion and lists possible directions for future research.

7.2 | Preliminaries

7.2.1 — Language Agnostic Approach

We consider a *language agnostic approach* to programming, where the code is assumed to be written in a given programming language $\mathcal{L}$. Programs represent n-ary partial recursive functions over some infinite denumerable domain $\mathbb{S}$ of semantic objects. These can be values, traces, data structures etc., as computed by the programming language $\mathcal{L}$. Because both $\mathbb{S}$ and $\mathcal{L}$ are infinite effectively denumerable, in the following sections we will make no distinction between $\mathbb{N}$, $\mathcal{L}$, $\mathbb{Z}$, and $\mathbb{S}$. All the following notions apply therefore to properties of arbitrary infinite denumerable sets. Two partial functions f and g are *extensionally equivalent*, denoted $f \cong g$, if $dom(f) = dom(g)$ and for any $x \in dom(f) : f(x) = g(x)$.

7.2.2 — Semantics and Program Properties

Given a program $P \in \mathcal{L}$ we denote by $\varphi_P : \mathbb{S} \to \mathbb{S} \cup \{\bot\}$ the partial function computed by P, where $\varphi_P(x) = \bot$ means that $\varphi_P(x)$ is undefined. Being $\mathcal{L}$ Turing complete, a property $S \subseteq \mathbb{S}$ is *recursive enumerable* (r.e. for short) if there exists $P \in \mathcal{L}$ such that $S = dom(\varphi_P)$ [51]. S is *recursive* when both S and $\overline{S}$ are r.e. . We denote by $\mathbb{W}_P \stackrel{\text{def}}{=} dom(\varphi_P)$. Recall that the set of all r.e. sets is $\wp^{\text{re}}(\mathbb{S}) \stackrel{\text{def}}{=} \left\{ \mathbb{W}_P \,\middle|\, P \in \mathcal{L} \right\}$. It is known that $\wp^{\text{re}}(\mathbb{S})$ is effectively denumerable and, whenever ordered by set inclusion, it forms a distributive lattice with $\varnothing$ and $\mathbb{S}$ as respectively bottom and top elements. Correspondingly, the set of all recursive sets $\wp^{\text{rec}}(\mathbb{S})$ is a Boolean algebra [49, 54, 63].

Programs $P \in \mathcal{L}$ are intended to provide an intensional representation (coding) of properties (or concepts) of concrete semantic objects in $\wp(\mathbb{S})$. An effective (partial recursive) procedure that associates with each program P its corresponding semantic property is called a semantics for $\mathcal{L}$: $[\![\cdot]\!] : \mathcal{L} \to \wp(\mathbb{S})$:

$$[\![P]\!] = \left\{ \varphi_P(x) \,\middle|\, x \in \mathbb{S} \wedge \varphi_P(x) \neq \bot \right\}$$

In the following we denote with $[\![P]\!]^{i/o}$ the input/output relational (denotational) semantics of P. It is well known that $[\![P]\!]^{i/o}$ can be specified by Abstract Interpretation of a trace-based semantics of P [15]. It is known that in general $[\![P]\!]$ is a r.e. for any $P \in \mathcal{L}$.

We need concrete semantic objects in a measurable space, since probability is a measure, making this a requirement for Learning on the concrete objects (see discussion in Section-6.1.1).

Thus, our concrete domain will always be a σ-algebra $\mathcal{E}$ on some set $\mathbb{S}$.

Because there exist denumerable sequences of recursive sets whose union is not recursive (but r.e.)[1] we consider a σ-algebra of recursive events $\mathcal{E} \subseteq \wp^{\text{rec}}(\mathbb{S})$ as an observable measurable space $\langle \mathbb{S}, \mathcal{E} \rangle$.

These can be (recursive) sets of execution traces, program control graphs, sets of numbers. Typically these are obtained by a further abstraction of a given concrete semantics.

This can be achieved by assuming that programs in our language are circuits or always terminating programs, or by considering the Abstract Interpretation of $[\![\cdot]\!]$ in an abstract domain A: $[\![\cdot]\!]^A$.

Note that while the correct terminology is to refer to the pair $\langle \mathbb{S}, \mathcal{E} \rangle$ as the measurable space; we will, however, use the term *space* to refer to $\mathbb{S}$ (like the sample space from probability parlance) and the term *domain* to refer to $\mathcal{E}$ (like the concrete domain from abstract interpretation parlance). A set $S \in \mathcal{E}$ is referred to as an *event* in probability parlance, and as a property in Abstract Interpretation parlance.

Example 7.2.1. The standard denotational input/output semantics of a programming language is defined for $\mathbb{S} = \mathbb{Z}$ and $\varphi_P : \mathbb{Z} \to \mathbb{Z} \cup \{\bot\}$ is defined as usual. Other static semantics can be defined analogously. For instance the control-flow graph (CFG) semantics of an imperative language $\mathcal{L}$ corresponds to choose $\mathbb{S}$ as the set of all graphs $\langle V, E \rangle$ with $V \subset \Pi$, where Π is the set of single statements expressible in $\mathcal{L}$, i.e., the possible program points of any program $P \in \mathcal{L}$, and $E \subseteq V \times V$. Denote by $\Pi(P)$ the set of all statements (program points) in P. In this case we have:

$$\varphi_P(\langle V, E \rangle) = \begin{cases} \langle V', E' \rangle & \text{if } V \subseteq \Pi(P) \wedge c' \in V' \wedge (c, c') \in E' \;\Leftrightarrow\; c \rightsquigarrow c' \text{ in } P \\ \bot & \text{otherwise} \end{cases}$$

is the function associating with each program P the corresponding CFG, where $c \rightsquigarrow c'$ represents that the control flow may flow from statement c to c'. In this case the property of a program P is the set of all subgraphs of the CFG of P. In this case $[\![P]\!]$ is always a recursive set.

[1] It is enough to consider the sequence of $T_n = \{\, P \mid \varphi_P \text{ is defined in } n\text{-steps} \,\}$ and $K = \cup_{n \in \mathbb{N}} T_n$.

7.3 | Information Theory

7.3.1 — Shannon Entropy

Hartely [28] first proposed the idea of a quantitative measure of information based on physical considerations, as opposed to psychological considerations. He quickly realized that a logarithmic function can be an intuitive measure of information.

The idea was further expanded upon and generalized by Shannon, who was the first to propose Entropy, a physical concept, as a measure of the information in his seminal paper in 1948 [60]. The ideas of Information theory based on Ergodic theory were then further developed and popularized also by the work of Shannon.

The Entropy of a discrete random variable X with probability distribution $P(x)$ is defined as $H(X) \stackrel{\text{def}}{=} -\sum_{x \in X} P(x)\mathsf{log}_2 P(x)$ [42]. Shannon showed that the Entropy $H(X)$ of the random variable X describes the minimum length in bits of the binary sequence needed by a receiver to reconstruct the object x [23].

Entropy is sometimes called the missing information: the larger the Entropy, the less a priori information one has on the value of the random variable [42, page 3].

When the random variable X in the above definition is a uniform probability distribution, implying that any object $x \in X$ may be selected with equal probability, the Entropy $H(X) = \mathsf{log}_2|X|$.

This explains the key idea of Information Theory, that the information content of an object depends on the domain from which the object is to be selected. The bits are required, in essence, to uniquely identify the index for the object in the domain which is shared by both the sender and the receiver.

7.3.2 — Kolmogorov Complexity

The notion of Kolmogorov Complexity has its roots in Probability Theory, Computability, and Information Theory [38]. The idea is that a sequence (string) may be compressed considerably provided that it exhibits enough regularity. This is captured by the size of the smallest program that can produce the string. The Kolmogorov Complexity $K(X)$ of a set of objects is the smallest program that may produce X as its semantics: $[\![P]\!] = X$. The importance of Kolmogorov Complexity is its invariance property: Kolmogorov Complexity is recursively invariant between acceptable enumerations of partial recursive functions, i.e., programs in $\mathcal{L}$.

That means that it is known that $K(X)$ is not in general computable.

7.3.3 — Algorithmic Information Theory

Algorithmic Information Theory is a term coined by Gregory Chaitin, describes the intersection of Information Theory with Computability Theory. The idea being to study Information Theory in the context of Turing Machines that may only decide upto recursive sets and enumerate r.e. sets.

A key idea here is that while Shannon's Entropy Measure and Kolmogorov's Complexity Measurement are different on the surface—the former being dependent on the domain of objects, while the latter dependent on the object itself; they are equivalent (upto a constant term) when considering a *universal probability distribution* that assigns a probability of $c * 2^{-K(x)}$ for any object x, where c is a constant and $K(x)$ denotes the Kolmogorov Complexity of the object. In general, for any recursive probability distribution, the expected value of Kolmogorov Complexity equals it's Shannon Entropy, upto a constant [38, 64].

Vereshchagin [66] showed that Kolmogorov Complexity $K(x)$ of a string $x \in \{0,1\}^*$ may be expressed as:

$$K(x) = \min_{i,p}\{K(i) + l(p) : T_i(p) = x\} + O(1)$$

where the minimum is taken over $p \in \{0,1\}^*$ and $i \in \{1,2,...\}$. The equation expresses that $K(x)$ is the sum of the description length $L_{\mathbb{N}}(i)$ of some Turing machine i when encoded using the standard code for integers, plus the length of a program p such that $T_i(p) = x$. $L_{\mathbb{N}}(i)$ may be replaced by $K(i)$: the shortest effective description of i.

Intuitively, one may understand that Shannon's idea was to describe an object in the context of some domain. In Kolmogorov and Chaitin's view of the two-part code description, the Turing Machine describes the domain, and the program describes the object in the context of the referenced domain [24].

This implies the existence of a *universal domain* of descriptions that all senders and receivers must refer to in the absence of apriori information on the restriction of the domain. This domain is the domain of all partial recursive functions, because they are *formally effective descriptions* [38, page 3].

Chaitin was able to compute the Entropy and information content of sets under this domain [9]. The idea was that given a Universal Turing Machine U, and a r.e. set S, let p be a bit sequence where each bit is obtained by independent toss of unbiased coin. Then he defined

$P_U(S)$ as the Algorithmic Probability that $U(p)$ enumerates S. The Algorithmic Entropy is defined as $H_U(S) = -\log_2(P_U(S))$, and Algorithmic Information I_U as the length in bits of the smallest string p such that $U(p)$ enumerates S. Clearly, $I_U(S)$ is nothing but the Kolmogorov Complexity of set S.

Given a Turing complete language $\mathcal{L}$, all recursively enumerable sets will have atleast one finite description in the language—the program $P \in \mathcal{L}$ that enumerates the set. Thus, for r.e. sets have $I_U < \infty$. Indeed this has been used as a definition for r.e. sets [9]. I_U bits is the absolute minimum information that is needed to communicate the set, modulo a constant term for the choice of $\mathcal{L}$, in the absence of apriori information that restricts the domain.

7.4 | Information Oriented Model of Computation

7.4.1 — Relating Information Theory and Computability Theory

We begin by relating the Entropy in Information Theory to decidability in Computability Theory. Namely, we present the idea: *To know a set, is to decide a set*.

Our goal with the framework is to be able to describe computation as communication of information. Information to be communicated is represented by a set S drawn from a space $\mathbb{S}$ using a probability distribution P.

We will use Information Theory to measure *how much* information is being communicated, and, Computability Theory to describe *what* information is being transmitted.

Information Theory uses Entropy as a measurement of information. Shannon Entropy is defined for a random variable that selects sets from a domain of sets via some specified probability distribution. Thus the information content is dependent upon the probability distribution and the domain.

For a given space $\mathbb{S}$, the domain will be a σ-algebra on $\mathbb{S}$. We assume the domain to be $\wp(\mathbb{S})$. Thus, all sets of elements in $\mathbb{S}$ are in the domain, and may be selected by a random variable using some probability distribution.

Next, we assume a uniform probability distribution. The uniform probability distribution places a restriction that all sets have the same probability for being selected. Thus, no information is available apriori about the set, from the distribution. Indeed, any other distribution would provide information about which elements are likely (or unlikely) to be in a set. This is why uniform probability distribution has the highest entropy for a given domain [4, section-1.6].

Entropy for a random variable X that may select any set S from the domain $\wp(\mathbb{S})$ with a uniform probability distribution is given by $H(X) = log_2\big(|\wp(\mathbb{S})|\big)$ bits.

Notice that the Entropy is dependent on the domain, rather than a specific set itself. This implies that under a uniform probability distribution, that is with no apriori information provided, all sets in the domain have the same amount of information: $log_2\big(|\wp(\mathbb{S})|\big)$ bits.

Now that we know *how much* information is needed to describe a set, we want to know *what* that information is.

A key idea from Information Theory, is that the amount of information within an object (or rather amount of information needed to describe an object), is the information needed to uniquely identify it from all other objects in consideration.

For example, consider a set S selected via uniform probability distribution from the domain $\wp(\mathbb{S})$, where $\mathbb{S} = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Here the number of possible sets is 2^8. Entropy is defined as $H(\wp(\mathbb{S})) = log_2(2^8)$ bits[2]. Thus, an 8 bit vector is needed to uniquely identify a specific set, in essence using 1 bit to mark the presence of absence of every element in the $\mathbb{S}$. Thus, the information contained within a set $S \in \wp(\mathbb{S})$ is: $\forall s \in \mathbb{S}$, Is $s \in S$ or not.

Indeed that is essentially the concept of decidability in Computability Theory. A function $\varphi_P : \mathbb{S} \longrightarrow \mathbb{S} \cup \{\perp\}$ is computable by program P in Turing Complete Language $\mathcal{L}$ implies that the set $S = dom(\varphi_P)$ is decidable by the Turing Machine represented by program P.

While in general, there may exist denumerable number of Turing Machines that decide a language, however, consider an abstract Turing Machine that abstracts out details on transitions and retains only the set of all reachable states, namely semantics are the concrete collecting semantics (refer discussion in Chapter-4, specifically Section-4.2.2), then, each such abstract Turing Machine uniquely describes the set it decides.

Note that a more concrete semantics would include extraneous details, other than just the elements in the set, and a more abstract domain will necessarily lose some information about the elements in the set.

Hence, the information communicated by a set S in some domain of recursive objects D, $D \subseteq \wp^{rec}(\mathbb{S})$ is that of an Abstract Turing Machine (abstracted at the concrete collecting semantics level) that decides the set S.

This is a central idea in our model where we infer whether communication was lossless based on our ability to decide the set at the receiver end.

[2]Note the abuse of notation. Entropy is defined over the random variable, not the domain. However we used the domain to emphasize that the probability distribution did not provide any useful information.

7.4.2 — An Informal Introduction

We present here an informal introduction to our model, and will introduce formalization as we proceed. We begin with the traditional Information Theory use case of information communication.

Consider two people- Alice and Bob that want to exchange some information. The information is represented by a set S of objects in domain $\wp(\mathbb{S})$, i.e. $S \in \wp(\mathbb{S})$. The idea is that Alice may choose any set from the domain $\wp(\mathbb{S})$ with equal probability, and send it to Bob over a communication channel according to some "pre-agreed strategy".

Recall the Chaitin two-part code descriptions. The idea was that an object may be described using a two part code- a program, and, a Turing Machine that when fed the program, generates the object. Intuitively, the Turing Machine part of the code describes the regular aspects of the description, and the program describes the irregular (random) aspects with respect to the Turing Machine [38, Section 2.1.1].

While Kolmogorov and Chaitin focused on the question of how to split the description to minimize description length. Namely, which Turing Machine to use to describe an object. The answer being to select the Turing Machine that squeezes out regularities only so far that the reduction in length of random aspects (program length) is more than increase in description of Turing Machine. Indeed this spawned the idea of *Minimum Description Length* principle in statistics and inductive reasoning.

We, on the other hand, have a different goal- that of describing computation through communication (lossless and otherwise). Thus, we generalize the idea of two-part codes from program and interpreter (Turing Machine) to include also abstract interpreters. Thus, we blend Abstract Interpretation into the scenario by letting the "pre-agreed strategy" be an abstract domain.

So let Alice and Bob's "pre-agreed strategy" be that of a shared abstract domain. Alice then sends a program in some language accepted by Bob the interpreter. Thus, Bob reconstructs the information by interpreting the program under the agreed upon abstract domain.

We discuss three possible communication scenarios:

1. Lossless Reconstruction: We say the information exchange is lossless if $\forall s \in \mathbb{S}$, Bob can confirm whether $s \in S$ with a "yes" or "no". Essentially, we say Bob demonstrates that he knows the set S by deciding the set S.

2. Sound Reconstruction: Sound information exchange is a lossy information exchange.

Namely, that some information has been lost and Bob may only decide partially whether $s \in S$. Thus when asked whether $s \in S$, Bob will answer with a ("no" or "maybe"). In the dual case, with a ("yes" or "maybe").

3. Conjectured Reconstruction: Conjectured Reconstruction is another lossy information exchange. Namely, only incomplete information is available such that Bob cannot make any sound judgements regarding $s \in S$, except the trivial judgement where he answers "yes" to all queries of the type "Is $s \in S$?". A possible approach is where he makes some assumptions/hypothesis to try to estimate the set.

7.4.3 — Going Formal

We now describe the scenario more formally. The model consists of five components listed below:

1. A measurable space $\langle \mathbb{S}, \wp(\mathbb{S}) \rangle$. If $\mathbb{S}$ is some arbitrary set, then we consider concrete domain as the σ-algebra on set $\mathbb{S}$. Namely, the collection of subsets of $\mathbb{S}$ that includes $\mathbb{S}$ itself, is closed under complement, and is closed under countable unions. As a choice, for the set $\mathbb{S}$, we fix the concrete domain as the largest σ-algebra on $\mathbb{S}$: $\wp(\mathbb{S})$, to allow Alice the widest possible choice of objects to communicate. Let sets be drawn from this domain, via some probability distribution, for communication.

2. A finite communication channel for Language $\mathcal{L}$ to share the "Program", the irregular part of description.

3. A pre-agreed abstract domain $\mathcal{A}$ consisting of only decidable sets, for describing the regular part of object description. The concrete and abstract domains are related by a concretization function γ and an abstraction function α.

4. Alice: the program synthesizer. Let Alice optionally have an oracle that can decide the halting problem.

5. Bob: the abstract computation. Bob is thus the abstract interpreter.

Assumption-1 states that concrete objects exist in a measurable space. It is a basic requirement for describing probabilistic distributions used in Machine Learning. Note that while we have set the domain to be $\wp(\mathbb{S})$, it is not a restriction, since Alice can always tweak the

probability distribution in a way that sets probability 0 for sets she wants to exclude from consideration, and alter the *effective domain*.
Often, we will let $\mathbb{S} \subseteq \mathbb{N}$ as it allows for simplicity without restricting the applicability of results. Recall our language agnostic approach from section-7.2. Even if we use $\mathbb{S} \subseteq \mathbb{N}$, objects in domain $\wp(\mathbb{S})$ can just as easily represent values, traces, data-structures, or any other semantic objects computed by the language $\mathcal{L}$ of the channel, and our results would still apply. The choice of probability distribution has to do with specific applications. We discuss this later in the section.

Assumption-2 dictates a finite communication channel, namely only finite amount of information may be communicated using the channel. In Programming Languages parlance, any program $P \in \mathcal{L}$ communicated over the channel is limited to finite program syntax. From an information theory perspective, it is known that a Turing Complete Language will be the most expressive language permitted on the channel, and it can represent upto r.e. sets in finite bits as discussed in Section-7.3.3.

Assumption-3 Let $\gamma : \mathcal{A} \longrightarrow \wp(\mathbb{S})$ and $\alpha : \wp(\mathbb{S}) \longrightarrow \mathcal{A}$ are the concretization and abstraction functions relating the abstract and concrete domains. The assumption that objects in abstract domain are decidable, is important because it allows verification on the abstract. Note that decidable here means that we want a Turing Machine to be able to compute whether, for any given element $s \in \mathbb{S}$, is $s \in a$? where a is any abstract object in $\mathcal{A}$. The answer to this question has no bearing on whether $s \in S$, the set being communicated. Conditions for that relationship are derived separately and are the subject of investigation here. Note that this assumption is quite general, since it is also a requirement in program verification by abstract interpretation, and for classification by Machine Learning.

Assumptions-4 and 5 describe Alice and Bob. The assumption-4 optionally provides Alice with access to an oracle that decides halting problem. This is needed so Alice can synthesize programs $P \in \mathcal{L}$ for any set (upto r.e.) that she may choose. Discussion of $S \notin \wp^{\text{re}}(\mathbb{S})$ does not make sense since even a Turing Complete Language may only express upto r.e. sets in finite syntax. Since Bob is limited by Turing Computability (Assumption-5), Bob may only accept Turing Complete or more abstract languages.

From an Information Theory perspective, Alice is the *source* and Bob is the *target*. The information to be sent is divided into two parts- the abstract domain $\mathcal{A}$, and the program P. The choice of probability distribution for selecting sets to be communicated has a direct

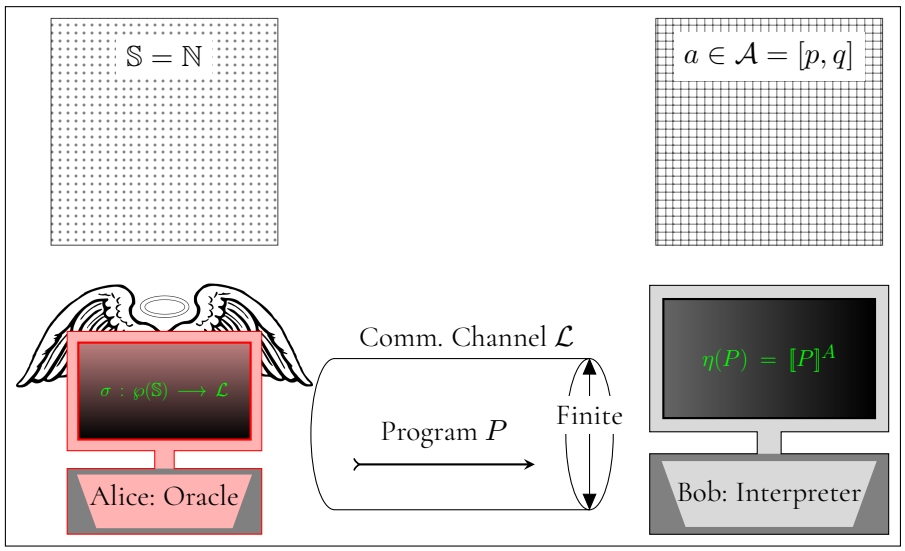


Figure 7.1: Information Oriented Model of Computation

impact on the choice of abstract domain and consequently the design of the language. These choices are implicit to Alice.

Consider for instance the domain $\wp(\mathbb{N})$ and say sound communication is required. A probability distribution that favors sets representable as intervals, for instance, over other sets mandates that abstract domain be that of intervals to maximize precision of communication most of time. Thus, the design of the language has to be such that it can represent intervals efficiently. Ofcourse, this assumes that the application in question favors precision.

We will discuss this more with appropriate examples. For now, suffice it to say that we have built an intuitive understanding that the choice of probability distribution, in conjunction with the application requirements, affects how the information transfer is split between the abstract domain and the program. The choice of uniform probability distribution, for instance, is a generic one because it favors all sets equally.

From an Abstract Interpretation perspective, Alice is a *program synthesizer*. We denote the work done by Alice by the functions: $\sigma : \wp(\mathbb{S}) \longrightarrow \mathcal{L}$.

Bob's task is to reconstruct the information, and then answer decidability questions of the type *For given* $s \in \mathbb{S}$, *is* $s \in S$*?*. We denote the work done by Bob with the function $\Omega : \mathcal{L} \times \mathbb{S} \longrightarrow \{0, 1\}$. We view this as a two part job, first of reconstruction: $\eta : \mathcal{L} \longrightarrow \mathcal{A}$, and second for supporting verification- $\omega : \mathcal{A} \times \mathbb{S} \longrightarrow \{0, 1\}$. Note that sometimes we have $\mathcal{A} : \mathbb{S} \longrightarrow \{0, 1\}$, in which case ω is id.

We can use Bob to explain any program or algorithm in our model.In the Applications section section-7.5), we explain Data Compression (lossless and sound compression schemes), Abstract Interpretation and Machine Learning through Bob. Thus, we vary η to correspond to a decompression algorithm, an abstract interpreter and a machine learner, respectively, all to do the same task of reconstructing the information sent by Alice.

We thus demonstrate the connection between these fields by showing that they solve instances of an abstract problem of set reconstruction. We will compare the input information quality, the tradeoffs on various abstractions, and output information quality in these various η.

We summarize the model in Figure-7.1. Let a concrete domain $\wp(\mathbb{S})$, abstract domain $\mathcal{A}$ and communication language $\mathcal{L}$. Then, let Alice select sets $S \in \wp(\mathbb{S})$ via a probabilistic distribution $\mathcal{D}$ and communicate to Bob. We then describe three possible cases on the reconstruction of the set S, by way of the output information quality:

1. Lossless Reconstruction: Here the information contained in the set S is equal to the information sent via the program and abstract domain. η describes the algorithm applied by Bob for reconstruction. For all $s \in \mathbb{S}$, Bob is certain whether $\omega(s) = 0$ or 1.
2. Sound Reconstruction: This is a lossy communication. η describes the algorithm applied by Bob for reconstruction. For all $s \in \mathbb{S}$, Bob is certain if $\omega(s) = 0$. Bob's assertions of $\omega(s) = 1$ are meaningless. Note that the case where Bob is certain if $\omega(s) = 1$ and assertions of type $\omega(s) = 0$ are meaningless is the dual. We will not discuss this case separately, because the duality is well understood in Abstract Interpretation community.
3. Conjectured Reconstruction: This is a lossy communication. η describes the algorithm applied by Bob for reconstruction. For all $s \in \mathbb{S}$, Bob answers, with some confidence $1 - \delta$ (where $\delta \in (0, 1)$), whether $\omega(s) = 0$ or 1, with some non-zero error rate.

7.4.4 — Lossless Reconstruction

If $S = \gamma([\![P]\!]^A)$, then reconstruction is lossless. We begin with some elementary results and then describe the model specifications that guarantee a lossless communication. We will defer the discussion of applications until after the discussion on all three scenarios of interest.

Theorem 7.4.1. *[Lossless Communication implies Decidability]: For any set S in domain of objects $\mathcal{E}$, if it is possible to communicate S in a lossless fashion, then there exist a Turing Machine that can decide S.*

Proof. Lossless communication implies that that there exist a program P such that S has a precise representation in the abstract domain: $\gamma(\eta(P)) = S$. Then, since elements in the abstract domain A are always decidable (refer model assumption-3), $\eta(P)$ can be used to decide membership in S. □

Remark 7.4.2 (Sets selected from a finite domain can be communicated in a lossless fashion)**.**

Proof. It is known that $\langle \mathbb{S}, \mathcal{E} \rangle$, where $\mathcal{E}$ is a σ-algebra on $\mathbb{S}$, has $|\mathcal{E}|$ finite. Since $|\mathcal{E}|$ is finite, the domain has finite Entropy. Thus, consider an abstract domain with $|\mathcal{E}|$ unique descriptions, then it is possible to communicate precisely, every set in $\mathcal{E}$ using a finite number of bits. □

Theorem 7.4.3. *Decidability does not imply lossless communication, except for trivial sets.*

Proof. Consider a decidable set $S \in \wp(\mathbb{S})$ that Alice wants to communicate to Bob. Every abstract domain must contain at the very least, a $\top$ element representing precisely the set $\mathbb{S}$. Hence the minimal abstract domain may only comprise of this $\top$. Analysis of any set (including all decidable sets), except the trivial set $\mathbb{S}$, that is the concrete representation of $\top$, over the abstract domain $\{\top\}$ will be incomplete; hence lossy communication. □

We now want to find the conditions such that communication between Alice and Bob is guaranteed to be always lossless.

From Theorem-7.4.1 we know that only total recursive sets may be communicated in a lossless fashion. As such, we restrict the domain $\wp(\mathbb{S})$ to $\mathcal{E} \subseteq \wp^{\text{rec}}(\mathbb{S})$. This ensure that sets in concrete domain are always decidable. Alice can then select sets from the domain $\mathcal{E}$ with a uniform probability distribution.

From an Information Theory perspective, for the communication to be always lossless between the concrete and abstract domains, it implies that there exists a bijection between the domains. Namely, every concrete set has a precise image in the abstract domain. Thus, α and γ are both injective and surjective functions.

This elaborates upon a fundamental feature of Information Theory. It is bijection blind. Namely, Information Theory is blind to the specific content of information. It only measures the amount of information. It's analogous to a weighing scale in that respect. It can measure the weight in a box as 5kg, but can't tell if it's weighing gold or tomatoes.

Theorem 7.4.4 (Under the assumption of uniform probability distribution and finite channel limit, Lossless communication of all sets in a domain implies finite domain)**.**

Proof. Under the uniform probability distribution, if all sets in a domain can be communicated losslessly, it implies that the abstract domain has a unique description for each set in concrete domain. Hence, there exists a bijection between the domains. This also means that knowing the abstract domain does not provide any meaningful information about the sets being communicated. Hence, the entire information in the set S, that is $\log_2|\mathcal{E}|$ is being communicated via the program. The finite channel width assumption implies $\log_2|\mathcal{E}|$ is finite. □

The communication between Alice and Bob can then be viewed as a compression scheme. Alice, the compressor, can optimize the Language syntax to allow for a cheaper encoding of concrete objects, and Bob, the decompressor, can reconstruct the object. We discuss the Huffman Encoding as a practical realization of the lossless communication scenario in section-7.5.1.

We can also view the communication as a program analysis problem. Let the concrete domain be $\wp(\mathbb{S})$. Then Alice selects sets for communication via a probability distribution $\mathcal{D}$. The distribution $\mathcal{D}$ assigns a set $S \in \wp^{\text{rec}}(\mathbb{S})$ some non-zero probability, and $S \notin \wp^{\text{rec}}(\mathbb{S})$ have a selection probability of zero. Then consider an abstract domain $\mathcal{A}$, abstraction function α and concretization function γ such that α, γ are bijections mapping $\wp^{\text{rec}}(\mathbb{S})$ to $\mathcal{A}$ and vice-versa. The encoding from Alice can be considered a program. Bob will then be an abstract interpreter and perform a complete analysis to reconstruct S.

Thus we summarize the model specifications discussed for lossless communication scenario in Table-7.1

Space $\mathbb{S}$	Domain $\mathcal{E} \subseteq \wp^{\text{rec}}(\mathbb{S})$: $\lvert\mathcal{E}\rvert < \infty$ Finite domain constraint
The finite communication channel $\mathcal{L}$	$\forall P \in \mathcal{L}$, P is an always terminating program
The abstract domain $\mathcal{A}$	$H(\mathcal{A}) = H(\mathcal{E}) \iff \alpha, \gamma$ are bijective
Alice: Program Synthesizer $\sigma : \mathcal{E} \longrightarrow \mathcal{L}$	σ is computable
Bob: Abstract Interpreter $\eta : \mathcal{L} \longrightarrow \mathcal{A}$	$\eta(P) = [\![P]\!]^{\mathcal{A}}$

Table 7.1: Model Specification for Lossless Communication Scenario

7.4.5 — Sound Reconstruction

When $S \subseteq \gamma(\eta(P))$, then reconstruction is sound. This scenario allows for some loss of information during the communication. However, the key idea is that the loss is in a controlled fashion, and hence the soundness guarantee.

Theorem 7.4.5. *[Sound Communication implies r.e. set]: For any set S in domain of objects $\wp(\mathbb{S})$, if it is possible to communicate S in a sound fashion, then there exist a Turing Machine that can partially decide S.*

Proof. Sound communication implies that that there exist a program P such that S has a sound approximation in the abstract domain: $S \subseteq \gamma\big(\eta(P)\big)$. Since elements in the abstract domain A are always decidable (refer model assumption-3), hence, $\eta(P)$ can be used to partially decide S. Namely, $\forall s \in \mathbb{S}$, if $\omega(s) = 0$, then $s \notin S$. If $\omega(s) = 1$, then membership of s in S is indeterminate. Dual case is can be proved in the same fashion. □

The last scenario was restricted to circuits and always terminating programs. Theorem-7.4.5 shows that here we are not bound by that restriction, and can communicate partial-recursive sets as well. From an Information Theory perspective, we are restricted to sets with finite algorithmic information ($I_U(S) < \infty$), namely r.e. sets.

Since we permit r.e. sets, we set $\mathbb{S} = \mathbb{N}$. Although this implies the domain includes all the sets in $\wp(\mathbb{N})$, once again, we can use the probability distribution to restrict the domain. Let Alice select sets $S \in \wp(\mathbb{S})$ via the *universal probability distribution* discussed in Section-7.3.3. This distribution assigns a non-zero probability only to r.e. sets because the shortest program length (Kolmogorov Complexity) is finite for only r.e. sets.

Since Program Synthesis is now undecidable, it means σ is not computable. Thus we allow Alice having an oracle. Thus Alice is the program synthesizer that can compress any r.e. set into a finite program in $\mathcal{L}$.

Theorem 7.4.6 (Sets selected via the universal probability distribution from the domain $\wp(\mathbb{N})$ can be communicated in a sound fashion by an Oracle)**.**

Proof. The domain $\wp(\mathbb{N})$ is a superset of all computably enumerable sets. The universal probability distribution (discussed in Section-7.3.3) assigns a non-zero probability to r.e. sets and 0 probability to all other sets. Thus, a random variable that selects sets from the domain via this probability distribution, will only select r.e. sets. For all r.e. sets, there exist atleast one program in Turing Complete Language. Program synthesis is in general undecidable, hence an Oracle is needed to synthesize the program, that can be communicated by Alice to Bob over the finite channel. Bob can then follow standard abstract interpretation to derive sound approximations in chosen abstract domain. □

Corollary 7.4.7 (All r.e. sets can be communicated in a sound fashion by an Oracle)**.**

Proof. For every r.e. set, there exist a finite program in a Turing Complete Language, that can be synthesized by an Oracle. Abstract interpretation provides a sound communication strategy. □

From an Information Theory perspective, after accounting for the probability distribution, there exist $|\mathbb{N}|$ objects in concrete domain to be communicated. For lossless communication, Bob needs a domain with equal cardinality to support a bijection (See Table-7.1). However, only finite bits may be communicated over the channel. Hence the communication will be lossy.

We discuss this case in application to Monotone Compression Schemes in Section-7.5.2 and to Abstract Interpretation in Section-7.5.3. Indeed, we will use this communication scenario to demonstrate the connection between Data Compression and Program Analysis in Section-7.5.3.

We also describe in this scenario a precision vs efficiency tradeoff. The idea is that while Alice first splits the information into a program and abstract domain, Bob may decide to further abstract the abstract domain. Abstract Interpretation describes the techniques needed to design sound approximate semantics to the language so that Bob can then reconstruct soundly.

For instance, Let Alice choose to send programs describing intervals. For this she has already told Bob to interpret the program under an abstract domain of intervals. Since the program Alice will send will only be describing an interval, Bob can either interpret under this domain, or choose a more abstract domain where sets contain even less information. Let Bob decide to interpret in the sign domain. Then Abstract Interpretation Framework provides Bob with the abstract semantics needed to soundly approximate the interval in the sign domains. We discuss these tradeoffs in detail in the Application Section-7.5.2

Space $\mathbb{S}$	Domain $\mathcal{E} \subseteq \wp^{\text{re}}(\mathbb{S})$
The finite communication channel $\mathcal{L}$	$\forall P \in \mathcal{L}, dom(\varphi_P) \in \wp^{\text{re}}(\mathbb{S})$
The abstract domain $\mathcal{A}$	$H(\wp(\mathbb{S})) > H(\mathcal{A})$
Alice: Program Synthesizer $\sigma : \wp(\mathbb{S}) \longrightarrow \mathcal{L}$	η is not computable. Hence Alice has Oracle
Bob: Abstract Interpreter $\eta : \mathcal{L} \longrightarrow \mathcal{A}$	$\eta(P) = [\![P]\!]^{\mathcal{A}}$

Table 7.2: Model Specification for Sound Communication Scenario

7.4.6 — Conjectured Reconstruction

Once again, consider the scenario of $\mathbb{S} = \mathbb{N}$. Previously, Alice leveraged the universal probability distribution to selectively send only r.e. sets.

We now remove that restriction and allow Alice sending arbitrary sets. Since it is not possible to encode arbitrary sets in Turing Complete Language, any finite representation will only represent a finite subset of the arbitrary set. We describe a Language to communicate arbitrary sets (partially) over the channel in Figure-7.2

$$\begin{array}{lll} \mathsf{stat} & ::= & \big(s, f(s)\big) \qquad\qquad (s \in \mathbb{S},\ f : \mathbb{S} \longrightarrow \{0,1\}) \\ \mathsf{prog} & ::= & \mathsf{stat}; \mathsf{stat}; \{\mathsf{stat};\} \qquad (\text{sequence of } m \text{ statements. } |m| < \infty) \end{array}$$

(a) Syntax

$$\frac{[\![\mathsf{prog}]\!] : \mathsf{stat}^m \longrightarrow (\mathbb{S} \longrightarrow \{0,1\})}{f_{\mathcal{D}} : \mathbb{S} \longrightarrow \{0,1\} \qquad \stackrel{\text{def}}{=} \qquad \begin{array}{ll} 1 & \text{if } \mathbb{P}_{\mathcal{D}}(s) \geq \frac{1}{2} \\ 0 & \text{if } \mathbb{P}_{\mathcal{D}}(s) < \frac{1}{2} \end{array}}$$

(b) Concrete Semantics

Figure 7.2: Language for Conjectured Reconstruction

Clearly, lossless communication would require Bob to evaluate f on all points in the space $\mathbb{S}$. However, that is not possible with the partial program with finite length m.

Let Alice select a set in the domain $\wp(\mathbb{S})$. Alice then sends a finite subset of points selected i.i.d. from the space $\mathbb{S}$, along with the knowledge of membership of these point w.r.t. her selected set S, using some probability distribution $\mathcal{D}$. Hence the program is a finite sequence of type $\mathbb{S} \times \{0,1\}$, with length m as shown in Figure-7.2a.

The semantics of the program, as shown in Figure-7.2b, essentially describe the indicator function for set S using the expected value for membership under the probability distribution $\mathcal{D}$. The probability distribution $\mathcal{D}$ is unknown to Bob.

The problem is that we need infinite length program (denumberable data points) to reconstruct f accurately, but we have only finite program of length m.

In such limited information scenario, sound assertions are not possible, and Bob may only make conjectures regarding the set S. Recall from Chapter-6 that this is a traditional learning theory problem. Should Bob choose to use the concrete domain, he will risk overfitting. It makes sense to choose some abstract domain $\mathcal{H}$ with finite VCdim($\mathcal{H}$)= d.

Learning Theory results specified in the Fundamental Theorem of Statistical Learning (refer Theorem-6.1.14), when interpreted in an Information Theoretic perspective, express the length of programs (amount of information needed for successful learning) as a function of VC-Dimension of the Hypothesis class. Thus, given a program, the theorem describes the viable Abstract Domains that Bob can learn under.

Hence the theorem state that Bob can *learn* a predictor $h : \mathbb{S} \longrightarrow \{0, 1\}$ from an abstract class $\mathcal{H}$ of *hypothesis* with VCdim($\mathcal{H}$)= d, using programs of length m, such that:

$$m \geq \frac{d + \log(1/\delta)}{\epsilon^2} \tag{7.1}$$

or , if the class $\mathcal{H}$ contains a hypothesis with zero error, then,

$$m \geq \frac{d + \log(1/\delta)}{\epsilon} \tag{7.2}$$

where $\delta, \epsilon \in (0, 1)$ with usual meanings from learnability theory. Namely, Bob will learn the predictor with confidence $1 - \delta$ and

- bounded by small error rate ϵ for programs with length m specified as function of $\delta, \epsilon, \mathrm{VCdim}(d)$ by equation-7.2 if the Realizability Assumption holds, or,
- bounded by small error rate ϵ from the Bayes Optimal Predictor for programs with length m specified as function of $\delta, \epsilon, \mathrm{VCdim}(d)$ by equation-7.1 if the Realizability Assumption does not hold.

For most problems, we cannot assume the Realizability assumption since Alice does not provide any additional information other than the program. Hence it makes sense to use the Agnostic PAC learning as viable strategy for communication.

We discuss the archtypical program analysis problem of lfp approximation with widening, as a conjectured reconstrution problem in section-7.5.4.

We know that an agnostic PAC learner will approximate the set S by approximating the indicator function for S: $f_S : \mathbb{S} \longrightarrow \{0, 1\}$ with some confidence $(1 - \delta)$ and small error ϵ from the best possible bayes optimal predictor.

We can demonstrate the impossibility of evaluating f with finite programs of this language using an information theory metric known as Relative Entropy. Relative Entropy is used, when approximating a distribution with another, (also called *Kullback-Leibler divergence*) to know how

much more information Bob needs:

$$KL(\mathcal{D}||\mathcal{H}) = -\int \mathcal{D}(x) \log_2 \left\{ \frac{\mathcal{H}(x)}{\mathcal{D}(x)} \right\} dx$$

Note that the KL divergence is a Relative Entropy measure, that computes the amount of average additional amount of information[3]required to specify the value of x (assuming we choose an efficient coding scheme) as a result of using an approximate distribution $\mathcal{H}(x)$ rather than the true distribution $\mathcal{D}(x)$. Always, $KL(\mathcal{D}||\mathcal{H}) \geq 0$, with equality only when $\mathcal{D}(x) = \mathcal{H}(x)$.

Following the approach from Chapter-6, in the absence of true (unknown) distribution $\mathcal{D}$, we approximate it with the available training data:

$$KL(\mathcal{D}||\mathcal{H}) \simeq \sum_{n=1}^{m} \{-\log_2 \mathcal{H}(x_n|\theta) + \log_2 \mathcal{D}(x_n)\}$$

The best estimation would require least amount of additional information, and thus minimize this function. Only the left term depends upon θ. Incidentally, that is also the negative log likelihood for θ under the distribution $\mathcal{H}(x|\theta)$ evaluated using the training set. Thus we see that minimizing this Kullback-Leibler divergence is equivalent to maximizing the likelihood function, a traditional Machine Learning technique.

Thus Agnostic PAC learning provides a viable technique for estimating the set in this scenario. We summarize the model spec in the table below.

Sample Space $\mathbb{S}$	$\|\mathbb{S}\| = \|\mathbb{N}\|$ Denumerable Space constraint
The finite communication channel $\mathcal{L}$	$P \in \mathcal{L}$ is a sequence of m data-points: $(s, f(s))$
The abstract domain $\mathcal{H}$	Realizability Assumption does not hold
Alice: Program Synthesizer $\sigma : \wp(\mathbb{S}) \longrightarrow \mathcal{L}$	m data-points selected iid using distribution $\mathcal{D}$ over $\mathbb{S} \times \{0, 1\}$
Bob: Agnostic PAC Learner $\eta : \mathcal{L} \longrightarrow \mathcal{H}$	$\eta(P) = h_P$: with confidence $(1 - \delta)$: $\mathbb{L}_{\mathcal{D}}(h_P) \leq \min_{h' \in \mathcal{H}} \mathbb{L}_{\mathcal{D}}(h') + \epsilon$

Table 7.3: Model Specification for Estimating Hypothesis Scenario

[3]measured in bits, since we have taken logarithm base 2.

7.5 | Application

7.5.1 — Huffman Compression

Compression is encoding the data (set of objects) into a compressed object that has a cheaper computer representation, yet can be used to reconstruct the original set of objects. Information Theory, or rather Algorithmic Information Theory, discussed in sections-7.3.1,7.3.2, and 7.3.3 forms the theoretical framework for Data Compression.

Compression maybe lossless or lossy. A lossless compression algorithm compresses data such that it can be decompressed to achieve exactly what was given before compression. A lossy compression algorithm, on the other hand, loses data that cannot be recovered.

We start with describing the general problem of Data Compression within our model and then discuss lossless compression and lossy compression schemes.

The Problem

The authors in [27], building upon the work [3], describe a monotone compression scheme as defined below. $[\mathbb{S}]^n$ denotes the family of n-element subsets of X.

Definition 7.5.1 (Monotone Compression Scheme)**.** Let m and d be two natural numbers with $m > d$. An $m \to d$ monotone compression scheme for a family $[\mathbb{S}]^{fin}$ of finite subsets of a set $\mathbb{S}$ is the pair functions: $\sigma : [\mathbb{S}]^m \to [\mathbb{S}]^d$ and $\eta : [\mathbb{S}]^d \to [\mathbb{S}]^{fin}$ such that whenever S is an m-element subset of $\mathbb{S}$: $S \subseteq (\eta \circ \sigma)(S)$

The problem of compression and reconstruction of finite sets has been presented as a two person game in [3].

Definition 7.5.2 (The finite superset reconstruction game)**.** There are two players: Alice ("the compressor") and Bob ("the reconstructor"). Alice gets as input a finite set $S \subseteq X$. She selects a subset $\sigma(S) = S', S' \subseteq S$ and sends it to Bob, according to a pre-agreed strategy. Bob then outputs a finite set $\eta(S') \subseteq X$. Their goal is to find a strategy for which $S \subseteq \eta(S')$ for every S.

Notice that the goal of the game is to find a strategy for sound reconstruction of finite sets. However, we want to examine lossless and sound compression schemes separately. Hence, in this subsection, we will assume the goal is to have lossless compression: $S = \gamma(\eta(S'))$ for every S. We will discuss sound compression schemes in the next subsection.

Indeed the game was an inspiration for our Information Oriented Model. Hence, we use the game description of Lossless Compression Schemes and assume the roles of compressor and decompressor for Alice and Bob.

Following the scenario described in Table-7.1, we assume $|\mathbb{S}| = n$ for some finite $n \in \mathbb{N}, n < \infty$. Then we know that lossless communication from concrete domain $\wp(samples)$ to abstract domain $\mathcal{A}$ is a lossless compression scheme. Lossless communication for all sets, under a uniform probability distribution, requires that $\wp(\mathbb{S})$ and $\mathcal{A}$ share a bijection with abstraction and concretization functions α and γ.

Note that under these conditions, the minimum cost of representation of a set in $\wp(\mathbb{S})$ and $\mathcal{A}$ is bound by Shannon Entropy of these domains. Hence, true compression is realized when we release the assumption of uniform probability distribution. Namely, we either assume that only some subset $\mathcal{E} \subset \wp(\mathbb{S})$ will be sent, in which case the information cost savings $H\big(\wp(\mathbb{S})\big) - H(\mathcal{E})$ can be realized by adjusting $\mathcal{A}$ to have a bijection with $\mathcal{E}$, the effective domain. Or, we consider the possibility that all sets from $\wp(\mathbb{S})$ are to be communicated, albeit with a non-uniform probability distribution.

The second situation is quite common in text compression where different alphabets occur with different frequency (non-uniform probability) in text. Consider that regular text has every character taking 1 byte (8-bits). However, compression can be realized by having variable length codes, namely shorter length codes for characters that appear more frequently (higher probability of occurrence) and longer codes for characters that occur infrequently.

Huffman Compression

We discuss one such lossless compression algorithm: Huffman Compression [31] and try to understand it via our model.

Consider alphabet $\mathbb{S} = (s_1, s_2, \ldots, s_n)$ of finite size n. Let the probabilities be defined by $W = (w_1, w_2, \ldots, w_n)$. What this means in the model is that Alice selects some text (a set S), the probability that $s_i \in S = w_i$.

The huffman encoding can be used to generate an efficient language, namely the codewords: $\mathcal{L}_{Huffman} = (c_1, c_2, \ldots, c_n)$, which is the tuple of (binary) codewords, where c_i is the codeword for $s_i, 1 \leq i \leq n$.

Thus, instead of using the generic language that assigns $\log_2|n|$ bits per alphabet, we use the efficient language that uses an average of $-\sum_{i=1}^{n} w_i \log_2 w_i$ bits.

Input ($\mathbb{S}$, W)	Symbol (s_i)	a	b	c	d	e	f	g	h	sum
	Probability Weight (w_i)	0.25	0.25	0.05	0.15	0.10	0.05	0.05	0.10	1
	Codewords (c_i)	01	10	11110	110	000	11111	1110	001	
	Codeword length (in bits)(l_i)	2	2	5	3	3	5	4	3	
	Contribution to weighted path length($l_i.w_i$)	0.5	0.5	0.25	0.45	0.30	0.25	0.20	0.30	2.75
	Contribution to Entropy($-w_i \log_2 w_i$)	0.332	0.411	0.521	0.423	0.518				$H(\mathcal{L}) = 2.72$

Table 7.4: Huffman Encoding

Alice will share the Huffman Language with Bob, and send him text encoded into this language, that Bob can interpret to decompress. Consider as example the huffman encoding generated by $\mathbb{S} = \{a, b, c, d, e, f, g, h\}$ in Table-7.4. An inefficient language would allocate 4 bits per symbol. Huffman builds an efficient language that minimizes Entropy (2.72), namely weighted average bits per character to 2.75. The difference between Entropy and weighted average bits per character is due to the fact that huffman requires integer bits per character.

Thus, Alice can use the Table-7.4 to synthesize program for any selected set S into $\mathcal{L}_{Huffman}$ by translating each character to it's binary encoding. For instance, consider $S = \{a, b, d\}$, then $\sigma(S) = \{\ 01\ 10\ 110\}$ is sent to Bob as a program. Bob then reconstructs, by interpreting the program back to concrete domain $\wp(\mathbb{S})$. The semantics are described in the table:$[\![\{\ 01\ 10\ 110\}]\!] = \{a\ b\ d\}$.

Thus we have demonstrated Huffman Encoding as lossless communication by abstraction and reconstruction. Further, we have also shown how choice of language effects efficiency of representation cost.

It is not only specific algorithms, but general problem frameworks may also be expressed under our Model. Next we discuss Monotone Compression Schemes as an instance of the sound communication scenario.

7.5.2 — Monotone Compression Schemes

Once again consider the general monotone compression scheme described in previous subsection.

The authors in [3] and [27] restrict S to a finite set and vary the cardinality of $\mathbb{S}$ and arrived at the following conclusions:

1. Trivial Case: Alice and Bob set σ and η to id.

2. Case $\mathbb{S}$ is finite: Alice sends $\varnothing$, and Bob outputs $\mathbb{S}$.

3. Case $\mathbb{S} = \mathbb{N}$ (Countably infinite $\mathbb{S}$): Alice sends $x_{max} = \mathsf{max}(S)$, and Bob outputs interval $[1, x_{max}]$.

4. Case $|\mathbb{S}| \geq \aleph_1$: Proving the existence of compression scheme for finite S requires proving continuum hypothesis to be true.

We fill keep $\mathbb{S} = \mathbb{N}$ fixed, and vary the cardinality of set S from empty, to finite, and, finally include infinite r.e. sets. We will then show that Abstract Interpretation presents a valid *strategy* for sound monotone data compression schemes. Namely, we will show that Alice's selecting and sending of the subset of *data-points*, is actually a program synthesis problem, where *program* is in some deterministic language implicit in the problem. Next, we will show that Reconstruction is indeed an *abstract interpretation* of the given *program*. Thus we will demonstrate an implicit Abstract Domain assumed in the sound compression scheme problem; and thereby demonstrate that the *pre-agreed strategy*, together with the subset S' communicated to Bob, in essence, form the two-part code as introduced in section-7.3.3 and referenced throughout our model. Finally, we will show that for the case of $S \in \wp^{\text{re}}(\mathbb{S})$, we automatically go from Data Compression to Program Analysis.

So, consider $\mathbb{S} = \mathbb{N}$, and let Alice select a set $S \in \wp(\mathbb{S})$:

Case $S = \varnothing$

Compression: When Alice gets as input an empty set $\varnothing$, there is indeed no information to compress. She simply sends the empty set to Bob.

$$\sigma(\varnothing) = \varnothing$$

Reconstruction: With no information on S, the soundness condition requires Bob to output the entire space. For each element $x \in \mathbb{S}$, Bob simply has no information whether $x \in S$ or not. So to err on caution, for soundness guarantee, Bob outputs x.

$$\eta(\varnothing) = \mathbb{S}$$

From an Information Theory perspective, the program provides the information to select a set within the domain. In the absence of the program, only the information provided by the domain is available, which is that selected set S is anything in the domain.

Case $S \in [\mathbb{S}]^{fin}$

As already discussed above, a trivial solution to the sound strategy problem for the case of finite S exists where Alice sends the single element $x_{max} = \mathsf{max}(S)$ and Bob reconstructs the interval $\{1, ..., x_{max}\}$.

We examine this reconstruction via the two part code description strategy. To that end, we view this compression scheme as leveraging an abstract interpreter and programs in a language $\mathcal{L}^\sharp$. The concrete domain is the poset $\langle \wp(\mathbb{S}), \subseteq \rangle$. The abstract domain is a set of intervals $\mathbb{I} = \{[1, x] | x \in \mathbb{N}\}$ ordered by $\subseteq$ relation. We also describe the corresponding abstraction function $\alpha : \wp(\mathbb{N}) \longrightarrow \mathbb{I}$ as $\alpha(S) \stackrel{\text{def}}{=} [1, \max(S)]$, and a concretization function $\gamma : \mathbb{I} \longrightarrow \wp(\mathbb{N})$ as $\gamma([1, x]) \stackrel{\text{def}}{=} \{n | 1 \leq n \leq x, n \in \mathbb{N}\}$. The language $\mathcal{L}^\sharp$ and the abstract semantics are defined in Figure-7.3.

$$\mathsf{stat} \quad ::= \quad d \qquad \text{(numeric constant, } d \in \mathbb{N})$$

(a) Syntax

$$\frac{[\![\mathsf{stat}]\!] : \mathbb{N} \longrightarrow \mathbb{I}}{[\![d]\!] \quad \stackrel{\text{def}}{=} \quad [1, d]}$$

(b) Abstract Semantics

Uncompressed Set S	$\{5, 7, 9, 11\}$
Compressed Program $P = \sigma(S)$	11
Reconstructed Set $\eta(P)$	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$

(c) Example Compression & Reconstruction

Figure 7.3: Alice Bob Language

Compression: The compression function $\sigma : \wp(\mathbb{N}) \to \mathcal{L}^\sharp$ is a program synthesizer for language $\mathcal{L}^\sharp$. It takes as input the concrete semantics (uncompressed object) and outputs a program. Note the compressed object has a two part description. The program is only one part; the other part is the abstract interpreter.

Reconstruction: The reconstruction function $\eta : \mathcal{L}^\sharp \to \wp(\mathbb{N})$ is an interpreter that takes input a program $P \in \mathcal{L}^\sharp$ and output the abstract semantics which are then concretized by applying γ. Thus, $\eta(P) \stackrel{\text{def}}{=} \gamma([\![P]\!]^{\mathbb{I}})$. Concrete semantics is a finite object and is computer representable since the language is Turing Incomplete.

Proving the Need for Abstract Domain

In the Definition-7.5.1, this concept of abstract interpreter and the language, in which the compressed object is encoded, is unclear and only informally referred to as the "strategy" used by Alice and Bob to communicate in the game (Definition-7.5.2).

In the example, the concrete domain for Alice is $\wp(\mathbb{N})$. Let Alice select a finite set $S \in \wp(\mathbb{N})$. Then let Alice select a subset $S' \subset S$. Let Alice communicate S' to Bob and Bob provably reconstruct a sound approximation: X. Thus, $S \subseteq X$ for any finite set S selected by Alice (assume uniform distribution over all finite S).

From an information theory perspective, it is known that without apriori communication regarding an abstract domain, Bob will reconstruct set S in the concrete domain $\wp(\mathbb{N})$. This domain has Entropy $H\big(\wp(\mathbb{N})\big) = \log_2|\wp(\mathbb{N})|$. Hence, Bob cannot reconstruct soundly with finite information contained in S'. Hence an abstract domain is required.

Hence the monotone compression scheme problem is an instance of the sound reconstruction scenario from section-7.4.5. A feature of this scenario is the precision efficiency tradeoff by choice of abstract domain. We examine the tradeoffs in the context of sound monotone compression schemes.

Precision vs Efficiency Tradeoffs

We use precision as an inverse measure of False Positives or extraneous elements that arise upon reconstruction. Efficiency has context specific meaning. For Program Analysis community, efficiency is inverse measure of computation costs- time & space. By efficient compression we will refer to a cheaper representation of compressed object.

In abstract interpretation, soundness is guaranteed by design. Precision and efficiency are traded. For sound compression strategies, we observe a similar behavior.

The two-part code description of interpreter and program gives two points of providing information—via the abstract domain, and via the program. Ergo, two points of attack for precision vs efficiency tradeoffs—the Language Syntax and the Abstract Domain.

A complex abstract domain will have a lower Conditional Entropy for the compressed object and hence require a higher Entropy program. That is to say that if the abstract domain has precise representations for a large number of objects, it does not tell much to Bob as to which object to expect. As such, he needs more information in the program, and hence a wider communication channel—more expressive language.

Hence, a complex abstract domain implies higher precision in general, but contributes to an increase in length of both components of the description since the interpreter is more complex and also the program is more complex. Hence the precision efficiency trade-offs.

For instance, a Turing Complete Language will be both sound and precise for all $[\mathbb{S}]^{fin}$. However, the expressivity of the language (and hence abstract domain) comes at a cost of efficiency. Hence it may be desirable to explore Turing Incomplete Languages (more abstract domains) to achieve higher compressibility at the cost of imprecision in the general case.

Revisit the example in Figure-7.3. While reconstruction was sound, it is imprecise in the general case. This is because several concrete objects do not have a precise representation in the abstract world. The compression function σ is a many-to-one map.

Thus Alice and Bob may decide to use a Turing Complete Language, with standard C-like syntax and semantics. The resulting encoding is shown in Figure-7.4. Clearly this approach is significantly less efficient. Thus, Alice and Bob may decide to use a more balanced approach

Uncompressed Set S	$\{5, 7, 9, 11\}$
Reconstructed Set $\eta(P)$	$\{5, 7, 9, 11\}$

(a) Example Compression & Reconstruction

```
begin
    d ← 5 ;
    while d ≤ 11
     do
        d ← d + 2 ;
    end
end
```

(b) Compressed Program $P = \sigma(S)$

Figure 7.4: Example Compression & Reconstruction in Turing Complete Language

by using a Turing Incomplete Language that is more expressive language than the first attempt. For instance, by including a lower bound in the language:

$$\mathcal{B} \stackrel{\text{def}}{=} \{[a, b] | a \in \mathbb{N}, b \in \mathbb{N}, a \leq b\}$$

Thus, there is a tradeoff between language expressivity (reconstruction precision) and efficiency-cost of representation of compressed object.

$$\textsf{stat} \quad ::= \quad [a,b] \qquad (\text{interval}, a \in \mathbb{N}, b \in \mathbb{N}, a \leq b)$$

(a) Syntax

$$\frac{[\![\textsf{stat}]\!] : \mathcal{B} \longrightarrow \wp(\mathbb{N})}{[\![[a,b]]\!] \quad \stackrel{\text{def}}{=} \quad \{a,...,b\}}$$

(b) Concrete Semantics

Uncompressed Set S	$\{5,7,9,11\}$
Compressed Program $P = \sigma(S)$	$[5,11]$
Reconstructed Set $\eta(P)$	$\{5,6,7,8,9,10,11\}$

(c) Example Compression & Reconstruction

Figure 7.5: Improved Precision Language

Program Analysis community makes the tradeoff by varying the abstract domains from more abstract to more complex until the property of interest can be decided. This is possible because abstract interpreters are designed to parse specific language syntax and allow varying abstract semantics. For instance an abstract interpreter may accept all C language syntax and allow interpretation of all C program in various abstract domains.

In compression, however, in practise, once the abstract domain is decided upon, for efficiency, the syntax is also adapted. This is so because having a more cumbersome syntax would increase the description length for any object, making the compression scheme inefficient.

Recall the Huffman encoding from section-7.5.1. The concrete and abstract domain have the same number of total objects. However, Huffman provides compression by way of language efficiency.

In the future, it would interesting to experiment following the abstract interpretation approach in compression. It may be possible to have a compressed object that could be fed to a variety of decompressing algorithms that may perform decompression of increasing quality at increasing computation costs. Ofcourse, such a compressed object would have a longer than minimum possible description length.

7.5.3 — Program Analysis: An extension to Compression Scheme

So far we have been restricted to finite S. Here we discuss compressing infinite S: namely, $S \in \wp^{\text{re}}(\mathbb{S})$.

Here, we use a Turing Complete Language with C-like syntax with standard notations for

syntax and semantics. This is required because only Turing Complete Languages may have a finite description (program syntax) for r.e. sets.

Strategy: Let Alice and Bob decide to use a Language with C-like syntax. Further, let the abstract interpreter be standard interval analysis [17] and assume appropriate abstraction and concretization functions.

Compression: Since $S \in \wp^{\text{re}}(\mathbb{S})$, it follows that there exist a program $P \in \mathcal{L}$ such that $[\![P]\!] = S$ for some program P in Turing Complete Language $\mathcal{L}$. S is indeed compressible in our language $\mathcal{L}$, as it has finite syntax.

Indeed there may be denumerable P with concrete semantics S. Thus, we use Kolmogorov Complexity to define the best (cheapest) representation. We define $\sigma_K : \wp(\mathbb{S}) \to \mathcal{L}$ such that $|\sigma_K(S)| = K(S)$ where $K(S)$ is the Kolmogorov Complexity of the set S, modulo the constant term due to choice of language $\mathcal{L}$. It should be noted that $K(S)$ and σ_K are in general not computable.

Recall that the sound reconstruction scenario of section-7.4.5 addresses the computability issue with an Oracle for Alice.

Thus, to proceed further, we assume an Oracle synthesizer that generates a Program P such that $[\![P]\!] = S$.

Reconstruction: Bob's goal is to deconstruct P into a set $\eta(P)$ such that $S \subseteq \eta(P)$ in finite time. Since $S \in \wp^{\text{re}}(\mathbb{S})$, S is not decidable in the general case. Bob's problem is the typical problem of sound program analysis, for which we know abstract interpretation to be a viable strategy. Widening operators guarantee finite time computation of abstract semantics for infinite domains.

Example 7.5.3. Consider $\mathbb{S}$ to be the space of natural numbers $\mathbb{N}$. Then, let Alice select an infinite set of odd numbers: $\{5, 7, 9, ..., \infty\}$. As a matter of strategy, Alice and Bob decide to use language $\mathcal{L}$, a Turing Complete Language with standard C-like syntax and semantics, for communication. Assume Alice is then (magically) able to translate S into the program in Figure-7.6.

Bob would apply standard interval analysis and reconstruct the set as the interval $[5, \infty]$. The γ allows for Bob to enumerate the set if needed.

```
begin
    i ← 5 ;
    while True do
        i ← i + 2 ;
    end
end
```

Figure 7.6: Magically Compressed Program $P = \sigma(S)$

Relating Program Analysis & Compression

Thus we have successfully demonstrated using Program Analysis for Data Compression. Although, the compression is non-computable, the idea that program analysis and compression are instances of the same general problem is rather interesting in itself.

When we restrict ourselves to always terminating programs only, concrete semantics is a finite object. Then both Program Synthesis and Interpretation (Computing Concrete Semantics) is decidable. Thus, a program synthesizer σ and Program Analyzer η form a viable compression scheme that guarantees soundness and precision for all $S \in [\mathbb{S}]^{fin}$ by encoding in a Turing Complete Language.

With that in mind, we re-define a compression scheme as:

Definition 7.5.4 (Compression Scheme)**.** A monotone compression scheme is the pair function $\langle \sigma, \eta \rangle$ for some subset S of the space $\mathbb{S}$ if there exist the pair functions: $\sigma : \wp^{\text{re}}(\mathbb{S}) \to \mathcal{L}$ that compresses (cheaply encodes) S in some language $\mathcal{L}$, and $\eta : \mathcal{L} \to \wp^{\text{re}}(\mathbb{S})$ that can be used to reconstruct S.

We further summarize the relationship between program analysis and data compression via the Table below.

Space of Semantic Objects $\mathbb{S}$	$\lvert\mathbb{S}\rvert = \aleph_0$
Alice Input Set S	$S \in \wp^{\text{re}}(\mathbb{S})$
Compression Scheme σ	Program Synthesizer $\sigma_P : \wp(\mathbb{S}) \to \mathcal{L}$
Bob's Input Program P	$[\![P]\!]^A \supseteq \alpha(S)$
Reconstruction Scheme η	Abstract Interpreter $\eta_A : P \to [\![P]\!]^A$

Table 7.5: Summary: Program Analysis and Compression

Finally, we examine the widening problem as an instance of the conjectured reconstruction scenario.

7.5.4 — Conjectured Reconstruction via Widening

Abstract Interpretation relies on Widening to accelerate convergence to compute the limit of the abstract operator. The widening operator for a template numerical domain, is binary operator ∇ on the constraint set $CS = \mathbf{M} \times \vec{V} \leq \vec{C}$ as follows:

$$\begin{aligned} &\nabla : CS \times CS \to CS \text{ such that:} \\ &\qquad CS_1 \nabla CS_2 \supseteq CS_1, CS_2 \text{ and} \\ &\qquad \forall \{CS_i\} \subseteq CS \text{ chain } \{CS_i^{\nabla}\} \text{ is stable finitely} \end{aligned} \tag{7.3}$$

It has very weak algebraic properties, and requires strong guarantees of converging to a fixpoint in finite computation steps. Although widening operators are essential to all abstract domains that allow large computation chains, or infinite domains, there does not exist a standard algorithm for designing new widening operator.

In this section, we examine the widening problem from our Information Oriented Model of Computation, to understand how widening makes it's approximations. We will show, that from an Information Perspective, widening input information quality corresponds to the conjectured reconstruction scenario of Section-7.4.6, even though the output quality is eventually comparable to the sound reconstruction scenario of Section-7.4.5.

Note that while Cousot & Cousot's widening defintion described above describes only a binary operator that given two arguments produces a sound approximation, we use the term widening to refer to the larger goal of widening that is to approximate the lfp F with F is the semantic transfer function for a loop.

Recall, the concrete collecting semantics for a loop: $\mathbb{S}[\![\textsf{while } \mathit{cond} \textsf{ do } s \textsf{ done}]\!]R$ is defined as a lfp solution to a semantic transfer function F: $F(X) \stackrel{\text{def}}{=} R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![\mathit{cond}]\!]X)$. Refer Table-4.2.

Sound abstract semantics for the loop statement: $\mathbb{S}^\sharp[\![\texttt{while } \mathit{cond} \texttt{ do } s \texttt{ done}]\!]R^\sharp$ are defined as $\mathbb{C}^\sharp[\![\neg \mathit{cond}]\!](\operatorname{lim} F^\sharp)$ where $F^\sharp(X^\sharp) \stackrel{\text{def}}{=} X^\sharp \nabla (R^\sharp \cup^\sharp \mathbb{S}^\sharp[\![s]\!](\mathbb{C}^\sharp[\![\mathit{cond}]\!]X^\sharp))$

The template widening operator, thus, takes as input two abstract collecting states generated by subsequent iterations, and generates a sound approximate abstract collecting state. It then checks to see if the new approximation is stable, and if not, will use the new of abstract collecting state to perform another widening operation. This cycle continues until either stability is achieved or the top element is reached, in which case also stability is achieved.

This iterative process of approximating the lfp can be viewed as an instance of the conjectured reconstruction scenario. Consider that input are two abstract states similar to receiving two data points.

After which, the operator *learns* a sound approximation (hypothesis) of abstract states. We call this a learning of hypothesis and not a proof, because it is necessarily a hypothesis and may not actually be a sound approximation of lfp F. Indeed typical abstract interpretation process requires checking the hypothesis for correctness, and failing which, more data points are generated.

The goal is to *learn* the target, a sound approximation of lfp F with finite data points.

Indeed the problem of approximating a function with finite data points is the hallmark of conjectured reconstruction scenario.

The presentation of widening problem as a learning problem opens up to experimentation with machine learners for systematic designing of new widening operators. Indeed we build upon this intuition and present a novel learning based widening algorithm in the next chapter.

It also opens up a framework for experimenting with learning program invariants. Although learning program invariants, and checking is an established technique in program analysis [61]. However, the idea of learning on abstract states is a novel idea.

7.6 | Conclusion

The model successfully combines Computability Theory, Information Theory and Abstract Interpretation into a framework that allows comparing and contrasting the information available and approximations made.

We connect Computability Theory with Information Theory by relating Entropy with decidability. By itself, Information Theory allows measurement of information, but makes no mention of the content of the information. A Turing Machine on the other hand describes exactly the information contained within by means of the language it decides. However, it does not provide for a way to measure that information. Our model addresses that limitation by bridging the two together.

Next, the model makes explicit the approximations with simultaneous description of the concrete and abstract representations via an Oracle (Alice) and a Turing Machine (Bob). Further, by means of the "pre-agreed strategy" of abstract domain, and the specified Language, we make explicit the study of precision and efficiency tradeoffs.

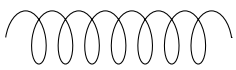
While Turing Model of computation views problems as computable and non-computable, Abstract Interpretation allows viewing them as precise computations and approximate computations. Our model adds a third category for "conjectures" by bringing in Machine Learning techniques under the umbrella.

To summarize, we measure *how much* information is to be communicated with Information Theory. The feasibility of communication, or rather, *What* information is communicated, is understood with Computability Theory. The mechanism of communication is then described by the algorithm under consideration.

We have demonstrated a study comparing approximations in the field of Data Compression, Program Analysis and Machine Learning in this chapter and opened up promising directions for cross disciplinary study of tools and techniques across these disciplines. The next two chapters build upon the insights here to further develop those ideas.

A possible future work may include extending the model with fuzzy sets and/or probabilistic approximations. It would be interesting to see what insights may be gathered by looking at works in these directions through the lens of the Information Oriented Model of Computation.

A yet another interesting direction of research would be to see if it is possible to describe quantum computations in the model. The Turing Model of Computation is based on the classical physics laws and does not provide opportunity to observe the effect of change of those laws. While in our model, the classical laws of physics are represented explicitly by means of Entropy. System Entropy behaves differently in the quantum world and hence quantum teleportation allows transferring infinite amounts of information across a quantum communication channel while using finite classical bits. It would be interesting to see if this model is able to describe quantum computation at all; and if so, then if it demonstrates any new insights in the quantum world.



"Mathematics is the study of analogies between analogies. All science is. Scientists want to show that things that don't look alike are really the same. That is one of their innermost Freudian motivations. In fact, that is what we mean by understanding"

Gian-Carlo Rota

8 Approximations in PL & ML

8.1 | Overview

In the last chapter, we presented a comparison between the approximations in Data Compression, Abstract Interpretation and Machine Learning. We were able to show all three techniques to be solving the problem of set reconstruction, albeit under different inputs and differing outputs.

Specifically, we saw that Abstract Interpretation receives superior input information (a complete program), while machine learning receives a weaker input (an incomplete program). The difference in input quality is reflected in output quality as well. Namely, while Abstract Interpretation output can be used to partially decide the set under reconstruction, Machine Learning output provides uncertain set reconstruction with probabilistic confidence (confidence less 1).

Motivated by these developments, we ask the question: *Are there any other instances of Machine Learning and Abstract Interpretation being used to solve related problems?*

In POPL-2015, Sumit Gulwani in his keynote address [25] presented an interesting way to look at the program synthesis problem. Paraphrasing Gulwani, consider a hoare-triplet $< P, S, Q >$ where P is a pre-condition, S is a program or statement, and Q is the post-condition, then we define the following problem:

1. Forward Program Analysis: Given P and $[\![S]\!]$, approximate Q
2. Backward Program Analysis: Given Q and $[\![S]\!]$, approximate P
3. Program Synthesis: Given P and Q, approximate $[\![S]\!]$

The last one being Gulwani's contribution, led to a surge in the research of ideas in Program Synthesis in the following years.

It is interesting to note that Abstract Interpretation (Forward and Backward) solves two of these problems, and, Machine Learning techniques are used for the third problem. Hence,

set reconstruction is not the only case where abstract interpretation and machine learning are solving related problems.

This brings up another interesting question: *Are Abstract Interpretation and Machine Learning solution techniques also instances of a single generalized framework?*

We attempt to answer that question in this chapter. We begin by exploring examples to demonstrate similarity in work-flows and follow up with mathematical generalization of supervised machine learning problem as approximating solutions to non-computable fix-point equations by computing monotonic abstract operators in a sound abstract domain- an abstract interpretation characteristic.

Next, we explore if abstract interpretation constructs such as join, meet and widening can be expressed as optimization problems. Finally, we discuss and compare the various challenges and overcoming techniques, observed and employed under either frameworks.

8.2 | Supervised Machine Learning: An Abstract Interpretation View

8.2.1 — Introduction

The Problem

Consider the archtypical example for Prediction Modeling by Supervised Machine Learning (Regression)- Say Alice wants to model the price of apartments in her city as a function of living area. A little bit of market research allows her to collect the data tabulated in Table-8.1. For simplicity, we assume that the price of apartments may not be affected by any other variables and the data collected is indeed accurate.

Consider $H = \{h : \mathsf{Area} \longrightarrow \mathsf{Price}\}$ to be the set of all maps from Area to Price. A naive approach would be to evaluate each $h \in H$ for fitness against the known data in Table-8.1. This approach is not feasible, as in general, such a set of all maps will have cardinality $\geq \aleph_0$ which makes the method non-terminating at best and non-computable in general (when cardinality is strictly greater than $\aleph_0$-the cardinality of set of all programs.. Thus, keeping in line with the philosophy of abstract interpretation, we start abstracting the domain until the problem becomes feasible.

We begin by making two key abstractions:

1. We reduce the domain of maps from all maps to a restricted domain of model class that is parameterized on some $\vec{\theta}$.

2. We choose to approximate the real parameter(s) in the model class to fixed precision reals, which are computable [69].

The first approximation is necessary to allow for a cheap representation of the model. With a model parameterized on $\vec{\theta}$, the memory cost for model representation is proportional to size of $\vec{\theta}$: $\mathcal{O}\left(sizeof\left(\vec{\theta}\right)\right)$.

The second assumption is necessary because reals are not computable.

Concrete Domain

One way to build an intuition towards choosing an appropriate domain (or *model class* as machine learning community calls it) is to plot a graph of the data (Table-8.1). A plot of the data in Table-8.1 is shown in Figure-8.1. After observing the plots, say Alice decides on a linear model for the purpose. If the data is too large and/or too many independent variables (features), Alice may choose to work with more complex models like Neural Networks.

Alice models the relationship between Living Area and Price as:

$$h\big(X,\theta\big) = \theta_0(x_0) + \theta_1(x_1) \tag{8.1}$$

where $X = \big(x_0, x_1\big)^T$ with $x_0 = 1,\ x_1$ is Living Area(feet^2), prediction function $h\big(X,\theta\big)$ estimates the Price$(1000\$)$ and $\theta = \big(\theta_0, \theta_1\big)^T$, with $\theta_0, \theta_1 \in \mathbb{R}$ being the parameters.

Now the question is how to select the best hypothesis from the restricted set of linear hypothesis. For this, Alice must decide on how she chooses to describe the accuracy or correctness of a map. So she defines a cost function $J : H \longrightarrow \mathbb{R}$ that maps a hypothesis to a $\mathbb{R}$, typically such that lower cost value corresponds to a more accurate (correct or usually best fitting) choice of hypothesis. A common choice for cost function in these type of problems is the Least Squared Cost function (Equation-6.11):

$$J\left(T, h\left(X,\vec{\theta}\right)\right) = \frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - h\left(X^{(i)},\vec{\theta}\right)\right)^2$$

where the set $T = \{\big(x^{(i)}, y^{(i)}\big) \mid 1 \leq i \leq m\}$ represents the training data.

The cost function induces a partial order on the set of hypotheses, thus creating the concrete domain poset $\big(H, \sqsubseteq\big)$, where $\sqsubseteq \stackrel{\text{def}}{=} h_1 \sqsubseteq h_2 \iff J(h_1) \leq J(h_2)$, where $h_1, h_2 \in H$.

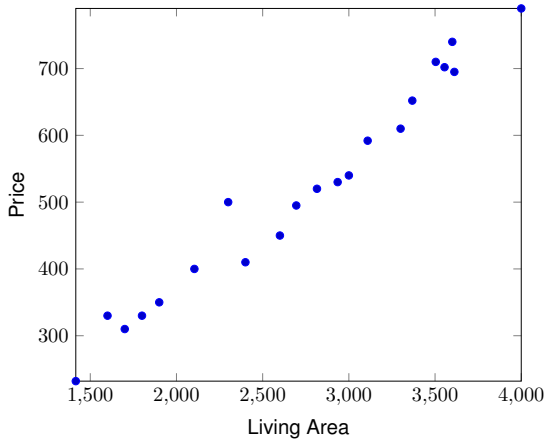


Figure 8.1: Plot of Training Data

Living Area (feet^2)	Price (1000$)
1,416	232
1,600	330
1,700	310
1,800	330
1,900	350
2,104	400
2,300	500
2,400	410
2,600	450
2,695	495
2,815	520
2,935	530
3,000	540
3,109	592
3,300	610
3,368	652
3,504	710
3,555	702
3,600	740
3,612	695
4,000	790

Table 8.1: Training Data for learning apartment Price as a function of Living Area

Infact, by Definition-2.2.2, the poset is a chain since $\forall h_1, h_2 \in H : \Big(J(h_1) \leq J(h_2)\Big) \vee \Big(J(h_2) \leq J(h_1)\Big)$.

Fixpoint Equation

This reduces the problem of finding the best fitting curve in the hypothesis class to an optimization problem:— *find the hypothesis that minimizes this cost function*. A naive way to find the optimal hypothesis would be to evaluate the cost function J on every hypothesis $h \in H$ and then output the hypothesis with least cost function. This is very expensive operation and not a feasible approach.

An alternate approach is the adjoint matrix method, which is also expensive operation and is to be avoided.

Gradient Descent is an algorithm for computing minima of function. Gradient descent with Least Squared Cost function as described in Equation-6.12:

$$\vec{\theta}^0 = \vec{\theta}^{init}$$

repeat until convergence: {

$$\theta_0^p := \theta_0^{(p-1)} + \beta(\sum_{i=1}^{m}(y^{(i)}) - h_\theta(x^{(i)}))$$

$$\theta_1^p := \theta_1^{(p-1)} + \beta(\sum_{i=1}^{m}(y^{(i)}) - h_\theta(x^{(i)})).x^{(i)}$$

} (8.2)

where β is the learning rate, x is the Living Area, $\vec{\theta}^{(p)}$ represents $\vec{\theta}$ after the p-th iteration and $(p \in \mathbb{N})$. $\vec{\theta}^{init}$ is assigned randomly.

Convergence implies $\vec{\theta}^{(p)} = \vec{\theta}^{(p-1)}$, or equivalently, the derivative of cost function to be zero:

$$\frac{\partial}{\partial \theta_j}\left(\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - h(X^{(i)}, \vec{\theta})\right)^2\right) = 0$$

Clearly, such points are fixpoint solutions to this fixpoint equation:

$$\vec{\theta^p} := F(\vec{\theta}^{(p-1)}) \tag{8.3}$$

where $F : H \longrightarrow H$ is an operator on the poset $(H, \sqsubseteq)$ described as:

$$F(\theta_j^{(p-1)}) = \theta_j^{(p-1)} + \beta(\sum_{i=1}^{m}(y^{(i)}) - h_\theta(X^{(i)})).x_j^{(i)} \quad (\text{ for } j \in \{0, 1\})$$

8.2.2 — Fixpoints in Machine Learning

The reduction of our prediction modeling example to a fixpoint problem is not an isolated case. We show next that infact Regression with Gradient Descent fits naturally into our model. We begin by describing the general problem statement, listing our assumptions and finally reduce the problem to that of solving a fixpoint equation. Further, we show that our method can be generalized to several other most common ML structures and algorithms just as well and present the first formal mathematical connection between the fields of Program Analysis and Machine Learning.

Problem Statement

We begin by describing the general Prediction Modeling problem where given a set of Training Data $T = \{(\vec{x^i}, y^i) \mid 1 \leq i \leq m\}$ where $\vec{x}$ is input feature vector of n features $\mid \vec{x} \mid = n \geq 1$, $\vec{x} \in X, y \in Y$ where X and Y represent the domain and co-domain of a hypothesis $h(\vec{x}, \vec{\theta}) \in H$, where H is set of all hypotheses parameterized by $\vec{\theta}$. The goal is to find the hypothesis $h \in H$, or rather the $\vec{\theta}$ that minimizes the cost function $J(\theta)$ when evaluated on dataset T. Further, we make the following assumptions:

1. Absence of latent and hidden features

2. Absence of noise in Training Data

3. Learnable Training Data: It is feasible to learn the parameters θ from T. We do not care for ill-formed problems, for instance, where we have only two training data points and we want to learn a cubic curve.

Our assumptions essentially refer to the completeness, accuracy and learnability of input data. These are essential to guarantee that given the right approximations, there exists sufficient information to learn the distribution from which the training data was sampled.

Concrete Domain

We define the concrete domain $C(H, \sqsubseteq)$ as the poset with objects as the hypotheses in the set H, ordered by the cost function $J(\theta) : H \longrightarrow D$, with $(D, \sqsubseteq_D)$ as the co-domain:

$$\sqsubseteq \stackrel{\text{def}}{=} h_1 \sqsubseteq h_2 \iff J(h_1) \sqsubseteq_D J(h_2) \tag{8.4}$$

Remark 8.2.1. The algebraic structure of the concrete domain is the same as the algebraic structure of co-domain of the cost function.

This is easy to prove since for any $h_1, h_2 \in H$, there exist corresponding images in D ordered by $\sqsubseteq_D$ and the Equation-8.4 uses the same ordering for h_1, h_2. Intuitively, the cost function defines the notion of better or worse hypothesis and hence is providing a way of comparing two hypotheses. Typically in Machine Learning, $J(\theta)$ maps hypotheses onto the set of Reals, which are a chain, and hence the concrete domain is a chain. However, strictly speaking from an Abstract Interpretation point of view, there is no such requirement for the co-domain of J to be $\mathbb{R}$ and can be any arbitrary poset $(D, \sqsubseteq_D)$. However, what Equation-8.4 is telling us is that algebraically, the poset $C(H, \sqsubseteq)$ has the similar structure as poset $(D, \sqsubseteq_D)$.

Fixpoint Equation

Gradient Descent algorithm for minimizing a function:

$$\begin{aligned}
&\bar{\theta}^0 = \bar{\theta}^{init} \\
&\text{repeat until convergence: \{} \\
&\qquad \theta_j^p := \theta_j^{(p-1)} - \beta \frac{\partial}{\partial \theta_j} J(\theta) \qquad \text{(for every j)} \\
&\}
\end{aligned} \tag{8.5}$$

where β is the learning rate, $\bar{\theta}^{(p)}$ represents $\bar{\theta}$ after the p-th iteration and $(p \in \mathbb{N})$. $\bar{\theta}^{init}$ is assigned randomly.

Convergence implies $\bar{\theta}^{(p)} = \bar{\theta}^{(p-1)}$. Indeed, this is a fixpoint equation and the Least Fixpoint of F(`lfp`F) is the best fitting hypothesis in the domain H:

$$\bar{\theta^p} := F(\bar{\theta}^{(p-1)}) \tag{8.6}$$

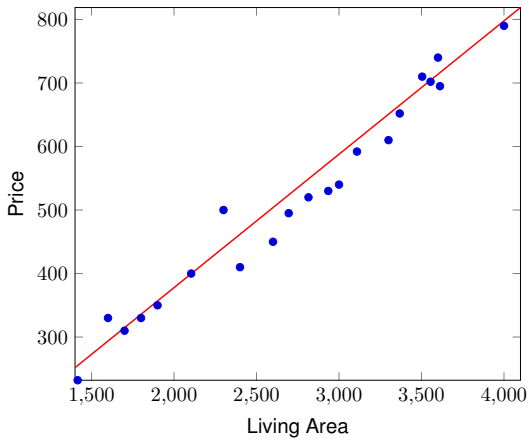


Figure 8.2: Linear Regression

where $F : H \longrightarrow H$ is an operator on the poset $(H, \sqsubseteq)$ described as:

$$F(\theta_j^{(p-1)}) = \theta_j^{(p-1)} + \beta \frac{\partial J(\vec{\theta})}{\partial \theta_j} \quad (\text{ for } j \in [1, |\ \vec{\theta}\ |])$$

Remark 8.2.2 (Input Dependent Domain)**.** In abstract interpretation, the ordering in the domain $\sqsubseteq$ is independent of the input to the analyzer. Here in machine learning, the domain changes the ordering of the abstract objects based on the particular training set input.

In abstract interpretation, concrete domain, and abstract domains for that matter, are unique to a language being analyzed and are independent of the input program. Here, the ingredients used to define the concrete domain:

1. The set of Hypotheses $H(x, \theta)$
2. The Cost function $J(\theta)$
3. The Training Set $T = \{(\vec{x}^i, y^i) \mid 1 \leq i \leq m\}$

While the first two ingredients are apparent from Equation-8.4, the last ingredient- The training set, is not obvious. It is however needed to evaluate $J(\theta)$ on hypotheses. This is a key differentiation from Abstract Interpretation since it implies that the computation of $\texttt{lim}F$ is not straightforward as in the case of abstract interpretation.

8.3 | A Regression Approach to Widening

8.3.1 — Preliminaries

Given a partial order P, we define-

Definition 8.3.1 (Width of Poset)**.** Width of a partial order P is defined as $w(P) = \texttt{sup}\{\mid A \mid$: A is an antichain in $P\}$.

Since $w(P)$ does not distinguish, for example, between ordered sets that contain arbitrarily large finite antichains and those that contain a countably infinite antichain. So additionally, we define $\mu(P)$ as the least cardinal κ such that $\kappa + 1 > \mid A \mid$ for every antichain A of P.

Definition 8.3.2 (Chain covering number)**.** We define $c(P)$ to be the least cardinal γ such that P is the union of γ chains in P.

Theorem 8.3.3. *In general,* $w(P) \leq c(P)$*; More specifically, if* P *is finite, then* $w(P) = c(P)$.

Definition 8.3.4 (Dimensions of Abstract Domain)**.** The minimal set of objects use to describe an abstract set forms the set of dimensions of that abstract domain. For example, an interval has two dimensions- lower and upper limit. In general, each row of the *template* matrix **M** corresponds to a different dimension.

Definition 8.3.5 (Dimension Chain)**.** A dimension chain C_D is a maximal chain $(X', \sqsubseteq)$ extracted from a poset $P(X, \sqsubseteq)$ such that $X' \subseteq X$ is constant in all dimensions other than D.

Definition 8.3.6 (Dimension Limit)**.** A dimension limit L_D is the greatest element of dimension chain C_D.

For abstract domains with infinite abstract objects, abstract interpretation may go on forever when for instance in case of non-terminating loops that produce a new abstract state on every iteration. To guarantee termination of analysis in finite steps for all programs being interpreted under an infinite abstract domain, abstract interpretation relies on Widening.

Cousot and Cousot defined Widening operator ∇ as a binary operator on the constraint set $CS = \mathbf{M} \times \vec{V} \leq \vec{C}$ as follows:

$$\begin{aligned} &\nabla : CS \times CS \to CS \text{ such that:} \\ &\qquad CS_1 \nabla CS_2 \supseteq CS_1, CS_2 \text{ and} \\ &\qquad \forall \{CS_i\} \subseteq CS \text{ chain } \{CS_i^{\nabla}\} \text{ is stable finitely} \end{aligned} \tag{8.7}$$

Intuitively, this definition presents widening as a binary operator that takes as input the collection of abstract states (collecting semantics) at the end of two subsequent iterations and results in a set of abstract states that contains both the input sets **and** performing another iteration with this set of abstract states as starting point does not result in addition of any new abstract states.

This definition provides very weak algebraic properties, requires strong termination guarantees and provides no systematic way to design such an operator. This is partly why even though design of a widening operator is an integral step in the design of an abstract interpreter using an infinite domain, not much work has been done to systematize the design of widening operator. While there exist works that derive widening of higher-level domains by lifting the widening of the base-level domain, the design of widening for base-level domains remains largely creative process, especially for numerical domains.

We propose an algorithmic approach to design of widening operator for numerical domains that can be represented with our constraint system model.

8.3.2 — Learning Widening

Intervals was the first infinite abstract domain introduced by Cousot [16]. Next came the polyhedra [18] and octagons [44]. The widening for polyhedra has since been improved upon [2]. We observe that all of these widening work by dropping constraints unstable w.r.t. iteration count. For example $[1, 2]\nabla[1, 3] = [1, \infty]$.

We view the problem of identifying unstable constraints as a learning theory problem. We thus view widening as a two step process-

1. Learn: Here, widening operator uses the abstract states, from two consecutive iterations, to learn a transformer (hypothesis in machine learning terminology) that relates iteration count (i) to an approximation of the set of abstract states collected upon i iterations. Thus the long chain is approximated to a single approximate transformer.

2. Stabilization: For each dimension in the abstract representation, we learn if the dimensional constraint is stable w.r.t. iteration count i. If stable, we keep the constraint. If unstable, we weaken the constraint to the dimension limit.

Learning Stability

To learn the instability, we use the linear model to learn the relationship between an element in vector $\vec{C}$ (and matrix **M**) and the iterator i. Note that the matrix **M** is constant as it is the shape *template*. Regression on **M** in such case may be skipped. Note that the domain of polyhedra is an example of a domain where **M** is not constant. We explain such domains in later sections of the chapter.

A linear model will always learn the element of $\vec{C}$ as either a constant or some linear function of i. Clearly, a constant element corresponds to a constant co-efficient w.r.t. iterations in the chain. Any other linear function corresponds to an instability as we move along the chain.

Given two points, linear regression simply learns the straight line joining the two points. Hence, given two constraint sets $CS_{i=1} = \{\mathbf{M} \times \vec{V} \leqslant \vec{C}\}$ and $CS_{i=2} = \{\mathbf{M} \times \vec{V} \leqslant \vec{D}\}$, the learned constraint set would be: $CS_i = \{\mathbf{M} \times \vec{V} \leqslant \vec{I}\}$ where $\vec{I}$ is defined below:

$$\vec{I}_x \stackrel{\text{def}}{=} (\vec{D}_x - \vec{C}_x).(i-1) + \vec{C}_x \tag{8.8}$$

Note that since collecting semantics are join morphism, $\forall x : \vec{D}_x \geq \vec{C}_x$.

Stabilization

A stable constraint implies that input iterations do not produce new abstract states along the dimension under consideration. It thus makes sense to preserve the constraint for precision.

Instability implies that abstract set grows in this dimension as interpretation proceeds through the two input iterations. Hence, by taking the limit of the chain, we replace the input dimensional constraints with the weakest constraint possible in the dimension.

Thus, $CS_{i=1} \nabla CS_{i=2} = \{\mathbf{M} \times \vec{V} \leqslant (\vec{C} \nabla \vec{D})\}$ where

$$[\vec{C} \nabla \vec{D}]_x = \begin{cases} \vec{C}_x & \text{when } \vec{D}_x = \vec{C}_x \\ \text{dimension limit } L_x & \text{otherwise} \end{cases} \tag{8.9}$$

Intuitively, since we do not know how many times the loop iterates and how the set of abstract states is growing with each iteration, we add all possible states along the dimension. This is what makes widening imprecise.

Data: Arrays $CS_{i=1}$, $CS_{i=2}$, DimensionLimits[D]
Result: $CS_{\nabla} = CS_{i=1} \nabla CS_{i=2}$
begin
  `// initialize empty constraint set`
  $CS_{\nabla} \leftarrow \phi$;
  $d \leftarrow 0$;
  while $d < D$ **do**
    $newConstraint \leftarrow LinearRegression(CS_{i=1}[d], CS_{i=2}[d])$;
    if *newConstraint is stable* **then**
      pass ;
    else
      `// newConstraint is unstable`
      $newConstraint \leftarrow CS_{i=2}[d].copy()$;
      $newConstraint.updateConstraintLimit(DimensionLimits[d])$;
    end
    $CS_{\nabla}.addToSet(newConstraint)$;
    $d++$;
  end
end

Algorithm 1: Template Widening Algorithm

Algorithm-1 summarizes our template widening algorithm.

Correctness

Note that Regression based Template Widening Equation-8.9 is the same result as described by Equation-5.8 describing the standard widening for the template abstract domain from Chapter-5 of fixed shape polyhedra. In addition to fixed shape polyhedra, the domain can also be used to model intervals, zones, octagons, etc. Hence, it is straightforward to prove the validity of our widening on this domain and also for intervals, zones and octagons as well.

8.3.3 — Examples

Example: Interval Widening

Consider P : `x` $= 1;$ `y` $= 0;$ `while x` < 10 $\{$`x` $++;$ `y` $++\}$. We use the domain of intervals I to infer the invariant that defines the bounds on x and y [16]. Intervals in two dimensions are conjunction of four lines: $a_1 \leqslant x \leqslant a_2 \wedge a_3 \leqslant y \leqslant a_4$ where real valued $a_1, a_2, ..., a_4$. In order to compute $[\![P]\!]^I$ during the fourth iteration inside the loop (point 2), we use widening

to generalize observations: intervals at $i = 2$ and $i = 3$ are described below.

$$\begin{aligned} point2(i=2) = \\ 1 \leqslant x \leqslant 2 \ \ \wedge \ \ 0 \leqslant y \leqslant 1 \end{aligned} \qquad \begin{aligned} point2(i=3) = \\ 1 \leqslant x \leqslant 3 \ \ \wedge \ \ 0 \leqslant y \leqslant 2 \end{aligned}$$

$$\begin{aligned} point2(i=2) \nabla point2(i=3) = \\ 1 \leqslant x \ \ \wedge \ \ 0 \leqslant y \end{aligned} \tag{8.10}$$

The widening on intervals extrapolates unstable bounds to infinity (equation-8.10). In this case abstract interpretation finds $a_1, a_2, ..., a_4$ such that the interval is a sound program invariant.

With linear regression we can automatically determine unstable bounds and therefore design the corresponding widening operation. The constraint sets at i=2 and i=3 can be expressed in the matrix form as:

$$CS_{i=2} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 2 \\ -1 \\ 1 \\ 0 \end{bmatrix}$$

$$CS_{i=3} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 3 \\ -1 \\ 2 \\ 0 \end{bmatrix}$$

Given the two constraint sets as input to the linear regressor, we obtain the new constraint set as a linear function of the iterator i as defined by equation-8.8:

$$CS_i = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} i \\ -1 \\ i-1 \\ 0 \end{bmatrix}$$

Next, we drop all rows with i in either matrix to obtains the widening.

$$CS_2 \nabla CS_3 \quad = \quad \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \quad \leqslant \quad \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Clearly the widening obtained above from regression is the same as that obtained by traditional methods in equation-8.10.

Example: Octagon Widening

Consider P : $\texttt{x} = \texttt{1}; \texttt{y} = \texttt{0};$ $\texttt{while}\ \texttt{x} < \texttt{10}\ \{\texttt{x}++; \texttt{y}++\}$. We must use the domain of octagons O to infer the invariant that relates x and y [44]. Octagons are conjunction of eight lines: $a_1 \leqslant x \leqslant a_2 \wedge a_3 \leqslant y \leqslant a_4 \wedge a_5 \leqslant a_6 x + y \leqslant a_7 \wedge a_8 \leqslant -a_6 x + y \leqslant a_9$ where real valued $a_1, a_2, ..., a_9$. In order to compute $[\![P]\!]^O$ during the fourth iteration inside the loop (point 2), we use widening to generalize observations: octagons at $i = 2$ and $i = 3$ are described below.

$$\begin{aligned} point2(i=2) = \\ 1 \leqslant x \leqslant 2 \ \wedge \ 0 \leqslant y \leqslant 1 \\ 1 \leqslant x + y \leqslant 3 \\ -1 \leqslant -x + y \leqslant -1 \end{aligned} \qquad \begin{aligned} point2(i=3) = \\ 1 \leqslant x \leqslant 3 \ \wedge \ 0 \leqslant y \leqslant 2 \\ 1 \leqslant x + y \leqslant 5 \\ -1 \leqslant -x + y \leqslant -1 \end{aligned}$$

$$\begin{aligned} point2(i=2) \nabla point2(i=3) = \\ 1 \leqslant x \ \wedge \ 0 \leqslant y \\ 1 \leqslant x + y \\ -1 \leqslant -x + y \leqslant -1 \end{aligned} \tag{8.11}$$

The widening on octagons extrapolates unstable bounds to infinity (equation-8.11). In this case abstract interpretation finds $a_1, a_2, ..., a_9$ such that the octagon is a sound program invariant.

With linear regression we can automatically determine unstable bounds and therefore design the corresponding widening operation. The constraint sets at i=2 and i=3 can be expressed in the matrix form as:

$$CS_{i=2} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 2 \\ -1 \\ 1 \\ 0 \\ 3 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

$$CS_{i=3} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} 3 \\ -1 \\ 2 \\ 0 \\ 5 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

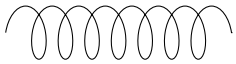
Given the two constraint sets as input to the linear regressor, we obtain the new constraint set as a linear function of the iterator i as defined by equation-8.8:

$$CS_i = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \leqslant \begin{bmatrix} i \\ -1 \\ i-1 \\ 0 \\ 2i-1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

Next, we drop all rows with i in either matrix to obtains the widening.

$$CS_2 \nabla CS_3 \quad = \quad \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \quad \leqslant \quad \begin{bmatrix} -1 \\ 0 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

Clearly the widening obtained above from regression is the same as that obtained by traditional methods in equation-8.11.



"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Turing

9 Conclusion

9.1 | Summary & Looking Ahead

We have presented an Information Oriented Model of Computation that allows to compare and contrast approximations across disciplines. We have successfully compared and related the problems of Data Compression, Machine Learning and Program Analysis as instances of set reconstruction problem, with varying input and output information quality.

Further, we have analyzed Supervised Machine Learning through the lens of Abstract Interpretation and demonstrated the Lattice Abstractions and Fixpoint Equations characteristic of Abstract Interpretation in Machine Learning.

We have also explained Widening, an Abstract Interpretation construct as a Learning Theory problem. Further, we have provided a learning based generic method to automatically develop widenings for new domains.

A next step would be study compressibility, learnability and analyzability (or the inverse-obfuscation) as related topics. Indeed learnability and compressibility have beeen studied together since the 1980s with the work of Warmuth et. al. [39]. Indeed this relationship was used recently in the seminal paper demonstrating independence of learnability to ZFC axioms [3]. Thus, we are hopeful that relating obfuscation with learnability and compressibility will also lead to positive strides in all disciplines. We present next a sketch of how obfuscation may be related to learnability. Namely, we conclude the dissertation with some ideas on how breaking learnability may lead to obfuscation.

9.2 | Learnability & Obfuscation

We have demonstrated widening to be a sound machine learner. This in turn implies that the regression limitations apply to this widening as well. This has significant impact on limits of abstract interpretation that depends on this widening for termination:

1. The noise independent of training data: The inability of the abstract domain to describe the complexity of trends in program states is fairly common in abstract interpretation. Researchers address this by switching to more precise domains. However switching to more precise domains does not necessarily result in stronger invariant [46]. Indeed, Sharma et. al. have explained this as a Bias Variance Trade-off [62].

2. Continuous and Global Nature of Learning: Hypothesis learned by regression is always continuous. Global nature implies that point discontinuity in one region may lead to errors all over the space.

Note however, that the above limitations hold if and only if the all of the below conditions are hold:

1. Domain has infinite chain condition and uses widening for termination.

2. Widening used is the template widening described above.

Condition-1 holds for most of the traditional domains in abstract interpretation literature. Condition-2 appears to be more restrictive. However, in practice, the implementation for widening for several domains can be shown to follow our regression template widening operator, such as:- Intervals, Octagons, and Polyhedra. Given the widespread use of the listed domains and their widening, the results are significant for practitioners.

9.2.1 — Model

Obfuscation is a semantic preserving transformation that makes an abstract interpreter imprecise while inducing no more than polynomial slowdown. For a compiler (semantic preserving transformer) τ to successfully obfuscate a program P implies $[\![P]\!]^\rho \subset [\![\tau(P)]\!]^\rho$ [21]. That is, there is some loss of information when analyzing $\tau(P)$, as compared to analysis of P, under the same abstraction ρ. The attacker is an abstract interpreter and a successful attack is a complete analysis $\rho([\![P]\!]) = [\![P]\!]^\rho$. In this model of obfuscation, attacker ρ is known and fixed at the time of designing the obfuscation which is targeted at making the specific abstraction incomplete.

Alternatively, we propose a new model of obfuscation that targets an entire class of abstract interpreters: those that rely on our template widening operator for termination. As previously explained, due to the global nature of model, discontinuities induce errors in analysis which is the definition of obfuscation described above.

Giacobazzi [21] measures the strength of an obfuscation by the size of set of domains made incomplete by the transformation. Since our obfuscation targets an entire class of analysis, it is a better obfuscation.

9.2.2 — Discontinuity Transformations

For this discussion, we restrict ourselves to program continuity model described by Chaudhari et. al. [11] . Following them, we consider only IMP numerical programs that terminate on all inputs. The semantic function for a program P is then map $[\![P]\!]$ of type $\Sigma \longrightarrow \Sigma$ such that for all states σ_{in}, $[\![P]\!](\sigma_{in})$ is the terminal state of P for execution starting from state σ_{in}.

We present three points to introduce discontinuity in programs:

1. Discontinuous Operation: e.g. x/y at y=0, +-* with $\perp$
2. Branch Conditions: Discontinuity at the "flip" point.
3. Loop Guards:

Further, we describe proof rules for sound discontinuity judgements.

9.2.3 — Example

Consider the program shown in Figure-9.1[left]. Interval analysis on the program using Interproc Analyzer is shown in Figure-9.2[left]. Note that the interval analysis is a successful attack (complete analysis). To obfuscate, we make the computation of i in the loop discontinuous. At $i = 5$, we jump the value by 100 and immediately after, drop the value back to 5. Figure-9.1[right] shows the transformed program. Note that the transformation is semantic preserving and does not induce more than polynomial slowdown. Figure-9.2[right] shows that interval analysis on the transformed program is incomplete. Thus, we successfully obfuscated the program by introducing discontinuity in computation. Additionally, the transformed program is incomplete for octagon and convex polyhedra domain as well.

```
var i:int;
begin
  i = 0;
  while (i<=10) do
    i = i + 1;
  done;
end
```

```
var i:int;
begin
  i = 0;
  while (i<=10) do
    if (i==5) then
       i = i + 100;
    endif;
    if (i==105) then
       i = i - 100;
    endif;
    i = i + 1;
  done;
end
```

Figure 9.1: Sample Programs

```
var i : int;
begin
  /* (L4 C5) top */
  i = 0; /* (L5 C8) [|i>=0; -i+11>=0|] */
  while i <= 10 do
    /* (L6 C18) [|i>=0; -i+10>=0|] */
    i = i + 1; /* (L7 C14)
                  [|i-1>=0; -i+11>=0|] */
  done; /* (L8 C7) [|i-11=0|] */
end
```

```
var i : int;
begin
  /* (L4 C5) top */
  i = 0; /* (L5 C8) [|i>=0; -i+106>=0|] */
  while i <= 10 do
    /* (L6 C18) [|i>=0; -i+10>=0|] */
    if i == 5 then
      /* (L7 C18) [|i-5=0|] */
      i = i + 100; /* (L8 C19) [|i-105=0|] */
    endif; /* (L9 C10) [|i>=0; -i+105>=0|] */
    if i == 105 then
      /* (L10 C20) [|i-105=0|] */
      i = i - 100; /* (L11 C19) [|i-5=0|] */
    endif; /* (L12 C10) [|i>=0; -i+105>=0|] */
    i = i + 1; /* (L13 C14)
                  [|i-1>=0; -i+106>=0|] */
  done; /* (L14 C7) [|i-11>=0; -i+106>=0|] */
end
```

Figure 9.2: Interval Analysis

Bibliography

[1] Tom M Apostol. Calculus, volume ii: Multi-variable calculus and linear algebra, with applications to differential equations and probability. 1969.

[2] Roberto Bagnara, Patricia M Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.

[3] Shai Ben-David, Pavel Hrubeš, Shay Moran, Amir Shpilka, and Amir Yehudayoff. Learnability can be undecidable. *Nature Machine Intelligence*, 1, 01 2019.

[4] Christopher M Bishop. *Pattern recognition and machine learning*. Springer Science+ Business Media, 2006.

[5] Rod M Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[6] Gregory J Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM (JACM)*, 13(4):547–569, 1966.

[7] Gregory J Chaitin. On the length of programs for computing finite binary sequences: statistical considerations. *Journal of the ACM (JACM)*, 16(1):145–159, 1969.

[8] Gregory J Chaitin. Randomness and mathematical proof. *Scientific American*, 232(5):47–53, 1975.

[9] Gregory J Chaitin. Algorithmic entropy of sets. *Computers & Mathematics with Applications*, 2(3-4):233–245, 1976.

[10] Gregory J Chaitin. Randomness in arithmetic. *Scientific American*, 259(1):80–85, 1988.

[11] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *ACM Sigplan Notices*, volume 45, pages 57–70. ACM, 2010.

[12] NV Chernikoba. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[13] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[14] Leo Corry. *David Hilbert and the axiomatization of physics (1898–1918): From Grundlagen der Geometrie to Grundlagen der Physik*, volume 10. Springer Science & Business Media, 2004.

[15] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.

[16] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Dunod*, 1976.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM, 1977.

[18] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[19] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analysers. In *International Static Analysis Symposium*, pages 317–333. Springer, 2012.

[20] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. New Mexico State University Portland, OR, 2003.

[21] Roberto Giacobazzi. Hiding information in completeness holes: New perspectives in code obfuscation and watermarking. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 7–18. IEEE, 2008.

[22] Janko Gravner. Lecture notes for introductory probability. *Chapter*, 13:151–160, 2010.

[23] Robert M Gray. *Entropy and information theory. First Edition, Corrected*. Springer Science & Business Media, 2013.

[24] Peter D Grünwald, Paul MB Vitányi, et al. Algorithmic information theory. *Handbook of the Philosophy of Information*, pages 281–320, 2008.

[25] Sumit Gulwani. Automating repetitive tasks for the masses. In *ACM SIGPLAN Notices*, volume 50, pages 1–2. ACM, 2015.

[26] Ian Hacking. *The emergence of probability: A philosophical study of early ideas about probability, induction and statistical inference*. Cambridge University Press, 2006.

[27] Klaas Pieter Hart. Machine learning and the continuum hypothesis. *arXiv preprint arXiv:1901.04773*, 2019.

[28] Ralph VL Hartley. Transmission of information 1. *Bell System technical journal*, 7(3):535–563, 1928.

[29] David Hilbert. The problems of mathematics. In *The Second International Congress of Mathematics*, 1900.

[30] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[31] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[32] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[33] SC Kleene. Introduction to metamathematics, new york (vannostrand), 1952.

[34] SC Kleene. Mathematical logic wiley & sons. 1967.

[35] Stephen Cole Kleene et al. λ-definability and recursiveness. *Duke mathematical journal*, 2(2):340–353, 1936.

[36] Andrei Nikolaevich Kolmogorov. Three approaches to the definition of the concept "quantity of information". *Problemy peredachi informatsii*, 1(1):3–11, 1965.

[37] Hervé Le Verge. A note on chernikova's algorithm. 1992.

[38] Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, 4th Edition*. Texts in Computer Science. Springer, 2019.

[39] Nick Littlestone and Manfred Warmuth. Relating data compression and learnability. 1986.

[40] Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8):491–502, 1973.

[41] Zohar Manna and Jean Vuillemin. Fixpoint approach to the theory of computation. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

[42] Marc Mezard, Marc Mezard, and Andrea Montanari. *Information, physics, and computation: Part-A Basics. Draft Version*. Oxford University Press, 2009.

[43] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects*, pages 155–172. Springer, 2001.

[44] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[45] Antoine Miné et al. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages*, 4(3-4):120–372, 2017.

[46] David Monniaux and Julien Le Guen. Stratified static analysis based on variable dependencies. *Electronic Notes in Theoretical Computer Science*, 288:61–74, 2012.

[47] Ernest Nagel and James R. Newman. Gödel's proof. *Scientific American*, 194(6):71–84,86, 1956.

[48] James B Nation. Notes on lattice theory, 1998.

[49] P. Odifreddi. *Classical Recursion Theory*. Studies in logic and the foundations of mathematics. Elsevier, 1999.

[50] David Park. Fixpoint induction and proofs of program properties. *Machine intelligence*, 5, 1969.

[51] E.L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.

[52] CK Raju. Probability in ancient india. In *Philosophy of Statistics*, pages 1175–1195. Elsevier, 2011.

[53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[54] H. Rogers. *Theory of recursive functions and effective computability*. The MIT press, 1992.

[55] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 25–41. Springer, 2005.

[56] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[57] Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group, 1970.

[58] Dana Scott and J.W. deBakker. A theory of program. *Unpublished Memo*, August 1969.

[59] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[60] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[61] Rahul Sharma. *Data-driven Verification*. PhD thesis, Stanford University, 2016.

[62] Rahul Sharma, Aditya V Nori, and Alex Aiken. Bias-variance tradeoffs in program analysis. *ACM SIGPLAN Notices*, 49(1):127–137, 2014.

[63] R. I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, 1980.

[64] Andreia Teixeira, Armando Matos, André Souto, and Luís Antunes. Entropy measures vs. kolmogorov complexity. *Entropy*, 13(3):595–611, 2011.

[65] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[66] Nikolai K Vereshchagin and Paul MB Vitányi. Kolmogorov's structure functions and model selection. *IEEE Transactions on Information Theory*, 50(12):3265–3290, 2004.

[67] Byron Emerson Wall. The history of probability, lecture slides for math 5400, 2007.

[68] M. Ward. The Closure Operators of a Lattice. *Annals of Mathematics*, 43(2):191–196, 1942.

[69] Martin Ziegler and Vasco Brattka. Turing computability of (non-)linear optimization. In *Proceedings of the 13th Canadian Conference on Computational Geometry (CCCG'2001)*, pages 181–184, 2001.