SOME APPLICATIONS OF FORMAL MATHEMATICS

A Dissertation

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

John V. Siratt

Peter Cholak, Director

Graduate Program in Mathematics

Notre Dame, Indiana

July 2024

SOME APPLICATIONS OF FORMAL MATHEMATICS

Abstract

by

John V. Siratt

The proof assistant is a powerful tool in formal methods. These programs allow the user to make formal mathematical statements and construct formal proofs. The connections with mathematics and logic provide motivation for a variety of problems. This dissertation applies formal mathematics to a selection of such problems. In Chapter 3 we establish methods to count the number of strong subtrees of height m in a binary tree of height n. Chapter 4 adapts notions of largeness defined for sets of natural numbers for use with trees. We examine matrix-weighted graph termination in Chapter 5, isolate two conditions from the termination criterion, and discuss their location in the arithmetic hierarchy. In Chapter 6 we show that the decision sets of problems such as termination have no maximal recursively enumerable subsets, and we discuss the position of the "smallest" non-trivial index sets in the arithmetic hierarchy. Chapter 7 reports on research into the characterization of neural nets and the formal verification of their properties.

This is for my family.

To my wife Chassidy and our children –

thank you for your love and support.

In memory of my parents and my brother –

thank you for encouraging me to aim high;

wish you could have been here.

CONTENTS

FIGURES

ACKNOWLEDGMENTS

I want to thank my committee – Peter Cholak (chair), Natasha Dobrinen, Aaron Dutle, David Galvin, and Sergei Starchenko. Their suggestions and notes have improved this work.

I am particularly grateful to my advisor Peter for his assistance and patience over the past five years. Aaron has shown me similar support in his role as my NASA Pathways mentor. The freedom they granted me in my research allowed me to combine my academic and professional interests.

I also appreciate the feedback of everyone else that read through my drafts. I particularly want thank David Chiang for his feedback on the neural net material, and Cesar Muñoz for his insights on termination and related topics. My wife, Chassidy, was not only pressed into service as an editor, but she also helped with many of the diagrams. This saved me hours of fiddling with TikZ, and for that I am forever grateful.

There are many others that helped me reach this point, although they may not have contributed directly to this dissertation. I'd like to take this opportunity to thank Lakeshia Legette Jones. It's hard to overstate her impact on my academic path – I probably would not have found my way to Notre Dame had she not played key roles in my introduction to NASA and in my successful application for an NSF Graduate Research Fellowship.

On the personal side, I couldn't have done this without my family. Their sacrifice and support are ultimately what enabled me to stay the course these past 12 years. Thank you. This work is dedicated to you.

Parts of this research was conducted under the National Science Foundation Graduate Research Fellowship Program grant DGE-1841556. Other parts were conducted while employed at the NASA Langley Research Center as part of the Pathways program. The remainder of the work was supported by Notre Dame's Richard and Peggy Notebaert Premier Fellowship. Without such generous financial assistance this work would not have been possible.

CHAPTER 1

INTRODUCTION

Broadly speaking, formal methods is the application of mathematical tools to hardware and software problems. One of the many tools used in this field is the proof assistant – this is a piece of software in which a user can specify objects and properties in some formal language, and construct machine verified proofs.

Proof assistants have found application in both formal methods and formal mathematics. It can sometimes be difficult to draw a clear line between the two. NASA's PVS library [35] includes an extensive collection of formally verified mathematics, much of which has been applied to practical problems.

We can however separate them by identifying their distinct motivations. Proof assistants are used for a variety of reasons in formal math, including, but not limited to, teaching, checking a difficult proof, and increasingly as a tool for new mathematical research. A nice recent survey on the use of proof assistants in mathematics is given by Avigad in [4]. This and related articles appeared in the April 2024 and July 2024 issues of the *Bulletin of the American Mathematical Society* dedicated to the effects of technology on mathematics (see [17] for an introduction to these issues by the guest editorial committee), so we may presume there is significant interest in such applications.

For formal methods, the motivation usually arises from the need to verify that a system satisfies a certain property. The website of the NASA Langley Formal Methods Team hosts a short list of system failures due to design errors, some of which resulted in millions of dollars in losses, and others in the loss of life (`https:`

`//shemesh.larc.nasa.gov/fm/fm-why-new.html`) [34]. In many of these cases, the mode of failure can be viewed as a violation of a property that might be easily stated in a formal language.

Formal methods makes significant use of applied mathematics. Any math that has real world application might find its way into a project that requires verification. This not only requires a broad familiarity with mathematics, but also creates the need for subject matter experts from specific fields of mathematics. We have encountered mathematicians working in formal methods coming from backgrounds in graph theory, algebraic topology, and various flavors of analysis.

Formal methods is also applied logic. A proof assistant implements a formal language with inference rules, so the connection with proof theory is apparent. In many proof assistants this is encoded with a type system, so type theory and the Curry-Howard correspondence comes into play. The tools of computability theory let us make statements about which things can be computed – including what kinds of proofs can be effectively automated. Model theory, modal logic, and many other ideas from logic also find a place in formal methods.

The following chapters will present research motivated by the author's work in formal methods. Together these chapters provide a broad, but my no means comprehensive, sample of the type of research problems that can arise in the field of formal methods.

1.1 Proof complexity

The first of these topics is that of proof complexity. When dealing with formal proofs it seems natural to gauge the difficulty of a proof by its length. It makes sense that a proof search would start with proofs of shorter length first, since there is a combinatorial explosion of possibilities as length increases. If a formal proof is required to certify that a design satisfies a particular property, the question of how long the proof may be has a direct impact on the number of man hours that will be dedicated to the job.

A mature theory of proof complexity could provide insights on how formal systems might be designed to lead to more efficient operations. A dearth of results which can be applied to practical proof systems, combined with the utility of such results, represents a potential of interesting research opportunities.

One interesting technique involves adapting conservation proofs into proofs of relative complexity using a form of proof theoretic forcing [25]. The method embeds a forcing interpretation for one proof system (or theory) within another, and has been applied in reverse mathematics to obtain proof complexity results for Ramsey's theorem for pairs.

The underlying conservation result [40] is general enough that it may be applied to similar combinatorics statements. The question of whether this method could be applied to some version of Milliken's tree theorem motivated the work that follows on tree combinatorics. Although we were not able to answer this, some interesting combinatorial results were obtained in the course of research.

Chapter 3 addresses the problem of how many *strong subtrees* of height m can be embedded within a binary tree of height n. This result does not have direct bearing on the problem of proof complexity. It was originally investigated as part of a potential argument using asymptotic combinatorics, and was completed out of curiosity.

Chapter 4 is concerned with adapting the largeness arguments used in the conservation result for Ramsey's theorem to trees. The work here was specifically designed to interface well with the work in [40]. This chapter includes some results demonstrating how largeness arguments for trees might proceed.

1.2 Termination analysis

Although the question of whether an algorithm terminates it the textbook case of an undecidable problem, it arises regularly during the course of formal methods work. Proving that an algorithm computes a total function is a core task of a formal methods practitioner. The failure of a critical piece of software to halt is a mode of failure that could have catastrophic consequences.

Termination analysis often involves restating the problem of totality in other forms that might be more conducive to certain forms of reasoning. The goal of these termination criterion is not to solve an undecidable problem, but to offer a tool that can be applied when a specific proof of termination is needed.

Chapter 5 will take one termination criterion and analyze it for sources of undecidability. The arguments in this chapter will make extensive use of techniques from computability theory that are used to place decision problems within the arithmetic hierarchy. The undecidable parts identified in the termination criterion can be targeted for a more heuristic approach which can allow partial automation of termination arguments.

Chapter 6 collects together some results that were inspired by the research into termination. The first section will discuss *minimal index sets*, which correspond to the collection of algorithms that compute a single fixed function. It will be shown that although each of these index sets are undecidable, they all fall at or below Π^0_2 on the arithmetic hierarchy. The second section will demonstrate that problems similar to establishing totality (specifically, those with decision sets which are *cylindrical*) do not have maximal recursively enumerable (r.e.) subsets. This may be interpreted as indicating there are not even theoretically ideal partial solutions.

1.3 Neural net verification

The final topic involves the problem of verifying neural nets. Machine learning is currently a hot topic, and with that comes the desire (and funding) to apply it to many different problems. When it comes to questions of safety and correctness, it is not at all obvious how we should approach neural net technologies.

The research highlighted in Chapter 7 is ongoing. We restrict our scope to *feed forward neural nets* which use the Rectified Linear Unit (ReLU) activation function to make binary classification decisions. We demonstrate that these kinds of neural nets can be expressed in an unquantified fragment of first order arithmetic – specifically, propositional formulas over vector inequalities on the input.

This gives us a way to fully specify the actions of a neural network of this type natively in the language of a proof assistant. Furthermore, the proofs are constructive, so an algorithm can be written to perform this translation. This work is being verified in the PVS [37] proof assistant. Although the result itself may be intuitively shown from previous results with moderate effort, the formalization of the work appears to be entirely novel.

1.4 Summary

These three topics are examples of a wide variety problems that can arise in formal methods research. Despite the diversity of the math and applications involved, the common theme is mathematical logic.

The history and development of formal methods is inextricable from mathematical logic. The idea of mechanizing proof was alive during the foundational crisis of mathematics. The implementations of theorem proving technology have progressed alongside the developments of modern computers, and sometime in the 90's it reached a point of maturity that allowed it to be used for commercial problems.

In the same way that cryptography reached into the heart of theoretical math and pulled number theory into the realm of applied sciences, formal methods provides a real-world relevance to logic research. Active communication between theory and practice could prove to be beneficial to both.

CHAPTER 2

BACKGROUND

This chapter is intended to serve as a reference for the remaining chapters. It contains definitions and discussions of material that should be addressed. Much of this material is referenced in multiple chapters, while some is essentially an extended footnote. For organizational simplicity it has been collected here in one place for easy referral.

The later chapters have been grouped into parts in accordance with their shared background and motivation. For ease of reference, the main sections here are numbered to correspond with those parts; Section 2.1 covers background for Part I, Section 2.2 covers Part II, and Section 2.3 covers Part III.

2.1 Combinatorics of binary trees

There are numerous results in reverse mathematics for Ramsey's theorem for pairs. In [12] Cholak, Jockush, and Slaman show that $\mathrm{RCA}_0{+}\mathrm{I}\Sigma_2{+}\mathrm{RT}^2_2$ is Π^1_1-conservative over $\mathrm{RCA}_0{+}\mathrm{I}\Sigma_2$ and that RT^2_2 does not imply $\mathrm{B}\Sigma_3$ over RCA_0. More recently, Chong, Slaman, and Yang [13] show that RT^2_2 does not imply $\mathrm{I}\Sigma_3$, and Patey and Yokoyama [40] show RT^2_2 is Π^0_3-conservative over $\mathrm{I}\Sigma_3$ (see Section 2.1.1).

Yet, in practical terms, showing that two systems can prove the same family of statements does not mean that they each can *easily* prove those statements. Knowing some system T' is Γ-conservative over another system T, we may ask, to what degree if any are the proofs of $\phi \in \Gamma$ shorter in T' than those in T? We call this the proof speed-up of T' over T. In [42] Pudlák referred to this study of the length of proofs as "the quantitative study of proofs" where we are concerned with the feasibility of the proofs rather than their existence.

In the same way that there are deep ties between reverse mathematics and computability, we have non-trivial connections between proof size and complexity theory. In the 1970's, Cook made explicit [15][16] the fact that $\mathcal{P} = \mathcal{NP}$ if and only if there is a proof system in which every propositional tautology has a "short" (polynomial) proof. In the time since, methods have been developed to extend this study of proof size to the systems that interest us in reverse mathematics. Similar to complexity theory, we quantify speed up by whether or not the changes in proof size are bounded by classes of arithmetic functions such as polynomial, exponential, or elementary (see Section 2.1.6 for a definition of elementary recursive functions).

In [25] Kołodziejczyk, Wong, and Yokoyama used methods from Avigad [3] to show that not only does RCA_0 only have a polynomial increase in size for $\forall\Sigma^0_2$ sentences in WKL_0+RT^2_2, but that the translation can be performed in polynomial time. In contrast, the authors also showed that RT^2_2 has non-elementary speed up over RCA^*_0 for Σ_1 sentences.

Milliken's tree theorem [31] is a Ramsey-like theorem for trees that covers substructures known as strong subtrees. In light of recent work on Milliken's tree theorem by d'Auriac, Cholak, Dzhafarov, Monin, and Patey [1], Cholak has asked if we can obtain similar results for Milliken's tree theorem for trees of depth 2. The focus will be on the binary tree version of this theorem.

2.1.1 Restricted induction

A common feature in reverse mathematics is limiting induction to classes of statements defined by their place in the arithmetic hierarchy (see Section 2.2.1).

Definition 2.1.1. *Define $I\Sigma^0_k$ to be induction restricted to Σ^0_k statements. In other words, we can only induct over existential statements with k alternating quantifiers*

– $\exists x_1, \forall x_2, \exists \cdots x_k, \psi(x_1, x_2, \ldots, x_k)$, *where* ψ *has only bounded quantification.*

With the pairing and projection functions, each quantified variable can represent an arbitrary number of variables as needed. It can be shown that $\mathrm{I}\Sigma^0_k$ is equivalent to $\mathrm{I}\Pi^0_k$ [45].

2.1.2 Binary trees and strong subtrees

Following the custom in computability theory, we may represent the infinite binary tree as the Cantor space $2^{<\omega}$ consisting of all finite binary strings. For finite binary trees we use the obvious analogue.

Definition 2.1.2. *The binary tree of height* n *is the set*

$$2^{<n} = \{\sigma \in 2^{<\omega} \mid |\sigma| < n\}$$

where $|\sigma|$ *is the length of the binary string* σ*.*

The ordering on $2^{<n}$ *is given by* $\sigma \preceq \tau$*, which is read as either* σ *precedes* τ*, or* τ *extends* σ*. Additionally,* $\sigma \prec \tau$ *denotes a proper extension.*

We use λ *to denote the root of such a tree, which is the empty string.*

General subtrees are not of interest to us here. We want the structure of the subtree to bear some resemblence to the parent. We might then define a subtree of height m as an order embedding of $2^{<m}$ into another binary tree. This is the type of subtree used in the previously mentioned work by Chubb, Hirst, and McNicholl [14], which was part of our motivation for weakening Milliken's tree theorem. These subtrees preserve the branching of $2^{<m}$ by ensuring that each subtree contains two incomparable extensions for any non-terminal vertex in the structure.

The strong subtrees which are required for Milliken's tree theorem must additionally preserve the meets and relative levels of its vertices. More formally,

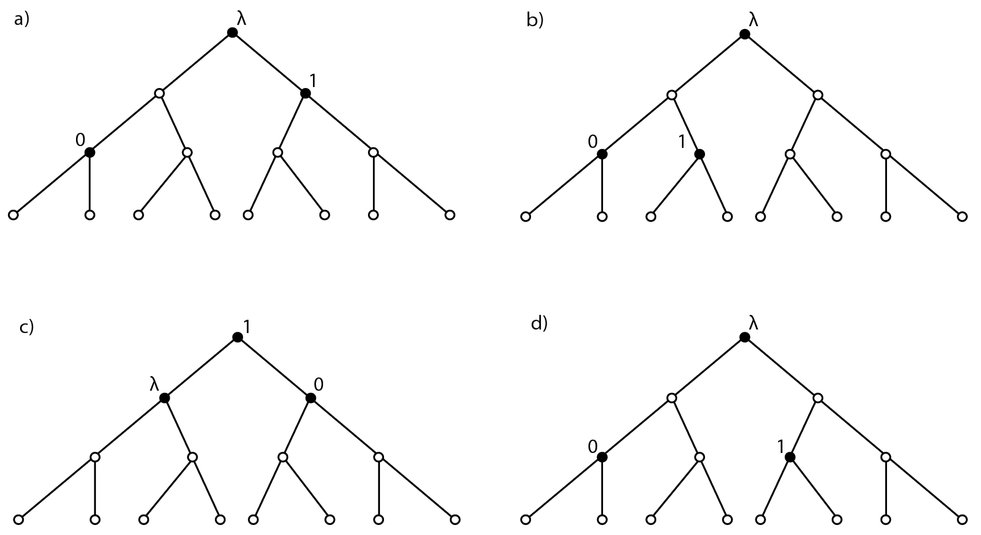

Figure 2.1. Diagrammatic representations of three mappings of $2^{<2}$ into $2^{<4}$. Only (d) is a strong subtree.

Definition 2.1.3. *A strong subtree $S \subseteq 2^{<n}$ of height m is given by a function $g : 2^{<m} \to 2^{<n}$ such that, for every $\sigma, \tau \in 2^{<m}$ we have the following properties:*

1. *$\sigma \preceq \tau$ if and only if $g(\sigma) \preceq g(\tau)$,*
2. *$|\sigma| = |\tau|$ implies that $|g(\sigma)| = |g(\tau)|$, and*
3. *for every $\gamma \in 2^{<m}$, $\gamma = \sigma \wedge \tau$ if and only if $g(\gamma) = g(\sigma) \wedge g(\tau)$.*

The function g is called a strong embedding. We denote the collection of all strong subtrees of $2^{<n}$ with height m by $\mathcal{S}_m(2^{<n})$.

Figure 2.1 shows several examples. The first three fail some criterion of a strong subtree. The example in (a) fails to preserve the relative levels of the vertices. That in (b) does not preserve meets. The third subtree (c) violates the requirement that the mapping be an order embedding. The final example (d) demonstrates one possible strong subtree of height 2 in $2^{<4}$.

With level preservation present in strong subtrees, it is often convenient to speak of the *level* a vertex occupies within a tree. With our current formalization of trees, this has some correspondence with the length of the string representing that vertex. It seems more natural to say the root occupies the first level, rather than the zeroth level, so we may define the level of σ to be $|\sigma| + 1$. Further, when dealing with an embedding of one tree within another, a vertex likely resides in different levels within each tree.

We then distinguish between the length of a vertex and the length of its image under the embedding. This seems a case of formality getting in the way, so we make the following definitions to return to something a bit more intuitive.

Definition 2.1.4. *Let S be a strong subtree of $T = 2^{<n}$ of height m, with $g : 2^{<m} \to 2^{<n}$ as its strong embedding. For $\sigma \in 2^{<m}$ we define $level_S(\sigma) = |\sigma| + 1$ and $level_T(\sigma) = |g(\sigma)| + 1$. We call these the level of σ in S, and the level of σ in T, respectively.*

This gives a sufficient notation to rigorously state much of what has only been mentioned informally to this point. For instance, the statement of weak Milliken's tree theorem for k-colorings of strong subtrees of height N within the binary tree $2^{<\omega}$ (WM_k^N) is: for every coloring $c : \mathcal{S}_N(2^{<\omega}) \to k$, there exists a c-monochromatic infinite strong subtree $S \subseteq \mathcal{S}_\omega(2^{<\omega})$. That is, for any coloring there is an infinite strong subtree for which all of its strong subtrees of height N have the same color.

The finite version for 2-colorings for fixed N is as follows.

Statement 2.1.5. *Finite WM_2^N: For every m there exists an n such that, for any coloring $c : \mathcal{S}_N(2^{<n}) \to 2$ there exists a c-monochromatic $S \subseteq \mathcal{S}_m(2^{<n})$.*

Here both the number of possible solutions and the number of possible colorings depend directly upon the number of strong subtrees of $2^{<n}$ with a particular height. Since our entire focus is on calculating these numbers, we will simplify the typography and let $f(n, m) = |\mathcal{S}_m(2^{<n})|$ for the duration. So for WM_2^N, we have $2^{f(n,N)}$ possible colorings and $f(n, m)$ possible solutions.

2.1.3 Largeness notions

The notion of largeness played an important role in the Paris-Harrington result [39]. Here the authors stated a version of the finite Ramsey theorem which required the solution to be *relatively large.* In that paper, a finite set of natural numbers X is said to be relatively large if $|X| \geq minX$.

While the finite Ramsey theorem is provable in Peano arithmetic, Paris and Harrington show that this large version is not. This was perhaps the first "natural" witness to Gödel's incompleteness theorem. Of particular interest is how this is proven - the Ramsey function for the Paris-Harrington statement can be shown to dominate all functions which are provably (total) recursive in Peano arithmetic.

This idea was extended and explored further by Ketonen and Solovay in [23], where we find a definition for a class of largeness notions defined by ordinals. Versions of this definition are also used in the work of various authors whose work is relevant here, so we will present the definition and related results largely as can be found in Patey and Yokoyama's recent work [40], with additional material taken from Kołodziejczyk's and Yokoyama's paper [24].

Definition 2.1.6. *For an ordinal $\alpha < \omega^\omega$ in Cantor normal form, and a finite set $X \subset \omega$, define $\alpha[m]$ by:*

- *If $\alpha = 0$, then $\alpha[m] = 0$.*
- *If $\alpha = \beta + 1$, then $\alpha[m] = \beta$.*
- *If $\alpha = \beta + \omega^n$ for $n > 0$, then $\alpha[m] = \beta + \omega^{n-1} \cdot m$.*

Let X be a finite set of cardinality k, and let $x_1 < \cdots < x_k$ be the elements of X. We say X is α-large if $\alpha[x_1]\cdots[x_k] = 0$ for recursive application of this definition. This is definable in $I\Sigma^0_1$.

By this definition we can see that every set is 0-large and any set X is $|X|$-large. We can also see that for a set X with cardinality greater than 1, X is ω-large if and only if $|X \setminus \{\min X\}| = \min X$. Hence ω-largeness more or less corresponds with the notion of a set being *relatively large*.

One particular result we will use extensively is found in [40] (where it is attributed to Hájek and Pudlák [22]) can be stated as follows.

Theorem 2.1.7. *Let $X < Y$ denote $\max X < \min Y$. Let X be a finite subset of ω.*

1. *X is $[\omega^n \cdot k]$-large if and only if there exists ω^n-large sets $X_1 < X_2 < \cdots < X_k$ such that $X = X_1 \sqcup X_2 \sqcup \cdots \sqcup X_k$.*
2. *For $n > 0$, X is ω^n-large if and only if there exists ω^{n-1}-large sets $X_1 < X_2 < \cdots < X_{\min X}$ such that $X = \{\min X\} \sqcup X_1 \sqcup X_2 \sqcup \cdots \sqcup X_{\min X}$.*

This is provable in $I\Sigma^0_1$.

In general, if a set X is $[\alpha_1 + \alpha_2]$-large, it can be decomposed as $X = X_1 \sqcup X_2$, where X_1 is α_2-large and X_2 is α_1-large. These decompositions are a primary tool in proving combinatorial statements which claim that any α-large set contains an β-large subset with a particular property.

The idea of largeness can be generalized by the following definition.

Definition 2.1.8. *Let $\mathcal{X}$ be a collection of finite subsets of ω. We say that $\mathcal{X}$ is a largeness notion if it is closed under supersets, and if every infinite set contains an element of $\mathcal{X}$ as a subset. Any element of $\mathcal{X}$ is said to be $\mathcal{X}$-large.*

Further, we say the largeness notion is regular when, if A is an $\mathcal{X}$-large set, B is an arbitrary set with $|B| \geq |A|$, and $f : A \to B$ is an order-preserving injection such that $f(n) \leq n$ for all $n \in A$, then B must also be $\mathcal{X}$-large.

This has recently found application in the proof-theoretic analysis of Ramsey's theorem. Patey and Yokoyama [40] used largeness to establish $\forall\Sigma^0_2$-conservation of WKL_0+RT^2_2 over RCA_0. This result was further strengthened by Kołodziejczyk, Wong, and Yokoyama [25] to demonstrate that this conservation only allows for polynomial improvement in proof size.

2.1.4 Conservation via largeness

The manner in which we extend largeness to trees is informed by our desire for compatibility with Patey and Yokoyama's generalization as outlined their Definitions 2.3, 2.4, 2.5, Proposition 2.5, and Theorem 3.1. In brief, the parts that primarily concern us might be stated by the following definition and result.

Definition 2.1.9. *Let Γ be a Ramsey-like statement $\forall f : \omega \to 2, \exists Y$, such that Y is infinite and $\Phi(f, Y)$, where Φ has the form: for every finite $G \subseteq Y$, some Δ^0_0 property ϕ holds for $f \restriction G$.*

Let $\alpha < \omega^\omega$, and $Z \subset \omega$ be a finite set. We say that Z is α-large(Γ) if for any $f : [0, |Z|) \to 2$, there exists an α-large $Y \subseteq Z$ such that $\Phi(f, b(Y))$ holds, where b is the order preserving bijection $Z \to [0, |Z|)$.

It's common to state conservation results in terms of the class of statements for which the result applies. In reverse mathematics, this often takes the form of notation from the analytic hierarchy (an extension, or generalization of the arithmetic hierarchy, see Section 2.2.1). The result we wish to state involves a class of statements denoted as $\forall\Sigma^0_2$. The class of $\forall\Sigma^0_2$ statements are those which are equivalent to Σ^0_2 formula with an outer block of universal quantification over both first and second order terms.

Theorem 2.1.10. *(Patey's and Yokoyama's Conservation Theorem.)*

If a Ramsey-like statement Γ as above is such that, for any $n \in \omega$, ω^n-large(Γ)ness is provably a largeness notion in $I\Sigma^0_1$, then $WKL_0 + \Gamma$ is a $\forall\Sigma^0_2$-conservative extension of $I\Sigma^0_1$.

For ω^n-large(Γ)ness, we get closure under supersets for free since, if a set contains a large solution, then every superset also contains that solution. To show that ω^n-large(Γ)ness is a largeness notion we only need to show that every infinite set contains a set of such largeness.

We are aided here by the fact that $\mathrm{I}\Sigma^0_1$ proves that ω^n-largeness is a largeness notion for any $n \in \omega$, as shown by Patey and Yokoyama. Following their lead, to prove that every infinite set contains an ω^n-large(Γ) set for some fixed n, we need only show that there exists some k such that an ω^k-large set contains an ω^n-large solution (all within $\mathrm{I}\Sigma^0_1$, of course).

Further, following Kołodziejczyk and Yokoyama [25], if we can show that there is in some sense a small function f such that, for every n, each $\omega^{f(n)}$-large set contains an ω^n-large solution, then we can strengthen the conservation result into a proof speedup result. Specifically, we would be able to show that the proof of any $\forall\Sigma^0_2$-formula provable in $\mathrm{RCA}_0 + \Gamma$ has a proof in RCA_0 that is at worst of polynomial increase in size. This strengthening is derived from Avigad's method of proof-theoretic forcing [3].

We will bridge the gap from these results to trees by fixing an encoding of the infinite binary tree onto ω and viewing the order isomorphism b as a function which labels vertices with the elements of a large set of numbers. This allows us to not only make statements about binary trees in the language of $\mathrm{I}\Sigma^0_1$, but to also make statements about the largeness of these *labelings* of trees.

First we wish to justify the intended application of the conservation theorem given in Theorem 2.1.10 to trees. To do so we appeal to another result in [40].

Theorem 2.1.11. *Let Γ be a formula of the form $\forall X \exists Y, \Psi(X,Y)$, where Ψ is Σ^0_3. There exists a Ramsey-like formula $\forall X, \exists Z,\ Z\ is\ infinite\ \wedge\ \Phi(X,Z)$ (as in Definition 2.1.9) such that*

$$WKL_0 \vdash \forall X, [(\exists Y, \Psi(X,Y)) \leftrightarrow (\exists Z,\ Z \text{ is infinite } \wedge\ \Phi(X,Z))].$$

To show that the conservation theorem applies to a tree theorem, it suffices to show that the property required of a solution is Σ^0_3-expressible. Provided our embeddings of trees and their substructures are computable, the common requirements of a solution being infinite and monochromatic are Π^0_2 and Π^0_1, respectively. The remaining requirement pertains to the structure of the solution itself.

For Milliken's tree theorem this means strong binary subtrees.

Theorem 2.1.12. *Given computable encodings, the property stating that a subset T of the binary tree is a strong subtree is Π^0_2.*

Proof. A subtree must satisfy three conditions to be strong. First, for each $\sigma, \tau \in T$, their meet must be in T. This is Π^0_1 since the meet is bounded by σ and τ, and the property of being a meet is computable.

Second, for each $\sigma \in T$ there must exist two immediate successors. This is Π^0_2 because both $\sigma \prec \tau$ and the statement $\forall \rho, \sigma \prec \rho \rightarrow \tau \preceq \rho$ are computable.

Finally, we require that the immediate successors be on the same level of the binary tree, which is also computable.

Using prenex normal form we can then obtain a Π^0_2 statement of the strong subtree property.

□

2.1.5 A bit about encodings

For the most part, encoding trees and their substructures in ω is routine. However, due to the order preservation needed in the application of Definition 2.1.9, we need a firm notion of what part of the binary tree an initial segment of ω conforms to. To do so we need to fix a linear ordering over the binary tree such as the following, which is illustrated in Figure 2.2.

Definition 2.1.13. *Let $<$ be defined over $2^{<\omega}$ as the quasi-lexical ordering $\sigma < \tau$ when:*

- $|\sigma| < |\tau|$, *or*

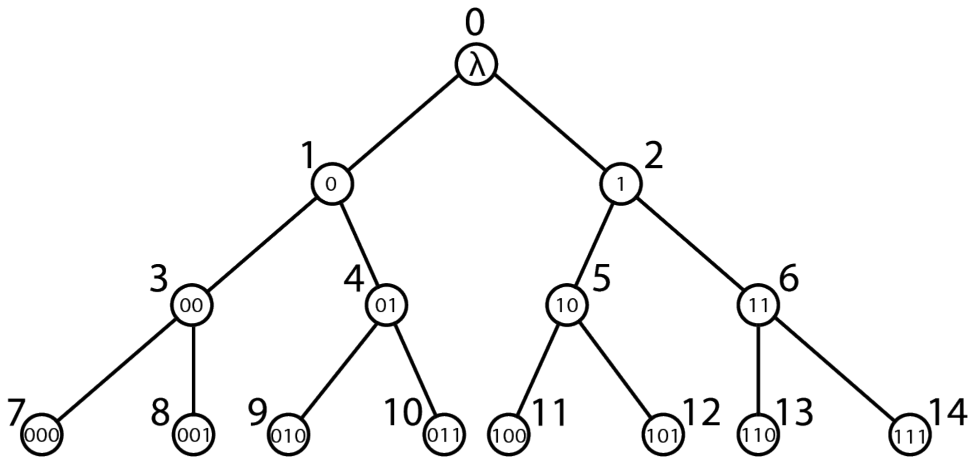

Figure 2.2. Mapping the binary tree to ω in such a way that an initial segment $[0, |Z|)$ always contains a binary tree of some size. This along with *labelings* covered in Chapter 4 take care of technical requirements in Definition 2.1.9

- $|\sigma| = |\tau|$ *and* $\rho^\frown 0 \preceq \sigma$ *and* $\rho^\frown 1 \preceq \tau$*, where* $\rho = \sigma \wedge \tau$*.*

Representing the vertices of the tree as finite binary strings, we can define the resulting order isomorphism in $\mathrm{I}\Sigma^0_1$.

Theorem 2.1.14. *Let* $bin(\sigma) = \sum_{i=0}^{|\sigma|-1} \sigma(i) \cdot 2^i$*. Then* $f(\sigma) = 2^{|\sigma|} - 1 + bin(\sigma)$ *is an order isomorphism from* $(2^{<\omega}, <)$*, as defined above, to* $(\omega, <)$*. This is provable in* $I\Sigma^0_1$*.*

Proof. This is routine arithmetic and only requires induction over computable predicates.

□

2.1.6 Elementary recursive functions

The elementary recursive functions are a subset of the primitive recursive functions which can be defined by replacing primitive recursion with bounded summations and products.

Definition 2.1.15. *A computable function is elementary recursive if it is:*

- *the zero function,*
- *the successor function,*
- *a projection function, or*
- *the monus function* $x \dot{-} y = \{\max x - y, 0\}$,

or belongs to the closure of these functions under:

- *function composition,*
- *bounded summation* $f(m, x_1, \ldots, x_n) = \sum_{i=0}^{m} g(i, x_1, \ldots, x_n)$*, or*
- *bounded product* $f(m, x_1, \ldots, x_n) = \prod_{i=0}^{m} g(i, x_1, \ldots, x_n)$.

These have proven useful in distinguishing proof complexity such as in work by Kołodziejczyk, Wong, and Yokoyama [25].

It is also notable for the fact that the elementary recursive functions define a complexity class that lies strictly between exponential time and primitive recursion. Further, removing bounded products from the above definition gives us the class of Skolem elementary functions (or the lower elementary recursive functions)[50], which yields a complexity class strictly below exponential time.

2.2 Verification and index sets

The field of formal methods applies mathematical logic to the verification of hardware and software [34]. There are many techniques for doing so, and we may apply these at various stages of development anywhere from initial design specifications to implementation code. One technique is to formally describe and reason about computational processes in a proof assistant such as the Prototype Verification System (PVS) [37].

While research in this field is far more varied than we will communicate here, one typical type of problem would be determining if an algorithm meets some behavioral requirement. This corresponds to a fundamental construction in computability theory known as the *index set*.

Definition 2.2.1. *Let $\{\phi_i\}$ be an effective enumeration of algorithms. We say that each i is an* index.

Then $X \subseteq \omega$ is an index set *if and only if, for every $i \in X$ and every $j \in \omega$, if ϕ_i and ϕ_j compute the same function, then $j \in X$.*

Since algorithms which compute the same function must have identical output, any property relating to the output defines an index set.

To ask then if the output of some algorithm ϕ_c exhibits a particular property is to ask if c belongs to the index set defined by that property. By Rice's theorem we know that there are only two computable index sets – the set containing all algorithms, and the set containing no algorithms. So in general, such verification is an undecidable problem.

This is one reason why formal verification presents us with such interesting and challenging problems. Regardless of how many advances we make, the process will never be one that can be completely automated, so there will always be a place for human creativity in the process.

2.2.1 Computability theory

Undecidable problems are not all the same. Perhaps the most practical distinction that can be made is that of the semi-decidable problems.

Definition 2.2.2. *A set $S \subseteq \omega$ is* computable *if its characteristic function χ_S is a computable function. It is* recursively enumerable *(r.e.) if S is the domain of a computable function, or equivalently the range of a total computable function.*

Computable sets are the decidable problems and their characteristic functions are decision procedures. An r.e. set is a semidecidable problem and a partial decision procedure can be defined in terms of either domain or range of a computable function.

Beyond this, we may classify sets (and the decision problems they represent) by the arithmetic hierarchy and reducibilities.

Definition 2.2.3 (Arithmetic hierarchy)**.** *Let Φ, Ψ be formulae in first order arithmetic.*

- *A formula Φ is Σ^0_0 and Π^0_0 if it contains no quantifiers.*
- *A formula $\exists \overline{x} \Phi(\overline{x})$ is Σ^0_n for $n > 0$ if Φ is Π^0_{n-1}.*
- *A formula $\forall \overline{x} \Phi(\overline{x})$ is Π^0_n for $n > 0$ if Φ is Σ^0_{n-1}.*

A set S is said to be Σ^0_n if there exists a Σ^0_n formula Φ such that

$$S = \{i \in \omega \mid \Phi(i)\}.$$

Similarly, it is Π^0_n if Φ is a Π^0_n formula. A set S is said to be Δ^0_n if there is a Σ^0_n formula Φ and a Π^0_n formula Ψ such that

$$\{i \in \omega \mid \Phi(i)\} = S = \{i \in \omega \mid \Psi(i)\}.$$

The Turing degrees are perhaps the most recognizable hierarchy, but any reducibility induces its own degree hierarchy. We will prefer the finer degree structure provided by 1-reducibility so that we may discuss completeness with respect to the arithmetic hierarchy.

Definition 2.2.4. *A set A is* 1-reducible *to B (written $A \leq_1 B$) if there exists a computable injective function f such that $x \in A$ if and only if $f(x) \in B$.*

Sets A and B are said to be 1-equivalent *($A \equiv_1 B$) if $A \leq_1 B$ and $B \leq_1 A$.*

A set S is Σ^0_n-complete *if S is Σ^0_n and for every Σ^0_n set X we have $X \leq_1 S$. We define Π^0_n-completeness in the same fashion.*

Closely related to the arithmetic hierarchy we have the following results collectively known as Post's theorem. These results establish the direct relationship between the arithmetic hierarchy as a syntactic classification and the relative computability of sets (computability using an oracle set).

Theorem 2.2.5 (Post's theorem)**.** *Let S be a set of natural numbers and $\emptyset^{[n]}$ be the nth jump (these are the relativized halting sets).*

- *S is Σ^0_{n+1} if and only if S is r.e. using a Π^0_n oracle.*
- *S is Σ^0_{n+1} if and only if S is r.e. using a Σ^0_n oracle.*
- *For $n > 0$ the nth Turing jump of the empty set $\emptyset^{[n]}$ is Σ^0_n-complete.*
- *S is Σ^0_{n+1} if and only if S is r.e. using $\emptyset^{[n]}$ as oracle.*
- *S is Δ^0_{n+1} if and only if S is relatively computable using $\emptyset^{[n]}$ as oracle.*

Using such hierarchies we can speak of a set X as having a degree of undecidability. Further, if we have a partial solution, expressed as a subset of X, then we can computably obtain a partial solution for any problem lower on the hierarchy via the functions witnessing reducibility.

2.2.2 Termination

One formal verification problem we will single out is termination. This is the problem of verifying that an algorithm will terminate on all possible inputs.

Although undecidable in general this is an important practical problem. For instance, it could be considered a serious safety issue if the software in an automated vehicle were shown to enter an infinite loop when presented with certain sensor data.

First there is a bit of notation to get out of the way.

Definition 2.2.6. *Fix an effective enumeration of Turing machines (or equivalent model of computation). We denote the ith Turing machine as ϕ_i. Further,*

- *for $n \in \omega$ let $\phi_i(n) \downarrow$ indicate that ϕ_i is defined at n,*
- *similarly let $\phi_i(n) \uparrow$ indicate that ϕ_i is undefined at n,*
- *$\phi_{i,t}(n)$ indicates the partial computation of ϕ_i on input n after t steps (concretely, the configuration of the Turing machine after t steps),*
- *$\phi_{i,t}(n) \downarrow$ indicates that ϕ_i with input n reaches the halting state in t or fewer steps.*

As witnessed by the Kleene T-predicate, determining whether $\phi_{i,t}(n) \downarrow$ is primitive recursive in i, t, n.

This notation allows us to define the following set.

Definition 2.2.7. *Define $Tot = \{i \mid \forall n, \exists t, \phi_{i,t}(n) \downarrow\}$ as the index set containing the indices for every algorithm which is defined for every input.*

This captures the problem of determining termination and is known to be Π^0_2-complete.

2.2.3 Termination analysis

Despite the impossibility of finding a general solution, there have been notable successes in practical termination analysis. One example is the work of Manolios and Vroon [28] wherein they applied their method of calling context graph (CCG) analysis to a collection of libraries for the ACL2 theorem prover. They reported that their algorithm successfully proved the termination of 98.7% of the 10,000 functions included.

This method has received further refinement. In [2] Avelar modified Manolios's and Vroon's work assigning measures to CCGs by instead using a matrix weighted graph (MWG). This allows the analysis to be accomplished by examining the products of the matrices along the possible paths of execution. Muñoz, Ayala-Rincón, Moscato, Dutle, Narkawicz, Almeida, Avelar, and M Ferreira Ramos formalized several termination criteria in the PVS proof assistant and established their equivalence

[33], as well as providing Dutle's procedure – an algorithm for determining if a specific MWG meets the MWG-termination criterion.

This implies that there is no way to take an algorithm and effectively construct the required MWG for that algorithm in a universal way. Otherwise we would obtain a decision procedure for Tot by composing this process with Dutle's procedure. It follows then that the undecidability inherent in totality must be present in the construction of correct MWGs. This process can be naturally broken down into steps, each with separate requirements to ensure that the resulting MWG is a correct representation of the input algorithm. We will provide an analysis of this construction from a computability perspective.

This termination criterion is not unique in holding logical interest. Steila and Yokoyama [48] performed a reverse-mathematical analysis of the termination theorem put forward by Podelski and Rybalchenko [41]. Where MWG-termination is formulated for a functional language using general recursion, this theorem is stated in terms of transition-based programming. As part of their work, Steila and Yokoyama obtain a stronger result establishing sufficient conditions not only for termination, but for termination within particular bounds.

Indeed any sufficient condition for termination characterizes a subset of Tot. Despite the apparent importance of such subsets in application, they do not seem to make a notable appearance in the literature of computability theory. It may be hoped then that mining the field of termination analysis could potentially provide interesting logical content.

2.3 Neural networks

Neural networks come in many forms, but their common building block is the *artificial neuron*. In modern applications, an artificial neuron takes the form $f(w \cdot x + b)$, where w is a vector of weight values, b is a bias value, and f is an activation function.

The most general kinds of neural networks are recurrent. Such networks may have cycles where information can be passed back to previous neurons. This can make the analysis of such networks difficult.

2.3.1 Feed forward neural networks

A neural network with no recurrence has a natural partial ordering on its neurons. When we divide these neurons into layers, we have a feed forward neural net. We will specifically be considering feed forward neural nets with totally connected layers.

Definition 2.3.1. *Define a* layer *as a collection of neurons which share the same input vector and for which the output of each neuron is a component in the output vector of the layer. A* (totally connected) feed forward neural net *is a sequence of layers $L_1, \ldots, L_n$ such that:*

- *the input to the network is the input to L_1,*
- *for each $i < n$, the output of L_i is the input to L_{i+1}, and*
- *the output of L_n is the output of the network.*

We say such a network is totally connected because when expressed as a diagram every neuron of one layer is connected to every neuron in the following layer.

This type of neural net allows us to extend the vector representation of neurons to entire layers as a matrix equation $F(W \cdot x + B)$. More explicitly, let $N_1, \ldots, N_k$ be neurons in a layer L, and let F be the function mapping the activation function to the elements of a vector. We then have

$$L(x) = F\left(\begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix} \cdot x + \begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix}\right),$$

where w_i is the weight vector of N_i, and b_i is its bias.

We say a feed forward network has a *depth* of n if it has exactly n layers. We say it has a *width* of k when the largest layer has k neurons.

A depth n feed forward network can then be expressed by

$$L_n(L_{n-1}(\cdots L_1(x)\cdots)) = F(W_n \cdot F(W_{n-1} \cdot F(\cdots W_1 \cdot x + B_1 \cdots) + B_{n-1}) + B_n).$$

A proof for the following remark would be technical and not very enlightening, but the idea behind it can be clearly expressed on a whiteboard.

Remark 2.3.2. *Given two feed forward neural nets f, g, we can easily compose these into another feedforward net $f(g(x))$. Differences in the output dimension of g and the input dimension of f do not pose a real problem since we can either ignore extra inputs with zero-weighted dummy neurons, or provide default values through dummy neurons where a higher dimension is needed.*

Similarly, we can construct a neural network that computes f and g in parallel and concatenates their output (or performs some other combination) by filling in the missing connections with connections with zero weights.

2.3.2 Binary classifiers

The classification of data is a common task to which machine learning is applied. This may involve an arbitrary number of categories, but we can always decompose a complex classification problem into multiple binary classification problems. We may then consider characteristic functions as the fundamental unit of a classification task.

We call a neural net that computes such a characteristic function a *binary classifier.* While it is common for the activation function to be consistent throughout a network, constraining the output to $\{0, 1\}$ requires some form of filtering. We will model this final component of a binary classifier as a feed forward layer with the activation function $step(x) = 1$ when $x > 0$, and $step(x) = 0$, otherwise.

Definition 2.3.3. *We say a feed forward neural net is an f-activated binary classifier if the final layer consists of a single neuron using the step function as its activation, and all of the other layers use f as the activation function.*

2.3.3 Expressiveness of neural nets

Perhaps the most basic informal question regarding neural nets is asking which problems they can be applied to. This problem of establishing how expressive a formalism is has a long history in logic (e.g. the equivalence of sequential logic with Büchi automata [6]). Such questions for neural nets began to be answered early, with McCulloch and Pitts observing that cycle-free networks of their artificial neurons computed a subset of the computable functions, but if augmented by an infinite tape and a reading mechanism were equivalent to Turing machines [29].

Universal approximation theorems are often used to show that a particular neural net architecture can represent functions of interest. The following result for arbitrary depth (and bounded width) ReLU-activated feed forward neural networks was shown by Lu, Pu, Wang, Hu, and Wang [27].

Theorem 2.3.4. *For any Lebesgue-integrable function $f : \mathbb{R}^n \to \mathbb{R}$ and any $\varepsilon > 0$, there exists a ReLU-activated feed forward network F of no more than width $n + 4$ such that*

$$\int_{\mathbb{R}^n} |f(x) - F(x)| dx < \varepsilon.$$

If we apply this result to a binary classifier, we find it is only limited utility. In practice we are only applying F to a countable subset of $X \subset \mathbb{R}^n$, so it is possible that F can produce the wrong answer for every $x \in X$ even while satisfying the above. The classic universal approximation result that applies to similar networks of arbitrary width and bounded depth is stated in terms of the sup norm, and so globally bounds the error, but is restricted to functions on a compact subset of $\mathbb{R}^n$ [19].

For the purposes of verifying properties that might be relevant for practical safety arguments, we may need to look at expressiveness through a different lens. Because many of the tools of formal methods are built on the foundations of mathematical logic, it seems reasonable to look to logic for an alternative.

Indeed, the connection between logic and machine learning has produced interesting connections. There exists a connection between between NIP (non-independence property) in model theory and PAC (probably approximately correct) learning, a common framework for a statistical analysis of machine learning [26][7]. At least as far back as 1967 Gold was exploring the question of which formal languages were computably learnable [20], and this connection with computability theory and computational learning theory has continued in recent years with the identification of connections with algorithmic randomness [5][30].

On the topic of expressibility, recent work by Chiang, Cholak, and Pillay [9] has extended existing work on a type of neural network known as a transformer encoder. The authors show that transformer encoders can express every formula in an extended first order logic called FOC[+;MOD], which in turn can express every transformer encoder with parameters of finite precision.

The formal properties of a neural network can be difficult to establish since even networks without recurrence are a complex composition of vector operations and nonlinear activation functions. If we can first translate a network into a more familiar formal language we might be presented with a more tractable problem. Such a transformation is not only helpful, but in practice should always be possible.

While we can perform clever tricks for neural nets using unrestricted real numbers as parameters to perform oracle computations, neural nets deployed on actual computer systems are bounded by the common limits of computation. A practical neural net that classifies its inputs as either being in one category or another is performing the role of a characteristic function for a computable set, and so can be expressed as a formula in the first order language of arithmetic using only bounded quantification.

This argument can even be extended to the full learning process. If we conceptualize the learning process as sequentially taking in data to produce a better model, we can envision this as a computable sequence of algorithms that should, ideally, converge toward the required function. This kind of convergence can only approximate Δ^0_2 sets (by the Limit Lemma, see for instance [47]). Any classification problem that can be learned through such a process must then be expressible with a formula in the first order language of arithmetic with no more than two alternating quantifiers.

The expressibility of ReLU-activated binary classifiers will be explored in Chapter 7. This is closely related to a number of results. Notably, we have the following equivalence result.

Theorem 2.3.5 (Arora, Basu, Mianjy, and Mukherjee [32])**.** *The functions computed by ReLU-activated binary classifiers with a single output dimension are precisely the functions $\mathbb{R}^n \to \mathbb{R}^{\geq 0}$.*

Similarly, we characterize the step-activated binary classifiers in Chapter 7. Together with the above theorem this allows us to that both step-activated and ReLU-activated binary classifiers compute the same functions.

The first steps in our original proof sketch took a similar approach to this, but we were unable to translate the informal reasoning into formal logic in a way that cleanly generalized to inputs of arbitrary dimension. A new argument was needed.

The proof in Chapter 7 has much more in common with a proof by Pan and Srikumar [38]. On review it turned out this work was very similar to ours but restricted to the case of networks of depth two. Our result can be seen as an extension of their work and provides a constructive mapping from ReLU-activated binary classifiers to a fragment of unquantified first order arithmetic.

There is a wealth of similar work out there, such as [51] [8] [36] [21]. However, to date we have found none that specifically serve our purpose.

PART I

COMBINATORICS OF BINARY TREES

CHAPTER 3

STRONG SUBTREE EMBEDDINGS

This chapter will discuss the number of strong subtrees of height m in the binary tree of height n, represented by the following function.

Definition 3.0.1. *Let* $S(n, m) = |\mathcal{S}_m(2^{<n})|$.

The collection of strong subtrees $\mathcal{S}_m(2^{<n})$ (see Section 2.1 for background definitions) may be partitioned according to the levels of $2^{<n}$ from which a subtree draws its elements.

Definition 3.0.2. *Define* $\mathcal{T}_{n,m}(\ell_1 < \cdots < \ell_m)$ *to be the collection of every subtree* $T \in \mathcal{S}_m(2^{<n})$ *such that level* i *of* T *is taken from level* ℓ_i *of* $2^{<n}$.

It follows from this definition that $\mathcal{S}_m(2^n) = \bigcup_{1\le\ell_1<\cdots<\ell_m\le n} \mathcal{T}_{n,m}(\ell_1, \cdots, \ell_m)$. Summing the count for each partition, the following will be shown in Section 3.1.

Theorem 3.0.3. *Let* $x_0 = 0$. *Then*

$$S(n, m) = \sum_{1\le x_1<\cdots<x_m\le n} \prod_{i=1}^{m} \left(2^{x_i - x_{i-1} - 1}\right)^{2^i - 1}.$$

Implementing this function as in Appendix A, some initial values of the function table can be computed (see Figure 3.1). Monotonic sequences can be extracted from the rows and from diagonals where $n > m$. Comparing these sequences to those documented in the On-line Encyclopedia of Integer Sequences (OEIS) [46] only a few connections were found.

$S(n,m)$	$n=1$	2	3	4	5	6	7	8
$m=1$	1	3	7	15	31	63	127	255
2	0	1	7	35	155	651	2667	10795
3	0	0	1	23	403	6603	106299	1703451
4	0	0	0	1	279	71827	18394315	4709050939
5	0	0	0	0	1	65815	4313323667	282677998234827
6	0	0	0	0	0	1	4295033111	18447026751295461523
7	0	0	0	0	0	0	1	18446744078004584727

Figure 3.1. Table illustrating some values of $S(n,m)$, counted as described in Theorem 3.0.3.

The degenerate case $S(n,1) = 2^n - 1$ is easy to identify since this simply counts the number of vertices in a binary tree of height n. The sequence generated by $S(n,2)$ would appear to correspond to the Gaussian binomial coefficient $\binom{n}{2}_2$ (OEIS A006095). More indirect relations to existing sequences seemed to exist in the table, motivating further study into ways to simplify the expression found in Theorem 3.0.3.

Counting in a different manner provides a more convenient expression for the diagonal sequences. The general solution for the diagonals consists of nested sums. The following is a corollary to Theorem 3.4.1.

Theorem 3.0.4. *Let* $þ_0^a = 1$*,* $þ_b^0 = 0$ *for* $b > 0$*, and* $þ_b^a = \sum_{i=1}^{a} 2^{2^i} \cdot þ_{b-1}^i$ *otherwise. Then*

$$S(m+k,m) = \sum_{i=0}^{k} \left(2^{k+1-i} - 1\right) þ_i^{m-1}.$$

In this form it can be seen that

$$S(m+1,m) = 3 + \sum_{i=1}^{m-1} 2^{2^i}$$

which computes a sequence closely related to OEIS A060803. More generally, this form illustrates that fixing a diagonal with a specific k allows for the expansion of the outer sum and the elimination of the recursive application of summation represented by þ.

3.1 Initial count

The first task is to count the number of subtrees in each partition $\mathcal{T}_{n,m}$.

Lemma 3.1.1. *For $n > m$ and $1 \leq \ell_1 < \cdots < \ell_m \leq n$, with $\ell_0 = 0$:*

$$|\mathcal{T}_{n,m}(\ell_1, \dots, \ell_m)| = \prod_{i=1}^{m} \left(2^{\ell_i - \ell_{i-1} - 1}\right)^{2^{i-1}}.$$

Proof. Fix n. If $m = 1$ we need only choose a single vertex from a fixed level ℓ_1, so there are $2^{\ell_1 - 1} = \left(2^{\ell_1 - \ell_0 - 1}\right)^{2^0}$ possibilities.

Suppose then that the claim holds for $m = k$ and consider $m = k + 1$, and fix $1 \leq \ell_1 < \cdots < \ell_{k+1} \leq n$. By our induction hypothesis we have $|\mathcal{T}_{n,k}(\ell_1, \dots, \ell_k)| = \prod_{i=1}^{k} \left(2^{\ell_i - \ell_{i-1} - 1}\right)^{2^{i-1}}$.

Fix $S \in \mathcal{T}_{n,k}(\ell_1, \dots, \ell_l)$. By taking elements from level ℓ_{k+1} of $2^{<n}$ we can extend S into a height $k + 1$ subtree.

The leaves of S consist of 2^{k-1} elements of level ℓ_k of $2^{<n}$. There are $2^{\ell_{k+1} - \ell_k - 1}$ possible extensions of $\sigma^\frown 0$ and $2^{\ell_{k+1} - \ell_k - 1}$ possible extensions of $\sigma^\frown 1$.

We have then

$$\left(2^{\ell_{k+1} - \ell_k - 1} \cdot 2^{\ell_{k+1} - \ell_k - 1}\right)^{2^{k-1}} = \left(2^{\ell_{k+1} - \ell_k - 1}\right)^{2^k}$$

extensions of S.

Multiplying by the possible choices of S we have

$$\left(2^{\ell_{k+1} - \ell_k - 1}\right)^{2^k} \cdot \prod_{i=1}^{k} \left(2^{\ell_i - \ell_{i-1} - 1}\right)^{2^{i-1}} = \prod_{i=1}^{k+1} \left(2^{\ell_i - \ell_{i-1} - 1}\right)^{2^{i-1}}.$$

□

This is then used to prove the first result given at the beginning of the chapter.

Proof. (Theorem 3.0.3)
Using Lemma 3.1.1, sum over all possible partions.

□

3.2 A different way to count

The method of counting in Section 3.1 seems messy. It involves a complicated sum over partitions, and an inductive argument over the number of fixed levels in each partition. It would be preferable to induct directly on the number of levels in the tree and establish a recurrence relation.

The strong subtree structure complicates this. Consider the problem of trying to calculate $S(n+1, m+1)$. We may do this in terms of $S(n, m)$ by considering how many ways we can extend each subtree in $\mathcal{S}_m(2^{<n})$ with vertices taken from level $n+1$.

For any particular subtree $S \in \mathcal{S}_m(2^{<n})$ this is dependent upon the distance between the leaf layer of S and n, leading to an argument similar to that found in the proof of Lemma 3.1.1 where we must consider the cases for each leaf layer.

It is easier to count the number of possible extensions when the layer to be extended can be fixed. It is probably not surprising then that the method which first showed success was a hybrid approach that counted $S(n, m)$ in terms of $S(n-1, m)$, $S(n-2, m)$, and a remainder term counting subtrees with fixed root and leaf layers.

Definition 3.2.1. *Let $T = 2^{<n}$. Define:*

- *the upper subtree $V = 2^{<n-1}$,*
- *the lower left subtree $W = \{\sigma \in T \mid 0 \preceq \sigma\}$, and*
- *the lower right subtree $Z = \{\sigma \in T \mid 1 \preceq \sigma\}$.*

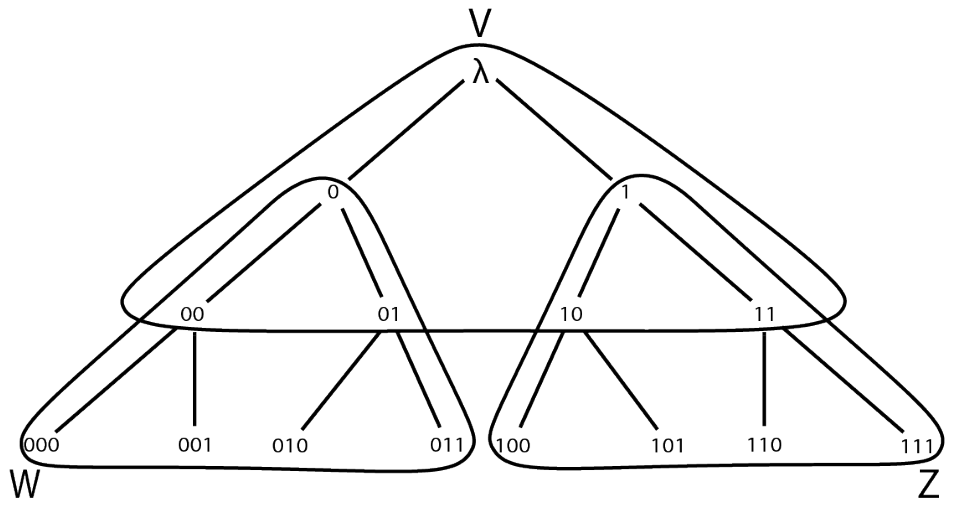

Figure 3.2. Example of V, W, Z as defined in 3.2.1.

The configuration of these subtrees are illustrated in Figure 3.2. For T of height n, each subtree V, W, Z has height $n-1$. If we count the strong subtrees in V, W, Z we double count those in the intersections $V \cap W$ and $V \cap Z$. Conveniently, these intersections are subtrees of height $n-2$.

The only obstacle to this providing a recurrence relation in two terms is that it fails to account for the subtrees rooted at λ with leaves in layer n.

Definition 3.2.2. *Let $T \subseteq 2^{<n}$ be a strong subtree of height m. If the root of T is λ and the leaves of T are leaves of $2^{<n}$, we call T a stretched subtree of $2^{<n}$. Further, let $\rho(n,m)$ be the number of stretched subtrees in $2^{<n}$ of height m.*

Theorem 3.2.3. *We have the partial recurrence relation,*

- $S(n,m) = 0$, *if* $n < m$,
- $S(n,m) = 1$, *if* $n = m$,
- $S(n,m) = 3 \cdot S(n-1,m) - 2 \cdot S(n-2,m) + \rho(n,m)$, *otherwise.*

Proof. The base cases are apparent, so assume $m < n$. We use the subtrees V, W, Z of $2^{<n}$ as defined above. For the subtrees at the intersections, we define the shorthand

$T_V = \mathcal{S}_m(V)$, $T_W = \mathcal{S}_m(W)$, $T_Z = \mathcal{S}_m(Z)$, $T_{VW} = \mathcal{S}_m(V \cap W)$, and $T_{VZ} = \mathcal{S}_m(V \cap Z)$. Additionally, let $R \subset \mathcal{S}_m(T)$ be the collection of stretched substrees of height m in $2^{<n}$.

Fix some $T \in \mathcal{S}_m(2^{<n})$ with i being the level of $2^{<n}$ where T is rooted and j being the level where T has leaves. There are four distinct and mutually exclusive possibilities, each following by definition:

1. $i = 1$ and $j = n$ if and only if $T \in R$,
2. $i > 1$ and $j < n$ if and only if $T \in T_{VW}$ or $T \in T_{VZ}$,
3. $i > 1$ and $j = n$ if and only if $T \in T_W \setminus T_{VW}$ or $T \in T_Z \setminus T_{VZ}$,
4. $i = 1$ and $j < n$ if and only if $T \in T_V \setminus [T_{VW} \cup T_{VZ}]$.

So $\mathcal{S}_m(2^{<n})$ is the disjoint union

$$R \bigcup \left(T_{VW} \cup T_{VZ}\right) \bigcup \left([T_W \setminus T_{VW}] \cup \ [T_Z \setminus T_{VZ}]\right) \bigcup T_V \setminus [T_{VW} \cup T_{VZ}] \, .$$

Since $T_{VW} \cap T_{VZ} = \emptyset$, $T_{VW} \subset T_W$, $T_{VZ} \subset T_Z$, and both $T_{VW}, T_{VZ} \subset T_V$, it follows that

$$\begin{aligned}
|\mathcal{S}_m(2^{<n})| &= |R| + |T_{VW}| + |T_{VZ}| + |T_W \setminus T_{VW}| + |T_Z \setminus T_{VZ}| \\
&\quad + |T_V \setminus [T_{VW} \cup T_{VZ}]\,| \\
&= |R| + |T_{VW}| + |T_{VZ}| + |T_W| - |T_{VW}| + |T_Z| - |T_{VZ}| + |T_V| \\
&\quad - [|T_{VW}| + |T_{VZ}|] \\
&= \rho(n,m) + S(n-2,m) + S(n-2,m) + S(n-1,m) - S(n-2,m) \\
&\quad + S(n-1,m) - S(n-2,m) + S(n-1,m) \\
&\quad - [S(n-2,m) + S(n-2,m)] \\
&= \rho(n,m) + 3 \cdot S(n-1,m) - 2 \cdot S(n-2,m).
\end{aligned}$$

□

3.3 Counting stretched subtrees

Counting stretched subtrees has the benefit of fixed beginning and ending layers. Because of this a recurrence relation is more forthcoming.

Theorem 3.3.1. *We have the recurrence relation,*

- $\rho(n,m) = 0$, *if* $n < m$,
- $\rho(n,m) = 1$, *if* $n = m$,
- $\rho(n,m) = \sum_{i=1}^{n-m+1} \left(2^{i-1}\right)^{2^{m-1}} \rho(n-i, m-1)$, *otherwise.*

Proof. Again, the base cases are apparent so we assume $n > m$. Let $T = 2^{<n}$ and $S \subset T$ be a stretched subtree of T with height m.

If $m = 1$, S being a stretched subtree of height one implies that T is also of height 1, so this degenerative case is covered in the base cases.

Suppose then $m > 1$. There is only one choice for the root of S. Let $k < n$ be the layer in T from which S takes its penultimate level of vertices. Then S restricted to its first $m-1$ levels is a stretched subtree of T restricted to its first k levels. This gives us $\rho(k, m-1)$ possibilities for the first $m-1$ levels of S.

There are 2^{m-2} vertices in the $m-1$ level of S. For each σ in that level, there are leaves in S which are extensions of $\sigma^\frown 0$ and $\sigma^\frown 1$. There are 2^{m-1} leaves in S, each of which needs a selection from T.

Each $\sigma^\frown i$ lies in the $k+1$ level of T, so there are $2^{n-(k+1)}$ possible extensions to choose from. Hence for fixed k, there are $\left(2^{n-k-1}\right)^{2^{m-1}}$ possible choices for the leaves of S. We need only sum over all possible k.

By our assumption of $k < n$, the maximum value of k would be $n-1$. Since k corresponds to the $m-1$ level of S, it can be no less than $m-1$. Therefore,

$$\rho(n,m) = \sum_{k=m-1}^{n-1} \left(2^{n-k-1}\right)^{2^{m-1}} \rho(k, m-1).$$

To index from 1, we reverse and shift the indices by letting $k = n - i$. Where k takes on values between $m-1$ to $n-1$, substituting $n-i$ for k shows that i varies from 1 to $n-m+1$. Hence,

$$\rho(n,m) = \sum_{i=1}^{n-m+1} \left(2^{i-1}\right)^{2^{m-1}} \rho(n-i, m-1)$$

as claimed.

□

This can be further simplified by the use of the recursive function þ mentioned in Theorem 3.0.4.

Definition 3.3.2. *Define* $þ_b^a$ *by:*

- $þ_0^a = 1$,
- $þ_b^a = 0$, *for* $b < 0$,
- $þ_b^a = 0$, *for* $a \leq 0$ *and* $b > 0$,
- $þ_b^a = \sum_{i=1}^{a} 2^{2^i} \cdot þ_{b-1}^i$, *for* $a > 0$ *and* $b > 0$.

Theorem 3.3.3. *For all* $n, m > 0$,

$$\rho(n,m) = þ_{n-m}^{m-1} = \sum_{i=1}^{m-1} 2^{2^i} \cdot þ_{n-m-1}^i.$$

Proof. We induct over m and suppose $m = 1$. If $n = 1$, by definition

$$þ_{1-1}^{1-1} = þ_0^0 = 1 = \rho(1,1).$$

If $n > 1$,

$$þ_{n-1}^0 = 0 = \rho(n,1)$$

since there can not be a stretched subtree of height one over any tree with height greater than one.

Consider then $m = k$ and assume

$$\forall n, \rho(n, k-1) = þ^{(k-1)-1}_{n-(k-1)} = þ^{k-2}_{n-k+1}.$$

If $n < k$ then $n - k < 0$, so

$$þ^{k-1}_{n-k} = 0 = \rho(n, k).$$

We induct over $n - k$, with our base case being $n - k = 0$. This gives

$$þ^{k-1}_{n-k} = þ^{k-1}_{0} = 1 = \rho(n, k) = \rho(k, k).$$

Assume then that $n = c > k$ and $\rho(c-1, k) = þ^{k-1}_{c-1-k}$. By Theorem 3.3.1,

$$\begin{aligned}
\rho(c, k) &= \sum_{i=1}^{c-k+1} \left(2^{i-1}\right)^{2^{k-1}} \rho(c-i, k-1) \\
&= \rho(c-1, k-1) + \sum_{i=1}^{c-k} \left(2^{i}\right)^{2^{k-1}} \rho(c-i-1, k-1) \\
&= \rho(c-1, k-1) + 2^{2^{k-1}} \sum_{i=1}^{c-k} \left(2^{i-1}\right)^{2^{k-1}} \rho(c-i-1, k-1) \\
&= \rho(c-1, k-1) + 2^{2^{k-1}} \rho(c-1, k).
\end{aligned}$$

Applying both induction hypotheses and the definition of þ, we get

$$\begin{aligned}
\rho(c-1, k-1) + 2^{2^{k-1}} \rho(c-1, k) &= þ^{k-2}_{c-k} + 2^{2^{k-1}} þ^{k-1}_{c-k-1} \\
&= \left[\sum_{i=1}^{k-2} 2^{2^{i}} þ^{i}_{c-k-1}\right] + 2^{2^{k-1}} þ^{k-1}_{c-k-1} \\
&= \sum_{i=1}^{k-1} 2^{2^{i}} þ^{i}_{c-k-1} \\
&= þ^{k-1}_{c-k}.
\end{aligned}$$

□

The representation in Theorem 3.3.3 has a number of benefits over that in Theorem 3.3.1. The resulting sum has a bound that is now only dependent upon m, the exponential expression depends only on i, and the superscript of þ is simply i. Perhaps most usefully, the subscript of þ is not dependent upon i, so for diagonals we have

$$\rho(m+k,m) = \sum_{i=1}^{m-1} 2^{2^i} \cdot þ_{k-1}^{i}.$$

This fixes the recursion depth of the diagonals, allowing full expansion and elimination of þ for fixed k.

3.4 Putting it all together

We can use the preferred properties of þ to derive a new expression for $S(n,m)$.

Theorem 3.4.1.

$$S(n,m) = \sum_{i=1}^{n} \left(2^{n+1-i} - 1\right) þ_{i-m}^{m-1} = \sum_{i=1}^{n} \left(2^{n+1-i} - 1\right) \rho(i,m)$$

Proof. If $n < m$ we know $S(n,m) = 0$. In the above expression, $i \leq n < m$ implies $i - m < 0$, so by $þ_{i-m}^{m-1} = 0$ by definition, making each term of the sum 0.

Suppose then that $n \geq m$. Let $k = n - m$ and substitute $m + k$ for n in the expression. Rewritten, our claim is then:

$$S(m+k,m) = \sum_{i=1}^{m+k} \left(2^{m+k+1-i} - 1\right) þ_{i-m}^{m-1}.$$

We use strong induction on k with base case $k = 0$. We have then

$$\begin{aligned}\sum_{i=1}^{m} \left(2^{m+1-i} - 1\right) þ_{i-m}^{m-1} &= þ_0^{m-1} + \sum_{i=1}^{m-1} \left(2^{m+1-i} - 1\right) þ_{i-m}^{m-1} \\ &= 1 + 0 = 1 = f(m,m)\end{aligned}$$

since $i - m < 0$ for $i \leq m - 1$.

We assume now that the equation holds for any $k \leq j$ and for any m. Extracting the final term of the sum and manipulating the remaining terms we obtain,

$$\begin{aligned}
&\textstyle\sum_{i=1}^{m+j+1} \left(2^{m+j+2-i} - 1\right) \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + \textstyle\sum_{i=1}^{m+j} \left(2^{m+j+2-i} - 1\right) \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + \textstyle\sum_{i=1}^{m+j} \left[\left(2^{m+j+2-i} - 2\right) + 1\right] \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + \textstyle\sum_{i=1}^{m+j} \left[2\left(2^{m+j+1-i} - 1\right) + 1\right] \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + \textstyle\sum_{i=1}^{m+j} \left[3\left(2^{m+j+1-i} - 1\right) - \left(2^{m+j+1-i} - 1\right) + 1\right] \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + \textstyle\sum_{i=1}^{m+j} \left[3\left(2^{m+j+1-i} - 1\right) - \left(2^{m+j+1-i} - 2\right)\right] \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + \textstyle\sum_{i=1}^{m+j} \left[3\left(2^{m+j+1-i} - 1\right) - 2\left(2^{m+j-i} - 1\right)\right] \text{þ}_{i-m}^{m-1} \\
&\qquad = \text{þ}_{j+1}^{m-1} + 3\left[\textstyle\sum_{i=1}^{m+j} \left(2^{m+j+1-i} - 1\right) \text{þ}_{i-m}^{m-1}\right] \\
&\qquad\qquad -2\left[\textstyle\sum_{i=1}^{m+j} \left(2^{m+j-i} - 1\right) \text{þ}_{i-m}^{m-1}\right]
\end{aligned}$$

By Theorem 3.3.3 $\rho(m+j+1, m) = \text{þ}_{j+1}^{m-1}$. Applying our induction hypothesis to $S(m+j, m)$ we have

$$S(m+j, m) = \sum_{i=1}^{m+j} \left(2^{m+j+1-i} - 1\right) \text{þ}_{i-m}^{m-1}.$$

Applying our induction hypothesis now to $f(m+j-1, m)$ we have

$$\begin{aligned}
S(m+j-1, m) &= \sum_{i=1}^{m+j-1} \left(2^{m+j-i} - 1\right) \text{þ}_{i-m}^{m-1} \\
&= \left(2^{m+j-(m+j)} - 1\right) \text{þ}_{j}^{m-1} + \sum_{i=1}^{m+j-1} \left(2^{m+j-i} - 1\right) \text{þ}_{i-m}^{m-1} \\
&= \sum_{i=1}^{m+j} \left(2^{m+j-i} - 1\right) \text{þ}_{i-m}^{m-1}.
\end{aligned}$$

Making these three substitutions and applying Theorem 3.2.3,

$$\begin{aligned}\sum_{i=1}^{m+j+1}\left(2^{m+j+2-i}-1\right)\mathfrak{b}_{i-m}^{m-1} &= \rho(m+j+1,m)+3\cdot S(m+j,m)\\ &\quad -2\cdot S(m+j-1,m)\\ &= S(m+j+1,m).\end{aligned}$$

With our induction complete, we need only apply Theorem 3.3.3 to show that

$$S(n,m)=\sum_{i=1}^{n}\left(2^{n+1-i}-1\right)\rho(i,m).$$

□

As a corollary to this we have Theorem 3.0.4.

CHAPTER 4

ORDINAL VALUED LARGENESS FOR BINARY TREES

Recall from the background information (Section 2.1) that ordinal-valued largeness notions such as those in the work of Ketonen and Solovay [23] were defined on sets of natural numbers. These were defined in such a way that the numerical values of the elements of a set played an important part in establishing largeness. Because trees are collections of vertices, not numbers, this kind of largeness does not immediately apply.

This shortcoming can be overcome simply by applying numeric labels to the vertices of trees. In computability theory and second order arithmetic it is common to encode trees as sets of natural numbers, but it seems a poor design choice to define largeness for trees in terms of some fixed but arbitrary encoding method. We will instead only assume that the method of labeling is in some sense monotone, using the standard encoding of the infinite binary tree $2^{<\omega}$ as reference.

Recall the definition of largeness provided in Section 2.1.3.

Definition 4.0.1. *Using the quasi-lexical ordering over $2^{<\omega}$ defined as $\sigma < \tau$ when:*

- $|\sigma| < |\tau|$, *or*
- $|\sigma| = |\tau|$ *and* $(\sigma \wedge \tau)^\frown 0 \preceq \sigma$ *and* $(\sigma \wedge \tau)^\frown 1 \preceq \tau$,

we say the pair $\langle T, f\rangle$ is an α-labeling if:

- *T is an α-large set (so necessarily finite), and*
- *$f : T \to 2^{<n}$ is order-preserving bijection for some n.*

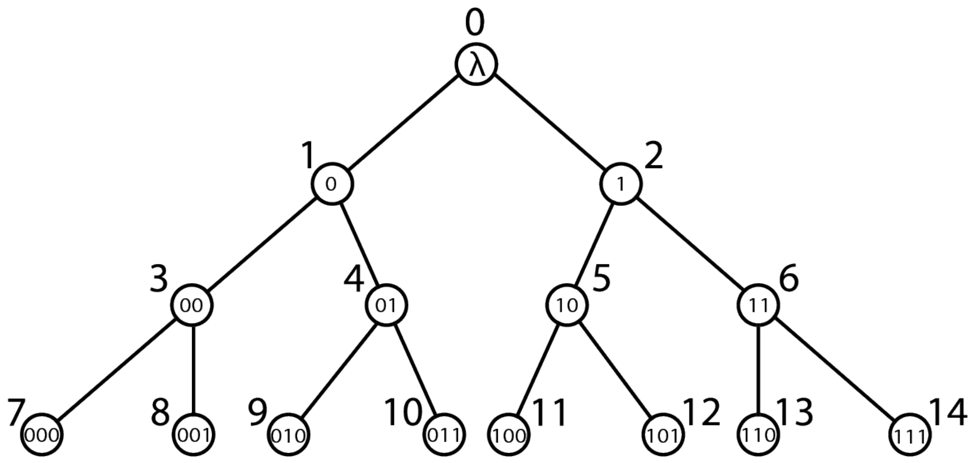

Figure 4.1. Linearly ordering the binary tree $2^{<\omega}$

Consider the embeddings of $2^{<m}$ into $2^{<n}$. Under this definition a labeling of $2^{<n}$ induces distinct labelings on each embedding. On the other hand, we can say that a tree is α-large if there exists an α-large labeling. Technically speaking, interpreting the combinatorial results in this chapter for trees is actually a special case of the more general results for labelings.

Even with these labelings standard largeness arguments fail because they produce an arbitrary subset rather than a subtree. This chapter will consider methods to perform largeness arguments on trees in ways that preserve the required structure. This provides the opportunity to apply new techniques to establish conservation and proof speed up results for tree theorems.

One potential target would be a weak version of Milliken's tree theorem restricted to binary trees, the statement of which follows. We call this weak Milliken's tree theorem for 2-colorings on strong subtrees of height 2, or simply WM_2^2 to follow a common notational format.

Statement 4.0.2. *(WM_2^2) For all 2-colorings on $\mathcal{S}_2(2^{<\omega})$ there exists an infinite strong subtree which is monochromatic.*

To apply the methods used by Patey and Yokoyama [40] to establish a conservation result for Ramsey's theorem for pairs, and its strengthening to a proof complexity result by Kołodziejczyk, Wong, and Yokoyama [25] we must establish an ordinal-valued largeness result. Abusing our notation by referring to trees and labelings interchangeably, this would take the form of a statement saying that for any coloring of strong subtrees of height 2 in some α-large binary tree, there exists a monochromatic strong subtree which is β-large.

Unfortunately the work in this chapter does not get us to that point. The main theorem will instead be an analogous statement for WM^1_2. This is both a weakening of the Halpern-Laüchli theorem for two colorings – restricting it to a single binary tree rather than a computably branching forest – and a strengthening of TT^1_2. The latter is known to be provable in RCA_0 [14] and the former is known to be computably true [1], so if the following produces a conservation result, it isn't a very interesting one.

Theorem 4.0.3. *Let T be an ω^{3n+9}-labeling with $\min T \geq 2$. For any 2-coloring of the vertices of T there exists an ω^n-large monochromatic strong subtree.*

This combinatorial theorem will serve as a practical example of how to adapt and apply largeness arguments to trees.

4.1 Large trees

From any sufficiently large arbitrary set, it is possible to extract a large labeling.

Lemma 4.1.1. *If X is ω^{n+1}-large with $\min X \geq 2$, there exists an ω^n-labeling $T \subseteq X$.*

Proof. For the case of $n = 0$ we note that ω^0-largeness is 1-largeness. Hence we only need to label the root vertex.

With $n > 0$ and $x \geq 2$, we have $X = \{x\} \sqcup Y_1 \sqcup Y_2$ with each Y_i being ω^n-large and $|Y_2| > |Y_1|$. We note $T = \{x\} \cup Y_1$ is ω^n-large by containing Y_1. If T labels a binary tree we are done.

If T does not label a binary tree, then there is a level k that is incompletely labeled. The completely labeled layers of the tree have $2^k - 1$ vertices, so $|Y_1| \geq 2^k - 1$. The layer that is incompletely labeled has 2^k vertices. Hence we need strictly fewer than 2^k additional labels. Because $|Y_2| > |Y_1|$, we can extend T with labels taken from Y_2 to complete the labeling. $\square$

The structure of a strong subtree relies heavily on levels. Given a subset of levels in a tree a strong subtree can always be constructed from those levels. By representing each level with a label we can then apply existing largeness results without breaking needed structure.

Definition 4.1.2. *Given a labeling T, let L_T be the collection of the largest vertex in each level of the tree. We call L_T the level set of T.*

It's easy to see that L_T dominates any initial segment of T, as well as any branch of T. Further, given $X \subseteq L_T$, X will dominate an initial segment of any subtree constructed from the levels corresponding to the labels in X. This allows us to retain both structure and largeness.

The approach to proving Theorem 4.0.3 will take the following form. In the coming sections we will define what *slices* and *stacks* are.

Remark 4.1.3. *Given a large stack of large slices of a tree, either each slice provides a suitable extension for constructing a large solution, or if one slice fails to do so it must itself contain a large solution.*

4.1.1 Tree slices

The argument in Remark 4.1.3 requires two tiers of large structure to be defined. We define slices first.

Definition 4.1.4. *Let T be a labeling. We call $X \subseteq T$ a slice of T if it is a collection of consecutive vertices in T. That is, $X = \{\sigma \in T \mid \tau_1 \leq \sigma \leq \tau_2\}$ for some $\tau_1, \tau_2 \in T$.*

If a slice is α-large, it may be called an α-slice. We denote the portion of T's level set contained in the slice X as $L_X = L_T \cap X$.

Lemma 4.1.5. *Let T be a labeling. If X is an ω-large slice of T with $\min X > 0$, then L_X is nonempty.*

Proof. We decompose $X = \{x\} \sqcup Y$. The vertex x is in some level k of the tree. Level k has 2^{k-1} total vertices, so to complete level k and obtain an element of L_T, we need no more than $2^{k-1} - 1$ vertices.

The first $k-1$ levels of the tree consists of $2^{k-1} - 1$ vertices. Since labels are assigned monotonically, the value of the label x must be larger than this. But since X is ω-large, by definition $x \leq |Y|$. Hence $|Y| > 2^{k-1} - 1$, providing enough labels to complete level k.

It follows then that Y contains an element of L_T.

□

Theorem 4.1.6. *Let T be a labeling with $\min T \geq 2$. If X is an ω^{n+2}-large slice of T, then L_X is $[\omega^n \cdot (\min X - 1)]$-large.*

Proof. We proceed by induction. For our base case we show that if X is an ω^3-large slice of T, then L_X is $[\omega \cdot (\min X - 1)]$-large.

First decompose $X = \{x\} \sqcup Y_1 \sqcup \cdots \sqcup Y_x$ with each Y_i being ω^2-large. Further decompose each $Y_i = \{y_i\} \sqcup Z_{(i,1)} \sqcup \cdots \sqcup Z_{(i,y_i)}$ with each $Z_{(i,j)}$ being ω-large. By Lemma 4.1.5 there is some $s_{(i,j)} \in L_{Z_{(i,j)}}$. For each Y_i define the subset $S_i = \{s_{(i,1)}, \ldots, s_{(i,y_i)}\}$.

We observe that $|S_i| = y_i$ and for each $i < x$ we have $s_{(i,y_i)} < y_{i+1}$. In particular, $s_{(i,y_i)} \leq |S_{i+1}| - 1$. From this we can see that $\{s_{(i,y_i)}\} \cup \Big[S_{i+1} \setminus \{s_{(i+1,y_{i+1})}\}\Big]$ is ω-large and well-defined for each $i < x$. Let W_i denote this set for each $i < x$.

Then $W_1 \sqcup \cdots \sqcup W_{x-1}$ is $[\omega \cdot (x-1)]$-large and by construction every element is in L_X.

For the inductive step, assume that the hypothesis holds for n and we will show that it holds for $n+1$. Consider an ω^{n+3}-large slice of T denoted by X.

Decompose $X = \{x\} \sqcup Y_1 \sqcup \cdots \sqcup Y_x$ with each Y_i being ω^{n+2}-large. By our induction hypothesis there is an $[\omega^n \cdot (\min Y_i - 1)]$-large $S_i \subseteq L_{Y_i}$. For each S_i we have the decomposition $Z_{(i,1)} \sqcup \cdots \sqcup Z_{(i,\min Y_i - 1)}$ with each $Z_{(i,j)}$ being ω^n-large.

For $i < x$ define $a_i = \min Z_{(i,\min Y_i-1)}$. Since $Z_{(i,\min Y_i-1)}$ is an ω-large set with positive minimal element, it must contain at least two elements. Hence

$$a_i < \max Z_{(i,\min Y_i-1)} < \min Y_{i+1}.$$

More precisely, $a_i \leq \min Y_{i+1} - 2$. Let

$$W_i = \{a_i\} \cup \left(Z_{(i+1,1)} \sqcup \cdots \sqcup Z_{(i+1,\min Y_{i+1}-2)}\right)$$

for $i < x$. Each of these is ω^{n+1}-large.

It follows then that $W_1 \sqcup \cdots \sqcup W_{\min X - 1}$ is $[\omega^{n+1} \cdot (\min X - 1)]$-large subset of L_X.

□

Corollary 4.1.7. *If T is an ω^{n+2}-labeling and $\min T \geq 2$, then L_T is ω^n-large.*

Proof. T is a slice of itself. Applying Theorem 4.1.6 to T, we see that L_T is $[\omega^n \cdot (\min T - 1)]$-large. By our assumption on the minimal element of T, we have ω^n-largeness.

□

4.1.2 Stacks of slices

The second large structure required for the argument in Remark 4.1.3 will be stacks, which are large collections of slices. We take inspiration for this from the concept of *groupings* used in Patey and Yokoyama [40] and in Kołodziejczyk and

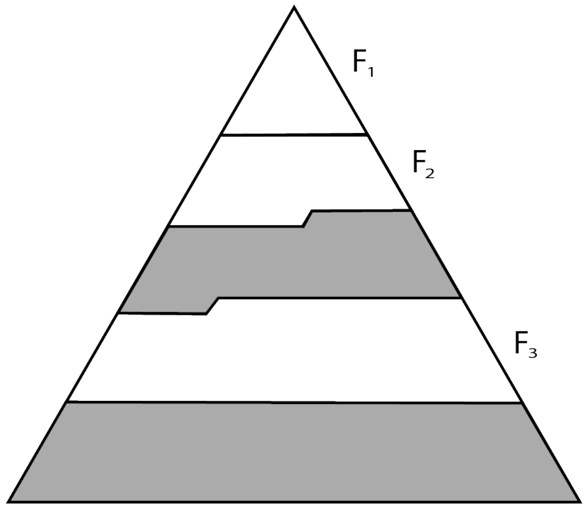

Figure 4.2. Each F_i is a slice. Taken together they form a stack. Note that a slice can start or end in the middle of a level.

Yokoyama [24]. However, unlike groupings, these are not defined in terms of a particular coloring.

Definition 4.1.8. *Let T be an α-labeling. We say $(F_1 < F_2 < \cdots < F_k)$ is an (ω^n, ω^m)-stack of T if $\{\max F_i\}$ is an ω^m-large set and each F_i is an ω^n-slice of T.*

We can see a diagrammatic example of a stack in Figure 4.2. Of course a labeling must have sufficient largeness in order to extract such a stack.

Lemma 4.1.9. *If X is an ω^{n+2m}-slice and $\min X \geq 2$, then it contains an (ω^n, ω^m)-stack.*

Proof. We proceed by induction on m. In the case of $m = 0$, X is ω^n-large and we seek an $(\omega^n, 1)$-stack. But $\{X\}$ is an $(\omega^n, 1)$-stack.

Suppose then that this holds for m and consider the case of $m + 1$. Let X be an ω^{n+2m+2}-slice. We seek to find an (ω^n, ω^{m+1})-stack in X. We have the decomposition $X = \{x\} \sqcup Y_1 \sqcup \cdots \sqcup Y_x$ with each Y_i being an ω^{n+2m+1}-slice. Since $x \geq 2$, we have at least Y_1 and Y_2.

We can decompose $Y_2 = \{y_2\} \sqcup Z_1 \sqcup \cdots \sqcup Z_{y_2}$ with each Z_i being ω^{n+2m}-slice. By our induction hypothesis then each Z_i contains an (ω^n, ω^m)-stack. Let $\mathcal{F} = (F_1 < \cdots < F_k)$ be the union of these stacks. By definition, each F_i is an ω^n-slice and $\{\max F_i\}$ is an $[\omega^m \cdot y_2]$-large set.

Observe that Y_1 is also an ω^n-slice. Further, $\max Y_1 < y_2$. Hence $\{Y_1\} \cup \mathcal{F}$ forms an (ω^n, ω^{m+1})-stack.

□

From this, our theorem follows as a corollary.

Theorem 4.1.10. *If T is an ω^{n+2m}-labeling and $\min T \geq 2$, then it contains an (ω^n, ω^m)-stack.*

Proof. T is an ω^{n+2m}-slice of itself, so we apply the previous lemma.

□

4.2 A tree largeness result

We now have enough to prove the existence of a set of levels large enough to be used to construct solutions for Theorem 4.0.3. We will use a pair of existing largeness results for sets to thin out the level sets in one case.

4.2.1 Some needed lemmas

The material in this section is taken from Kołodziejczyk's and Yokoyama's work on largeness [24]. In particular, one specific result is needed for one case of the proof of Theorem 4.0.3, which in turn requires an additional definition and lemma in order to apply to the tree largeness arguments found above.

Definition 4.2.1. *A set $X = \{x_1, \ldots, x_k\}$ is exp-sparse if $4^{x_i} < x_{i+1}$ for every $i < k$.*

Lemma 4.2.2. *(Kołodziejczyk and Yokoyama sparseness lemma)*

Every $[\omega^{n+3} + 1]$-large set contains an ω^n-large and exp-sparse subset.

Lemma 4.2.3. *(Kołodziejczyk and Yokoyama partition lemma)*

a) Let X be $[\omega^n \cdot 2]$-large and exp-sparse. If $X = Y_1 \sqcup Y_2$, then either Y_1 or Y_2 is ω^n-large.

b) Let X be $[\omega^n \cdot 4k]$-large and exp-sparse. If $X = Y_1 \sqcup Y_2$, then either Y_1 or Y_2 is $[\omega^n \cdot k]$-large.

More specifically, we are interested in the case where X is $[\omega^n \cdot 4^k]$-large and partitioned into k sets $X = Y_1 \sqcup \cdots \sqcup Y_{2^k}$. We can apply Lemma 4.2.3(b) k times to show that at least one partition is ω^n-large.

4.2.2 Applying tree largeness arguments

The proof of Theorem 4.0.3 will involve a density-like argument. We will not explicitly take a density approach, but comparisons can be made to similar arguments such as those found in chapter one of Todorcevic for the Halpern-Laüchli theorem [49], and in the work on Milliken's theorem for forests by d'Auriac, et al [1].

The proof will be split into two cases based on whether a particular property is satisfied. In order to state our conditions more clearly, we will explicitly define a term that will concisely capture the idea of extending a vertex by precisely N levels.

Definition 4.2.4. *Let T be a labeling and fix $\sigma \in T$. We say that $\tau \in T$ is an N-extension of σ if $\sigma \preceq \tau$ and $|\tau| - |\sigma| = N$.*

The bulk of our proof resides in the case where we can not find a monochromatic set of N-extensions for a collection of vertices within a slice. We treat this case separately here. The other case will be handled afterward in the proof of Theorem 4.0.3.

Lemma 4.2.5. *Special case.*

Let S be an ω^{n+7}-slice of a labeling T where $\min T \geq 2$, and M be such that $M < |\min S|$. Fix a 2-coloring of the vertices of T and suppose there exists no N

and color c such that, every element of 2^M has an N-extension of color c in S. There must exist an ω^n-large monochromatic strong subtree in S.

Proof. We first note that by these assumptions $\min S \geq 3$. By Theorem 4.1.6 then $S \cap L_T$ is $[\omega^{n+5} \cdot 3]$-large. We discard the least element of the intersection by defining $L = (S \cap L_T) \setminus \{\min(S \cap L_T)\}$, where each element of L now corresponds to a complete level contained within S. It is apparent that L is $[\omega^{n+5} + 1]$-large with $\min L > 3$.

Applying Lemma 4.2.2 to L, we obtain an ω^{n+2}-large and exp-sparse subset L'. Let L' be enumerated $\{x_1, x_2, y_1, \ldots, y_{k'}\}$.

The elements of T corresponding to 2^M are $\{\sigma_1, \ldots, \sigma_{2^M}\}$. The label $\sigma_{2^M} > 2^{M+1}$, so by our assumptions $x_1 > \min S > 2^{M+1}$. Because L' is exp-sparse we then know that $x_2 > 4^{x_1} \geq 4^{2^{M+1}} > 4^{M+1}$.

Decomposing ω^{n+2}-large L', we have $L' = \{x_1\} \sqcup Y_1 \sqcup \cdots Y_{y_1}$ with each Y_i being ω^{n+1}-large. By the previous paragraph, $Y = Y_1 \setminus \{x_2\}$ is $[\omega^n \cdot 4^{k+1}]$-large.

Let $T \restriction Y = \{\sigma \in T \mid \exists y \in Y,\ |\sigma| = |y|\}$, that is the set of all vertices in every level represented in Y. For each σ_i define

$$T_i = \{\tau \in T \restriction Y \mid \sigma_i \prec \tau\}.$$

Each T_i then contains the descendants of σ_i in the levels represented by Y. Note that $Y = \{y_1, \ldots, y_k\}$ for some k, and define $T_i(j) = \{\sigma \in T_i \mid |\sigma| = |y_j|\}$ to be the jth level of vertices in T_i. See Figure 4.3 for a diagrammatic presentation of this.

For any fixed j, if there is no monotone $T_i(j)$ then we can choose $\tau_1 \in T_1(j)$, $\ldots$, $\tau_{2^M} \in T_{2^M}(j)$, of the same color, violating our hypothesis. If there is only one monotone $T_i(j)$, then we can do the same with each chosen extension having the color of $T_i(j)$. We must then have at least two monotone sets $T_a(j)$ and $T_b(j)$ for each j. We choose either of these. Without loss of generality, we say it is $T_a(j)$. Let $z_j = a$. This gives us a sequence $z_1, \ldots, z_{2^M}$,

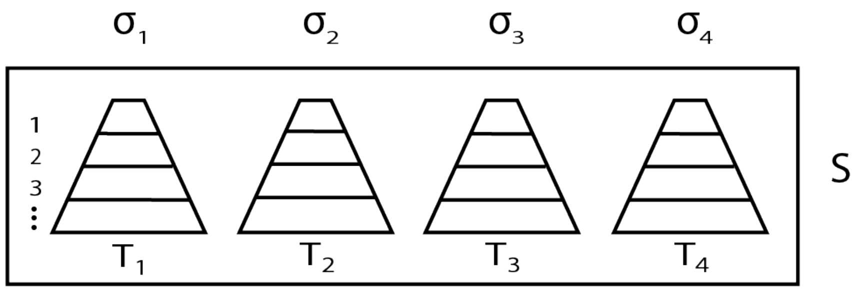

Figure 4.3. The descendants of each vertex σ_i in slice S is the set T_i. Each T_i can be separated by levels, here labeled $1, 2, 3, \ldots$ Each T_i then has constituent levels $T_i(1), T_i(2)$, etc. By our assumptions, for each n, there must be at least two monotone sets $T_a(n)$ and $T_b(n)$.

We can then partition $Y = Z_1 \sqcup \cdots \sqcup Z_{2^M}$ where each $Z_i = \{y_j \in Y \mid z_j = i\}$. By k applications of Lemma 4.2.3 to $[\omega^n \cdot 4^{k+1}]$-large Y we we know that at least one Z_c is $[\omega^n \cdot 4]$-large. By construction, the elements $y_i \in Z_c$ represent monochromatic levels $T_c(i)$. Since we have two colors, one more application of Lemma 4.2.3 gives us an ω^n-large subset $Z' \subseteq Z_c$ where every level represented by a vertex in Z' has the same color.

Let $\mathcal{A} = \{i \leq k \mid y_i \in Z'\}$ and define $T' = \bigcup_{i \in \mathcal{A}} T_c(i)$. The set T' is monochrome and contains every descendant of σ_c found on the levels of T represented by the elements of Z'. It is straightforward to construct a strong tree from T' of height $|Z'|$. Such a tree has an initial segment dominated by Z' and so is ω^n-large. □

With the details of this case aside, the idea behind the rest of the proof can be easily summarized. Taking a labeling, a stack of sufficient dimension can be extracted to perform one of two constructions. If monochromatic extensions occur with enough frequency, then a new level can be added to the construction for each slice in the stack. The largeness of the set of maximal elements in the slices guarantees by domination that the solution is large enough.

If one slice fails to provide the needed extensions, then the lemma above holds. This allows us to extract a large solution from that particular slice.

Proof. (Theorem 4.0.3)

T is $\omega^{(n+7)+2(n+1)}$-large. Applying Theorem 4.1.10 we obtain an $(\omega^{n+7}, \omega^{n+1})$-stack $(S_1, \ldots, S_k)$. If there exists an M and S_i in our stack satisfying Lemma 4.2.5 then we are done.

Supposing otherwise, we can recursively construct a strong subtree T' where each level is a monochrome set taken from its corresponding slice. We begin by taking a root $\lambda \in S_1$. By our assumption there exists an N and c such that each element of $2^{|\lambda|+1}$, including $\lambda^\frown 0$ and $\lambda^\frown 1$, have an N-extension in S_2 of color c. We continue similarly for each subsequent level.

By the definition of stacks, $\{\max S_i\}$ is ω^{n+1}-large. Since T' takes one row from each S_i, the set $\{\max S_i\}$ dominates $L_{T'}$, hence our level set is also ω^{n+1}-large.

We observe that if we separate the levels of T' by color, we have a 2-partition of $L_{T'} = L_0 \sqcup L_1$. By Lemma 4.2.3(a) one of these subsets L_i must be ω^n-large. With L_i it's a simple matter to construct a strong subtree of color i.

□

4.3 Proof theoretic considerations

Because of their intended application, the proofs by Patey and Yokoyama [40] and by Kołodziejczyk and Yokoyama [24] may all be performed in $\mathrm{I}\Sigma^0_1$. The proofs in this chapter can be as well.

Restricting arguments to ordinals below ω^ω guarantees that they can be encoded numerically via Cantor normal form. The induction arguments used in this chapter are of the form "if X is an $\omega^{f(n)}$-large P-structure, then Y is an ω^n-large Q-structure."

Trees and labelings are finite sets, so they pose no challenge. Properties such as largeness, or being a slice, a stack, or a strong subtree are all finitely checked, hence computable. None of these then are impediments to formalizing the arguments in $\mathrm{I}\Sigma^0_1$.

The fact that the methods here fail to produce a suitable proof for WM^2_2 lies in the fact that all attempts required stronger induction and ordinals that did not fall below ω^ω. This does not imply that such a result is out of reach. Future work on this approach could lead to stronger largeness arguments that overcome this issue.

PART II

VERIFICATION AND INDEX SETS

CHAPTER 5

MATRIX WEIGHTED GRAPH SOUNDNESS

As discussed in the background material in Section 2.2, termination analysis is the evaluation of an algorithm to determine whether or not it will halt on all possible inputs. We know this to be an undecidable problem in general. In practice however there has not only been success in applying methods of termination analysis to individual algorithms, but in partially automating certain methods.

Matrix weighted graphs (MWGs) are one tool used in termination analysis [2]. Such graphs are used to represent a recursive algorithm F as an annotated graph structure G. Similar to the method of calling context graphs (CCG) [28] on which it improves, the function computed by F is total if and only if every circuit through the graph exhibits a particular property.

We state this result for MWGs precisely here and will define the terms in the following section. This result has been formally verified on recursive functions definable in a language called PVS0 by Muñoz, Ayala-Rincón, Moscato, Dutle, Narkawicz, Almeida, Avelar, and M Ferreira Ramos [33].

Theorem 5.0.1. *Let F be an algorithm definable in PVS0 and defined by an enumeration of its operations together with the conditions under which each is applied. Further, assume that the operations not involving a recursive call to F be total.*

Then F terminates on all inputs if and only if there exists an MWG G with the following properties:

- *G represents F,*

- *G is edge sound,*
- *there exists a measure configuration for G,*
- *G is measure sound with respect to this measure configuration, and*
- *G exhibits the MWG termination property.*

We will call these five properties the MWG termination conditions.

Determining whether an algorithm terminates on all inputs is precisely the problem of deciding whether an index lies within Tot, so this is not only undecidable, but a Π_2^0 problem. While MWG termination offers necessary and sufficient conditions for termination, the actual determination of when these conditions are met is computationally more difficult.

A measure configuration, as defined in Definition 5.2.3, requires a well founded order. The termination condition requiring one to exist implicitly requires the ability to check well-foundedness. This is famously a non-first-order property, and we can easily see that any decision procedure that determines well-foundedness can immediately serve as an oracle for termination.

Consider the model of computation consisting of Turing machines with total transition relations. Fix some algorithm ϕ_c and let $\phi_{c,t}(n)$ represent the configuration of the Turing machine computing ϕ_c on input n after t steps. We call $\phi_{c,t}(n)$ a partial computation.

Let $P = \{\phi_{c,t}(n)\}$ be the set of all partial computations of ϕ_c. We define a partial ordering on P as $\phi_{c,t}(n) < \phi_{c,s}(m)$ if there exists some k such that $\phi_{c,t}(n) = \phi_{c,s+k}(m)$ – that is, if there is some sequence of computations that leads from the Turing configuration $\phi_{c,s}(m)$ to $\phi_{c,t}(n)$. This relation is r.e. (Π_1^0).

We note that since the transition relation is total, the only time a computation may cease is if the halting state is reached. This leaves two possibilities for nonconvergence – either P contains a cycle or an infinite descending chain. If neither is the case, then for every n, we must have $\phi_c(n)\downarrow$. So $c \in Tot$ if and only if $(P,<)$ is well-founded.

The requirement to have a well founded ordering is itself sufficient to account for the undecidability in determining whether the MWG termination conditions apply. This strength of well-foundedness is not a fresh topic in either logic or termination analysis (see [48] for an example at the intersection of these two fields). Furthermore, if we are interested in automating the process, narrowing the scope of what orderings are under consideration is a good place to start simplifying things. For our purposes in this chapter, Definition 5.3.1 will restrict our measure configurations to those using the standard ordering $(\omega,<)$.

We may then ask if there are other sources of undecidability here. For the remaining termination conditions, we will see that whether G represents F is a decidable problem, and Muñoz, et al,[33] have provided a decision procedure that determines when G exhibits the MWG termination property.

This leaves the conditions regarding edge soundness and measure soundness. We will show that any Π^0_1 set is 1-reducible to both of these problems. More specifically, we will show that for certain restricted classes of MWGs and measure configurations these problems are Π^0_1-complete. The details by which we show this suggests possible further work involving tighter restrictions that might produce a partial decision procedure.

5.1 MWG termination

A matrix-weighted graph (MWG) is a directed graph with its vertices labeled with additional information representing the computational flow of an algorithm. This extra information includes *calling contexts*, which encode the conditions under which a recursion occurs and what information to pass into that recursion, and *measure matrices*, which, along with a finite collection of measures, are an attempt to show that the recursion is well-founded.

We defer formally addressing the relation between MWGs and the algorithms they are intended to represent until the next section. In this section we will define the terms introduced in Theorem 5.0.1. We begin by defining measure matrices.

Definition 5.1.1. *A* measure matrix *is a square matrix valued in* $\{0, 1, \star\}$*, with matrix multiplication induced by the following Cayley tables.*

+	0	1	$\star$
0	0	1	$\star$
1	1	1	$\star$
$\star$	$\star$	$\star$	$\star$

*	0	1	$\star$
0	0	0	0
1	0	1	$\star$
$\star$	0	$\star$	$\star$

The commutativity of both operations is apparent in the symmetry of their Cayley tables. Because we are dealing with only three values, it is relatively easy to verify that addition is associative and that multiplication is distributive over addition. This implies that the multiplication of measure matrices is also associative.

Further, for each n the standard $n \times n$ **1**-matrix acts as multiplicative identity. Hence the $n \times n$ measure matrices form a finite semigroup under matrix multiplication.

Definition 5.1.2. *A* calling context *is a pair* (P, f) *where* $P : \omega^k \to 2$ *is a computable total function we interpret as a truth predicate on the natural numbers, and* $f : \omega^k \to \omega$ *is computable.*

We will denote P *as the condition of the calling context, and* f *is the parameter transformation.*

Here k is the number of parameters involved in the recursion, but with pairing and projection functions we are free to simplify these to $P : \omega \to 2$ and $f : \omega \to \omega$. The total function P is intended to be the condition under which a recursive call occurs. The function f is the operation we perform on inputs to obtain the parameters to be passed into the recursive call. We do not required f to be total because some recursive functions call themselves to produce new parameters. The Ackermann function in Section 5.2 is an example of this.

We associate measure matrices and calling contexts with a directed graph in the following way.

Definition 5.1.3. *Fix n. An matrix weighted graph (MWG) is a directed graph $G = (V_G, E_G)$ such that:*

- *V_G is the collection of vertices in G labeled with calling contexts,*
- *E_G is the collection of directed edges in G where (v_i, v_j) indicates an edge from v_i to v_j (also written $v_i \to v_j$), and*
- *every outgoing edge from a vertex v is labeled with an $n \times n$ measure matrix M_v.*

Here n will correspond with the number of measure functions needed to have a proper measure configuration. These have no bearing on edge soundness, but are needed to establish measure soundness. For convenience we often abbreviate a vertex labeled by (A, a) simply as A, and its associated measure matrix as M_A.

A few terms from graph theory will be needed, which we define here.

Definition 5.1.4. *Fix G.*

- *A* path *in G is a sequence of vertices $\{v_0, \dots, v_k\}$ such that for every $0 \le i < k$, $(v_i, v_{i+1}) \in E_G$.*
- *A* circuit *in G is a path ending where it began with $v_0 = v_k$.*

Measure matrices can be extended to paths by matrix multiplication. Given a path $w = \{v_0, \dots, c_k\}$, the measure matrix for w is $M_w = \prod_{i=0}^{k} M_i$. We are particularly interested in the measure matrices of circuits.

Definition 5.1.5. *A measure matrix is said to be* positive *when it contains a $\star$ on its diagonal.*

This is enough to define the MWG termination property.

Definition 5.1.6. *An MWG is said to exhibit the MWG termination property if the measure matrix of every circuit is positive.*

Although an MWG may have an infinite number of circuits, the finiteness of the measure matrix semigroup allows us to place a bound on the length of circuits that must be checked as shown by Muñoz, et al. [33]. The Dutle procedure described in that paper leverages this bound to provide an effective means to check for the MWG termination property.

5.2 Correct MWGs

Informally, we want an MWG to capture the paths of computation in a recursive algorithm. This method is tailored to algorithms specified in a particular way, the general form of which looks like

$$F(x) = \begin{cases} g_1(x) & P_1(x) \\ \vdots & \vdots \\ g_k(x) & P_k(x) \end{cases}$$

where $\{P_1, \dots, P_k\}$ partitions the domain and the g_i are the functions applied to the input corresponding to each condition.

The flow of computation for $F(x)$ involves determining which P_i is satisfied by x, performing some operation on x, then either returning the result as an output or passing it back to F in a recursive call. We assume that termination has been inductively established – that is, if g_i does not call F, then it terminates on all inputs satisfying P_i.

As an example consider the Ackermann function, defined as

$$A(n,m) = \begin{cases} m+1 & n=0 \\ A(n-1,1) & n>0 \wedge m=0 \\ A(n-1,A(n,m-1)) & n>0 \wedge m>0. \end{cases}$$

Here the first case has the condition $n = 0$ and there is no recursion. Since there is no recursion in this case it is disregarded for the MWG termination analysis. The second case has the condition $n > 0 \wedge m = 0$ and computes its output in terms of the Ackermann function itself, $A(n-1,1)$. To represent this the MWG must have a vertex labeled with the calling context

$$(A,a) = (\lambda(n,m).[n>0 \wedge m=0], \lambda(n,m).[n-1,1]).$$

The final case has the condition $n > 0 \wedge m > 0$. Here we have nested recursions that must be treated separately. Once we determine that the condition holds for n, m, we must calculate $x = A(n, m-1)$, then use that to calculate $A(n-1, x)$. The MWG must have two vertices labeled with the calling contexts

$$(B,b) = (\lambda(n,m).[n>0 \wedge m>0], \lambda(n,m).[n-1,A(n,m-1)]), \text{ and}$$

$$(C,c) = (\lambda(n,m).[n>0 \wedge m>0], \lambda(n,m).[n,m-1]).$$

If we use the above definition of the Ackermann function to compute something like $A(2,3)$, these three vertices represent intermediate steps. In these intermediate steps we would check to see which condition held for the current parameters, then proceed down one or more paths to continue the computation. If at any point the parameters fail to satisfy the conditions of any of the vertices, then that line of computation will have come to an end at a non-recursive case.

Definition 5.2.1. *Let G be an MWG and F be an algorithm defined in terms of conditions and corresponding operations, which include k recursion calls. We say that $(P_1, f_1), \ldots, (P_k, f_k)$ are the calling contexts of F if the ith recursive call has the form $F(f_i(x))$ and occurs when $P_i(x)$ is true.*

Further, we say that G represents F, or G is a graph of F when G has k vertices labeled with the calling contexts of F.

The calling contexts of an algorithm F are all the information needed to perform an MWG termination analysis on F. The directed edges connecting the vertices of G are intended to represent possible transitions between steps of computation. MWG termination analysis does not require every edge to be in use, only that they are consistent with the algorithm definition. We call this property *edge soundness* and define it here.

Definition 5.2.2. *Let G be an MWG. We say that G has* edge soundness *if, for every $(A, a), (B, b) \in V_G$, when there exists an $i \in \omega$ such that $A(i)$ and $B(a(i))$, the edge $A \to B$ is in E_G.*

In other words, if a transition is possible, it must be represented as an edge. Any fully connected MWG then is trivially edge sound since every possible transition is represented. Figure 5.1 illustrates such a fully connected MWG for the Ackermann function, without showing the measure matrices.

The ability to leave extra edges in the MWG proves to be useful. In the Ackermann example the edge soundness problems for $B \to B$ and $B \to C$ say that the edges must exist when

$$\exists x, y, x > 0 \land y > 0 \land x - 1 > 0 \land A(x, y - 1) > 0$$

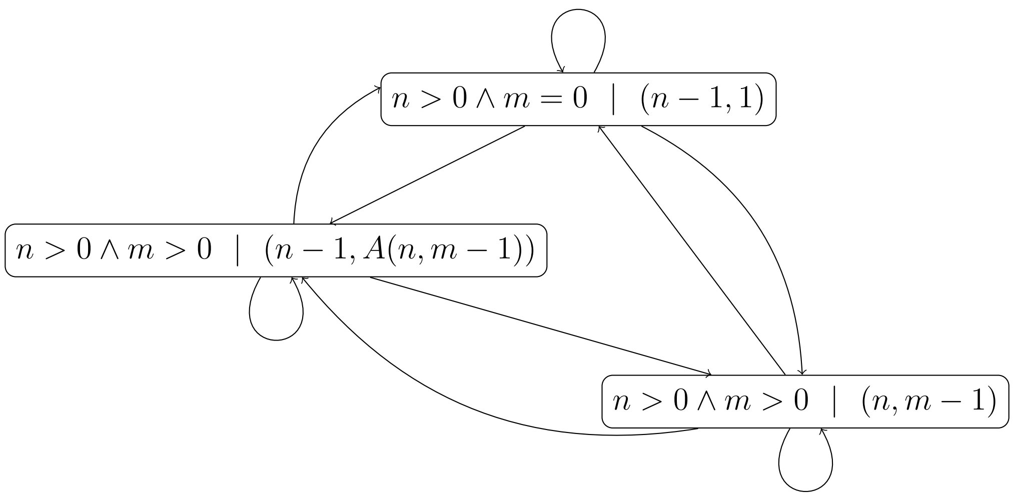
$n > 0 \wedge m = 0 \;\mid\; (n-1, 1)$
$n > 0 \wedge m > 0 \;\mid\; (n-1, A(n, m-1))$
$n > 0 \wedge m > 0 \;\mid\; (n, m-1)$

Figure 5.1. A fully-connected MWG, sans measure matrices, for the Ackermann function. This is trivially edge sound.

Since the terminal step of the Ackermann algorithm is the application of the successor function, we know that $A(n, m) > 0$ for any inputs. It's easy to see in this case then that an MWG representing A must have both of these edges to remain edge sound. In more ambiguous cases we may simply add in an extra edge for transitions where edge soundness can not be determined.

It's easy to imagine algorithms where MWG termination analysis could lead to some circularity since its possible for edge soundness to depend upon the termination properties of the algorithm itself when a recursion occurs in the parameter transformation. We would argue that this is more of a programming problem than a problem for logic, however, since such circumstances can be avoided.

Indeed, we may even limit the conditions and parameter transformations in our calling contexts to be primitive recursive. This does not restrict the class of functions we can consider. Let f be a computable function with an index i. Then f can be computed by defining

$$F_i(n,t) = \begin{cases} \phi_i(n) & \phi_{i,t}(n)\downarrow \\ F_i(n,t+1) & \phi_{i,t}(n)\uparrow \end{cases}$$

because for every choice of t, $f(n) = F_i(n,t)$.

Here the condition $\phi_{i,t}(n)\downarrow$ is a terminal case that does not include a recursive call to F_i. The condition itself ensures that $\phi_i(n)$ terminates for any satisfying n. The single recursion corresponds to the calling context

$$(\lambda(n,t).[\phi_{i,t}\uparrow], \lambda(n,t).[n,t+1]).$$

Since this is essentially an application of Kleene normal form, both components of the calling context are primitive recursive. We may say the calling context itself is primitive recursive.

An edge sound MWG that represents an algorithm F captures all possible recursions, so proving termination is a matter of showing that no infinite path through the MWG corresponds to an actual computation. This is done by demonstrating that such a computation would induce an infinitely descending chain in a well-founded ordering. In [33] the authors do this by way of measure functions and measure matrices.

Definition 5.2.3. *Let G be an MWG with $n \times n$ measure matrices. Let $(S,<)$ be a well-founded ordering, and $\{\mu_1,\dots,\mu_n\}$ be a collection of functions $\omega \to S$ called* measure functions*. We call such a collection of measure functions along with their associated ordering a* measure configuration*.*

We say that G has measure soundness *relative to a measure configuration when for every $(A,a) \in V$ the following hold for each $i,j \leq k$:*

- *$M_A(i,j) = \star$ implies that for all $v \in \omega$ such that $A(v)$, we have $\mu_i(v) > \mu_j(a(v))$, and*
- *$M_A(i,j) = 1$ implies that for all $v \in \omega$ such that $A(v)$, we have $\mu_i(v) \geq \mu_j(a(v))$.*

We note that any MWG with 0-matrices as its measure matrices will always be sound since a 0 entry in the matrix doesn't imply any information about the relation of a parameter to its transformed value.

We note here that there is no question whether or not a uniform process exists to generate MWGs which are edge and measure sound. The existence of constructions which are trivially sound makes this simple. Unfortunately these MWGs will not typically exhibit the MWG termination property.

These trivially sound graphs do not contain enough information to establish termination. For most algorithms it will be the edges missing from the graph and more complicated measure matrices which allow us to prove totality. It is in determining when an edge can be removed and when a nontrivial matrix is sound that undecidability becomes an issue.

5.3 MWG soundness properties

The analysis of edge soundness began by adapting the method by which Cholak embedded Post's correspondence problem into Prolog [10]. This work can be found in Appendix B. Once it became clear how this embedding forced the edge soundness problem to be undecidable, a more general approach presented itself. This is what will be covered in this section.

As discussed in the previous section, any computable function can be expressed by an algorithm using only primitive recursive conditions and parameter transformations. Such a restriction then seems a good place to start since we theoretically lose nothing. Furthermore, an algorithm utilizing components which are not primitive recursive would not be very practical due to the resources and time it would require to execute any but the smallest of inputs.

The primitive recursive functions have the added benefit of being recursively enumerable.

Definition 5.3.1. *Fix some decidable language for expressing primitive recursive functions (such as Gödel's primitive recursive notation). We will say that a function is in primitive recursive form when it is expressed in this language. Similarly, we will say that a calling context is in primitive recursive form when its condition and parameter transformation are in primitive recursive form.*

Let $\mathcal{E}$ be the collection of all MWGs with calling contexts in primitive recursive form. We will call these the primitive recursive MWGs.

Recall in Definition 5.2.3 measure configurations required a well-founded ordering. Here we will fix an ordering. Let $\mathcal{W}$ be the collection of all compatible (that is, the number of measure functions matches the dimension of the measure matrices) combinations of primitive recursive MWGs and all measure configurations consisting of the standard ordering $(\omega, <)$ along with measure functions in primitive recursive form. We will call these the primitive recursive MWG configurations.

Since the properties of being an MWG and being in primitive recursive form are computable, $\mathcal{E}$ and $\mathcal{W}$ are computable. We also note the following property of Π^0_1 sets and their counterparts, the Σ^0_1 sets.

Theorem 5.3.2 (Uniformization Theorem)**.** *Any Π^0_1 (Σ^0_1) set can be written in the form $\{i \mid \forall x, \psi(i,x)\}$ (respectively, $\{i \mid \exists x, \psi(i,x)\}$), where ψ is primitive recursive.*

This allows us to define for any Π^0_1 set the following family of functions which have MWGs in $\mathcal{E}$ and MWG configurations in $\mathcal{W}$.

Definition 5.3.3. *Let $X = \{i \mid \forall x, \psi(i,x)\}$ be Π^0_1. As noted above this can be done* *in such a way that ψ is primitive recursive. Define a family of functions T_n such that*

$$T_n(x) = \begin{cases} T_n(x) & \neg\psi(n,x) \\ 0 & o.w. \end{cases}$$

For fixed n, x, we observe that $T_n(x) = 0$ when $\psi(n,x)$, otherwise it loops. Hence $n \in X$ if and only if $\forall x, \psi(n,x)$ if and only if T_n computes a total function, specifically, the constant 0 function. We will refer to the collection of algorithms $\{T_n\}$ as the tests for X.

The only calling context required for T_n is

$$(\lambda(x).[\neg\psi(n,x)], \lambda(x).[x])$$

which is primitive recursive.

Lemma 5.3.4. *Any Π^0_1 set has a 1-reduction to the subset $\mathcal{E}'$ of $\mathcal{E}$ consisting of the edge sound primitive recursive MWGs.*

Proof. Fix some Π^0_1 set $X = \{i \mid \forall x, \psi(i,x)\}$.

For every i define G_i to be the MWG consisting of a single vertex labeled with the calling context

$$A = (\lambda(x).[\neg\psi(i,x)], \lambda(x).[x])$$

with no edges.

We can see that G_i represents T_i and that it is trivially measure sound. This calling context is primitive recursive, so we may assume $G_i \in \mathcal{E}$.

For distinct numbers n, m we may also assume $\psi(n,x)$ and $\psi(m,x)$ are syntactically distinct. If they were not, we could replace ψ with $\psi' \equiv \psi(n,x) \wedge n = n$ to achieve distinctness. Therefore the construction of G_i is a 1-1 mapping of ω into $\mathcal{E}$.

G_i is edge sound precisely when there exists no x such that $\neg\psi(i,x)$. Hence G_i is edge sounds if and only if $\forall x, \psi(i,x)$.

Therefore $i \in X$ if and only if $G_i \in \mathcal{E}'$.

□

Lemma 5.3.5. *Any Π^0_1 set has a 1-reduction to the subset $\mathcal{W}'$ of $\mathcal{W}$ consisting of the measure sound primitive recursive MWG configurations.*

Proof. Proceed as in the previous proof, but include the edge $A \to A$ labeled with the 1×1 measure matrix $[\star]$ in the construction of G_i. To complete the configuration we use the standard ordering $(\omega, <)$ and the measure function $\mu(x) = x$. Call this configuration C_i and note that it is in $\mathcal{W}$.

We can see that $\mu(x) > \mu(x)$ is never true, so this configuration can only be measure sound if for every x we have $\psi(i,x)$. Therefore $i \in X$ if and only if $C_i \in \mathcal{W}'$.

□

Because $\mathcal{E}$ and $\mathcal{W}$ are computable, and the components of the MWGs and measure configurations in these sets are total (indeed, primitive recursive), we can say more.

Theorem 5.3.6. *$\mathcal{E}'$ and $\mathcal{W}'$ are each Π^0_1-complete. That is, the restrictions of the edge soundness and measure soundness problems considered in this section are both 1-equivalent to the complement of the halting problem, and Turing equivalent to the halting problem itself.*

Proof. Given Lemmas 5.3.4 and 5.3.5, we need only demonstrate that $\mathcal{E}'$ and $\mathcal{W}'$ are Π^0_1-definable. By Post's theorem it is sufficient to show that these two sets are co-r.e.

We will start with showing that the complement of $\mathcal{E}'$ is r.e. Let G be an MWG. We can computably test whether or not $G \in \mathcal{E}$. If not, then $G \in \overline{\mathcal{E}'}$. If it is, then we continue, noting that A, B, a are all total functions. G has a finite number of possible edges, so we can identify which ones are missing. For each missing edge

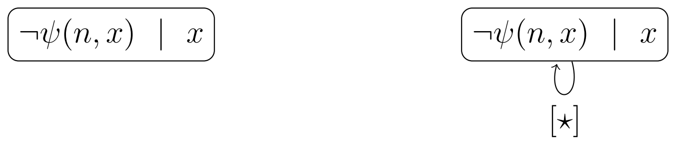

Figure 5.2. The MWGs used for Lemma 5.3.4 (left) and Lemma 5.3.5 (right).

$(A,a) \to (B,b)$, we search for an x such that $A(x)$ and $B(a(x))$. If we find one, then this indicates that G can not be edge sound, so $G \in \overline{\mathcal{E}'}$.

This process is effective and fails to terminate precisely when $G \in \mathcal{E}'$.

Next we show that the complement of $\mathcal{W}'$ is r.e. Let G be an MWG with compatible measure configuration (meaning we have exactly enough measure functions in the configuration to match the dimensions of the measure matrices in G). If G and it's measure configuration are not in $\mathcal{W}$ then we know it is in $\overline{\mathcal{W}'}$. If it is, then we have a finite number of matrices M in G. For each one we need to check a finite number of combinations of indices i,j. If $M(i,j) = \star$, we search for an x such that $\mu_i(x) \leq mu_j(a(x))$. If $M(i,j) = 1$, we search for an x such that $\mu_i(x) < \mu_j(a(x))$. If we find such an x, then G can not be measure sound with respect to this measure configuration. This means this combination fo G and measure configuration must be in $\overline{\mathcal{W}'}$.

Since G and the measure configuration are in $\mathcal{W}$, we know μ_i, μ_j, a are all total. Hence the process is effective and fails to terminate precisely when G and the measure configuration are in $\mathcal{W}'$. $\square$

Although this method of restricting MWG termination analysis remains undecidable, it is a notable simplification in complexity.

5.4 Conclusion

In this chapter we considered the application of MWG termination analysis restricted to recursive functions that only used testing conditions and calculated parameter transformations expressed in primitive recursive form. The results show that this is a Π^0_1 problem as opposed to termination analysis in general being at least Π^0_2.

This particular restriction does not limit the family of functions that can be represented algorithmically, only the form in which they may be expressed. Since algorithms in practice are likely to only use sub-exponential computations, further restrictions might be explored. The method used here to show the Π^0_1 completeness of both edge soundness and measure soundness will fail without the full use of primitive recursiveness since we would no longer be able to embed every Π^0_1 set.

In Figure 5.2 we can see that the MWGs used in Lemma 5.3.4 and Lemma 5.3.5 are very small. This indicates that we can not rely alone on the structure of MWGs to develop a partial decision procedure – some syntactic restrictions on the algorithms is also needed.

CHAPTER 6

MINIMAL INDEX SETS AND MAXIMAL R.E. SUBSETS

The motivation of the previous chapter involved the desire to automate termination analysis (see Section 2.2 for more background). This involves devising a partial decision procedure for totality, which defines an r.e. subset of Tot. It is important to note that such a solution need not be an index set – indeed, such a solution is impossible – it needs only to include a sufficient number of cases to be useful in practice. It's natural to ask what we might mean here by "sufficient."

In general suppose we have some practical problem P we wish to develop a *sound* partial decision procedure for – that is, for any instance x, $S(x)$ implies $P(x)$. We also wish for S to be an algorithmic, so it must define an r.e. set.

We can always define S in such a way that $\{x \mid S(x)\}$ is finite (or even empty), but such a solution is likely to be of limited utility since it would amount to a list of special cases.

On the other hand, having a *complete* solution so that $S(x)$ if and only if $P(x)$ is often either impossible or not at all economical. We need $\{x \mid S(x)\}$ to be, in some sense, *large enough* for the problem at hand.

Remark 6.0.1. *We then informally define the term* "sufficient coverage" *to capture this requirement. We say that a partial solution S provides sufficient coverage for the problem P when $\{x \mid S(x)\}$ is large enough to be useful for our current purposes.*

This is obviously highly dependent upon the application in mind. There are two examples at the extreme that we might call *broad* solutions and *deep* solutions.

Suppose the problem P can be partitioned into classes $\{P_i\}$. A broad solution would identify at least one element in each class so that $P_k \cap \{x \mid S(x)\}$ is never empty. On the other hand, a deep solution would identify at least one entire class – that is, for some k, $P_k \subseteq \{x \mid S(x)\}$.

In the case of Tot, it can be naturally decomposed into disjoint unions of index sets. At the most granular, these index sets have the form $\{i \mid \phi_i = f\}$, so that each partition of Tot corresponds to all of the indices of algorithms computing a specific computable total function.

These "small" index sets are notable enough that they deserve a name.

Definition 6.0.2. *A minimal nontrivial index set, or simply* minimal index set*, is a set $I_f = \{i \mid \phi_i = f\}$ for some computable function f. Where we have an index for the function ϕ_c, we may denote its minimal index set as I_{ϕ_c}, or abbreviate it by I_c when the meaning would be clear. These are minimal in the sense that they do not properly contain any nonempty index set.*

A broad solution for termination analysis in this sense would be an r.e. subset of Tot containing at least one element for every minimal index set in Tot. This possibility is immediately ruled out by one of the foundational proofs in computability theory – a recursive enumeration of computable total functions falls to a diagonal argument.

On the other extreme, a deep solution would be an r.e. subset of Tot containing at least one minimal index set (but not requiring the deep subset to itself be an index set). Rice's theorem states that no non-trivial index set is computable. While there do exist r.e. index sets, these have a very particular form. The following result is a special case of a result found in [44].

Theorem 6.0.3 (Rice-Shapiro)**.** *Let X be an r.e. set. Then X is an index set if and only if there exists an r.e. collection of finite functions (graphs) $\{g_j\}$ such that $X = \bigcup_j \{i \mid \phi_i \supseteq g_j\}$, where $f \supseteq g$ means f is an extension of g.*

Proof. It's easy to see that $\{i \mid \phi_i \supseteq f\}$ for some finite function f is an r.e. index set, and similarly so for the union of an r.e. collection of such index sets.

Going the other direction, suppose that X is an r.e. index set.

We first show that for $i \in X$ and for any c such that $\phi_c \supseteq \phi_i$, then $c \in X$.

Let h be a computable function with domain X. Define

$$g(x,n) = \begin{cases} \phi_i(n) & \exists t, \phi_{i,t}(n) \downarrow \wedge h_t(x) \uparrow \\ \phi_c(n) & \exists t, \phi_{i,t}(n) \uparrow \wedge h_t(x) \downarrow \\ \phi_c(n) & \mu t.[\phi_{i,t}(n) \downarrow] = \mu t.[h_t(x) \downarrow] \\ \uparrow & o.w. \end{cases}$$

By applying the s-m-n and recursion theorems, we have some f and k such that $\phi_k(n) = \phi_{f(k)}(n) = g(k,n)$.

If we suppose $k \notin X$, then $\forall t, h_t(k) \uparrow$. It follows then that $\phi_k = \phi_i$, so $k \in X$ – a contradiction.

If instead $k \in X$, we know $\exists t, h_t(k) \downarrow$. Because ϕ_c extends ϕ_i, we have $\phi_k = \phi_c$ and hence $c \in X$.

We next show that if $i \in X$ then there exists some $k \in X$ such that ϕ_k is a finite function and ϕ_i extends ϕ_k.

Define

$$g(x,n) = \begin{cases} \phi_i(n) & \exists t, \phi_{i,t}(n) \downarrow \wedge h_t(x) \uparrow \\ \uparrow & o.w. \end{cases}$$

Again by s-m-n and recursion theorems we have f and k such that $\phi_k(n) = g(k,n)$.

If $k \notin X$ then $\forall t, h_t(x) \uparrow$ then $\phi_k = \phi_i$, leading to a contradiction.

If $k \in X$ then $\exists t, h_t(x) \downarrow$. Hence for every $n \geq t$, $\phi_k(n) \uparrow$, so ϕ_k is a finite function. For every $n < t$ for which $\phi_i(n) \downarrow$, $\phi_k(n) = \phi_i(n)$, so ϕ_i is an extension of ϕ_k.

Finally, we can enumerate the finite functions and recursively test whether they are in X via h. This gives us an r.e. collection $\{g_j\}$ of every finite function with an index in X. Since we have shown that X is closed under extensions, it follows that $X \supseteq \bigcup_j \{i \mid \phi_i \supseteq g_j\}$.

Because for every $c \in X$, ϕ_c extends some finite function with an index in X, this must be an equality.

□

It follows from this result that any r.e. index set then must contain indices for partial functions. Such a set can not be a subset of Tot. We again emphasize that a deep subset of Tot need not be an index set, only contain one.

While it is not immediately apparent whether there exists an r.e. deep subset of Tot, minimal index sets turn out to be interesting in another way. For any total function f we have $I_f \equiv_1 Tot$. So Tot is 1-equivalent to every component of it's minimal index set decomposition $Tot = \bigcup_{f \in \mathcal{T}} \{i \mid \phi_i = f\}$, where $\mathcal{T}$ is the collection of all computable total functions.

Much more can be shown about the 1-degrees of the minimal index sets. In Section 6.1 the following will be proven.

Theorem 6.0.4. *Let f be a computable function and $K_1 = \{i \mid W_i \neq \emptyset\}$ be the halting set.*

- *If f has an infinite computable domain (including if f is total), then $I_f \equiv_1 Tot$. Hence I_f is Π_2-complete*
- *If f has empty domain, then $I_f = \overline{K_1}$. Since K_1 is Σ_1-complete, I_f is Π_1-complete.*
- *If f has a nonempty domain, then $K_1 \leq_1 I_f$.*
- *If f has a finite domain, then $\overline{K_1} \leq_1 I_f$ as well, so I_f is strictly Δ_2. Furthermore, if g also has finite domain, then $I_f \equiv_1 I_g$.*

As a corollary to this, the Turing degree of all minimal index sets falls within the range $\emptyset'$ to $\emptyset''$.

This is particularly interesting since each minimal index set corresponds to the problem of verifying if a particular algorithm computes a particular function. While this result doesn't completely settle the structure of the 1-degrees for minimal index sets, it does provide tight upper and lower bounds for the degree of complexity of all such verification problems. Specifically it tells us that the problem of verifying which algorithms compute some f requires at least a halting oracle, but no more than a totality oracle.

Returning to the problem of sufficient coverage for Tot, the extremes of breadth and depth do not seem very good characterizations of what we want. The natural desire is that S will cover as many eventualities as possible, which is something neither deep solutions nor broad solutions would provide. We must approach the problem from a different angle.

Imagine we have some existing partial solution S we wish to improve upon. For an undecidable problem, we may assume that $\{x \mid P(x)\} \backslash \{x \mid S(x)\}$ is infinite.

We can extend S either finitely or with an infinite class that is not already included. The former can be done empirically, adding specific cases as needed. This may be repeated an arbitrary number of times without exhausting the potential for improvement. The latter case requires a conceptual improvement by identifying a property – which might correspond to a syntactic feature or a heuristic method – that determines an entire class of cases.

It seems reasonable then that a partial solution might be considered sufficient when no more conceptual improvements can be made and only finite iterative improvements remain. This line of thinking leads us to the following definition.

Definition 6.0.5. *For sets A, B we say $A \equiv_{fin} B$ when the set difference is finite. Let $X \subseteq Y$ be infinite sets with X being r.e. We say that X is maximal in Y when, for every r.e. Z such that $X \subseteq Z \subseteq Y$, either $X \equiv_{fin} Z$ or $Z \equiv_{fin} Y$.*

If Y itself is not r.e., X is maximal when every r.e. extension of X in Y is only *a finite extension.*

In Section 6.2 we will prove the following specific statement, then extend it to a stronger result.

Theorem 6.0.6. *No r.e. set is maximal in Tot.*

If such a maximal set existed it would be unique since the union of two r.e. sets is also r.e. Since no such set exists, we may conclude that no single method of termination analysis offers an optimal partial solution. Indeed, there is no effective method of combining an infinite number of partial solutions that would suffice since an r.e. union of r.e. sets is also r.e.

6.1 Minimal index sets

A central goal in software verification is to prove that a particular program exhibits some specified property. Consider for instance the problem of determining if an algorithm computes the square of the input. By Rice's theorem, this is the undecidable problem of testing membership in the index set $\{i \mid \phi_i(n) = n^2\}$.

In general index sets can be arbitrarily complex. Let X be an arbitrary set and $\phi_{x(i)}(n) = g(i,n)$ where $g(i,n) = 1$ if $n = i$ and is undefined otherwise. Define $A_c = \{i \mid \forall n, (n = c \wedge \phi_i(n)\downarrow) \vee (n \neq c \wedge \phi_i(n)\uparrow)\}$ and $Y = \bigcup_{c \in X} A_c$. We observe that x is a 1-reduction of X into Y,

$$i \in X \Rightarrow A_i \subset Y \Rightarrow x(i) \in Y$$

$$i \notin X \Rightarrow A_i \cap Y = \emptyset \Rightarrow x(i) \notin Y$$

hence by choice of X we can define an index set Y of any arbitrary degree.

It is often the case that we have a specific function in mind when performing software verification. The corresponding index set in this case is a minimal index set.

For this there is a bound on the degree.

Lemma 6.1.1. *Every minimal index set is in* Π_2.

Proof. Take any c. We can easily see that if $\phi_i = \phi_c$, then $W_i = W_c$ and, for every $n \in W_c$, $\phi_i(n) = \phi_c(n)$. So for every n, either $\phi_i(n) \uparrow$ and $\phi_c(n) \uparrow$, or $\phi_i(n) \downarrow = \phi_c(n) \downarrow$.

We can write this as

$$\forall n, [\forall t, \phi_{i,t}(n) \uparrow \wedge \phi_{c,t}(n) \uparrow] \vee [\exists t, \phi_{i,t}(n) \downarrow = \phi_{c,t}(n) \downarrow].$$

Putting this in a prenex normal form we can write I_c as a Π_2 set,

$$\{i \mid \forall n, t \exists s, [\phi_{i,t}(n) \uparrow \wedge \phi_{c,t}(n) \uparrow] \vee [\phi_{i,s}(n) \downarrow = \phi_{c,s}(n) \downarrow]\}.$$

$\square$

It follows then that any minimal index set I has a reduction to Tot. Hence, via this reduction, a partial solution for termination analysis can be applied for the software verification problem exemplified by I. Furthermore, Tot itself is the union of the minimal index sets for the computable total functions. Now we will show that each of these is 1-equivalent to Tot.

Lemma 6.1.2. *Let f be a total computable function. Then $I_f \equiv_1 Tot$.*

Proof. Define

$$h(i,n) = \begin{cases} f(n) & \phi_i(n) \downarrow \\ \uparrow & o.w. \end{cases}$$

By s-m-n there exists a 1-1 total computable $x(i)$ s.t. $\phi_{x(i)}(n) = h(i,n)$. We see that

$$i \in Tot \implies \forall n, \phi_i(n) \downarrow \implies \forall n, \phi_{x(i)}(n) \downarrow = f(n) \implies x(i) \in I_f$$

and

$$i \notin Tot \implies \exists n, \phi_i(n) \uparrow \implies \exists n, \phi_{x(i)}(n) \uparrow \implies x(i) \notin I_f.$$

Hence $Tot \leq_1 I_f$.

□

We can strengthen this result by extending it to every computable function with a computable infinite domain.

Lemma 6.1.3. *Let f be a computable function with a computable infinite domain. Then $I_f \equiv_1 Tot$.*

Proof. Let c be an index for f, so W_c is the computable infinite domain of f. Define $p(n) = |\{i < n \mid i \in W_c\}|$. This is total computable with infinite range and for every element in $W_c = \{a_0 < a_1 < ...\}$, $p(a_i) = i$.

Define

$$h(i,n) = \begin{cases} f(n) & n \in W_c \wedge \phi_i(p(n)) \downarrow \\ \uparrow & o.w. \end{cases}$$

By s-m-n we have a 1-1 total computable function $x(i)$ such that $\phi_{x(i)}(n) = h(i,n)$.

Suppose $i \in Tot$. Then for any n, $\phi_i(p(n)) \downarrow$. So for each $n \in W_c$, $\phi_{x(i)}(n) \downarrow = f(n)$, and for each $n \notin W_c$, $\phi_{x(i)}(n) \uparrow$. Hence $x(i) \in I_f$.

Suppose instead $i \notin Tot$. There must be some k for which $\phi_i(k) \uparrow$. Since W_c is infinite, there is some n in the monotone enumeration of W_c for which $p(n) = k$. Since $n \in W_c$, $f(n)$ is defined but $\phi_{x(i)}(n)$ is not. Hence $x(i) \notin I_f$.

Therefore $Tot \leq_1 I_f$.

□

We can bound below with the 1-degrees of the Σ_1-complete and Π_1-complete sets.

Fact 6.1.4. *Let E denote the computable function with empty domain. We can then* *write its minimal index set in in Π_1 form,*

$$I_E = \{i \mid \forall n, t, \phi_{i,t}(n) \uparrow\}.$$

This is the complement of $K_1 = \{i \mid W_i \neq \emptyset\}$ which is known to be Σ_1-complete. It follows then that I_E is Π_1-complete.

Lemma 6.1.5. *Let f be a computable function with nonempty domain. Then $K_1 \leq_1 I_f$.*

Proof. Let f be a computable function with nonempty domain. We obtain a 1-reduction from $K_1 = \overline{I_E}$ to I_f via s-m-n with

$$\phi_{x(i)}(n) = h(i,n) = \begin{cases} f(n) & \exists k, t, \phi_{i,t}(k) \downarrow \\ \uparrow & o.w. \end{cases}$$

We can see that

$$i \in K_1 \implies \exists k, t, \phi_{i,t}(k) \downarrow \implies \forall n, \phi_{x(i)}(n) = f(n) \implies x(i) \in I_f$$

and

$$i \notin K_1 \implies \forall k, t, \phi_{i,t}(k) \uparrow \implies \forall n, \phi_{x(i)}(n) \uparrow \implies x(i) \notin I_f.$$

Thus $K_1 \leq_1 I_f$.

□

Of the minimal index sets, I_E is strictly alone in it's 1-degree.

Lemma 6.1.6. *No other minimal index set is 1-reducible to I_E.*

Proof. Suppose that $I_f \leq_1 I_E$. By Lemma 6.1.5 it would follow that $K_1 \leq_1 I_E$. But any function witnessing the 1-reducibility of K_1 to I_E would also witness that of I_E to K_1 since they are complements. This would imply that $K_1 \equiv_1 I_E$.

It would follow then by the Myhill Isomorphism Theorem that K_1 and I_E are computably isomorphic – that is, there exists a computable bijection between the two. Since I_E is Π_1, this would imply that the Σ_1 set K_1 is also Π_1, and hence Δ_1. But Δ_1 sets are computable and we know K_1 is not.

□

The minimal index sets of computable functions with finite domain can be further restricted to the lower half of the hierarchy.

Lemma 6.1.7. *Let f be a computable function with finite domain. Then $I_f \in \Delta_2$.*

Proof. Let c be an index for f so that the domain of f is the finite set X. Note

$$I_f = \{i \mid [\forall n \in X, \exists t, \phi_{i,t}(n) \downarrow= f(n)] \wedge [\forall n \notin X, \forall t, \phi_{i,t}(n) \uparrow\}.$$

Because the universal quantifier in the first clause is bounded, it is a Σ_1 statement. The second clause is Π_1. Since both clauses are independent, we can put this into prenex normal form as either a Σ_2 or Π_2 statement. Hence the set is in Δ_2.

□

The functions with nonempty finite domains all share a 1-degree, hence we can conveniently represent the degree with a computable function on a singleton domain.

Lemma 6.1.8. *Let f be a computable function with a nonempty finite domain and let g be a computable function with a singleton domain. Then $I_g \equiv_1 I_f$.*

Proof. Let X be the domain of f and, without loss of generality, let $\{0\}$ be the domain of g. Define $p(n) = \{i < n \mid i \in X\}$ and

$$\phi_{x(i)}(n) = \begin{cases} f(n) & n \in X \wedge \exists t, \phi_{i,t}(0) \downarrow= g(0) \\ \phi_i(1 + n - p(n)) & o.w. \end{cases}$$

Then

$$i \in I_g \implies \forall n, \phi_{x(i)}(n) = \begin{cases} f(n) & n \in X \\ \uparrow & n \notin X \end{cases} \implies x(i) \in I_f$$

and

$$i \notin I_g \implies \left\{ \begin{array}{ll} \phi_i(0)\uparrow & \implies \forall n \in X, \phi_{x(i)}(n)\uparrow \\ \phi_i(0)\downarrow \neq g(0) & \implies \forall n \in X, \phi_{x(i)}(n)\uparrow \\ \exists n > 0, \phi_i(n)\downarrow & \implies W_i \neq X \end{array} \right\} \implies x(i) \notin I_f$$

giving us $I_g \leq_1 I_f$.

Define

$$\phi_{y(i)}(n) = \begin{cases} g(0) & n = 0 \wedge \forall k \in X \exists t, \phi_{i,t}(k)\downarrow = f(k) \\ \phi_i(\mu c.[c \geq n-1 \wedge c \in \overline{X}]) & o.w. \end{cases}$$

Note that since $\overline{X}$ is infinite and computable, $\mu c.[c \geq n-1 \wedge c \in \overline{X}]$ is defined for all $n > 0$, and for each $k \in \overline{X}$ there exists some $n > 0$ s.t. $\mu c.[c \geq n-1 \wedge c \in \overline{X}] = k$.

We then have

$$i \in I_f \implies \phi_{y(i)}(n) = \begin{cases} g(0) & n = 0 \\ \uparrow & n > 0 \end{cases} \implies y(i) \in I_g$$

and

$$i \notin I_f \implies \left\{ \begin{array}{ll} \exists k \in X, \phi_i(k)\uparrow & \implies \phi_{y(i)}(0)\uparrow \\ \exists k \in X, \phi_i(k)\downarrow \neq f(k) & \implies \phi_{y(i)}(0)\uparrow \\ \exists k \notin X, \phi_i(k)\downarrow & \implies \exists n > 0, \phi_{y(i)}(n)\downarrow \end{array} \right\}$$

$$\implies y(i) \notin I_g.$$

Hence $I_f \leq_1 I_g$.

$\square$

This allows us to more precisely fix the location of the 1-degree for computable functions with nonempty finite domains at the level of Δ_2.

Lemma 6.1.9. *Let f be a computable function with nonempty finite domain. Then $I_E \leq_1 I_f$.*

Proof. By the previous lemma we may assume that the domain of f is $\{0\}$. Define

$$\phi_{x(i)}(n) = \begin{cases} f(0) & n = 0 \\ \phi_i(n-1) & o.w. \end{cases}$$

If $i \in I_E$ then $x(i) \in I_f$. If $i \notin I_E$, there exists n such that $\phi_i(n)\downarrow$. It follows that $\phi_{x(i)}(n+1)\downarrow$ so $x(i) \notin I_f$.

$\square$

Corollary 6.1.10. *If f is computable with finite domain, I_f is strictly Δ_2.*

Proof. If I_f is not strictly in Δ_2, then it must be in either Σ_1 or Π_1.

In the first case we would have $I_E \leq_1 I_f \leq_1 K_1$, and in the second $K_1 \leq_1 I_f \leq_1 I_E$.

$\square$

Hence, the minimal index set for every computable function with computable domain falls into one of three 1-degrees. These degrees are linearly ordered $I_E <_1 I_0 <_1 Tot$, where I_0 represents the minimal index set for a function defined only at 0.

There remain open questions in regards to the minimal index sets for computable functions with strictly r.e. domains. These include:

1. Do the minimal index sets of functions with strictly r.e. domains all have the same 1-degree?

2. Is I_0 1-reducible to the minimal index set of some function with a strictly r.e. domain?
3. If not, then is I_E 1-reducible to some function with a strictly r.e. domain?

A positive answer to (1) and (2) would extend the above linear ordering of 1-degrees. A positive answer to (1) and (3) would still establish I_E as the bottom element of the ordering.

The more interesting possibility would be a negative answer for (1). If this is the case, then we have what is perhaps a much more complicated structure. It raises the question of how the r.e. degree of a function's domain might affect the 1-degree of its minimal index set.

6.2 Maximal r.e. subsets of index sets

It is worth noting that if applying this definition of r.e. maximality to ω, we obtain a familiar notion of maximality found in computability theory. Friedberg [18] showed that there are strictly r.e. maximal sets in ω. Yet in the case of Tot this fails to be true.

Theorem 6.2.1. *The index set Tot has no maximal r.e. subset.*

Proof. Let $X \subseteq Tot$ be infinite and r.e. Since there is no enumeration of the total computable functions, there must be some f with no indices in X. Let c be an index for f.

By the padding lemma we can effectively extend i into an infinite set of indices for f, call it F. Since $X \cup F$ is r.e., there must still be a total computable function g s.t. no index of g is in $X \cup F$. Hence $X \neq_{fin} X \cup F \neq_{fin} Tot$.

□

With a bit of work we can extend this result to a much larger class of sets which contains every non-r.e. index set.

Definition 6.2.2. *A set Y is a cylinder if $Y \equiv_1 Y \times \omega$. Equivalently, Y is a cylinder if there exists A s.t. $Y \equiv_1 A \times \omega$.*

Theorem 6.2.3. *If Y is a non-r.e. cylinder it has no maximal r.e. subset.*

Proof. Let f be the computable bijection from Y to $Y \times \omega$ required by Myhill's theorem. Let $X \subseteq Y$ be an infinite r.e. set. Then $f[X]$ is an infinite r.e. subset of $Y \times \omega$.

Define $g(n) = \pi_0(f(k(n)))$, where k is a computable total function with $k[\omega] = X$. Observe that $g[\omega] \subset Y$, and each $a \in Y \backslash g[\omega]$ represents a *splinter* $\{a\} \times \omega$ that does not intersect with $f[X]$.

Because Y is non-r.e., $Y \backslash g[\omega]$ must be infinite. Let $a, b \in Y \backslash g[\omega]$ be distinct. We then have $f[X]$, $\{a\} \times \omega$, and $\{b\} \times \omega$, are pairwise disjoint.

Because f is a computable bijection, $A = f^{-1}[\{a\} \times \omega]$ and $B = f^{-1}[\{b\} \times \omega]$ are both infinite r.e. subsets of Y, and are pairwise disjoint with X. Hence $X \not\equiv_{fin} X \cup A \not\equiv_{fin} X \cup A \cup B \subset Y$.

□

Since every index set is a cylinder, this result subsumes the previous. Theorem 6.2.1 was retained for its connection to the motivating material, and because it is a simple illustration of the same argument found in its generalization, without the additional scaffolding.

PART III

NEURAL NETS

CHAPTER 7

ONGOING RESEARCH ON THE VERIFICATION OF NEURAL NETS

In Section 2.3 of the background material we argued that establishing the expressiveness of neural networks in terms of formal logic could assist in formally verifying their behavior.

Neural nets are often regarded as black boxes. If we want to make claims about their behavior, we need a solid understanding of what is going on inside. Characterizing families of neural nets by their expressive power is one way to do this.

This chapter is a report on ongoing research. The primary result we wish to communicate is the following characterization binary classifiers constructed from ReLU-activated feed forward neural nets of arbitrary size.

Theorem 7.0.1. *The characteristic functions computed by ReLU-activated binary classifiers are precisely those which can be defined in the form*

$$\{x \mid \phi(v_1 \cdot x + r_1 > 0, \dots, v_k \cdot x + r_k > 0)$$

where $v_1, \dots, v_k \in \mathbb{R}^n$, $r_1, \dots, r_k \in \mathbb{R}$, and ϕ is a propositional formula over k terms.

Although stated in terms of vector inequalities, this is actually an equivalence between ReLU-activated binary classifiers and an unquantified fragment of first order arithmetic. This equivalence can be proven mathematically from the following result for general ReLU-activated feed forward neural nets.

Theorem 7.0.2 (Arora, Basu, Mianjy, Mukherjee [32])**.** *The functions computed by ReLU-activated feed forward nets are precisely the continuous piecewise linear functions.*

We are not as interested in the direction which shows that ReLU-activated binary classifiers can represent an arbitrary formula in the given form. Our focus is on the direction which indicates that these are the only functions which can be represented by these networks.

This representation gives us the ability to describe a network logically. If proven constructively, it provides an algorithm for the translation. This provides both a tool and a target for verification. It can be used as a tool to assist in the verification of neural nets, but any algorithm claiming to implement the translation should itself be verified.

Proving such a translation exists by using continuous piecewise linear functions is intuitive clear. However, such intuition relies heavily on details that we can easily fill in by visualizing linear functions in Euclidean spaces. To produce a translation algorithm or a formal proof, these details must be filled in explicitly, leading to a lot of overhead that results in needlessly complex proofs (and potentially an inefficient translation algorithm).

A different approach was needed. The proofs in this chapter communicate the general ideas behind our current work in building a constructive formal proof that ReLU-activated binary classifiers can be expressed in this fragment of first order logic. On review, the techniques seem very similar to those used by Pan and Srikumar [38]. Their result was restricted to ReLU-activated neural nets with two layers, so our work can be seen as an extension of theirs.

Parts of this work have already been formally proven in PVS [37]. This is an ongoing process that proceeds alongside the pen and paper research. The desired end result is a formally verified translation algorithm and a workflow for verifying the properties of ReLU-avtivated binary classifiers. This will serve as a proof of concept for further research into the formal verification of machine learning.

7.1 ReLU and step activation

Analyzing the behavior of neural nets is complicated by activation functions. Every layer of a feed forward network consists of a linear operation composed with a nonlinear function. The output of a deep neural network is the result of numerous compositions of these nonlinear layers. It's these activation functions which limit our ability to simplify the computations into something more comprehensible.

Some of the earliest activation functions were threshold functions such as in the foundational work of McCulloch and Pitts [29]. This was before bias was introduced, and so each neuron might have its own distinct activation threshold. Using the modern convention of assigning bias to a neuron, we may instead represent the thresholds used by McCulloch and Pitts, as well as other authors, by a single activation function,

$$step(x) = \begin{cases} 1 & x > 0 \\ 0 & o.w. \end{cases}$$

This leads to activation behavior that is easy to analyze. Since the outputs at each layer are vectors in $\{0,1\}^n$, for some n, we can see how much of the network is just computing a boolean function. This allows us to conceptualize a step-activated binary classifier as a proposition over the inequalities representing the behavior of the neurons in the first layer (see Lemma 7.2.1).

Using such threshold functions has largely fallen out of favor because they do not have properties suitable for modern training techniques.

Perhaps the most widely used activation function for deep neural networks is the

Rectified Linear Unit (ReLU) function [43],

$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & o.w. \end{cases}$$

This serves as a compromise in that it resembles the step function, yet retains properties more favorable for learning via backpropagation.

Because the range of ReLU is the non-negative reals, ReLU-activated networks can obviously express a wider variety of functions than step-activated networks are able to. This observation does not, however, take into account the fact that one of the common problems to which deep neural networks are applied is that of classifying the members of a space.

Such classification may be done in various ways and involve any number of possible classification categories. In any case, the final output can be reduced to a number of binary decisions – does this input belong to this particular category or not? As such, a classification problem can be represented as a number of binary-valued processes running in parallel.

To obtain such a binary output from a ReLU-activated network, some form of threshold must be applied. It can be shown that ReLU-activated binary classifiers are no more expressive than step-activated binary classifiers. This was shown by Pan and Srikumar [38] for ReLU-activated binary classifiers of depth two. In this chapter we extend this result to deep neural network classifiers of arbitrary depth (Theorem 7.2.3).

By showing that any ReLU-activated binary classifier can be represented by a step-activated binary classifier, we also show that the behavior of these neural nets can be described as a propositional formula over a collection of vector inequalities over the input. This gives us a convenient language in which to reason about the properties of a particular network.

7.2 Reducing ReLU-activated binary classifiers to step-activated binary classifiers

When speaking about neural nets in general, it's easy to see how we might link smaller networks up into a larger network. A (totally connected) feed forward neural network has additional structural requirements – the neurons must be grouped into sequential *layers* with each neuron in one layer connected to every neuron in the previous and subsequent layers.

As noted in Section 2.3 of the background information, we can piece together large feed forward networks from smaller ones by adding in zero weighted connections where needed. This observation is key for building networks recursively for propositional formula.

Lemma 7.2.1. *The characteristic functions computed by step-activated binary classifiers are precisely those which can be defined in the form*

$$\{x \mid \phi(v_1 \cdot x + r_1 > 0, \ldots, v_k \cdot x + r_k > 0)$$

where $v_1, \ldots, v_k \in \mathbb{R}^n$*,* $r_1, \ldots, r_k \in \mathbb{R}$*, and* ϕ *is a propositional formula over* k *terms.*

Proof. ($\rightarrow$) Let F be a step-activated binary classifier of depth greater than one. From each first layer neuron $step(w \cdot x + b)$ we have the inequality $w \cdot x + b > 0$. Since the remaining layers of the step network are restricted to inputs in $\{0, 1\}^n$, they compute a Boolean function. We know these correspond to propositional logic.

($\leftarrow$) The inequalities are represented naturally by neurons in the first layer of the network we construct. It suffices to show that we can construct networks for $P \wedge Q$ and $\neg P$. This represents a functionally complete set of logical operators, so from these networks we can piece together a network for any propositional formula.

The neuron $step(1 \cdot P + 1 \cdot Q - 1)$ computes the conjunction of P and Q. The neuron $step(-1 \cdot P + 1)$ computes the negation of P. $\square$

We next move on to ReLU-activated binary classifiers. The following lemma will serve as the base case for our induction. This is essentially the same result reported by Pan and Srikumar [38]. The proof was arrived at independently prior to identifying the result in the literature.

Lemma 7.2.2. *For any two-layer ReLU-activated binary classifier of width n taking inputs from $\mathbb{R}^m$, there exists an equivalent step-activated binary classifier of width $2^n + n$.*

Proof. Such a classifier has the form $f(x) = step(w_2 \cdot ReLU(w_1 \cdot x + b_1) + b_2)$, with w_1 being an $n \times m$ matrix, b_1, w_2 being n-dimensional vectors, and b_2 being scalar. Expanding out the weight matrices this is

$$step(w_2 \cdot ReLU(w_1 \cdot x + b_1) + b_2) = step\left(w_2 \cdot \begin{bmatrix} ReLU(w_{1,1} \cdot x + b_{1,1}) \\ \vdots \\ ReLU(w_{1,n} \cdot x + b_{1,n}) \end{bmatrix} + b_2 \right)$$

$$= step(b_2 + \sum_{i=1}^{n} w_{2,i} \cdot ReLU(w_{1,i} \cdot x + b_{1,i}))$$

Here $w_{1,i}$ denotes the ith row of w_1 and $b_{1,i}$ denotes the ith element of b_1. For each i define $A_i(x)$ as the inequality $w_{1,i} \cdot x + b_{1,i} > 0$, and the scalar values $y_i = w_{2,i} w_{1,i}$ and $z_i = w_{2,i} b_{1,i}$.

We observe that $A_i(x)$ holds if and only if $w_{2,i} \cdot ReLU(w_{1,i} \cdot x + b_{1,i}) = y_i x + z_i$, since in this case $ReLU$ acts as the identity function. Otherwise if $\neg A_i(x)$, we have $w_{2,i} \cdot ReLU(w_{1,i} \cdot x + b_{1,i}) = 0$. Below we define some notation that will reflect this reasoning on a larger scale.

Considering the entire sequence of predicates $A_1(x), \ldots, A_n(x)$, each possibility corresponds naturally to an element of 2^n where $A_i(x)$ holds if and only if $\tau_{i-1} = 1$. For each $\tau \in 2^n$ we define

$$P_\tau(x) \equiv \bigwedge_{i=1}^{n} \pm_{\tau_{i-1}} A_i(x)$$

where $\pm_0 A$ denotes $\neg A$ and $\pm_1 A$ is A. Further, define

$$Y_\tau = [\tau_0 y_1 \quad \cdots \quad \tau_{n-1} y_n] \qquad \text{and} \qquad Z_\tau = b_2 + \sum_{i=1}^{n} \tau_{i-1} z_i.$$

These weight matrices and bias vectors reflect which parts of the network are zeroed out in each particular case, as per the previous paragraph. Thus we have $P_\tau(x)$ if and only if $w_2 \cdot ReLU(w_1 \cdot x + b_1) + b_2 = Y_\tau \cdot x + Z_\tau$.

We now define

$$F(x) \equiv \bigwedge_{\tau \in 2^n} P_\tau(x) \to Y_\tau \cdot x + Z_\tau > 0.$$

We observe that this is a formal proposition over the behaviors of $2^n + n$ neurons – the n neurons from the first layer of f as represented in the antecedents, and neurons with weights Y_σ and biases Z_σ, for each $\sigma \in 2^n$, which are represented in the consequents.

We claim that $F(x)$ holds if and only if $f(x) = 1$. Fix an input x.

($\to$) For one and only one $\tau \in 2^n$ can $P_\tau(x)$ hold. The clauses containing every other possibility will be vacuously true. By our assumption then we have that $Y\tau \cdot x + Z_\tau > 0$. But $Y\tau \cdot x + Z_\tau = w_2 \cdot ReLU(w_1 \cdot x + b_1) + b_2$, so $f(x) = 1$.

($\leftarrow$)By contrapositive, assume $F(x)$ does not hold. This can only happen if $P_\tau(x)$ holds for some $\tau \in 2^n$ while $Y_\tau \cdot x + Z_\tau <= 0$. But we know $P_\tau(x)$ implies $Y_\tau \cdot x + Z_\tau = w_2 \cdot ReLU(w_1 \cdot x + b_1) + b_2$. It follows then that $f(x) = 0$.

With the logical equivalence of $f(x) = 1$ and $F(x)$ established, we next observe that $F(x)$ is a boolean combination of vector inequalities of the form $w \cdot x + b > 0$. By Lemma 7.2.1

□

We can repeat this construction an arbitrary number of times and recursively construct a step-activated binary classifier.

Theorem 7.2.3. *Let f be a ReLU-activated binary classifier of width n taking inputs from $\mathbb{R}^m$. There exists a step-activated binary classifier g such that $g(x) = f(x)$ for all inputs.*

Proof. We proceed by induction on the depth of f with Lemma 7.2.2 serves as our base case. Suppose then that the theorem holds for all ReLU-activated binary classifiers of depth n and that f has depth $n+1$.

Let $f(x) = h(L_1(x))$ where L_1 is the behaviors of the first layer on the input and the remainder of the network is represented by h. Then h is a ReLU-activated binary classifier of depth n, so by our induction hypothesis we have some step-activated binary classifier h' which is equivalent to h.

Let L_2 be the first layer of h' and h'' be the rest (so that $h' = h'' \circ L_2$), and let k be the width of L_2. The composition $L_2 \circ L_1$ outputs binary vectors of some dimension k (for which we may note that $k <= c + 2^c$ where c is the width of the second layer in the original ReLU network).

Fix one of the k neurons in L_2 with some weight w and bias b. Then the behavior of that neuron on the output of L_1 has the form $step(w \cdot L_1(x) + b)$, and so is a ReLU-activated binary classifier of depth 2. Applying Lemma 7.2.2 k times, once for each of these subnetworks ending at a neuron in L_2 gives us a sequence of k separate step-activated binary classifiers that reproduce the behavior of $L_2 \circ L_1$.

We note here that each of these k separate classifiers shares the original n neurons from L_1 in common. The new 2^n neurons constructed for each are dependent upon a particular neuron in L_2. Hence we can construct this network with $n + k2^n$ neurons.

As noted in Remark 2.3.2 we can construct from these k networks a single network N that performs their behavior in parallel. Hence $N(x) = L_2(L_1(x))$. Defining $g(x) = h''(N(x))$, we see that $g(x) = f(x)$.

$\square$

Applying Lemma 7.2.1 to the step-activated classifier obtained in this theorem gives us the direction of Theorem 7.0.1 we needed.

7.3 Additional work

The methods used in the proofs here are all effective, so we may construct an algorithm to translate a ReLU-activated binary classifier into a collection of vector inequalities and a propositional formula. While arrays of ReLU-activated neurons are not very amenable to analysis since the nonlinear behavior is nested from layer to layer, it is much easier to reason about when particular inequalities hold true for an input and whether a propositional formula is satisfiable.

The assurance of machine learning is a problem currently receiving much attention in the field of formal methods. Although we can see that a direct application of the methods found in the proofs in this chapter would lead to a translation that has exponential growth in size, These can all be arithmetically computed from the original weights and biases of the neural net.

Furthermore, similar computations would take place in the original neural net, so there is no real increase in computational space or time. What we are essentially doing here is “front-loading” these computations in a sense, but doing so in a way that we can eliminate a large number of them based on which inequalities the input satisfies.

Although the family of neural nets we address here is limited, ReLU-activated feed forward networks and classification problems are an important part of machine learning and presented a reasonable starting point for exploring this sort of approach to machine learning verification. Much work remains, but there was also a lot of work required to get to this point.

To prepare for this stage of formalization we designed a new matrix formalism for the NASA PVS libraries [35]. We can think of defining a new formalism as choosing a concrete model for a particular theory we wish to study, or fixing a particular encoding in computability theory. Our choices here have a very real impact on how things will be proven and how easily those proofs will be attained. Prior existing matrix formalisms in the NASA PVS libraries relied on lists of lists of numbers to represent matrices. Using this data structure required extensive use of induction and proving that matrices satisfied strict requirements on their dimensions when performing operations.

In order to support freely combining neural nets, more flexibility was desired. The new matrix formalism supports performing matrix operations with more freedom, performing proofs by indexing (which is not possible with lists of lists), and defining matrices taking their values from a user defined set. The fundamental representation of a matrix in this formalism is a pair consisting of a function $f(x, y)$ returning the values of the matrix at index (x, y), and a tuple of natural numbers giving bounds, beyond which f can only return some default value (also user provided).

The design and implementation of these new matrices too the better part of a year. The formal work in this chapter was the problem we chose to test the robustness of the new matrices. More details on the formalization may be found in Appendix C.

APPENDIX A

COUNTING CODE FOR STRONG SUBTREES

The calculations for counting the number of strong subtrees of height m in the binary tree of height n in Chapter 3 were extensive. At some point it becomes helpful to automate these. This was first done in Python, and later in Standard ML. These programs were used to generate sequences and tables of values that were used for comparison and study. Further, they offer an empirical way to experience the equations found in the corresponding sections.

The code below was written in Standard ML and ran successfully under Poly/ ML 5.7.1. Python was discarded as it is entirely unsatisfactory for any significant recursive application. A dialect of ML was chosen because it is a functional programming language and because ML are often considered to have good readability.

Unfortunately, recent versions of Poly/ML have chosen 64 bit integers as default. We were able to work around this by using a large integer datatype, with the primary cost being a bit of clutter up front to define some convenient syntax and some common functionality that might have otherwise been available for the default integer representation.

First we introduce the large integers under the alias of `Big`, define a few common list functions for lists of such numbers, functions to facilitate creating sequences and tables, and some mathematical operations on those numbers including exponentiation, iterated product, and iterated sum.

```
type Big = LargeInt.int
```

```
fun big x = Int.toLarge x

fun nth (list: Big list) (index: Big) =
  if (index < big (length list))
  then List.nth (list, Int.fromLarge index)
  else 0

fun cons (value: Big) (list: Big list) = [value]@list

fun make_row (start: Big) (stop: Big) (fix: Big) function =
  if start = stop then [function (start, fix)]
  else (cons (function (start, fix))
             (make_row (start+1) stop fix function))

fun make_table row1 rown col1 coln function =
  if row1 = rown then [make_row col1 coln row1 function]
  else [make_row col1 coln row1 function]@
       (make_table (row1+1) rown col1 coln function)

infix 8 ^
fun (x: Big)^0: Big = 1
|   (x: Big)^(y: Big) =
      if y > 0 then x * (x^(y-1))
      else 0

fun product (start, stop: Big) expression: Big =
  if start <= stop then (expression start) *
```

```
                      (product (start+1, stop) expression)
  else 1


fun sum (start, stop: Big) expression: Big =
  if start <= stop then (expression start) +
                      (sum (start+1, stop) expression)
  else 0
```

Next we develop the code needed for an initial count of $f(n, m)$. This corresponds to the formula found in Theorem 3.0.3. The code here was split across multiple functions for easier writing and easier reading.

```
fun initial_term (list: Big list) (index: Big): Big =
  let val a = (fn i => nth list i)
  in
    (2^(a(index) - a(index-1) - 1))^(2^(index-1))
  end


fun initial_product (bound: Big) (list: Big list): Big =
  product (1,bound) (fn i => initial_term list i)


fun initial_sum 1 bound expr list =
      sum (1, bound) (fn i => expr ([0,i]@list))
|   initial_sum reps bound expr list =
      if reps > 1 then sum (reps, bound)
                         (fn i => initial_sum (reps-1) (i-1)
                          expr (cons i list))
      else 0
```

```
fun f (n, m) = initial_sum m n (initial_product m) []
```

The results of Theorem 3.3.1, and Theorem 3.2.3 calculated with that result, are reflected in the code below. These are more clear than the previous due to the lack of iterated sums and products.

```
fun rho (n, m: Big): Big =
  if n <= 0 orelse m <= 0 then 0
  else if n > 1 andalso m = 1 then 0
  else if n < m then 0
  else if n = m then 1
  else sum (1, n-m+1)
           (fn i => (2^(i-1))^(2^(m-1)) * rho(n-i, m-1))


fun frec (n, m: Big): Big =
  3*f(n-1, m) - 2*f(n-2, m) + rho(n,m)
```

Here we have Definition 3.3.2 for þ and the calculation of ρ and f with this function. This corresponds to Theorems 3.3.3 and 3.4.1.

```
fun thorn (a, 0: Big): Big = 1
|   thorn (a, b: Big) =
      case (a <= 0, b < 0) of
        (_, true) => 0
      | (true, false) => 0
      | (false, false) => sum (1, a)
                          (fn i => 2^(2^i) * thorn(i, b-1))
```

```
fun rhothorn (n, m: Big): Big = thorn(m-1, n-m)


fun fthorn (n, m: Big): Big =
  sum (1, n) (fn i => (2^(n+1-i) - 1) * thorn(m-1, i-m))
```

APPENDIX B

EMBEDDING POST'S CORRESPONDENCE PROBLEM INTO EDGE SOUNDNESS

The original proof of Lemma 5.3.4 took a very different form. It began as a direct application of a similar result for the programming language Prolog. There was a desire to obtain an MWG as simple as possible, so the essence of this original idea was distilled down into the form currently found in the chapter on termination.

We recall that G is edge-sound when, for every $(A, a), (B, b) \in V_B$ and $i \in \omega$, if $A(i)$ and $B(a(i))$ then $A \to B$ is in E_G. Since A and B are computable predicates and we can assume a is total, we see that this definition witnesses that the collection of all edge-sound MWGs $\mathcal{G}$ is a Π^0_1 set. To show that it is strictly Π^0_1 we need only reduce another Π^0_1 set to $\mathcal{G}$.

For this we modify a proof by Cholak in [10] and Cholak and Blair in [11] which embeds the Post correspondence problem into Prolog to show the undecidability of local stratification.

Definition B.0.1 (Post correspondence problem)**.** *Let Γ be an alphabet with $|\Gamma| \geq 2$. Fix a computable enumeration p of pairs of finite matched length sequences of words $\langle(\alpha_0, \ldots, \alpha_n), (\beta_0, \ldots, \beta_n)\rangle$ in Γ. A solution is a finite sequence of indices $k_1, \ldots, k_m$ such that*

$$\alpha_{k_1}^\frown \alpha_{k_2}^\frown \cdots ^\frown \alpha_{k_m} = \beta_{k_1}^\frown \beta_{k_2}^\frown \cdots ^\frown \beta_{k_m}.$$

Let $\mathcal{P}$ be the collection of $i \in \omega$ such that $p(i)$ has a solution.

It is known that $\mathcal{P}$ is a Σ^0_1-complete problem, so its complement is Π^0_1-complete. Cholak demonstrated how, given an instance of the Post correspondence problem, to construct a Prolog program which terminates precisely when the input fails to be a solution.

We can modify this approach to construct algorithms in general recursive form that accomplish the same. The only obstacle arises from the inability to arbitrarily restrict the input of a general recursive function. We can work around this by including a validity check in our cases.

Definition B.0.2. *Fix an instance of the Post correspondence problem. Given a finite sequence of natural numbers $s = \langle n_0, \cdots, n_k \rangle$ and two words w_1, w_2, we say that the tuple (s, w_1, w_2) is valid if*

$$w_1 = \alpha_{n_0}^\frown \cdots ^\frown \alpha_{n_k} \text{ and } w_2 = \beta_{n_0}^\frown \cdots ^\frown \beta_{n_k}.$$

This property is computable and may be written as $valid(s, w_1, w_2)$.

With this property defined for each instance of the Post correspondence problem we can consider the following function.

$$p(s, t, w_1, w_2) =$$

$$\begin{cases} 0 & \neg valid(s, w_1, w_2) \\ 0 & valid(s, w_1, w_2) \land t = \emptyset \land s = \emptyset \\ 1 & valid(s, w_1, w_2) \land t = \emptyset \land s \neq \emptyset \land w_1 \neq w_2 \\ p(\emptyset, s, \emptyset, \emptyset) & valid(s, w_1, w_2) \land t = \emptyset \land s \neq \emptyset \land w_1 = w_2 \\ p(s^\frown n_0, t', w_1^\frown \alpha_{n_0}, w_2^\frown \beta_{n_0}) & valid(s, w_1, w_2) \land t = n_0^\frown t' \end{cases}$$

Here the sequence t in each step represents a partial solution we use to construct

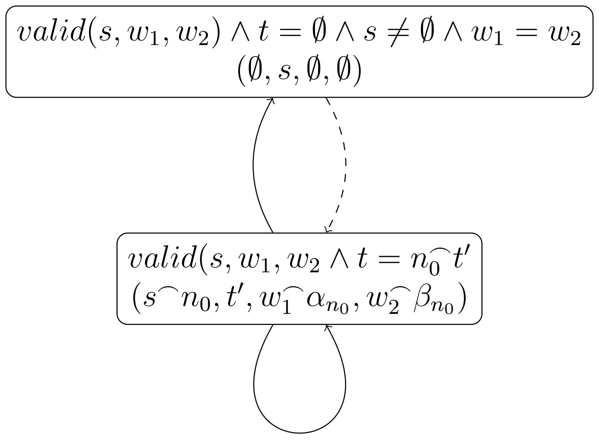
$valid(s, w_1, w_2) \wedge t = \emptyset \wedge s \neq \emptyset \wedge w_1 = w_2$
$(\emptyset, s, \emptyset, \emptyset)$
$valid(s, w_1, w_2 \wedge t = n_0^\frown t'$
$(s^\frown n_0, t', w_1^\frown \alpha_{n_0}, w_2^\frown \beta_{n_0})$

Figure B.1. The MWG we construct for Post's correspondence problem. The dashed arrow represents the missing edge which encodes the complement of this undecidable problem.

the words w_1, w_2, and s accumulates each step taken from t as it is used. The first case essentially aborts the function if the input does not represent the partial verification of a solution. The second case avoids the trivial solution.

The remaining three cases are those that occur when the input represents a valid potential solution to the given instance. The third and fourth cases correspond to the failure and success of verifying t as a solution, respectively. The fifth cases shows the intermediate steps taken in the verification attempt.

An MWG for p must have two vertices, (A, a) for the fourth case and (B, b) for the fifth case. We construct G with these vertices and the edges $B \to B$ and $B \to A$. It's apparent from the case conditions A that leaving out the edge $A \to A$ is sound. The edge-soundness of G then only relies on the missing edge $A \to B$. Since this transition only occurs when there is a solution to this instance of the Post correspondence problem, determining edge soundness for G also determines whether this instance is a member of $\overline{P}$. See Figure B.1.

This reliance on a single edge where edge-soundness implies membership suggests a less cumbersome approach. Generalizing our construction we can formulate a method by which, given any Π^0_1 set X, we can construct a family of simple MWGs which are edge-sound precisely when they correspond to an element of X. This is the form the final proof takes.

APPENDIX C

GENERIC MATRICES FOR THE NASA PVS LIBRARY

The NASA PVS library has various implementations of vectors and matrices, but these are largely defined in terms of real numbers, so they are not suited for every applications. As an example, we might work with matrices of functions for a problem involving differential equations. For arguments of this type it would be useful to have a common formal structure with associated lemmas which is not tied to a specific type.

Another desired improvement would be the ability to freely index matrices and vectors in a proof. Existing definitions often use lists to represent vectors and lists of lists to represent matrices. Proofs on these definitions are typically performed inductively, even when the argument might naturally be made without such a heavy tool as formal induction.

To better better support such work, *generic matrices* were defined. In the specification language of PVS [37] these are defined using a type known as a *record.* Mathematically, we can define them as follows.

Definition C.0.1. *A generic matrix M valued in set X is a function $f : \omega \times \omega \to X$ and a pair of numbers $(n, m) \in \omega \times \omega$ such that, for any (a, b) such that $a \geq n$ or $b \geq m$, $f(n, m)$ is some default value. We call the function the* indexing function *of M and the pair of numbers the* dimensions *of M, with n being the* height *of M and m being the* width.

For technical reasons we also restrict the dimension (n, m) to be such that $n = 0$ if and only $m = 0$. We define generic vectors in a similar manner.

For the purpose of this appendix, we will write the indexing function of M at (i, j) as $M(i, j)$, and the dimension of M as $dim\ M = (height\ M, width\ M)$.

In order to define operations on these generic structures, an operation on the set of values along with a default value with certain properties must be provided. Using a pointwise matrix operation as an example, we require the default value to be idempotent with respect to the underlying operation to preserve bounds. Hence we can do the following.

Definition C.0.2. *We define the pointwise operation $M + N$ by $[M + N](i, j) = M(i, j) + N(i, j)$, with*

$$dim\ M + N = (\max\{height\ M, height\ N\}, \max\{width\ M, width\ N\}).$$

This is simple compared to the corresponding definitions for matrices represented by lists of lists, where we would have to recursively construct a new list of lists. With lists of lists we not only have to work recursively, but we often have to make sure that our lists are of the correct length in order for the operations to be well-defined. In a formal proof, we have to prove this explicitly every time we invoke a matrix operation.

By design, operations on generic matrices are not restricted by their dimensions. Since the dimension only bounds the non-default values, a generic matrix can be treated as if it has been embedded into a larger matrix with the default value occupying the extra entries. This allows us to concisely specify things like sparse matrices or block matrices.

The primary strength of the list of lists representation is that it is convenient for computation. The dimensions of a generic matrix bound the relevant part of the index functions domain and the algebraic properties of the default value guarantee that these bounds are preserved, so the operations are computable. Even so, it is not clear to PVS how it should display the results of such an operation. This problem is closely related to ground terms in mathematical logic, and the undecidability of equivalence in the lambda calculus.

To take full advantage of the computational tools PVS provides, we define mappings between lists of lists and generic matrices. A mapping from lists of lists to generic matrices allows the user to input a matrix's values explicitly, while the other direction allows us to tell PVS how a generic matrix should be displayed to the user.

We might define the first as $Abstract(M : list\ of\ lists)$ is the zero generic matrix if M has length 0, otherwise it is the generic matrix with the indexing function

$$f(i,j) = \begin{cases} M_{i,j} & i < |M| \wedge j < \max_{i<|M|}\{|M_i|\} \\ default & o.w. \end{cases}$$

and dimension $(|M|, \max_{i<|M|}\{|M_i|\})$. We can remove the max function and simplify this someway by requiring the lists in M to each have the same length.

The mapping from generic matrices to lists of lists is defined by two recursive functions. The first function is over the rows of the generic matrix, which applies as well to generic vectors,

$$eval(v : generic\ vector) = \begin{cases} \lambda & |v| = 0 \\ v_0 ^\frown eval(\pi(v)) & o.w. \end{cases}$$

where $^\frown$ is list concatenation, λ is the empty list, and π is the projection of v into a space of one fewer dimensions, removing it's first component. The matrix function can then be defined in terms of this,

$$Eval(M : generic\ matrix) = \begin{cases} \lambda & height\ M = 0 \\ eval(M_0) ^\frown Eval(\Pi(M)) & o.w. \end{cases}$$

where $\Pi(M)$ is M without its first row.

Using the definitions specified in PVS we can formally prove the following lemma.

Lemma C.0.3. *Let L be a list of lists containing lists of the same length, and M be a generic matrix. Then $Eval(M) = L$ if and only if $M = Abstract(L)$.*

This result allows us to more easily utilize PVS to do the work for us in proofs requiring extensive computation by providing the link between the two forms.

More information on this formalization can be found by reviewing the NASA PVS library [35] (in the v8.0 branch, pending our full migration to PVS 8.0) on Github).

BIBLIOGRAPHY

1. P.-E. Anglès d'Auriac, P. A. Cholak, D. D. Dzhafarov, B. Monin, and L. Patey. Milliken's tree theorem and its applications: a computability-theoretic perspective. *Mem. Amer. Math. Soc.*, 293(1457), 2024.

2. A. B. Avelar. Formalização da automação da terminação através de grafos com matrizes de medida. 2014.

3. J. Avigad. Formalizing forcing arguments in subsystems of second-order arithmetic. *Annals of Pure and Applied Logic*, 82(2):165–191, 1996.

4. J. Avigad. Mathematics and the formal turn. *Bulletin of the American Mathematical Society*, 61(2):225–240, 2024.

5. F. Z. Blando. *Patterns and Probabilities: A Study in Algorithmic Randomness and Computable Learning.* Stanford University, 2020.

6. J. R. Büchi. On a decision method in restricted second order arithmetic, logic, methodology and philosophy of science (proc. 1960 internat. congr.), 1962.

7. H. Chase and J. Freitag. Model theory and machine learning. *Bulletin of Symbolic Logic*, 25(3):319–332, 2019.

8. K.-L. Chen, H. Garudadri, and B. D. Rao. Improved bounds on neural complexity for representing piecewise linear functions. *Advances in Neural Information Processing Systems*, 35:7167–7180, 2022.

9. D. Chiang, P. Cholak, and A. Pillay. Tighter bounds on the expressivity of transformer encoders. In *International Conference on Machine Learning*, pages 5544–5562. PMLR, 2023.

10. P. Cholak. Post correspondence problem and prolog programs. Technical report, Technical report, Univ. of Wisc., Madison, 1988.(manuscript), 1988.

11. P. Cholak and H. A. Blair. The complexity of local stratification. *Fundamenta Informaticae*, 21(4):333–344, 1994.

12. P. A. Cholak, C. G. Jockusch, and T. A. Slaman. On the strength of Ramsey's theorem for pairs. *Journal of Symbolic Logic*, pages 1–55, 2001.

13. C. T. Chong, T. A. Slaman, and Y. Yang. The inductive strength of Ramsey's theorem for pairs. *Advances in Mathematics*, 308:121–141, 2017.

14. J. Chubb, J. L. Hirst, and T. H. McNicholl. Reverse mathematics, computability, and partitions of trees. *The Journal of Symbolic Logic*, 74(1):201–215, 2009.

15. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

16. S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *The journal of symbolic logic*, 44(1):36–50, 1979.

17. M. Fraser, A. Granville, M. H. Harris, C. McLarty, E. Riehl, and A. Venkatesh. Will machines change mathematics? *Bulletin (New Series) of the American Mathematical Society*, 61(2), 2024.

18. R. M. Friedberg. Three theorems on recursive enumeration. i. decomposition. ii. maximal set. iii. enumeration without duplication. *The Journal of Symbolic Logic*, 23(3):309–316, 1958.

19. K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

20. E. M. Gold. Language identification in the limit. *Information and control*, 10(5): 447–474, 1967.

21. A. Goujon, A. Etemadi, and M. Unser. On the number of regions of piecewise linear neural networks. *Journal of Computational and Applied Mathematics*, 441: 115667, 2024.

22. P. Hájek and P. Pudlák. *Metamathematics of first-order arithmetic*, volume 3. Cambridge University Press, 2017.

23. J. Ketonen and R. Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2):267–314, 1981.

24. L. A. Kołodziejczyk and K. Yokoyama. Some upper bounds on ordinal-valued Ramsey numbers for colourings of pairs. *Selecta Mathematica*, 26(4):56, 2020.

25. L. A. Kołodziejczyk, T. L. Wong, and K. Yokoyama. Ramsey's theorem for pairs, collection, and proof size. *arXiv preprint arXiv:2005.06854*, 2020.

26. M. C. Laskowski. Vapnik-chervonenkis classes of definable sets. *Journal of the London Mathematical Society*, 2(2):377–384, 1992.

27. Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. *Advances in neural information processing systems*, 30, 2017.

28. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *International Conference on Computer Aided Verification*, pages 401–414. Springer, 2006.

29. W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

30. A. N. Migunov. *Randomness and dimension in computational learning and analog computation*. PhD thesis, Iowa State University, 2022.

31. K. R. Milliken. A Ramsey theorem for trees. *Journal of Combinatorial Theory, Series A*, 26(3):215–237, 1979.

32. A. Mukherjee, A. Basu, R. Arora, and P. Mianjy. Understanding deep neural networks with rectified linear units. In *International Conference on Learning Representations*, 2018.

33. C. A. Muñoz, M. Ayala-Rincón, M. M. Moscato, A. M. Dutle, A. J. Narkawicz, A. A. Almeida, A. B. Avelar, and T. M Ferreira Ramos. Formal verification of termination criteria for first-order recursive functions. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

34. NASA Langley Formal Methods. The NASA Langley Formal Methods Research Program, 2024. URL `https://shemesh.larc.nasa.gov/fm/`.

35. NASA Langley Formal Methods. NASALib, 2024. URL `https://github.com/nasa/pvslib`.

36. S. Ovchinnikov. Max-min representation of piecewise linear functions. *Contributions to Algebra and Geometry*, 43(1):297–302, 2002.

37. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

38. X. Pan and V. Srikumar. Expressiveness of rectifier networks. In *International conference on machine learning*, pages 2427–2435. PMLR, 2016.

39. J. Paris. A mathematical incompleteness in Peano arithmetic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 1133–1142. Elsevier, 1977.

40. L. Patey and K. Yokoyama. The proof-theoretic strength of Ramsey's theorem for pairs and two colors. *Advances in Mathematics*, 330:1034–1070, 2018.

41. A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 32–41. IEEE, 2004.

42. P. Pudlák. The lengths of proofs. *Handbook of proof theory*, 1998.

43. P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

44. H. G. Rice. On completely recursively enumerable classes and their key arrays. *The Journal of Symbolic Logic*, 21(3):304–308, 1956.

45. S. G. Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2009.

46. N. J. A. Sloane and T. O. F. Inc. The on-line encyclopedia of integer sequences, 2020. URL `http://oeis.org/?language=english`.

47. R. I. Soare. *Turing computability: Theory and applications*, volume 300. Springer, 2016.

48. S. Steila and K. Yokoyama. Reverse mathematical bounds for the termination theorem. *Annals of pure and applied logic*, 167(12):1213–1241, 2016.

49. S. Todorcevic. *Introduction to Ramsey spaces*, volume 174. Princeton University Press, 2010.

50. S. Volkov. On the class of Skolem elementary functions. *Journal of Applied and Industrial Mathematics*, 4:588–599, 2010.

51. S. Wang and X. Sun. Generalization of hinging hyperplanes. *IEEE Transactions on Information Theory*, 51(12):4425–4431, 2005.

This document was prepared & typeset with pdfLaTeX*, and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.2017.2[2017/05/09])*