Formalized Structured Coding (WIP)

Loris Jautakas

Decemeber 2022

Abstract

We wish to generalize the notion of an algebraic code. We first review some general coding theory, and we define a general notion of a structured code. This can allow for the study of generalized algebraic codes, not just the study of a code using a specific algebraic structure. We also include an appendix that covers basic model theory. Please note this document and the project as a whole is a work in progress. If you happen to catch an error, please let us know.

Contents

1 Introduction

The study of algebraic structures has traditionally been concrete. Usually when someone begins studying algebra, they generally first fix some algebraic structure (or structures) to study, and then they study properties of that algebraic structure, or of maps between two structures. There are many applications of algebra in computer science, like in coding theory. Generally it is also true that when people learn about traditional algebraic codes, they fix some algebraic structure, and consider codes that use that structure in some way. In this paper we generalize some applications of algebra using model theory. Model theory is the study of logic for general algebraic structures. This means that we can not only talk about standard algebraic objects such as groups or rings, but also some nontraditional structures such as ordered sets or even graphs.

In section 2, we have a brief review about the basics of coding theory. We cover a high level overview of channel coding: the process of encoding information for error correction, source coding: the process of encoding information for data compression, and cryptographic coding: the process of encoding information for private communication over a public channel. We then define a couple of performance measures, including error probability, asymptotic probability of error, and time complexity.

In section 3, we define the notion of a structured code. This is a generalized version of what it means to be a traditional algebraic code. Then we give a few examples such as a code of ordered binary strings, a classic linear code, and touch on what homomorphic encryption is. This formalism can allow for the study of general algebraic codes, such as the code of ordered binary strings, which is typically not considered an algebraic code.

At the end of this paper you will find an appendix that covers some essential concepts in computability theory and model theory. These unfortunately may be necessary to read over in order to understand the material. The model theory section is detailed and abstract, so we have tried to design the examples to fit with the concrete problems that we cover elsewhere.

2 General Coding

2.1 Summary

In this section we very briefly summarize general coding. We cover some of the basic coding problems including source coding, channel coding, and encryption. We then define some notions of performance measures.

2.2 Coding problems

There are three main types of code that we will examine. These are:

1. Source coding
2. Channel coding
3. Cryptographic coding

We briefly cover the simplified definitions of these codes below:

Definition 2.1: Standard codes

A code is a tuple

$$\mathbf{C} = (X, Y, \text{enc}, \text{dec}) \tag{1}$$

Such that:

- X, Y are sets, X is called the message set, with elements $m \in X$ called messages
- $\text{enc} : X \to Y$: called the encoder
- $\text{dec} : Y \to X$: called the decoder

From the above items, we typically define extra items such as:

- $C = \text{im}(\text{enc})$: also called the code set, with elements $\text{enc}(x) = c \in C$ called code words
- $X' = \text{im}(\text{dec} \circ \text{enc})$

Source codes

Source codes are used for data compression. We usually wish to store data in a smaller set, and thus we would need to add the condition that $|Y| \leq |X|$.

Channel codes

Channel codes are used for the transmission of information over a noisy channel. Thus we need to add redundancy: we need to add the condition that $|Y| \geq |X|$.

Cryptographic codes

Cryptographic codes are a bit different. The purpose of a cryptographic code is to transmit messages over a public channel in secret. Since we almost always want the sender and receiver to be able to communicate without errors, we also add the condition that $|Y| \geq |X|$, otherwise there may exist an encoder that maps two messages to the same item, and therefore decoding would be impossible. So we should also add the condition that $\text{dec} \circ \text{enc} = I_{\mathcal{X}}$, i.e we can encode and decode without any errors. We usually call the process of encoding as encryption, and the process of decoding as decryption. Generally also we call elemenets in the code set as the corresponding cyphertext for some message. Generally we also assume that there is some private information that the sender and receiver share, called a key.

In all of these coding schemes, first a random message m is chosen from X with some distribution $p_x(\cdot)$. Then the sender encodes that message using enc to get codeword $y = \text{enc}(x)$. After this, the reciver recives some y'. For channel coding, y is sent through some noisy communications channel to become $y' = \text{Er}(y)$, where Er is some random function that applies some error to y. In source codes, $y' = y$. In cryptographic codes, some third party (called the adversary) gets access to y, but also $y' = y$. Then in all codes the receiver applies $\text{dec}(y') := x'$, the estimate for the message sent by the sender. In cryptographic codes, the adversary also has some decoding map which they apply to get x'_A, the adversaries estimate of the sent message.

2.3 Performance measures

So far, we have only considered the definition of a code. This does not tell us how good a code is. Here we cover a couple of different performance measures that may be good for different scenarios.

Definition 2.2: Probability of error

Suppose you are given some code $\mathbf{C} = (X, Y, \text{enc}, \text{dec})$. Let $p_X(\cdot)$ be a distribution on X that corresponds to what message the sender picks to send. Then define the probability of error over code $\mathbf{C}$ with input distribution p_x as:

$$p_{err}(\mathbf{C}, p_X) = \Pr_{x \sim p_X}[x \neq \text{dec} \circ \text{enc}(x)] \tag{2}$$

Note that [Definition 2.2] is useful for source coding, if we compress something we may not always get it back. It can be extended for channel coding as well, because we may have some unrecoverable error that occurs when we attempt to send some code word. We extend the definition simply by changing $\text{dec} \circ \text{enc}(x)$ above to $\text{dec} \circ f \circ \text{enc}(x)$, where f is the channel, and could possibly add some error.

Definition 2.3: Asymptotic probability of error

Suppose we have an infinite sequence of codes $\mathbf{C}_1, \mathbf{C}_2, \ldots$. We define the asymptotic probability of error as:

$$p_{aerr}((\mathbf{C}_n)_n, p_X)) = \lim_{n \to \infty} \Pr_{x \sim p_X}[x \neq dec_n \circ \text{enc}_n(x)] \tag{3}$$

In information theory, the concept of asymptotic probability of error is used a lot. Claude Shannon's source coding theorem states that we can achieve capacity with asymptotic probability of error 0, meaning that there is some sequence of source codes $\mathbf{C}_1, \ldots \mathbf{C}_n$ with corresponding $X_1, Y_1, X_2, Y_2, \ldots$ such that $|Y_n| \cong n * H[p_X]$, where $H[\cdot]$ is the Shannon entropy of p_X, the distribution on the messages, such that $p_{aerr}((\mathbf{C}_n)_n, p_X) = 0$.

A different measure of performance is the time/space complexity needed for the encoder and decoder. This is in some sense more of a practical performance measure, but still has some theoretical interest. There are also

performance measures that you can define on cryptographic codes, but those require further explanation, and we will not cover these in this paper.

3 Structured Coding

3.1 Introduction

In this section, we present a formalism of algebraic codes, which we call structured codes. Generally, algebraic codes are introduced concretely, for example usually when someone learns about linear codes, they are taught that a linear code is some code where the encoder is linear. By using model theory, we can generalize this idea, and give the most general definition of what a structured code should be.

3.2 Structured Codes Introduction

Definition 3.1: $[\mathcal{L}, T]$-code

Given a language $\mathcal{L}$ and a $\mathcal{L}$-theory T, an $[\mathcal{L}, T]$**-code** is a tuple

$$\mathfrak{C} = (\mathcal{X}, \mathcal{Y}, \phi_e, \phi_d, \Pi) \tag{4}$$

Such that:

- $\mathcal{X}, \mathcal{Y}$ are $[\mathcal{L}, T]$-structures. $\mathcal{X}$ is called the **message structure**.
- $\phi_e : \mathcal{X} \to \mathcal{Y}$ is a $\mathcal{L}$-homomorphism.
- $\phi_d : \mathcal{Y} \to \mathcal{D}$ as some $\mathcal{L}$-homomorphism with the constraint that if $y = \phi_e(x) \in \operatorname{im} \phi_e$ for some $x \in X$, then $\phi_d(y) = \phi_d(\phi_e(x)) = [x]_{\ker \phi_e}$.
- $\Pi : \mathcal{X}/\ker(\phi_e) \to \mathcal{X}$ is a $\mathcal{L}$-homomorphism with the constraint that $\Pi([x]_{\ker \phi_e}) = x' \in [x]_{\ker \phi_e}$. This means that $\Pi([x]_{\ker \phi_e})$ will just pick some element from the given equivalence class $[x]_{\ker \phi_e}$.

From the above items, we typically define extra items such as:

- $\mathcal{C} = \operatorname{im}(\phi_e)$ (also called the **code structure)**
- $\mathcal{D} = \mathcal{X}/\ker(\phi_e)$
- $\text{enc} = \phi_e$
- $\text{dec} = \Pi \circ \phi_d$
- $\mathcal{X}' = \operatorname{im}(\Pi) = \operatorname{im}(\text{dec} \circ \text{enc})$ (is an $\mathcal{L}$-structure)

Generally we call $\mathcal{X}$ the **message structure** and $\mathcal{C}$ the **code structure**. We call ϕ_e the **encoder**, and we call $\Pi \circ \phi_d$ the **decoder**. If we think back to the coding problem from [Definition 2.1]: for a structured code, encoding is now done via some structure preserving map, i.e. $\text{enc} = \phi_e$, and decoding is now done first by applying some structured map ϕ_d to get a quotient structure $\mathcal{D}$, and then by using Π to pick a representative $\in \mathcal{X}$, so putting them together we get $\text{dec} = \Pi \circ \phi_d$.

Note that some of the time, we will have $\mathcal{X} = \mathcal{Y}$, and so we can define a simpler version of a structured code, called an endocode.

Definition 3.2: $[\mathcal{L}, T]$-endocode

A $[\mathcal{L}, T]$ endocode is a tuple

$$\mathfrak{C} = (\mathcal{X}, \phi_e, \phi_d \Pi) \tag{5}$$

where we let $\mathfrak{C} = (\mathcal{X}, \mathcal{X}, \phi_e, \phi_d, \Pi)$ (which is a $[\mathcal{L}, T]$-code as defined above).
Note that the isomorphism theorem [theorem A.1] tells us that $\mathcal{D} = \mathcal{X}/\ker \phi_e \simeq \operatorname{im} \phi_e$.

We now need to show a basic result that we will use later:

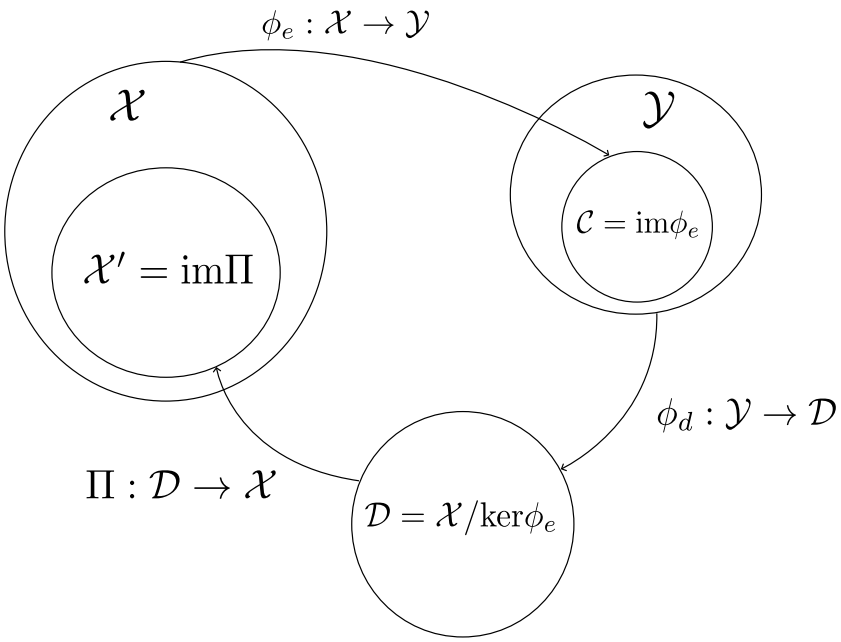

Figure 1: Diagram of a structured code from [Definition 3.1]

Lemma 3.1: Uniqueness of ϕ_d for surjective ϕ_e

Let $\mathfrak{C} = (\mathcal{X}, \mathcal{Y}, \phi_e, \phi_d, \Pi)$ be a $[\mathcal{L}, T]$-code. Let ϕ_e be surjective (onto). Then ϕ_d must be unique.

Proof

Note that if we prove this is true in $\mathcal{L}_\emptyset$, then it must be true in all structures. This because any $\mathcal{L}$-homomorphism is automatically a $\mathcal{L}_\emptyset$-homomorphism, so if the $\mathcal{L}_\emptyset$-homomorphism satisfying the constraint of ϕ_d is unique, any $\mathcal{L}$-homomorphism ϕ_d satisfying the constraint must also be unique.
Note that a $\mathcal{L}_\emptyset$ homomorphism is just any map, so we just need to show there is a unique map ϕ_d satisfying the constraint that if $y = \phi_e(x) \in \text{im}\,\phi_e$ for some $x \in X$, then $\phi_d(y) = \phi_d(\phi_e(x)) = [x]_{\ker \phi_e}$.
Because ϕ_e is onto, for every $y \in Y$ there is a corresponding $x \in X$ such that $y = \phi_e(x)$. Thus ϕ_d must map to $[x]_{\ker \phi_e}$, and so any two potential ϕ_d will have to map to the same thing, and so ϕ_d must be unique.

3.3 Structured Code Examples

Example 3.1: $[\mathcal{L}_\leq, T_{\leq n}]$-code

Recall $\mathcal{B}_{n,\leq}$ is the structure of n-length binary strings with an order relation [Example A.8]. We wish to construct a structured code that preserves order. To do this, we need to enforce that the code structure has domain that has finite size, (say less than n for simplicity) and to have the same structure as the message structure, i.e it should still be a list of fully ordered elements with no cycles.

Defining $T_{\leq n}$

First we want to find a theory $T_{\leq n}$ that says that any model of $T_{\leq n}$ is a collection of ordered numbers (at least up to isomorphism). The reason why we use numbers here instead of bitstrings is that we do not want to limit ourselves to models that can only have size some power of two (if we did, given n it would be easy to make an $\mathcal{L}_\emptyset$-sentence that says that any model must have size some power of two and have size less than 2^n).
Let $T_{\leq n}$ consist of the following $\mathcal{L}_\leq$ sentences:

1. $\forall_x x \leq x$
2. $\forall_{x,y,z}(x \leq y \wedge y \leq z) \to x \leq z$
3. $\forall_{x,y}(x \leq y \wedge y \leq x) \to x = y$

4. $\forall_{x,y} x \leq y \vee y \leq x$

These are the axioms of a totally ordered set. We also need some axioms that tell us that a model should have no more than n elements:

$$\text{Let } \psi_{\leq n} \text{be } \forall_{x_1,\ldots x_n,x_{n+1}} \left[\bigwedge_{i=1}^{n-1}(x_i \neq x_{i+1}) \to \bigwedge_{i=1}^{n} x_{n+1} = x_i\right] \tag{6}$$

This sentence says that if we have a collection of n distinct elements, then any $n+1$-th element will have to be in that collection. Thus any model satisfying $\psi_{\leq n}$will have size at most n. Thus

$$\mathcal{M} \models \psi_{\leq n} \iff |\operatorname{dom} \mathcal{M}| \leq n \tag{7}$$

5. $\psi_{\leq n}$, which is $\forall_{x_1,\ldots x_n,x_{n+1}} \left[\bigwedge_{i=1}^{n-1}(x_i \neq x_{i+1}) \to \bigwedge_{i=1}^{n} x_{n+1} = x_i\right]$

Note that it is possible for a structure to satisfy these axioms, but have an order which loops around, meaning that you could have some structure that can have $a \leq b \leq c \leq d \leq a \leq \ldots$. Thus if we enforce that there must be some minimum element, we should have a finite total ordered set with a minimum element, so it must be equivalent to some set of ordered numbers up to isomorphism.

6. $\exists_m \forall_x m \leq x$

Example $[\mathcal{L}_{\leq}, T_{\leq n}]$-source code (3:2 ordered binary code)

Let $\mathcal{X} = \mathcal{B}_{3,\leq}$, and $\mathcal{Y} = \mathcal{B}_{2,\leq}$. Note $\mathcal{X} \models T_{\leq 3}$, and $\mathcal{Y} \models T_{\leq 3}$, so they are $[\mathcal{L}_{\leq}, T_{\leq n}]$ structures.
Define $\phi_e : B_3 \to B_2$ that just drops the rightmost bit. So $\phi_e(000) = \phi_e(001) = 00$.
This induces $\mathcal{D} = \mathcal{B}_{3,\leq}/\ker(\phi_e)$ as the set of equivalence classes that map to the same thing under ϕ_e. It can be shown that ϕ_e is in fact a $\mathcal{L}_{\leq}$-homomorphism, because ignoring the last bit preserves the ordering relation. In this case, this implies that $\mathcal{D}$ has domain $\{\{000, 001\}, \{010, 011\}, \{100, 101\}, \{110, 111\}\}$. Technically, these sets will not look like $\{000, 001\}$, but will look like $\{(000, 001), (001, 000)\}$, but because $\ker \phi_e$ is an equivalence relation, it doesn't care about order, so from now on we will write $\{000, 001\}$ instead of $\{(000, 001), (001, 000)\}$.
Note that there is only one unique $\phi_d : \mathcal{Y} \to \mathcal{D}$ that can be made because ϕ_e is surjective due to [Lemma 3.1], and ϕ_d is the one mapping from $x \mapsto \phi_e^{-1}(\{x\})$, i.e. ϕ_d takes:

- $00 \mapsto_{\phi_d} \{000, 001\}$
- $01 \mapsto_{\phi_d} \{010, 011\}$
- $10 \mapsto_{\phi_d} \{100, 101\}$
- $11 \mapsto_{\phi_d} \{110, 111\}$

Now Π is any map that for each set above picks an element to map to. Ideally we would pick the element that has the higher probability, but sometimes due to time constraints we are not able to. So for instance let $\Pi : \mathcal{D} \to \mathcal{X}$ pick the element that has an even number of ones. Thus Π takes:

- $\{000, 001\} \mapsto_{\Pi} 000$
- $\{010, 011\} \mapsto_{\Pi} 011$
- $\{100, 101\} \mapsto_{\Pi} 101$
- $\{110, 111\} \mapsto_{\Pi} 110$

Summary

Note that the above construction gives us the encoding map (which is a $\mathcal{L}_{\leq}$-homomorphism) $\text{enc} : \mathcal{X} \to \mathcal{Y}$ which takes:

- $\{000, 001\} \ni x \mapsto_{\text{enc}} 00$

- $\{010, 011\} \ni x \mapsto_{\text{enc}} 01$
- $\{100, 101\} \ni x \mapsto_{\text{enc}} 10$
- $\{110, 111\} \ni x \mapsto_{\text{enc}} 11$

and a decoding map (also a $\mathcal{L}_{\leq}$-homomorphism) which takes:

- $00 \mapsto_{\text{dec}} 000$
- $01 \mapsto_{\text{dec}} 011$
- $10 \mapsto_{\text{dec}} 101$
- $11 \mapsto_{\text{dec}} 110$

Example 3.2: $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-code

First recall $\mathcal{L}_{\mathbb{Z}_2-vspace} = \{\vec{0}, +, -\} \cup \{\cdot_0, \cdot_1\}$ as defined in [Example A.4]. Let $\mathcal{Z}_2^n = (\mathbb{Z}_2^n, \vec{0}^{\mathbb{Z}_2^n}, \bar{\oplus}, \bar{\oplus}, \to \vec{0}^{\mathbb{Z}_2^n}, I)$ be the vector space of vectors in $\mathbb{Z}_2^n$, i.e n length vectors with the standard 0 vector, addition and subtraction as bitwise xor (vector addition mod 2), $\cdot_0$ as the function that just returns the 0 vector, and $\cdot_1$ as the identity operation. Let $T_{\mathbb{Z}_2 vspace}$ be the first order axiomatization of vector spaces ([12]). The important thing to note is that given any $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-structure $\mathcal{M}$, and $\mathcal{S} \leq \mathcal{M}$ will also be a $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-structure, because $T_{\mathbb{Z}_2 vspace}$ only contains universal sentences, so by [Lemma A.1] $\mathcal{M} \models T_{\mathbb{Z}_2 vspace} \implies \mathcal{S} \models T_{\mathbb{Z}_2 vspace}$, and so $\mathcal{S}$ is a $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-structure.

General $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-code

Note that a general $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-code will have the same idea as a linear code on the vector space $\mathcal{Z}_2^n$. Encoding is traditionally done via some matrix multiplication by matrix G called the generator matrix ($\mathcal{L}_{\mathbb{Z}_2-vspace}$-homomorphism is the same as a linear map between $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$ structures by [Example A.13]), and decoding is done by some parity check matrix H (corresponds to ϕ_d) yields a syndrome, and then given some syndrome with an output vector, and so you pick a vector in the original space to correspond to that syndrome and output vector. The only difference is that for linear codes, there are restrictions on what the parity check matrix can be, but in this case, the only restriction is that it is some matrix.

Example $[\mathcal{L}_{\mathbb{Z}_2-vspace}, T_{\mathbb{Z}_2 vspace}]$-code (Hamming 7-4 code)

Here we deal with $\mathcal{X} = \mathbb{Z}_2^4, \mathcal{Y} = \mathbb{Z}_2^7$. We use row multiplication for linear codes. First define the generator matrix G:

$$G := \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{8}$$

Let $\phi_e = \text{enc} : \mathcal{X} \to \mathcal{Y}$ be the encoder that takes $x \in \mathbb{Z}_2^4$ to $(x^T G)^T$. Note this is a linear operation, so it is a $\mathcal{L}_{\mathbb{Z}_2-vspace}$-homomorphism by [Example A.13]. Next let us define a parity check matrix H:

$$H := \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Then note decoding is done by first multiplying some given column vector y by H to get Hy, which we call the syndrome. Then, we apply the syndrome decoding on (y, Hy) where Hy tells us what error $e \in \mathbb{Z}_2^7$ has occurred, and then y maps to the inverse of the error, i.e $y \mapsto y \bar{\oplus} e$. Then Π can just pick whichever input has the highest probability.

Example 3.3: Homomorphic encryption

Homomorphic encryption is the study of cryptographic codes that preserve structure. Homomorphic encryption is just a structured code, where the only thing that changes is that there is some cryptographic performance measure says that it should be hard for an adversary to recover the message that was sent. Note that the encoding and decoding process are homomorphisms. For instance, start by fixing some language $\mathcal{L}$, with $\mathcal{L}$-theory T. If $\mathfrak{C} = (\mathcal{X}, \mathcal{Y}, \phi_e, \phi_d, \Pi)$ is a $[\mathcal{L}, T]$ code, then this allows for encryption in use for some operations. This means that if some person (call them Alice) wished to use someone else (call them Bob) to perform some computation that was evaluating some term $t^{\mathcal{X}}(\bar{a})$ without revealing secret variables $\bar{a}$, Alice could send the term (just symbols with variable symbols $x_1, \dots x_n$ in place of $a_1, \dots a_n$) "$t(\bar{x})$" to Bob and send $\text{enc}(a_1), \text{enc}(a_2), \dots \text{enc}(a_n)$ also to Bob. Then Bob can evaluate $t^{\mathcal{Y}}(\text{enc}(a_1), \dots \text{enc}(a_n))$, send this back to Alice. Then when Alice applies dec, because it is a $\mathcal{L}$-homomorphism:

$$\text{dec}\left[t^{\mathcal{Y}}((\text{enc}(a_1), \dots \text{enc}(a_n))\right] = t^{\mathcal{X}}(a_1, a_2, \dots a_n) = t^{\mathcal{X}}(\bar{a}) \tag{10}$$

(if we assume that $\text{dec} \circ \text{enc} = I_{\mathcal{X}}$, which is usually the case for cryptographic codes.)
Note in this case Alice can recover $t^{\mathcal{X}}(\bar{a})$, which is what she wanted to compute, without revealing $\bar{a}$, or even what $\mathcal{X}, \text{enc}, \text{dec}$ are either. This means that Bob does not even need to know what code Alice is using if all she wanted to do was use him for computation.

4 Conclusion

In this report, we have generalized the notion of a code using model theory, which allows us to get some intuition about generalized algebraic objects. Structured coding has a few implications, one of note is homomorphic encryption, mentioned in [Example 3.3]. We mentioned that this allows for encryption in use, meaning that operations can be performed on the message without actually revealing the message. This is incredibly useful. Take for instance the problem of running a program with a cloud computing provider, such as Amazon Web Services (AWS). If you run a computation on one of their servers, even if the communication between AWS and your computer is secure, if someone had access to the memory on AWS's servers, they could see what the computation is doing. However if your computation had some sort of homomorphic encryption, you could just send the cyphertext to AWS, and they would apply some operation, and because you are using a structured code, when they send back the cyphertext you can decode it, and because encoding and decoding is a homomorphism, if the operation that you want to perform can be done with successive symbols from the language, then the operation will be preserved through encoding and decoding. Thus you could safely rely on AWS's computing resources without revealing your message.

The point of this example (and the paper in general) is that while studying concrete applications of algebra can give you some interesting ideas, if we generalize, then we can bring together many different topics and find connections between seemingly different topics.

Finally, we end with some thoughts on possible directions for future work:

- Formalize the notion of a probabilistic homomorphism, i.e one that preserves structure with high probability (perhaps maybe asymptotic?) and then study how good this is for coding. This might need us to extend the notion of probabilistic finite model theory, so that we can find probability of terms, formulas, etc. Then prove a similar result
- How to create randomly constructed codes (without a fixed code structure). (for $\mathbb{Z}_2^n$ you have a random matrix, for $\mathcal{B}_{n,\leq}$ you need some sort of procedure to create a random mapping that preserves order.) Maybe we can get a general procedure? might be tough though.
- We would like to measure the performance of different structured codes using some performance measures that we found in the general coding section. One example is that the code of ordered binary strings has some ordered binary operation, so this allows for ϕ_d and Π to perform a binary search, which could result in a significant speedup.
- Try more nonstandard structured codes, like a graph code.

A Model Theory

Here we introduce the standard notations and definitions of model theory. We also give some basic results.

A.1 Summary

In this paper we wish to study some topics in the intersection of probability and algebra, and we wish to generalize these ideas. We can do this through the lens of model theory, which is just universal algebra with logic. In this section, we cover basic definitions and results that we will use from model theory. Everything in model theory is based on a list of about 10 or so definitions, so we need to cover a bit of definitions and basic results in this section. It is pretty abstract, but hopefully the examples should help give a more concrete intuition.

If you come from a OOP (object oriented programming) background, model theory has a nice interpretation. You can look at model theory as the study of a specific types of classes in some object oriented programming language. In this section, we also give the corresponding OOP interpretation of what these concepts mean.

Even if you are familiar with model theory, note that we also define some additional perhaps non-standard items in this section that we may use later. These are: $\mathcal{L}_\emptyset, \mathcal{L}_\leq, \mathcal{L}_{grp}, \mathcal{L}_{ring}, \mathcal{L}_{oring}, \mathcal{B}_n, \mathcal{B}_{n,\leq}$, $[\mathcal{L}, T]-$structures, and the isomorphism theorem.

A.2 Languages

Definition A.1: Language

A langauge $\mathcal{L} = \mathcal{L}_f \cup \mathcal{L}_R \cup \mathcal{L}_c$ is a set consisting of symbols that can be:

- Constant symbols $\in \mathcal{L}_c$
- Function symbols $\in \mathcal{L}_f$ of arity $n \in \mathbb{Z}_+$
- Relation symbols $\in \mathcal{L}_R$ of arity $n \in \mathbb{Z}_+$

(Recall that arity is just the number of arguments)

Note that these are just symbols, and no more. By themselves they do not have any meaning, other than restricting the amount of arguments a function/relation can have.

Example A.1: $\mathcal{L}_\emptyset = \{\}$

The simplest example of a language is $\mathcal{L} = \{\} = \emptyset$. We call this language $\mathcal{L}_\emptyset$, the language of pure sets. This will represent objects with no additional structure.

Example A.2: $\mathcal{L}_\leq = \{\leq\}$

Let $\mathcal{L} = \{\leq\}$ be the order language, where $\leq$ is a relation of arity 2. In this language the symbol $\leq$ generally represents some notion of less than or equal to, but it does not have to. We call this language $\mathcal{L}_\leq$.

Example A.3: $\mathcal{L}_{grp}$

Let $\mathcal{L}_{grp} = \{e, \cdot, ^{-1}\}$ be the language of groups, where e is a constant symbol, $\cdot$ is a function symbol with arity 2, and $^{-1}()$ is a function symbol with arity 1. Generally we tend to think of e as representing some sort of identity element, $\cdot(,)$ as representing some sort of group operation, and $^{-1}()$ as representing the act of taking the inverse of an element.

Example A.4: $\mathcal{L}_{\mathbb{F}-vspace}$

Vector spaces are a little more complicated. Note that usually we think of vector spaces as being a set of vectors paired with a base field. As we will see later, it makes sense to fix the field (call it $\mathbb{F}$), and define the language of $\mathbb{F}$-vector spaces as: $\mathcal{L}_{\mathbb{F}-vspace} = \{\vec{0}, +, -\} \cup \{\cdot_r : r \in \mathbb{F}\}$, where $\vec{0}$ is a constant symbol, $+, -$ are function symbols of arity 2, and we have a $\cdot_r$ for every field element (possibly infinitely many

of these) that is a unary function symbol. This is meant to represent scalar multiplication by some field element, but it does not necessarily have to.

Example A.5: $\mathcal{L}_{ring}$, $\mathcal{L}_{oring}$

Let $\mathcal{L}_{ring} = \{0, 1, +, -, *\}$ be the language of rings, and $\mathcal{L}_{oring} = \mathcal{L}_{ring} \cup \{\leq\}$ be the lanugage of ordered rings, where $0, 1$ are constant symbols, $+, -, *$ are function symbols with arity 2, and $\leq$ is a relation symbol with arity 2.

Analogy A.1: Languages in OOP

For model theory, think of the primitive types as constants, functions and relations. When you define a language, it is similar to making an interface that defines all the types that you will later implement in some class. In other words when you define a language, you are fixing some sort of symbols to be implemented later, with the restriction that all symbols will either be a function symbol, relation symbol, or a constant symbol.
Consider $\mathcal{L}_{oring}$ as defined in [Example A.5]. The corresponding OOP idea would be the following interface:

```
Interface L_oring:
    Constant zero
    Constant one
    Function plus(arg1, arg2)
    Function minus(arg1, arg2)
    Function mult(arg1, arg2)
    Relation lessthan(arg1, arg2)
```

(an interface is just a schema that forces you to implement everything in it with valid types)

A.2.1 $\mathcal{L}$-terms

In model theory, we need to reference finite combinations of symbols from the language. This can be done by $\mathcal{L}$-terms.

Definition A.2: $\mathcal{L}$-terms

The set of $\mathcal{L}$-terms are defined inductively:

- Every constant symbol $c \in \mathcal{L}_c$ is a $\mathcal{L}$-term
- Every variable symbol v_i is a $\mathcal{L}$-term
- If $t_1, \dots t_{n_f}$ are $\mathcal{L}$-terms and $f \in \mathcal{L}_f$ is a function symbol, then $f(t_1, \dots t_{n_f})$ is a $\mathcal{L}$-term

---

Think of the set of $\mathcal{L}$ terms as the set of valid strings built from symbols in the language, i.e. it is a string that would "compile". So if you had a unary function and you tried to put in two arguments, it would not be a valid term.

Example A.6: $\mathcal{L}$-Terms in $\mathcal{L}_{\leq}$, $\mathcal{L}_{grp}$

Consider $\mathcal{L}_{\leq}$. The language does not contain function or constant symbols, so the only terms are variable symbols, such as "x", or something like "y".
Consider $\mathcal{L}_{grp}$. All of the following are valid terms:

- "x": variable symbol
- "e": constant symbol
- "$\cdot(x, e)$": For convenience we write this as "$x \cdot e$".
- "$(x \cdot e) \cdot x$"

Something that is not a term could be:

- $\cdot(x)$

Sometimes we put quotes around terms, but other times we just leave them as is. The quotes are to emphasize that these are just symbols, not evaluations. This leads to the interpretations of terms, given below.

Analogy A.2: Terms in OOP

Terms in the OOP analogy are likewise valid compositions of symbols. The only thing that is enforced is that the string has valid types. You can think of this as saying that it will compile, so for L_oring in [Analogy A.1] you would have

```
plus(x,one)
```

compile (if x is a variable symbol), and so be a valid term, but something like

```
plus(one)
```

would not compile, so it is not a valid term.

A.2.2 $\mathcal{L}$-Structures

Definition A.3: $\mathcal{L}$-Structure

An $\mathcal{L}$-structure $\mathcal{M}$ consists of:

- A nonempty set M called the domain/universe
- For each $f \in \mathcal{L}_{\mathcal{F}}$ of arity n_f, a function $f^{\mathcal{M}} : M^{n_f} \to M$
- For each $R \in \mathcal{L}_{\mathcal{R}}$ of arity n_R, a relation $R^{\mathcal{M}} \subseteq M^{n_R}$
- For each $c \in \mathcal{L}_{\mathcal{C}}$, an element $c^{\mathcal{M}} \in M$

We call such $f^{\mathcal{M}}, R^{\mathcal{M}}, c^{\mathcal{M}}$ interpretations of symbols f, R, c in the structure $\mathcal{M}$. By convention, we use a calligraphic font for structures. If a calligraphic letter say $\mathcal{H}$ is given as a structure, by convention the domain is considered to be the corresponding capital letter, in this case $H := \text{dom}(\mathcal{H})$.

Example A.7: $\mathcal{L}_\emptyset$-structure

Consider $\mathcal{L}_\emptyset$. Then a $\mathcal{L}$-structure is simply a set. So if we have some set A, then we can say $\mathcal{A} = (A)$ is an $\mathcal{L}_\emptyset$ structure.

Example A.8: $\mathcal{B}_n, \mathcal{B}_{n,\leq}$, $\mathcal{L}_\leq$-structure

Let $B_n = \{0,1\}^n$, and $\leq^{\mathcal{B}_n}$ be the relation that evaluates whether the bitstring $x \leq^{\mathcal{B}_n} y$ when you interpret x, y as the n bit binary representation of a number. Then we can define $\mathcal{L}_\emptyset$-structure $\mathcal{B}_n = (B_n)$, and $\mathcal{L}_\leq$-structure $\mathcal{B}_{n,\leq} = (B_n, \leq^{\mathcal{B}_n})$.

Example A.9: Atypical $\mathcal{L}_\leq$-structure

Let the set $A = \{\text{Cat}, \text{Dog}, \ldots\}$ be the set of all animals. Define $\leq^{\mathcal{A}}$ as the relation where $a \leq^{\mathcal{A}} b$ means both animals a and b are Mammals. Then $\mathcal{A} = (A, \leq_A)$ is a valid $\mathcal{L}_\leq$ structure, (because A is a set and $\leq^{\mathcal{A}}$ is a relation on A with arity 2) even though $\leq^{\mathcal{A}}$ has nothing to do with the traditional interpretation of $\leq$.

Analogy A.3: $\mathcal{L}$-structures in OOP

Structures in the OOP analogy are classes that implement the language interface. For L_oring in [Analogy A.1] you would have a corresponding $\mathcal{L}_{oring}$-structure class as:

```
class L_oring_structure implements L_oring:
    Set domain
    Constant zero in domain
    Constant one in domain
    Function plus(arg1 in domain, arg2 in domain) -> in domain
    Function minus(arg1 in domain, arg2 in domain) -> in domain
    Function mult(arg1 in domain, arg2 in domain) -> in domain
    Relation lessthan(arg1 in domain, arg2 in domain) -> bool

    Constructor(dom_in, zero_in, one_in, plus_in, minus_in, mult_in, lessthan_in):
        this.domain = dom_in
        this.zero = zero_in
        this.one = one_in
        this.plus = plus_in
        this.minus = minus_in
        this.lessthan = lessthan_in
```

The important thing to note is that any intance of L_oring_structure is just some class that has implementations of the constants, functions, and relations listed above, acting in some specified domain. It does not necessarily have to be an ordered ring, just as in [Example A.9] where the $\leq$ just was a relation on two animals, and had nothing to do with the idea of less than. As long as your structure has this type schema, (i.e has all the types from L_Oring as well as some domain that is a set), then it will correspond to a valid $\mathcal{L}_{oring}$-structure. So when we define the $\mathcal{L}_{oring}$-structure $\mathcal{Z}_2 = (\{0,1\}, 0, 1, \text{xor}(,), \text{xor}(,), \leq)$, we essentially are just calling $\mathcal{M}$ =L_oring_structure.Constructor$(\{0,1\}, 0, 1, \text{xor}(,), \text{xor}(,), \leq)$.

A.2.3 $\mathcal{L}$-Term interpretations

Above we defined the concept of a $\mathcal{L}$-structure, and we saw that we had a interpretation for every symbol in the language. Recall that we can also combine finite symbols in the language (and variable symbols) to produce terms. So now we must define a similar concept for terms, namely the interpretation of $\mathcal{L}$-terms.

Definition A.4: Interpretations of $\mathcal{L}$-terms

Given $\mathcal{L}$ structure $\mathcal{M}$ where t is a term involving variables $v_1, \dots v_n$, t has interpretation $t^{\mathcal{M}}$ in $\mathcal{M}$ as a function $M^n \to M$, defined as:

- If t is a constant symbol c, then $t^{\mathcal{M}}(\bar{a}) = c^{\mathcal{M}}$
- If t is a variable v_j, then $t^{\mathcal{M}}(\bar{a}) = a_j$
- If t is $f[t_1, \dots t_{n_f}]$, then $t^{\mathcal{M}}(\bar{a}) = f^{\mathcal{M}}[t_1^{\mathcal{M}}(\bar{a}), \dots t_{n_f}^{\mathcal{M}}(\bar{a})]$

Think of these as generalizations of all the elements of the domain M that you can get from only referencing the language:

- Constant symbols are interpreted as elements of M
- Variables should take values in M
- f has codomain M, so evaluating $f(t_1, \dots t_{n_f})$ should give something $\in M$

You could also think of these as generalized polynomials:

Example A.10: Interpretation of terms in $\mathbb{R}$

Let $\mathcal{R} = (\mathbb{R}, +^{\mathbb{R}}, -^{\mathbb{R}}, 0^{\mathbb{R}}, 1^{\mathbb{R}})$ be the $\mathcal{L}_{ring}$-structure corresponding to the real numbers as a ring.

- Let t_1 be "0". $t_1^{\mathcal{R}} : \emptyset \to \mathbb{R}$ is constant symbol "0" acting on arguments $\bar{a}$ so $t_1^{\mathcal{R}}(\bar{a}) = 0^{\mathcal{R}}(\bar{a}) = 0^{\mathcal{R}}$
- Let t_2 be "+(0,0)". $t_2^{\mathcal{R}} : \emptyset \to \mathbb{R}$ is "+(t_1, t_1)" (domain is $\emptyset$ because t_1 has no variable symbols) which becomes $+^{\mathbb{R}}(0^{\mathbb{R}}, 0^{\mathbb{R}})$
- Let t_3 be "$-(v_j, +(t_1, t_1))$". $t_3^{\mathcal{R}}(\bar{a}) : \mathbb{R} \to \mathbb{R}$ is "$-(v_j, t_2)$", and this is the same as $-^{\mathbb{R}}(a_j, +^{\mathbb{R}}(0^{\mathbb{R}}, 0^{\mathbb{R}})$

So for rings, we can think of the set of terms as just being the set of polynomials with arbitrary variable symbols.

Example A.11: Interpretation of terms in $\mathcal{L}_{\text{group}}$-structure

Let $\mathcal{Z} = (\mathbb{Z}, 0^{\mathbb{Z}}, +^{\mathbb{Z}})$ be the $\mathcal{L}_{\text{group}} = \{e, \cdot\}$ structure corresponding to the integers with $0^{\mathbb{Z}}$ as the identity and $+^{\mathbb{Z}}$ as the group operation. The following are valid interpretations of terms in $\mathcal{Z}$:

- Let t_1 be "e". $t_1^{\mathcal{Z}}(\bar{a}) : \emptyset \to \mathbb{R}$ is the constant symbol corresponding to $0^{\mathbb{Z}}$
- Let t_2 be "$e + v_j$". $t_2^{\mathcal{Z}}(\bar{a}) : \mathbb{R} \to \mathbb{R}$ is the function that adds one to whatever you plug in.

We can see that terms are just generalized polynomials. They are functions that can contain multiple variables, constants, and functions applied to these, and then evaluate what that result is in different $\mathcal{L}$ structures. Think of a term $t^{\mathcal{M}}(\bar{a})$ as the evaluation of the generalized polynomial t in the context of the structure $\mathcal{M}$.

Note that terms do not contain relation symbols. Terms will always evaluate to something in the domain.

Analogy A.4: Interpretation of terms

The interpretation of terms can be thought of as just what the compiler evaluates terms that a programmer might write down as. So if we had the term t be "$1 + 0$" that would be have interpretation equivalent to:

```
M = new L_oring_structure(...) # suppose M is given instance of the
L_oring_structure class.
t = M.plus(M.one, M.zero) # interpretation of "1 + 0" in structure M.

M.domain.contains(t) == true #always evaulates to true.
```

A.2.4 $\mathcal{L}$-Formulas

Up until this point, we could call what we have done as just universal algebra. We have only abstracted away the concept of structure. Note however we have left out relation symbols from terms, and this is because we have no logic built in yet. To fix this, we now define the notion of a formula, and we finally see where universal algebra splits off from model theory.

Definition A.5: $\mathcal{L}$-Formulas

The set of $\mathcal{L}$ formulas are defined inductively:

1. If s and t are terms, then $s = t$ is a formula
2. If $R \in \mathcal{L}_{\mathcal{R}}$ has arity n_R and $t_1, \dots t_{n_R}$ are terms, then $R(t_1, \dots t_{n_R})$ is a formula
3. If φ is a formula, then so is $\neg\varphi$
4. If φ and ψ are formulas, then so are $\varphi \wedge \psi$ and $\psi \vee \varphi$
5. If φ is a formula, then so are $\exists_v \varphi$ and $\forall_v \varphi$

If a formula is built only with 1 and 2, then it is called an atomic formula. If a formula is built without 5, then it is called a quantifier free formula.
One thing to note is that $=$ is a relation, but is not in our language. This is because we use equality in almost any structure, so we assume that the $=$ relation exists, and has the interpretation of just saying if two things are equal.
Another thing to note is that we sometimes use $\to$ (implication) and $\leftrightarrow$ (equivalence) in formulas. This can be done because implies and equivalence can be built from and, or and negation.

Example A.12: $\mathcal{L}_{grp}, \mathcal{L}_{\leq}$ formulas

Recall that $\mathcal{L}_{grp} = \{\cdot, e\}$. Then the following are valid $\mathcal{L}_{grp}$ formulas $\phi(a)$:

- $\forall_x (e \cdot x = x) \vee (e = e)$
- $a * a = e$

If we consider $\mathcal{L}_{\leq}$ $(= \{\leq\})$, some possible formulas $\phi(a, b)$ could be:

- $a < b$
- $\exists_z z \leq a \wedge z = b$
- $\neg(a < b)$

You can think of a formula as something similar to a term. Terms eventually output something in the domain, while formulas should "output" either true or false. So if you see a term in a formula, it must be in some relation in the language, or it must be in some equality relation. Thus we can see that formulas can consist of quantified and logical statements regarding if two generalized polynomials are equal/have some other relation. So first order logic only has the power to say a quantified logical statement about if two generalized polynomials in the language are equal or have some other specified relation.

Analogy A.5: Formulas in OOP

Formulas are simply any function that is built from checking if terms are equal, or have some relation, with logical operations. So for instance given the interface L_oring in [Analogy A.1], some formulas could be:

```
define phi_1(input_1, input_2) -> bool{
    return input_1 == input2
}
define phi_2(input_1) -> bool{
    return  L_oring.lessthan(L_oring.zero, L_oring.one) and phi_1(input1)
}
```

Now we define some more basic definitions relating to formulas.

Definition A.6: $\mathcal{L}$-Sentence

A $\mathcal{L}$ sentence is an $\mathcal{L}$-formula without any free variables. So each variable should be bound by (i.e appear as the parameter of) some quantifier in the variable's scope.

Definition A.7: $\mathcal{L}$-Theory

A set of $\mathcal{L}$-sentences T is called an $\mathcal{L}$-theory.

A.2.5 Truth

Note that the notion of truth is sensitive to your model. Some $\mathcal{L}$-formula could be true in one model, but be false in another. So we must first define what truth is.

Definition A.8: Truth

Let $\mathcal{M}$ be an $\mathcal{L}$ structure, and $\varphi(x_1, \dots x_n)$ be an $\mathcal{L}$ formula, and let $a_1, \dots a_n \in M$. We define $\mathcal{M} \models \varphi(a_1, \dots a_n)$ by induction on formulas:

1. Case φ is $s = t$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff $s^{\mathcal{M}} = t^{\mathcal{M}}$
2. Case φ is $R(t_1, \dots t_n)$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff $(t_1^{\mathcal{M}}(\bar{a}), \dots t_n^{\mathcal{M}}(\bar{a})) \in R^{\mathcal{M}}$
3. Case φ is $\neg\psi$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff $\mathcal{M} \not\models \psi(\bar{a})$
4. Case φ is $\psi_1 \wedge \psi_2$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff $\mathcal{M} \models \psi_1(\bar{a})$ and $\mathcal{M} \models \psi_2(\bar{a})$
5. Case φ is $\psi_1 \vee \psi_2$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff $\mathcal{M} \models \psi_1(\bar{a})$ or $\mathcal{M} \models \phi_2(\bar{a})$
6. Case φ is $\exists_y \psi(\bar{x}, y)$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff there is a $b \in M$ such that $\mathcal{M} \models \psi(\bar{a}, b)$

7. Case φ is $\forall_y \psi(\bar{x}, y)$:
   Say $\mathcal{M} \models \varphi(\bar{a})$ iff for every $b \in M$, $\mathcal{M} \models \psi(\bar{a}, b)$

If $\mathcal{M} \models \varphi(\bar{a})$, then we say $\mathcal{M}$ satisfies $\varphi(\bar{a})$, or that $\varphi(\bar{a})$ is true in $\mathcal{M}$.

Definition A.9: Truth of a theory

Given some $\mathcal{L}$-theory T, (set of $\mathcal{L}$-sentences) we say $\mathcal{M} \models T$ iff $\mathcal{M} \models \phi$ for every ϕ in T.

Definition A.10: Theory of a model

Let $\mathcal{M}$ be some $\mathcal{L}$-structure. Define

$$\text{Th}(\mathcal{M}) = \{\mathcal{L}\text{-sentences } \phi : \mathcal{M} \models \phi\} \tag{11}$$

So $\text{Th}(\mathcal{M})$ is the set of all sentences that are true in a model.

A.2.6 Algebraic Structure

Definition A.11: Algebraic Structure

Given some language $\mathcal{L}$, suppose we have $\mathcal{L}$ structure $\mathcal{M}$ with a set of $\mathcal{L}$-sentences T (sometimes called the axioms of T). We say $\mathcal{M}$ is a $[\mathcal{L}, T]$-structure if $\mathcal{M}$ is an $\mathcal{L}$-structure and $\mathcal{M} \models T$.

A.2.7 $\mathcal{L}$-Homomorphisms, Embeddings, Isomorphisms

Now we consider maps between different $\mathcal{L}$-structures. We like to focus on maps that preserve some sort of structure, namely the structure that we specify in the language.

Definition A.12: $\mathcal{L}$-homomorphism

A $\mathcal{L}$-Homomorphism η is a function between two $\mathcal{L}$-structures $\eta : \mathcal{M} \to \mathcal{N}$ such that η preserves all interpretations of symbols in $\mathcal{L}$. Formally:

- For each function symbol f of arity n, and each $a_1, \dots a_n \in M$: $\eta(f^{\mathcal{M}}(a_1, \dots a_n)) = f^{\mathcal{N}}(\eta(a_1), \dots \eta(a_n))$
- For each relation symbol R of arity n, and each $a_1, \dots a_n \in M$: $(a_1, \dots a_n) \in R^{\mathcal{M}} \implies (\eta(a_1), \dots \eta(a_n)) \in R^{\mathcal{N}}$
- For each constant symbol c : $c^{\mathcal{M}} = c^{\mathcal{N}}$

Definition A.13: $\mathcal{L}$-embedding, $\mathcal{L}$-isomorphism

A $\mathcal{L}$-embedding η is a one to one $\mathcal{L}$-homomorphism, with the additional constraint that the implication for relation symbols goes both ways. In other words:

1. For each relation symbol R of arity n, and each $a_1, \dots a_n \in M$: $(a_1, \dots a_n) \in R^{\mathcal{M}} \iff (\eta(a_1), \dots \eta(a_n)) \in R^{\mathcal{N}}$

A $\mathcal{L}$-isomorphism is a onto $\mathcal{L}$-embedding.

Example A.13: Linear maps in vector spaces are the same as homorphisms

$\mathcal{L}_{\mathbb{F}-vspace}$-structure $\mathcal{V}$ is called a vector space if it satisfies a list of first order axioms[12].
Given two vector spaces $\mathcal{V} = (V, +^V, -^V, \vec{0}^V, \cdot_r^V : r \in \mathbb{F})$ and $\mathcal{W} = (W, +^W, -^W, \vec{0}^W, \cdot_r^W : r \in \mathbb{F})$ recall a $\mathcal{L}_{\mathbb{F}-vspace}$-homomorphism $f : V \to W$ is any function that preserves interpretations of function symbols and constants. More explicitly:

1. f preserves $+$:
   $f(v_1 +^V v_2) = f(v_1) +^W f(v_2)$
   $f(+^V(v_1, v_2)) = +^W(f(v_1), f(v_2))$
2. f preserves $-$:
   $f(v_1 -^V v_2) = f(v_1) -^W f(v_2)$
   $f(-^V(v_1, v_2)) = -^W(f(v_1), f(v_2))$
3. f preserves $\vec{0}$:
   $f(\vec{0}^V) = \vec{0}^W$
4. f preserves $\cdot_r$ for every $r \in \mathbb{F}$
   $f(r * v_1) = r * f(v_1)$
   $f(\cdot_r^V(v_1)) = \cdot_r^W(f(v_1))$

Recall that f is said to be linear if the following properties hold:

1. f preserves $+$:
   $f(v_1 +^V v_2) = f(v_1) +^W f(v_2)$
   $f(+^V(v_1, v_2)) = +^W(f(v_1), f(v_2))$
2. f preserves $\cdot_r$ for every $r \in \mathbb{F}$
   $f(r * v_1) = r * f(v_1)$
   $f(\cdot_r^V(v_1)) = \cdot_r^W(f(v_1))$

Note that f being a $\mathcal{L}_{\mathbb{F}-vspace}$-homomorphism implies that it is linear, because f being linear is just points (1) and (4). The tricky part is showing the reverse direction, namely that f being linear means f is a $\mathcal{L}_{\mathbb{F}-vspace}$-homomorphism.
Using the axioms we can simplify the conditions on f to say that f must be linear if the following hold:

Recall one of the axioms of vector spaces states for any vector a, $\cdot_{0^{\mathbb{F}}}(a) = \vec{0}$. So consider an f that preserves $\cdot_r$ for every $r \in \mathbb{F}$. Since the number $0^{\mathbb{F}} \in \mathbb{F}$, it must be that for any $v \in V$: $\cdot_{0^{\mathbb{F}}}^V(v) = \vec{0}$. This means:
$f(\vec{0}^V) = f(\cdot_{0^{\mathbb{F}}}^V(v)) = \cdot_{0^{\mathbb{F}}}^W(f(v))$. Since $f(v) \in W$, we can apply the axiom again to note that this must $= \vec{0}^W$. Thus we have preserved $\vec{0}$. This means that if we preserve scalar multiplication, we do not need to preserve the zero vector, as this is guarenteed by the axioms. A similar argument can show the other properties.

A.2.8 Substructures

Note that on occasion, we will have an $\mathcal{L}$-structure inside of a larger $\mathcal{L}$-structure. This is called a substructure.

Definition A.14: Substructure

If $\mathcal{M}, \mathcal{N}$ are $\mathcal{L}$ structures with $M \subseteq N$ and the inclusion map $1_M : M \to N$ (the identity on M) is an $\mathcal{L}$-embedding, then $\mathcal{M}$ is a substructure of $\mathcal{N}$. (sometimes written as either $\mathcal{M} \leq \mathcal{N}$ or simply $\mathcal{M} \subseteq \mathcal{N}$)

Example A.14: Substructures of Groups

Note that the notion of substructure is sensitive to the language $\mathcal{L}$. So for instance if $\mathcal{L}_{\text{group}} = \{e, \cdot\}$, then a substructure of a group does not preserve inverses and therefore is not a group, however if $\mathcal{L}_{\text{group}} = \{e, \cdot, \cdot^{-1}\}$ then a substructure of a group must preserve inverses and therefore would be a group.

Lemma A.1: Substructures preserve universal formulas

Let $\mathcal{S} \leq \mathcal{B}$, and let $\phi(\bar{x})$ be a universal formula. Then for any $\bar{s} = s_1, \ldots s_n \in S$: $\mathcal{B} \models \phi(\bar{s}) \implies \mathcal{S} \models \phi(\bar{s})$

Proof

This fact is somewhat intuitive, if $\phi(\bar{a})$ is a universal formula, then if it is true in the larger structure, it must be that $\phi(\bar{a})$ is true in the smaller structure, because $\phi(\bar{a})$ is $\forall_{\bar{x}}\psi(\bar{x},\bar{a})$ for some x, and so if $\psi(\bar{x},\bar{s})$ is true when we plug in any elements $x_1, x_2, \ldots x_n \in B$, it must be that if $x_1, x_2, \ldots \in S$ that $x_1, x_2, \ldots \in B$, so if it is true in B, it must be true in S. There is a formal argument using induction on the set of terms, but it is too long to give this here.

A.2.9 Kernel

Definition A.15: Kernel of $\mathcal{L}$-homomorphism

Let f be a $\mathcal{L}$-homomorphism. Then define

$$\ker f = \{(x,y) : f(x) = f(y)\} \tag{12}$$

Example A.15: Kernel for linear map

Let $f : \mathbb{C}^3 \to \mathbb{C}^2$ be the map corresponding to matrix multiplication:

$$f : \vec{v} \mapsto \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \vec{v} \tag{13}$$

Note that f is a homomorphism as it is linear, and note that

$$\ker f = \left[\text{span}\left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\right)\right]^2 \tag{14}$$

A.2.10 Quotient Structures

In this paper we will only quotient with the kernel, so we do not need to define the general quotient structure.

Definition A.16: Quotient with $\ker f$

Let $\mathcal{A}, \mathcal{B}$ be two $\mathcal{L}$-structures with $\mathcal{L}$-homomorphism $f : A \to B$. We define $\mathcal{A}/\ker f$ as a new $\mathcal{L}$-structure with domain as the equivalence classes of $\ker f$. More formally, this means that:

$$\text{dom}[\mathcal{A}/\ker f] = \{[a]_{\ker f} : a \in A\} := \{C : \exists_{x \in A} C = \{y : (x,y) \in \ker f\}\} \tag{15}$$

You can think of this as saying that $C \in \text{dom}[\mathcal{A}/\ker f]$ if and only if C is a set such that anything in C maps to the same thing through f.

Intuitively, we think of this quotient as not caring about the difference between inputs unless they result to a different output.

A.2.11 Isomorphism theorem

This theorem will be useful when we define the notion of a structured code in [Definition 3.1].

Theorem A.1: Isomorphism theorem

Let $\phi : A \to B$ be a $\mathcal{L}$-homomorphism between two $\mathcal{L}$-structures $\mathcal{A}, \mathcal{B}$. Then $\text{im}(\phi)$ is a substructure of $\mathcal{B}$, then $\mathcal{A}/\ker\phi$ is isomorphic to $\text{im}\,\phi$.

The intuition for this theorem is somewhat clear. $\mathcal{A}/\ker\phi$ treates two items that map to the same thing through ϕ as the same. This induces an invertible map $\hat{\phi} : \mathrm{dom}[\mathcal{A}/\ker f] \leftrightarrow \mathrm{im}\, B$ that outputs whatever ϕ would, and so is invertible, because given some element $\phi(a) \in \mathrm{im}\, B$, we can map it back to $[a]_{\ker\phi}$, the equivalence class of a under $\ker\phi$. $\hat{\phi}$ is a isomorphism because ϕ preserves structure, so $\hat{\phi}$ must as well.

B Bibliography

References

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[2] Nikolay Bazhenov, Ekaterina Fokina, and Luca San Mauro. Learning families of algebraic structures from informant. *Information and Computation*, 275:104590, 2020.

[3] Nikolay Bazhenov and Luca San Mauro. On the turing complexity of learning finite families of algebraic structures. *Journal of Logic and Computation*, 31(7):1891–1900, 2021.

[4] Justin Bledin. An even shorter model theory. 2007.

[5] Ronald Fagin. Finite-model theory - a personal perspective. *Theoretical Computer Science*, 116(1):3–31, 1993.

[6] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007:1–10, 2007.

[7] Ziyuan Gao, Frank Stephan, Guohua Wu, and Akihiro Yamamoto. Learning families of closed sets in matroids. In *International Conference on Teaching and Computational Science*, pages 120–139. Springer, 2012.

[8] E Mark Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[9] Valentina S Harizanov and Frank Stephan. On the learnability of vector spaces. *Journal of Computer and System Sciences*, 73(1):109–122, 2007.

[10] Sanjay Jain and Frank Stephan. Learning by switching type of information. *Information and Computation*, 185(1):89–104, 2003.

[11] Neil D. Jones and Alan L. Selman. Turing machines and the spectra of first-order formulas. *Journal of Symbolic Logic*, 39(1):139–150, 1974.

[12] Richard Kaye. Vector spaces, Sep 2015.

[13] Steffen Lange, Thomas Zeugmann, and Sandra Zilles. Learning indexed families of recursive languages from positive data: A survey. *Theoretical Computer Science*, 397(1-3):194–232, 2008.

[14] Wentian Li. On the relationship between complexity and entropy for markov chains and regular languages. *Complex systems*, 5(4):381–399, 1991.

[15] Wafik Boulos Lotfallah. Strong 0-1 laws in finite model theory. *The Journal of Symbolic Logic*, 65(4):1686–1704, 2000.

[16] Wolfgang Merkle and Frank Stephan. Trees and learning. *Journal of Computer and System Sciences*, 68(1):134–156, 2004.

[17] Robert I Soare. *Turing computability: Theory and applications*. Springer, 2016.

[18] Frank Stephan and Yuri Ventsov. Learning algebraic structures from text. *Theoretical Computer Science*, 268(2):221–273, 2001.

[19] Thomas Zeugmann and Sandra Zilles. Learning recursive functions: A survey. *Theoretical Computer Science*, 397(1-3):4–56, 2008.