Computation with Infinite Programs

MSc Thesis *(Afstudeerscriptie)*

written by

Ethan S. Lewis
(born January 6, 1993 in Newark, Delaware USA)

under the supervision of **Prof. Benedikt Löwe** and **Lorenzo Galeotti MSc**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
June 28, 2018	Dr. Merlin Carl
	Dr. Tejaswini Deoskar (chair)
	Lorenzo Galeotti MSc (supervisor)
	Prof. Benedikt Löwe (supervisor)
	Dr. Benjamin Rin
	Dr. Piet Rodenburg

Institute for Logic, Language and Computation

Abstract

Koepke introduced a machine model of computation that uses infinite time and space. In our thesis, we generalize Koepke's model by allowing for infinite programs in addition to infinite time and space. With this new model of computation, we prove generalizations of basic results from finite computability theory. Furthermore, we introduce the axiom of computable enumerability and show that it is independent of Gödel-Bernays class theory with the axiom of global choice. Assuming this axiom, we prove results that characterize the computably enumerable and decidable classes, as well as the computable functions.

Contents

Chapter 1

Introduction

In Turing's seminal paper [18], he introduced an abstract model of computation, which we now refer to as a Turing machine. This machine executes a finite program in a finite amount of time using a finite amount of tape. Eventually, Turing machines became the accepted standard for finite computability theory. Since then, there have been attempts to generalize computability theory to an infinite setting. One of the earliest known examples is α-recursion theory, which is detailed by Sacks in [16]. This generalization, however, is based on definability using formulas as opposed to some concrete model of computation such as a Turing machine. More recently, there have been various proposals for a machine model of infinitary computation. The earliest of these are the infinite time Turing machines (ITTMs) of Hamkins, Kidder, and Lewis defined in [6]. As their name suggests, these machines generalize Turing machines by allowing a computation to proceed for any ordinal number of steps. While these machines generalize the time in which a computation can occur, they still restrict the machine to a tape of length ω. Thus, ITTMs generalize time, but they do not generalize space. Another example that generalizes time, but not space, are the infinite time register machines defined by Koepke in [10] and further developed in [1, 12].

There is a noticeable asymmetry, however, if one allows a machine to take any ordinal number of steps while also restricting the space it can use. A far more natural approach is the one suggested in [9], in which Koepke defines what we will refer to as Koepke machines. These machines can take any ordinal number of steps, but they also have a class-length tape indexed by the ordinals. Thus, Koepke machines generalize Turing machines in both time and space.

While generalizations have been made for time and space, none of these models generalize the last key feature of a Turing machine: its internal programming. Each of the models mentioned still require that programs be finite. In this thesis, we will examine what happens when we relax this requirement. In particular, we will investigate the consequences of allowing Koepke machines to execute infinite programs. In doing so, we will introduce the axiom of com-

putable enumerability and show that by assuming this axiom, we can give exact characterizations of the computably enumerable and decidable classes, as well as the computable functions.

1.1 Thesis Overview

In Chapter 2, we will cover some important background material. The most important topics will be Gödel-Bernays class theory, infinite strings, and Koepke machines. In Chapter 3, we will give our definition of a program and the associated definitions for computability of a function and decidability of a class of strings. We will also define an encoding for programs, which will allow us to define a halting problem that is provably undecidable. Furthermore, we will show that there exists a universal program. In Chapter 4, we will define a restricted halting problem for each infinite cardinal. We will show that programs of cardinality less than a given cardinal cannot decide their associated halting problem, while programs of greater cardinality can. This demonstrates that each increase in the cardinality of programs gives rise to a stronger notion of computability. In Chapter 5, we will define semidecidability and computable enumerability. We will show that the equivalence of these notions is independent of Gödel-Bernays class theory with the axiom of global choice. This is a marked difference from finite computability theory in which semidecidability and computable enumerability are equivalent. In Chapter 6, we will prove results that characterize enumerability, decidability, and computability in terms of the Lévy hierarchy.

Chapter 2

Preliminaries

2.1 Class Theory

With our model of computation, we will be far more interested in proper classes than sets. Therefore, it is important that we work in a theory of classes. For this reason, we have decided to work in Gödel-Bernays class theory.

The variables in Gödel-Bernays range over classes. We say that a class X is a set if it belongs to another class (i.e. there exists a class Y such that $X \in Y$). Following convention, we will use lower case letters to denote classes that are sets, and we will use upper case letters to denote classes in general. In particular, we will use $(\exists x)\varphi(x)$ and $(\forall x)\varphi(x)$ as abbreviations for the formulas $(\exists X)[(\exists Y)(X \in Y) \wedge \varphi(X)]$ and $(\forall X)[(\exists Y)(X \in Y) \to \varphi(X)]$, respectively.

Note that there exists a translation $\varphi \mapsto \varphi^{\mathrm{GB}}$ from formulas in the language of ZF to formulas in the language of Gödel-Bernays, which is given as follows:

$$\begin{aligned}
(x = y)^{\mathrm{GB}} &= (\exists Z)(X \in Z) \wedge (\exists Z)(Y \in Z) \wedge X = Y \\
(x \in y)^{\mathrm{GB}} &= (\exists Z)(X \in Z) \wedge (\exists Z)(Y \in Z) \wedge X \in Y \\
(\neg\varphi)^{\mathrm{GB}} &= \neg\varphi^{\mathrm{GB}} \\
(\varphi \wedge \psi)^{\mathrm{GB}} &= \varphi^{\mathrm{GB}} \wedge \psi^{\mathrm{GB}} \\
\big((\exists x)\varphi\big)^{\mathrm{GB}} &= (\exists X)[(\exists Y)(X \in Y) \wedge \varphi^{\mathrm{GB}}] \\
\big((\forall x)\varphi\big)^{\mathrm{GB}} &= (\forall X)[(\exists Y)(X \in Y) \to \varphi^{\mathrm{GB}}]
\end{aligned}$$

Therefore, every formula in the language of ZF can be identified with a formula in the language of Gödel-Bernays. For this reason, the language of Gödel-Bernays can be considered an extension of the language of ZF.

2.1.1 The Axioms of Gödel-Bernays Class Theory

The axioms of Gödel-Bernays class theory are often broken into five groups labelled A through E, corresponding to the presentation given by Gödel in [5].

Gödel's presentation gives a finite axiomatization of the theory, but we will give an axiomatization that highlights the theory's similarities with ZFC:

A. 1. Extensionality: If X and Y have the same elements, then $X = Y$.
   2. Pairing: For sets x and y, there is a set $z = \{x, y\}$.

B. Comprehension: For classes $X_1, \ldots, X_n$ and for any formula φ in which only set variables are quantified, there is a class $Y = \{x \mid \varphi(x, X_1, \ldots, X_n)\}$.

C. 1. Infinity: There is a set x such that $\emptyset \in x$, and $y \cup \{y\} \in x$ if $y \in x$.
   2. Union: For every set x, there is a set $y = \bigcup x$.
   3. Power Set: For every set x, there is a set $y = \mathcal{P}(x)$.
   4. Replacement: For every function F and set x, there is a set $y = \{F(z) \mid z \in x\}$.

D. Regularity: For every nonempty class X, there is a set $y \in X$ such that $y \cap X = \emptyset$.

E. Global Choice: There is a function F such that $F(x) \in x$ for every nonempty set x.

Let GB denote the theory with axioms A through D, and let GBC denote the theory with axioms A through E.

As mentioned previously, the axioms of GBC are very similar to the axioms of ZFC. In fact, GBC is a conservative extension of ZFC, which is to say that the following theorem holds [3, Corollary 4.1]:

Theorem 2.1. If φ is a sentence in the language of ZF, then φ is provable in ZFC if and only if φ^{GB} is provable in GBC. △

This theorem was proven independently by several people. For more information on its history, we refer the reader to [3, p. 242] and [4, p. 381]. As a consequence of this theorem, we have the following corollary:

Corollary 2.2. ZFC is consistent if and only if GBC is consistent. △

Proof. Suppose GBC is inconsistent. Then GBC can prove that $\emptyset \neq \emptyset$. Since this is expressible in the language of ZF, it follows that ZFC also proves that $\emptyset \neq \emptyset$. Hence, ZFC is inconsistent. Therefore, GBC is consistent if ZFC is consistent. The converse holds by essentially the same argument. □

Thus, GBC can be seen as the weakest possible extension of ZFC that includes classes. It is for this reason that we have decided to use GBC as opposed to some other theory of classes such as Morse-Kelley [8, Appendix], which is not a conservative extension of ZFC.

2.1.2 The Lévy Hierarchy

The *Lévy hierarchy* is a hierarchy of formulas that was introduced by Azriel Lévy in [15]. Typically these are formulas in the language of ZF, and even though we are working in GBC, we will still be interested in the hierarchy associated with (the translations of) formulas in the language of ZF. At the bottom of the hierarchy, we have the Δ_0 formulas:

Definition 2.3. The Δ_0 formulas are defined inductively as follows:

(1) $x = y$ and $x \in y$ are Δ_0 formulas.

(2) If φ and ψ are Δ_0 formulas, then $\neg\varphi$ and $\varphi \wedge \psi$ are Δ_0 formulas.

(3) If φ is a Δ_0 formula, then $(\exists x \in y)\varphi$ and $(\forall x \in y)\varphi$ are Δ_0 formulas.

$\triangle$

To define the rest of the hierarchy, we let the Σ_0 and Π_0 formulas be the Δ_0 formulas. Then we say that a formula is Σ_{n+1} if it is of the form $(\exists x)\varphi$, where φ is Π_n. Similarly, we say that a formula is Π_{n+1} if it is of the form $(\forall x)\varphi$, where φ is Σ_n.

Using the Lévy hierarchy, we can define a similar hierarchy for classes:

Definition 2.4. For any class A, a class B is $\Sigma_n(A)$ if $B = \{x \mid \varphi(x, a_1, \ldots, a_m)\}$, where φ is Σ_n and $a_1, \ldots, a_m \in A$. The definition of $\Pi_n(A)$ is similar. We say that a class is $\Delta_n(A)$ if it is both $\Sigma_n(A)$ and $\Pi_n(A)$. $\triangle$

Note that if a class is $\Sigma_n(V)$, this just means that the class is Σ_n-definable with set parameters. Furthermore, a class is $\Sigma_n(\emptyset)$ if it is Σ_n definable without parameters. In this case, we say that the class is Σ_n. Similarly, we say that a class is Π_n or Δ_n if it is $\Pi_n(\emptyset)$ or $\Delta_n(\emptyset)$, respectively.

2.1.3 The Gödel Pairing Function

The *Gödel pairing function*, denoted by Γ, is a bijection from $\mathrm{Ord} \times \mathrm{Ord}$ to Ord. The definition of Γ is based on the rank function for a set-like well-order. In general, for any well-founded, set-like relation $\prec$ on a class A, let $\mathrm{rank}_\prec a$ denote the rank of a with respect to $\prec$. That is, $\mathrm{rank}_\prec : A \to \mathrm{Ord}$ is given recursively by $\mathrm{rank}_\prec a = \bigcup\{\mathrm{rank}_\prec b + 1 \mid b \prec a\}$. Now we can give a formal definition of Γ:

Definition 2.5. The function $\Gamma : \mathrm{Ord} \times \mathrm{Ord} \to \mathrm{Ord}$ is given by $\Gamma(\alpha, \beta) = \mathrm{rank}_\prec(\alpha, \beta)$, where $\prec$ is the set-like well-order such that $(\alpha, \beta) \prec (\gamma, \delta)$ if and only if $(\alpha \cup \beta, \alpha, \beta) < (\gamma \cup \delta, \gamma, \delta)$, where $<$ denotes lexicographical order. $\triangle$

One of the important features of Γ is that $\Gamma[\omega_\alpha \times \omega_\alpha] = \omega_\alpha$ for every α [7, Theorem 3.5]. As a consequence, $\aleph_\alpha + \aleph_\beta = \aleph_\alpha \cdot \aleph_\beta = \max\{\aleph_\alpha, \aleph_\beta\}$. Later we will see that Γ and Γ^{-1} are computable.

2.1.4 The Set H_κ

The set V_ω is sometimes referred to as the set of all hereditarily finite sets. This is because $x \in V_\omega$ if and only if $\mathrm{TC}(x)$ is finite, where $\mathrm{TC}(x)$ denotes the transitive closure of x. We can generalize this idea to any infinite cardinal κ by defining H_κ as the set of all x such that $|\,\mathrm{TC}(x)| < \kappa$. Moreover, we can prove the following:

Proposition 2.6. If κ is regular, then $H_\kappa = \{x \subseteq H_\kappa \mid |x| < \kappa\}$. △

Proof. Let κ be regular.

($\subseteq$) Suppose $x \in H_\kappa$. Then $|\,\mathrm{TC}(x)| < \kappa$. Since $|\,\mathrm{TC}(y)| \leq |\,\mathrm{TC}(x)| < \kappa$ for every $y \in x$, we have that $x \subseteq H_\kappa$. Furthermore, $|x| \leq |\,\mathrm{TC}(x)| < \kappa$. Hence, $x \subseteq H_\kappa$ and $|x| < \kappa$.

($\supseteq$) Suppose $x \subseteq H_\kappa$ and $|x| < \kappa$. Since $x \subseteq H_\kappa$, we have that $|\,\mathrm{TC}(y)| < \kappa$ for every $y \in x$. Note that $\mathrm{TC}(x) = x \cup \bigcup_{y \in x} \mathrm{TC}(y)$. Therefore, since κ is regular and $|x| < \kappa$, we have that $|\,\mathrm{TC}(x)| = |x \cup \bigcup_{y \in x} \mathrm{TC}(y)| < \kappa$. Hence, $x \in H_\kappa$.

□

2.2 Strings

Even in the finite setting, computation can be thought of as performing basic operations on strings. In this regard, infinitary computation is no different. What is different, however, is that infinitary computations often involve *infinite* strings. So what do we mean when we refer to an infinite string? This is best defined in terms of sequences: A *sequence* is a function whose domain is an ordinal. The *length of a sequence* is its domain. If the range of a sequence f is a subset of a class A, then we say that f is a *sequence of elements of* A.

Now we can give a precise definition of a string: A *string* is a sequence of bits (i.e. a sequence of elements of the set $\{0, 1\}$). We will use $2^{<\mathrm{Ord}}$ to denote the class of all strings, and in general, for any class A, let $A^{<\mathrm{Ord}} = \bigcup_{\alpha \in \mathrm{Ord}} {}^{\alpha}A$. That is, $A^{<\mathrm{Ord}}$ is the class of all sequences of elements of A.

There is one particular sequence, however, that is worth assigning its own symbol. This sequence is know as the *empty sequence*, and we denote it by ε. Strictly speaking, the empty sequence is just the empty set, but it is convenient to have an alternative symbol for when we want to focus on its properties as a sequence. One should also note that the empty sequence is a string, and therefore, we will also refer to it as the *empty string*.

In addition to the empty string, it is convenient to have a notation for other strings of finite length. The notation we will use for this purpose is surrounding the string in straight quotes. For example, we will use '1011' to denote the string $\{(0, 1), (1, 0), (2, 1), (3, 1)\}$.

As notation for infinite strings, we will often use χ_A^B to denote the characteristic function of A restricted to B. If A is a class of ordinals and B is an ordinal, then χ_A^B is a string. In this case, $A \cap B$ is the support of the string χ_A^B. In general, the *support* of a string s, denoted $\operatorname{supp} s$, is the set $\{\alpha \in \operatorname{dom} s \mid s(\alpha) = 1\}$.

Having defined sequences and strings, we can now define some basic operations thereon. The first of these is concatenation, which is the operation of extending one sequence using another sequence.

Definition 2.7. For sequences f and g, their *concatenation* is given by

$$f \sqcup g = f \cup \big\{\big(\operatorname{dom} f + \alpha, g(\alpha)\big) \mid \alpha \in \operatorname{dom} g\big\}$$

In general, the concatenation $\bigsqcup_{\alpha<\beta} f_\alpha$ of a sequence of sequences is defined recursively on β as follows:

$$\begin{aligned}
\bigsqcup_{\alpha<0} f_\alpha &= \varepsilon \\
\bigsqcup_{\alpha<\gamma+1} f_\alpha &= \left(\bigsqcup_{\alpha<\gamma} f_\alpha\right) \sqcup f_\gamma \\
\bigsqcup_{\alpha<\lambda} f_\alpha &= \bigcup_{\gamma<\lambda} \bigsqcup_{\alpha<\gamma} f_\alpha
\end{aligned}$$

△

One of the nice properties of strings is their capacity to encode a wide variety of information. An example of this that will prove tremendously useful is the ability to encode any sequence of strings as a single string. Such strings will fill the role of the list data type often found in computer programming languages. But before we can define the standard encoding of a sequence of strings, we must first introduce the operation of bitwise doubling. In short, the bitwise doubling of a string s is the string that results from repeating each bit in s twice. For example, the bitwise doubling of the string '1011' would be the string '11001111'. More formally,

Definition 2.8. The *bitwise doubling* of a string s is given by

$$2s = \bigsqcup_{\alpha \in \operatorname{dom} s} (2 \times \{s(\alpha)\})$$

△

This operation is not very interesting on its own, but it is important for encoding a sequence of strings as a single string. We will define this encoding now, starting with pairs of strings and then generalizing it to arbitrary sequences.

Definition 2.9. For strings s and t, let $[s,t] = 2s \sqcup \text{'01'} \sqcup 2t \sqcup \text{'01'}$. In general, the *list* $[s_\alpha \mid \alpha < \beta]$ of a sequence of strings is defined recursively on β as follows:

$$\begin{aligned}[s_\alpha \mid \alpha < 0] &= \varepsilon \\ [s_\alpha \mid \alpha < \gamma + 1] &= [s_\alpha \mid \alpha < \gamma] \sqcup 2s_\gamma \sqcup \text{'01'} \\ [s_\alpha \mid \alpha < \lambda] &= \bigcup_{\gamma<\lambda} [s_\alpha \mid \alpha < \gamma]\end{aligned}$$

△

We will use List to denote the class of all lists. That is, $s \in \text{List}$ if there exists a sequence f of strings such that $s = [f(\alpha) \mid \alpha \in \operatorname{dom} f]$. Note that by this definition, a list is itself a string, so List is a class of strings. For finite sequences of strings, we will often just write the list as $[s_1, \ldots, s_n]$.

The lists we have just defined have a couple of nice properties. The first of these is that they are effective. By that we mean that a program should be able to determine each of the elements in a list, and this will indeed be the case for the programs we will define later. The second nice property of these lists is that the concatenation of two lists is the list of the concatenation of their underlying sequences. That is, if f and g are sequences of strings, then $[f(\alpha) \mid \alpha \in \operatorname{dom} f] \sqcup [g(\alpha) \mid \alpha \in \operatorname{dom} g] = [(f \sqcup g)(\alpha) \mid \alpha \in \operatorname{dom}(f \sqcup g)]$. This makes the task of appending strings to a list very simple for a program to perform.

As a final note, we will often want to refer to the elements of a list. For this reason, we will use the notation $s \sqsubseteq t$ to denote that s is one of the strings in the list t. To be precise, for a string s and a list t, we will write $s \sqsubseteq t$ if $s \in \operatorname{ran} f$, where f is the unique sequence of strings such that $t = [f(\alpha) \mid \alpha \in \operatorname{dom} f]$.

2.3 Partial Functions

Our definition of a program will be in terms of partial functions, and the function computed by a program will often be partial. For this reason, we give the following definition and notation for partial functions:

Definition 2.10. A class F is a *partial function* from A to B, denoted by $F : A \dashrightarrow B$, if there exists a subclass A' of A such that F is a function from A' to B. We say F is *total* if $A' = A$. △

Since the exact domain of a partial function is unspecified, the following notation will sometimes be useful:

Definition 2.11. For partial functions $F : A \dashrightarrow B$ and $G : A \dashrightarrow B$, we write $F(a) \simeq G(a)$ if either $a \notin \operatorname{dom} F \cup \operatorname{dom} G$, or $a \in \operatorname{dom} F \cap \operatorname{dom} G$ and $F(a) = G(a)$. △

2.4 Koepke Machines

Our model of computation is based on the work of Koepke [9] in which he defined a generalization of a Turing machine to allow for infinite time and space. We will take this generalization one step further by allowing for infinite programs, but before we do, it is a good idea to review Koepke machines. To properly motivate the definition of a Koepke machine, we will give a brief and informal review of Turing machines, and then we will explain the additional nuances that come from allowing for infinite time and space.

2.4.1 Intuition

A Turing machine can be thought of as a machine with three components. The first of these is a tape that we assume to be as long as necessary for the given computation. This tape is divided into cells, and on each of these cells is written a symbol from some fixed alphabet. For us, this alphabet will consist of the numerals 0 and 1 and also a blank symbol. The second component is a tape head which moves left and right along the tape. The tape head can read the symbol written on the cell at its current location, and it can also change the current symbol on the cell. The third component is the internal programming of the machine, which determines the machine's behavior based upon the contents of the machine's tape. In particular, the machine's program specifies what symbol the tape head should write on the cell at its current location, whether the head should move left or right on the tape, and what the machine's next internal state should be. This determination is made based upon the machine's current internal state and the current symbol written on the cell at the tape head's current position.

Now imagine one wanted to allow a Turing machine to use an infinite amount of time. The computation would proceed normally for the first ω steps, but what happens at time ω? In particular, how are the tape contents, head position, and program state determined at time ω? The answer that Koepke gives is that they are determined by taking inferior limits. That is, the program state at time ω is the inferior limit of all prior program states. Similarly, the symbol on a given cell and the head position at time ω are determined by taking the inferior limits of all prior values, respectively. What is nice about this method is that it works for any limit ordinal, not just ω. Furthermore, the standard rules of execution used by a Turing machine will also work at any successor stage $\alpha > \omega$. Therefore, taking inferior limits at limit stages in a computation allows a Turing machine to use an infinite amount of time, but moreover, defining the head position in terms of the inferior limit of all prior values implicitly assumes that the machine's tape can extend beyond ω. Hence, this also allows a machine to use an infinite amount of space.

2.4.2 Definition

We would like to formalize what we have just discussed. The definitions we will give differ from those found in [9], but none of these differences affect what is or is not computable. We begin with the definition of a Koepke program:

Definition 2.12. A *Koepke program* is a partial function of the form $p : n \times 3 \dashrightarrow n \times 3 \times 2$, where $n < \omega$. △

This reflects the informal definition given earlier because the elements of n represent program states, the elements of 3 represent tape symbols (think of 2 as a blank symbol), and the elements of 2 represent the directions the tape head can move (left and right). Next, we define the execution of a Koepke program:

Definition 2.13. The *execution* of a Koepke program p on the *input* $s : \beta \to 2$ is a triple

$$S : \theta \to \text{Ord} \qquad H : \theta \to \text{Ord} \qquad T : \theta \to {}^{\beta \cup \theta}3$$

representing program state, head position, and tape contents, respectively, at each step $\tau < \theta$. Moreover, these functions satisfy the following requirements:

(1) θ is a successor ordinal, or $\theta = \text{Ord}$. (The program *halts* if θ is a successor ordinal, and the program *diverges* if $\theta = \text{Ord}$.)

(2) $S(0) = H(0) = 0$. (The starting state and head position are both zero.)

(3) $T(0)|_\beta = s$ and $T(0)_{\geq \beta} = 2$. (The initial tape content is the input string, possibly followed by blank symbols.)

(4) If $\tau < \theta$ and $p(S(\tau), T(\tau)_{H(\tau)})$ is undefined, then $\tau + 1 = \theta$. (The program halts if it reaches a state and symbol combination for which it has no accompanying instruction.)

(5) If $\tau < \theta$ and $p(S(\tau), T(\tau)_{H(\tau)}) = (\gamma, m, n)$, then $\tau + 1 < \theta$ and

$$S(\tau + 1) = \gamma$$

$$H(\tau + 1) = \begin{cases} H(\tau) + 1 & \text{if } n = 1 \\ H(\tau) - 1 & \text{if } n = 0 \text{ and } H(\tau) \text{ is a successor ordinal} \\ 0 & \text{otherwise} \end{cases}$$

$$T(\tau + 1)_\delta = \begin{cases} m & \text{if } \delta = H(\tau) \\ T(\tau)_\delta & \text{otherwise} \end{cases}$$

(6) If $\tau < \theta$ is a limit ordinal, then

$$S(\tau) = \liminf_{\gamma \to \tau} S(\gamma) \qquad H(\tau) = \liminf_{\gamma \to \tau} H(\gamma) \qquad T(\tau)_\delta = \liminf_{\gamma \to \tau} T(\gamma)_\delta$$

We say that p *halts after* τ *steps* if $\theta = \tau + 1$. If p halts after τ steps, its *output* is $T(\tau)|_\eta$, where η is the least ordinal such that either $\eta = \operatorname{dom} T(\tau)$ or $T(\tau)_\eta \notin 2$. (This ensures that the output of a halting program is also a string.) If p diverges, then it has no output. △

One of the important results that Koepke proved [9, Theorem 6.2] is the following:

Theorem 2.14 (Koepke)**.** A string s is the output of a Koepke program on an input string with finite support if and only if s is constructible. △

Chapter 3

Infinite Programs

Having taken care of the necessary preliminaries, we are prepared to define our programs and to prove some basic results.

3.1 Definitions

Recall that a Koepke program is a partial function of the form $p : n \times 3 \dashrightarrow n \times 3 \times 2$, where $n < \omega$. The most obvious way to generalize this definition is to allow for infinitely many program states (i.e. replace $n < \omega$ with $\alpha \in \mathrm{Ord}$). This is precisely what our definition of a program will be:

Definition 3.1. A *program* is a partial function of the form $p : \alpha \times 3 \dashrightarrow \alpha \times 3 \times 2$, where $\alpha \in \mathrm{Ord}$. We denote the class of all programs by Prog. △

Now recall Definition 2.13, which is the definition for the execution of a Koepke program. We can, and will, use this exact same definition as the definition for the execution of a program. Next, we define what it means for a function to be computable:

Definition 3.2. For any program p, let $F_p : 2^{<\mathrm{Ord}} \dashrightarrow 2^{<\mathrm{Ord}}$ denote the class of all pairs (s, t) such that p halts on s and t is the output of p on s. We say that a function $F : 2^{<\mathrm{Ord}} \dashrightarrow 2^{<\mathrm{Ord}}$ is *computable* if $F = F_p$ for some program p, in which case, we say that p *computes* F. △

This definition is specifically for functions on strings. To define computability for functions on ordinals, we give the following definition:

Definition 3.3. A function $F : {}^{m}\mathrm{Ord} \to {}^{n}\mathrm{Ord}$ is *computable* if the function $G_n \circ F \circ G_m^{-1} : 2^{<\mathrm{Ord}} \dashrightarrow 2^{<\mathrm{Ord}}$ is computable, where $G_n : (\alpha_1, \ldots, \alpha_n) \mapsto [\chi_{\alpha_1}^{\alpha_1}, \ldots, \chi_{\alpha_n}^{\alpha_n}]$. △

Implicit in this definition is the fact that an ordinal α is encoded as the string χ_α^α, and that a tuple of ordinals is encoded as a list of their respective encodings.

However, when describing programs in a proof, we will often disregard this formal encoding because the exact details of the encoding are often irrelevant to the proof itself. Instead, we will reason about operations on ordinals and lists at a higher level.

In addition to computing functions, we will also be interested in deciding classes of strings:

Definition 3.4. A class A of strings is *decidable* if there exists a (total) computable function $F : 2^{<\mathrm{Ord}} \to \{\text{'0'}, \text{'1'}\}$ such that $F(s) = \text{'1'}$ if and only if $s \in A$. Moreover, if p is a program such that $F = F_p$, then we say that p *decides* A. $\triangle$

3.2 Basic Results

At this point, we should emphasize that our programs can compute anything a Turing machine can compute. Furthermore, they can also compute anything a Koepke program can compute because every Koepke program is a finite program. So then a natural question to ask is what can infinite programs do that finite programs cannot do? In short, the only real advantage an infinite program has over a finite one is the ability to have infinite strings hard-coded into the program. This allows the program to write out the hard-coded string and then use said string in its computation. The next two propositions demonstrate the power that hard-coding infinite strings affords an infinite program.

Proposition 3.5. Every set of strings is decidable. $\triangle$

Proof. Let a be a set of strings. Then there exists a bijection $f : \alpha \to a$ for some ordinal α. Let $t = [f(\beta) \mid \beta < \alpha]$. Given an input s, a program can write out the list t and check if $s \sqsubset t$. If so, then the program halts and outputs '1', and otherwise the program outputs '0'. Such a program will decide a, so a is decidable. $\square$

Proposition 3.6. Every function $f : 2^{<\mathrm{Ord}} \dashrightarrow 2^{<\mathrm{Ord}}$ that is a set is computable. $\triangle$

Proof. Let $f : 2^{<\mathrm{Ord}} \dashrightarrow 2^{<\mathrm{Ord}}$ be a set, and let $a = \{[s,t] \mid (s,t) \in f\}$. Then a is a set, so there exists a bijection $g : \alpha \to a$ for some ordinal α. Let $u = [g(\beta) \mid \beta < \alpha]$. Let p be a program that when given an input s, writes out the list u, checks whether $[s,t] \sqsubset u$ for some string t, and if so, outputs t. Otherwise, p diverges. Then clearly p computes f, so f is computable. $\square$

Initially it may seem that these results imply that our notion of computation is trivial in the sense that we can compute anything, but this is actually not the case. As we will show, there is a halting problem for our programs that is not decidable. This does not contradict the results above because, as we will see, the halting problem is a proper class and not a set. Therefore, while every set of strings is decidable, there are proper classes that are not decidable. This is

analogous to the fact that a Turing machine can decide any finite set of natural numbers, but there are infinite sets that are not decidable.

3.2.1 Computing the Gödel Pairing Function

In order to define the halting problem, we will need a way to encode programs as strings. There are, of course, a variety of ways to do this, but the encoding we will give uses Γ. For this reason, it will be useful to show that both Γ and Γ^{-1} are computable.

Theorem 3.7. The function Γ is a computable bijection such that its inverse is also computable and $\Gamma[\omega_\alpha \times \omega_\alpha] = \omega_\alpha$ for all α. △

Proof. It suffices to show that Γ and its inverse are computable. To show that Γ^{-1} is computable, note that if given an ordinal γ, a program can list the first $\gamma + 1$ pairs of ordinals according to the order $\prec$ given in Definition 2.5. This is done by incrementing an ordinal variable μ, and for each value of μ, listing the pairs in $\{(\alpha, \mu) \mid \alpha < \mu\}$ according to their first coordinate, followed by the pairs in $\{(\mu, \beta) \mid \beta \leq \mu\}$, listed according to their second coordinate. This process repeats until the program has listed $\gamma + 1$ pairs of ordinals, at which point the program can output the pair at position γ in the list. This pair will be the value of $\Gamma^{-1}(\gamma)$.

To show that Γ is computable, note that if given a pair of ordinals (α, β), a program can increment an ordinal variable γ, and for each value of γ, compute $\Gamma^{-1}(\gamma)$ and compare it to (α, β), repeating this process until the two are equal. Since Γ is a bijection, this must occur eventually, at which point the program can output γ. This will be the value of $\Gamma(\alpha, \beta)$. □

On its own, Γ is quite useful, but it would also be useful to have functions with similar properties for triples, quadruples, quintuples, etc. To this end, we give the following definition:

Definition 3.8. For $n > 0$, the function $\Gamma_n : {}^n\text{Ord} \to \text{Ord}$ is defined recursively on n so that Γ_1 is the identity function on ordinals, and for $m > 0$, $\Gamma_{m+1}(\alpha_1, \ldots, \alpha_m, \alpha_{m+1}) = \Gamma(\Gamma_m(\alpha_1, \ldots, \alpha_m), \alpha_{m+1})$. △

From this definition, we obtain the following corollary of Theorem 3.7:

Corollary 3.9. The function Γ_n is a computable bijection such that its inverse is also computable and $\Gamma[{}^n\omega_\alpha] = \omega_\alpha$ for all α. △

Often when using Γ_n, the value of n will be clear from context. For this reason, we will often drop the subscript n and just use Γ to denote Γ_n.

3.2.2 The Halting Problem

As mentioned previously, we will be using Γ to encode programs as strings. This encoding is defined in terms of the support of a string. We say that a string s

is a *code* for the program p if $\operatorname{supp} s = \Gamma[p]$. More explicitly, s is a code for p if

$$\{\alpha \in \operatorname{dom} s \mid s(\alpha) = 1\} = \{\Gamma_5(\alpha, m, \beta, n, k) \mid (\alpha, m, \beta, n, k) \in p\}$$

Note that every string is a code for at most one program, but not every string is a code for some program. For strings that do not encode any program, we will associate them with a program that computes the identity function on strings. Strictly speaking, the empty set is an example of such a program because it immediately halts on every input, and therefore, its input is always its output. Therefore, to assign a program to every string, we give the following definition:

Definition 3.10. The function $\pi : 2^{<\mathrm{Ord}} \to \mathrm{Prog}$ is given by

$$\pi(s) = \begin{cases} p & \text{if } \operatorname{supp} s = \Gamma[p] \\ \emptyset & \text{otherwise} \end{cases}$$

△

As a final matter of notation before defining the halting problem, we will write $p \downarrow s$ if the program p halts on the input s, and we will write $p \uparrow s$ if the program p diverges on the input s. Furthermore, we will write $p \overset{\downarrow}{\alpha} s$ if p halts after fewer than α steps on the input s. Now we can finally give a definition of the halting problem:

Definition 3.11. The halting problem is the class

$$K = \{[s, t] \in \mathrm{List} \mid \pi(s) \downarrow t\}$$

△

As promised, we can prove the following theorem:

Theorem 3.12. The halting problem is undecidable. △

Proof. Assume to the contrary that there exists a program p that decides K. Let p' be a program that when given an input s, computes $F_p([s, s])$, and then halts if and only if $F_p([s, s]) = \text{'0'}$. Let e be a code for p'. Then we have that

$$\begin{aligned} p' \downarrow e &\iff \pi(e) \downarrow e && (e \text{ is a code for } p') \\ &\iff F_p([e, e]) = \text{'1'} && (p \text{ decides } K) \\ &\iff p' \uparrow e && (\text{definition of } p') \end{aligned}$$

This creates a contradiction. Hence, K is undecidable. □

Thus, we have shown that our notion of computation is indeed nontrivial in the sense that not everything is computable.

3.2.3 The Universal Program

The encoding we have introduced to define the halting problem can also be used to prove that there is a universal program:

Theorem 3.13. There exists a program p such that $F_p(s) \simeq F_{\pi(s_0)}(s_1)$ if $s = [s_0, s_1]$ for strings s_0 and s_1, and otherwise, $p \uparrow s$. △

Proof. Let p be a program that when given an input s, checks if s is a two-element list. If not, the program diverges. Otherwise, $s = [s_0, s_1]$ for strings s_0 and s_1, in which case p can determine s_0 and s_1 from s. Therefore, p can write out the elements of $p' = \Gamma^{-1}[\operatorname{supp} s_0]$ because Γ^{-1} is computable. Then p can check whether p' is a program by checking that

(1) $\beta, \delta \in 3$ and $\zeta \in 2$ for every $(\alpha, \beta, \gamma, \delta, \zeta) \in p'$

(2) If $(\alpha, \beta, \gamma, \delta, \zeta), (\alpha, \beta, \gamma', \delta', \zeta') \in p'$, then $(\gamma, \delta, \zeta) = (\gamma', \delta', \zeta')$

If p' is a program, then p has all the information it needs to simulate the execution of p' on the input s_1. If p' is not a program, then p halts and outputs s_1. Therefore, p is a program such that $F_p(s) \simeq F_{\pi(s_0)}(s_1)$ if $s = [s_0, s_1]$ for strings s_0 and s_1, and otherwise, $p \uparrow s$. □

Note that the universal program is finite because it does not require any hard-coded, infinite strings. Therefore, the universal program is a Koepke program, so every program is equivalent to a Koepke program with a string parameter.

Chapter 4

A Hierarchy of Computability

Until now, we have not said much about the length or size of our programs other than the fact that they are either finite or infinite. A natural question to ask, however, is to what extent the order type or cardinality of states allowed in a program affects what they can compute. We will demonstrate that the order type is relatively unimportant, but the cardinality does make a difference. Furthermore, we will show that this leads to a hierarchy of computability based on the cardinalities of programs.

4.1 Trimmed Programs

It will be helpful for this chapter (and in general) to focus our attention on trimmed programs. Before defining what a trimmed program is, we introduce some notation: For any program p, let $\operatorname{sts} p$ denote the set of states in p. That is, $\operatorname{sts} p = \bigcup \big\{ \{\alpha, \beta\} \mid (\exists m, n, k)(\alpha, m, \beta, n, k) \in p \big\}$. A program p is *trimmed* if $\operatorname{sts} p$ is an ordinal. The motivation behind this definition is that a trimmed program has no "gaps" between states. The removal of these gaps is analogous to removing blank lines in a computer program, hence the use of the word trimmed.

We will show that every program is equivalent to a trimmed program. By equivalent, we mean that they compute the same function. In general, the programs p and p' are *equivalent* if $F_p = F_{p'}$. Showing that every program is equivalent to a trimmed program is not entirely trivial, especially in the case of infinite programs. One of the reasons for this is demonstrated in the following example:

Example 4.1. Consider the following program:

$$\{(\alpha, 2, \alpha + 1, 1, 1) \mid \alpha < \omega \text{ or } \alpha = \omega + 1\}$$

Note that this program is not trimmed because it has a gap at ω. On the input ε, this program will output χ^{ω}_{ω}. If we naively remove the gap by shifting the

state $\omega + 1$ down to ω, we have the following program:

$$\{(\alpha, 2, \alpha + 1, 1, 1) \mid \alpha < \omega + 1\}$$

This program is indeed trimmed, but it will output $\chi_{\omega+1}^{\omega+1}$ when given the input ε. Therefore, in general, we cannot remove gaps at limit ordinals by simply shifting states. △

One may notice that for the program in the example above, we could just remove the state $\omega + 1$ altogether and obtain an equivalent trimmed program, but in general this will not always work. What will always work, however, is a combination of shifting states and adding limit states. This method is used in the proof of the following theorem:

Theorem 4.2. Every program of cardinality less than $\aleph_\alpha$ is equivalent to a trimmed program of cardinality less than $\aleph_\alpha$. △

Proof. Let p be a program of cardinality less than $\aleph_\alpha$, and let β be the order type of $(\overline{\operatorname{sts} p}, <)$, where $\overline{\operatorname{sts} p} = \{\bigcup x \mid x \subseteq \operatorname{sts} p\}$ (i.e. the closure of $\operatorname{sts} p$). Then there exists an order isomorphism $f : \beta \to \overline{\operatorname{sts} p}$. Note that f is continuous because $\overline{\operatorname{sts} p}$ contains all its limit points. Now let

$$p' = \{(f^{-1}(\gamma), m, f^{-1}(\delta), n, k) \mid (\gamma, m, \delta, n, k) \in p\}$$

Then p' is equivalent to p. To see why, note that if S_p and $S_{p'}$ are the functions denoting the state of p and p' respectively at each stage in their execution, then $f\big(S_{p'}(\tau)\big) = S_p(\tau)$ at every stage τ. This is obviously true at successor stages, but it is also true at limit stages because f is a continous order isomorphism, so

$$f\left(\liminf_{\tau\to\lambda} S_{p'}(\tau)\right) = \liminf_{\tau\to\lambda} f\big(S_{p'}(\tau)\big)$$

Therefore,

$$\begin{aligned}
f\big(S_{p'}(\lambda)\big) &= f\left(\liminf_{\tau\to\lambda} S_{p'}(\tau)\right) && \text{(definition of } S_{p'}\text{)} \\
&= \liminf_{\tau\to\lambda} f\big(S_{p'}(\tau)\big) && (f \text{ is cont. order isomorphism}) \\
&= \liminf_{\tau\to\lambda} S_p(\tau) && \text{(induction hypothesis)} \\
&= S_p(\lambda) && \text{(definition of } S_p\text{)}
\end{aligned}$$

Since these programs perform the same operations at corresponding states, it follows that these programs must be equivalent.

We are not quite done yet. Although $\overline{\operatorname{sts} p'}$ is an ordinal because $\overline{\operatorname{sts} p'} = \overline{f^{-1}[\operatorname{sts} p]} = f^{-1}\big[\overline{\operatorname{sts} p}\big] = \beta$, it is not necessarily the case that $\overline{\operatorname{sts} p'} = \operatorname{sts} p'$. In other words, there may still be "gaps" at limit ordinals. Therefore, we need to fill in these gaps. To do so, let

$$p'' = p' \cup \{(\gamma, m, \beta, m, 1) \mid (\gamma, m) \in (\overline{\operatorname{sts} p'} \setminus \operatorname{sts} p') \times 3\}$$

Then p'' is equivalent to p'. To understand why, note that if p' enters a state in $\overline{\operatorname{sts} p'} \setminus \operatorname{sts} p'$, then it will halt. Similarly, if p'' enters a state in $\overline{\operatorname{sts} p'} \setminus \operatorname{sts} p'$, then it will move its tape head to the right once and enter the state β, at which point it will also halt. Since the execution of p'' and p' is the same at every other state, these programs are equivalent, so p'' is equivalent to p.

And finally, p'' is trimmed because $\operatorname{sts} p'' = \overline{\operatorname{sts} p'} \cup \{\beta\} = \beta + 1$ if $\overline{\operatorname{sts} p'} \setminus \operatorname{sts} p'$ is nonempty, and otherwise, $\operatorname{sts} p'' = \operatorname{sts} p' = \overline{\operatorname{sts} p'} = \beta$. Furthermore, $|\overline{\operatorname{sts} p'}| < \aleph_\alpha$ because $|\operatorname{sts} p'| = |\operatorname{sts} p| < \aleph_\alpha$, so $|\operatorname{sts} p''| < \aleph_\alpha$. Hence, p'' is a trimmed program of cardinality less than $\aleph_\alpha$ that is equivalent to p. □

A nice consequence of the theorem we have just proven is that if p is a program of cardinality less than $\aleph_\alpha$, then we can assume that $\operatorname{sts} p$ is an ordinal less than ω_α. Can we make any further assumptions about the specific value of $\operatorname{sts} p$? It turns out we can. If p is a program of cardinality $\aleph_\alpha$, then we can assume that $\operatorname{sts} p = \omega_\alpha + 1$. This is demonstrated in the proof of the following thereom:

Theorem 4.3. Every program of cardinality at most $\aleph_\alpha$ is equivalent to a trimmed program p such that $\operatorname{sts} p \leq \omega_\alpha + 1$. △

Proof. Let p be a program of cardinality at most $\aleph_\alpha$. By Theorem 4.2, we can assume p is trimmed. If $|p| < \aleph_\alpha$, then since p is trimmed, $\operatorname{sts} p < \omega_\alpha$, so clearly $\operatorname{sts} p \leq \omega_\alpha + 1$. Now suppose $|p| = \aleph_\alpha$. Then there exists a bijection $f : \omega_\alpha \to \operatorname{sts} p$. Let

$$q = \{(f^{-1}(\beta), m, f^{-1}(\gamma), n, k) \mid (\beta, m, \gamma, n, k) \in p\}$$
$$\prec = \{(\beta, \gamma) \in \omega_\alpha \times \omega_\alpha \mid f(\beta) < f(\gamma)\}$$

Note that $\prec$ is a well-order on ω_α because $\operatorname{sts} p$ is an ordinal.

The idea behind the rest of the proof is to give an encoding of q and $\prec$ as a single string of length ω_α, and then argue that there exists a program p' that writes this string on its tape, decodes q and $\prec$, reconstructs p from q and $\prec$, and then simulates p. Afterwards, we will explain why p' is a trimmed program such that $\operatorname{sts} p' = \omega_\alpha + 1$. To this end, we begin by giving the encoding of q and $\prec$.

Since Γ is such that $\Gamma[{}^n\omega_\alpha] = \omega_\alpha$, it follows that $\Gamma[q] \subseteq \omega_\alpha \supseteq \Gamma[\prec]$. Therefore, we can encode q as $s_q = \chi^{\omega_\alpha}_{\Gamma[q]}$ and $\prec$ as $s_\prec = \chi^{\omega_\alpha}_{\Gamma[\prec]}$, and then we can "zip" these two strings together into a single string s of length ω_α:

$$s = \bigsqcup_{\beta < \omega_\alpha} \big\{\big(0, s_q(\beta)\big), \big(1, s_\prec(\beta)\big)\big\}$$

An infinite program can contain an encoding of s in its states and can therefore write s on its tape. After this, the program can use s to write q and $\prec$ because Γ^{-1} is computable. Furthermore, the program can list the elements of ω_α according to the order $\prec$. To see why, first note that for every $\beta < \omega_\alpha$, the

program can build a list of all $\gamma \prec \beta$. This is done by incrementing γ, and for each value of $\gamma < \omega_\alpha$, checking if $(\gamma, \beta) \in \prec$ and adding it to the list if so. This process repeats until the program has gone through all values of $\gamma < \omega_\alpha$. (The program can check that $\gamma < \omega_\alpha$ because the program can determine ω_α from s.)

Using the procedure above, the program can build a list u of the elements of ω_α according to the order $\prec$. This is done by incrementing an ordinal variable β, and for each value of $\beta < \omega_\alpha$, creating a list of all $\gamma \prec \beta$ as explained above. Then the program can compare this list to u. If they contain the same elements (not necessarily in the same order), it means that β is the next ordinal according to the order $\prec$. Therefore, if this occurs, the program appends β to u, resets β to 0, and repeats this process to search for the next ordinal to add to the list u. Otherwise, the program just increments β and continues its current search. This process repeats until the program goes through all values of $\beta < \omega_\alpha$ without adding any elements to u, at which point u will be a list of the elements of ω_α according to the order $\prec$.

Now note that by listing the elements of ω_α according to the order $\prec$, this allows the program to compute f because $f(\beta) = \operatorname{rank}_\prec \beta$, which is just the position of β in the list u. Therefore, the program can use f and q to write p and then use this to simulate p on the given input. Since this program simulates p, the two must be equivalent.

Finally, we must explain why the program p' just described is a trimmed program such that $\operatorname{sts} p' = \omega_\alpha + 1$. To explain this, note that p' can be thought of as having two main parts: the part that writes down s, which requires ω_α consecutive states, and the part that does the necessary computations on s to simulate p, which requires only a finite number n of states. Therefore, we can organize p' as follows: At state zero, p' does nothing and transitions to state $n + 1$, and then from state $n + 1$ through (but not including) state ω_α, p' writes down s. Having transitioned through all these states, p' will be at a limit stage, and its state will be ω_α. Therefore, at state ω_α, p' does nothing and transitions to state 1, and then states 1 through (and including) n can be used to perfrom the necessary computations on s to simulate p. Therefore, p' is trimmed (because we can assume its finite portion is trimmed by Theorem 4.2) and $\operatorname{sts} p' = \omega_\alpha + 1$. □

The important consequence of this theorem is that the order type of a program's states is relatively unimportant. This is because any program can be rewritten as a program p such that $\operatorname{sts} p = \omega_\alpha + 1$ for some α. What is important, however, is the program's cardinality. For this reason, we give the following definitions:

Definition 4.4. A function is *κ-computable* if it is computed by a program of cardinality less than κ. △

Definition 4.5. A class of strings is *κ-decidable* if it is decided by a program of cardinality less than κ. △

4.2 The Restricted Halting Problem

What we would like to show is that if $\kappa > \lambda \geq \omega$, then the notion of κ-computability is strictly stronger than the notion of λ-computability. That is, there are κ-computable functions that are not λ-computable. If this is the case, then there is a hierarchy of computability notions based on the cardinalities of programs.

For solving this problem, a natural starting point would be the halting problem. We know that the halting problem is not decidable by any program, but maybe we can define a modified halting problem that is κ-decidable but not λ-decidable. To this end, we give the following definition:

Definition 4.6. For any infinite cardinal κ, the κ-restricted halting problem is the set $K_\kappa = \{s \in K \mid |s| < \kappa\}$. △

Indeed we can show that K_κ is not κ-decidable. The proof is very similar to the proof that K is not decidable, but there are a few added details to consider, so we will give the proof in full.

Theorem 4.7. K_κ is not κ-decidable. △

Proof. Assume to the contrary that K_κ is κ-decidable. Then there exists a program p of cardinality less than κ that decides K_κ. Let p' be a program that when given an input s, computes $F_p([s,s])$, and then halts if and only if $F_p([s,s]) = \text{'0'}$. Note that p' will also be a program of cardinality less than κ. By Theorem 4.2, we can assume p' is trimmed, so due to the properties of Γ, p' has a code e of cardinality less than κ. Since e has cardinality less than κ, it follows that $[e,e]$ also has cardinality less than κ, but then we have that

$$\begin{aligned} p' \downarrow e &\iff \pi(e) \downarrow e && (e \text{ is a code for } p') \\ &\iff F_p([e,e]) = \text{'1'} && (p \text{ decides } K_\kappa) \\ &\iff p' \uparrow e && (\text{definition of } p') \end{aligned}$$

This creates a contradiction. Hence, K_κ is not κ-decidable. □

We can also show that K_κ is in fact κ^+-decidable. To prove this, we will need the following lemma:

Lemma 4.8. If p is a program and s is a string such that both are of cardinality less than $\kappa > \omega$, then p halts on the input s only if it halts after fewer than κ steps on the input s. △

Proof. It is already known [2, Proposition 1] that if given an input s of cardinality less than $\kappa > \omega$, a finite program will halt on s only if it halts after fewer than κ steps on the input s. Let p be a program and let s be a string such that both are of cardinality less than $\kappa > \omega$. Since p has cardinality less than κ, it has a code e of cardinality less than κ, so the string $[e,s]$ is of cardinality

less than κ. Since the universal program is finite, it will halt on $[e, s]$ only if it halts after fewer than κ steps on the input $[e, s]$. Therefore, p halts on the input s only if it halts after fewer than κ steps on the input s because simulating a program's execution takes at least as many steps as the execution itself. □

In essence, this lemma gives us a κ^+-computable halting criterion for programs and input strings of cardinality less than $\kappa > \omega$. We can use this fact to prove the following theorem:

Theorem 4.9. K_κ is κ^+-decidable. △

Proof. We must consider two cases. For the first case, suppose $\kappa = \omega$. Then K_κ is countable, so there exists a bijection $f : \omega \to K_\kappa$. Let $t = [f(\alpha) \mid \alpha < \omega]$, and note that t is also countable. Let p be a program that when given an input s, writes out the list t and checks if $s \sqsubseteq t$. If so, then p halts and outputs '1', and otherwise p outputs '0'. Clearly p decides K_κ. Furthermore, p has cardinality less than ω_1 because t is countable. Hence, K_κ is κ^+-decidable.

Now suppose $\kappa > \omega$. Let p be a program that when given an input s, writes out κ and checks that s is shorter than κ. If not, then p halts and outputs '0'. Otherwise, p checks that $s = [s_0, s_1]$ for strings s_0 and s_1, and if not, p halts and outputs '0'. Otherwise, p checks if $\pi(s_0) \overset{\downarrow}{\kappa} s_1$ by simulating $\pi(s_0)$ for κ steps on the input s_1. By Lemma 4.8, it follows that $\pi(s_0) \downarrow s_1$ if and only if $\pi(s_0) \overset{\downarrow}{\kappa} s_1$, so p can determine if $[s_0, s_1] \in K_\kappa$ by determining if $\pi(s_0) \overset{\downarrow}{\kappa} s_1$. Therefore, p decides K_{κ^+}. Furthermore, p has cardinality less than κ^+ because $\kappa < \kappa^+$ and because the universal program is finite. Hence, K_κ is κ^+-decidable. □

Therefore, if $\kappa > \lambda \geq \omega$, then K_λ is κ-decidable but not λ-decidable. Hence, by increasing the cardinality of programs, we indeed obtain a stronger notion of computability.

Chapter 5

Computable Enumerability

One of the most commonly studied topics in finite computability theory is computable enumerability. Typically, a set of natural numbers (or equivalently, a set of finite strings) is defined as computably enumerable if it is the domain of a (Turing) computable function, but strictly speaking, this is not so much a definition of computable enumerability as it is a definition of semidecidability. It just so happens that in finite computability theory, these concepts are equivalent. The reason for their equivalence has to do with the fact that the set of natural numbers (or equivalently, the set of finite strings) is computably enumerable. When we generalize to an infinite setting, however, the class of ordinals is computably enumerable, but the class of strings is not necessarily computably enumerable. This fact has some interesting consequences which we will address in this chapter.

5.1 Enumerability vs Semidecidability

To be precise, we will say that a class A of strings is *semidecidable* if $A = \operatorname{dom} F$ for some computable function F. This captures the intuitive notion of semidecidability because if a class A of strings is semidecidable, then there exists a program that can "recognize" the elements of A. In contrast, we will say that a class A of strings is *computably enumerable* if there exists a set-like well-order $\prec$ on A such that the function $\operatorname{rank}_{\prec}^{-1} : \operatorname{Ord} \dashrightarrow A$ is computable. This captures the intuitive notion of computable enumerability because if a class A of strings is computably enumerable, then there exists a program that can list the elements of A according to some set-like well-order. In light of the previous chapter, we can also give definitions for *κ-semidecidable* and *κ-computably enumerable* in the obvious way (i.e. replace the word "computable" with "κ-computable" in each definition).

Although the notions of semidecidability and computable enumerability are equivalent in finite computability theory, they are not necessarily equivalent for

our programs. We can, however, show that semidecidability always follows from computable enumerability:

Theorem 5.1. Every κ-computably enumerable class of strings is κ-semidecidable. $\triangle$

Proof. Let A be a κ-computably enumerable class of strings. Then there exists a set-like well-order $\prec$ on A such that $\operatorname{rank}_{\prec}^{-1}$ is κ-computable. Let p be a program that when given an input s, increments an ordinal variable α, and for each value of α, computes $\operatorname{rank}_{\prec}^{-1} \alpha$ and compares it to s. If they are equal, p halts. Otherwise, the process repeats. Note that $p \downarrow s$ if and only if $s \in A$, so $A = \operatorname{dom} F_p$. Furthermore, p is a program of cardinality less than κ because $\operatorname{rank}_{\prec}^{-1}$ is κ-computable. Hence, A is κ-semidecidable. $\square$

In the special case that A is a class of (codes for) ordinals, then the converse of the previous theorem also holds. That is, computable enumerability follows from semidecidability:

Theorem 5.2. Every κ-semidecidable class of ordinals is κ-computably enumerable. $\triangle$

Proof. Let A be a κ-semidecidable class of ordinals. Then there exists a program p of cardinality less than κ such that $A = \operatorname{dom} F_p$. Let $\prec$ be a set-like well-order on A such that $\alpha \prec \beta$ if and only if $\alpha' < \beta'$, where α' is the least ordinal such that $\Gamma^{-1}(\alpha') = (\alpha, \gamma)$ and $p \overset{\downarrow}{\gamma} \alpha$. Now it remains to show that $\operatorname{rank}_{\prec}^{-1}$ is κ-computable.

Given an ordinal α, a program can list the first $\alpha + 1$ ordinals according to the order $\prec$. This is done by incrementing an ordinal variable β, and for each value of β, computing $\Gamma^{-1}(\beta) = (\gamma, \delta)$ and checking if $p \overset{\downarrow}{\delta} \gamma$ by simulating p for δ steps on the input γ. If $p \overset{\downarrow}{\delta} \gamma$ and γ is not already on the list being constructed, then γ is added to the list. Otherwise, γ is not added to the list. This process repeats until the program has listed the first $\alpha + 1$ ordinals according to the order $\prec$, at which point the program can output the ordinal at position α. This will be the value of $\operatorname{rank}_{\prec}^{-1} \alpha$. Note that if A is a set and not a proper class, then it may be that $\alpha \notin \operatorname{dom} \operatorname{rank}_{\prec}^{-1}$, but in this case, the program just described will diverge as desired. Since Γ^{-1} is ω-computable and p has cardinality less than κ, it follows that $\operatorname{rank}_{\prec}^{-1}$ is κ-computable. Hence, A is κ-computably enumerable. $\square$

Thus, for classes of ordinals, the notions of semidecidability and computable enumerability are equivalent. The same is not necessarily true for classes of strings. In particular, the halting problem is an example of a class that is semidecidable, but not necessarily computably enumerable. Moreover, the class of strings is an example of a class that is even decidable, but not necessarily computably enumerable. It is the case, however, that the halting problem and the

class of all strings are computably enumerable if and only if every semidecidable class of strings is computably enumerable:

Theorem 5.3. The following are equivalent:

(1) Every κ-semidecidable class of strings is κ-computably enumerable.

(2) K is κ-computably enumerable.

(3) $2^{<\mathrm{Ord}}$ is κ-computably enumerable.

△

Proof. To show that (1) implies (2), it suffices to show that K is κ-semidecidable. Note that $K = \operatorname{dom} F_p$, where p is the universal program (see Theorem 3.13). Since the universal program is finite, it follows that K is κ-semidecidable.

To show that (2) implies (3), suppose K is κ-computably enumerable. Then there exists a set-like well-order $\prec$ on K such that $\operatorname{rank}_{\prec}^{-1}$ is κ-computable. Now recall that the empty set is a program that immediately halts on every input. Since ε is a code for the empty set, it follows that $[\varepsilon, s] \in K$ for every $s \in 2^{<\mathrm{Ord}}$. Thus, we can define a set-like well-order $\prec'$ on $2^{<\mathrm{Ord}}$ such that $s \prec' t$ if and only if $[\varepsilon, s] \prec [\varepsilon, t]$. Now it remains to show that $\operatorname{rank}_{\prec'}^{-1}$ is κ-computable.

Given an ordinal α, a program can list the first $\alpha + 1$ strings according to the order $\prec'$. This is done by incrementing an ordinal variable β, and for each value of β, computing $\operatorname{rank}_{\prec}^{-1} \beta$. If $\operatorname{rank}_{\prec}^{-1} \beta = [\varepsilon, t]$ for some string t, then the program appends t to the list it is constructing. Repeating this process, the program will eventually list the first $\alpha + 1$ strings according to the order $\prec'$, at which point the program can output the string at position α. This will be the value of $\operatorname{rank}_{\prec'}^{-1} \alpha$. Hence, $\operatorname{rank}_{\prec'}^{-1}$ is κ-computable because $\operatorname{rank}_{\prec}^{-1}$ is κ-computable, so $2^{<\mathrm{Ord}}$ is κ-computably enumerable.

To show that (3) implies (1), suppose $2^{<\mathrm{Ord}}$ is κ-computably enumerable, and let A be a κ-semidecidable class. Then there exists a set-like well-order $\prec$ on $2^{<\mathrm{Ord}}$ such that $\operatorname{rank}_{\prec}^{-1}$ is κ-computable, and there exists a program p of cardinality less than κ such that $A = \operatorname{dom} F_p$. Let $\prec'$ be a set-like well-order on A such that $s \prec' t$ if and only if $\alpha_s < \alpha_t$, where α_s is the least ordinal such that $\Gamma^{-1}(\alpha_s) = (\operatorname{rank}_{\prec} s, \beta)$ and $p \overset{\downarrow}{\beta} s$. Now it remains to show that $\operatorname{rank}_{\prec'}^{-1}$ is κ-computable.

Given an ordinal α, a program can list the first $\alpha + 1$ strings according to the order $\prec'$. This is done by incrementing an ordinal variable β, and for each value of β, computing $\Gamma^{-1}(\beta) = (\gamma, \delta)$. Then the program can determine if $p \overset{\downarrow}{\delta} s$, where $s = \operatorname{rank}_{\prec}^{-1} \gamma$, by simulating the program p for δ steps on the input s. If $p \overset{\downarrow}{\delta} s$ and s is not already on the list being constructed, then s is appended to the list. Repeating this process, the program will list the first $\alpha + 1$ strings according to the order $\prec'$, at which point it can output the string at position α. This will be the value of $\operatorname{rank}_{\prec'}^{-1} \alpha$. Note that if A is a set and not a proper class, then it may be that $\alpha \notin \operatorname{dom} \operatorname{rank}_{\prec'}^{-1}$, but in this case, the

program just described will diverge as desired. Hence, $\text{rank}_{\prec'}^{-1}$ is κ-computable because $\text{rank}_{\prec}^{-1}$ is κ-computable and because p is a program of cardinality less than κ. Thus, A is κ-computably enumerable. □

In the following section, we will demonstrate that the statement "the class of strings is computably enumerable" is independent of GBC. As a consequence of the theorem we have just shown, this implies that the equivalence of computable enumerability and semidecidability is independent of GBC.

5.2 Independence of Enumerability

For convenience, we will refer to the statement "the class of strings is computably enumerable" as ACE, which is an abbreviation for "the axiom of computable enumerability". Similarly, we will use ACE_κ to refer to the statement "the class of strings is κ-computably enumerable". One should note that for each infinite cardinal κ, ACE_κ implies ACE. Furthermore, due to the definition of computably enumerable, ACE implies that the class of strings is well-ordered.

We should also point out that ACE is expressible in the language of ZF. This is because ACE is equivalent to the statement "there exists a program p such that p halts on all and only the (codes for) ordinals, every string is the output of p on some ordinal input, and for any two distinct ordinals, the outputs of p on these ordinals are distinct". This statement only requires quantifying over sets because halting executions are sets. Hence, ACE is equivalent to a statement in the language of ZF.

As the word "axiom" may suggest, one can show that ACE and ACE_κ are independent of GBC. The proof involves showing that both ACE_κ and $\neg$ACE are consistent with GBC. To show that ACE_κ is consistent with GBC, we will show that ACE_κ is a consequence of $\text{V} = \text{L}$. To prove this, however, we need the following result from Koepke [9, Theorem 6.2]:

Theorem 5.4 (Koepke)**.** A string s is the output of a finite program on an input string with finite support if and only if s is constructible. △

This theorem is important because a program can "simultaneously" execute every finite program on every possible string with finite support, which allows a program to enumerate the strings in L. Therefore, if we assume $\text{V} = \text{L}$, then a program can enumerate the entire class of strings. This is demonstrated in the proof of the following lemma:

Lemma 5.5. If $\text{V} = \text{L}$, then $2^{<\text{Ord}}$ is ω-computably enumerable. △

Proof. Suppose $\text{V} = \text{L}$. Then it follows from Theorem 5.4 that every string is the output of some finite program on an input string with finite support. Let $\text{fin} : \text{Ord} \to \omega$ be the function that maps an ordinal α to the unique $n < \omega$ such that $\alpha = \omega \cdot \beta + n$ for some ordinal β. It is not that hard to show that fin is

ω-computable. Let $\prec$ be a set-like well-order on $2^{<\mathrm{Ord}}$ such that $s \prec t$ if and only if $\alpha_s < \alpha_t$, where α_s is the least ordinal such that

(1) $\Gamma^{-1}(\alpha_s) = (\beta, \gamma, \delta, \zeta, \eta)$

(2) $\operatorname{fin}\gamma + 1 = m$ and $\operatorname{fin}\zeta + 1 = n$

(3) $\Gamma_m^{-1}(\beta) = (\beta_1, \ldots, \beta_m)$ and $\Gamma_n^{-1}(\delta) = (\delta_1, \ldots, \delta_n)$

(4) $e = \chi_b^{\beta'}$, where $b = \{\beta_1, \ldots, \beta_m\}$ and $\beta' = \bigcup b$

(5) $u = \chi_d^{\delta'}$, where $d = \{\delta_1, \ldots, \delta_n\}$ and $\delta' = \bigcup d$

(6) $\pi(e) \downarrow_\eta u$

(7) $F_{\pi(e)}(u) = s$

In other words, α_s is the least ordinal that encodes a finite program with code e, an input string u with finite support, and an ordinal η, such that $\pi(e) \downarrow_\eta u$ and $F_{\pi(e)}(u) = s$. Since we are assuming V = L, such an ordinal must exist for every string s. Now it remains to show that $\mathrm{rank}_\prec^{-1}$ is ω-computable.

Given an ordinal α, a program can list the first $\alpha + 1$ strings according to the order $\prec$. This is done by incrementing an ordinal variable θ, and for each value of θ, computing e, u, and η as defined above. Using these, the program can determine if $\pi(e) \downarrow_\eta u$ by simulating $\pi(e)$ for η steps on the input u. If $\pi(e) \downarrow_\eta u$, then the program can determine if $F_{\pi(e)}(u)$ is already in the list it is constructing, and if not, it can add $F_{\pi(e)}(u)$ to the list. This process repeats until the program has listed $\alpha + 1$ strings, at which point it can output the string at position α. This will be the value of $\mathrm{rank}_\prec^{-1}$. Hence, $\mathrm{rank}_\prec^{-1}$ is computable. Furthermore, the program just described is finite because the universal program is finite and because Γ^{-1} and fin are both ω-computable. Thus, $2^{<\mathrm{Ord}}$ is ω-computably enumerable. □

Therefore, ACE_ω is a consequence of V = L, so for every infinite cardinal κ, ACE_κ is a consequence of V = L. To show that ¬ACE is consistent with GBC, we will need a couple results from [3, Theorems 3.1 and 4.2]:

Theorem 5.6 (Easton)**.** If GB is consistent, then the axiom of global choice cannot be proven in the system GB + AC. △

Theorem 5.7. If φ is a sentence in the language of ZF, then φ^{GB} is provable in GBC if and only if φ^{GB} is provable in GB + AC. △

We will also need the following lemma:

Lemma 5.8. If $2^{<\mathrm{Ord}}$ is well-ordered, then V can be well-ordered. △

Proof. Suppose $\prec$ is a well-order on $2^{<\mathrm{Ord}}$. For every α, let $\prec_\alpha = \prec \cap ({}^{\alpha}2 \times {}^{\alpha}2)$. Let $P : V^{<\mathrm{Ord}} \to V^{<\mathrm{Ord}}$ be given by

$$P(f) = \left\{ \left(\operatorname{rank}_{\prec_{\operatorname{dom} f}} t, \{ f(\beta) \mid \beta \in \operatorname{supp} t \} \right) \mid t \in {}^{\operatorname{dom} f}2 \right\}$$

Note that $P(f)$ is a surjection from some ordinal to $\mathcal{P}(\operatorname{ran} f)$. Using P, we can define a function $F : \mathrm{Ord} \to V$ as follows:

$$\begin{aligned} F_0 &= \varepsilon \\ F_{\alpha+1} &= F_\alpha \sqcup P(F_\alpha) \qquad\qquad F = \bigcup_{\alpha \in \mathrm{Ord}} F_\alpha \\ F_\lambda &= \bigcup_{\alpha < \lambda} F_\alpha \end{aligned}$$

It can be shown by induction that $\operatorname{ran} F_\alpha = V_\alpha$ for every α, so F is a surjection. Therefore, we can define a well-order $\prec'$ on V by letting $x \prec' y$ if and only if $\alpha_x < \alpha_y$, where α_x is the least ordinal such that $F(\alpha_x) = x$. Thus, V can be well-ordered. □

Note that the proof of this lemma does not require choice, so it can be proven in GB. Therefore, the axiom of global choice is a theorem of GB + ACE. With this in mind, we can prove the following:

Theorem 5.9. For every infinite cardinal κ, ACE_κ and ¬ACE are consistent with GBC. △

Proof. Let κ be an infinite cardinal. By Lemma 5.5, ACE_κ is a consequence of V = L. Since V = L is consistent with GBC, it must be that ACE_κ is consistent with GBC. Furthermore, ¬ACE is consistent with GBC. To demonstrate this, assume to the contrary that GBC proves ACE. Since ACE is expressible in the language of ZF, it follows by Theorem 5.7 that ACE can be proven in GB + AC. But then by Lemma 5.8, the axiom of global choice can be proven in GB + AC, which contradicts Theorem 5.6. Hence, ¬ACE is consistent with GBC. □

Corollary 5.10. For every infinite cardinal κ, ACE and ACE_κ are independent of ZFC and GBC. △

It is interesting that the axiom of global choice follows from ACE and that the reverse does not hold, but intuitively, the reason for this is because ACE requires that there exists a choice function that is "computable" in the sense that it is definable from some program. The axiom of global choice, however, places no such requirements on the choice function. This is similar to how V = L implies the axiom of global choice and the reverse does not hold. As demonstrated in Lemma 5.5, there is clearly a connection between ACE and V = L. The following theorem clarifies the exact nature of this connection:

Theorem 5.11. V = L is equivalent to ACE_ω. △

Proof. The forward implication follows from Lemma 5.5. As for the reverse implication, note that if the class of strings is computably enumerable by a finite program, then every string is constructible. This is a consequence of Theorem 5.4. Therefore, it suffices to show that every set is constructible if every string is constructible.

Suppose every string is constructible. The proof follows by $\in$-induction. Suppose $x \subseteq L$. Then there exists an ordinal α such that $x \subseteq L_\alpha$. Let β be the order type of $(L_\alpha, <_L)$, where $<_L$ is the canonical well-order on L, and let s be the unique string of length β such that $\operatorname{supp} s = \operatorname{rank}_{<_L}[x]$. Since we are assuming that every string is constructible, there exists some $\gamma > \beta \geq \alpha$ such that $s \in L_\gamma$. Then x is definable over L_γ because $x = \{y \in L_\alpha \mid \operatorname{rank}_{<_L} y \in \operatorname{supp} s\}$, so $x \in L$. Hence, every set is constructible, so $\mathrm{V} = \mathrm{L}$. □

Therefore, if the class of strings is computably enumerable by a finite program, then we can say something very interesting about V (i.e. that $\mathrm{V} = \mathrm{L}$). But what if the class of strings is computably enumerable, but not by a finite program? More generally, we have the following open question:

Open Question 5.12. What can be said of V if we assume $\mathrm{ACE}_{\kappa^+} \wedge \neg\mathrm{ACE}_\kappa$ for some infinite cardinal κ? △

Chapter 6

Characterizing Computation

The purpose of this chapter is to characterize enumerability, decidability, and computability in terms of the Lévy hierarchy.

6.1 Hereditary Lists

One of the first things we want to prove is that every $\Delta_0(V)$ class of strings is decidable. To this end, it will be helpful to have codes for every set. Before we can give a definition of these codes, we need to prove a proposition about the relation $\sqsubseteq$. Recall that $s \sqsubseteq t$ if s is a string and t is a list such that $s \in \operatorname{ran} f$, where f is the unique sequence of strings such that $t = [f(\alpha) \mid \alpha \in \operatorname{dom} f]$. We can prove the following proposition about $\sqsubseteq$:

Proposition 6.1. $\sqsubseteq$ is a well-founded, set-like relation. △

Proof. To show that $\sqsubseteq$ is set-like, note that $\{s \mid s \sqsubseteq t\} = \operatorname{ran} f$ for some sequence f of strings such that $t = [f(\alpha) \mid \alpha \in \operatorname{dom} f]$. Since f is a set, its range is a set, so $\{s \mid s \sqsubseteq t\}$ is a set.

To show that $\sqsubseteq$ is well-founded, assume to the contrary that there exists an infinite descending chain of strings:

$$s_0 \ni s_1 \ni s_2 \ni s_3 \ni \cdots$$

Note that if $s \sqsubseteq t$, then $\operatorname{dom} s < \operatorname{dom} t$. Therefore, there exists an infinite descending chain of ordinals:

$$\operatorname{dom} s_0 > \operatorname{dom} s_1 > \operatorname{dom} s_2 > \operatorname{dom} s_3 > \cdots$$

This, however, is impossible. Hence, $\sqsubseteq$ is well-founded. □

Since $\sqsubseteq$ is a well-founded, set-like relation, we can give the two definitions found below. For more details, see [7, pp. 66-68].

Definition 6.2. Λ is the class of strings that are hereditarily lists. That is, Λ is the unique class such that $\Lambda = \{s \in \mathrm{List} \mid \{t \mid t \sqsubseteq s\} \subseteq \Lambda\}$. △

Definition 6.3. For every $s \in \Lambda$, let $\hat{s} = \{\hat{t} \mid t \sqsubset s\}$. △

The importance of Λ comes from the fact that every $s \in \Lambda$ encodes the set $\hat{s}$. The following are a few examples of elements of Λ and the sets that they encode:

$$\hat{\varepsilon} = \emptyset$$
$$\widehat{[\varepsilon]} = \{\emptyset\}$$
$$\widehat{[\varepsilon, [\varepsilon]]} = \{\emptyset, \{\emptyset\}\}$$

If we let $\Lambda_\kappa = \{s \in \Lambda \mid |s| < \kappa\}$, then we can show the following:

Theorem 6.4. If κ is regular, then $s \mapsto \hat{s}$ is a surjection from Λ_κ to H_κ. △

Proof. Let κ be regular. The proof follows by $\in$-induction. Suppose that for all $y \in x$, if $y \in H_\kappa$, then there exists $s \in \Lambda_\kappa$ such that $\hat{s} = y$. Suppose $x \in H_\kappa$. Then $x \subseteq H_\kappa$ and $|x| < \kappa$. By the induction hypothesis, for every $y \in x$, there exists $s \in \Lambda_\kappa$ such that $\hat{s} = y$. Therefore, there exists a function $f : x \to \Lambda_\kappa$ such that $\widehat{f(y)} = y$ for every $y \in x$. Since $|x| < \kappa$, there exists a bijection $g : \alpha \to \operatorname{ran} f$ for some $\alpha < \kappa$. Let $s = [g(\beta) \mid \beta < \alpha]$. Note that $s \in \Lambda$ because s is a list and $\{t \mid t \sqsubset s\} = \operatorname{ran} f \subseteq \Lambda$. Furthermore, $|s| < \kappa$ because κ is regular and $|t| < \kappa$ for every $t \sqsubset s$. Hence, $s \in \Lambda_\kappa$. Moreover, $\hat{s} = \{\hat{t} \mid t \sqsubset s\} = \{\widehat{f(y)} \mid y \in x\} = x$, so there exists $s \in \Lambda_\kappa$ such that $\hat{s} = x$. Thus, it follows by induction that for every $x \in H_\kappa$, there exists $s \in \Lambda_\kappa$ such that $\hat{s} = x$. □

Corollary 6.5. $s \mapsto \hat{s}$ is a surjection from Λ to V. △

Thus, every set is encoded by some element of Λ. While this is important, the critical feature of these codes is that a program can determine if $\hat{s} = \hat{t}$ and $\hat{s} \in \hat{t}$ from the codes s and t. For $s, t \in \Lambda$, let $s \mathrel{\hat{=}} t$ if and only if $\hat{s} = \hat{t}$, and let $s \mathrel{\hat{\in}} t$ if and only if $\hat{s} \in \hat{t}$. Then we can prove the following theorem:

Theorem 6.6. The classes Λ, $\mathrel{\hat{=}}$, and $\mathrel{\hat{\in}}$ are all ω-decidable. △

Proof. The programs described below use recursion. Formally speaking, this can be implemented using a call stack, which itself can be implemented using a list, but we will not concern ourselves with the details of such an implementation. For examples of recursion being used in infinitary computations, see [10, 14].

(Λ) Let p be a program that when given a string s, checks if s is a list. If not, then p halts and outputs '0'. Otherwise, p checks that $F_p(t) =$ '1' for every $t \sqsubset s$. If so, then p halts and outputs '1'. Otherwise, p halts and outputs '0'. Note that p decides Λ, so Λ is decidable.

($\mathrel{\hat{=}}$) Note that for $s, t \in \Lambda$, we have that

$$\begin{aligned} s \mathrel{\hat{=}} t &\iff \{\hat{u} \mid u \sqsubset s\} = \{\hat{v} \mid v \sqsubset t\} \\ &\iff (\forall u \sqsubset s)(\exists v \sqsubset t)(u \mathrel{\hat{=}} v) \wedge (\forall v \sqsubset t)(\exists u \sqsubset s)(u \mathrel{\hat{=}} v) \end{aligned}$$

This equivalence gives us an algorithm for deciding $\hat{=}$.

Let p be a program that when given strings s and t, checks that $s, t \in \Lambda$. If not, then p halts and outputs '0'. Otherwise, p checks that for every $u \sqsubseteq s$, there exists $v \sqsubseteq t$ such that $F_p(u, v) =$ '1', and that for every $v \sqsubseteq t$, there exists $u \sqsubseteq s$ such that $F_p(u, v) =$ '1'. If so, then p halts and outputs '1'. Otherwise, p halts and outputs '0'. Note that p decides $\hat{=}$, so $\hat{=}$ is decidable.

($\hat{\in}$) Note that for $s, t \in \Lambda$, we have that

$$\begin{aligned} s \mathrel{\hat{\in}} t &\iff \hat{s} \in \{\hat{v} \mid v \sqsubseteq t\} \\ &\iff (\exists v \sqsubseteq t)(s \mathrel{\hat{=}} v) \end{aligned}$$

This gives us an algorithm for deciding $\hat{\in}$.

Let p be a program that when given strings s and t, checks that $s, t \in \Lambda$. If not, then p halts and outputs '0'. Otherwise, p checks that $s \mathrel{\hat{=}} v$ for some $v \sqsubseteq t$. If so, then p halts and outputs '1'. Otherwise, p halts and outputs '0'. Note that p decides $\hat{\in}$, so $\hat{\in}$ is decidable.

Note that each of these programs is finite, so each of these classes is indeed ω-decidable. □

As a final note, the structure $(\Lambda, \hat{=}, \hat{\in})$ is very similar to the class of points defined by Koepke in [11, 14], and just as the class of points is a model of ZFC, one could also prove that $(\Lambda, \hat{=}, \hat{\in})$ is a model of ZFC.

6.2 Decidability of $\Delta_0(V)$ Classes

Using the codes discussed in the previous section, we want to show that every $\Delta_0(V)$ class of strings is decidable. In order to do this, we will need to show that the truth value of every Δ_0 sentence with set parameters is computable. To this end, we give the following definition:

Definition 6.7. For every Δ_0 formula φ, let φ' be given recursively as follows:

$$\begin{aligned} (x = y)' &= (x \mathrel{\hat{=}} y) & (x \in y)' &= (x \mathrel{\hat{\in}} y) \\ (\neg\psi)' &= \neg\psi' & (\psi_0 \wedge \psi_1)' &= \psi_0' \wedge \psi_1' \\ \big((\exists x \in y)\psi\big)' &= (\exists x \sqsubseteq y)\psi' & \big((\forall x \in y)\psi\big)' &= (\forall x \sqsubseteq y)\psi' \end{aligned}$$

△

The importance of φ' comes from the fact that every relation and connective in φ' is computable. This is significant because φ and φ' have the following relationship:

Lemma 6.8. For every Δ_0 formula φ and every $s_1, \ldots, s_n \in \Lambda$, we have that $\varphi(\hat{s}_1, \ldots, \hat{s}_n)$ if and only if $\varphi'(s_1, \ldots, s_n)$. △

Proof. The proof follows by induction on the complexity of φ. By the definitions of $\hat{=}$ and $\hat{\in}$, we have that $\hat{s} = \hat{t}$ if and only if $s \mathrel{\hat{=}} t$, and $\hat{s} \in \hat{t}$ if and only if $s \mathrel{\hat{\in}} t$. The cases for negation and conjunction are straightforward. For existential quantification, note that

$$\begin{aligned}(\exists x \in \hat{s}_i)\psi(x, \hat{s}_1, \ldots, \hat{s}_n) &\iff (\exists t \sqsubseteq s_i)\psi(\hat{t}, \hat{s}_1, \ldots, \hat{s}_n) && \text{(definition of } \hat{s}_i\text{)} \\ &\iff (\exists t \sqsubseteq s_i)\psi'(t, s_1, \ldots, s_n) && \text{(induction hypothesis)}\end{aligned}$$

The case for universal quantification follows by a similar argument. □

Therefore, if we can show that a program can determine the truth value of any sentence of the form $\varphi'(s_1, \ldots, s_n)$, where φ is Δ_0 and $s_1, \ldots, s_n \in \Lambda$, then we will be able to show that every $\Delta_0(V)$ class of strings is decidable. As a technical note, we should say something about encoding a sentence like $\varphi'(s_1, \ldots, s_n)$. There is obviously a finite encoding of φ' for each Δ_0 formula φ. Therefore, if $s_{\varphi'}$ is the code for $\varphi'(x_1, \ldots, x_n)$ and $s_1, \ldots, s_n \in \Lambda$, then a program can use $\left[s_{\varphi'}, [s_1, \ldots, s_n]\right]$ as the code for $\varphi'(s_1, \ldots, s_n)$. Now we would like to show that when given (the code for) $\varphi'(s_1, \ldots, s_n)$, a program can determine its truth value:

Lemma 6.9. There is a finite program that determines the truth value of any sentence of the form $\varphi'(s_1, \ldots, s_n)$, where φ is Δ_0 and $s_1, \ldots, s_n \in \Lambda$. △

Proof. A finite program can determine the truth value of $\varphi'(s_1, \ldots, s_n)$ by recursion on its subformulas. The truth value of atomic sentences can be determined because $\hat{=}$ and $\hat{\in}$ are decidable by a finite program, and the truth values for negations and conjunctions can be determined in the obvious way. The truth value of $(\forall x \sqsubseteq s_i)\psi'(x, s_1, \ldots, s_n)$ can be determined by determining the truth value of $\psi'(t, s_1, \ldots, s_n)$ for every $t \sqsubseteq s_i$ and returning '1' if and only if $\psi'(t, s_1, \ldots, s_n)$ is true for every $t \sqsubseteq s_i$. The truth values of existentially quantified sentences can be determined in a similar manner. □

Now we are almost prepared to prove that every $\Delta_0(V)$ class of strings is decidable. However, there is one more detail that we need to cover. If we want to show that a program can decide the class of strings defined by $\varphi(x, a_1, \ldots, a_n)$, where φ is Δ_0, then we will need codes for $a_1, \ldots, a_n$ as well as for the program's input string. The codes for $a_1, \ldots, a_n$ will come from Λ and can be hard-coded into the program. The code for each input string, however, will need to be computed at runtime. Fortunately this is possible:

Lemma 6.10. There exists an ω-computable function $C : 2^{<\mathrm{Ord}} \to \Lambda$ such that $\widehat{C(s)} = s$ for every string s. △

Proof. In order to define C, we need to define two other functions first. Let

$O : \mathrm{Ord} \to \Lambda$ be given recursively by

$$\begin{aligned} O(0) &= \varepsilon \\ O(\alpha+1) &= O(\alpha) \sqcup [O(\alpha)] \\ O(\lambda) &= \bigcup_{\alpha<\lambda} O(\alpha) \end{aligned}$$

Note that $\widehat{O(\alpha)} = \alpha$ for every ordinal α. Furthermore, O is computable. To see why, note that if given an ordinal α, then for every $\beta \leq \alpha$, a program can determine whether β is zero, a successor ordinal, or a limit ordinal, and then the program can perform some operation on a string variable s depending upon the value of β. If β is zero, the program sets $s = \varepsilon$. If β is a successor ordinal, then the program computes $s \sqcup [s]$ and sets this as the new value of s. If β is a limit ordinal, then the program does nothing to s because the value of s at this stage in the computation will be the union of all prior values of s as desired. Thus, after having performed these operations for every $\beta \leq \alpha$, the resulting value of s will be $O(\alpha)$. Hence, O is computable.

For the second function, let $P : \Lambda \times \Lambda \to \Lambda$ be given by $P(s,t) = \big[[s],[s,t]\big]$. Note that $\widehat{P(s,t)} = \big\{\{\hat{s}\},\{\hat{s},\hat{t}\}\big\} = (\hat{s},\hat{t})$ for every $s,t \in \Lambda$. Furthermore, P is clearly computable. Now with O and P, we can define C. Let $C : 2^{<\mathrm{Ord}} \to \Lambda$ be given by $C(s) = \big[P\big(O(\alpha), O(s(\alpha))\big) \mid \alpha \in \operatorname{dom} s\big]$. Note that for every string s, we have that $\widehat{C(s)} = \big\{\big(\alpha, s(\alpha)\big) \mid \alpha \in \operatorname{dom} s\big\} = s$. Furthermore, C is clearly computable because O and P are computable. As a final note, each of the programs described is finite, so C is ω-computable. □

Now we can finally prove the desired theorem:

Theorem 6.11. If κ is regular, then every $\Delta_0(H_\kappa)$ class of strings is κ-decidable. △

Proof. Let κ be regular, and let A be a $\Delta_0(H_\kappa)$ class of strings. Then there exists some Δ_0 formula φ and sets $a_1, \ldots, a_n \in H_\kappa$ such that

$$A = \{s \in 2^{<\mathrm{Ord}} \mid \varphi(s, a_1, \ldots, a_n)\}$$

By Theorem 6.4, for each a_i there exists $s_i \in \Lambda_\kappa$ such that $\hat{s}_i = a_i$.

Now to show that A is κ-decidable, let p be program that when given an input s, computes $s_0 = C(s)$, writes down the sentence $\varphi'(s_0, s_1, \ldots, s_n)$, and then determines the truth value of $\varphi'(s_0, s_1, \ldots, s_n)$. Since $\varphi'(s_0, s_1, \ldots, s_n)$ is true if and only if $\varphi(s, a_1, \ldots, a_n)$ is true, p decides A. Furthermore, p has cardinality less than κ because $s_1, \ldots, s_n$ have cardinality less than κ. □

Corollary 6.12. Every $\Delta_0(V)$ class of strings is decidable. △

It should be noted that the decidability of $\Delta_0(V)$ classes of strings does not depend on whether we assume that $2^{<\mathrm{Ord}}$ is computably enumerable. In contrast, many of the results in the next section will require assuming that $2^{<\mathrm{Ord}}$ is computably enumerable.

6.3 Characterization Results

In this section, we will prove results that characterize enumerability, decidability, and computability. We will begin with enumerability.

6.3.1 Enumerability of $\Sigma_1(V)$ Classes

Recall Theorem 5.3, which highlights some of the consequences of assuming that $2^{<\mathrm{Ord}}$ is computably enumerable. We can improve this theorem by adding a statement about $\Sigma_1(V)$ classes:

Theorem 6.13. If κ is regular, then the following are equivalent:

(1) Every κ-semidecidable class of strings is κ-computably enumerable.

(2) The halting problem is κ-computably enumerable.

(3) $2^{<\mathrm{Ord}}$ is κ-computably enumerable.

(4) Every $\Sigma_1(H_\kappa)$ class of strings is κ-computably enumerable.

△

Proof. Let κ be regular. The equivalence of (1), (2), and (3) follows from Theorem 5.3. Therefore, it suffices to show the equivalence of (3) and (4). Note that (4) implies (3) because $2^{<\mathrm{Ord}}$ is Δ_0.

To show that (3) implies (4), suppose $2^{<\mathrm{Ord}}$ is κ-computably enumerable. Then there exists a set-like well-order $\prec$ on $2^{<\mathrm{Ord}}$ such that $\mathrm{rank}_\prec^{-1}$ is κ-computable. Let A be a $\Sigma_1(H_\kappa)$ class of strings. Then there is a Δ_0 formula φ and $a_1, \ldots, a_n \in H_\kappa$ such that

$$A = \{s \in 2^{<\mathrm{Ord}} \mid (\exists x)\varphi(s, x, a_1, \ldots, a_n)\}$$

By Theorem 6.4, for each a_i there exists $s_i \in \Lambda_\kappa$ such that $\hat{s}_i = a_i$. Let $\prec'$ be a set-like well-order on A such that $s \prec' t$ if and only if $\alpha_s < \alpha_t$, where α_s is the least ordinal for which there exists $u \in \Lambda$ such that $\Gamma^{-1}(\alpha_s) = (\mathrm{rank}_\prec s, \mathrm{rank}_\prec u)$ and $\varphi(s, \hat{u}, a_1, \ldots, a_n)$. Now it remains to show that $\mathrm{rank}_{\prec'}^{-1}$ is κ-computable.

Given an ordinal α, a program can list the first $\alpha + 1$ strings according to the order $\prec'$. This is done by incrementing an ordinal variable β, and for each value of β, computing $\Gamma^{-1}(\beta) = (\gamma, \delta)$. Then the program can compute $(s, u) = (\mathrm{rank}_\prec^{-1} \gamma, \mathrm{rank}_\prec^{-1} \delta)$ and check if $u \in \Lambda$. If not, then the program continues with the next value of β. Otherwise, the program computes $v = C(s)$ and determines the truth value of $\varphi(s, \hat{u}, a_1, \ldots, a_n)$ by writing down the sentence $\varphi'(v, u, s_1, \ldots, s_n)$ and determining its truth value. If $\varphi'(v, u, s_1, \ldots, s_n)$, then the program appends s to the list it is constructing. This process repeats until the program has listed $\alpha + 1$ strings, at which point it outputs the string at position α. This will be the value of $\mathrm{rank}_{\prec'}^{-1} \alpha$. Note that if A is a set and not a proper class, then it may be that $\alpha \notin \mathrm{dom}\, \mathrm{rank}_{\prec'}^{-1}$, but in this case, the program

just described will diverge as desired. Hence, $\text{rank}_{\prec'}^{-1}$ is computable. Furthermore, $\text{rank}_{\prec'}^{-1}$ is κ-computable because $\text{rank}_{\prec}^{-1}$ is κ-computable and $s_1, \ldots, s_n$ have cardinality less than κ. Hence, A is κ-computably enumerable. □

The theorem we have just proven, in conjunction with the following theorem, gives us a characterization of the computably enumerable classes of strings.

Theorem 6.14. Every κ-computably enumerable class of strings is $\Sigma_1(H_\kappa)$. △

Proof. Suppose A is a κ-computably enumerable class of strings. Then A is enumerated by some program p of cardinality less than κ. By Theorem 4.2, we can assume p is trimmed, so $p \in H_\kappa$. Now note that there is some Δ_0 formula $\varphi(x, y, z, w)$ that expresses "x is a halting execution of the program y on the input z and the output of y is w". Since A is the range of F_p, we have that

$$A = \{s \in 2^{<\text{Ord}} \mid (\exists e)(\exists t)\varphi(e, p, t, s)\}$$

Hence, A is $\Sigma_1(H_\kappa)$. □

Corollary 6.15. If κ is regular and $2^{<\text{Ord}}$ is κ-computably enumerable, then the κ-computably enumerable classes of strings are precisely those that are $\Sigma_1(H_\kappa)$. △

Corollary 6.16. If $2^{<\text{Ord}}$ is computably enumerable, then the computably enumerable classes of strings are precisely those that are $\Sigma_1(V)$. △

Before we end this section, note that by Theorem 6.13 and Theorem 5.1, ACE implies that every $\Sigma_1(V)$ class of strings is semidecidable. What is not immediately clear, however, is if the converse also holds. In other words, we have the following open question:

Open Question 6.17. Is $2^{<\text{Ord}}$ computably enumerable if and only if every $\Sigma_1(V)$ class of strings is semidecidable? △

6.3.2 Decidability of $\Delta_1(V)$ Classes

Next, we can characterize decidability:

Theorem 6.18. If κ is regular and $2^{<\text{Ord}}$ is κ-computably enumerable, then the κ-decidable classes of strings are precisely those that are $\Delta_1(H_\kappa)$. △

Proof. Let κ be regular and suppose $2^{<\text{Ord}}$ is κ-computably enumerable.

($\Rightarrow$) Suppose A is a κ-decidable class of strings. Then A is decided by some program p of cardinality less than κ. By Theorem 4.2, we can assume p is trimmed, so $p \in H_\kappa$. Now let $\varphi(x, y, z)$ be a Δ_0 formula that expresses "x is a halting execution of the program y on the input z" and let $\psi(x, y, z, w)$ be a Δ_0 formula that expresses "x is a halting execution of the program y

on the input z and the output of y is w". Since executions are unique and since p halts on every input, we have that

$$\begin{aligned} A &= \{s \in 2^{<\mathrm{Ord}} \mid (\exists e)[\varphi(e,p,s) \wedge \psi(e,p,s,\text{'1'})]\} \\ &= \{s \in 2^{<\mathrm{Ord}} \mid (\forall e)[\varphi(e,p,s) \to \psi(e,p,s,\text{'1'})]\} \end{aligned}$$

Hence, A is $\Delta_1(H_\kappa)$.

($\Leftarrow$) Suppose A is a $\Delta_1(H_\kappa)$ class of strings. Then A is both $\Sigma_1(H_\kappa)$ and $\Pi_1(H_\kappa)$. Since A is $\Pi_1(H_\kappa)$, it follows that $2^{<\mathrm{Ord}} \setminus A$ is $\Sigma_1(H_\kappa)$. Since we are assuming κ is regular and that $2^{<\mathrm{Ord}}$ is κ-computably enumerable, it follows from Theorem 6.13 that both A and $2^{<\mathrm{Ord}} \setminus A$ are κ-computably enumerable. Therefore, let p and q be programs witnessing that A and $2^{<\mathrm{Ord}} \setminus A$ are κ-computably enumerable, respectively. Using p and q, we can show that A is κ-decidable.

Given an input s, a program can increment an ordinal variable α, and for each value of α, compute $\Gamma^{-1}(\alpha) = (\beta, \gamma)$ and check if $p \overset{\downarrow}{}_\beta \gamma$ or if $q \overset{\downarrow}{}_\beta \gamma$. If $p \overset{\downarrow}{}_\beta \gamma$, then the program can check if $F_p(\gamma) = s$. If so, then the program halts and outputs '1'. Similarly, if $q \overset{\downarrow}{}_\beta \gamma$, then the program can check if $F_q(\gamma) = s$. If so, then the program halts and outputs '0'. If neither $F_p(\gamma) = s$ nor $F_q(\gamma) = s$, then the process repeats. Since every string is an element of either A or $2^{<\mathrm{Ord}} \setminus A$, this process will eventually terminate. Hence, the program just described decides A. Furthermore, this program will have cardinality less than κ because both p and q have cardinality less than κ. Thus, A is κ-decidable.

As a final note, the reason we check that p and q halt after a given number of steps is because it may be that A or $2^{<\mathrm{Ord}} \setminus A$ is a set, in which case F_p or F_q will not be defined on every ordinal.

□

Corollary 6.19. If $2^{<\mathrm{Ord}}$ is computably enumerable, then the decidable classes of strings are precisely those that are $\Delta_1(V)$. △

6.3.3 Computability of $\Sigma_1(V)$ Functions

And finally, we can characterize computability:

Theorem 6.20. If κ is regular and $2^{<\mathrm{Ord}}$ is κ-computably enumerable, then the κ-computable functions on strings are precisely those that are $\Sigma_1(H_\kappa)$. △

Proof. Let κ be regular and suppose $2^{<\mathrm{Ord}}$ is κ-computably enumerable.

($\Rightarrow$) Suppose F is a κ-computable function on strings. Then there exists a program p of cardinality less than κ that computes F. By Theorem 4.2, we can assume p is trimmed, so $p \in H_\kappa$. Now let $\varphi(x,y,z,w)$ be a Δ_0

formula that expresses "x is a halting execution of the program y on the input z and the output of y is w". Then we have that

$$F = \{(s,t) \in 2^{<\mathrm{Ord}} \times 2^{<\mathrm{Ord}} \mid (\exists e)\varphi(e,p,s,t)\}$$

Hence, F is $\Sigma_1(H_\kappa)$.

($\Leftarrow$) Suppose F is a $\Sigma_1(H_\kappa)$ function on strings. Then there exists some Δ_0 formula φ and sets $a_1, \ldots, a_n \in H_\kappa$ such that

$$F = \{(s,t) \in 2^{<\mathrm{Ord}} \times 2^{<\mathrm{Ord}} \mid (\exists x)\varphi(s,t,x,a_1,\ldots,a_n)\}$$

By Theorem 6.4, for each a_i there exists $s_i \in \Lambda_\kappa$ such that $\hat{s}_i = a_i$, and since we are assuming $2^{<\mathrm{Ord}}$ is κ-computably enumerable, there exists a set-like well-order $\prec$ on $2^{<\mathrm{Ord}}$ such that $\mathrm{rank}_\prec^{-1}$ is κ-computable. Now note that when given an input s, a program can increment an ordinal variable α, and for each value of α, compute $\mathrm{rank}_\prec^{-1}\,\alpha$ and check if $\mathrm{rank}_\prec^{-1}\,\alpha = [u,v]$ for strings u and v such that $v \in \Lambda$. If not, then the program continues with the next value of α. Otherwise, the program can compute $t_0 = C(s)$ and $t_1 = C(u)$ and determine the truth value of $\varphi(s,u,\hat{v},a_1,\ldots,a_n)$ by writing down and determining the truth value of $\varphi'(t_0,t_1,v,s_1,\ldots,s_n)$. If $\varphi'(t_0,t_1,v,s_1,\ldots,s_n)$ is true, then the program halts and outputs u. Otherwise, the program continues with the next value of α. Such a program will compute F, so F is computable. Furthermore, F is κ-computable because $\mathrm{rank}_\prec^{-1}$ is κ-computable and because $s_1,\ldots,s_n$ have cardinality less than κ.

□

Corollary 6.21. If $2^{<\mathrm{Ord}}$ is computably enumerable, then the computable functions on strings are precisely those that are $\Sigma_1(V)$. △

Chapter 7

Conclusion

In review, we have defined a model of computation based on infinite programs. This framework has a number of similarities to finite computability theory. In many ways, finite sets of natural numbers correspond to sets of strings in our framework, and similarly, infinite subsets of the natural numbers correspond to proper classes of strings. Just as every finite set of natural numbers is decidable by some Turing machine, every set of strings is decidable by some program. Moreover, we have shown that there exists a halting problem, which is a proper class that is not decidable. This is similar to the halting problem for Turing machines, which is an infinite subset of the natural numbers that is not decidable by any Turing machine. Moreover, for each infinite cardinal κ, we have defined a κ-restricted halting problem. This halting problem is not decidable by any program of cardinality less than κ, but it is decidable by programs of cardinality at least κ. Therefore, restricting our model of computation to programs of cardinality less than κ gives rise to a notion of κ-computability, and if $\kappa > \lambda$, then κ-computability is a strictly stronger notion of computation than λ-computability. This shows that there is a hierarchy of computability notions based on the cardinalities of programs.

There is, however, one key difference between our framework and finite computability theory. The difference is that, while the natural numbers are computably enumerable by a Turing machine, the class of strings is not necessarily computably enumerable. In particular, the axiom of computable enumerability is independent of Gödel-Bernays class theory with the axiom of global choice. As a consequence, the equivalence of semidecidability and computable enumerability is not guaranteed. This stands in stark contrast to finite computability theory, in which a set of natural numbers is semidecidable if and only if it is computably enumerable. However, by assuming the axiom of computable enumerability, we have succeeded in giving exact characterizations of enumerability, decidability, and computability in terms of the Lévy hierarchy. In particular, the computably enumerable classes of strings are precisely those that are $\Sigma_1(V)$, the decidable classes of strings are precisely those that are $\Delta_1(V)$, and the com-

putable funtions are precisely those that are $\Sigma_1(V)$.

7.1 Future Work

There are a number of directions for future research that stem from the material presented in this thesis. One possible direction would be to investigate the parallels between our framework and α-recursion theory in a manner similar to [13]. In this paper, Koepke and Seyfferth used Koepke machines with ordinal parameters. It is likely that similar results could be shown by using infinite programs instead of ordinal parameters.

Another possible direction would be to investigate the consequences of giving programs access to an oracle. To properly implement this, one could augment a Koepke machine with an additional read-write tape. Then at each step in a program's execution, the program is aware of whether the string currently written on this tape belongs to some fixed class of strings, and the program can react accordingly. With a proper definition of oracle programs, one can investigate a variety of topics related to relative computability. For example, one could define Turing reductions and Turing equivalence. With these definitions, one could show that the most common variations of the halting problem are equivalent. In particular, one could show that K is equivalent to the following variants:

$$\{s \in 2^{<\mathrm{Ord}} \mid \pi(s) \downarrow s\} \qquad\qquad \{s \in 2^{<\mathrm{Ord}} \mid \pi(s) \downarrow \varepsilon\}$$

This matches the traditional theory but differs from the theory for infinite time Turing machines. Part of the reason these classes are equivalent in our framework is because one can prove a generalized version of the Parameter Theorem [17, Theorem 3.5 from Chapter I] for infinite programs.

With oracle programs, one could also define a jump operator on classes of strings. Our characterizations of the computably enumerable and decidable classes of strings can be seen as a generalized version of only a small part of Post's theorem [17, Theorem 2.2 from Chapter IV]. It is likely that these characterizations can be generalized to programs with access to the nth jump of the empty set as an oracle, which would give a more complete generalization of Post's theorem. Furthermore, assuming Morse-Kelley [8, Appendix], one could define the αth jump for any class of strings, where α can be any ordinal. With a generalized jump operator, it would be interesting to see to what extent Post's theorem could be generalized.

There are also many other results from finite computability theory that we have not discussed. Assuming the axiom of computable enumerability, it is likely that many of these can be generalized to our framework. Therefore, it would be worthwhile to investigate how much of the traditional theory carries over.

Bibliography

[1] Merlin Carl, Tim Fischbach, Peter Koepke, Russell Miller, Miriam Nasfi, and Gregor Weckbecker. The basic theory of infinite time register machines. *Archive for Mathematical Logic*, 49(2):249–273, Mar 2010.

[2] Merlin Carl, Benedikt Löwe, and Benjamin G. Rin. Koepke machines and satisfiability for infinitary propositional languages. In Jarkko Kari, Florin Manea, and Ion Petre, editors, *Unveiling Dynamics and Complexity - 13th Conference on Computability in Europe, CiE 2017, Turku, Finland, June 12-16, 2017, Proceedings*, volume 10307 of *Lecture Notes in Computer Science*, pages 187–197, Cham, 2017. Springer International Publishing.

[3] Ulrich Felgner. Choice functions on sets and classes. In Gert H. Müller, editor, *Sets and Classes: On The Work by Paul Bernays*, volume 84 of *Studies in Logic and the Foundations of Mathematics*, pages 217–255. Elsevier, 1976.

[4] José Ferreirós. *Labyrinth of Thought: A History of Set Theory and Its Role in Modern Mathematics*. Birkhäuser Basel, Basel, 2007.

[5] Kurt Gödel. *The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis with the Axioms of Set Theory*. Annals of Mathematics Studies. Princeton University Press, 1940.

[6] Joel David Hamkins and Andy Lewis. Infinite time turing machines. *The Journal of Symbolic Logic*, 65(2):567–604, 2000.

[7] Thomas Jech. *Set Theory: The Third Millennium Edition, Revised and Expanded*. Springer Monographs in Mathematics. Springer Berlin Heidelberg, 2006.

[8] John L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer New York, 1975.

[9] Peter Koepke. Turing computations on ordinals. *The Bulletin of Symbolic Logic*, 11(3):377–397, 2005.

[10] Peter Koepke. Infinite time register machines. In Arnold Beckmann, Ulrich Berger, Benedikt Löwe, and John V. Tucker, editors, *Logical Approaches to Computational Barriers, Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30-July 5, 2006, Proceedings*, volume 3988 of *Lecture Notes in Computer Science*, pages 257–266, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] Peter Koepke and Martin Koerwien. Ordinal computations. *Mathematical Structures in Computer Science*, 16(5):867–884, 2006.

[12] Peter Koepke and Russell Miller. An enhanced theory of infinite time register machines. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 306–315, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[13] Peter Koepke and Benjamin Seyfferth. Ordinal machines and admissible recursion theory. *Annals of Pure and Applied Logic*, 160(3):310–318, 2009.

[14] Peter Koepke and Ryan Siders. Register computations on ordinals. *Archive for Mathematical Logic*, 47(6):529–548, Sep 2008.

[15] Azriel Lévy. *A Hierarchy of Formulas in Set Theory*, volume 57 of *Memoirs of the American Mathematical Society*. American Mathematical Society, 1965.

[16] Gerald E. Sacks. *Higher Recursion Theory*. Perspectives in Logic. Cambridge University Press, 2017.

[17] Robert I. Soare. *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Perspectives in Mathematical Logic. Springer Berlin Heidelberg, 1999.

[18] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.