**Computable Structure Theory**

In mathematics, we know there are some concepts—objects, constructions, structures, proofs—that are more complex and difficult to describe than others. Computable structure theory quantifies and studies the complexity of mathematical structures, structures such as graphs, groups, and orderings.

Written by a contemporary expert in the subject, this is the first full monograph on computable structure theory in 20 years. Aimed at graduate students and researchers in mathematical logic, it brings new results of the author together with many older results that were previously scattered across the literature and presents them all in a coherent framework, making it easier for the reader to learn the main results and techniques in the area for application in their own research. This volume focuses on countable structures whose complexity can be measured within arithmetic; a forthcoming second volume will study structures beyond arithmetic.

ANTONIO MONTALBÁN is Professor of Mathematics at the University of California, Berkeley.

PERSPECTIVES IN LOGIC

The *Perspectives in Logic* series publishes substantial, high-quality books whose central theme lies in any area or aspect of logic. Books that present new material not now available in book form are particularly welcome. The series ranges from introductory texts suitable for beginning graduate courses to specialized monographs at the frontiers of research. Each book offers an illuminating perspective for its intended audience.

The series has its origins in the old *Perspectives in Mathematical Logic* series edited by the Ω-Group for "Mathematische Logik" of the Heidelberger Akademie der Wissenschaften, whose beginnings date back to the 1960s. The Association for Symbolic Logic has assumed editorial responsibility for the series and changed its name to reflect its interest in books that span the full range of disciplines in which logic plays an important role.



More information, including a list of the books in the series, can be found at
www.aslonline.org/books/perspectives-in-logic/

PERSPECTIVES IN LOGIC

# *Computable Structure Theory*

## Within the Arithmetic

ANTONIO MONTALBÁN
*University of California, Berkeley*

ASSOCIATION FOR SYMBOLIC LOGIC

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India

79 Anson Road, #06–04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781108423298
DOI: 10.1017/9781108525749

Association for Symbolic Logic
Richard A. Shore, Publisher
Department of Mathematics, Cornell University, Ithaca, NY 14853
http://aslonline.org



First published 2021

*A catalogue record for this publication is available from the British Library.*

ISBN 978-1-108-42329-8 Hardback

Dedicated to my family back home:
Manu, Facu, Mariana, Javier, Patricia, and Nino.

# CONTENTS

# PREFACE

We all know that in mathematics there are proofs that are more difficult than others, constructions that are more complicated than others, and objects that are harder to describe than others. The objective of *computable mathematics* is to study this complexity, to measure it, and to find out where it comes from. Among the many aspects of mathematical practice, this book concentrates on the complexity of structures. By *structures*, we mean objects like rings, graphs, or linear orderings, which consist of a domain on which we have relations, functions, and constants.

Computable structure theory studies the interplay between complexity and structure. By *complexity*, we mean descriptional or computational complexity, in the sense of how difficult it is to describe or compute a certain object. By *structure*, we refer to algebraic or structural properties of mathematical structures. The setting of computable structure theory is that of infinite countable structures and thus, within the whole hierarchy of complexity levels developed by logicians, the appropriate tools come from computability theory: Turing degrees, the arithmetic hierarchy, the hyperarithmetic hierarchy, etc. These structures are like the ones studied in model theory, and we will use a few basic tools from there too. The intention is not, however, to effectivize model theory, and our motivations are very different that those of model theory. Our motivations come from questions of the following sort: Are there syntactical properties that explain why certain objects (like structures, relations, or isomorphisms) are easier or harder to compute or to describe?

The objective of this book is to describe some of the main ideas and techniques used in the field. Most of these ideas are old, but for many of them, the style of the presentation is not. Over the last few years, the author has developed new frameworks for dealing with these old ideas—for instance, for forcing, r.i.c.e. relations, jumps, Scott ranks, and back-and-forth types. One of the objectives of the book is to present these frameworks in a concise and self-contained form.

The modern state of the field, and also the author's view of the subject, has been influenced greatly by the monograph by Ash and Knight [AK00]

published in 2000. There is, of course, some intersection between that book and this one. But even within that intersection, the approach is different.

The intended readers are graduate students and researchers working on mathematical logic. Basic background in computability and logic, as is covered in standard undergraduate courses in logic and computability, is assumed. The objective of this book is to describe some of the main ideas and techniques of the field so that graduate students and researchers can use it for their own research.

This book is part I of a monograph that actually consists of two parts: *within the arithmetic* and *beyond the arithmetic*.

*Part I*, *Within the arithmetic*, is about the part of the theory that can be developed below a single Turing jump. The first chapters introduce what the author sees as the basic tools to develop the theory: $\omega$-presentations, relations, and $\exists$-atomic structures, as treated by the author in [Mon09, Mon12, Mon13c, Mon16a]. Many of the topics covered in Part I (like Scott sentences, 1-generics, the method of true stages, categoricity, etc.) will then be generalized through the transfinite in part II. $\Sigma$-small classes, covered in the last chapter, have been a recurrent topic in the author's work, as they touch on many aspects of the theory and help to explain previously observed behaviors (cf. [HM12, HM14, Mon10, Mon13b]).

*Part II*, *Beyond the arithmetic*, moves into the realm of the hyperarithmetic and the infinitary languages. To fully analyze the complexity of a structure, staying within the arithmetic is not enough. The hyperarithmetic hierarchy goes far enough to capture the complexity levels of relations in almost all structures, though we will see there are some structures whose complexity goes just beyond. The first half of Part II develops the basic theory of infinitary logic, $\Pi^1_1$ sets, and the hyperarithmetic hierarchy. In the second half, the main chapters are those on forcing and the $\alpha$-priority method. The exposition of forcing is only aesthetically new (similar to that in [HTMM]). The presentation of Ash's $\alpha$-priority method will be more than just aesthetically different. It will use the method of $\alpha$-true stages developed in [Mon14b]. We also draw connections with descriptive set theory, and some of the more recent work from [Mon13a, Mon15, MM18]. The chapter on comparability of classes treats old topics like Borel reducibility, but also newer topics like effective reducibility of classes of computable structures (cf. [FF09, FFH$^+$12, Mon16b]) and the connections between functors and interpretability (cf. [HTMMM, HTMM]). Here is the tentative list of chapters of part II [MonP2]:

Chapter 1. Ordinals
Chapter 2. Infinitary logic
Chapter 3. Computably infinitary languages

**Acknowledgements.** Many people helped in different ways throughout the long process that was writing this book, and I'm grateful to them all, though it'd be impossible to name them all. Many people sent me comments and typos along the years. First, I'd like to thank James Walsh for proof reading the whole book. Julia Knight, Peter Cholak, Richard A. Shore, and Barbara Csima ran seminars in Notre Dame, Cornell, and Waterloo following earlier drafts. They then sent me typos and comments, and got their students to send me typos too—that was extremely useful. In particular, Julia Knight gave me lots of useful feedback. So did my students Matthew Harrison-Trainor and Noah Schweber, and also Asher Kach, Jonny Stephenson, and Dino Rossegger.

I learned the subject mostly from Julia Knight and from my Ph.D. advisor Richard A. Shore. I also learned a lot form Ted Slaman, Rod Downey, and Denis Hirschfeldt. I owe them all a great debt.

My work was partially supported by NSF grants DMS-1363310 and DMS-1700361, by the Packard fellowship, and by the Simons fellowship 561299.

# NOTATION AND CONVENTIONS

The intention of this section is to refresh the basic concepts of computability theory and structures and set up the basic notation we use throughout the book. If the reader has not seen basic computability theory before, this section would be too fast an introduction and we recommend starting with other textbooks like Cutland [Cut80], Cooper [Coo04], Enderton [End11], or Soare [Soa16].

**The computable functions.** A function is *computable* if there a purely mechanical process to calculate its values. In today's language, we would say that $f : \mathbb{N} \to \mathbb{N}$ is computable if there is a computer program that, on input $n$, outputs $f(n)$. This might appear to be too informal a definition, but the Turing–Church thesis tells us that it does not matter which method of computation you choose, you always get the same class of functions from $\mathbb{N}$ to $\mathbb{N}$. The reader may choose to keep in mind whichever definition of computability feels intuitively more comfortable, be it Turing machines, $\mu$-recursive functions, lambda calculus, register machines, Pascal, Basic, C++, Java, Haskell, or Python.[1] We will not use any particular definition of computability, and instead, every time we need to define a computable function, we will just describe the algorithm in English and let the reader convince himself or herself that it can be written in the programing language he or she has in mind.

The choice of $\mathbb{N}$ as the domain and image for the computable functions is not as restrictive as it may sound. Every hereditarily finite object[2] can be encoded by just a single natural number. Even if formally we define computable functions as having domain $\mathbb{N}$, we think of them as using any kind of finitary object as inputs or outputs. This should not be surprising. It is what computers do when they encode everything you see on the screen using finite binary strings, or equivalently, natural numbers written in binary. For instance, we can encode pairs of natural numbers by a single number using the *Cantor pairing function* $\langle x, y\rangle \mapsto ((x+y)(x+y+1)/2+y)$,

[1] For the reader with a computer science background, let us remark that we do not impose any time or space bound on our computations—computations just need to halt and return an answer after a finitely many steps using a finite amount of memory.

[2] A hereditarily finite object consist of a finite set or tuple of hereditarily finite objects.

which is a bijection from $\mathbb{N}^2$ to $\mathbb{N}$ whose inverse is easily computable too. One can then encode triples by using pairs of pairs, and then encode $n$-tuples, and then tuples of arbitrary size, and then tuples of tuples, etc. In the same way, we can consider standard effective bijections between $\mathbb{N}$ and various other sets like $\mathbb{Z}$, $\mathbb{Q}$, $V_\omega$, $\mathcal{L}_{\omega,\omega}$, etc. Given any finite object $a$, we use Quine's notation $\ulcorner a \urcorner$ to denote the number coding $a$. Which method of coding we use is immaterial for us so long as the method is sufficiently effective. We will just assume these methods exist and hope the reader can figure out how to define them.

Let

$$\Phi_0, \Phi_1, \Phi_2, \Phi_3, \dots$$

be an enumeration of the computer programs ordered in some effective way, say lexicographically. Given $n$, we write $\Phi_e(n)$ for the output of the $e$th program on input $n$. Each program $\Phi_e$ calculates the values of a *partial computable function* $\mathbb{N} \rightharpoonup \mathbb{N}$. Let us remark that, on some inputs, $\Phi_e(n)$ may run forever and never halt with an answer, in which case $\Phi_e(n)$ is undefined. If $\Phi_e$ returns an answer for all $n$, $\Phi_e$ is said to be *total*—even if total, these functions are still included within the class of partial computable functions. The *computable functions* are the total functions among the partial computable ones. We write $\Phi_e(n)\downarrow$ to mean that this computation *converges*, that is, that it halts after a finite number of steps; and we write $\Phi_e(n)\uparrow$ to mean that it *diverges*, i.e., it never returns an answer. Computers, as Turing machines, run on a step-by-step basis. We use $\Phi_{e,s}(n)$ to denote the output of $\Phi_e(n)$ after $s$ steps of computation, which can be either not converging yet $(\Phi_{e,s}(n)\uparrow)$ or converging to a number $(\Phi_{e,s}(n)\downarrow = m)$. Notice that, given $e, s, n$, we can decide whether $\Phi_{e,s}(n)$ converges or not, computably: All we have to do is run $\Phi_e(n)$ for $s$ steps. If $f$ and $g$ are partial functions, we write $f(n) = g(m)$ to mean that either both $f(n)$ and $g(m)$ are undefined, or both are defined and have the same value. We write $f = g$ if $f(n) = g(n)$ for all $n$. If $f(n) = \Phi_e(n)$ for all $n$, we say that $e$ is an *index* for $f$. The *Padding Lemma* states that every partial computable function has infinitely many indices—just add dummy instructions at the end of a program, getting essentially the same program, but with a different index.

In his famous 1936 paper, Turing showed there is a partial computable function $U : \mathbb{N}^2 \to \mathbb{N}$ that encodes all other computable functions in the sense that, for every $e, n$,

$$U(e, n) = \Phi_e(n).$$

This function $U$ is said to be a *universal partial computable function*. It does essentially what computers do nowadays: You give them an index for a program and an input, and they run it for you. We will not use $U$ explicitly throughout the book, but we will constantly use the fact that we

can computably list all programs and start running them one at the time, using $U$ implicitly.

We identify subsets of $\mathbb{N}$ with their characteristic functions in $2^{\mathbb{N}}$, and we will move from one viewpoint to the other without even mentioning it. For instance, a set $A \subseteq \mathbb{N}$ is said to be *computable* if its characteristic function is.

An *enumeration* of a set $A$ is nothing more than an onto function $g : \mathbb{N} \to A$. A set $A$ is *computably enumerable* (*c.e.*) if it has an enumeration that is computable. The empty set is computably enumerable too. Equivalently, a set is computably enumerable if it is the domain of a partial computable function.[3] We denote

$$W_e = \{n \in \mathbb{N} : \Phi_e(n)\downarrow\} \quad \text{and} \quad W_{e,s} = \{n \in \mathbb{N} : \Phi_{e,s}(n)\downarrow\}.$$

As a convention, we assume that $W_{e,s}$ is finite, and furthermore, that only on inputs less than $s$ can $\Phi_e$ converge in less than $s$ steps. One way to make sense of this is that numbers larger than $s$ should take more than $s$ steps to even be read from the input tape. We sometimes use Lachlan's notation: $W_e[s]$ instead of $W_{e,s}$. In general, if $a$ is an object built during a construction and whose value might change along the stages of the construction, we use $a[s]$ to denote its value at stage $s$. A set is *co-c.e.* if its complement is c.e.

Recall that a set is computable if and only if it and its complement are computably enumerable.

The *recursion theorem* gives us one of the most general ways of using recursion when defining computable functions. It states that for every computable function $f : \mathbb{N}^2 \to \mathbb{N}$ there is an index $e \in \mathbb{N}$ such that $f(e, n) = \varphi_e(n)$ for all $n \in \mathbb{N}$. Thus, we can think of $f(e, \cdot) = \varphi_e(\cdot)$ as a function of $n$ which uses its own index, namely $e$, as a parameter during its own computation, and in particular is allowed to call and run itself.[4] An equivalent formulation of this theorem is that, for every computable function $h : \mathbb{N} \to \mathbb{N}$, there is an $e$ such that $W_{h(e)} = W_e$.

**Sets and strings.** The natural numbers are $\mathbb{N} = \{0, 1, 2, \dots\}$. For $n \in \mathbb{N}$, we sometimes use $n$ to denote the set $\{0, \dots, n-1\}$. For instance, $2^{\mathbb{N}}$ is the set of functions from $\mathbb{N}$ to $\{0, 1\}$, which we will sometimes refer to as *infinite binary sequences* or *infinite binary strings*. For any set $X$, we use $X^{<\mathbb{N}}$ to denote the set of finite tuples of elements from $X$, which we call *strings* when $X = 2$ or $X = \mathbb{N}$. For $\sigma \in X^{<\mathbb{N}}$ and $\tau \in X^{\leq\mathbb{N}}$, we use $\sigma^\frown\tau$ to denote the concatenation of these sequences. Similarly, for $x \in X$, $\sigma^\frown x$ is obtained by appending $x$ to $\sigma$. We will often omit the $^\frown$ symbol and just

[3] If $A = range(g)$, then $A$ is the domain of the partial function that, on input $m$, outputs the first $n$ with $g(n) = m$ if it exists.

[4] To prove the recursion theorem, for each $i$, let $g(i)$ be an index for the partial computable function $\varphi_{g(i)}(n) = f(\varphi_i(i), n)$. Let $e_0$ be an index for the total computable function $g$, and let $e = g(e_0)$. Then $\varphi_e(n) = \varphi_{g(e_0)} = f(\varphi_{e_0}(e_0), n) = f(g(e_0), n) = f(e, n)$.

write $\sigma\tau$ and $\sigma x$. We use $\sigma \subseteq \tau$ to denote that $\sigma$ is an initial segment of $\tau$, that is, that $|\sigma| \leq |\tau|$ and $\sigma(n) = \tau(n)$ for all $n < |\sigma|$. This notation is consistent with the subset notation if we think of a string $\sigma$ as its graph $\{\langle i, \sigma(i)\rangle : i < |\sigma|\}$. We use $\langle\rangle$ to denote the empty tuple. If $Y$ is a subset of the domain of a function $f$, we use $f \upharpoonright Y$ for the restriction of $f$ to $Y$. Given $f \in X^{\leq\mathbb{N}}$ and $n \in \mathbb{N}$, we use $f \upharpoonright n$ to denote the initial segment of $f$ of length $n$. We use $f \Vert n$ for the initial segment of length $n+1$. For a tuple $\bar{n} = \langle n_0, \dots, n_k\rangle \in \mathbb{N}^{<\mathbb{N}}$, we use $f \upharpoonright \bar{n}$ for the tuple $\langle f(n_0), \dots, f(n_k)\rangle$. Given a nested sequence of strings $\sigma_0 \subseteq \sigma_1 \subseteq \cdots$, we let $\bigcup_{i\in\mathbb{N}} \sigma_i$ be the possibly infinite string $f \in X^{\leq\mathbb{N}}$ such that $f(n) = m$ if $\sigma_i(n) = m$ for some $i$.

Given $f, g \in X^{\mathbb{N}}$, we use $f \oplus g$ for the function $(f \oplus g)(2n) = f(n)$ and $(f \oplus g)(2n+1) = g(n)$. We can extend this to *$\omega$-sums* and define $\bigoplus_{n\in\mathbb{N}} f_n$ to be the function defined by $(\bigoplus_{n\in\mathbb{N}} f_n)(\langle m, k\rangle) = f_m(k)$. Conversely, we define $f^{[n]}$ to be the $n$th column of $f$, that is, $f^{[n]}(m) = f(\langle n, m\rangle)$. All these definitions work for sets if we think in terms of their characteristic functions. So, for instance, we can encode countably many sets $\{A_n : n \in \mathbb{N}\}$ with one set $A = \{\langle n, m\rangle : m \in A_n\}$.

For a set $A \subseteq \mathbb{N}$, the complement of $A$ with respect to $\mathbb{N}$ is denoted by $A^c$.

A *tree* on a set $X$ is a subset $T$ of $X^{<\mathbb{N}}$ that is closed downward, i.e., if $\sigma \in T$ and $\tau \subseteq \sigma$, then $\tau \in T$ too. A *path* through a tree $T$ is a function $f \in X^{\mathbb{N}}$ such that $f \upharpoonright n \in T$ for all $n \in \mathbb{N}$. We use $[T]$ to denote the set of all paths through $T$. A tree is *well-founded* if it has no paths.

**Reducibilities.** There are various ways to compare the complexity of sets of natural numbers. Depending on the context or application, some may be more appropriate than others.

**Many-one reducibility.** Given sets $A, B \subseteq \mathbb{N}$, we say that $A$ is *many-one reducible* (or *m-reducible*) to $B$, and write $A \leq_m B$, if there is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $n \in A \iff f(n) \in B$ for all $n \in \mathbb{N}$. One should think of this reducibility as saying that all the information in $A$ can be decoded from $B$. Notice that the classes of computable sets and of c.e. sets are both closed downwards under $\leq_m$. A set $B$ is said to be *c.e. complete* if it is c.e. and, for every other c.e. set $A$, $A \leq_m B$.

Two sets are *m-equivalent* if they are $m$-reducible to each other, denoted $A \equiv_m B$. This is an equivalence relation, and the equivalence classes are called *m-degrees*

There are, of course, various other ways to formalize the idea of one set encoding the information from another set. Many-one reducibility is somewhat restrictive in various ways: (1) to figure out if $n \in A$, one is allowed to ask only one question of the form "$m \in B$?"; (2) the answer to "$n \in A$?" has to be the same as the answer to "$f(n) \in B$?". Turing reducibility is much more flexible.

**One-one reducibility.** 1-*reducibility* is like $m$-reducibility but requiring the reduction to be one-to-one. The equivalence induced by it, 1-*equivalence*, is one of the strongest notions of equivalence between sets in computability theory—a computability theorist would view sets that are 1-equivalent as being the same. Myhill's theorem states that two sets of natural numbers are 1-equivalent, i.e., each is 1-reducible to the other, if and only if there is a computable bijection of $\mathbb{N}$ that matches one set with the other.

**Turing reducibility.** Given a function $f : \mathbb{N} \to \mathbb{N}$, we say that a partial function $g : \mathbb{N} \rightharpoonup \mathbb{N}$ is *partial $f$-computable* if it can be computed by a program that is allowed to use the function $f$ as a primitive function during its computation; that is, the program can ask questions about the value of $f(n)$ for different $n$'s and use the answers to make decisions while the program is running. The function $f$ is called the *oracle* of this computation. If $g$ and $f$ are total, we write $g \leq_T f$ and say that $g$ is *Turing reducible* to $f$, that $f$ *computes* $g$, or that $g$ is *$f$-computable*. The class of partial $f$-computable functions can be enumerated the same way as the class of the partial computable functions. Programs that are allowed to query an oracle are called *Turing operators* or *computable operators*. We list them as $\Phi_0, \Phi_1, \ldots$, and we write $\Phi_e^f(n)$ for the output of the $e$th Turing operator on input $n$ when it uses $f$ as oracle. Notice that $\Phi_e$ represents a fixed program that can be used with different oracles. When the oracle is the empty set, we may write $\Phi_e$ for $\Phi_e^\emptyset$ matching the previous notation.

As we already mentioned, for a fixed input $n$, if $\Phi_e^f(n)$ converges, it does so after a finite number of steps $s$. As a convention, let us assume that in just $s$ steps, it is only possible to read the first $s$ entries from the oracle. Thus, if $\sigma$ is a finite substring of $f$ of length greater than $s$, we could calculate $\Phi_e^\sigma(n)$ without ever noticing that the oracle is not an infinite string.

> *Convention*: For $\sigma \in \mathbb{N}^{<\mathbb{N}}$, $\Phi_e^\sigma(n)$ is shorthand for $\Phi_{e,|\sigma|}^\sigma(n)$, which runs for at most $|\sigma|$ stages.

Notice that given $e, \sigma, n$, it is computable to decide if $\Phi_e^\sigma(n)\!\downarrow$.

As the class of partial computable functions, the class of partial $X$-computable functions contains the basic functions; is closed under composition, recursion, and minimization; can be listed in such a way that we have a universal partial $X$-computable function (that satisfies the s-m-n theorem). In practice, with very few exceptions, those are the only properties we use of computable functions. This is why almost everything we can prove about computable functions, we can also prove about $X$-computable functions. This translation is called *relativization*. All notions whose definition are based on the notion of partial computable function can be relativized by using the notion of partial $X$-computable function instead. For instance, the notion of c.e. set can be relativized to that of c.e. in $X$ or $X$-c.e. set:

These are the sets which are the images of $X$-computable functions (or empty), or, equivalently, the domains of partial $X$-computable functions. We use $W_e^X$ to denote the domain of $\Phi_e^X$.

When two functions are Turing reducible to each other, we say that they are *Turing equivalent*, which we denote by $\equiv_T$. This is an equivalence relation, and the equivalence classes are called *Turing degrees*.

Computable operators can be encoded by computable subsets of $\mathbb{N}^{<\mathbb{N}} \times \mathbb{N} \times \mathbb{N}$. Given $\Phi \subseteq \mathbb{N}^{<\mathbb{N}} \times \mathbb{N} \times \mathbb{N}$, $\sigma \in \mathbb{N}^{<\mathbb{N}}$, $n$, $m$, we write $\Phi^\sigma(n) = m$ as shorthand for $\langle \sigma, n, m\rangle \in \Phi$. Then, given $f \in \mathbb{N}^{\mathbb{N}}$, we let

$$\Phi^f(n) = m \iff (\exists \sigma \subset f)\, \Phi^\sigma(n) = m.$$

We then have that $g$ is computable in $f$ if and only if there is a c.e. subset $\Phi \subseteq \mathbb{N}^{<\mathbb{N}} \times \mathbb{N} \times \mathbb{N}$ such that $\Phi^f(n) = g(n)$ for all $n \in \mathbb{N}$. A standard assumption is that $\langle \sigma, n, m\rangle \in \Phi$ only if $n, m < |\sigma|$.

We can use the same idea to encode c.e. operators by computable subsets of $\mathbb{N}^{<\mathbb{N}} \times \mathbb{N}$. Given $W \subseteq \mathbb{N}^{<\mathbb{N}} \times \mathbb{N}$, $\sigma \in \mathbb{N}^{<\mathbb{N}}$, and $f \in \mathbb{N}^{\mathbb{N}}$, we let

$$W^\sigma = \{n \in \mathbb{N} : \langle \sigma, n\rangle \in W\} \quad \text{and} \quad W^f = \bigcup_{\sigma \subset f} W^\sigma.$$

We then have that $X$ is c.e. in $Y$ if and only if there is a c.e. subset $W \subseteq \mathbb{N}^{<\mathbb{N}} \times \mathbb{N}$ such that $X = W^Y$. A standard assumption is that $\langle \sigma, n\rangle \in W$ only if $n < |\sigma|$.

**Enumeration reducibility.** Recall that an enumeration of a set $A$ is just an onto function $f : \mathbb{N} \to A$. Given $A, B \subseteq \mathbb{N}$, we say that $A$ is *enumeration reducible* (or *e-reducible*) to $B$, and write $A \leq_e B$ , if every enumeration of $B$ computes an enumeration of $A$. Selman [Sel71] showed that we can make this reduction uniformly: $A \leq_e B$ if and only if there is a Turing operator $\Phi$ such that, for every enumeration $f$ of $B$, $\Phi^f$ is an enumeration of $A$. (See Theorem 4.2.2.) Another way of defining enumeration reducibility is via *enumeration operators*: An enumeration operator is a c.e. set $\Theta$ of pairs that acts as follows: For $B \subseteq \mathbb{N}$, we define

$$\Theta^B = \{n : (\exists D \subseteq_{fin} B)\, \langle \ulcorner D \urcorner, n\rangle \in \Theta\},$$

where $\subseteq_{fin}$ means 'finite subset of'. Selman also showed that $A \leq_e B$ if and only if there is an enumeration operator $\Theta$ such that $A = \Theta^B$.

The Turing degrees embed into the enumeration degrees via the map $\iota(A) = A \oplus A^c$. It is not hard to show that $A \leq_T B \iff \iota(A) \leq_e \iota(B)$.

**Positive reducibility.** We say that $A$ *positively reduces* to $B$, and write $A \leq_p B$, if there is a computable function $f : \mathbb{N} \to (\mathbb{N}^{<\mathbb{N}})^{<\mathbb{N}}$ such that, for every $n \in \mathbb{N}$, $n \in A$ if and only if there is an $i < |f(n)|$ such that every entry of $f(n)(i)$ is in $B$ (cf. [Joc68]). That is,

$$n \in A \iff \bigvee_{i<|f(n)|} \bigwedge_{j<|f(n)(i)|} f(n)(i)(j) \in B.$$

Notice that $\leq_p$ implies both Turing reducibility and enumeration reducibility, and is implied by many-one reducibility. In particular, the classes of computable sets and of c.e. sets are both closed downwards under $\leq_p$.

**The Turing jump.** Let $K$ be the domain of the universal partial computable function. That is,

$$K = \{\langle e, n\rangle : \Phi_e(n)\downarrow\} = \bigoplus_{e\in\mathbb{N}} W_e.$$

$K$ is called the *halting problem*.[5] It is not hard to see that $K$ is c.e. complete. Using a standard diagonalization argument, one can show that $K$ is not computable.[6] It is common to define $K$ as $\{e : \Phi_e(e)\downarrow\}$ instead—the two definitions give 1-equivalent sets. We will use whichever is more convenient in each situation. We will often write $0'$ for $K$.

We can relativize this definition and, given a set $X$, define the *Turing jump* of $X$ as

$$X' = \{e \in \mathbb{N} : \Phi_e^X(e)\downarrow\}.$$

Relativizing the properties of $K$, we get that $X'$ is $X$-c.e.-complete, that $X \leq_T X'$, and that $X' \not\leq_T X$. The Turing degree of $X'$ is strictly above that of $X$—this is why it is called a jump. The jump defines an operation on the Turing degrees. Furthermore, for $X, Y \subseteq \mathbb{N}$, $X \leq_T Y \iff X' \leq_m Y'$.

The double iteration of the Turing jump is denoted $X''$, and the $n$-th iteration by $X^{(n)}$.

**Vocabularies and languages.** Let us quickly review the basics about vocabularies and structures. Our vocabularies will always be countable. Furthermore, except for a few occasions, they will always be computable.

A *vocabulary* $\tau$ consists of three sets of symbols $\{\mathsf{R}_i : i \in I_R\}$, $\{\mathsf{f}_i : i \in I_F\}$, and $\{\mathsf{c}_i : i \in I_C\}$; and two functions $a_R\colon I_R \to \mathbb{N}$ and $a_F\colon I_F \to \mathbb{N}$. Each of $I_R$, $I_F$, and $I_C$ is an initial segment of $\mathbb{N}$. The symbols $\mathsf{R}_i$, $\mathsf{f}_i$, and $\mathsf{c}_i$ represent *relations*, *functions*, and *constants*, respectively. For $i \in I_R$, $a_R(i)$ is the arity of $\mathsf{R}_i$, and for $i \in I_F$, $a_F(i)$ is the arity of $\mathsf{f}_i$.

A vocabulary $\tau$ is *computable* if the arity functions $a_R$ and $a_F$ are computable. This only matters when $\tau$ is infinite; finite vocabularies are trivially computable.

Given such a vocabulary $\tau$, a $\tau$-*structure* is a tuple

$$\mathcal{M} = (M; \{R_i^{\mathcal{M}} : i \in I_R\}, \{f_i^{\mathcal{M}} : i \in I_F\}, \{c_i^{\mathcal{M}} : i \in I_C\}),$$

where $M$ is just a set called the *domain* of $\mathcal{M}$, and the rest are interpretations of the symbols in $\tau$. That is, $R_i^{\mathcal{M}} \subset M^{a_R(i)}$, $f_i^{\mathcal{M}}\colon M^{a_F(i)} \to M$, and $c_i^{\mathcal{M}} \in M$. A *structure* is a $\tau$-structure for some $\tau$.

[5] The 'K' is for Kleene.

[6] If it were computable, so would be the set $A = \{e : \langle e, e\rangle \notin K\}$. But then $A = W_e$ for some $e$, and we would have that $e \in A \iff \langle e, e\rangle \notin K \iff e \notin W_e \iff e \notin A$.

Given two $\tau$-structures $\mathcal{A}$ and $\mathcal{B}$, we write $\mathcal{A} \subseteq \mathcal{B}$ to mean that $\mathcal{A}$ is a substructure of $\mathcal{B}$, that is, that $A \subseteq B$, $f_i^{\mathcal{A}} = f_i^{\mathcal{B}} \restriction A^{a_F(i)}$, $R_j^{\mathcal{A}} = R_j^{\mathcal{B}} \restriction A^{a_R(i)}$ and $c_k^{\mathcal{A}} = c_k^{\mathcal{B}}$ for all symbols $\mathsf{f}_i$, $\mathsf{R}_j$ and $\mathsf{c}_k$. This notation should not be confused with $A \subseteq B$ which only means that the domain of $\mathcal{A}$ is a subset of the domain of $\mathcal{B}$. If $\mathcal{A}$ is a $\tau_0$-structure and $\mathcal{B}$ a $\tau_1$-structure with $\tau_0 \subseteq \tau_1$,[7] $\mathcal{A} \subseteq \mathcal{B}$ means that $\mathcal{A}$ is a $\tau_0$-substructure of $\mathcal{B} \restriction \tau_0$, where $\mathcal{B} \restriction \tau_0$ is obtained by forgetting the interpretations of the symbols of $\tau_1 \setminus \tau_0$ in $\mathcal{B}$. $\mathcal{B} \restriction \tau_0$ is called the $\tau_0$-*reduct* of $\mathcal{B}$, and $\mathcal{B}$ is said to be an *expansion* of $\mathcal{B} \restriction \tau_0$.

Given a vocabulary $\tau$, we define various languages over it. First, recursively define a $\tau$-*term* to be either a variable $x$, a constant symbol $\mathsf{c}_i$, or a function symbol applied to other $\tau$-terms, that is, $\mathsf{f}_i(t_1, \dots, t_{a_F(i)})$, where each $t_j$ is a $\tau$-term we have already built. The *atomic* $\tau$-*formulas* are the ones of the form $\mathsf{R}_i(t_1, \dots, t_{a_R(i)})$ or $t_1 = t_2$, where each $t_i$ is a $\tau$-term. A $\tau$-*literal* is either a $\tau$-atomic formula or a negation of a $\tau$-atomic formula. A *quantifier-free* $\tau$-*formula* is built out of literals using conjunctions, disjunctions, and implications. If we close the quantifier-free $\tau$-formulas under existential quantification, we get the *existential* $\tau$-*formulas*, of $\exists$-*formulas*. Every $\tau$-existential formula is equivalent to one of the form $\exists x_1 \cdots \exists x_k \, \varphi$, where $\varphi$ is quantifier-free. A *universal* $\tau$-*formula*, or $\forall$-*formula*, is one equivalent to $\forall x_1 \cdots \forall x_k \, \varphi$ for some quantifier-free $\tau$-formula $\varphi$. An *elementary* $\tau$-*formula* is built out of quantifier-free formulas using existential and universal quantifiers. We also call these the *finitary first-order formulas*.

Given a $\tau$ structure $\mathcal{A}$, and a tuple $\bar{a} \in A^{<\mathbb{N}}$, we write $(\mathcal{A}, \bar{a})$ for the $\tau \cup \bar{\mathsf{c}}$-structure where $\bar{\mathsf{c}}$ is a new tuple of constant symbols and $\bar{\mathsf{c}}^{\mathcal{A}} = \bar{a}$. Given $R \subseteq \mathbb{N} \times A^{<\mathbb{N}}$, we write $(\mathcal{A}, R)$ for the $\widetilde{\tau}$ structure where $\widetilde{\tau}$ is defined by adding to $\tau$ relations symbols $\mathsf{R}_{i,j}$ of arity $j$ for $i, j \in \mathbb{N}$, and $\mathsf{R}_{i,j}^{\mathcal{A}} = \{\bar{a} \in A^j : \langle i, \bar{a} \rangle \in R\}$.

**Orderings.** Here are some structures we will use quite often in examples. A *partial order* is a structure over the vocabulary $\{\leq\}$ with one binary relation symbol which is transitive ($x \leq y \ \& \ y \leq z \to x \leq z$), reflexive ($x \leq x$), and anti-symmetric ($x \leq y \ \& \ y \leq x \to x = y$). A *linear order* is a partial order where every two elements are comparable ($\forall x, y \ (x \leq y \vee y \leq x)$). We will often add and multiply linear orderings. Given linear orderings $\mathcal{A} = (A; \leq_A)$ and $\mathcal{B} = (B; \leq_B)$, we define $\mathcal{A} + \mathcal{B}$ to be the linear ordering with domain $A \sqcup B$, where the elements of $A$ stand below the elements of $B$. We define $\mathcal{A} \times \mathcal{B}$ to be the linear ordering with domain $A \times B$ where $\langle a_1, b_1 \rangle \leq_{A \times B} \langle a_2, b_2 \rangle$ if either $b_1 <_B b_2$ or $b_1 = b_2$ and $a_1 \leq_A a_2$—notice we compare the second coordinate first.[8] We will use $\omega$ to denote the linear ordering of the natural numbers and $\mathbb{Z}$ and $\mathbb{Q}$ for the orderings of the integers and the rationals. We denote the finite linear

[7] By $\tau_0 \subseteq \tau_1$ we mean that every symbol in $\tau_0$ is also in $\tau_1$ and with the same arity.
[8] $\mathcal{A}$ times $\mathcal{B}$ is $\mathcal{A}$ $\mathcal{B}$ times.

ordering with $n$ elements by $\boldsymbol{n}$. We use $\mathcal{A}^*$ to denote the reverse ordering $(A; \geq_A)$ of $\mathcal{A} = (A, \leq_A)$. For $a <_{\mathcal{A}} b \in \mathcal{A}$, we use the notation $\mathcal{A} \upharpoonright (a, b)$ or the notation $(a, b)_{\mathcal{A}}$ to denote the open $\{x \in A : a <_{\mathcal{A}} x <_A b\}$. We also use $\mathcal{A} \upharpoonright a$ to denote the initial segment of $\mathcal{A}$ below $a$, which we could also denote as $(-\infty, a)_{\mathcal{A}}$.

As mentioned above, a *tree* $T$ is a downward closed subset of $X^{<\mathbb{N}}$. As a structure, a tree can be represented in various ways. One is as a partial order $(T; \subseteq)$ using the ordering on strings. Another is as a graph where each node $\sigma \in T$ other than the root is connected to its parent node $\sigma \upharpoonright |\sigma - 1|$, and there is a constant symbol used for the root of the tree. We will refer to these two types of structures as *trees as orders* and *trees as graphs*.

A partial order where every two elements have a least upper bound $(x \vee y)$ and a greatest lower bound $(x \wedge y)$ is called a *lattice*. A lattice with a top element 1, a bottom element 0, and where $\vee$ and $\wedge$ distribute over each other, and every element $x$ has a *complement* (that is an element $x^c$ such that $x \vee x^c = 1$ and $x \wedge x^c = 0$) is called a *Boolean algebra*. The vocabulary for Boolean algebras is $\{0, 1, \vee, \wedge, \cdot^c\}$, and the ordering can be defined by $x \leq y \iff y = x \vee y$.

**The arithmetic hierarchy.** Consider the structure $(\mathbb{N}; 0, 1, +, \times, \leq)$. In this vocabulary, the *bounded formulas* are built out of the quantifier-free formulas using bounded quantifiers of the form $\forall x < y$ and $\exists x < y$. A $\Sigma^0_1$ formula is one of the form $\exists x\, \varphi$, where $\varphi$ is bounded; and a $\Pi^0_1$ formula is one of the form $\forall x\, \varphi$, where $\varphi$ is bounded. By coding tuples of numbers by a single natural number, one can show that formulas of the form $\exists x_0 \exists x_1 \ldots \exists x_k\, \varphi$ are equivalent to $\Sigma^0_1$ formulas. Post's theorem asserts that a set $A \subseteq \mathbb{N}$ is c.e. if and only if it can be defined by a $\Sigma^0_1$ formula. Thus, a set is computable if and only if it is $\Delta^0_1$, that is, if it can be defined both by a $\Sigma^0_1$ formula and by a $\Pi^0_1$ formula.

By recursion, we define the $\Sigma^0_{n+1}$ formulas as those of the form $\exists x\, \varphi$, where $\varphi$ is $\Pi^0_n$; and the $\Pi^0_{n+1}$ formulas as those of the form $\forall x\, \varphi$, where $\varphi$ is $\Sigma^0_n$. A set is $\Delta^0_n$ if it can be defined by both a $\Sigma^0_n$ formula and a $\Pi^0_n$ formula. Again, in the definition of $\Sigma^0_{n+1}$ formulas, using one existential quantifier or many makes no difference. What matters is the number of alternations of quantifiers. Post's theorem asserts that a set $A \subseteq \mathbb{N}$ is c.e. in $0^{(n)}$ if and only if it can be defined by a $\Sigma^0_{n+1}$ formula. In particular, a set is computable from $0'$ if and only if it is $\Delta^0_2$. The Shoenfield *Limit Lemma* says that a set $A$ is $\Delta^0_2$ if and only if there is a computable function $f : \mathbb{N}^2 \to \mathbb{N}$ such that, for each $n \in \mathbb{N}$, if $n \in A$ then $f(n, s) = 1$ for all sufficiently large $s$, and if $n \notin A$ then $f(n, s) = 0$ for all sufficiently large $s$. This can be written as $\chi_A(n) = \lim_{s \to \infty} f(n, s)$, where $\chi_A$ is the characteristic function of $A$ and the limit with respect to the discrete topology of $\mathbb{N}$ where a sequence converges if and only if it is eventually constant.

The language of second-order arithmetic is a two-sorted language for the structure $(\mathbb{N}, \mathbb{N}^{\mathbb{N}}; 0, 1, +, \times, \leq)$. The elements of the first sort, called *first-order elements*, are natural numbers. The elements of the second sort, called *second-order elements* or *reals*, are functions $\mathbb{N} \to \mathbb{N}$. The vocabulary consists of the standard vocabulary of arithmetic, $0$, $1$, $+$, $\times$, $\leq$ which is used on the first-order elements, and an application operation denoted $F(n)$ for a second-order element $F$ and a first-order element $n$. A formula in this language is said to be *arithmetic* if it has no quantifiers over second-order objects. Among the arithmetic formulas, the hierarchy of $\Sigma^0_n$ and $\Pi^0_n$ formulas are defined exactly as above. Post's theorem that $\Sigma^0_1$ sets are c.e. also applies in this context: For every $\Sigma^0_1$ formula $\psi(F, n)$, where $n$ a number variable and $F$ is a function variable, there is c.e. operator $W$ such that $n \in W^F \iff \psi(F, n)$. We can then build the computable tree $T_n = \{\sigma \in \mathbb{N}^{<\mathbb{N}} : n \notin W^\sigma\}$ and we have that $\psi(F, n)$ holds if and only if $F$ is not a path through $T_n$. A $\Pi^0_1$ *class* is a set of the form $\{F \in \mathbb{N}^{\mathbb{N}} : \psi(F)\}$ for some $\Pi^0_1$ formula $\psi(F)$. The observation above shows how every $\Pi^0_1$ class is of the form $[T]$ for some computable tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$.