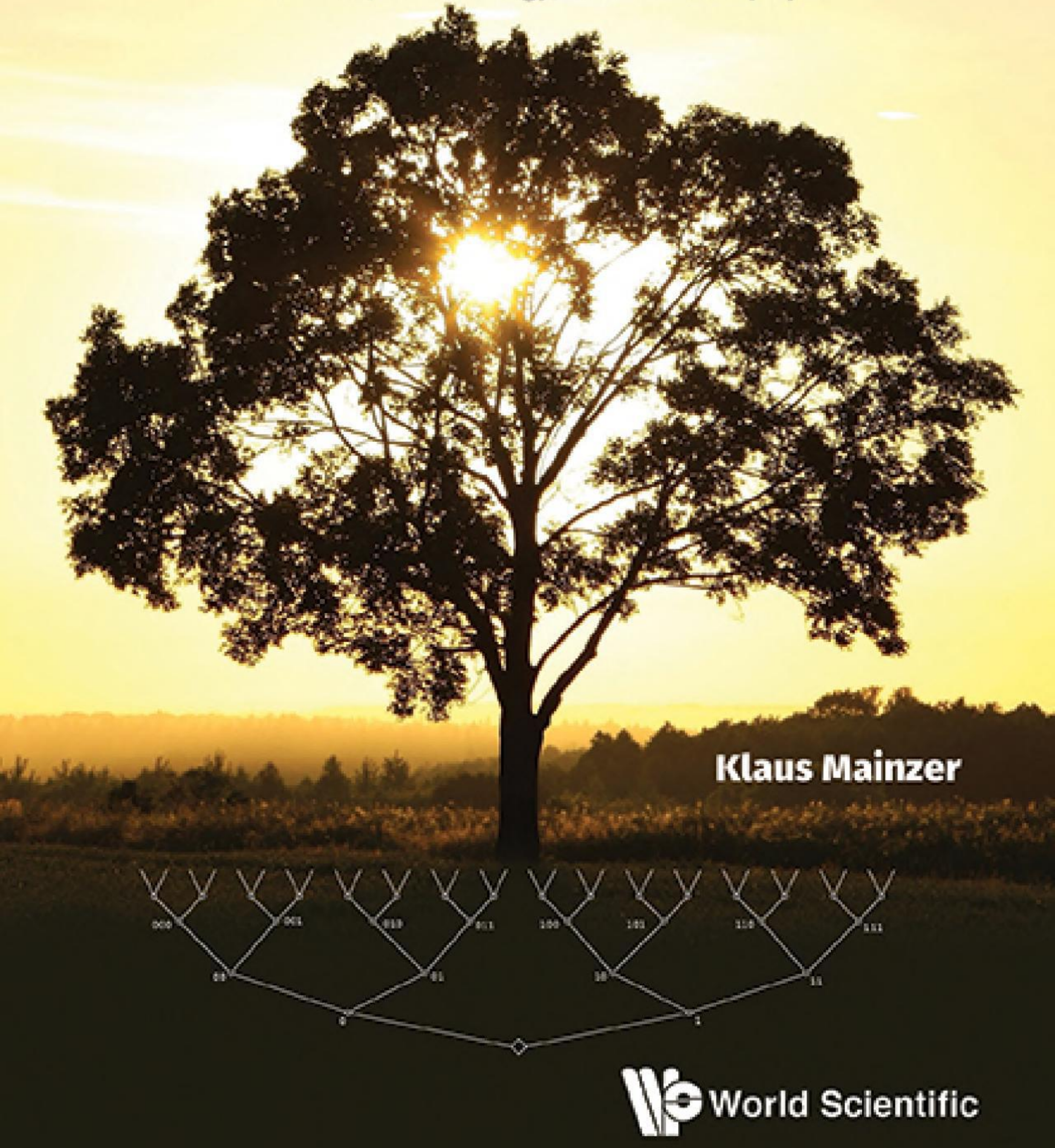
The Digital and the Real World

Computational Foundations of Mathematics, Science, Technology, and Philosophy



Klaus Mainzer

World Scientific

The Digital and the Real World

Computational Foundations of Mathematics, Science, Technology, and Philosophy

Klaus Mainzer
Technical University of Munich, Germany

World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI • TOKYO

Prof. em. Dr. Klaus Mainzer
Emeritus of Excellence
Graduate School of Computer Science
Technical University of Munich
Germany
e-mail: mainzer@tum.de

Published by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

Library of Congress Cataloging-in-Publication Data
Names: Mainzer, Klaus, author.
Title: The digital and the real world : computational foundations of mathematics, science, technology, and philosophy / by Klaus Mainzer (Technical University of Munich, Germany).
Description: New Jersey : World Scientific, 2017. | Includes bibliographical references.
Identifiers: LCCN 2017023302 | ISBN 9789813225480 (hc : alk. paper)
Subjects: LCSH: Logic, Symbolic and mathematical. | Mathematics--Philosophy.
Classification: LCC QA8.4 .M35 2017 | DDC 510.1--dc23
LC record available at https://lccn.loc.gov/2017023302

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

For any available supplementary material, please visit
http://www.worldscientific.com/worldscibooks/10.1142/10583#t=suppl

Desk Editors: V. Vishnu Mohan/Tan Rok Ting

Typeset by Stallion Press
Email: enquiries@stallionpress.com

Printed in Singapore

Contents

Chapter 1

Introduction

Historically, the theory of algorithms and computability started in the beginning of the last century with foundational debates on logics, mathematics, and philosophy. Therefore, the *digital world* has its origins in logical, mathematical and philosophical theories.

Alan Turing anticipated the development of the digital general purpose computer by his famous Turing machine (Turing, 1936). Here, we have a finite state control device with a read/write head and a two-way unlimited tape consisting of an unlimited number of cells. The control device is regulated by a program which is a finite set of instructions with states of the machine, operations for printing 0, 1 and * (blank) into the cells, moving left or right one cell, and stopping the tape.

A number-theoretic function is defined to be computable if it is the input–output function of a Turing machine. Further on, a set of natural numbers is decidable if there is an effective procedure (characteristic function) that given any element of the set will decide in a finite number of steps whether or not the element is in the set. In short, for decidability of a set, its characteristic function must be Turing computable.

All number-theoretic propositions can be coded with the digits 0 and 1. Thus, Turing's theory of computability in computer science is reduced to the digital paradigm. The Turing machine is mathematically equivalent to many other effective procedures (e.g., register machines, recursive number-theoretic functions). Therefore, Church's thesis demands that Turing-computability means (digital) computability in general.

But, there seems to be a *deep gap between digital computer science and mathematics.* Many mathematical disciplines like the calculus heavily depend on the continuous nature of real numbers. In (classical) physics, the dynamics of nature is modeled by continuous curves and differential equations. Their solutions correspond to physical events which can be predicted or explained as causal effects under certain constraints. At least in classical physics, nature is assumed to be continuous, and even quantum physics uses real and complex-valued quantities of fundamental laws of continuous differential equations (e.g., Schrödinger equation).

Our perceptions of the world seem to be continuous: Our bodily sensors receive analog signals of, e.g., electromagnetic and acoustic waves. Behind all that, there is an old and deep conflict in philosophy between the continuous and discrete characteristics of nature since the Antiqutity. Is matter continuous (Aristotle) or atomic (Democritus)? Nowadays, in technology, we distinguish bewteen digital (discrete) and analog (continuous) signal processing.

Before the digital paradigm of computer science started with the Turing machine, there was already an old tradition of *real algorithms in mathematics.* Since antiquity, undecidability problems of geometrical constructions (e.g., trisection of an angle) were discussed (Mainzer, 1980). The final answer was found in the beginning of the 19th century by Galois's result on the non-solvability by radicals of polynomial equations of degree 5 or more. These algorithms manipulate real numbers. There is a long standing tradition of decidability results in algebra and analysis leading to modern numerical analysis. We remind the reader of Newton's method for finding (approximate) zeros of polynomials in one variable. Newton's method was one of the first search algorithms in numerical analysis and scientific computing.

Nowadays, *practical algorithms* to solve equations of physics, models of climate, weather prediction, or financial mathematics mainly refer to numerical analysis. Therefore, we not only need logical–mathematical foundations of digital computing (Turing, 1936) but also foundational studies of real algorithms. They have a long tradition in mathematics from Newton, Gauss, Euler, etc., to modern numerical analysis and scientific computing.

What do real computability and decidability mean? It is amazing that, from a logical point of view, the digital world of natural numbers seems to be much more complicated than the continuous world of real analysis, although counting digits is easily done by children and the naturals numbers are a subset of the real numbers: But, according to Gödel's famous proof, arithmetic is incomplete, and a closed real field is decidable.

How far can we transfer the results of computability and decidability from digital to real mathematics? These results have consequences for the discrete (digital) and the analog (continuous and real) and with that to the modern analog world of sensor technology and human experience by analog sensors.

Computability of algorithms in computer science is also deeply linked with the *provability of theorems in mathematics*. Since antiquity, truth of mathematical theorems was verified by logical proofs in axiomatic systems. Logical-axiomatic proofs became the paradigm in science and philosophy (Hilbert, 1918). In the 20th century, the question arose whether the true propositions of a theory can be described completely, correctly, and consistently in formal systems. Famous logicians, mathematicians, and philosophers of the 20th century (e.g., Hilbert, Gödel, and Turing) demonstrated the possibilities and limitations of formalization.

It is well known that mathematical proofs sometimes only guarantee the existence of a solution without providing an effective algorithm and constructive problem-solving procedure. The reason is that mathematics as it is normally practised is based on the law of excluded middle: Either a statement is true or its negation is true. According to this law, it is sufficient to show indirectly that the assumption of non-existence of a solution is false (Mainzer, 1970). But, for practical applications, it is, of course, a great disadvantage that indirect proofs do not tell us how to find a constructive solution step by step (Bishop, 1967).

From a digital point of view, constructive proofs should be realized by digital systems. If mathematics is restricted to computable functions of natural numbers, Turing machines will do the job. But, in higher mathematics and physics (e.g., functional analysis), we have to consider functionals and spaces of higher types. Therefore, a general concept of a digital information system is necessary to compute finite approximations of functionals of higher types. Contrary to the machine orientation of computational mathematics, intuitionistic mathematics (INT) is rooted in the philosophy of human creativity. Mathematics is understood as a human activity of constructing and proving step by step. In the rigorous sense of Brouwer's intuitionism, we even get a concept of real continuum and infinity which differs from ordinary understanding of classical mathematics. In the foundational research programs of proof mining and reverse mathematics, we offer degrees of constructivity and provability to classify problem solving and proving for different mathematical applications.

Proof mining has the aim to obtain the missing information in an incomplete theorem and to extract effective procedures by a purely logical analysis of mathematical proofs and principles (Kohlenbach, 2008). Sometimes, it is even sufficient to find bounds for search processes of problem solving. At least, proof mining tells us how far away a proof is from being constructive. The research program of proof mining goes back to Georg Kreisel's proof-theoretic research. He illustrates the extraction of effective procedures from proofs as "unwinding proofs" (Feferman, 1996).

From a theoretical point of view, proof mining is an important link between logic, mathematics, and computer science. Logic is no longer only a formal activity besides and separated from mathematics. Actually, metatheorems of proof mining deliver *logical tools to solve mathematical problems* more effectively and to obtain new mathematical information: Proofs are more than verification of theorems!

From a practical point of view, proof theory also has practical consequences in applied computer science. In this case, instead of formal theories and proofs, we consider formal models and computer programs of processes, e.g., in industry. Formal derivations of formula correspond to, e.g., steps of industrial production. In order to avoid additional costs of mistakes and biases, we should test and prove that the *computer programs* are correct and sound before applying them. *Automated theorem proving* is applied to integrated designs and verification. Software and hardware designs must be verified to prevent costly and life-threatening errors. Dangerous examples in the past were flaws in the microchips of space rockets and medical equipments. Network security of the Internet is a challenge for banks, governments, and commercial organizations. In the age of Big Data and increasing complexity of our living conditions, rigorous proofs and tests are urgently demanded to avoid a dangerous future with overwhelming computational power running out of control.

But, in real mathematics, proofs cannot all be reduced to effective procedures. Actually, there are *degrees of constructivity, computability, and provability*. How strong must a theory be in order to prove a certain theorem? Therefore, we try to determine which axioms are required to prove a theorem. How constructive and computational are these assumptions?

Instead of going forward from axioms to theorems in usual proofs, we prove in the reverse way (backward) from a given theorem to the assumed axioms. Therefore, this research program is called *reverse mathematics* (Friedman, 1975). If an axiom system S proves a theorem T and theorem T together with an axiom system S' (the reversal) prove axiom system S, then S is called equivalent to theorem T over S'. In order to determine the degrees of constructivity and computability, one tries to characterize mathematical theorems by equivalent subsystems of arithmetic. The formulas of these arithmetic subsystems can be distinguished by different degrees of complexity.

In (second-order) arithmetic, all objects are represented as natural numbers or sets of natural numbers. Therefore, proofs about real numbers must use Cauchy sequences of rational numbers which can be represented as sets of natural numbers. In reverse mathematics, theorems and principles of real mathematics are characterized by equivalent subsystems of arithmetic with different degrees of constructivity and computability (Ishihara, 2005). Some of them are computable in Turing's sense, but others are definitely not and need stronger tools beyond Turing computability. Therefore, it is proved that only parts of real mathematics can be reduced to the digital paradigm. Obviously, reverse mathematics and proof mining are important research programs to clarify and deepen the connections between mathematics, computer science, and logic in a rigorous way.

Another important bridge between logic, mathematics, and computer science is type theory. Types of data (resp., terms) are distinguished in computer programs of computer science as well as in formal systems of mathematics, in order to avoid software bugs (resp., logical paradoxes). Martin-Löf's *intuitionistic type theory* offers both a philosophical foundation of constructive mathematics as well as a proof assistant in computer science (e.g., Coq). Recently, homotopy type theory extends formalization from set-theoretical objects up to categories related to more and more large cardinals. *Homotopy type theory* (HoTT) tried to develop a *universal* (''*univalent*'') *foundation of mathematics* as well as computer languages with respect to proof assistants for advanced mathematical proofs. The question arises how far univalent foundations of mathematics can be constructive.

Philosophically, constructive foundations of mathematics are deeply rooted in the epistemic tradition of efficient reasoning with the *principle of parsimony* in explanations, proofs, and theories. It was the medieval logician and philosopher William of Ockham (1285–1347) who first demanded that one should prefer explanations and proofs with the fewest number of assumed abstract concepts and principles. The reason is that, according to Ockham, universals (e.g., mathematical sets) are only abstractions from individuals (the elements of a set) without real existence. Ockham's principle of parsimony later became popular as ''Ockham's razor'' reducing abstract principles as far as possible.

The question arises how far reduction is possible without loss of essential information. In order to analyze real computation in mathematics, we need an extension of *digital computability beyond Turing computability. Real computing machines* can be considered idealized analog computers accepting real numbers as given infinite entities (Blum *et al.*, 1989). They are not only theoretically interesting but also practically inspiring with respect to their applications in scientific computing and technology (e.g., sensor technology).

These aspects are also exciting for *computational neuroscience* and *cognitive science*. Appropriate neural networks or cellar automata are computationally equivalent to Turing machines. Further on, we can distinguish a hierarchy of automata and machines which can realize formal languages with increasing complexity. Examples are finite automata accepting regular languages or Turing machines accepting Chomsky grammars of natural languages. Actually, (recurrent) neural networks with rational numbers as synaptic weights are computationally equivalent to Turing machines accepting recursive languages like Chomsky grammars. It can be proven that (recurrent) neural networks with non-computable real weights can even operate on non-recursive languages (Siegelmann, 1994). Human brains also operate on non-recursive natural languages. If analog neural networks are considered as models of human brains, they seem to be more powerful than digital Turing machines.

The foundational debate on the digital and analog, discrete and continuous also has deep consequences for *physics*. In classical physics, the real numbers are assumed to correspond to continuous states of physical reality. For example, electromagnetic or gravitational fields are mathematically modeled by continuous manifolds. Thus, "real computing" (i.e., computing with real numbers) seems to model effective procedures in a continuous reality. Fluid dynamics illustrates this paradigm with continuous differential equations.

But, in modern quantum physics, a "coarse grained" reality seems to be more appropriate. Quantum systems are characterized by discrete quantum states which can be defined as quantum bits. Instead of classical information systems, following the digital concept of a Turing machine, quantum information systems with quantum algorithms and quantum bits open new avenues to *digital physics*. *Information* and *information systems* are no longer fundamental categories only of information theory, computer science, and logic but also of *physics* (Mainzer, 2016b).

But, is it possible to reduce the real world to a quantum computer as an extended concept of a universal quantum Turing machine? ‘‘It from bit’’ proclaimed physicist John A. Wheeler (1990). On the other side, fundamental symmetries of physics (e.g., Lorentz symmetry and electroweak symmetry) are continuous (Audretsch and Mainzer, 1996; Mainzer, 2005). Einstein’s space–time is also continuous. Are they only abstractions (in the sense of Ockham) and approximations to a discrete reality?

Some authors proclaim a discrete cosmic beginning with an initial quantum system (e.g., quantum vacuum) evolving in an expanding macroscopic universe with increasing complexity which can be approximately modeled by continuous mathematics. In this case, physical reality is discrete and continuous mathematics with real numbers only a (ingenious) human invention. But physical laws including quantum theory (e.g., Hilbert space) are infused with real numbers and the mathematics of the continuum. Thus, from an extreme Platonic point of view, we could also assume a layer much deeper than the discrete quantum world — the universe of mathematical structures themselves as primary reality with infinity and continuity.

Anyway, besides all ontological speculations, mathematics is fundamental for science and cannot be reduced to the digits and the discrete nature of computer science. We should distinguish *degrees of constructivity*, *computability*, *and provability* in the rigorous sense of *real computing*, *proof mining*, and *reverse and univalent mathematics.*

In this case, Stephen Wolfram’s vague concept of computational equivalence can be made precise and corrected (Wolfram, 2002). He demanded that the natural processes of atomic, molecular, and cellular systems should be considered as ‘‘computationally equivalent’’ procedures of Turing machines, cellular automata, and neural networks (in the sense of a physically extended Church’s thesis). This idea already came up with John von Neumann’s and Konrad Zuse’s cellular automata (Zuse, 1969). At least, the mathematical models and theories of these natural systems can be distinguished by different degrees of complexity. Therefore, we need the corresponding mathematical theories and models with their equations and laws in order to derive reliable predictions, classifications, and explanations. Quasi-empirical computer experiments in the sense of Wolfram, even with powerful computers and algorithms, are not sufficient.

The exponentially increasing power of supercomputers and global networks is overwhelming. Success and efficiency in science and economy seem to only depend on fast algorithms and huge databases. In economy, they rapidly predict future trends and profiles of products and customers. In science, they recognize correlations and patterns in huge collections of data (e.g., elementary particle colliders in high energy physics, machine learning algorithms in molecular biology). Science (like economy) seems to be more and more driven by fast algorithms and a huge amount of data: *Big Data –The end of theory*? asked Chris Anderson, an influential publisher of digital media.

Wolfram even proclaimed a "new kind of science", in which *computer experiments* would replace mathematical proofs and theories. Wolfram had simulated extensive pattern formations of cellular automata on high performance computers, discovered some remarkable correlations, and classified the patterns based on his observations. But one can prove that only the fundamental mathematical laws of cellular automata allow accurate forecasting and classification of patterns (Mainzer and Chua, 2011).

This argument can be extended to the emergence of patterns in nature, taking in physics, chemistry, biology, and brain research (Mainzer and Chua, 2013). Here too, we found that it was only when the basic equations were known that accurate declarations and forecasts could be made on the emergence of structure and patterns. What we can generally say about science is that theory is often the best way to solve a problem. How will a mountain of data help me if I don't know what I am looking for?

Correlation cannot replace causation. Causation is explained by causal laws which are mathematically represented by equations of dynamical systems. Quasi-empirical computer experiments with Big Data may be helpful for a rough orientation. But, we need a deep analysis of the computational foundations in mathematics, science, technology and philosophy for reliable tools of problem solving in a digital world. In the end, we need *well-founded mathematical theories* with different *degrees of constructivity*, *computability*, *and provability*.

This book starts with "Basics of Computability" (Chapter 2) from Turing's theory of computability to decidable and undecidable problems, followed by "Hierarchies of Computability" (Chapter 3) with computational degrees and complexity of problem solving. Chapter 4 explains "Constructive Proof Theory" with different degrees of constructive proofs. Chapters 2–4 consider computability and provability on the basis of elementary number theory. In order to prepare proof mining in higher mathematics, Turing-computable functions are no longer sufficient. What do computable functionals of higher type (e.g., functions or sets of number-theoretic functions) mean? Chapter 5 "Computational Mathematics and Digital Information Systems" introduces a general concept of information system beyond Turing computability. The question arises how far the corresponding functionals can be approximated by constructive procedures.

Chapter 6 considers the foundations of "Intuitionistic Mathematics and Human Creativity". INT is not only interesting because of its rigorous ban of the law of excluded middle. Brouwer's philosophical understanding of mathematical constructing and proving leads to a different concept of real continuum and infinity. His fan theorem and bar theorem are consequences of the intuitionistic philosophy. In the past, intuitionism and classical mathematics often fought against one another as different ideological schools. But, independent of ideological points of view, these principles can be considered as additional axioms which may be accepted or not like all axioms or hypotheses in science. Anyway, independent of their philosophical meaning, we can prove mathematical implications and equivalent theorems of these principles in order to classify mathematical theorems and theories according to their degrees of constructivity and provability.

A main motivation of this book is to overcome the gaps between logic, mathematics, and computer science. Nowadays, logic is often considered as a discipline of only theoretical and philosophical interest with no practical relevance for ordinary mathematics and its applications. In Chapters 7 and 8, we study two foundational research programs bridging the gap between theory and application, philosophy and problem solving.

Chapter 7 is dedicated to ‘‘Proof Mining bridging Logic, Mathematics, and Computer Science’’. Starting with extractions of effective information from proofs in elementary number theory, proof mining is extended to, e.g., numerical analysis and functional analysis which are important for practical applications. The main idea is that proofs can be characterized by more or less constructive and computable functionals. Actually, these functionals can be considered as elements of information systems with different degrees of complexity below or beyond Turing computability. Historically, Gödel started with the class of primitive recursive functionals which was extended in his Dialectica interpretation. Later on, a variety of similar interpretations were studied to analyze proofs in different mathematical theories. Sometimes, functional interpretations make the extraction of effective computer programs possible to realize proofs automatically. There are interactive proof assistants such as Coq (Bertot and Castéran, 2004), HOL, Isabelle (Nipkow *et al.*, 2002), or MINLOG (Schwichtenberg, 2006). We will consider some applications of MINLOG because of its natural understanding of constructive logic. In general, automated theorem proving has deep technical and societal impact in order to prevent errors and flaws in software and hardware design, in the Internet and global communication (cf. Chapter 16). This is an important breakthrough to link proof theory with computer science and mathematics.

The link between mathematics, proof theory, and computer science is also supported by the research program of ‘‘Reverse Mathematics bridging Logic, Mathematics, and Computer Science’’, which is considered in Chapter 8. Reverse mathematics allows one to determine the proof-theoretic strength, degree of computability and complexity of theorems by classifying them with respect to equivalent theories and proofs. Many theorems of classical mathematics can be classified by subsystems of second-order arithmetic $\mathbb{Z}_2$ with variables of natural numbers and variables of sets and functions of natural numbers.

Chapter 9 considers the development "From Intuitionistic to Homotypy Type Theory — Bridging Logic, Mathematics, and Computer Science". Actually, types of data resp. terms are used in programming languages as well as in formal systems of mathematics in order to avoid software failures resp. logical paradoxes. An important evolution taking place in current mathematics is the transition into a new era of automated tools for proof construction and verification which are known as proof assistants. Martin-Löf intuitionistic type theory was an important step to the most powerful proof assistants (such as Coq). An exciting development is the recent *univalent foundations project* of the Institute for Advanced Study (Princeton) which provides new semantics for type theories using notions from geometry and topology up to abstract mathematical categories. The key question is how far constructive proof procedures have the effect of extending the validity of mathematical reasoning to the widest possible context.

Chapter 10, "Real Computability and Real Analysis", transfers the digital concepts of computability and decidability to the continuous world of real numbers. Decision machines and search procedures are introduced over real and complex numbers. A universal machine (generalizing Turing's universal machine in the digital world) can be defined over a mathematical ring (and with that over real and complex numbers). On this basis, decidability and undecidability of several theoretical and practical mathematical problems (e.g., Mandelbrot set, Newton's method) can rigorously be proved.

Like in the digital world, we can introduce a "Complexity Theory of Real Computing" (Chapter 11). Polynomial time reductions as well as the class of NP problems are studied over a general mathematical ring (and with that on real and complex numbers). Real computability can be extended in a polynomial hierarchy for unrestricted machines over a mathematical field.

Chapter 12 discusses the consequences of "Real Computing and Neural Networks". Neural networks are models of natural brains in evolution with different degrees of complexity. Mathematically, they are equivalent to computing machines with different degrees of complexity from finite automata to Turing machines. Otherwise, the simulation of neural networks on digital computers would not be possible. Computing machines can understand formal languages according to their degree of computability from simple regular languages (finite automata) to Chomsky grammars of natural languages (Turing machines). Analog neural networks with real synaptic weights are beyond digital Turing computability. They correspond to real computing. Actually, they can operate on non-computable languages like human brains.

Brains and computers are examples of information processing machines. What does information mean? Chapter 13 distinguishes the "Complexity of Algorithmic Information". Chaitin's concept of algorithmic information is closely connected with Gödel's incompleteness and Turing's halting problem of Turing machines (Chaitin, 2007). Laws and theories can be understood as algorithmic information compression. But Chaitin's algorithmic information is reduced to the digital world.

If we consider dynamical systems in the "real" world of physics, chemistry or biology, we need concepts of continuous information flow which are explained in Chapter 14, "Complexity of Information Dynamics". Basic information measures from Shannon's information entropy to Kolmogorov–Sinai entropy are distinguished. How can continuous dynamics be related to discrete symbolic dynamics?

Chapter 15 "Digital and Real Physics," considers the foundations of digital physics. In this case, the universe itself is assumed to be a gigantic information system with quantum bits as elementary states. Deutsch (1985) discussed the quantum versions of Turing computability and Church's thesis. They are interesting for quantum computers, but still reduced to the digital paradigm. Obviously, modeling the "real" universe also needs real computing. Are continuous models only useful approximations of an actually discrete reality?

With Chapter 16, the book ends with "Digital and Real Computing in the Social World". The chapter starts with the question why it is so difficult to model the social world. Economics and financial mathematics are highly mathematized, but they miss the success of explanation and predictions of mathematical natural sciences. We consider the degrees of constructivity, provability, and computability of fundamental economic and financial theorems in the sense of reverse mathematics. Ockham's principle of parsimony is actually a rule of economic efficiency in theory building: The degree of theoretical abstractions corresponds to the ‘‘price’’ we are willing to pay for a reliable tool of problem solving.

Scientific modeling of complex social systems is extended to global information and computer networks of Internet of Things (IoT) with Big Data. These complex networks are digital with their apps and supercomputers and also analog with trillions of sensors. Their mathematical modeling is a great challenge of scientific computing. It is also deeply connected with real computing, proof mining and reverse mathematics. Chapter 17 is with a philosophical outlook and a plea for more foundational research in an age of exponentially growing technologies.

Chapter 2

Basics of Computability

The early historical roots of computer science can be found in age of classical mechanics. The mechanization of thoughts begins with the invention of mechanical devices for performing elementary arithmetic operations automatically. A mechanical calculation machine executes serial instructions step by step. In general, the traditional design of a mechanical calculation machine contains the following devices.

First, there is an input mechanism by which a number is entered into the machine. A selector mechanism selects and provides the mechanical motion to cause the addition or subtraction of values on the register mechanism. The register mechanism is necessary to indicate the value of a number stored within the machine, technically realized by a series of wheels or disks. If a carry is generated because one of the digits in the result register advances from 9 to 0, then that carry must be propagated by a carry mechanism to the next digit or even across the entire result register. A control mechanism ensures that all gears are properly positioned at the end of each addition cycle to avoid false results or jamming the machine. An erasing mechanism has to reset the register mechanism to store a value of zero.

Wilhelm Schickard (1592–1635), Professor of Hebrew, oriental languages, mathematics, astronomy, and geography, is presumed to be the first inventor of a mechanical calculating machine for the first four rules of arithmetic. The adding and subtracting part of his machine is realized by a gear drive with an automatic carry mechanism. The multiplication and division mechanism is based on Napier's multiplication tables. Blaise Pascal (1623–1662), the brilliant French mathematician and philosopher, invented an adding and subtracting machine with a sophisticated carry mechanism which in principle is still realized in our hodometers of today (Williams, 1985).

But it was Leibniz's mechanical calculating machine for the first four rules of arithmetic which contained each of the mechanical devices from the input, selector, and register mechanism to the carry, control, and erasing mechanism. The Leibniz's machine became the prototype of a hand calculating machine. If we abstract from the technical details and particular mechanical constructions of Leibniz's machine, then we get a model of an ideal calculating machine.

Definition of Leibniz's mechanical calculating machine. Figure 1 is a scheme of this ideal machine with a crank C and three number stores SM, TM, RM. Natural numbers can be entered in the set-up (input) mechanism SM by the set-up handles SH. If crank C is turned to the right, then the contents of SM are added to the contents of the result mechanism RM (Register Machine), and the contents of the turning mechanism TM (Turing Machine) are raised by 1. A turn to the left with crank C subtracts the contents of SM from the contents of RM and diminishes the contents of TM by 1.

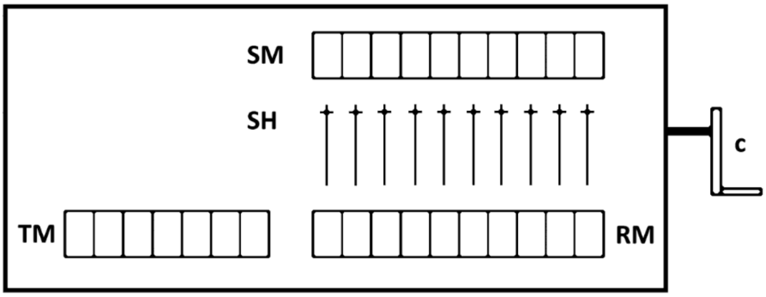


Fig. 1. Hand calculating machine.

Addition means the following. At the beginning of the calculation, the erasing procedure is implemented by setting TM and RM to zero. Then the first number is set up in SM by SH. A turn to the right of crank C transports this number into RM. In other words, the number is added to the zero in RM. Now, the second number is set up in the SM and added to the contents of RM by a turn to the right. The sum of both numbers can be read in the RM. After turning the crank twice to the right, the TM shows 2. Multiplication only means a repeated addition of the same number. The product $b \cdot a$ results from adding the number a to itself b times.

Leibniz even designed a mechanical calculating machine for the binary number system with only two digits 0 and 1, which he discovered some years earlier (von Mackensen, 1974). He described a mechanism for translating a decimal number into the corresponding binary number and vice versa. As modern electronic computers only have two states 1 (electronic impulse) and 0 (no electronic impulse), Leibniz truly became one of the pioneers of computer science.

Leibniz's historical machines suffered from many technical problems because the materials and technical skills then available were not up to the demands. Nevertheless, his design is part of a general research program (Leibniz, 1998).

Leibniz's mathesis universalis

Leibniz's *mathesis universalis* (Scholz, 1961) intended to simulate human thinking by calculation procedures (“algorithms”) and to implement them on mechanical calculating machines. Leibniz proclaimed two basic subdisciplines:

- An *ars iudicandi* should allow every scientific problem to be decided by an appropriate arithmetic algorithm after its codification into numeric symbols.
- An *ars inveniendi* should allow scientists to seek and enumerate possible solutions of scientific problems.

Leibniz's *mathesis universalis* already seems to foreshadow the famous Hilbert program in our century with its demands for formalization and axiomatization of mathematical knowledge. Actually, Leibniz developed some procedures to formalize and codify languages. He was deeply convinced that there are universal algorithms to decide all problems in the world by mechanical devices.

In the 19th century, it was the English mathematician and economist Charles Babbage who not only constructed the first program-controlled calculation machine (the "analytical engine") but also studied its economic and social consequences (Bromley, 1982). A forerunner of his famous book *On the Economy of Machinery and Manufactures* (1841) was Adam Smith's idea of economic laws, which paralleled Newton's mechanical laws. In his book *The Wealth of Nations*, Smith described the industrial production of pins as an algorithmic procedure and anticipated Henry Ford's idea of program-controlled mass production in industry.

The modern formal logic of Frege and Russell and the mathematical proof theory of Hilbert and Gödel have been mainly influenced by Leibniz's program of *mathesis universalis*. The hand calculating machine which was abstracted from the Leibniz's machine can easily be generalized to Marvin Minsky's so-called register machine (Minsky, 1961; Sheperdson and Sturgis, 1963). It allows the general concept of computability to be defined in modern computer science.

Definition of a register machine. A hand calculating machine had only two registers TM and RM, and only rather small natural numbers can be input. An ideal register machine has finite number of registers which can store any finite number of a desired quantity. The registers are denoted by natural numbers $i = 1, 2, 3, \ldots$. The contents of register i are denoted by $\langle i \rangle$. As an example, the device $\langle 4 \rangle := 1$ means that the content of the register with number 4 is 1. The register is empty if it has the content 0.

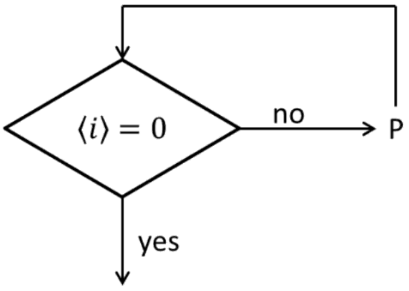
In the hand calculating machine, an addition or subtraction was realized only for the two registers $\langle \mathrm{SM} \rangle$ and $\langle \mathrm{RM} \rangle$, with $\langle \mathrm{SM} \rangle + \langle \mathrm{RM} \rangle$ or $\langle \mathrm{RM} \rangle - \langle \mathrm{SM} \rangle$ going into the register RM. In a register machine, the result of subtraction $\langle i \rangle - \langle j \rangle$ should be 0 if $\langle j \rangle$ is greater than $\langle i \rangle$. This modified subtraction is denoted by $\langle i \rangle \dot{-} \langle j \rangle$. In general, the program of an ideal register machine is defined using the following elementary procedures as building blocks:

(1) Add 1 to $\langle i \rangle$ and put the result into register i, in short $\langle i \rangle := \langle i \rangle + 1$.
(2) Subtract 1 from $\langle i \rangle$ and put the result into register i, in short: $\langle i \rangle := \langle i \rangle \dot{-} 1$.

These two elementary procedures can be composed using the following concepts:

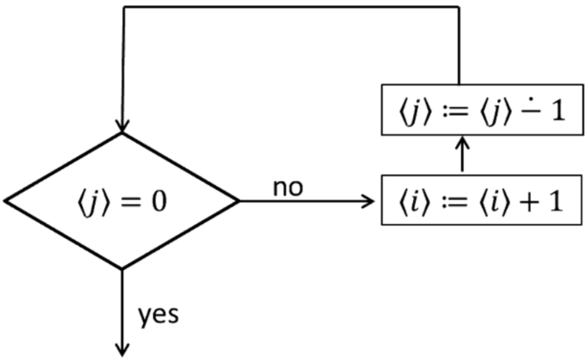
(3) If P and Q are well-defined programs, then the chain $P \to Q$ is a well-defined program. $P \to Q$ means that a machine has to execute program Q after program P.
(4) The iteration of a program, which is necessary for multiplication, for instance, as iterated addition is controlled by the question of whether a certain register is empty.

A diagram illustrates this feedback:



If P is a well-defined program, then execute P until the content of the register with number i is zero.

Example of a register machine. Each elementary operation (1) and (2) of a program is counted as a step of computation. A simple example is the following addition program:



Each state of the machine is illustrated by the following matrix, which incrementally adds the content y of register $\langle j\rangle$ to the content x of register $\langle i\rangle$ and simultaneously decrements the content of $\langle j\rangle$ to zero. The result $x+y$ of the addition is shown in register $\langle j\rangle$:

$$\begin{array}{rl} \langle i\rangle & \langle j\rangle \\ x & y \\ x+1 & y \dot{-} 1 \\ \vdots & \vdots \\ x+y & y \dot{-} y \end{array}$$

Definition of register machine computability. A register machine (RM) with program F is defined to compute a function f with n arguments if for arbitrary $x_1, \ldots, x_n$ in the register $1, \ldots, n$ (and zero in all other ones), the program F is executed and stops after a finite number of steps with the arguments of the function in the register $1, \ldots, n$ and the function value $f(x_1, \ldots, x_n)$ in register $n+1$.

The program

$$\begin{array}{c} \langle 1\rangle := x_1; \ldots; \langle n\rangle := x_n \\ \downarrow \\ F \\ \downarrow \\ \langle n+1\rangle := f(x_1, \ldots, x_n) \end{array}$$

works according to a corresponding matrix. A function f is called computable by an RM (RM-computable) if there is a program F computing f.

The number of steps which a certain program F needs to compute a function f is determined by the program and depends on the arguments of the function. The complexity of program F is measured by a function $s_F(x_1, \ldots, x_n)$ computing the steps of computation according to program F. Sometimes, $s_F(x_1, \ldots, x_n)$ is also said to measure the computational time of program F. For example, the matrix of the addition program for $x + y$ shows that y elementary steps of adding 1 and y elementary steps of subtracting 1 are necessary. Thus $s_F(x, y) = 2y$.

Definition of RM complexity. As an RM-computable function f may be computed by several programs, a function g is called the step counting function of f if there is a program F to compute f with $g(x_1, \ldots, x_n) = s_F(x_1, \ldots, x_n)$ for all arguments $x_1, \ldots, x_n$. The complexity of a function is defined as the complexity of the best program computing the function with the least number of steps.

Obviously, Minsky's register machine is an intuitive generalization of a hand calculating machine *à la* Leibniz. But, historically, some other equivalent formulations of machines were at first introduced independently by Alan Turing (1936) and Emil Post (1936) in 1936.

Definition of a Turing machine. A Turing machine (Fig. 2) can carry out any effective procedure provided it is correctly programmed. It consists of

(a) a control box in which a finite program is placed,
(b) a potentially infinite tape, divided lengthwise into squares,
(c) a device for scanning, or printing on one square of the tape at a time, and for moving along the tape or stopping, all under the command of the control box.

If the symbols used by a Turing machine are restricted to a stroke / and a blank *, then an RM-computable function can be

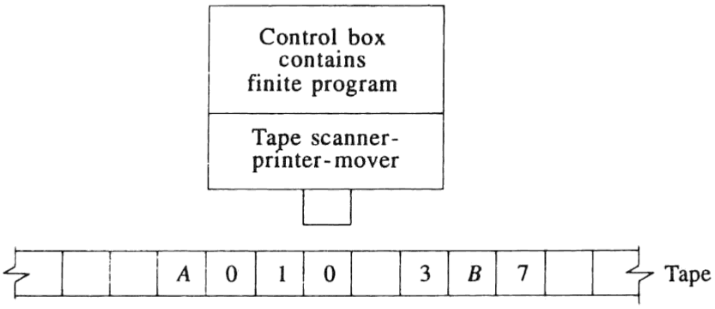


Fig. 2. Turing machine.

proved to be computable by a Turing machine and vice versa. We must remember that every natural number x can be represented by a sequence of x strokes (for instance 3 by $///$), each stroke on a square of the Turing tape. The blank * is used to denote that the square is empty (of the corresponding number is zero). In particular, a blank is necessary to separate sequences of strokes representing numbers. Thus, a Turing machine computing a function f with arguments $x_1, \ldots, x_n$ starts with tape $\cdots * x_1 * x_2 * \cdots * x_n * \cdots$ and stops with $\cdots * x_1 * x_2 * \cdots * x_n * f(x_1, \ldots, x_n) * \cdots$ on the tape.

From a logical point of view, a general purpose computer — as constructed by associates of John von Neumann in America and independently by Konrad Zuse in Germany — is a technical realization of a universal Turing machine which can simulate any kind of Turing program (Herken, 1995). Analogously, we can define a universal register machine which can execute any kind of register program. Actually, the general design of a von Neumann computer consists of a central processor (program controller), a memory, an arithmetic unit, and input–output devices. It operates step by step in a largely serial fashion. A present-day computer *à la* von Neumann is really a generalized Turing machine.

The efficiency of a Turing machine can be increased by the introduction of several tapes, which are not necessarily one-dimensional, each acted on by one or more heads, but reporting back to a single control box which coordinates all the activities of the machine (Fig. 3) (Arbib, 1987, p. 131). Thus, every computation of such a

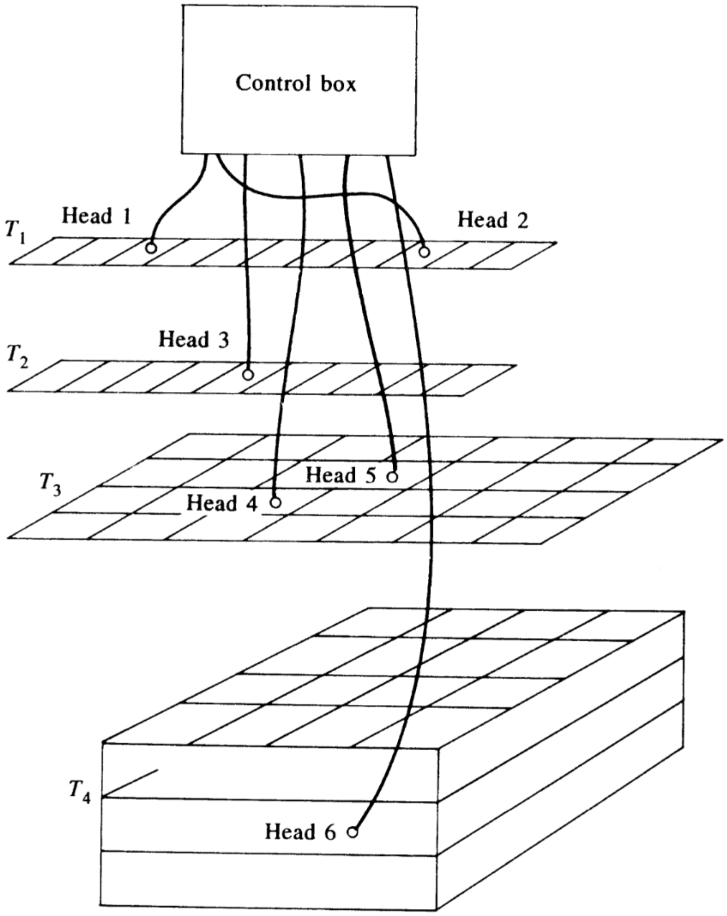


Fig. 3. Turing machine with several tapes.

more effective machine can be done by an ordinary Turing machine. Concerning the complex system approach, even a Turing machine with several multidimensional tapes remains a sequential program-controlled computer, differing essentially from self-organizing systems like neural networks.

Besides Turing and register machines, there are many other mathematically equivalent procedures for defining computable functions. Recursive functions are defined by unbounded search, procedures of functional substitution and iteration, beginning with some elementary functions, for instance, substitution and iteration, beginning with some elementary functions, for instance, the successor function $n(x) = x + 1$, which are obviously computable. All these definitions of computability by Turing machines, register machines, recursive functions, etc., can be proved to be mathematically equivalent. Obviously, each of these precise concepts defines a procedure which is intuitively effective. Thus, Alonzo Church postulated his thesis that the informal intuitive notion of an effective procedure is identical with one of these equivalent precise concepts, such as that of a Turing machine.

Church's thesis. *Every computational procedure* (*algorithm*) *can be calculated by a Turing machine.*

Church's thesis cannot be proved, of course, because mathematically precise concepts are compared with an informal intuitive notion. Nevertheless, the mathematical equivalence of several precise concepts of computability which are intuitively effective confirms Church's thesis. Consequently, we can speak about computability, effectiveness, and computable functions without referring to particular effective procedures ("algorithms") like Turing machines, register machines, recursive functions, etc. According to Church's thesis, we may in particular say that every computational procedure (algorithm) can be calculated by a Turing machine. So every recursive function, as a kind of machine program, can be calculated by a general purpose computer (Feferman, 2006).

To be more precise, let us consider a class of functions which was accepted as intuitively computable from the very beginning (Kleene, 1974; Shoenfield, 1967).

Definition of primitive recursive functions. The class $\mathcal{F}_{pr}$ of primitive recursive functions is defined by the following conditions:

(1) Z, S, p_i^n belong to $\mathcal{F}_{pr}$:

Z is the zero function with $Z(x) = 0$,
S is the successor function with $S(x) = x + 1$,
p_i^n is a projection function with $p_i^n(x_0, \ldots, x_n) = x_i$ $(0 \leq i \leq n)$.

(2) $\mathcal{F}_{pr}$ is closed under composition:

If the functions f, g_j belong to $\mathcal{F}_{pr}$ with $f : \mathbb{N}^k \to \mathbb{N}$ and $g_j : \mathbb{N}^n \to \mathbb{N}$ $(1 \leq j \leq k)$, then there is a function h from $\mathcal{F}_{pr}$ satisfying

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_k(\vec{x})) \quad (\text{with } \vec{x} = x_1, \ldots, x_n).$$

(3) $\mathcal{F}_{pr}$ is closed under recursion:

For functions $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$, there is a function h from $\mathcal{F}_{pr}$ such that

$$h(0, \vec{x}) = f(\vec{x}),$$
$$h(S(y), \vec{x}) = g(h(y, \vec{x}), y, \vec{x}).$$

(4) $\mathcal{F}_{pr}$ is the least class satisfying conditions (1)–(3).

The class of primitive recursive functions can be extended by an intuitively computable search procedure for minimal numbers satisfying certain computable conditions.

Definition of recursive functions. The class $\mathcal{F}_r$ of total recursive functions is the least class of functions satisfying the conditions (1)–(3) of $\mathcal{F}_{pr}$ (with $\mathcal{F}_r$ replacing $\mathcal{F}_{pr}$) and the following condition:

(5) $\mathcal{F}_r$ is closed under the application of the μ-operator:

$f(\vec{y}) = \mu x(g(x, \vec{y}) = 0)$, i.e., the least x satisfying $g(x, \vec{y}) = 0$ with g from $\mathcal{F}_r$ if such an x exists, or formally $\forall y \exists x(x, \vec{y}) = 0$.

Sometimes it is not clear that functions are totally defined. $f(x) \simeq g(x)$ means that function $f(x)$ is defined iff function $g(x)$ is defined, and if they are defined, then their values are equal. Now, we can extend the class of total recursive functions to the class of partial recursive functions.

Definition of partial recursive functions. The class $\mathcal{F}'_r$ of partial recursive functions is defined by the following conditions:

(1) $\mathcal{F}_r$ is a subset of $\mathcal{F}'_r$.
(2) $\mathcal{F}'_r$ is closed under composition:
If the functions h, g_j belong to $\mathcal{F}'_r$ with $h : \mathbb{N}^k \to \mathbb{N}$ and $g_j : \mathbb{N}^n \to \mathbb{N}(1 \leq j \leq k)$, then there is a function f of $\mathcal{F}'_r$ satisfying

$$f(\vec{x}) \simeq h(g_1(\vec{x}), \ldots, g_k(\vec{x})).$$

(3) $\mathcal{F}'_r$ is closed under the application of the μ-operator:
$f(\vec{y}) \simeq \mu x(g(x, \vec{y}) = 0)$, i.e., the least x satisfying $g(x, \vec{y}) = 0$ with g from $\mathcal{F}_r$ if such an x exists, or formally $\forall y \exists x(x, \vec{y}) = 0$.

It turns out that all partial recursive functions are computable in the sense of Church's thesis. Now, we are able to define effective procedures of decision and enumerability, which were already demanded by Leibniz's program of a *mathesis universalis* (Davis, 1958). The characteristic function χ_M of a subset M of natural numbers is defined by $\chi_M(x) = 1$ if x is an element of M, and as $\chi_M(x) = 0$ otherwise.

Definition. A set M is defined as effectively decidable if its characteristic function saying whether or not a number belongs to M is effectively computable.

Programs and computations consist of lists of symbols which can be coded by natural numbers. Then, we can define a decidable predicate $T(x, \vec{y}, z)$ (Kleene's T-predicate) with the meaning that "z codes a (terminating) computation according to program x for arguments $\vec{y}$". The total computable result-extracting function U extracts the result from the code for a terminating computation. It can be proven that each computable function f with code x of its computer program can be represented by the following form.

Kleene's normal form.

$$\begin{aligned} f(\vec{y}) &\simeq U(\mu z \chi_T(x, \vec{y}, z) = 1) \\ &\simeq U(\mu z T(x, \vec{y}, z)). \end{aligned}$$

The partially defined expression $[x](\vec{y})$ denotes the result of applying program x to the arguments of function f. Actually, x is the code number of a machine program which is also called "machine number" of the computable function f. Obviously, $[x](\vec{y})$ is defined if there is a (terminating) computation z according to program x for input arguments $\vec{y}$:

$$[x](\vec{y}) \text{ is defined } \leftrightarrow \exists z T(x, \vec{y}, z).$$

A set M is defined as effectively enumerable if there exists an effective (computable) procedure f for generating its elements, one after another (formally $f(1) = x_1, f(2) = x_2, \ldots$ for all elements $x_1, x_2, \ldots$ from M). The definition of recursive enumerability can be generalized from sets to predicates.

Definition of recursive enumerability. A predicate P is recursively enumerable, if P is empty or there is a recursive function f with $P(x) \leftrightarrow \exists y f(y) = x$ for all x.

Recursive enumerability can be interpreted as a formal definition of Leibniz's *ars inveniendi*. It can be proven (Hermes, 1961, p. 189) that for all recursively enumerable predicates P there is a recursively decidable predicate Q.

Theorem. $P(x) \leftrightarrow \exists y\, Q(x, y)$ *for all* x.

Proof. ("$\rightarrow$") Let P be recursively enumerable. If P is empty, then P is also recursively decidable. We define $Q(x, y) : \leftrightarrow P(x) \wedge y = y$.

It follows by definition

$$\begin{aligned} P(x) &\leftrightarrow \exists y(P(x) \wedge y = y) \\ &\leftrightarrow \exists y Q(x, y) \text{ for all } x. \end{aligned}$$

If P is not empty, then there is by definition a recursive function f with

$$\begin{aligned} &P(x) \leftrightarrow \exists y f(y) = x \\ &\qquad \leftrightarrow \exists y Q(x, y) \text{ with a recursive predicate } Q(x, y) : \leftrightarrow f(y) = x. \end{aligned}$$

("←") We assume that P can be represented as

$$P(x) \leftrightarrow \exists y\, Q(x,y) \text{ with a recursively decidable predicate } Q.$$

We can assume that P is not empty, otherwise the statement is trivial. Now, we assume a number $\bar{x}$ with $P(\bar{x})$ and define a recursive function:

$$f(y) = \begin{cases} \sigma_{2,1}(y) & \text{if } Q(\sigma_{2,1}(y), \sigma_{2,2}(y)), \\ \bar{x} & \text{otherwise} \end{cases}$$

with primitive recursive encoding function

$$\sigma_2(x,y) := 2^x(2y+1) \dot{-} 1$$

and inverse functions

$$\sigma_{2,1}(z) = \exp(0, z+1),$$
$$\sigma_{2,2}(z) = \frac{\frac{z+1}{2^{\exp(0,z+1)}} \dot{-} 1}{2}.$$

Obviously, we get

$$\sigma_{2,1}(\sigma_2(x,y)) = x,$$
$$\sigma_{2,2}(\sigma_2(x,y)) = y,$$
$$\sigma_2(\sigma_{2,1}(z), \sigma_{2,2}(z)) = z.$$

In the next step, we prove that f enumerates the elements of P according to the definition of recursive enumerability, i.e., $P(x) \leftrightarrow \exists y f(y) = x$.

("→") In case of $P(x)$, there is a z with $Q(x,z)$ according to the assumed representation of P. In this case, we define $y = \sigma_2(x,z)$. It follows $Q(\sigma_{2,1}(y), \sigma_{2,2}(y))$. By definition of f, we get $f(y) = \sigma_{2,1}(y) = x$.

("←") Now, we assume that there is a y with $f(y) = x$. We must prove $P(x)$. According to the definition of f, we have to distinguish the following two cases:

(1) $Q(\sigma_{2,1}(y), \sigma_{2,2}(y))$ is true: In this case, $f(y) = \sigma_{2,1}(y)$. It follows $Q(f(y), \sigma_{2,2}(y))$, i.e.,

$Q(x, \sigma_{2,2}(y))$. Therefore, there is a z with $Q(x, z)$.

(2) $Q(\sigma_{2,1}(y), \sigma_{2,2}(y))$ is not true: According to our definition of f, it is $f(y) = \bar{x}$. According to the assumption of "$\leftarrow$", it is $x = \bar{x}$. It follows $P(x)$ because of $P(\bar{x})$. □

The last theorem and Kleene's normal form can now be used to characterize recursive enumerable predicates (Rogers, 1967).

Kleene's enumeration of recursively enumerable predicates. Let P be a recursively enumerable predicate

$$P(y) \leftrightarrow \exists z\, Q(y, z)$$

with Q recursive for all y. Define a recursive partial function $f(y) \simeq \mu z Q(y, z)$ with machine number x. Then

$$P(y) \leftrightarrow [x](y) \text{ is defined } \leftrightarrow \exists z\, T(x, y, z).$$

The number x is called the recursive enumerable (RE) index of the recursive enumerable predicate P. Intuitively spoken, Kleene's T-predicate enumerates all recursive enumerable predicates by their RE indices (Soare, 1987).

Post's theorem of decidability and enumerability

Decidability can be characterized by enumerability: A set or predicate is recursively decidable if the set or predicate and its complementary set or predicate are recursively enumerable (Post's theorem). The complementary set $\bar{M}$ of M contains all elements which do not belong to M. If M and $\bar{M}$ are recursively enumerable, then we can enumerate their elements step by step in order to decide if a given number does belong to M or not. Thus, M is recursively decidable. By definition, it follows that every recursively decidable set is recursively enumerable. But there are recursively enumerable sets which are not decidable. These are the first hints that there are limits to Leibniz's original program of a *mathesis universalis*, based on the belief in universal decision procedures.

At this point, Turing's famous halting problem comes in: Is there a universal decision procedure to determine whether an arbitrary computer program stops after finite steps for an arbitrary input? Turing proved that the halting problem is in principle unsolvable. Then Gödel's incompleteness (Gödel, 1931) is only a corollary of Turing's proof (Chaitin, 1998).

Turing started his proof with the question whether real numbers are computable. A real number like $\pi = 3.1415926\ldots$ has an infinite number of digits that seem to be randomly distributed behind the decimal point. Nevertheless, there are simple finite programs for calculating the digits step by step with increasing precision of π. In this sense, π is called a computable real number. In a first step, Turing constructed an uncomputable real number.

Definition of an uncomputable number. According to Kleene's normal form, a machine program can be coded by a machine number. Imagine a list of all possible computer programs that are ordered according to their increasing machine numbers $p_1, p_2, p_2, \ldots$. If a program computes a real number with an infinite number of digits behind the decimal point (e.g., π), then they should be written down behind the corresponding program number. (The number before the decimal point is neglected.) Otherwise, there is a blank line in the list (Chaitin, 1998, p. 10):

$$p_1 - .d_{\underline{11}}d_{12}d_{13}d_{14}d_{15}d_{16}d_{17}\ldots$$

$$p_2 - .d_{21}d_{\underline{22}}d_{23}d_{24}d_{25}d_{26}d_{27}\ldots$$

$$p_3 - .d_{31}d_{32}d_{\underline{33}}d_{34}d_{35}d_{36}d_{37}\ldots$$

$$p_4$$

$$p_5 - .d_{51}d_{52}d_{53}d_{54}d_{\underline{55}}d_{56}d_{57}\ldots$$

$$\vdots$$

Following Cantor's diagonal procedure, Turing changed the underlined digits on the diagonal of the list and put these changed digits (marked with $\neq$) together into a new number with a decimal point in front:

$$-. \neq d_{11} \neq d_{12} \neq d_{13} \neq d_{14} \neq d_{15} \neq d_{16} \neq d_{17} \cdots .$$

This new number cannot be in the list because it differs from the first digit of the first number behind p_1, the second digit of the second number behind p_2, etc. Therefore, it is an uncomputable real number. With this number, Turing got the unsolvability of the halting problem.

Unsovability of the halting problem

If we could solve the halting problem, then we could decide if the nth computer program ever puts out an nth digit behind the decimal point. In this case, we could actually carry out Cantor's diagonal procedure and compute a real number, which, by its definition, has to differ from any computable real number.

The unsolvability of the halting problem refutes Hilbert's Entscheidungsproblem. If there is a complete formal axiomatic system from which all mathematical truth follows, then it would give us a procedure to decide if a computer program will ever halt. We just run through all the possible proofs until we either find a proof that the program halts, or we find a proof that it never halts. So if Hilbert's finite set of axioms from which all the mathematical truth should follow were possible, then by running through all possible proofs while checking which ones are correct, we would be able to decide if a computer program halts. That is impossible using Turing's proof.

The unsolvability of Turing's halting problem is not only fundamental for computability theory, but it also has deep consequences for mathematical foundations. An example is Hilbert's 10th problem which is also proved to be in principle unsolvable due to Turing's halting problem. In 1900, David Hilbert asked for an algorithm which will decide whether a so-called diophantine equation has a solution (Hilbert, 1900). This was the 10th problem of his famous list of 23 mathematical problems which were still unsolved at the beginning of the 20th century. Algebraic equations, which involve only multiplication, addition, and exponentiation of whole numbers, were named after the third-century Greek mathematician Diophantos of Alexandria.

Unsovability of Hilbert's 10th problem

In 1970, J. V. Matiyasevich from the Steklov Institute of Mathematics in former Leningrad (St. Petersburg) proved that Hilbert's 10th problem is equivalent to Turing's halting problem and, consequently, not decidable (Matiyasevich, 1970, 1993).

Matiyasevich used results of Davis *et al.* (1961). According to Lagrange's representation of natural numbers as sum of four quadratic whole numbers, Hilbert's 10th problem can be reduced to the existence of solutions in natural numbers.

A predicate D is called diophantine if it is definable *by* predicates $x + y = z$, $x \cdot y = z$, $x^y = z$, and logical operations $\vee$ (or), $\wedge$ (and), and $\exists$ (existence quantifier):

$$\begin{aligned} &D(x_1, \ldots, x_n) \\ &\qquad \leftrightarrow \exists y_1, \ldots, y_r \quad P(x_1, \ldots, x_n, \quad y_1, \ldots, y_r) \text{ with } P \text{ recursive} \\ &\qquad \leftrightarrow \exists y_1, \ldots, y_r \quad \chi_p(x_1, \ldots, x_n, \quad y_1, \ldots, y_r) = 1 \end{aligned}$$

with computable characteristic function χ_p as polynom.

According to our theorem of recursive enumerability, it follows that every diophantine predicate is recursively enumerable. Vice versa, it can be proven that every enumerable predicate is diophantine. Matiyasevich used the Fibonacci sequence to define an appropriate diophantine predicate. The halting problem can be represented by an enumerable, but not decidable predicate. Therefore, the corresponding diophantine predicate is also not decidable.

Chapter 3

Hierarchies of Computability

According to Church's thesis, Turing computability is a representative definition of computability in general. In the following, we want to consider problems with degrees of complexity below and beyond this limit. Below this limit, there are many practical problems concerning certain limitations on how much the speed of an algorithm can be increased. Especially among mathematical problems, there are some classes of problems that are intrinsically more difficult to solve algorithmically than others. Thus, there are degrees of computability for Turing machines which are made precise in complexity theory in computer science.

Complexity classes of problems (or corresponding functions) can be characterized by complexity degrees, which give the order of functions describing the computational time (or number of elementary computational steps) of algorithms (or computational programs) depending on the length of their inputs. The lengths of inputs may be measured by the number of decimal digits. According to the machine language of a computer, it is convenient to represent decimal numbers in their binary codes with only binary numbers 0 and 1 and to define their length by the number of binary digits. For instance, 3 has the binary code $11 = 1 \cdot 2^1 + 1 \cdot 2^0$ with the length 2.

Definition of linear computational time. A function f has linear computational time if the computational time of f is not greater than $c \cdot n$ for all inputs with length n and a constant c.

Example of linear computational time. The addition of two (binary) numbers has obviously only linear computational time. For instance, the task $3 + 7 = 10$ corresponds to the binary calculation

$$\begin{array}{r} 011 \\ \underline{111} \\ 1010 \end{array}$$

which needs five elementary computational steps of adding two binary digits (including carrying). We remind the reader that the elementary steps of adding binary digits are $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, $1 + 1 = 0$ carry 1. It is convenient to assume that the two numbers which should be added have equal length. Otherwise, we simply start the shorter one with a series of zeros, for instance 111 and 011 instead of 11. In general, if the length of the particular pair of numbers which should be added is n, the length of a number is $\frac{n}{2}$, and thus, we need no more than $\frac{n}{2} + \frac{n}{2} = n$ elementary steps of computation including carrying.

Definition of quadratic computational time. A function f has quadratic computational time if the computational time of f is not greater than $c \cdot n^2$ for all inputs with length n and a constant c.

Example of quadratic computational time. A simple example of quadratic computational time is the multiplication of two (binary) numbers. For instance, the task $7 \cdot 3 = 21$ corresponds to the binary calculation:

$$\begin{array}{r} \underline{111 \cdot 011} \\ 000 \\ 111 \\ \underline{111} \\ 10101 \end{array}$$

According to former conventions, we have $n = 6$. The number of elementary binary multiplications is $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$. Including carrying, the number of elementary binary additions is $\frac{n}{2} \cdot \frac{n}{2} - \frac{n}{2} = \frac{n^2}{4} - \frac{n}{2}$. In all, we get $\frac{n^2}{4} + \frac{n^2}{4} - \frac{n}{2} = \frac{n^2}{2} - \frac{n}{2}$, which is smaller than $\frac{n^2}{2}$.

Definition of polynomial and exponential computational time.

- A function f has polynomial computational time if the computational time of f is not greater than $c \cdot n^k$, which is assumed to be the leading term of a polynomial $p(n)$.
- A function f has exponential computational time if the computational time of f is not greater than $c \cdot 2^{p(n)}$.
- Many practical and theoretical problems belong to the complexity class P of all functions which can be computed by a deterministic Turing machine in polynomial time.

In the history of mathematics, there have been some nice problems of graph theory to illustrate the basic concepts of complexity theory (Grötschel *et al.*, 1988).

Complexity of Euler's Königsberg river problem

In 1736, the famous mathematician Leonard Euler (1707–1783) solved one of the first problems of graph theory. In the city of Königsberg, the capital of former Eastern Prussia, the so-called old and new river Pregel are joined in the river Pregel. In the 18th century, there were seven bridges connecting the southern s, northern n, and eastern e regions with the island i (Fig. 4a). Is there a route which crosses each bridge only once and returns to the starting point?

Euler reduced the problem to graph theory. The regions n, s, l, e are replaced by vertices of a graph, and the bridges between two regions by edges between the corresponding vertices (Fig. 4b).

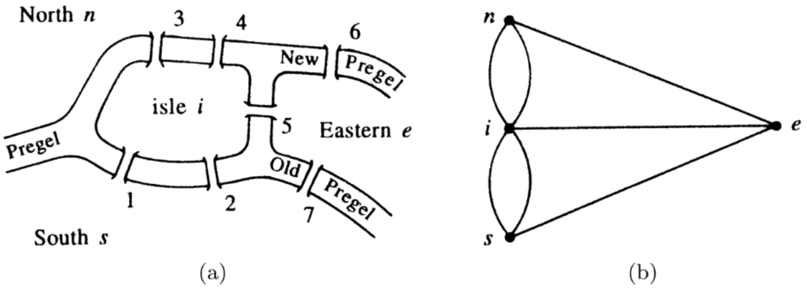


Fig. 4. Euler's Königsberg river problem.

In the language of graph theory, Euler's problem is whether for every vertex there is a route (an "Euler circuit") passing each edge exactly once, returning finally to the starting point. For arbitrary graphs, Euler proved that an Euler circuit exists if and only if each vertex has an even number of edges (the "Euler condition"). As the graph of Fig. 4b does not satisfy this condition, there cannot be a solution of Euler's problem in this case. In general, there is an algorithm testing an arbitrary graph by Euler's condition if it is an Euler circuit. The input of the algorithm consists of the set V of all vertices $1, \ldots, n$ and the set E of all edges, which is a subset of the set with all pairs of vertices. The computational time of this algorithm depends linearly on the size of the graph, which is defined by the sum of the numbers of vertices and edges.

Complexity of Hamilton's problem

In 1859, the mathematician William Hamilton (1805–1865) introduced a rather similar problem that is much more complicated than Euler's problem (Mainzer, 2007a, pp. 189–190). Hamilton considered an arbitrary graph, which means nothing else than a finite collection of vertices, a certain number of pairs of which are connected together by edges. Hamilton's problem is whether there is a closed circuit (a "Hamilton's circuit") passing each vertex (not each edge as in Euler's problem) exactly once. Figure 5 shows a graph with a Hamilton circuit passing the vertices in the order of numbering.

However, unlike the case of Euler's problem, we do not know any condition which exactly characterizes whether a graph contains a Hamilton circuit or not. We only can define an algorithm testing whether an arbitrary graph contains a Hamilton circuit or not. The algorithm tests all permutations of vertices to see if they form a Hamiltonian circuit. As there are $n!$ different permutations of n vertices, the algorithm does not need more than $c \cdot n!$ steps with a constant c to find a solution. It can easily be proved that an order of $n!$ corresponds to an order of n^n. Consequently, an algorithm for the Hamilton problem needs exponential computational time, while the Euler problem can be solved algorithmically in linear computational

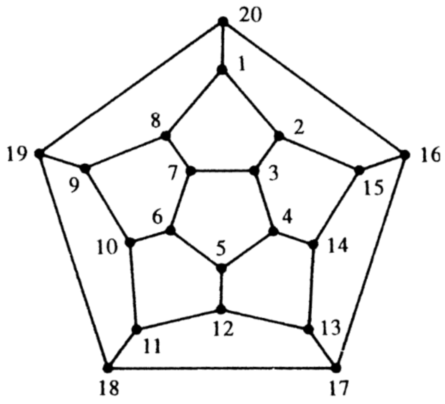


Fig. 5. Hamilton's problem.

time. Thus, Hamilton's problem cannot practically be solved by a computer even for small numbers n.

The main reason for a high computational time may be a large number of single subcases which must be tested by a deterministic computer which is allowed to choose a computational procedure at random among a finite number of possible ones instead of performing them step by step in a serial way. Let us consider Hamilton's problem again. An input graph may have n vertices $v_1, \ldots, v_n$. A non-deterministic algorithm chooses a certain order $v_{i_1}, \ldots, v_{i_n}$ of vertices in a non-deterministic, random way. Then the algorithm tests whether this order forms a Hamiltonian circuit. The question is whether for all number $j (j = 1, \ldots, n-1)$ the successive vertices v_{i_j} and $v_{i_{j+1}}$ and the beginning and starting vertices v_{i_n} and v_{i_1} are connected by an edge. The computational time of this non-deterministic algorithm depends linearly on the size of the graph.

In general, NP means the complexity class of functions which can be computed by a non-deterministic Turing machine in polynomial time. Hamilton's problem is an example of an NP-problem. Another NP-problem is the "travelling salesman problem", which is rather like Hamilton's problem except that the various edges have numbers attached to them. One seeks that Hamilton circuit for which the sum of the numbers or more intuitively the distance travelled by the salesman is a minimum.

Can NP-problems be reduced to P-problems?

By definition every P-problem is an NP-problem. But it is a crucial question of complexity theory whether P = NP or, in other words, whether problems which are solved by non-deterministic computers in polynomial time can also be solved by a deterministic computer in polynomial time.

Hamilton's problem and the travelling salesman problem are examples of so-called NP-complete problems.

Complexity of NP-complete problems

A problem is called an NP-complete problem if any other NP-complete problem can be converted into it in polynomial time.

Consequently, if an NP-complete problem is actually proved to be a P-problem (if for instance a deterministic algorithm can be constructed to solve Hamilton's problem in polynomial time), then it would follow that all NP-problems are actually in P. Otherwise if P $\neq$ NP then no NP-complete problem can be solved with a deterministic algorithm in polynomial time.

Obviously, complexity theory delivers degrees for the algorithmic power of Turing machines or Turing-type computers. The theory has practical consequences for scientific and industrial applications. But does it imply limitations for mathematics and the human mind? The fundamental questions of complexity theory (for example N = NP or N $\neq$ NP) refer to the measurement of the speed, computational time, storage capacity, and so on, of algorithms. What about the complexity of mathematical problems beyond Turing computability?

Enumerability is only the first step on a ladder with increasing computational complexity. By adding unrestricted quantifiers to recursive predicates, degrees of computability can be extended beyond Turing computability (Hinman, 1978; Kleene, 1955a; Shoenfield, 1967):

Definition of arithmetical hierarchy. A predicate P is arithmetical iff it has an explicit definition

$$(*) \quad P(x) \leftrightarrow Q_1 \dots Q_n \, R(x, x_1, \dots, x_n)$$

for all numbers x with R recursive and each number quantifier Q_i for either $\exists x_i$ or $\forall x_i$ with number variables x_i.

Two quantifiers are of the same kind if they are both existential or both universal. Two adjacent quantifiers of the same kind can be replaced by a single quantifier (contraction of quantifiers).

A predicate is $\Sigma_n^0(\Pi_n^0)$ for $n \geq 1$ iff it has an explicit definition $(*)$ with no two adjacent quantifiers of the same kind and the first quantifier existential (universal), e.g., for all x,

$$\Sigma_2^0 : P(x) \leftrightarrow \exists x_1 \forall x_2 R(x, x_1 x_2),$$
$$\Pi_2^0 : P(x) \leftrightarrow \forall x_1 \exists x_2 R(x, x_1 x_2).$$

Every arithmetical predicate is Σ_n^0 or Π_n^0 for some $n \geq 1$. This classification of arithmetical predicates is called the arithmetical hierarchy.

A predicate is Δ_n^0 iff it is both Σ_n^0 and Π_n^0.

Every recursively enumerable predicate P can be represented by $P(x) \leftrightarrow \exists y \, Q(x, y)$ for all x with a recursive predicate Q. Therefore, Σ_1^0 is the class of recursively enumerable predicates. Recursive predicates are closed with respect to logical connectives, e.g., $\neg$ (negation), $\vee$ (disjunction: "or"), and $\wedge$ (conjunction: "and"). In classical logic, $\exists y \, Q(x, y)$ is equivalent with $\neg \forall y \neg Q(x, y)$. It follows that P is Δ_1^0 iff both P and $\neg P$ are enumerable because of double negation. Thus, because of Post's theorem, Δ_1^0 is the class of the recursive (Turing-computable) predicates.

There are general properties of the arithmetical hierarchy:

(1) A recursive predicate is Σ_n^0 and Π_n^0 for all n.
(2) $\Sigma_m^0 \subseteq \Sigma_n^0(\Pi_n^0)$, $\Pi_m^0 \subseteq \Sigma_n^0$ (Π_n^0) for all $n > m$: If predicate P is Σ_m^0 or Π_m^0, then P is Σ_n^0 and Π_n^0.
(3) If P and Q are Σ_n^0 (Π_n^0), then $P \vee Q$ and $P \wedge Q$ are Σ_n^0 (Π_n^0).
(4) If P is Σ_n^0 (Π_n^0), then $\neg P$ is Π_n^0 (Σ_n^0).

Therefore, the arithmetical hierarchy can be illustrated in the following way:

$$\begin{array}{ccc}
\vdots & \vdots & \vdots \\
\Sigma^0_{n+1} & \Delta^0_{n+1} & \Pi^0_{n+1} \\
\Sigma^0_n & \Delta^0_n & \Pi^0_n \\
\vdots & \vdots & \vdots \\
\Sigma^0_2 & \Delta^0_2 & \Pi^0_2 \\
\Sigma^0_1 & \Delta^0_1 & \Pi^0_1
\end{array}$$

Kleene's normal form of recursive functions allows to enumerate all recursive functions and recursively enumerable predicates (Chapter 2). It can be generalized for the arithmetical hierarchy:

Kleene's arithmetical enumeration theorem. *For each $n \geq 1$, there is a Σ^0_n (Π^0_n) predicate which enumerates the class of all Σ^0_n (Π^0_n) predicates.*

Further on, it can be proven that the inclusions between the Σ^0_n and Π^0_n predicates, i.e., $\Sigma^0_m \subseteq \Sigma^0_n$ (Π^0_n), $\Pi^0_m \subseteq \Sigma^0_n$ (Π^0_n) for all $n > m$, are the only ones of their type:

Kleene's arithmetical hierarchy theorem. *For each $n \geq 1$, there is a Σ^0_n predicate which is not Π^0_n and hence not Σ^0_m or Π^0_m for any $m > n$. Then $\neg P$ is Π^0_n but not Σ^0_n and hence not Σ^0_m or Π^0_m for any $m > n$.*

Relative computability and oracle machines

Computations can depend on external agencies supplying answers to questions about a set (or property) M by an unknown procedure. According to Turing, the external agency of an unknown procedure is called an oracle (Turing, 1939). In Fig. 6, the computation of a function value $f(x)$ depends on an oracle deciding if certain values satisfy a property M (Rogers, 1967). Most of our calculations in

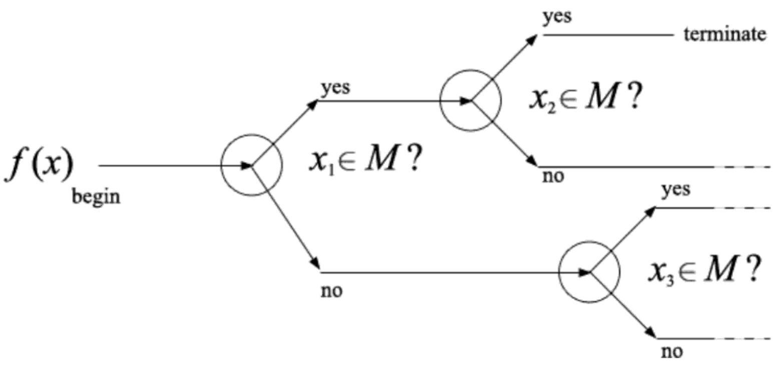


Fig. 6. Relative computability with an oracle.

everyday life depend on assumptions we rely on without knowing a final confirmation. Thus, oracles lead to an important extension of relativized computation.

A function f is computable relative to M iff the program of f is extended with an oracle M. An oracle can also be a machine operation (total function) φ supplying values $\varphi(x)$ by an unknown procedure. Random oracles depend on random procedures.

Definition of relative computability.

- A function f is recursive in φ iff f is computable relative to φ.
- A predicate P is recursive (decidable) in φ iff its characteristic function χ_P is computable relative to φ.
- P is Σ_n^0 (Π_n^0, Δ_n^0) in φ iff P is defined by Σ_n^0 (Π_n^0, Δ_n^0) quantifiers and a predicate R recursive in φ.

Relative computability with respect to a function φ can be generalized to a set Φ of functions $\varphi_0, \varphi_1, \ldots$ We say that set Φ is recursive in set Ψ with functions $\psi_0, \psi_1, \ldots$ if every member of Φ is recursive in Ψ. If Ψ is empty, then the functions recursive in Ψ are just the recursive functions. Therefore, relativized Turing computability includes Turing computability as a special case. Obviously, relativized computability satisfies the property of transitivity: If Φ is