Turing machine problems and solutions pdf

Problem of determining whether a given program will finish running or continue forever This article includes a list of general references, but it remains largely unverified because it lacks sufficient corresponding inline citations. Please help to improve this article by introducing more precise citations. (September 2018) (Learn how and when to remove this template message) In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. For any program f that might determine if programs halt, a "pathological" program g, called with some input, can pass its own source and its input to f and then specifically do the opposite of what f predicts g will do. No f can exist that handles this case. A key part of the proof is a mathematical definition of a computer and program, which is known as a Turing machine; the halting problem is undecidable over Turing machines. It is one of the first cases of decision problems proven to be unsolvable. This proof is significant to practical computing efforts, defining a class of applications which no programming invention can possibly perform perfectly. Jack Copeland (2004) attributes the introduction of the term halting problem to the work of Martin Davis in the 1950s.[1] Background The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long and use an arbitrary amount of storage space before halting. The question is simply whether the given program will ever halt on a particular input. For example, in pseudocode, the program while (true) continue does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program print "Hello, world!" does halt. While deciding whether these programs halt is simple, more complex programs prove problematic. One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever. Turing proved no algorithm exists that always correctly decides whether, for a given arbitrary program and input, the program halts when run with that input. The essence of Turing's proof is that any such algorithm can be made to contradict itself and therefore cannot be correct. Programming consequences Some infinite loops can be quite useful. For instance, event loops are typically coded as infinite loops.[2] However, most subroutines are intended to finish (halt).[3] In particular, in hard real-time computing, programmers attempt to write subroutines that are not only guaranteed to finish (halt), but are also guaranteed to finish before a given deadline.[4] Sometimes these programmers use some general-purpose (Turing-complete) programming language, but attempt to write in a restricted style—such as MISRA C or SPARK—that makes it easy to prove that the resulting subroutines finish before the given deadline.[citation needed] Other times these programmers apply the rule of least power—they deliberately use a computer language that is not quite fully Turing-complete. Frequently, these are languages that guarantee all subroutines finish, such as Coq.[citation needed] Common pitfalls The difficulty in the halting problem lies in the requirement that the decision procedure must work for all programs and inputs. A particular program either halts on a given input or does not halt. Consider one algorithm that always answers "halts" and another that always answers "does not halt". For any specific program and input, one of these two algorithms answers correctly, even though nobody may know which one. Yet neither algorithm solves the halting problem generally. There are programs (interpreters) that simulate the execution of whatever source code they are given. Such programs can demonstrate that a program does halt if this is the case: the interpreter itself will eventually halt its simulation, which shows that the original program halted. However, an interpreter will not halt if its input program does not halt, so this approach cannot solve the halting problem as stated; it does not successfully answer "does not halt" for programs that do not halt. The halting problem is theoretically decidable for linear bounded automata (LBAs) or deterministic machines with finite memory. A machine with finite memory has a finite number of configurations, and thus any deterministic program on it must eventually either halt or repeat a previous configuration: ...any finite-state machine, if left completely to itself, will fall eventually into a perfectly periodic repetitive pattern. The duration of this repeating pattern cannot exceed the number of internal states of the machine... (italics in original, Minsky 1967, p. 24) Minsky notes, however, that a computer with a million small parts, each with two states, would have at least 21,000,000 possible states: This is a 1 followed by about three hundred thousand zeroes ... Even if such a machine were to operate at the frequencies of cosmic rays, the aeons of galactic evolution would be as nothing compared to the time of a journey through such a cycle (Minsky 1967 p. 25): Minsky states that although a machine may be finite, and finite automata "have a number of theoretical limitations": ...the magnitudes involved should lead one to suspect that theorems and arguments based chiefly on the mere finiteness [of] the state diagram may not carry a great deal of significance. (Minsky p. 25) It can also be decided automatically whether a nondeterministic machine with finite memory halts on none, some, or all of the possible sequences of nondeterministic decisions, by enumerating states after each possible decision. History Further information: Algorithm § History: Development of the notion of "algorithm" This section is in list format, but may read better as prose. You can help by converting this section, if appropriate. Editing help is available. (February 2020) The halting problem is historically important because it was one of the first problems to be proved undecidable. (Turing's proof went to press in May 1936, whereas Alonzo Church's proof of the undecidability of a problem in the lambda calculus had already been published in April 1936 [Church, 1936].) Subsequently, many other undecidable problems have been described. Timeline 1900: David Hilbert poses his "23 questions" (now known as Hilbert's problems) at the Second International Congress of Mathematicians in Paris. "Of these, the second was that of proving the consistency of the 'Peano axioms' on which, as he had shown, the rigour of mathematics depended". (Hodges p. 83, Davis' commentary in Davis, 1965, p. 108) 1920–1921: Emil Post explores the halting problem for tag systems, regarding it as a candidate for unsolvability. (Absolutely unsolvable problems and relatively undecidable propositions – account of an anticipation, in Davis, 1965, pp. 340–433.) Its unsolvability was not established until much later, by Marvin Minsky (1967). 1928: Hilbert recasts his 'Second Problem' at the Bologna International Congress. (Reid pp. 188–189) Hodges claims he posed three questions: i.e. #1: Was mathematics complete? #2: Was mathematics consistent? #3: Was mathematics decidable? (Hodges p. 91). The third question is known as the Entscheidungsproblem (Decision Problem). (Hodges p. 91, Penrose p. 34) 1930: Kurt Gödel announces a proof as an answer to the first two of Hilbert's 1928 questions [cf Reid p. 198]. "At first he [Hilbert] was only angry and frustrated, but then he began to try to deal constructively with the problem... Gödel himself felt—and expressed the thought in his paper—that his work did not contradict Hilbert's formalistic point of view" (Reid p. 199) 1931: Gödel publishes "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I", (reprinted in Davis, 1965, p. 5ff) 19 April 1935: Alonzo Church publishes "An Unsolvable Problem of Elementary Number Theory", wherein he identifies what it means for a function to be effectively calculable. Such a function will have an algorithm, and "...the fact that the algorithm has terminated becomes effectively known ..." (Davis, 1965, p. 100) 1936: Church publishes the first proof that the Entscheidungsproblem is unsolvable. (A Note on the Entscheidungsproblem, reprinted in Davis, 1965, p. 110.) 7 October 1936: Emil Post's paper "Finite Combinatory Processes. Formulation I" is received. Post adds to his "process" an instruction "(C) Stop". He called such a process "type 1 ... if the process it determines terminates for each specific problem." (Davis, 1965, p. 289ff) 1937: Alan Turing's paper On Computable Numbers With an Application to the Entscheidungsproblem reaches print in January 1937 (reprinted in Davis, 1965, p. 115). Turing's proof departs from calculation by recursive functions and introduces the notion of computation by machine. Stephen Kleene (1952) refers to this as one of the "first examples of decision problems proved unsolvable". 1939: J. Barkley Rosser observes the essential equivalence of "effective method" defined by Gödel, Church, and Turing (Rosser in Davis, 1965, p. 223, "Informal Exposition of Proofs of Gödel's Theorem and Church's Theorem") 1943: In a paper, Stephen Kleene states that "In setting up a complete algorithmic theory, what we do is describe a procedure ... which procedure necessarily terminates and in such manner that from the outcome we can read a definite answer, 'Yes' or 'No,' to the question, 'Is the predicate value true?'." 1952: Kleene (1952) Chapter XIII ("Computable Functions") includes a discussion of the unsolvability of the halting problem for Turing machines and reformulates it in terms of machines that "eventually stop", i.e. halt: "... there is no algorithm for deciding whether any given machine, when started from any given situation, eventually stops." (Kleene (1952) p. 382) 1952: "Martin Davis thinks it likely that he first used the term 'halting problem' in a series of lectures that he gave at the Control Systems Laboratory at the University of Illinois in 1952 (letter from Davis to Copeland, 12 December 2001)." (Footnote 61 in Copeland (2004) pp. 40ff) Formalization In his original proof Turing formalized the concept of algorithm by introducing Turing machines. However, the result is in no way specific to them; it applies equally to any other model of computation that is equivalent in its computational power to Turing machines, such as Markov algorithms, Lambda calculus, Post systems, register machines, or tag systems. What is important is that the formalization allows a straightforward mapping of algorithms to some data type that the algorithm can operate upon. For example, if the formalism lets algorithms define functions over strings (such as Turing machines) then there should be a mapping of these algorithms to strings, and if the formalism lets algorithms define functions over natural numbers (such as computable functions) then there should be a mapping of algorithms to natural numbers. The mapping to strings is usually the most straightforward, but strings over an alphabet with n characters can also be mapped to numbers by interpreting them as numbers in an n-ary numeral system. Representation as a set Main article: Decision problem The conventional representation of decision problems is the set of objects possessing the property in question. The halting set K = {(i, x) | program i halts when run on input x} represents the halting problem. This set is recursively enumerable, which means there is a computable function that lists all of the pairs (i, x) it contains. However, the complement of this set is not recursively enumerable.[5] There are many equivalent formulations of the halting problem; any set whose Turing degree equals that of the halting problem is such a formulation. Examples of such sets include: {i | program i eventually halts when run with input 0} {i | there is an input x such that program i eventually halts when run with input x}. Proof concept The proof that the halting problem is not solvable is a proof by contradiction. To illustrate the concept of the proof, suppose that there exists a total computable function halts(f) that returns true if the subroutine f halts (when run with no inputs) and returns false otherwise. Now consider the following subroutine: def g(): if halts(g): loop_forever() halts(g) must either return true or false, because halts was assumed to be total. If halts(g) returns true, then g will call loop_forever and never halt, which is a contradiction. If halts(g) returns false, then g will halt, because it will not call loop_forever; this is also a contradiction. Overall, halts(g) can not return a truth value that is consistent with whether g halts. Therefore, the initial assumption that halts is a total computable function must be false. The method used in the proof is called diagonalization - g does the opposite of what halts says g should do. The difference between this sketch and the actual proof is that in the actual proof, the computable function halts does not directly take a subroutine as an argument; instead it takes the source code of a program. The actual proof requires additional work to handle this issue. Moreover, the actual proof avoids the direct use of recursion shown in the definition of g. Sketch of proof The concept above shows the general method of the proof; this section will present additional details. The overall goal is to show that there is no total computable function that decides whether an arbitrary program i halts on arbitrary input x; that is, the following function h is not computable (Penrose 1990, p. 57–63): h (i , x) = { 1 if program i halts on input x , 0 otherwise. {\displaystyle h(i,x)={\begin{cases}1&{\text{if }}{\text{ program }}i{\text{ halts on input }}x,\\0&{\text{otherwise.}}\end{cases}}} Here program i refers to the i th program in an enumeration of all the programs of a fixed Turing-complete model of computation. f(i,j) i 1 2 3 4 5 6 j 1 1 0 0 1 0 1 2 0 0 0 1 0 0 3 0 1 0 1 0 1 4 1 0 0 1 0 0 5 0 0 0 1 1 1 6 1 1 0 0 1 0 f(i,i) 1 0 0 1 1 0 g(i) U 0 0 U U 0 Possible values for a total computable function f arranged in a 2D array. The orange cells are the diagonal. The values of f(i,i) and g(i) are shown at the bottom; U indicates that the function g is undefined for a particular input value. The proof proceeds by directly establishing that no total computable function with two arguments can be the required function h. As in the sketch of the concept, given any total computable binary function f, the following partial function g is also computable by some program e: g (i) = { 0 if f (i , i) = 0 , undefined otherwise. {\displaystyle g(i)={\begin{cases}0&{\text{if }}f(i,i)=0,\\{\text{undefined}}&{\text{otherwise.}}\end{cases}}} The verification that g is computable relies on the following constructs (or their equivalents): computable subprograms (the program that computes f is a subprogram in program e), duplication of values (program e computes the inputs i,i for f from the input i for g), conditional branching (program e selects between two results depending on the value it computes for f(i,i)), not producing a defined result (for example, by looping forever), returning a value of 0. The following pseudocode illustrates a straightforward way to compute g: procedure compute_g(i): if f(i, i) == 0 then return 0 else loop forever Because g is partial computable, there must be a program e that computes g, by the assumption that the model of computation is Turing-complete. This program is one of all the programs on which the halting function h is defined. The next step of the proof shows that h(e,e) will not have the same value as f(e,e). It follows from the definition of g that exactly one of the following two cases must hold: f(e,e) = 0 and so g(e) = 0. In this case h(e,e) = 1, because program e halts on input e. f(e,e) ≠ 0 and so g(e) is undefined. In this case h(e,e) = 0, because program e does not halt on input e. In either case, f cannot be the same function as h. Because f was an arbitrary total computable function with two arguments, all such functions must differ from h. This proof is analogous to Cantor's diagonal argument. One may visualize a two-dimensional array with one column and one row for each natural number, as indicated in the table above. The value of f(i,j) is placed at column i, row j. Because f is assumed to be a total computable function, any element of the array can be calculated using f. The construction of the function g can be visualized using the main diagonal of this array. If the array has a 0 at position (i,i), then g(i) is 0. Otherwise, g(i) is undefined. The contradiction comes from the fact that there is some column e of the array corresponding to g itself. Now assume f was the halting function h, if g(e) is defined (g(e) = 0 in this case), g(e) halts so f(e,e) = 1. But g(e) = 0 only when f(e,e) = 0, contradicting f(e,e) = 1. Similarly, if g(e) is not defined, then halting function f(e,e) = 0, which leads to g(e) = 0 under g's construction. This contradicts the assumption of g(e) not being defined. In both cases contradiction arises. Therefore any arbitrary computable function f cannot be the halting function h. Computability theory Main article: Computability theory The typical method of proving a problem to be undecidable is with the technique of reduction[clarification needed]. To do this, it is sufficient to show that if a solution to the new problem were found, it could be used to decide an undecidable problem by transforming instances of the undecidable problem into instances of the new problem. Since we already know that no method can decide the old problem, no method can decide the new problem either. Often the new problem is reduced to solving the halting problem. (The same technique is used to demonstrate that a problem is NP complete, only in this case, rather than demonstrating that there is no solution, it demonstrates there is no polynomial time solution, assuming P ≠ NP.) For example, one such consequence of the halting problem's undecidability is that there cannot be a general algorithm that decides whether a given statement about natural numbers is true or false. The reason for this is that the proposition stating that a certain program will halt given a certain input can be converted into an equivalent statement about natural numbers. If we had an algorithm that could find the truth value of every statement about natural numbers, it could certainly find the truth value of this one; but that would determine whether the original program halts, which is impossible, since the halting problem is undecidable. Rice's theorem generalizes the theorem that the halting problem is unsolvable. It states that for any non-trivial property, there is no general decision procedure that, for all programs, decides whether the partial function implemented by the input program has that property. (A partial function is a function which may not always produce a result, and so is used to model programs, which can either produce results or fail to halt.) For example, the property "halt for the input 0" is undecidable. Here, "non-trivial" means that the set of partial functions that satisfy the property is neither the empty set nor the set of all partial functions. For example, "halts or fails to halt on input 0" is clearly true of all partial functions, so it is a trivial property, and can be decided by an algorithm that simply reports "true." Also, this theorem holds only for properties of the partial function implemented by the program; Rice's Theorem does not apply to properties of the program itself. For example, "halt on input 0 within 100 steps" is not a property of the partial function that is implemented by the program—it is a property of the program implementing the partial function and is very much decidable. Gregory Chaitin has defined a halting probability, represented by the symbol Ω, a type of real number that informally is said to represent the probability that a randomly produced program halts. These numbers have the same Turing degree as the halting problem. It is a normal and transcendental number which can be defined but cannot be completely computed. This means one can prove that there is no algorithm which produces the digits of Ω, although its first few digits can be calculated in simple cases. While Turing's proof shows that there can be no general method or algorithm to determine whether algorithms halt, individual instances of that problem may very well be susceptible to attack. Given a specific algorithm, one can often show that it must halt for any input, and in fact computer scientists often do just that as part of a correctness proof. But each proof has to be developed specifically for the algorithm at hand; there is no mechanical, general way to determine whether algorithms on a Turing machine halt. However, there are some heuristics that can be used in an automated fashion to attempt to construct a proof, which succeed frequently on typical programs. This field of research is known as automated termination analysis. Since the negative answer to the halting problem shows that there are problems that cannot be solved by a Turing machine, the Church–Turing thesis limits what can be accomplished by any machine that implements effective methods. However, not all machines conceivable to human imagination are subject to the Church–Turing thesis (e.g. oracle machines). It is an open question whether there can be actual deterministic physical processes that, in the long run, elude simulation by a Turing machine, and in particular whether any such hypothetical process could usefully be harnessed in the form of a calculating machine (a hypercomputer) that could solve the halting problem for a Turing machine amongst other things. It is also an open question whether any such unknown physical processes are involved in the working of the human brain, and whether humans can solve the halting problem (Copeland 2004, p. 15). Gödel's incompleteness theorems This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (August 2019) (Learn how and when to remove this template message) The concepts raised by Gödel's incompleteness theorems are very similar to those raised by the halting problem, and the proofs are quite similar. In fact, a weaker form of the First Incompleteness Theorem is an easy consequence of the undecidability of the halting problem. This weaker form differs from the standard statement of the incompleteness theorem by asserting that an axiomatization of the natural numbers that is both complete and sound is impossible. The "sound" part is the weakening: it means that we require the axiomatic system in question to prove only true statements about natural numbers. Since soundness implies consistency, this weaker form can be seen as a corollary of the strong form. It is important to observe that the statement of the standard form of Gödel's First Incompleteness Theorem is completely unconcerned with the truth value of a statement, but only concerns the issue of whether it is possible to find it through a mathematical proof. The weaker form of the theorem can be proved from the undecidability of the halting problem as follows. Assume that we have a sound (and hence consistent) and complete axiomatization of all true first-order logic statements about natural numbers. Then we can build an algorithm that enumerates all these statements. This means that there is an algorithm N(n) that, given a natural number n, computes a true first-order logic statement about natural numbers, and that for all true statements, there is at least one n such that N(n) yields that statement. Now suppose we want to decide if the algorithm with representation a halts on input i. We know that this statement can be expressed with a first-order logic statement, say H(a, i). Since the axiomatization is complete it follows that either there is an n such that N(n) = H(a, i) or there is an n' such that N(n') = ¬ H(a, i). So if we iterate over all n until we either find H(a, i) or its negation, we will always halt, and furthermore, the answer it gives us will be true (by soundness). This means that this gives us an algorithm to decide the halting problem. Since we know that there cannot be such an algorithm, it follows that the assumption that there is a consistent and complete axiomatization of all true first-order logic statements about natural numbers must be false. Generalization Many variants of the halting problem can be found in computability textbooks (e.g., Sipser 2006, Davis 1958, Minsky 1967, Hopcroft and Ullman 1979, Börger 1989). Typically their undecidability follows by reduction from the standard halting problem. However, some of them have a higher degree of unsolvability. The next two examples are typical. Halting on all inputs The universal halting problem, also known (in recursion theory) as totality, is the problem of determining, whether a given computer program will halt for every input (the name totality comes from the equivalent question of whether the computed function is total). This problem is not only undecidable, as the halting problem, but highly undecidable. In terms of the arithmetical hierarchy, it is Π 2 0 {\displaystyle \Pi _{2}^{0}} -complete (Börger 1989, p. 121). This means, in particular, that it cannot be decided even with an oracle for the halting problem. Recognizing partial solutions There are many programs that, for some inputs, return a correct answer to the halting problem, while for other inputs they do not return an answer at all. However the problem "given program p, is it a partial halting solver" (in the sense described) is at least as hard as the halting problem. To see this, assume that there is an algorithm PHSR ("partial halting solver recognizer") to do that. Then it can be used to solve the halting problem, as follows: To test whether input program x halts on y, construct a program p that on input (x,y) reports true and diverges on all other inputs. Then test p with PHSR. The above argument is a reduction of the halting problem to PHS recognition, and in the same manner, harder problems such as halting on all inputs can also be reduced, implying that PHS recognition is not only undecidable, but higher in the arithmetical hierarchy, specifically Π 2 0 {\displaystyle \Pi _{2}^{0}} -complete. Lossy computation A lossy Turing machine is a Turing machine in which part of the tape may non-deterministically disappear. The Halting problem is decidable for lossy Turing machine but non-primitive recursive.[6]:92 Oracle machines See also: Turing jump A machine with an oracle for the halting problem can determine whether particular Turing machines will halt on particular inputs, but they cannot determine, in general, if machines equivalent to themselves will halt. See also Busy beaver Gödel's incompleteness theorem Brouwer-Hilbert controversy Kolmogorov complexity P versus NP problem Termination analysis Worst-case execution time Notes ^ In none of his work did Turing use the word "halting" or "termination". Turing's biographer Hodges does not have the word "halting" or words "halting problem" in his index. The earliest known use of the words "halting problem" is in a proof by Davis (1958, p. 70–71): "Theorem 2.2 There exists a Turing machine whose halting problem is recursively unsolvable. "A related problem is the printing problem for a simple Turing machine Z with respect to a symbol Si". Davis adds no attribution for his proof, so one infers that it is original with him. But Davis has pointed out that a statement of the proof exists informally in Kleene (1952, p. 382). Copeland (2004, p 40) states that: "The halting problem was so named (and it appears, first stated) by Martin Davis [cf. Copeland footnote 61]... (It is often said that Turing stated and proved the halting theorem in 'On Computable Numbers', but strictly this is not true)." ^ McConnell, Steve (2004), Code Complete (2nd ed.), Pearson Education, p. 374, ISBN 9780735636972 ^ Han-Way Huang. "The HCS12 / 9S12: An Introduction to Software and Hardware Interfacing". p. 197. quote: "... if the program gets stuck in a certain loop, ... figure out what's wrong." ^ David E. Simon. "An Embedded Software Primer". 1999. p. 253. quote: "For hard real-time systems, therefore, it is important to write subroutines that always execute in the same amount of time or that have a clearly identifiable worst case." ^ Moore, Cristopher; Mertens, Stephan (2011). The Nature of Computation. Oxford University Press. pp. 236–237. doi:10.1093/acprof:oso/9780199233212.001.0001. ISBN 978-0-19-923321-2. ^ Abdulla, Parosh Aziz; Jonsson, Bengt (1996). "Verifying Programs with Unreliable Channels". Information and Computation. 127 (2): 91–101. doi:10.1006/inco.1996.0053.

References Turing, A. M. (1937). "On Computable Numbers, with an Application to the Entscheidungsproblem". Proceedings of the London Mathematical Society. Wiley. s2-42 (1): 230–265. doi:10.1112/plms/s2-42.1.230. ISSN 0024-6115., Turing, A. M. (1938). "On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction". Proceedings of the London Mathematical Society. Wiley. s2-43 (1): 544–546. doi:10.1112/plms/s2-43.6.544. ISSN 0024-6115. This is the epochal paper where Turing defines Turing machines, formulates the halting problem, and shows that it (as well as the Entscheidungsproblem) is unsolvable. Sipser, Michael (2006). "Section 4.2: The Halting Problem". Introduction to the Theory of Computation (Second ed.). PWS Publishing. pp. 173–182. ISBN 0-534-94728-X. c2:HaltingProblem Church, Alonzo (1936). "An Unsolvable Problem of Elementary Number Theory". American Journal of Mathematics. 58 (2): 345–363. doi:10.2307/2371045. JSTOR 2371045. B. Jack Copeland ed. (2004), The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma, Clarendon Press (Oxford University Press), Oxford UK, ISBN 0-19-825079-7. Davis, Martin (1965). The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems And Computable Functions. New York: Raven Press.. Turing's paper is #3 in this volume. Papers include those by Godel, Church, Rosser, Kleene, and Post. Davis, Martin (1958). Computability and Unsolvability. New York: McGraw-Hill.. Alfred North Whitehead and Bertrand Russell, Principia Mathematica to *56, Cambridge at the University Press, 1962. Re: the problem of paradoxes, the authors discuss the problem of a set not be an object in any of its "determining functions", in particular "Introduction, Chap. 1 p. 24 "...difficulties which arise in formal logic", and Chap. 2.I. "The Vicious-Circle Principle" p. 37ff, and Chap. 2.VIII. "The Contradictions" p. 60ff. Martin Davis, "What is a computation", in Mathematics Today, Lynn Arthur Steen, Vintage Books (Random House), 1980. A wonderful little paper, perhaps the best ever written about Turing Machines for the non-specialist. Davis reduces the Turing Machine to a far-simpler model based on Post's model of a computation. Discusses Chaitin proof. Includes little biographies of Emil Post, Julia Robinson. Marvin Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, Inc., N.J., 1967. See chapter 8, Section 8.2 "Unsolvability of the Halting Problem." Roger Penrose, The Emperor's New Mind: Concerning computers, Minds and the Laws of Physics, Oxford University Press, Oxford England, 1990 (with corrections). Cf. Chapter 2, "Algorithms and Turing Machines". An over-complicated presentation (see Davis's paper for a better model), but a thorough presentation of Turing machines and the halting problem, and Church's Lambda Calculus. John Hopcroft and Jeffrey Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading Mass, 1979. See Chapter 7 "Turing Machines." A book centered around the machine-interpretation of "languages", NP-Completeness, etc. Andrew Hodges, Alan Turing: The Enigma, Simon and Schuster, New York. Cf. Chapter "The Spirit of Truth" for a history leading to, and a discussion of, his proof. Constance Reid, Hilbert, Copernicus: Springer-Verlag, New York, 1996 (first published 1970). Fascinating history of German mathematics and physics from 1880s through 1930s. Hundreds of names familiar to mathematicians, physicists and engineers appear in its pages. Perhaps marred by no overt references and few footnotes: Reid states her sources were numerous interviews with those who personally knew Hilbert, and Hilbert's letters and papers. Edward Beltrami, What is Random? Chance and order in mathematics and life, Copernicus: Springer-Verlag, New York, 1999. Nice, gentle read for the mathematically inclined non-specialist, puts tougher stuff at the end. Has a Turing-machine model in it. Discusses the Chaitin contributions. Moore, Cristopher; Mertens, Stephan (2011), The Nature of Computation, Oxford University Press, ISBN 9780191620805 Ernest Nagel and James R. Newman, Godel's Proof, New York University Press, 1958. Wonderful writing about a very difficult subject. For the mathematically inclined non-specialist. Discusses Gentzen's proof on pages 96–97 and footnotes. Appendices discuss the Peano Axioms briefly, gently introduce readers to formal logic. Taylor Booth, Sequential Machines and Automata Theory, Wiley, New York, 1967. Cf. Chapter 9, Turing Machines. Difficult book, meant for electrical engineers and technical specialists. Discusses recursion, partial-recursion with reference to Turing Machines, halting problem. Has a Turing Machine model in it. References at end of Chapter 9 catch most of the older books (i.e. 1952 until 1967 including authors Martin Davis, F. C. Hennie, H. Hermes, S. C. Kleene, M. Minsky, T. Rado) and various technical papers. See note under Busy-Beaver Programs. Busy Beaver Programs are described in Scientific American, August 1984, also March 1985 p. 23. A reference in Booth attributes them to Rado, T.(1962), On non-computable functions, Bell Systems Tech. J. 41. Booth also defines Rado's Busy Beaver Problem in problems 3, 4, 5, 6 of Chapter 9, p. 396. David Bolter, Turing's Man: Western Culture in the Computer Age, The University of North Carolina Press, Chapel Hill, 1984. For the general reader. May be dated. Has yet another (very simple) Turing Machine model in it. Egon Börger. "Computability, Complexity, Logic". North-Holland, 1989. Stephen Kleene, Introduction to Metamathematics, North-Holland, 1952. Chapter XIII ("Computable Functions") includes a discussion of the unsolvability of the halting problem for Turing machines. In a departure from Turing's terminology of circle-free nonhalting machines, Kleene refers instead to machines that "stop", i.e. halt. Sven Köhler, Christian Schindelhauer, Martin Ziegler, On approximating real-world halting problems, pp.454-466 (2005) ISBN 3540281932 Springer Lecture Notes in Computer Science volume 3623: Undecidability of the Halting Problem means that not all instances can be answered correctly; but maybe "some", "many" or "most" can? On the one hand the constant answer "yes" will be correct infinitely often, and wrong also infinitely often. To make the question reasonable, consider the density of the instances that can be solved. This turns out to depend significantly on the Programming System under consideration. Logical Limitations to Machine Ethics, with Consequences to Lethal Autonomous Weapons - paper discussed in: Does the Halting Problem Mean No Moral Robots? Nicholas J. Daras and Themistocles M. Rassias, Modern Discrete Mathematics and Analysis: with Applications in Cryptography, Information Systems and Modeling Springer, 2018. ISBN 978-3319743240. Chapter 3 Section 1 contains a quality description of the halting problem, a proof by contradiction, and a helpful graphic representation of the Halting Problem. External links Scooping the loop snooper - a poetic proof of undecidability of the halting problem animated movie - an animation explaining the proof of the undecidability of the halting problem A 2-Minute Proof of the 2nd-Most Important Theorem of the 2nd Millennium - a proof in only 13 lines Retrieved from "