Turing Machines, Oracle Turing Machines, and the Turing Hierarchy

Stephen Flood

August 9, 2006

Given the many functions that are used in mathematics and our own finitude, it is a natural question to ask which of them can be computed in a finite amount of time by a finite computing agent. Clearly, it will be impossible to fully compute any function that has an irrational number in either its domain or range. If an irrational number is in the domain of a function, it would be impossible even to read the input in a finite amount of time. Similarly, if an irrational number is in the range of a function, it would be impossible to write the number in a finite amount of time.

Because of this, it makes sense to restrict our questions about the computability of a function to functions from rational numbers to rational numbers. Because of the bijection between natural numbers and rationals, it also makes sense to further restrict the question to functions from the natural numbers to the natural numbers.

In order to determine which of these functions are computable, we model the 'best' possible finite computer: one with infinite memory.

1 Turing Machines

Definition 1. A Turing Machine can be thought of as consisting of 5 parts:

1. An infinite two way tape consisting of cells, each of which contain either a 1 or a blank (B).

2. A Read/Write head that rests on one cell of the tape that can be in one of finitely many states q. We denote the set of all possible states by Q.

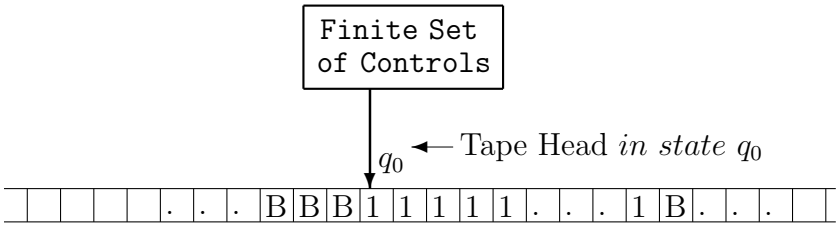

Figure 1: A Turing Machine in starting state q_0, with first input symbol 0.

3. A set of transition rules of the following form: If the head in state q reads a cell with contents s, the head changes to state p, writes t to the tape cell, and either moves left or right. This is written $\delta(q, s) = (p, t, D)$, or equivalently (q, s, p, t, D), where D is either left or right.

4. A special state $q_0 \in Q$ called the *start state*.

5. A subset $F \subseteq Q$ consisting of the *accepting states*.

A computation on a finite input begins with the input written on the tape, and the read/write head on the leftmost non-blank cell in state q_0. The computation proceeds according to the transition rules until (if ever) the head enters an accept state (a state $p \in$ F). If this happens, the number of 1's written on the tape after the computation halts is called the output of the computation.

Since there are only finitely many states and tape symbols, it follows that there are only finitely many transition rules. This allows us to number every possible Turing Machine code by the natural numbers in such a way that, given the number encoding a Turing machine program, it is possible to immediately recover the program.

- We write $\varphi_e(x)$, where $x \in \mathbb{N}$ to denote the computation of the Turing machine encoded by e where the input tape consists of a string of x 1's.

- We write $\varphi_e(x) \downarrow$ if the computation of the Turing machine encoded by e on in put x enters an accept state after applying finitely many transition rules. We write $\varphi_e(x) = t$ if $\varphi_e(x) \downarrow$ and there are t 1's on the tape at the end of the computation.

- We write $\varphi_e(x) \uparrow$ if the computation of the Turing machine encoded by e on in put x never enters an accept state. In this case, we say that the computation *diverges*.
- We say that a computation accepts after s steps, if after at most s moves of the read/write head, the head enters an accepting state. Otherwise, we say that the computation of s steps diverges. We refer to the computation of e on x after s steps by writing $\varphi_{e,s}(x)$.

Definition 2. $W_e =_{def} dom(\varphi_e) = \{\ x \mid \varphi_e(x) \downarrow\}$.

Definition 3. If $U \subseteq \mathbb{N}$, we say that a function $f : U \to \mathbb{N}$ is *partially computable* (p.c.) if there is a T.M. m s.t.

$$f(s) = t \iff \varphi_m(s) = t$$

Where the Turing machine takes a string of s 1's as input, and halts with a string of t 1's on the tape, and

$$\varphi_m(r) \uparrow \iff r \notin dom(f)$$

Definition 4. We say that a function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if f is p.c. and total.

Definition 5. We define the *characteristic function* of a set A to be the function

$$\chi_A(x) =_{def} \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

Definition 6. a set $A \subseteq \mathbb{N}$ is *computable* if the characteristic function χ_A is computable.

Definition 7. a set $A \subseteq \mathbb{N}$ is *computably enumerable* (c.e.) if it is the domain of a partial computable function.

Remark. we write $< \cdot, \cdot >$ to mean the usual bijection $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Proposition 8. *a set A is c.e. iff A is the range of some total computable function or $A = \emptyset$.*

Proof. $\Longleftarrow$. If $A = \emptyset$ then A is c.e. Now suppose $A = ran(f)$ for f a total computable function. Then let m be the index of the Turing machine that, on input x checks if there is any x s.t. $f(y) = x$, and halts and accepts iff there is such a y. Clearly $A = W_m$.

$\Longrightarrow$. Let $A = W_e \neq \emptyset$. Find the least integer $< a, t >$ s.t. $a \in W_{e,t}$. Define the computable function f by

$$f(< s, x >) = \begin{cases} x & \text{if } x \in W_{e.s+1} - W_{e.s}; \\ a & \text{otherwise} \end{cases}$$

Clearly $A = \text{rng}(f)$. Then if $x \in W_e$, choose the least s s.t. $x \in W_{e,s+1}$. Then $f(< s, x >) = x$, so $x \in \text{rng}(f)$. □

Examples of computable sets include most subsets of the natural numbers that you normally think of, such as the evens, the odds, the primes, and any finite set. Examples of noncomputable sets are harder to obtain, as are sets that are not even c.e.

Definition 9. $K =_{def} \{ x \mid \varphi_x(x) \downarrow \}$.

Definition 10. $K_0 =_{def} \{ < x, y > \mid \varphi_x(y) \downarrow \}$.

Proposition 11. *K is c.e.*

Proof. define a T.M. with index u s.t. $\varphi_u(x) = \varphi_x(x)$ In other words, on input x, u simulates the Turing machine encoded by x on x.
Clearly $x \in K \iff \varphi_u(x) \downarrow$, hence $K = W_u$ is c.e.. □

Proposition 12. *K is not computable.*

Proof. Suppose K does have a computable characteristic function. Then the function

$$f(x) =_{def} \begin{cases} \varphi_x(x) + 1 & \text{if } x \in K \\ 0 & \text{if } x \notin K \end{cases}$$

is computable. But $\forall\ x,\ f \neq \varphi_x$. $\rightarrow\leftarrow$ □

Claim 13. *K and K_0 have the 'same information content' in the following sense: although neither is computable, if we know the characteristic function of one then we can computably acquire the characteristic function of the other.*

Proof. $\Longrightarrow$ clearly, $x \in K \iff < x, x > \in K_0$

$\Longleftarrow$ If we wish to determine whether or not an arbitrary $< x, y > \in K_0$, we can write a program that on input $< x, y >$, outputs a program $m(x, y)$ s.t. $\varphi_{m(x,y)}(m(x, y)) = \varphi_x(y)$. In other words, m is a computable function.

Intuitively, m can be constructed to output a Turing machine code with extra information on the end, s.t. when given a Turing machine code with extra information on the end, simulates the first part of the code on the second part of the code. The actual precise proof of the existence of this function involves the s-m-n theorem, which is beyond the scope of this talk. □

In other words, even though neither K nor K_0 are computable, each is computable *given the other*. This idea the motivation for the Oracle Turing Machine.

2 Oracle Turing Machine

An oracle Turing machine is the same as a normal Turing Machine, only with the addition of a second tape, called the *oracle tape*. The cells on the oracle tape can contain either blanks (B), 0's, or 1's. Given a set A, an **Oracle Turing Machine with Oracle A** will have the characteristic function of the set A written onto the tape. The Oracle tape head begins on the cell containing $\chi_A(0)$. The next cell to the right of this cell contains $\chi_A(1)$, and so on in increasing order. To the left of the cell that the Oracle tape head begins on, there are all blanks.

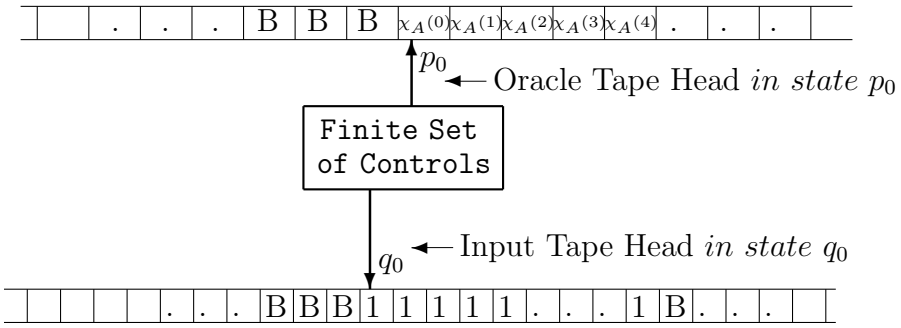

Figure 2: An Oracle Turing Machine with oracle A.

An oracle Turing machine computation proceeds in the same way as a normal Turing machine computation. Although the oracle head has a start state p_0, the Turing machine halts iff the read/write head enters an accept state. The key difference between a Turing machine with oracle A and a normal Turing machine is that a halting computation of the oracle Turing machine can check whether or not finitely many numbers are in A.

Remark. Since any description of an Oracle Turing machines is finite, it is again possible to effectively encode them into the natural numbers. We write $\Phi_e^A(x)$ to denote the computation of the e^{th} oracle Turing machine with Oracle A on input x.

Definition 14. Fin $=_{def}$ $\{ x \mid W_x \text{ is finite}\}$.

Claim 15. Fin *is neither computable nor c.e.*

Definition 16. We say that a set B is *c.e. in A* if $\exists e$ *s.t.* B=$dom(\Phi_e^A)$.

Proposition 17. Fin *is c.e. in* K_0.

Proof.

$$x \in \text{Fin} \iff (\exists z)(\forall y > z)[\varphi_x(y) \uparrow]$$

We will define a Turing machine with index m s.t. on input $< x, z >$, m simulates

$\varphi_{x,1}(z+1)$
$\varphi_{x,2}(z+2)$
$\varphi_{x,2}(z+1)$
$\varphi_{x,3}(z+3)$
$\varphi_{x,3}(z+2)$
$\varphi_{x,3}(z+1)$
$\vdots$

It is clear from this definition that $\forall s$, m simulates $\varphi_{x,s}(z+t)$, $\forall t \leq s$.
We specify that m halts on input $< x, y > \iff \varphi_{x,s}(z+t) \downarrow$ for some $s, t \in \mathbb{N}$.
But such a s, t exist $\iff < m, < x, z >> \in K_0$
It therefore follows that

$$x \in \text{Fin} \iff \exists z\ \varphi_m(< x, z >) \uparrow$$

$$\iff \exists z \ < m, < x, z >> \notin K_0$$

Therefore, we can define a Turing machine n s.t. $\Phi_n^{K_0}(x)$ checks all possible $z \in \mathbb{N}$ until, if ever $\varphi_m(< x, z >) \uparrow$, and halts if it finds such a z.
Then Fin $= dom(\Phi_n^{K_0})$ □

This proof shows that K_0, and therefore K, are very natural sets to use as oracles, since both K and K_0 directly encode information about the behavior of a Turing machine on some input string.

3 The Turing Hierarchy

We generalize the set K by defining the *jump operator*.

Definition 18. We define $K^A =_{def} \{ x \mid \Phi_x^A(x) \downarrow \}$. We say that K^A is called the *jump* of A and is denoted by A' (read as 'A prime').

Proposition 19. *A' is c.e. in A.*

Proof. This is immediate from the definition of A'. □

Proposition 20. *A' is not computable in A.*

Proof. suppose A' is computable in A. Then the function

$$f(x) =_{def} \begin{cases} \Phi_x^A(x) + 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

is computable in A. But $\forall\ x,\ f \neq \Phi_x^A$. $\rightarrow\leftarrow$ □

Definition 21. We say that $A \leq_T B$ iff A is computable in B.

Definition 22. We say that $A \equiv_T B$ iff $A \leq_T B$ and $B \leq_T A$.

Definition 23. We define $\deg(A) = \{ B \mid B \equiv_T A \}$.

Definition 24. We define $\deg(\emptyset^{(n)}) =_{def} \mathbf{0^{(n)}}$.

The Turing Hierarchy is the infinite increasing chain of degrees:

$$\mathbf{0} < \mathbf{0'} < \mathbf{0''} < \mathbf{0^{(3)}} < \mathbf{0^{(4)}} < \ldots < \mathbf{0^{(n)}} < \ldots$$

In a very precise sense, if B is any computable set,

$$\emptyset \equiv B$$

$$\emptyset' = K^{\emptyset} \equiv K$$

$$\emptyset'' \equiv \text{Fin}$$

4 Further Study

There are several undergraduate courses that touch on computability theory. One good course is *Mathematical Logic II* (MATH 25800). This is a winter quarter course that will be taught by Joseph Mileti. It is an introduction axiomatic set theory, computability theory, and model theory. Its main prerequisite is Mathematical Logic I (MATH 25700). A second good course is *Introduction to Complexity Theory* (MATH 28100). This course is a mixture between computability theory (what can be computed) and complexity theory (how fast can something be computed).

If you are looking for further reading in computability theory in particular, or mathematical logic in general, you should contact Professor Soare. The material presented in this talk comes from a working draft of Professor Soare's new book in computability theory.