CSC 438F/2404F **Notes** (T. Pitassi) **Fall, 2019**

Computability Theory

This section is partly inspired by the material in "A Course in Mathematical Logic" by Bell and Machover, Chap 6, sections 1-10.
Other references: "Introduction to the theory of computation" by Michael Sipser, and "Computability, Complexity, and Languages" by M. Davis and E. Weyuker.

Our first goal is to give a formal definition for what it means for a function on $\mathbb{N}$ to be computable by an algorithm. Historically the first convincing such definition was given by Alan Turing in 1936, in his paper which introduced what we now call Turing machines. Slightly before Turing, Alonzo Church gave a definition based on his lambda calculus. About the same time Gödel, Herbrand, and Kleene developed definitions based on recursion schemes. Fortunately all of these definitions are equivalent, and each of many other definitions proposed later are also equivalent to Turing's definition. This has lead to the general belief that these definitions have got it right, and this assertion is roughly what we now call "Church's Thesis".

A natural definition of computable function f on $\mathbb{N}$ allows for the possibility that $f(x)$ may not be defined for all $x \in \mathbb{N}$, because algorithms do not always halt. Thus we will use the symbol ∞ to mean "undefined".

Definition: A *partial function* is a function

$$f : (\mathbb{N} \cup \{\infty\})^n \to \mathbb{N} \cup \{\infty\}, n \geq 0$$

such that $f(c_1, ..., c_n) = \infty$ if some $c_i = \infty$.

In the context of computability theory, whenever we refer to a function on $\mathbb{N}$, we mean a partial function in the above sense.

Definitions:

$$\text{Domain}(f) = \{\vec{x} \in \mathbb{N}^n \mid f(\vec{x}) \neq \infty\}$$

where $\vec{x} = (x_1 \cdots x_n)$ We say f is *total* iff Domain$(f) = \mathbb{N}^n$ (i.e. if f is always defined when all its arguments are defined).

Turing machines

A Turing Machine is specified by a 7-tuple, $M = \{Q, \Sigma, \Gamma, \delta, q_1, B, \{q_2\}\}$, where $Q = \{q_1, q_2, \ldots, q_k\}$ is a finite set of states, Σ is a finite set consisting of the input alphabet, including the two elements 0 and 1; Γ is a finite set consisting of the tape alphabet, and such that $\Sigma \subseteq \Gamma$; q_1 is the designated start state, q_2 is the designated halt state, and $B \in \Gamma$ is a special blank symbol not in Σ.

The transition function, δ, is a function from $Q \times \Sigma$ to $Q \times \Gamma \times \{L, R\}$.

Σ^* denotes the set of all finite length strings over Σ. Let $x \in \Sigma^*$. We visualize a Turing machine M over Σ on input x as consisting of an tape consisting of an infinite sequence of *cells*, $c_0, c_1, \ldots,$. There is a tape *head* which points to one cell of the tape, and at every point in time, the Turing machine is in one state $q_i \in Q$. Each cell of the tape contains an element from Γ. M on input x operates as follows. Initially M is in the start state q_1, and the infinite tape consists of x written on the first $|x|$ consecutive cells, followed by all blank symbols (B). At every time step, M makes one transition, according to δ. If M is in state q and the tape head is currently reading symbol s at time t, and $\delta(q, s) = (q', s', L/R)$, then at time $t+1$, the new state is q', the symbol s is replaced by s', and the tape head moves left/right one cell. (If the tape head is already at the leftmost position, then on a L move, the head stays in place.) The computation terminates if ever the head of M is pointing at the first cell, and the current state is the final state, q_2. At this point in time, if the first cell contains the symbol 1, then M on x accepts, and otherwise, M on x does not accept.

A Turing Machine accepts a language, $L \subseteq \Sigma^*$ if and only if for every $x \in L$, M on x accepts, and for every $x \notin L$, M on x does not accept.

We can describe the computation of M on x at time t by a *configuration.* Let q be the leftmost tape cell such that all cells to the right of q contain the blank symbol, B. Then the configuration consists of the contents of the tape up to cell q, plus the current state, q, plus the location, i, of the tape head. We will represent this information by the string $s_1, s_2, ..., (q, s_i), s_{i+1}, \ldots, s_q$, where $s_1, \ldots, s_q$ are the contents of the tape cells up to cell q, and where q is the current state. (The location of q within the string tells us the location of the head.) The initial configuation of M on x is therefore the string $(q_1, x_1), x_2, \ldots, x_{n-1}, x_n$.

The computation of M on x is described by a sequence of configurations, $c_1, c_2, \ldots$, where c_1 is the initial configuration of M on x, and such that each c_i follows from the previous configuration c_{i-1} by applying the transition function δ. If M accepts x, then this sequence of configurations is finite, and the last configuration begins with the sequence $(q_2, 1)$. If M does not accept x, then either the sequence is finite, and the last configuration begins with the sequence (q_2, y), for some $y \neq 1$, or the sequence of configurations is infinite.

If M accepts x in m steps, we will visualize the corresponding sequence of configurations as a *tableaux*, or $m - by - m$ matrix, where the first row of the matrix is the start configuration, and row i is the configuration at time t.

Turing machines can also be defined to compute a function (as opposed to accepting a language) as follows. Let f be a k-ary partial function from domain D to range R. We can encode a k-tuple $a_1, \ldots, a_k$ by a string x, where x consists of the concatenation of a_1, a_2, ..., separated by blank symbols. Let M be a Turing Machine. Suppose that M on input x halts. Then the output of M on input x is the string y, where y is the contents of the tape up to and including the cell pointed to by the tape head. M computes f if the following conditions hold. (1) on all inputs x in the domain of f, M halts and outputs $f(x)$; (2) on all inputs x not in the domain of f, M does not halt on x.

Let R be a k-ary relation. Similarly we will encode k-tuples by strings x. M accepts R if for all k-tuples $x \in R$, M on x halts and outputs 1, and for all $x \notin R$, M on x either does not halt, or halts and does not output 1.

Our form of **Church's Thesis:**

Every algorithmically computable function is TM-computable.

Here the notion "algorithmically computable" is not a precise mathematical notion, but rather an intuitive notion. It is understood that the algorithms in question have unlimited memory. In the case of register machines, this means that each register can hold an arbitrarily large natural number.

Church's Thesis will be discussed further at the end of this section, after we have given many examples of computable functions.

Exercise 1 *Write Turing machine programs to compute each of the following functions:*

$f_1(x) = x + 1$
$f_2(x, y) = x \cdot y$

Be sure to respect our input/output conventions for TM's.

Let $R \subseteq \mathbb{N}^n$. Thus R is an n-ary relation (predicate). We will think of R as a total 0-1 valued function as follows: $R(\vec{x}) = 0$ iff $\vec{x} \in R$, and $R(\vec{x}) = 1$ otherwise.

Encoding Turing Machines by Numbers

We want to associate a number with each Turing machine over the alphabet $\Sigma = \{0, 1\}$. Here is one way to do this.

Our convention is that the states of a TM are always called $Q = \{q_1, q_2, \ldots, q_n\}$, where q_1 is always the start state, and q_2 is always the halt state. Similarly, assume that the tape symbols $\Sigma = \{x_1, x_2, \ldots, x_k\}$ where $x_1 = 0$ and $x_2 = 1$. Let "left" be denoted by D_1 and let "right" be denoted by D_2. Then we represent a the transition $\delta(q_i, x_j) = (q(k, x_l, D_m)$ by $0^i 1 0^j 1 0^k 1 0^l 1 0^m$. The code for M is: $111 code_1 11 code_2 11 \ldots 11 code_r 111$, where $code_i$ is the code for one of the possible transitions.

Example Let $M = (Q = \{q_1, q_2, q_3\}, \Sigma = \{0, 1\}, \Gamma = \{0, 1, B\}, \delta, q_1, B, \{q_2\})$, where we have the following transitions and their corresponding codes:

- $\delta(q_1, 1) = (q_3, 0, R)$, $c_1 = 0^1 1 0^2 1 0^3 10100$
- $\delta(q_3, 0) = (q_1, 1, R)$, $c_2 = 0^3 101010^2 100$
- $\delta(q_3, 1) = (q_2, 0, R)$, $c_3 = 0^3 1 0^2 1 0^2 1 0^2 100$
- $\delta(q_3, B) = (q_3, 1, L)$, $c_4 = 0^3 1 0^3 1 0^3 1 0^2 10$

Thus M is encoded by the string $111c_1 11c_2 11c_3 11c_4 111$, and the pair $(M, 1011)$ is encoded by the string $111c_1 11c_2 11c_3 11c_4 1111011$. Note that given an encoding $< M, w >$, we can recover the associated Turing machine M and string w.

A Universal Turing Machine

A Universal Turing machine U takes as input a pair $< M, w >$ where M encodes a Turing machine, T_M over $\Sigma = \{0, 1\}$ and w is a string over $\{0, 1\}$. U accepts $< M, w >$ if and only if the machine T_M on input w accepts. Note that if T_M on w does not accept, then U may not halt on input $< M, w >$.

The language accepted by U is L_U. L_U is the set of strings $< M, w >$ such that M encodes a TM, T_M and T_M accepts w.

L_U is recursively enumerable (r.e.). That is, there is a machine U that takes as input strings $< M, w >$ and such that: (a) If $< M, w >$ does not encode a valid Turing machine, T_m and string w, then U does not accept $< M, w >$. (b) otherwise if T_M accepts w then U halts on $< M, w >$ and accepts; (c) otherwise if T_M does not accept w, then U does not accept w.

We will describe a 3-tape TM, U. First U checks to see if $< M, w >$ is a valid encoding of a Turing machine, T_M and input w. If not, then U halts and does not accept. Otherwise, on input $< M, w >$, U will simulate the computation of T_M on w. The first tape of U contains the code of T_M. The second tape of U will contain the contents of the tape of TM T_M as it is being simulated on input w. And finally, the third tape of U will contain state information.

- Initially, $< M, w >$ is on tape 1.
- Check tape 1 to make sure it is a valid input. That is, valid codes for M begin and end with "111" and each code $code_i$ is separated by 11, and the transitions are of the form $0^i 10^j 10^k 10^l 10^m$, where $m = 1$ or 2.
- Initialize tape 2 to contain $\$w$.
- Initialize tape 3 to contain $\$0$ (this is the start state, q_1, in unary).
- Initialize tape 1 to hold $11\$code_1 11 code_2 11 \dots 11 code_r 111$
- Repeat: (a) If tape 3 holds $\$000$ (rejecting state q_3, halt and reject; (b) if tape 3 holds $\$00$ (accept state q_2), halt and accept; (3) otherwise let x_j be the symbol scanned by head 2 and let 0^i be the contents of tape 3. Scan tape 1 from \$ to 111, looking for a string beginning with $110^i 10^j 1$. If no such string is found, halt and reject. If such a string is found, say it is $110^i 10^j 10^k 10^l 10^m$. Put 0^k on tape 3, and write x_l on the tape cell scanned by head 2, and then move that head in direction D_m one cell.

Notation: $\{z\}$ = the program $\mathcal{P}$ s.t. $\#(\mathcal{P}) = \hat{z}$

$$\text{Thus } \{z\} = \begin{cases} \text{the program } P \text{ such that } \#(P) = z \text{ if } P \text{ exists} \\ \Lambda \text{ (empty program) otherwise} \end{cases}$$

Kleene T predicate (Important)

Definition: For each $n \geq 1$ we define the n+2-ary relation T_n by the condition $T_n(z, x_1, \cdots, x_n, y)$ holds iff y codes the computation of program $\{z\}$ on input $\vec{x}$. For $n = 1$ we sometimes write T instead of T_1.

Theorem: (Kleene) For each $n \geq 1$ T_n is a recursive relation.

Proof: $T_n(z, \vec{x}, y)$ holds iff y codes a computation $(u_0, ..., u_t)$, where the initial state u_0 satisfies

$$u_0 = p_0^0 p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n}$$

and for all $i < t$

$$u_{i+1} = Nex(u_i, z)$$

and $halt(u_t, z)$ (the last state is halting) and for all $i < t$

$$\neg halt(u_i, z)$$

(no intermediate state is halting).

More formally, setting $t = lh(y) \dot{-} 1$ (so $t = \max i \leq y[p_i | y]$)

$$\begin{aligned} T_n(z, \vec{x}, y) = [&(y)_0 = p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n} \quad \{\text{initial state}\} \\ &\wedge \forall i < t[(y)_{i+1} = Nex((y)_i, z)] \\ &\wedge halt((y)_t, z) \ \{\text{last state is halting}\} \\ &\wedge \forall i < t \neg halt((y)_i, z) \ \{\text{no intermediate state is halting}\} \ \Box \end{aligned}$$

Output function

We define the output function $U(y)$ to be the contents of register R_1 in the final state of the computation coded by program $\{y\}$.

Thus $U(y) = ((y)_{lh(y) \dot{-} 1})_1$, and hence U is primitive recursive.

Notation: $\{z\}_n$ is the n-ary function computed by program $\{z\}$

Kleene Normal Form Theorem: There is a primitive recursive function U and for each $n \geq 1$ a primitive recursive predicate T_n such that

$$\{z\}_n(x_1, \ldots, x_n) = U(\mu y T_n(z, \vec{x}, y))$$

Proof: Immediate from the definitions above. Note: The least y is the only y satisfying the condition $T_n(...)$. Also note that y will not exist if the program doesn't halt, so $\{z\}_n(\vec{x})$ is undefined in this case.

Corollary: Every computable function is recursive, and can be obtained using at most one application of μ.

Universal Functions

Notation: $\Phi_n(z, \vec{x}) = \{z\}_n(\vec{x})$

Φ_n is called a universal function, since it codes every computable function of n variables, as z varies.

Corollary: The universal function Φ_n is recursive (and hence computable), for $n = 1, 2, ...$

A program computing Φ_n is called an *interpreter.*

The function Φ_1 is universal for the set of all unary computable functions. Thus if we define $\phi_i(x) = \Phi_1(i, x)$ then

$$\phi_0, \phi_1, ...$$

is an enumeration of all (partial) unary computable functions. It turns out that it is essential to include nontotal functions in order to get a computable universal function.

Theorem: There is no computable universal function for the set of all *total* computable unary functions.

Proof: Let $f_0, f_1, ...,$ be a list of all total computable unary functions, in any order, possibly with repetitions. Let

$$F(z, x) = f_z(x)$$

We will show that F is not computable. This is because if F were computable, then the "diagonal" function

$$D(x) = F(x, x) + 1 = f_x(x) + 1$$

would also be a total computable unary function. But then D must be in the list $f_0, f_1, ...$ That is, $D = f_e$ for some e. But then $f_e(e) = D(e) = f_e(e) + 1$, a contradiction. Hence F is not computable.

Exercise 2 *Prove that there is no computable universal relation $RU(x, y)$ for all computable unary relations.*

Exercise 3 *Let $A(n, x) = A_n(x)$ be Ackermann's Function (page* **??***). Define*

$$UP(z, x) = U(\min y < (A((z)_0, x) + z)\ T((z)_1, x, y))$$

where $T(z, x, y)$ is the Kleene T-predicate. (See page **??** *for the notation $(z)_x$.) (Compare the definition of UP with the Kleene Normal Formal Theorem. Use the facts stated about Ackermann's function together with results above to prove the following.*

(a) Prove that UP is a total computable function.

(b) Prove that for each $e \in \mathbb{N}$, the unary function $g_e(x) = UP(e, x)$ is primitive recursive.

(c) Prove that for each unary primitive recursive function $f(x)$ there is $e \in \mathbb{N}$ such that $f = g_e$ (where g_e is defined in part (b)). Use the Kleene Normal Form Theorem, and the following strengthening of the Theorem, page **??***:*

Fact: *Every primitive recursive function $f(\vec{x})$ is computable by a RM program $\mathcal{P}$ such that the function $Comp_{\mathcal{P}}(\vec{x})$ is primitive recursive, where*

$$Comp_{\mathcal{P}}(\vec{x}) \text{ is the number coding the computation of } \mathcal{P} \text{ on input } \vec{x}$$

(d) Give a diagonal argument showing that UP is not primitive recursive.

Exercise 4 *Use a diagonal argument to prove that*

$$H(x) = \mu y T(x, x, y)$$

has no total computable extension. That is, show that if the function $BIG(x)$ is total and agrees with $H(x)$ whenever $H(x)$ is defined, then $BIG(x)$ is not computable.

Church's Thesis: Every "algorithmically computable function" is computable (i.e. computable on a RM).

This statement cannot be proved because it is not precise. But it is a strong claim about the robustness of our formal notion of computable function. In general, if we give an informal algorithm to compute a function, then we can claim that it is computable, by Church's Thesis.

Alonzo Church proclaimed this famous "thesis" in a footnote to a paper in 1936. Actually, he did not talk about RM's, but rather claimed that every algorithmically computable function is definable using the λ-calculus which he had invented. A little later Alan Turing published his famous paper defining what are now called Turing machines, and argued, more convincingly than Church, that every algorithmically computable function is computable on a Turing machine. (Hence "Church's Thesis" is sometimes called the "Church-Turing thesis".) Turing proved that Church's λ-definable functions coincide with the Turing computable functions. It turns out that both of these coincide with the functions computable on a RM, which we have shown coincide with the recursive functions. In fact, many other formalisms for defining algorithmically computable functions have been given, and all of them turn out to be equivalent. This robustness is a powerful argument in favour of Church's thesis.