Computer Science 513

First Hard Problems

Instructor: Wayne Eberly

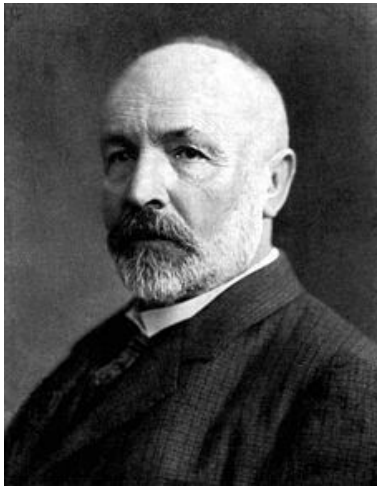
Department of Computer Science
University of Calgary

Lecture #10

Goals for Today

- Review (or introduce) ***diagonalization*** as a proof technique.
- Use this to identify a first function that is, provably, not computable.
- Continue the development of technical results, based on the material from the previous lecture, that will help us to use diagonalization to prove that other functions are not computable, as well.
- Introduce “computable sets” and “computably enumerable sets” — identifying sets that are provably not computable, and provably not computably enumerable, as well.

Diagonalization



- The ***Diagonalization Method*** was discovered by the mathematician Georg Cantor in 1873.

Diagonalization

- Cantor was interested in comparing the "sizes" — or ***cardinalities*** — of infinite sets.
- Cantor used diagonalization to prove that various infinite sets have *different* cardinalities — in a sense, some infinite sets are "bigger" than others.
- For example, $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ have the *same* cardinality — they are all "countable" (even though $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q}$) — but $\mathbb{R}$ is an "uncountable" infinite set — it has a higher cardinality than $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{Q}$.
- We will use diagonalization to identify a first predicate that is provably not computable.

Diagonalization

- The ***objective*** of diagonalization is generally to prove that a certain *function* $g : \mathbb{N} \to \mathbb{N}$ (respectively, set $T \subseteq \mathbb{N}$) does not belong to a certain *class* $\mathcal{C}$ of functions (respectively, sets).
- The ***first step*** in the use of this method is to identify an ***enumeration*** (or "listing") of the elements of $\mathcal{C}$ —- that is, to describe a way to list $\mathcal{C}$ as

$$\mathcal{C} = \{f_0, f_1, f_2, f_3, \dots\}$$

if $\mathcal{C}$ is a class of ***functions***, and to list $\mathcal{C}$ as

$$\mathcal{C} = \{S_0, S_1, S_2, S_3, \dots\}$$

if $\mathcal{C}$ is a class of ***sets*** – such that every element of $\mathcal{C}$ appears ***at least*** once in this listing.

Diagonalization

- This makes it possible to introduce and use a two-dimensional ***table:***

	0	1	2	3	4	...
f_0 (or S_0)						
f_1 (or S_1)						
f_2 (or S_2)						
f_3 (or S_3)						
f_4 (or S_4)						
$\vdots$						

- As shown above, ***rows*** of the table correspond to elements of $\mathcal{C}$ — using the enumeration of $\mathcal{C}$ that has been selected. ***Columns*** of the table correspond to the elements of $\mathbb{N}$, listed in the usual order.

Diagonalization

- Suppose that $\mathcal{C}$ is a class of ***functions***. Then, for $i, j \in \mathbb{N}$, the entry of the table in row i and column j of this table is

$$f_i(j)$$

  — possibly $\uparrow$, if f_i is a partial function.

- On the other hand if $\mathcal{C}$ is a class of ***sets*** then, for $i, j \in \mathbb{N}$, the entry of the table in row i and column j of this table is

$$p_{S_i}(j)$$

  where $p_{S_i} : \mathbb{N} \to \{0, 1\}$ is the ***characteristic function*** of S_i. That is, this entry of the table is 1 if $j \in S_i$ and is 0 if $j \notin S_i$.

Diagonalization

- One should now assume that g (or T) is in $\mathcal{C}$, in order to obtain a ***contradiction***.
- If $\mathcal{C}$ is a class of *functions* then this assumption should be used to describe another function $f_C : \mathbb{N} \to \mathbb{N}$ such that
  - $f_C \in \mathcal{C}$, but
  - $f_C(n) \neq f_n(n)$ for every integer $n \geq 0$.

  Thus $f_C \neq f_n$ — because these functions disagree at the n^{th} ***diagonal*** entry in the above table. Therefore $f_C \notin \mathcal{C}$ after all — and a ***contradiction*** has been obtained.

Enumerating $\mathcal{S}$-Programs

- Recall, from the previous lecture, that every $\mathcal{S}$-program $\mathcal{P}$ can be represented, or "encoded, by a natural number, $\#(\mathcal{P})$.
- While multiple $\mathcal{S}$ programs can have the same encoding, all the $\mathcal{S}$-programs encoded by a given number $n \in \mathbb{N}$ compute the same partial or total function because the only difference between these programs concerns the number of copies of the unlabelled "dummy" statement

$$Y \leftarrow Y$$

that they end with.

Enumerating Turing-Computable Functions

- In particular, for every number $n \in \mathbb{N}$, there exists a unique $\mathcal{S}$-program, $\mathcal{P}_n$, such that $\#(\mathcal{P}_n) = n$ and $\mathcal{P}$ does not end with the unlabelled "dummy" statement $Y \leftarrow Y$.
- Since every partial or total function is Turing-computable if and only if it is $\mathcal{S}$-computable, this gives a way to list — or "enumerate" — all the Turing-computable partial or total functions from $\mathbb{N}$ to $\mathbb{N}$ as

$$f_0, f_1, f_2, \ldots$$

where $f_n : \mathbb{N} \to \mathbb{N}$ is the function, with one input, that is computed by the $\mathcal{S}$-program $\mathcal{P}_n$, for all $n \in \mathbb{N}$.

The "Halting Predicate" is not Computable

Let us consider one more total predicate, HALT : $\mathbb{N}^2 \to \{0, 1\}$. For all $x, y \in \mathbb{N}$, if $y = \#(\mathcal{P})$ then HALT$(x, y) = 1$ if $\mathcal{P}$ halts when executed on the input x — and HALT$(x, y) = 0$, otherwise.

Claim: HALT is ***not*** a computable predicate.

Proof:

- By contradiction. Suppose that this predicate ***is*** computable.

The "Halting Predicate" is not Computable

- Then there exists some $\mathcal{S}$-program $\mathcal{Q}$ that computes the same partial function, $f_c : \mathbb{N} \to \mathbb{N}$, as the following — which uses a macro for HALT.

$$\begin{array}{rl} & Z_1 \leftarrow \text{HALT}(X_1, X_1) \\ [A_1] & \text{IF } Z_1 \neq 0 \text{ GOTO } A_1 \end{array}$$

- Let $n = \#(\mathcal{Q})$. Either $\mathcal{Q}$ halts on the input n or it does not.

The "Halting Predicate" is not Computable

- ***Case:*** $\mathcal{Q}$ ***does*** halt when on executed on input n
  - Then Z_1 receives the value 1 after the first statement, above, is executed.
  - Consequently, the above program fails to terminate — because the second statement is an infinite loop.
  - The partial function computed by the above program — and $\mathcal{Q}$ — is **undefined** at value n.
  - Thus $\mathcal{Q}$ ***does not*** halt when executed on the input n.

We may therefore conclude that this first case is impossible — $\mathcal{Q}$ cannot halt when executed on the input n.

The "Halting Predicate" is not Computable

- ***Case:*** $\mathcal{Q}$ ***does not*** halt when executed on input n.
  - Then Z_1 receives the value 0 after the first statement, above, is executed.
  - The above program therefore halts after the execution of one more statement — a single execution of the second instruction, shown above.
  - The partial function computed by this program —and $\mathcal{Q}$ — is, therefore, defined at value n.
  - Thus $\mathcal{Q}$ ***does*** halt when expected on input n.

  Thus *this* case is impossible too!
- A ***contradiction*** has now been obtained, so the initial assumption must be incorrect: The predicate HALT is ***not*** computable. □
- ***Note:*** Thus this predicate is not ***primitive recursive***, either.

The "Halting Predicate" is not Computable

Note that ***diagonalization*** was used here:

- The class, $\mathcal{C}$, of functions considered here was the set of computable partial or total functions $f : \mathbb{N} \to \mathbb{N}$ — enumerated by the encodings of the $\mathcal{S}$-programs that compute them.
- The function, shown not to be in $\mathcal{C}$, was the function called "f_C" in the above proof.

Enumerating Computable Functions

For each integer $n \geq 1$, let $\Phi^{(n)} : \mathbb{N}^{n+1} \to \mathbb{N}$ be the function such that, for all $x_1, x_2, \ldots, x_n, y \in \mathbb{N}$,

$$\Phi^{(n)}(x_1, x_2, \ldots, x_n, y) = f_y(x_1, x_2, \ldots, x_n)$$

where $f_y : \mathbb{N}^n \to \mathbb{N}$ is the (partial) function computed by the program $\mathcal{P}$ such that $y = \#(\mathcal{P})$.

- Thus $f_0, f_1, f_2, \ldots$ is a ***enumeration*** (or "listing") of all computable partial or total functions from $\mathbb{N}^n$ to $\mathbb{N}$ — in which functions are listed more than once

Enumerating Computable Functions

- In *Computability, Complexity and Languages*, the above function $f_y : \mathbb{N}^n \to \mathbb{N}$ is often written as

$$\Phi_y^{(n)}(x_1, x_2, \ldots, x_n)$$

  instead. This notation will be in these notes too.
- Furthermore, the superscript is sometimes deleted when $n = 1$. That is $\Phi^{(1)}(x, y)$ is sometimes written as $\Phi(x, y)$, and $\Phi_y^{(1)}$ is sometimes written as Φ_y.

Universality Theorem

Universality Theorem: For every positive integer n, the above function $\Phi^{(n)} : \mathbb{N}^{n+1} \to \mathbb{N}$ is computable.

Proof: This is the partial function that is computed by the "universal $\mathcal{S}$-program" that was presented in the previous lecture. □

The Parameter Theorem

The following is a significant technical result that relates the various functions $\Phi^{(n)}$ for different values of n.

Parameter Theorem: For all positive integers n and m there exists a primitive recursive function

$$S_m^n : \mathbb{N}^{n+1} \to \mathbb{N}$$

such that, for all $x_1, x_2, \ldots, x_m, u_1, u_2, \ldots, u_n, y \in \mathbb{N}$,

$$\Phi^{(m+n)}(x_1, x_2, \ldots, x_m, u_1, u_2, \ldots, u_n, y) = \Phi^{(m)}(x_1, x_2, \ldots, x_m, S_m^n(u_1, u_2, \ldots, u_n, y)).$$

Note: This is also called the ***s-m-n Theorem,*** as well as the ***Iteration Theorem*** in some older texts.

The Parameter Theorem

What This Means:

- There is a primitive recursive function S_m^n mapping the encoding of any $\mathcal{S}$-program $\mathcal{P}$ to a mapping of another $\mathcal{S}$-program that — effectively — includes statements

$$\begin{aligned} X_{m+1} &\leftarrow u_1 \\ X_{m+2} &\leftarrow u_2 \\ &\vdots \\ X_{m+n} &\leftarrow u_n \end{aligned}$$

  before the *beginning* of $\mathcal{P}$ — so that if $\mathcal{P}$ has $m + n$ inputs, then this program has at most m inputs.

The Parameter Theorem

How This is Proved: By induction on n.

- If $n = 1$ then $S^n_m = S^1_m$ should be a function $S^m_1 : \mathbb{N}^2 \to \mathbb{N}$ such that, for all $u_1, y \in \mathbb{N}$, $S^1_m(u_1, y)$ is the encoding of a program starting with u_1 copies of the statement

$$X_{m+1} \leftarrow X_{m+1} + 1$$

and continuing with the program encoded by y. A consideration of the encodings of programs, introduced in the previous lecture, allows an expression for the value of $S^M_1(u_1, y)$ to be derived as a function of u_1 and y. This expression can be used to establish that this function is primitive recursive, as claimed.

The Parameter Theorem

- For the inductive step, it suffices to note that if k is an integer such that $k \geq 1$, and $S_m^k : \mathbb{N}^{k+1} \to \mathbb{N}$ is a primitive recursive function with the properties given in the claim, then the function $S_m^{k+1} : \mathbb{N}^{k+2} \to \mathbb{N}$ such that, for all $u_1, u_2, \ldots, u_{k+1}, y \in \mathbb{N}$,

$$S_m^{k+1}(u_1, u_2, \ldots, u_{k+1}, y) = S_m^k(u_1, u_2, \ldots, u_k, S_m^1(u_{k+1}, y))$$

  satisfies the desired properties, when k is replaced by $k + 1$, and where $S_m^1 : \mathbb{N}^2 \to \mathbb{N}$ is the function described in the basis. This function is easily shown to be primitive recursive if S_m^k is. □

The supplemental document includes a more detailed proof of this and other claims given in this lecture.

Characteristic Functions of Sets

Definition: Let n be an integer such that $n \geq 1$. The ***characteristic function*** of a set $S \subseteq \mathbb{N}^n$ is the predicate $p_S : \mathbb{N}^n \to \{0, 1\}$ such that, for all $x_1, x_2 \ldots, x_n \in \mathbb{N}$,

$$p_S(x_1, x_2 \ldots, x_n) = \begin{cases} 1 & \text{if } (x_1, x_2, \ldots, x_n) \in S, \\ 0 & \text{otherwise.} \end{cases}$$

Restricting Attention to the Case $n = 1$

- Henceforth we will restrict attention to the case that $n = 1$, so that we will consider subsets, S, of $\mathbb{N}$.
- Subsets of $\mathbb{N}^n$, for $n \geq 2$, are considered in the exercises for this lecture.

Computable Sets

Henceforth — following the convention in *Computability, Complexity and Languages* — we will say that "a set S belongs to a class of *functions*" if its *characteristic function* does. This leads to the following.

Definition: A set $S \subseteq \mathbb{N}$ is a ***computable set*** if its characteristic function $p_S : \mathbb{N} \to \{0, 1\}$ is a Turing-computable (or "recursive") predicate.

- In older references, these are also called ***recursive sets***. We will see later that recursive sets are closely related to ***decidable languages***, as these were defined in CPSC 351 (or 313).

Computably Enumerable Sets

Definition: A set $S \subseteq \mathbb{N}$ is a ***computably enumerable set*** if there exists a Turing-computable partial function $g : \mathbb{N} \to \mathbb{N}$ such that

$$S = \{x \in \mathbb{N} \mid g(x) \text{ is defined}\}.$$

- These also called ***recursively enumerable*** sets in older texts. It is standard (in older references) to abbreviate the phrase "recursively enumerable set" as ***r. e. set***.
- We will see later that computably enumerable sets are closely related to ***recognizable languages***, as these were defined in CPSC 351 or 313.

Relationships Between Computable and Computably Enumerable Sets

Proofs of the next results are straightforward — or closely resemble proofs for similar results that were (ideally) presented in CPSC 351 or 313. The supplemental document includes their proofs.

Claim #1: If a set $S \subseteq \mathbb{N}$ is ***computable*** then S is also ***computably enumerable.***

Claim #2: Let $S \subseteq \mathbb{N}$. If both S and its complement $\overline{S}$ are computably enumerable, then the set S is computable.

Closure Properties for Computably Enumerable Sets

Claim #3: If $S, T \subseteq \mathbb{N}$ are both computably enumerable sets, then so is $S \cup T$.

- The proof of this is an ***exercise*** that will be included in the set of practice problems for this lecture.
- ***Hint:*** Modify the proof of the previous result.

Closure Properties for Computably Enumerable Sets

The proof of the following is even simpler — and is included in the supplement for this lecture.

Claim #4: If $S, T \subseteq \mathbb{N}$ are both computably enumerable sets, then so is $S \cap T$.

Enumeration Theorem

For all $n \in \mathbb{N}$, let

$$W_n = \{x \in \mathbb{N} \mid \text{The program } \mathcal{P}, \text{ such that } n = \#(\mathcal{P}), \text{ halts on input } x\}.$$

- This is defined in *Computability, Complexity and Languages*, somewhat differently, as follows:

$$W_n = \{x \in \mathbb{N} \mid \Phi^{(1)}(x, n) \text{ is defined}\}$$

  where the partial function $\Phi^{(1)} : \mathbb{N}^2 \to \mathbb{N}$ is as defined earlier in this lecture.

- It really does follow, immediately, from the definition given for $\Phi^{(1)}$ earlier on, that the above definitions of W_n are equivalent.

Enumeration Theorem

Enumeration Theorem: A set S is computably enumerable if and only if there exists an integer n such that $S = W_n$.

- The proof of this "theorem" is trivial once you notice that the definitions of W_n, given on the previous slide, are equivalent — this "theorem" is essentially just a restatement of the definition of a "computably enumerable set."
- This theorem gets its name because it asserts that the sequence

  $$W_0, W_1, W_2, \ldots$$

  is an enumeration of all of the recursively enumerable sets (with every such set actually being repeated infinitely often).

A Computably Enumerable Set That is Not Computable

Let $K \subseteq \mathbb{N}$ such that

$$K = \{n \in \mathbb{N} \mid n \in W_n\} =$$
$$\{n \in \mathbb{N} \mid \text{the program whose encoding is } n \text{ halts on input } n\}.$$

A Computably Enumerable Set That is Not Computable

Claim #5: The above set K is computably enumerable.

Proof:

- Consider the partial or total function that is computed by the following program.

$$
\begin{array}{ll}
 & Z_1 \leftarrow 0 \\
 & Z_2 \leftarrow X_1 \\
[A_1] & Z_2 \leftarrow \mathrm{Halt}_1(X_1, Z_2, Z_1) \\
 & \text{IF } Z_2 \neq 0 \text{ GOTO } A_2 \\
 & Z_1 \leftarrow Z_1 + 1 \\
 & \text{GOTO } A_1
\end{array}
$$

A Computably Enumerable Set That is Not Computable

- The partial or function function $g : \mathbb{N} \to \mathbb{N}$ computed by this program *is* certainly computable, and for all $x \in \mathbb{N}$,

$$g(x) = \begin{cases} 0 & \text{if } x \in K \\ \uparrow & \text{otherwise.} \end{cases}$$

- It follows by the definition of a "computably enumerable set" — using the above partial or total function — that K is computably enumerable, as claimed. □

A Computably Enumerable Set That is Not Computable

Claim #6: $\overline{K}$ is ***not*** computably enumerable.

Proof: By contradiction.

- Suppose that $\overline{K}$ is computably enumerable.
- It follows by the Enumeration Theorem that $\overline{K} = W_n$ for some natural number n.

A Computably Enumerable Set That is Not Computable

- It now follows that

$$
\begin{aligned}
n \in K \iff & \text{ The program encoded by } n \text{ halts on input } n \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(by the definition of } K\text{)} \\
\iff & \ n \in W_n \qquad\qquad\qquad\qquad \text{(by the definition of } W_n\text{)} \\
\iff & \ n \in \overline{K} \qquad\qquad\qquad\qquad \text{(since } \overline{K} = W_n\text{).}
\end{aligned}
$$

- This is certainly a ***contradiction*** — establishing that $\overline{K}$ is not computably enumerable, as claimed. □

A Computably Enumerable Set That is Not Computable

Claim #7: K is not computable.

Proof: By contradiction.

- Suppose that K is computable. Then the characteristic function $p_K : \mathbb{N} \to \{0, 1\}$ is Turing-computable.
- However, this implies that the characteristic function $p_{\overline{K}} : \mathbb{N} \to \{0, 1\}$ of $\overline{K}$ is Turing-computable, since $p_{\overline{K}} = \neg p_K$, and the set of Turing-computable functions is a PRC class.
- It now follows that $\overline{K}$ is computable. It is therefore computably enumerable, as well — but this ***contradicts*** the result that has just been established.
- Therefore K is not computable □

Computably Enumerable Sets: Closure Properties

Note: We have now shown that the class of computably enumerable sets is provably *closed* under

- union and
- intersection,

but that this class of sets is provably *not closed* under

- complementation.

Indeed, K is computably enumerable but $\overline{K}$ is not.

Exactly the same results can be established for the class of Turing-recognizable languages that are studied in CPSC 351.

Computably Enumerable Sets: Alternative Characterizations

The proofs of the next few claims are reasonably straightforward — and are included in the supplement for this lecture. They are less important than the results that they establish.

With that noted, the next result is proved using one the primitive recursive "Halt" predicates that have already been introduced.

Claim #8: Let $S \subseteq \mathbb{N}$ be a computably enumerable set. Then there exists a primitive recursive predicate $R : \mathbb{N}^2 \to \{0, 1\}$ such that, for all $x \in \mathbb{N}$, $x \in S$ if and only if there exists a number $t \in \mathbb{N}$ such that $R(x, t) = 1$.

Computably Enumerable Sets: Alternative Characterizations

Claim #9: Let $S \subseteq \mathbb{N}$ be a ***nonempty*** computably enumerable set. Then there exists a primitive recursive function $f : \mathbb{N} \to \mathbb{N}$ such that

$$S = \{f(n) \mid n \in \mathbb{N}\},$$

that is, S is the ***range*** of the primitive recursive function f.

Claim #10: Let $f : \mathbb{N} \to \mathbb{N}$ be a Turing-computable partial or total function and let

$$S = \{f(x) \mid x \in \mathbb{N} \text{ and } f(x) \text{ is defined}\},$$

so that S is the ***range*** of a Turing-computable partial or total function. Then S is computably enumerable.

Computably Enumerable Sets: Alternative Characterizations

Theorem: Suppose that S is a ***nonempty*** subset of $\mathbb{N}$. Then the following are equivalent:

(a) S is computably enumerable.

(b) S is the range of some primitive recursive function.

(c) S is the range of some Turing-computable total function.

(d) S is the range of some Turing-computable partial or total function.

Computably Enumerable Sets: Summary

Proof:

- It follows by Claim #9 that $(a) \Rightarrow (b)$.
- It should be clear — by considering the kinds of functions mentioned — that $(b) \Rightarrow (c)$, and $(c) \Rightarrow (d)$.
- Finally, Claim #10 establishes that $(d) \Rightarrow (a)$. □