Oracle Turing Machines and the Arithmetical Hierarchy

Dr. K.W. Regan, Spring 1994 class notes, adapted for CSE596, Fall 2008

Abstract

These notes supplement Section 3.9 of the Homer-Selman text with more material on *reductions* among problems, and on several concepts that are also useful in complexity theory.

1. Oracle Machines and Turing Reductions

An *oracle Turing machine* $M = (Q, \Sigma, \Gamma, \delta, \Delta, s, F, O)$ is defined just like a normal Turing machine, except that $O \subseteq Q$ is a special collection of *oracle query states*. The actions of M in a query state are controlled not by the character read on the tape, but by an "external module" which consists of a language A called the *oracle set*. The oracle A can be any given language of strings over some alphabet $\Lambda \subseteq \Gamma$. (Usually we have $\Lambda = \Sigma$, and usually $\Sigma = \{0, 1\}$.) We'll assume that all OTMs discussed here are deterministic. The *purpose* of oracle TMs is to provide a formal model with which to study questions of the form, "If I could solve X, what else could I do?"

To specify how this works in an intuitive manner, let us suppose that M uses the tape delimiters $\wedge$ and \$, and that Γ also contains the characters '?', '0' (for 'no'), and '1' (for 'yes'). (Note: $\wedge, \$, ?$ are not in Λ.) Every arc into a query state $q_? \in O$ has the form $(p, c, ?, S, q_?)$, while there are exactly two arcs out of the query state, one on '0' and one on '1'. When M enters state $q_?$, the *query string* z is identified just like in the "relaxed output convention": z is the longest consecutive string of characters *over* Λ beginning in the cell to the right of the $\wedge$ marker. If z belongs to the oracle set A, then the oracle module "magically" overwrites the '?' by a '1', and the arc on '1' is taken. If $z \notin A$, then the oracle responds '0', and the '0' arc is taken. After that, computation proceeds as normal, up until another query state is entered.

Definition 1.1. A language A *Turing-reduces* to a language B, written $A \leq_T B$, if there is an oracle Turing machine M such that M is total with oracle B, and $L(M^B) = A$.

This is like the definition of A being recursive, except for the presence of the oracle set B. It is also customary to say that A is *recursive in* B. In our present example, the Halting Problem is recursive in the emptiness problem for 2HDFAs; i.e., $HP \leq_T$ 2HDFA-$\emptyset$. Turing did indeed invent the idea of an oracle TM and a Turing reduction. In this particular case, M_0 only needs *one* (1) oracle call and follows a pre-set *truth-table* of what to say when the oracle responds: here, $0 \mapsto$ ACCEPT and $1 \mapsto$ REJECT, and so we can be more specific and write $HP \leq_{1tt} X$.

In fact, the only examples I know of languages B, A such that $B \leq_T A$ but not in one oracle call are too technical to give here. Consider $B :=$ "HP twice" $:= \{ code(M_1)code(M_2)x_1\#x_2 : M_1(x_1) \downarrow \wedge M_2(x_2) \downarrow \}$. You might think this requires two calls to an oracle X such as $X = K$ to solve, but you can do it in one call by first combining M_1 and M_2 into a machine M_3 that halts iff both of them do, and then asking a single question relevant to M_3.

Now suppose that we replace an oracle language X by its complement $\widetilde{X}$. Then we could switch the wires at the query state to change M_0 to an OTM $\widetilde{M_0}$ whose responses to the oracle are $1 \mapsto$ ACCEPT and $0 \mapsto$ REJECT. That is, $\widetilde{M_0}$ behaves like "Simon the oracle says: accept w iff $z_w \in \widetilde{X}$." Then we can be even more specific and write $HP \leq_m \widetilde{X}$. This is read, "HP *many-one* reduces to $\widetilde{X}$." (The 'many' allows for the possibility that many different input strings w might lead to the same one query string z_w, but usually in practice z_w uniquely depends on w, and so we could actually speak of a *one-one* reduction, written $HP \leq_1 \widetilde{X}$. But "many-one" carries the specifics far enough for my purposes.) In lecture you will have seen the equivalent definition:

Definition 1.2. A language A *many-one reduces* to a language B, written $A \leq_m B$, if there exists a total recursive function f such that for all strings w, $w \in A \iff f(w) \in B$.

2. Relativized Language Classes

Now we can adapt the notation used for the classes of recursive and *r.e.* languages to the case of oracle machines. For *any* oracle set A:

(a) $\text{RE}^A := \{L$: there is an OTM M such that $L(M^A) = L\}$.

(b) $\text{REC}^A := \{L$: there is an M such that $L(M^A) = L$ and M^A is total$\}$.

(c) co-$\text{RE}^A := \{$complements of languages in $\text{RE}^A\}$.

Many theorems about the classes REC and RE go through untouched by the presence of a given oracle set A. That is to say, they *relativize*. Some examples:

Theorem 2.1. *For any oracle set A, REC^A is closed under complements.*

Proof. Given $L \in \text{REC}^A$, this means there is an OTM M_0 such that M_0 is total with oracle set A and $L(M_0^A) = L$. Presuming that M_0 has been put into the normal form which makes M_0 "crash-free" and gives M_0 distinguished accept and reject states q_{acc} and q_{rej}, let M_1 be obtained from M_0 by interchanging q_{acc} and q_{rej}. Since M_A^0 is total, M_1^A is total. And $L(M_1^A) = \widetilde{L}$ (the complement of L). So $\widetilde{L} \in \text{REC}^A$ too. □

Theorem 2.2. *For any oracle set A, $\text{RE}^A \cap$ co-$\text{RE}^A = \text{REC}^A$.*

Proof. First suppose $L \in \text{REC}^A$. Then clearly $L \in \text{RE}^A$. By Theorem 2.1, $\widetilde{L} \in \text{REC}^A$, so $\widetilde{L} \in \text{RE}^A$ too. If $\widetilde{L} \in \text{RE}^A$, then by definition $L \in$ co-RE^A. So $L \in \text{RE}^A \cap$ co-RE^A.

Going the other way, let $L \in \text{RE}^A \cap$ co-RE^A. Then there are oracle TMs M_1 and M_2 such that $L(M_1^A) = L$ and $L(M_2^A) = L$. Thus on any input x, exactly one of M_1 and M_2 has an accepting computation on input x. Build an OTM M_3 which on any input x makes two copies of x, simulates one step of M_1 on the first copy, one step of M_2 on the other copy, then back for one more step of M_1 on the first copy, and so on. (The copies can be marked off with separate delimiters $\wedge_1, \$_1, \wedge_2, \$_2$; and if $\$_1$ ever runs up against $\wedge_2$, call the "make-more room" procedure from class.) If the step of M_1 or M_2 enters a query state of M_1 or M_2, then M_3 has the same query state, and the oracle response works the same. M_3^A is total because exactly one of M_1^A and M_2^A will stop and accept; if it is M_1, then M_3 accepts, and if it is M_2 that accepts, then M_3 rejects. (If either M_1 or M_2 rejects ahead of time, M_3 takes the appropriate action.) Since $L = L(M_3^A)$, $L \in \text{REC}^A$. □

This proof too is really just like the one given in lecture for the non-oracle case. Now for two results which say when you can do away with the oracle, and when you can't:

Theorem 2.3. *If* $A \in \mathrm{REC}$, *then* $\mathrm{RE}^A = \mathrm{RE}$ *and* $\mathrm{REC}^A = \mathrm{REC}$.

Proof. Given that A is recursive, there is a total non-oracle TM MA accepting A. Let M_0 be an oracle TM which accepts L with oracle A. Then instead of making $q_?$ in M_0 a query state, run an arc $(q_?, ?, \wedge, R, s_A)$ to the start state s_A of M_A, wire the arcs

$$\{\, (f_A, \wedge, 1, S, q_?), (f_A, \mathit{otherwise}, \mathit{same}, L, f_A) \,\}$$

to the accept state f_A of M_A (this moves left until it finds the *new* $\wedge$ and changes it to a '1'), and do likewise for the reject state r_A of M_A and '0'. This changes M_0 to a *non-oracle* TM M_1 which still accepts L (since the "subroutine" M_A always terminates), and M_1 is total if M_0^A is total. Thus the conclusions about L follow. □

Theorem 2.4. *It is* not *the case that for all* $B \in \mathrm{RE}$, $\mathrm{RE}^B = \mathrm{RE}$.

Proof. This time, let us take B to be the language of the *Acceptance Problem*; namely, $AP := \{\, \mathit{code}(M)x : M \text{ accepts } x \,\}$. Define $AP(B) := \{\, \mathit{code}(M)x : x \in L(M^B) \,\}$. Then $AP(B) \in \mathrm{RE}^B$. Now the proof that $AP \notin \mathrm{REC}$ from class is not affected at all by the presence or absence of the oracle set B. To wit: If $AP(B) \in \mathrm{REC}^B$, then the complement $\widetilde{AP}(B)$ of $AP(B)$ is in REC^B, by Theorem 2.1. Hence the language $D(B) := \{\mathit{code}(M) : M \text{ is an OTM such that with oracle } B, M^B \text{ does not accept } \mathit{code}(M)\}$ also belongs to REC^B. That means there is a TM M_1 such that $L(M_1^B) = D(B)$. What does M_1 with oracle B do on input $x := \mathit{code}(M_1)$, pray tell? If it accepts, then $\mathit{code}(M_1) \in D(B)$ [by $L(M_1^B) = D(B)$], but by definition of $x \in D(B)$, M_1^B does not accept $\mathit{code}(M_1)$. And if M_1^B does not accept $x = \mathit{code}(M_1)$, then $x \in D(B)$, so it should have accepted. Either way there's a contradiction. Thus we conclude: $AP(B) \notin \mathrm{REC}^A$.

Now the key observation is that RE is contained in REC^B—because for any *r.e.* language L, you can take a fixed TM M_L accepting L, and given any instance x of the decision problem for L, convert it to the oracle query $\mathit{code}(M)x$ for B $(= AP)$, which gives the answer you want. So from above, we have $AP(B) \notin \mathrm{RE}$. Hence $AP(B)$ itself is a language which shows that for $B := AP$, $\mathrm{RE}^B \neq \mathrm{RE}$. □

For some final notation, given any *class* $\mathcal{C}$ of languages, define

$$\mathrm{RE}^{\mathcal{C}} := \{\, L(M^A) : A \in \mathcal{C} \,\}, \quad \text{and} \quad \mathrm{REC}^{\mathcal{C}} := \{\, L(M^A) : A \in \mathcal{C} \text{ and } M^A \text{ is total} \,\}.$$

Theorem 2.3 showed that $\mathrm{RE}^{\mathrm{REC}} = \mathrm{RE}$. Then we can inductively define the classes

$$\textstyle\sum_0^0 := \mathrm{REC}, \quad \sum_1^0 := \mathrm{RE}, \quad \text{and for } k \geq 1, \quad \sum_k^0 := \mathrm{RE}(\sum_{k-1}^0).$$

Also define $\prod_k^0 :=$ co-$\sum_k^0$ for each k; thus $\prod_0^0 = \mathrm{REC}$ and $\prod_1^0 =$ co-RE. We could also write *e.g.* $\sum_2^0$ as $\mathrm{RE}^{\mathrm{RE}}$, $\sum_3^0$ as $\mathrm{RE}(\sum_2^0)$, and so on. Further notation is: $\Delta_0^0 = \Delta_1^0 = \mathrm{REC}$, and for $k \geq 2$, $\Delta_k^0 = \mathrm{REC}(\sum_{k-1}^0)$. These classes taken together form the so-called *arithmetical hierarchy*. The '0' on top stands for "type-0 arithmetic." The continuation of this handout will explain what this means, and where the name "Arithmetical Hierarchy" came from.

3. Predicates

Let $\Sigma := \{0, 1\}$, and let us identify strings in Σ^* with natural numbers in dyadic notation. Let $\Sigma' := \Sigma \cup \{\#\}$, where $\#$ is a new *separator symbol*. If we want to be purists, we can re-code Σ' back over Σ^* by mapping $0 \to 00$, $1 \to 11$, $\# \to 01$. Later on we *will* be purists!

Call an m-place predicate $R(\cdot, \cdot, \ldots, \cdot)$ *decidable* if the language $L_R := \{w_1 \# w_2 \# \ldots \# w_m :$ each w_i is in Σ^* and $R(w_1, \ldots, w_m)$ holds$\}$ is recursive. As a matter of convention, we will allow variables to refer *only* to strings in Σ^*. Here are three important examples of decidable predicates:

(I) The *Kleene T-predicate:* $T(code(M), x, \vec{c}) \equiv \vec{c}$ is a valid halting computation of the TM M on input x.

(II) The *Kleene U-predicate:* $U(\vec{c}, z) \equiv \vec{c}$ is the code of a valid halting computation, and z is the output of $\vec{c}$ according to the "relaxed" output convention.

(III) The *acceptance predicate:* $Acc(code(M), x, \vec{c}) \equiv M$ is an acceptor and $\vec{c}$ is a valid accepting computation of M on input x.

Technotes: It's getting tiring to write $code(M)$ all the time. So let's write $T(v, x, \vec{c})$ for the assertion that M_v on input x has the valid halting computation $\vec{c}$. If a string $v \in \Sigma^*$ is not the code of a TM, then let's consider M_v to be a TM which has no halting computations at all. After identifying strings v with natural numbers i, we'll soon take up the habit of writing M_i and $T(i, x, \vec{c})$. Stephen Kleene actually wrote U as a function of $\vec{c}$ and wrote the predicate as '$U(\vec{c}) = z$.' The symbol U might cause confusion with the universal Turing machine; I didn't want to change it. If we identify acceptors with transducers that compute characteristic functions, then we could define $Acc(v, x, \vec{c})$ as $T(v, x, \vec{c})\&U(\vec{c}, 1)$, or more intuitively, as "$T(v, x, \vec{c}) \ \wedge \ U(\vec{c}) =$ ACCEPT."

Note that these predicates are not only decidable, they are "easy" in that they can be verified by a two-head DFA. Historically, the role of "easy" was applied to the so-called *primitive recursive* functions and languages, but from the modern point of view of complexity theory, there are primitive recursive languages which are really hard to decide. Every 2HDFA language, however, requires at most $2n$ steps on inputs of length n, and this is pretty easy. The only fact you need to know about 2HDFAs is that a 2HDFA can decide whether a sequence of IDs $I_0 \# I_1 \# \ldots \# I_t$ represents a valid halting computation by a single-tape Turing machine M, by starting with one head in I_0 and the other in I_1 checking that any difference follows by a valid move of M, and carrying on with the heads straight across to I_t. The language of valid halting computations by M is called $VALCOMPS(M)$, and when M is a universal TM, just $VALCOMPS$. The following fact tells us that we will be able to base our theory on "easy" predicates.

Proposition 3.1. *Suppose $R(x, y)$ is a decidable predicate. Then we can find 2HDFA-computable predicates $R_1(x, z)$ and $R_2(x, z)$ such that for all fixed $x \in \Sigma^*$:*

(a) $(\exists y\, R(x, y)) \iff (\exists z\, R_1(x, z))$

(b) $(\forall y\, R(x, y)) \iff (\forall z\, R_2(x, z))$.

Proof. Let r be the code of a total TM which accepts L_R. We may presume that M_r is deterministic. Then for all $x \in \Sigma^*$:

(a) $(\exists y\, R(x, y)) \iff (\exists y)(\exists \vec{c})\ Acc(r, x\#y, \vec{c})$, and

(b) $(\forall y\, R(x, y)) \iff (\forall y)(\forall \vec{c})[Acc(r, x\#y, \vec{c}) \ \vee \ \neg T(r, x\#y, \vec{c})]$.

To see that the equivalence in (b) holds, read the right-hand side as saying "for all y, the only halting computations of M_r on input $x\#y$ are accepting." Since M_r is total, M_r always has a halting computation on any input $x\#y$, so this is equivalent to "for all y, M_r accepts $x\#y\#$," which in turn is equivalent to the left-hand side. To build a 2HDFA which decides the part in $[\ldots]$, take S from the lecture on *VALCOMPS* and 2HDFA's, but re-program S to accept iff its input is *not* the code of a *rejecting* computation.

You may have one quibble: the right-hand sides are supposed to have single quantifiers '$(\exists z)$' and '$(\forall z)$', not the double quantifiers over y and $\vec{c}$. However, two adjacent *like quantifiers* can always be *coalesced* into one. Formally, I do this via the following *tupling function*, defined for all finite sequences $x_1, x_2, \ldots, x_k$ of strings in $\{0,1\}^*$ by:

$$\langle x_1, x_2, \ldots, x_k\rangle =_{\text{def}} \mathit{double}(x_1)01\mathit{double}(x_2)01\ldots01\mathit{double}(x_k),$$

where *double* is defined recursively by $\mathit{double}(\epsilon) = \epsilon$, $\mathit{double}(0x) = 00\mathit{double}(x)$, and $\mathit{double}(1x) = 11\mathit{double}(x)$. For example, $\langle 101, 01, \epsilon, 00\rangle = 11001101001101010000$. What this really does is map the earlier alphabet $\{0,1,\#\}$ back onto strings over $\{0,1\}$, and for most purposes we can use the notations $x\#y$ and $\langle x, y\rangle$ interchangeably. The only reason for the change now is that earlier we said quantifiers would only range over strings in $\{0,1\}^*$. Now the right-hand side of (a) can be replaced by:

$$(\exists z)\left[(\exists y, \vec{c})z = \langle y, \vec{c}\rangle \ \wedge\ \mathit{Acc}(r, x\#y, \vec{c})\right].$$

The point is that now there is a 2HDFA S' which, given input $x\#z$ (or $\langle x, z\rangle$), can decide the statement inside the $[\ldots]$ (since r is fixed). Note that the quantifiers on y and $\vec{c}$ inside the $[\ldots]$ are really "bounded" in their scope by z once z is given by the outside quantifier. Indeed, y and $\vec{c}$ can be read off from any "good" z, and if we introduced notation analogous to *car* and *cdr* in LISP, we could write instead:

$$(\exists z)\left[\mathit{Acc}(r, x\#\mathit{car}(z), \mathit{cdr}(z))\right].$$

But then we'd have to worry about issues such as: what if z is not in the range of the tupling function on two strings? Also, the form with y and $\vec{c}$ is more intuitive to read. Happily, all of these problems can be taken care of by a 2HDFA—I'll leave you to imagine how to modify the one from class for the *Acc* predicate so that it also checks that z has the right form. [1]

This ability to "coalsce" two like quantifiers into one is often taken for granted. What makes a predicate complicated isn't so much the number of variables or quantifiers as the number of *alternations* between "there exists" and "for all" in its statement. Ground level is when the predicate is decidable, and as the above shows, one can then re-write the predicate with one quantifier, indeed one with bounded scope. A decidable predicate is also known as a $\sum_0$-*predicate*, a $\prod_0$-*predicate*, and most preferably, as a Δ_0-*predicate*. The following inductive definition formalizes the alternation of quantifiers.

Definition 3.1. Let $k \geq 1$. A predicate $P(\ldots)$ is a $\sum_k$-*predicate* if $P(\ldots)$ has the form $(\exists z_k Q(\ldots))$ where $Q(\ldots)$ is a $\prod_{k-1}$-predicate. $P(\ldots)$ is a $\prod_k$-*predicate* if $P(\ldots)$ has the form $(\forall z_k Q(\ldots))$ where $Q(\ldots)$ is a $\sum_{k-1}$-predicate.

[1] Put another way, we can cope with the fact that the tupling function is not *onto*; e.g., 001011 is not in the range. The challenge to make a *pairing function* be onto and still be computable by a finite-state machine led to my paper on that subject.

For example, the predicate $(\forall x)(\exists \vec{c})T(i, x, \vec{c})$ is a $\prod_2$-predicate and expresses that the TM M_i is total. The predicate $(\forall x)(\forall \vec{c})\neg T(i, x, \vec{c})$, which expresses that M_i has empty domain, only counts as a $\prod_1$-predicate because the two universal quantifiers can be "lumped together."

Definition 3.1 only allows formal predicates P in which all quantifiers are out in front. Such a predicate is said to be in *prenex normal form*. The advantage of prenex normal form is that it is easy to see which variables are *bound*; *i.e.*, in the scope of a quantifier on that variable, and which are *free*.

There are conventions in writing predicates which are confusing and take getting used to. Variables such as x and $y_1, y_2, \ldots, y_k$ are really *formal symbols*, but they also stand for the strings or numbers used to "fill them in." When we write $P(x, y_1, y_2, \ldots, y_k)$, the *formal variables* are considered to be listed in a certain order inside P already. The '$(x, y_1, y_2, \ldots, y_k)$' is *not* formally part of the predicate—it is *only* a reminder telling the reader all or *some* of the variables in the predicate. One can use this device to instantiate some or all of the arguments; *e.g.*, '$P(0110, y_1, \ldots, y_{k-1}, 101)$.' It is helpful to think of a predicate P as a procedure that returns values of type BOOLEAN, where the free variables of P are those listed as arguments in the procedure header, and the bound variables are those declared locally in the next line of P. Thus the standard convention in writing a predicate P is to list after P exactly those variables which are free in P. For instance, by quantifying on the decidable predicate $Acc(i, x, \vec{c})$ we can define the $\sum_1$-predicate

$$Accs(i, x) := (\exists \vec{c})\ Acc(i, x, \vec{c}).$$

Then the predicate

$$AccAll(i) := (\forall x)\ Accs(i, x)$$

is obtained by universal quantification on the $\sum_1$-predicate $Accs(i, x)$, so by Definition 3.1 it is a $\prod_2$-predicate. Unrolling it all the way down to a decidable base predicate gives,

$$AccAll(i) \leftrightarrow (\forall x)(\exists \vec{c})\ Acc(i, x, c),$$

where '$\leftrightarrow$' stands for "is syntactically equivalent to." Here we can see the "for all..., there exists... [decidable]" form which characterizes $\prod_2$-predicates. On the other hand, it is also conventional to refer to a predicate P without listing any variables at all, since they are all declared in the header and body of P anyway. with all this in mind, the following is really the same as the definition of 'L_R' at the beginning of this section:

Definition 3.2. Let R be a predicate with free variables $x_1, \ldots, x_m$. Then R is said to *represent* the language $L_R := \{w_1 \# \cdots \# w_m : \text{each } w_i \text{ is in } \Sigma^* \text{ and } R(w_1, \ldots, w_m) \text{ holds}\}$.

4. The Arithmetical Hierarchy Theorems

The following is called the "quantifier definition of the arithmetical hierarchy":

Definition 4.1. Let $k \geq 0$. Then a language $L \subseteq \Sigma'^*$ is a $\sum_k$-*language* if there is a $\sum_k$-predicate R such that $L = L_R$. L is a $\prod_k$-*language* if there is a $\prod_k$-predicate S such that $L = L_S$. L is a Δ_k-*language* if it is both a $\sum_k$-language and a $\prod_k$-language.

By Proposition 3.1 it follows that as long as $k \geq 1$, for any $\sum_k$-language L, we can in fact find a $\sum_k$-predicate R such that R represents L *and* the part of R in $[\ldots]$ is decidable by a 2HDFA. To see this, just find the last '$\exists'$ or '$\forall'$ quantifier block in the predicate R (it will be '$\exists'$ if k is even and '$\forall'$ if k is odd), and apply the construction in Proposition 3.1 to what follows. A similar fact

holds for $\prod_k$-languages L and $\prod_k$-predicates S whose "base" is decidable by a 2HDFA. Historically, using numbers in place of strings, it was observed that the part in $[\ldots]$, in addition to being primitive recursive, can also be written as an *arithmetical formula*; *i.e.*, a logical formula involving arithmetic expressions. Examples of arithmetical formulas are $\mathit{Divides}(m,n) := (\exists r)(n = mr)$ and $\mathit{Prime}(n) := n > 1 \ \wedge \ (\forall m)[\mathit{Divides}(m,n) \rightarrow (m = 1 \ \vee \ m = n)]$. Over twenty pages of K. Gödel's famous 1931 paper proving his "First Incompleteness Theorem" were devoted to the details of translating the decidable predicate $\mathit{IsProof}(S, \vec{c}) \equiv$ "$\vec{c}$ is a formal proof of the statement S in Bertrand Russell's formal system called *Principia Mathematica*" into an arithmetical formula π. Then it was observed that Kleene's T-predicate could be "arithmetized" in a similar manner.[2] Thus putting back the quantifier blocks makes the whole $\sum_k$-predicate R into an arithmetical formula that represents L. This is where the term "Arithmetical Hierarchy" came from. The vital link between this and the "oracle" definition we originally gave for it is:

Theorem 4.1. *(The* Arithmetical Hierarchy Theorem *of Stephen Kleene). Let $k \geq 0$, and let $L \subseteq \Sigma'*$ Then*

(a) L is a $\sum_k$-language if and only if $L \in \sum_k^0$.

(b) L is a $\prod_k$-language if and only if $L \in \prod_k^0$.

(c) L is a Δ_k-language if and only if $L \in \Delta_k^0$.

Proof. First let us observe that for any $k \geq 0$, (a) and (b) are equivalent, and both imply (c). Let use write e.g. $\vec{x}$ for tuples $(x_1, \ldots, x_m)$ of formal variables, and use the same notation for strings $\vec{x} \in \Sigma'^*$ regarded as having the form $x_1 \# x_2 \# \cdots \# x_m$ where each x_i belongs to Σ^*. For $(a \implies b)$, let L be a $\prod_k$-language. Then there is a $\prod_k$-predicate R of the form $(\forall \vec{y}_1)(\exists \vec{y}_2) \ldots S(\vec{x}, \vec{y}_1, \ldots, \vec{y}_k)$ with S decidable that represents L. The negation $\neg R$ of R represents the complement $\widetilde{L}$ of L. Bringing the negation "inside" makes $\neg R$ equivalent to $(\exists \vec{y}_1)(\forall \vec{y}_2) \ldots \neg S(\vec{x}, \vec{y}_1, \ldots, \vec{y}_k)$. Since '$\neg S'$ is still decidable predicate, this is a $\sum_k$-predicate that represents $\widetilde{L}$. By (a), $\widetilde{L} \in \sum_k^0$. Hence by the original definition of $\prod_k^0$, $L \in \prod_k^0$. The other half of (b), and the implication $(b \implies a)$, follow by similar arguments. Now for (c), let L be a Δ_k-language. Then L is both a $\sum_k$-language and a $\prod_k$-language. Hence by (a) and (b), $L \in \sum_k^0 \cap \prod_k^0$. By Theorem 2.2 (namely: for all oracle sets A, $\mathrm{RE}^A \cap \text{co-RE}^A = \mathrm{REC}^A$), it follows that $L \in \Delta_k^0$. Conversely, if $L \in \Delta_k^0$ then $L \in \sum_k^0 \cap \prod_k^0$, so by (a) and $(b)L$ is both a $\sum_k$-language and a $\prod_k$-language, so L is a Δ_k-language.

Now we prove (a) by induction on k, also using the equivalence of (a) and (b) to help carry through the induction hypothesis. The base case $k = 0$ is clear: a language L is recursive iff the predicate '$x \in L'$ is decidable. So let $k \geq 1$, and assume (a) and (b) for $k - 1$.

First suppose L is a $\sum_k$-language, and let $R := (\exists \vec{y}_1)(\forall \vec{y}_2) \ldots S(\vec{x}, \vec{y}_1, \ldots, \vec{y}_k)$ be a $\sum_k$-predicate that represents L. Now $\vec{y}_1$ is a sequence of variables $y_{11} y_{12} \ldots y_{1 j_1}$ in an "existential block," but by using the *double* trick and sticking the decoding of strings $y \in ((00 \cup 11)^* 01)^{j_1 - 1}(00 \cup 11)^*$ as part of the decidable predicate S, we can replace $\vec{y}_1$ by a single variable y_1 ranging over strings in Σ^*. Taking this modification as done, now define

$$L' := \{\, \vec{x} \# y : y \in \Sigma^* \ \wedge \ (\forall \vec{y}_2) \ldots S(\vec{x} \# y, \vec{y}_2, \ldots, \vec{y}_k) \,\}.$$

[2] We *could* do it by converting the transition function δ of a 2HDFA into a program in an "equational programming language," but the details are much more laborious than the string-based coding of tuples I've given in lectures. By the way: several more pages of Gödel's paper went into arithmetically defining his so-called *β-function* for coding sequences of numbers by a single number—e.g., $\beta(a,b,c) = 2^a 3^b 5^c$. The tupling function has the same purpose for strings.

The whole reason we cared about having $y \in \Sigma^*$ is so we could identify the 'y' part in '$\vec{x}\#y$.' Then L' is represented by a $\prod_{k-1}$ predicate, so by the induction hypothesis applied to (b), $L' \in \prod^0_{k-1}$. And $L \in \mathrm{RE}^{L'}$, because the program:

y := 0;
if $\vec{x}\#y \in L'$ then *accept*
else increment y and try again.

accepts L with oracle L'. Similarly $L \in \mathrm{RE}^A$ where A is the complement of L', by changing the sentence to "...if $\vec{x}\#y \notin A$ then..."—and since $A \in \sum^0_{k-1}$, $L \in \sum^0_k$ by the definition of $\sum^0_k$ as $\mathrm{RE}(\sum^0_{k-1})$.

Going the other way, suppose $L \in \sum^0_k$. Then there is a language $A \in \sum^0_{k-1}$, $A \subseteq \Sigma^*$, and an oracle TM M such that $L = L(M^A)$. By the induction hypothesis, there is a $\sum_{k-1}$-predicate R of the form $(\exists z_1)(\forall z_2)(\exists z_3)\ldots S(\vec{x}, z_1, \ldots, z_{k-1})$ that represents A. Then $(\forall z_1)(\exists z_2)\ldots \neg S(\vec{x}, z_1, \ldots, z_{k-1})$ represents $\widetilde{A}$. Now define a *big* predicate:

> $B(M, x, \vec{c}, \vec{s}, \vec{t}, \vec{v}) := [\vec{c}$ is an accepting computation of M on input x that has some number m of query IDs in which the oracle answer was '*yes*'—i.e., the next ID listed in $\vec{c}$ has state q_{yes}—and some number n of query IDs in which the oracle is recorded as answering 'no', and $\vec{s}$ equals a list $s_1\#\cdots\#s_m$ of the query strings $s_i \in \Sigma^*$ which were answered 'yes', and $\vec{t}$ similarly lists those strings $t_1, \ldots, t_n$ which were answered 'no' in $\vec{c}]$ *and:*
>
> $(\forall z \in \Sigma^*)$: if $z \in ((00 \cup 11)^*01)^{m+n-1}01(00 \cup 11)^*$, then, decoding z uniquely as $z_{12}, \ldots z_{m2}, z_{11}, \ldots, z_{n1}$, we have:
>
> - for each i, $1 \le i \le m$, $(\exists z_3)\ldots S(s_i, v_i, z_{i2}, z_3, \ldots, z_{k-1})$, and
> - for each j, $1 \le j \le n$, $(\exists z'_2)\ldots \neg S(t_j, z_{j1}, z'_2, \ldots, z'_{k-1})$.

The part of B in $[\ldots]$ is all decidable by examining $\vec{c}$. Hence the leading quantifier in B is the '$(\forall z \in \Sigma^*)$'. Everything which follows can be represented by a $\sum_{k-2}$-predicate. Hence B is equivalent to a $\prod_{k-1}$ predicate. (A more rigorous way to see this is to show that the complement of B is accepted by some TM with an oracle in $\sum^0_{k-2}$ and hence belongs to $\sum^0_{k-1}$, using the induction hypothesis again to get this last sentence.) Finally, observe that for all x,

$$x \in L \iff (\exists \vec{c}, \vec{s}, \vec{t}, \vec{v})B(M, x, \vec{c}, \vec{s}, \vec{t}, \vec{v}).$$

Hence L is represented by a $\sum_k$-predicate, so it is a $\sum_k$-language. The rest of the proof now follows by induction. □

This is sometimes called the *weak hierarchy theorem*, and the *strong hierarchy theorem* is the name for the following: (I use '$\subset$' for *proper* containment.)

Theorem 4.2 (Kleene, after Turing). *For all $k \ge 0$, $\sum^0_k \subset \sum^0_{k+1}$.*

Proof. Define by induction: $AP_0 := \emptyset$, and for each $k \ge 1$, $AP_k := \{code(M)x : M$ with oracle AP_{k-1} accepts $x\}$. Then AP_1 is essentially the same as the language of the Acceptance Problem as defined in class, while AP_2 is the same as the language $AP(B)$ where $B := AP$ in the proof of

Theorem 2.4 above. For each k, $AP_k \in \sum_k^0$. The proof of Theorem 2.4 extends to show that for all k, $AP_{k+1} \notin \mathrm{REC}(AP_k)$. Since $AP_{k+1} \in \sum_{k+1}^0$ while $\sum_k^0 \subseteq \mathrm{REC}(AP_k)$, $\sum_k^0 \subset \sum_{k+1}^0$. In fact, by the theorem that for all oracle sets A, $\mathrm{RE}^A \cap \text{co-RE}^A = \mathrm{REC}^A$, it follows that for all $k \geq 1$, $\sum_k^0 \neq \prod_k^0$. For every $k \geq 1$, the language

$$AP_k \oplus \widetilde{AP}_k := \{\, 0x \mid x \in AP_k \,\} \cup \{\, 1y \mid y \in \widetilde{AP}_k \,\}$$

is recursive in AP_k (by a 1tt reduction, in fact) and so belongs to Δ_{k+1}^0, but belongs to neither $\sum_k^0$ nor $\prod_k^0$. Hence we actually have the stronger conclusion that for all $k \geq 1$, $\sum_k^0 \subset \sum_k^0 \cup \prod_k^0 \subset \Delta_{k+1}^0 \subset \sum_{k+1}^0$. □

5. Tarski's Theorem on the Undefinability of Truth

The "Strong Hierarchy Theorem" has the following intuitive reading: for each $k > 0$, there are some languages A, in particular $A := AP_k$, that can be defined using k quantifiers on a recursive predicate, but are *not* definable with any fewer than k quantifiers. Instead of using the machine-defined languages AP_k, we can use the proof of the "Weak Hierarchy Theorem" to the same effect. A $\sum_k$*-sentence* is a $\sum_k$-predicate which has *no* free variables.

Definition 5.1. For each $k \geq 1$, let $T_k := \{code(Q) : Q \text{ is a } \sum_k \text{ sentence that is } true\}$, where *code* is some reasonable encoding of the language of logic over Σ^*.

Theorem 5.1. *For each $k \geq 1$, and every language $A \in \sum_k^0$, $A \leq_m T_k$, and T_k itself belongs to $\sum_k^0$.*

Proof. By the weak hierarchy theorem, A is representable by a $\sum_k$-predicate of the form $R(x) := (\exists y_1)(\forall y_2) \ldots S(x, y_1, \ldots, y_k)$. For any *string* $w \in \Sigma^*$, let R_w be the sentence $R(w)$; i.e.,

$$R_w := (\exists y_1)(\forall y_2) \ldots S(w, y_1, \ldots, y_k).$$

Clearly there is a total recursive function σ such that for all w, $\sigma(w)$ gives the code of R_w. Then for all $w \in \Sigma^*$, $w \in A \iff R(w)$ is true $\iff R_w \in T_k$. So $A \leq_m T_k$.

To see that T_k itself belongs to $\sum_k^0$, just program a TM M that given as input any encoding of a $\sum_k$-sentence $(\exists y_1)(\forall y_2) \ldots Q(y_1, \ldots, y_k)$, uses T_{k-1} as an oracle to hunt for a string w such that the $\prod_{k-1}$ sentence $(\forall y_2) \ldots Q(w, y_2, \ldots, y_k)$ is true. By induction we have $T_{k-1} \in \sum_{k-1}^0$, so $T_k \in \mathrm{RE}(\sum_{k-1}^0)$, so $T_k \in \sum_k^0$. □

Corollary 5.2 (Tarski's Theorem). *There is no arithmetical definition of arithmetical truth: the language TRUTH of true arithmetical sentences does not belong to the arithmetical hierarchy.*

Proof. Suppose it did. Then for some k, we would have $TRUTH \in \sum_k^0$. But clearly, for any k, $T_k \leq_m TRUTH$. In particular, $T_{k+1} \leq_m TRUTH$. Since $AP_{k+1} \leq_m T_{k+1}$, this would give $AP_{k+1} \leq_m TRUTH$, hence $AP_{k+1} \in \sum_k^0$, but we know this is false. □

Generally, for any language class $\mathcal{C}$, a language B with the properties $B \in \mathcal{C}$ and (for all $A \in \mathcal{C}$, $A \leq_m B$) is said to be *complete* for $\mathcal{C}$. For each $k \geq 0$, both AP_k and T_k are complete for $\mathcal{C}$. If D is another language such that $D \in \mathcal{C}$ and $B \leq_m D$, then D is also complete for $\mathcal{C}$.

Examples: (1) The language FIN $:= \{i : L(M_i) \text{ is finite}\}$ is represented by the predicate

$$(\exists y \in \Sigma^*)(\forall x \in \Sigma^*)(\forall \vec{c} \in \Sigma^*)[x > y \to \neg Acc(i, x, \vec{c})].$$

Since this is a $\sum_2$-predicate, FIN $\in \sum_2^0$. As an exercise (this or something like it will be covered in lecture), show that $T_2 \leq_m$ FIN. Hence FIN is $\sum_2^0$-complete.

(2) The language TOT belongs to $\prod_2^0$, since for any TM M,

$$code(M) \in TOT \iff (\forall x)(\exists \vec{c})\, T(code(M), x, \vec{c}).$$

To show that it is complete for $\prod_2^0$, let A be any language in $\prod_2^0$. Then there is a recursive predicate R such that for all x,

$$x \in A \iff (\forall y)(\exists z)\, R(x, y, z).$$

Now given x, let $f(x)$ be the code of a TM M_x that on any input y executes

$$z := 0;\ \text{WHILE (NOT } R(x, y, z)) \text{ DO } z := z + 1 \text{END-WHILE; ACCEPT.}$$

Then $x \in A \iff M$ is total, and since the code of M_x is straightforwardly computable once x is given, this is a many-one reduction from A to TOT.

Hence *any* definition of FIN or TOT in arithmetic must use at least two unbounded quantifiers, and any two-quantifier definition of FIN must begin with 'there exists...', and for TOT, 'for all...'

END OF HANDOUT.