

Diagonalization

Another example:
 Let TOT be the set of all numbers p such that p is the number of a program that computes a total function $f(x)$ of one variable:
 $TOT = \{z \in \mathbb{N} \mid (\forall x) \Phi(x, z) \downarrow\}$

Since $\Phi(x, z) \downarrow \Leftrightarrow x \in W_z$,
 TOT is simply the set of numbers z such that W_z is the set of all nonnegative integers.

Theorem 6.1: TOT is not recursively enumerable.

November 10, 2009 Theory of Computation
Lecture 16: Computation on Strings I 1

Diagonalization

Proof:
 Suppose that TOT were r.e.
 Since $TOT \neq \emptyset$ (we know that there are total unary functions), by Theorem 4.9 there is a computable function $g(x)$ such that $TOT = \{g(0), g(1), g(2), \dots\}$.
 Let $h(x) = \Phi(x, g(x)) + 1$.
 Since any $g(x)$ is the number of a program that computes a total function, $\Phi(x, g(x))$ is defined for all x , and $h(x)$ is a computable function.
 Let $h(x)$ be computed by a program with number p .
 Then $p \in TOT$, which means that $p = g(i)$ for some i . Then
 $h(i) = \Phi(i, g(i)) + 1$ by definition of h
 $= \Phi(i, p) + 1$ since $p = g(i)$
 $= h(i) + 1$ since h is computed by p . **Contradiction!**

November 10, 2009 Theory of Computation
Lecture 16: Computation on Strings I 2

Diagonalization

$\Phi(0, g(0))$	$\Phi(1, g(0))$	$\Phi(2, g(0))$...
$\Phi(0, g(1))$	$\Phi(1, g(1))$	$\Phi(2, g(1))$...
$\Phi(0, g(2))$	$\Phi(1, g(2))$	$\Phi(2, g(2))$...
...

The elements on the **diagonal** make it impossible for the function h to be computed by any of the programs $g(x)$.

November 10, 2009 Theory of Computation
Lecture 16: Computation on Strings I 3

Diagonalization

Theorem 6.1 gives a reason why we base our studies of computability on partial rather than total functions:

By Church's Thesis, Theorem 6.1 shows that there is no algorithm to determine whether an \mathcal{L} program computes a total function.

November 10, 2009 Theory of Computation
Lecture 16: Computation on Strings I 4

Reducibility

Another important technique for determining nonrecursive sets is the **reducibility** method.

Once some set (such as the set K) has been shown to be nonrecursive, we can use that set to give other examples of nonrecursive sets.

November 10, 2009 Theory of Computation
Lecture 16: Computation on Strings I 5

Reducibility

Definition: Let A, B be sets. Then A is **many-one reducible** to B , written $A \leq_m B$, if there is a computable function f such that
 $A = \{x \in \mathbb{N} \mid f(x) \in B\}$.
 In other words, $x \in A$ if and only if $f(x) \in B$.
 "Many-one" means that f does not have to be one-one.

If $A \leq_m B$, then testing membership in A is "no harder than" testing membership in B .
 To test whether $x \in A$ we can compute $f(x)$ and then test whether $f(x) \in B$.

November 10, 2009 Theory of Computation
Lecture 16: Computation on Strings I 6

Reducibility

Theorem 6.2: Suppose $A \leq_m B$.

1. If B is recursive, then A is recursive.
2. If B is r.e., then A is r.e.

Proof: Let $A = \{x \in \mathbb{N} \mid f(x) \in B\}$, where f is computable, and let $P_B(x)$ be the characteristic function of B.

Then $A = \{x \in \mathbb{N} \mid P_B(f(x))\}$.

If B is **recursive**, then $P_B(f(x))$, the characteristic function of A, is computable, so A is recursive.

If B is **r.e.**, then $B = \{x \in \mathbb{N} \mid g(x) \downarrow\}$ for some partially computable function g.

Then $A = \{x \in \mathbb{N} \mid g(f(x)) \downarrow\}$, and since $g(f(x))$ is partially computable, A is r.e.

November 10, 2009

Theory of Computation
Lecture 16: Computation on Strings I

7

Reducibility

We will often use Theorem 6.2 in the following form:
If A is not recursive (r.e.), then B is not recursive (r.e.).

Example:

$$K_0 = \{z \in \mathbb{N} \mid \Phi_{f(z)}(f(z)) \downarrow\} = \{\langle x, y \rangle \mid \Phi_y(x) \downarrow\}$$

Obviously, K_0 is r.e. However, we can show that K_0 is not recursive by reducing K to K_0 .

$$K = \{n \in \mathbb{N} \mid n \in W_n\}.$$

Now $x \in K$ if and only if $\langle x, x \rangle \in K_0$, and the function $f(x) = \langle x, x \rangle$ is computable.

Therefore, $K \leq_m K_0$, and K_0 is not recursive.

November 10, 2009

Theory of Computation
Lecture 16: Computation on Strings I

8

Numerical Representation of Strings

So far, our programs in the language \mathcal{L} have been using **natural numbers** as their inputs and output.

For many applications, however, we would prefer to perform computations on **strings** on some alphabet instead.

You remember that we introduced a **numbering of \mathcal{L} programs** so that \mathcal{L} programs could be used as input and output of another (or the same) \mathcal{L} program.

With regard to strings, we will use the same approach:

We will **associate numbers with strings** on A in a one-one manner.

November 10, 2009

Theory of Computation
Lecture 16: Computation on Strings I

9

Numerical Representation of Strings

We will use a system that is very similar to our everyday one-one mapping of **natural numbers** to **strings of digits**.

There we have a set D of digits, and we define an order s_0, \dots, s_9 on these digits:

$$D = \{s_0, \dots, s_9\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

There are $n = 10$ elements in our set of digits.

Then any string w of digits can be written as

$$w = s_{i_k} s_{i_{k-1}} \dots s_{i_1} s_{i_0},$$

where $0 \leq i_m \leq n - 1$ and $k = |w| - 1$.

November 10, 2009

Theory of Computation
Lecture 16: Computation on Strings I

10

Numerical Representation of Strings

For example, if we have the string $w = 372$, then

$$k = 2, i_2 = 3, i_1 = 7, i_0 = 2.$$

To find the number associated with this string, we use exactly the following formula:

$$x = i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \dots + i_1 \cdot n^1 + i_0 \cdot n^0$$

$$x = 3 \cdot 10^2 + 7 \cdot 10^1 + 2 = 372.$$

If $w = 372$ is an **octal** representation of an integer, then we would have $n = 8$ and therefore:

$$x = 3 \cdot 8^2 + 7 \cdot 8^1 + 2 = 192 + 56 + 2 = 250$$

November 10, 2009

Theory of Computation
Lecture 16: Computation on Strings I

11

Numerical Representation of Strings

Now let us develop such a method for strings on an alphabet A.

Remember that the set of all strings on an alphabet A, including the empty string, is called A^* .

Again, let us assume that there is a particular **order** of symbols in A.

We write $A = \{s_1, \dots, s_n\}$ and define that the sequence s_1, \dots, s_n corresponds to this order of symbols.

Then any string w on A can be written as

$$w = s_{i_k} s_{i_{k-1}} \dots s_{i_1} s_{i_0}, \text{ where } 1 \leq i_m \leq n \text{ and } k = |w| - 1.$$

The **empty string** is indicated by $w = 0$.

November 10, 2009

Theory of Computation
Lecture 16: Computation on Strings I

12

Numerical Representation of Strings

Then we use exactly the same formula as before to associate w with an integer x :

$$x = i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \dots + i_1 \cdot n^1 + i_0 \cdot n^0.$$

With $w = 0$ we associate the number $x = 0$.

For example, consider the alphabet $A = \{a, b, c\}$ and the string $w = caba$.

$$\text{Then } x = 3 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 = 81 + 9 + 6 + 1 = 97.$$

Now why is this representation unique?

We can prove this by showing how to retrieve the subscripts i_0, i_1, \dots, i_k from x for any $x > 0$.