Assignment 11

Karan Sikka
ksikka@andrew.cmu.edu
E
May 5, 2012

The ZFCs Are Alive

---

a) We assumed that there is a proof of T in ZFC. Since ZFC is a subset of $ZFC + \neg S$, we know there is a proof of T in Z'.

$$T \implies \text{“If there is a proof of } S \text{ in } ZFC \text{ then } S \text{ is true”}$$

$\implies$

$$\text{“There isn't a proof of } S \text{ in } ZFC \text{ or } S \text{ is true”}$$

Recall that $\neg S$ is an axiom in ZFC. $\implies$

$$\text{“There isn't a proof of } S \text{ in } ZFC\text{”}$$

$\implies$

$$\neg A$$

We have shown a Z' proof of $\neg A$ □

b) “ZFC is consistent” $\iff$ “$\forall S \in ZFC$, at least one of S and $\neg S$ is unprovable”

We can generalize the above proof for any statement S, since the above proof is not dependant on it's definition. Therefore, let S be an arbitrary statement, and by the logic in part A, we can prove that there does not exist a proof of S. See the following logic (A uses the generalized definition of S).
$\neg A = \neg$ “there is a proof of S in ZFC”
$\neg A =$ “S is unprovable in ZFC”

Therefore, by the definition of consistency, It is possible to show that ZFC is consistent. □

c) don't know

d) don't know

When You Know L's In NP

Want to show:
$L \in NP \implies L^* \in NP$

Proof:
If $L \in NP$ then there exists a verifier for membership in L which runs in polynomial time. Call this verifier f, a function which takes a string and returns true if the string is in the Language.

The verifier for L^* takes a string input and a certificate, which is a list of integers. The verifier will return true if the string input is in the Kleene star of L. The certificate for $w \in L^*$ is a list of integers such that the ith element of the list is the length of w_i in w (see definition of Kleene star in problem statement).

The verifier will follow this procedure:

1. if the certificate is the empty list, check if the string is empty. if it is, return true. if not, return false.
2. check that the sum of the elements in the certificate is equal to the length of the input string. if this does not hold, return false.
3. Call the first element in the certificate x. If the verifier for L returns false when fed the first x characters of the input string, return false.
4. Return the result of the verifier called on the rest of the input string, with the certificate as the list without the first element.

This verifier is correct: The empty string is accepted. The invariant that the sum of the lengths of the concatted strings (as recorded in the certificate) equals the entire string holds. The verifier returns true by the recursive definition of membership in Kleene star of L: $w \in L^* \iff w = w_p v$ such that $w_p \in L, v \in L^*$

This verifier runs in poly-time: Let k be the length of the certificate and let s be the length of the input string. The algorithm calls the verifier k times. The verifier runs in polynomial time, so we can say it is upper bounded by s^p for some $p, p \neq 0$. Then the algorithm is $O(ks^p)$ which is clearly polynomial time.

Raindrops On Roses And Subsets Of Numbers

Part 1: You can solve the subset sum problem using a partition oracle in poly-time:

The algorithm:
Given L, a list of integers, and k, an integer.

1. Calculate the sum of the numbers, call it SUM
2. Add $(2k -$ SUM $)$ to L and call it L'

3. Return the result of PARTITION L'

Correctness:
PARTITION L returns true iff there is a 2-partition of L such that the sum of one partition is equal to the sum of the other. Say those partitions are S_1 and S_1 If PARTITION L returns true, then:

$$\begin{aligned} &\text{SUM } S_1 = \text{ SUM } S_2 \quad \wedge \quad \text{SUM } S_1 + \text{ SUM } S_2 = \text{ SUM } L \\ \implies &\text{SUM } S_1 = \text{ SUM } S_2 \quad \wedge \quad \text{SUM } S_1 + \text{ SUM } S_1 = \text{ SUM } L \\ \implies &\text{SUM } S_1 = \text{ SUM } S_2 \quad \wedge \quad 2 * \text{ SUM } S_1 = \text{ SUM } L \\ \implies &\text{SUM } S_1 = \text{ SUM } S_2 = \frac{\text{SUM } L}{2} \end{aligned}$$

Consider $L \cup \{2k - \text{ SUM } L\}$. We will call this set L' for convenience. We know that $\text{SUM } L' = \text{ SUM } L + 2k - \text{ SUM } L = 2k$. Therefore, if L' is accepted in PARTITION, there are two sets in L' which partition L' and sum to k. The element $2k - \text{ SUM } L$ can only be in one of these sets, so the other set is a subset of L which sums to k. Therefore, this reduction is a mapping reduction.

It's poly-time:
Calculating the sum of numbers is at most poly-time on the length of the inputs. Adding an element to the list is at most poly-time. Therefore the overall reduction is at most poly-time.

Part 2: You can solve the partition problem using a subset sum oracle in poly-time:

1. Let L be a list of integers.
2. Calculate the sum of the numbers, call it SUM.
3. If SUM is odd, return false.
4. Else, return the result of subset sum (L,SUM/2).

Correctness:
A list of integers is accepted in PARTITION if and only if there exists a 2-partitioning of the list such that each partition sums to (SUM L)/2. Note that SUM L must be an even number for (SUM L)/2 to be even, and for there to exist such a partitioning. If $(L, \frac{\text{SUM } L}{2})$ is accepted in SUBSETSUM then there exists a subset of L, called S_1 which sums to $\frac{\text{SUM } L}{2}$. In addition, we can conclude that $\exists S_2$ such that $S_2 = L \setminus S_1$. Therefore an integer list is accepted in PARTITION if and only if $\frac{\text{SUM } L}{2}$ is even and $(L, \frac{\text{SUM } L}{2})$ is accepted in SUBSETSUM. Therefore this is a mapping reduction.

It's poly-time: Calculating the sum of numbers is at most poly-time on the length of the list. Checking whether it is even or odd is at most poly-time by doing the mod of SUM. Therefore the overall reduction is at most poly-time.

So Long, Farewell, auf Widersehen, Goodbye!

Aur revoir!

a) **Want to show:** `YO_IT_WENT_OFF` $\in$ **NP**

The verifier V takes $\{\langle m, n\rangle\}$, the encoding of m and n in binary, and the certificate d which is the binary representation of an integer, which represents the divisor of n.

What V does is it first checks if d is within it's bounds as specified ($1 < d < m$). It will return false if this condition is violated. Then it checks if d is a divisor of n by computing d times the result of integer division of n and d and checking whether this is equal to n itself. It it is, return true. Else return false.

Multiplication, integer division, and checking equality are polynomial in number of bits. Therefore the computation described above will run in polynomial time. □

b) **Want to show:** `YO_IT_WENT_OFF` $\in$ **P** implies that there exists a polynomial time algorithm to find the prime factorization of an integer.

We assume that `YO_IT_WENT_OFF` $\in$ **P**, which implies that there exists a polynomial time oracle for `YO_IT_WENT_OFF`.

`YO_IT_WENT_OFF` takes in $\langle m, n\rangle$ as an input, but we can choose to put in m, n such that $m = n$. Then, `YO_IT_WENT_OFF` returns true iff there is a diviser between 1 and n. This is equivalent to an oracle which tells if a number is composite.

The Algorithm (for an input n):

(a) If `YO_IT_WENT_OFF`(n) is false, then n is prime, so output n and halt. Else, `YO_IT_WENT_OFF`(n) is true, so n is composite.

(b) For all integers d such that $1 < d < n$, check whether d is a divisor of n ($n \mod d$ `==` 0).

(c) Identify the smallest d which satisfies this condition and call it p. By a lemma below, this is necessarily a prime factor of n.

(d) Output $\{p, \langle$ result of this algorithm on $n\rangle\}$

Lemma: The smallest divisor of a composite number is a prime factor of that number.
Proof: AFSOC that the smallest divisor d of a composite number m is not prime. Then there is a prime factor of d which is less than d. But this prime factor is also a factor of m and is smaller than d. Contradiction. □

When the algorithm halts, the result is the prime factorization of the input. The recursive call in (4) take a prime factor out of the prime factorization of a composite input until there is only 1 prime in the prime factorization (the halting condition).

The smallest possible prime factor is 2. Therefore, each recursive call of the algorithm divides n by at least 2. Thus, it reduces the number of bits of the input by at least 1. Therefore the algorithm will call itself at most $log(n)$ times, as the size of the input decreases linearly in the number of bits.

Step 1 is poly-time on n, step 2 is linear on n, step 3 is linear, step 4 is the recursion. The overall algorithm runs in log times a poly, which is still polynomial runtime. □