

# CS303 (Spring 2008)- Solutions to Assignment 11

## Problem 1

- (a) It means that for every program  $P$ : if  $P \in X$ , and  $P \equiv P'$  (that is,  $P$  and  $P'$  produce the same output for *every* input), then  $P' \in X$  also.
- (b) Intuitively, this means that  $X$  as a set characterizes a property of *the function that programs compute*, rather than the syntax of the program or the way in which the computation happens.
- (c) It means that  $f(x) = x$ , i.e., the function  $f$  keeps  $x$  where it was.
- (d) In this context, it means that  $P$  and  $f(P)$  (we now use  $P$  instead of  $x$  for inputs) compute *the same function* (while they need not be exactly the same program). That is,  $f(P) \equiv P$ .

## Problem 2

- (a) We can use Rice's Theorem. To apply it, we need to show that neither USC nor its complement are empty, and that USC respects functions. Then, we immediately get that USC is undecidable. The set is not empty because for instance the program `{ return 'USC'; }` is in it. The complement is not empty because the program `{ return 'UCLA'; }` is in the complement. It respects functions because if  $P \in \text{USC}$ , then  $P$  outputs 'USC' for all even numbers. Now, if  $P \equiv P'$ , then  $P'$  produces the same output for each input  $x$  as  $P$ . In particular,  $P'$  also outputs 'USC' for each even number  $x$ . Thus,  $P' \in \text{USC}$  as well.  
(We could also instead reduce from HALT, but that would be more work.)
- (b) We can apply exactly the same reasoning here as for the previous example. MST is not empty because any reasonable implementation of Kruskal's Algorithm from class would be in the set. On the other hand, the program `{ return 'USC'; }` is not in MST. So it remains to show that MST respects functions. If  $P \in \text{MST}$ , then for each input  $G$ ,  $P$  returns the MST of  $G$ . If  $P \equiv P'$ , then  $P'$  returns the same result for each  $G$ , i.e., it also computes the MST of  $G$ . Thus,  $P' \in \text{MST}$  as well. Thus, MST respects functions, and must be undecidable.
- (c) For this problem, we cannot use Rice's Theorem, because  $\text{HALT}^+$  does not respect functions. For instance, the programs `{ return 0; }` and `{ y = x + y + z; return 0; }` compute the same function, but the first one is not in  $\text{HALT}^+$ , while the second one is. Instead, we'll have to do a reduction from HALT from scratch.

The reduction function  $f$ , when given a program  $P$ , produces the following new program  $P'$  (which gets an input  $x$ ).

```
{
  y = x + y + z;
  Execute P(P);
  return 0;
}
```

If  $P \in \text{HALT}$ , i.e.,  $P(P) \downarrow$ , then the resulting program  $P'$  will terminate on each input, in particular on its own source code  $P'$ . Since it also contains the statement "`y = x+y+z`",  $P' \in \text{HALT}^+$ . On the other hand, if  $P \notin \text{HALT}$ , i.e.,  $P(P) \uparrow$ , then  $P'$  does not terminate on any input (it always hangs on the second line), in particular not on input  $P'$ . Thus, while it does contain the statement "`y=x+y+z`", it is still not in  $\text{HALT}^+$ , as desired. This proves that the reduction is correct. The function  $f$  is also total and recursive, since it only performs some simple syntactic code rewriting.

### Problem 3

Intuitively, this shows that for every two sets  $X, Y$ , there is another set  $Z$  that is “at least as difficult to decide” as both of  $X, Y$ . The way we do this is (for instance) by encoding  $X$  in the even numbers, and  $Z$  in the odd numbers. Let’s assume (as we often do), that both  $X, Y$  are sets of numbers. Formally, we define  $Z$  as the following sets of numbers: for each  $x \in X$ ,  $Z$  contains the number  $2x$ , and for each  $y \in Y$  the number  $2y + 1$ . On the other hand, if  $x \notin X$ , then  $Z$  does not contain the number  $2x$ , and similarly, if  $y \notin Y$ ,  $Z$  does not contain  $2y + 1$ .

To reduce  $X \leq_m Z$ , we use the mapping  $f : x \mapsto 2x$ . That is clearly total and recursive (multiply the input by 2). If  $x \in X$ , then by definition,  $f(x) = 2x \in Z$ , and if  $x \notin X$ , by definition  $f(x) \notin Z$ . To reduce  $Y \leq_m Z$ , we use  $g : y \mapsto 2y + 1$ , which is also total and recursive. If  $y \in Y$ , then by definition of  $Z$ , we have  $g(y) = 2y + 1 \in Z$ , while if  $y \notin Y$ , then  $g(y) \notin Z$ .

### Problem 4

For both of these problems, we will use the Recursion Theorem, much like the example we saw in class.

- (a) We define the following program rewrite function  $f$ . Given an input program  $P$ , it produces the following program  $P' = f(P)$  (which itself gets an input  $k$ ).

```
{
  for (i = 1; i <= k; i ++ )
    print P;
}
```

$f$  is clearly total and recursive, since it just rewrites the code of  $P$  a little. By the recursion theorem, it therefore has a fixpoint  $P$ , i.e., a program  $P$  such that  $P \equiv f(P)$ . That means that on each input  $k$ ,  $P$  and  $f(P)$  output the same result. But we know what  $f(P)$  outputs on input  $k$ , namely  $k$  times the source code of  $P$ . Therefore, on input  $k$ ,  $P$  itself must also output  $k$  times the source code of  $P$ , i.e.,  $k$  times its own source code.

- (b) We define the following program rewrite function  $f$ : given an input (a program  $P$ ), it outputs the following program  $P' = f(P)$ :

```
{
  print "{ print P; }";
}
```

That is,  $f$  outputs a program  $f(P)$  which outputs a program which prints the program  $P$  (notice that  $f$  does not simply output a program  $f(P)$  which prints  $P$  — that’s what we did in class). While this is a little convoluted to think about,  $f$  is clearly not difficult to write.  $f$  is total and recursive, so it has a fixpoint  $P$ , so that  $P \equiv f(P)$ . Now we know that  $f(P)$  prints out the program `{ print P; }`. Let’s call that program  $Q$ . And because  $P$  and  $f(P)$  compute the same function,  $P$  must also output the program  $Q$ . On the other hand, by definition,  $Q$  is the program `{ print P; }`, so it prints the program  $P$ . Thus, we have two programs  $P, Q$  such that  $P$  prints the source code of  $Q$ , and  $Q$  prints the source code of  $P$ . And that’s what we wanted.