HW #4

Sam Quinn

CS321

02/2/2016

1) To show that we must show that the cardinality of is equivalent to to the cardinality of . To show that is equivalent to we show that there is a one-to-one and onto relationship to the set of all naturals. Natural numbers are the numbers ranging from 0 to infinity.

Pair( i, j ) {

return i x j = z

}

The program described above takes two natural numbers, multiplies them and returns the output z. With natural numbers any natural number that is natural multiplied to any other natural will be in the set . Because the values can never be less than zero or more than infinity z has a one-to-one relationship to the set of all natural numbers. z is also onto because every number in the set of all naturals is represented.

2) Show that each of the following sets have a recognizer.

1. The Prime Numbers
   1. Each prime number can be defined with either Yes (is prime) and No (is not prime). You could take all of the factors of the number up to the square root and check if it is truly prime or not. Because you could compute this problem checking a set of primes has a **recognizer.**
2. The fibonacci Numbers
   1. If we were to check if any number given was a fibonacci number or not there are definite sets that would either give a yes (is a fibonacci number) or No (not a fibonacci number). To find out if a number is a fibonacci number you could start from 0 and build a set of all fibonacci numbers which you would be able to determine if the number is a part of or not.
3. Syntactically correct Pascal programs
   1. Assuming that Pascal programs will be passed into a compiler it must check that the code is correct. The compiler will output either a Yes (compiled syntactically correct) or No (did not compile, not syntactically correct). Because there are not infinitely long programs the compiler would eventually either get to the end and return Yes or halt at the first syntax error and return No.
4. Tautologies in the Propositional Calculus
   1. Assuming that the calculus that is being rewritten in different words is inherently solvable you would be able to compare the results. The results would either return True (same output) or False (not the same output).

3) Show that the following sets have acceptors.

1. Perfect numbers
   1. Perfect numbers have acceptors since they will either return True (is perfect) or will return False (not perfect). For this particular problem the program should never “fail to halt” since it is fully computable to sum all of the divisors and check if the sum is 2n.
2. “elementary” functions whose integrals are “elementary” functions
   1. An acceptor could return True for every integral that is an elementary function. The acceptor would fail to halt if the function did not return an elementary function since you can indefinitely take an integral of something.
3. Programs which print all the works of Shakespeare
   1. The acceptor for this program would return True as soon as all of the works created by Shakespeare have been printed. Since there is a finite set of Shakespeare's work the acceptor would not fail to halt or ever halt and return False.
4. Theorems of a formal system
   1. While each formal system has a set of theorems that the system is built upon, each theorem in the set has sub-theorems of their own. However all the theories that are used in math is a countably infinite set which make an acceptor which allow the acceptor to return True.
5. Composite numbers
   1. Composite numbers has the same cardinality as the set of all natural numbers. Because you would be able to determine if a number is composite or not by determining if the number has more than one divisor that is not prime. This problem would return True eventually thus having an acceptor.

4) Show that the following sets have rejectors.

1. Programs which only output prime numbers
   1. A rejector for this program would check if the returned number is actually prime or not. If the number is prime the rejector would fail to halt, but if the number returned is not prime the rejector would reject the returned number and say No.
2. Programs which never Halt on any input
   1. If a program never halts it is indeterminate whether the final result would be True or False. But since the program is never halting the rejector would also never halt resulting in a never halting program.
3. Non-perfect numbers
   1. A rejector would be able to sum up all of the divisors of a number and if the sum does not equal 2n then that number would continue. If a number would be a perfect number the rejector would halt and say No.
4. Diophantine equations which have NO solution
   1. Because the Diophantine equations do not have a solution it will have the rejector stuck in the does not halt state and the rejector would never return No.

5) Show that the set of Fibonacci numbers is Primitive Recursive.

Primitive recursive functions are functions that take natural numbers in and do a modification and then pass that value back to itself. Every number in the Fibonacci sequence is apart of the natural numbers, furthermore every Fibonacci number can be described with . The Fibonacci sequence cannot be computed by a bounded loop it must use the recursive method. The Fibonacci algorithm works in the natural and does a simple modification to each input, and cannot be implemented with bounds, you can say that the fibonacci sequence is a primitive recursive function.

6) Use the **diagonal** argument to show that there is a SET which is not Primitive Recursive.

Primitive recursive functions are recursive functions that have recognizers. Because these programs have recognizers every primitive recursive function must halt with either a Yes or a No. To show that there is a set of recursive functions that are not primitive we will use a diagonal argument.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | t0 | t1 | t2 | ... |
| P0 | p0(0) | p0(1) | p0(2) | p0(n) |
| p1 | p1(0) | p1(1) | p1(2) | p1(n) |
| p2 | p2(0) | p2(1) | p2(2) | p2(n) |
| ... | pn(0) | pn(1) | pn(2) | pn(n) |

The table above describes a the relationship of recursive functions to the inputs. The rows are functions and the columns are the inputs. If we create a recognizer p\* that will take in functions p and inputs t and will definitely return either a Yes or definitely return a No, while this function will be a recognizer just like the p’s in the table there is no way for us to ever tell that for every natural number t that the recursive function p will return a Yes or No. Therefore we can say that p\* is false in both conditions, which says that there are recursive functions that will not halt, also known as recursive non-primitive recursive.