Homework 4

Michael Anderson

April 27, 2011

CS517

Prof. Cull

1

I decided to write my own Turing Machine simulator, since it seemed simple enough. Code for the simulator and the example Turing Machines are attached. Each line of the sample TMs (excepting comments and whitespace of course) is a tuple from the machine's transition function in the following order:

$$Q_{cur} \quad \Gamma_{cur} \quad \rightarrow \quad Q_{next} \quad \Gamma_{next} \quad \{R, S, L\}_{next}$$

Following are some sample results from my simulator's textual interface. The star in the first line indicates the position of the tape head.

```
              *
_ X X X X X X _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Current state: HALT
Number of steps executed: 12
```

This very simple Turing Machine is a unary adder. I initialized the tape to **XXX+XXX** in this example.

```
          *
_ 1 0 1 1 0 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Current state: HALT
Number of steps executed: 235
```

A binary adder given the input **1010+1100**.

```
              *
_ _ _ _ _ N O _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Current state: HALT
Number of steps executed: 66
```

A palidrome recognizer given the input **abababbaba**.

```
                                *
_ 1 0 0 0 , 1 1 0 1 | _ 1 0 0 X + 1 1 0 X 1 _ _ _ _ _ _ _ _ _ _ _ _ _ _

Current state: 14
Number of steps executed: 1754

[R]un to end, or [Enter] to step:
```

A base 2 Fibonacci generator with no input given, because it is programmed to initialize the tape with **1,1**. This one does not halt, but we can see it is currently in the process of adding 8 and 13 (both visible on the left).

2

3

One way to do this is to compress the input string into a string that uses a larger internal working alphabet. As an example, suppose that $K = 4$ and $\Sigma = \{0, 1\}$. Now let $\Gamma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$, to allow conversion of binary input strings into a hexadecimal internal representation. Now instead of taking four steps to operate on each group of four bits, take one step to operate on each hex digit. For arbitrary values of K and $|\Sigma|$, make $|\Gamma| = K^{|\Sigma|}$, so that K-strings of symbols in Σ can be represented as a single symbol in Γ.

Modern computer architecture operates in this fashion. Even though we normally think of digital data as a series of bits, modern CPUs perform computations on 32 or 64 bits in parallel, allowing a 32x or 64x speedup for large operands.

In the computer architecture world, this parallelism is accomplished by simply having enough hardware to process all of the bits at once. In the Turing Machine world, for each symbol in Γ create a set of rules that is analogous to the all of the computation that would be performed on its corresponding K-string of Σ.

4

The number of strings over Σ^* is a countable infinity. This can be demonstrated by simply listing them in some "alphabetical" order, and numbering them by this order. For example if $\Sigma = \{a, b\}$, we could have $0 \leftrightarrow \epsilon, 1 \leftrightarrow a, 2 \leftrightarrow b, 3 \leftrightarrow aa, 4 \leftrightarrow ab, 5 \leftrightarrow ba, \ldots$

The number of sets over Σ^*, however, is an uncountable infinity. Since each such set is a subset of Σ^*, there are 2^{Σ^*} such subsets. By Cantor's Theorem,

the powerset of a countably infinite set is an uncountably infinite set.

As was demonstrated in class, there are only a countably infinite number of programs, and therefore a countably infinite number of possible Turing Machines. Every stored program is stored as a sequence of bits, and any sequence of bits can be interpreted as some natural number (with the proviso of prepending a 1 to resolve the ambiguity of leading zeroes).

Since the number of sets over Σ^* is an uncountable infinity, and the number of Turing Machines is a countable infinity, by the diagonalization argument there exist sets over Σ^* which cannot be accepted by a Turing Machine.

5

All algebraic numbers are computable. Numerical algorithms such as bisection and Newton's method for root-finding compute the value of the root of any polynomial with rational coefficients to any arbitrarily high degree of accuracy.

Not all computable numbers are algebraic. Non-algebraic numbers such as π and e (we can say $\pi/4$ or $e/3$ to get them into the interval (0,1)) can be represented as convergent series with computable terms, and computed to any arbitrarily high degree of accuracy.

Successively applying pairing functions gives us that rational numbers are countable, and polynomials with rational coefficients are countable. Furthermore, since each polynomial has only a finite number of real roots, the real roots of polynomials with rational coefficients are also countable.

From problem (4) there are only a countable number of Turing Machines, and it is given that every computable number has a corresponding Turing Machine that generates it to any degree of accuracy, so the computable numbers are also countable.

Since there are a countably infinite number of both algebraic and computable numbers, these sets are the same size.

6

S is in Recursive iff it has a recognizer, so show that a recognizer for S can be used to make a strong generator $GEN_S(n)$ such that
$\forall n\ GEN_S(n) < GEN_S(n+1)$, and vice versa.

From the notes, given a recognizer for S we can make a monotically increasing strong generator by identifying members of S with the recognizer and not outputting any previously outputted member of S:

$$GEN_S(n) = \begin{cases} n & \text{if } Rec_S = \text{YES and } n \notin \{Gen_S(0), \ldots, Gen_S(n-1)\} \\ GEN_S(n+1) & \text{otherwise} \end{cases}$$

Now given a monotonically increasing strong generator $GEN_S(n)$ for S, we can build a recognizer $Rec_S(n)$ for S by feeding the natural numbers into GEN_S until we generate a value that is either larger than n or equal to n:

```
RecS(n)
    FOR EACH i in {0,1,2,3,...}
        IF GenS(i) = n
            RETURN(YES)
        IF GenS(i) > n
            RETURN(NO)
```