Some Notes on Problems and Complexity

Paul Cull

Department of Computer Science
Oregon State University

April 15, 2011

Chapter 1

Generators

1.1 Grammars

We have described computable sets in terms of *acceptors*, *rejectors*, and *recognizers*. Each of these devices takes a potential element of the set as input and returns an answer or doesn't halt. We want to turn this around and describe devices which take a natural number as input and return an element of the set. We will call such devices **generators** because they generate the set.

Our prototypic example is the Fibonacci numbers. We already know how to write several different programs which compute FIB(n), i.e. when given n as an input these programs each produce the n^{th} Fibonacci number.

This example, suggests the definition

Definition 1.1.1. A **generator** for the set, SET, is a (total) computable function from $\mathbb{N}$ **onto** SET.

This means that for each natural number, the program outputs an element of SET, and for each s in SET, there is an n so that the generator produces s.

This definition requires that the generator always produces an output. We can relax this requirement to get a *partial generator*.

Definition 1.1.2. A **partial generator** for the set, SET, is a partial computable function from $\mathbb{N}$ **onto** SET.

Here, for each natural number, the partial generator either outputs an element of SET or fails to halt, and for each s in SET, there is an n so that the partial generator produces s.

Notice that in our Fibonacci example, the generator is not only a (total) generator, but also has the property that each Fibonacci number is produced exactly one. (Ignore the quibble that FIB(1) = FIB(2) = 1.) The definition permits a generator to repeatedly produce the same element. For example, some element could be produced for an infinite sequence of inputs. So, we want a stronger definition of generators.

Definition 1.1.3. A **strong generator** for the set, SET, is a (total) computable function from $\mathbb{N}$ **one-to-one** and **onto** SET.

We also want to capture the idea that some generators produce results *quickly*. A first attempt at defining quick generators is:

Definition 1.1.4. A **strong primitive recursive generator** for the set, SET, is a primitive recursive function from $\mathbb{N}$ **one-to-one** and **onto** SET.

We may also want to allow primitive recursive generators to produce some elements for several different inputs, and so we define:

Definition 1.1.5. A **(weak) primitive recursive generator** for the set, SET, is a primitive recursive function from $\mathbb{N}$ **onto** SET.

But some primitive recursive functions *can* take a long time to compute, and we usually mean Polynomial time when we say *quickly*, so we define:

Definition 1.1.6. A **(weak) polynomial time generator** for the set, SET, is a polynomial time function from $\mathbb{N}$ **onto** SET.

Definition 1.1.7. A **strong polynomial time generator** for the set, SET, is a polynomial time function from $\mathbb{N}$ **one-to-one** and **onto** SET.

Since the inputs to these functions are natural numbers, and we reasonably assume that the natural number n is encoded in binary (or some other base), then n is represented using about $\log n$ bits. So, our polynomial time requirement in the above definitions seems to require run-time $O(\,(\log n)^K\,)$, for some K. But, this ignores the output. For a generator, GEN(n), polynomial time is $O(\,(\log(\,\mathrm{GEN}(n)) + \log n)^K\,)$, for some K, i.e. polynomial time in both the size of the input and the size of the output. For the Fibonacci numbers, FIB(n) is $\Theta(\,\lambda_0^n\,)$ and $\log(\,\mathrm{FIB}(n)\,) = \Theta(\,n\,)$. Since we know programs which generate the Fibonacci numbers using $O(\,n^2\,)$ time, we would say that these programs are polynomial time generators, and since we can modify these programs so that they only produce each of the Fibonacci numbers once, we would call these modified programs strong polynomial time generators.

Ex 1.1. Consider the following simple generator:

Input n
ADD 1
Truncate the leading 1
Append a trailing 0
Output

Show that this generates the EVEN numbers, i.e. the binary strings which end in 0.
How big is the output?
Calculate the running time of this generator as a function of n. What type of generator is this?

1.2 TOT is HARD

Theorem 1. *Let* **TOT** $= \{M \mid M$ *halts on each and every input* $\}$.
TOT *is not in* **RE** $\cup$ **coRE**.

If **TOT** were in **RE**, then **TOT** would have an acceptor. If **TOT** had an acceptor, we could use it to enumerate the recognizers. But, by a diagonal argument, the recognizers cannot be enumerated.

Could **TOT** be in **coRE**?
Even if we could guess, we'd have to guess an input t on which **M** doesn't halt, and then show that **M** fails to halt when given t.
If **TOT** were in **coRE** then we could effectively list the programs which **FAIL** to halt on at least some input. Now consider the machines $\mathbf{M}_t$ which mimic **M** given t, so that $\mathbf{M}_t$ **FAILS** to halt on all inputs if **M** doesn't halt when given t. Since we have assumed an enumeration, we can look at this enumeration and say **YES** to $\mathbf{M}_t$ when and if it shows up in the enumeration. This gives us an *acceptor* for those (**M**, t)'s which fail to halt. Since we already have an acceptor for those (**M**, t)'s which halt, we'd have a recognizer for the **HALT-SET** and by previous diagonal argument, we've shown that no such recognizer can exist.

LATER: Show that $\mathbb{N}^*$ is countable. Trick: $\mathbb{N}^K$ is countable, for each K. Consider the table whose K^{th} row the listing of $\mathbb{N}^K$. For each n, $C^*(n) = x^{PAIR_1^{-1}(n)}_{PAIR_2^{-1}(n)}$, that is the n^{th} element of $\mathbb{N}^*$ is the $(PAIR_2^{-1}(n))^{\text{th}}$ element of $\mathbb{N}^{PAIR_1^{-1}(n)}$, i.e. go to the $PAIR_1^{-1}(n)^{\text{th}}$ row of the table, and find the $PAIR_2^{-1}(n)^{\text{th}}$ element of this row.