Some Notes on Problems and Complexity

Paul Cull

Department of Computer Science
Oregon State University

April 22, 2011

Chapter 1

Generators

1.1 Types of Generators

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 $O(\,(\log(GEN(\mathrm{n})) + \log n\,)^K\,)$, for some K, i.e. polynomial time in both the size of the input and the size of the output. For the Fibonacci numbers, $\mathrm{FIB}(n)$ 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 Generators and Recognizers

How general is the idea of a generator? Clearly, every recursive set has a partial generator. Since each recursive set, **S**, has a recognizer, we can define a partial generator for the set in terms of the recognizer by:

$$Gen_{\mathbf{S}}(n) \;=\; \begin{cases} n & \text{if } Rec_{\mathbf{S}}(n) \;=\; \textbf{YES} \\ \text{Does NOT HALT} & \text{if } Rec_{\mathbf{S}}(n) \;=\; \textbf{NO}. \end{cases}$$

Obviously, this generator produces every element in **S** and does not output any element not in **S**.

To get a **generator** for **S**, we need to assume that **S** is non-null since obviously the null set cannot have a generator because there is nothing in the null set to generate. So, let us assume that x_0 is an element of **S** and define

$$Gen_{\mathbf{S}}(n) \;=\; \begin{cases} n & \text{if } Rec_{\mathbf{S}}(n) \;=\; \textbf{YES} \\ x_0 & \text{if } Rec_{\mathbf{S}}(n) \;=\; \textbf{NO}. \end{cases}$$

This is a generator because it always outputs an element of **S** and each n in **S** is generated. This generator has the defect that it will output the element x_0 multiple times. In fact, if the complement of **S** is infinite, x_0 will be output for an infinite number of n's.

If **S** is an infinite set, we can build a **strong generator** by remembering the elements that have been output and never outputting a remembered element.

$$Gen_{\mathbf{S}}(n) \;=\; \begin{cases} n & \text{if } Rec_{\mathbf{S}}(n) \;=\; \textbf{YES} \text{ and } n \notin \{Gen_{\mathbf{S}}(0), \cdots, Gen_{\mathbf{S}}(n-1)\} \\ Gen_{\mathbf{S}}(n+1) & \text{Otherwise.} \end{cases}$$

Because **S** is infinite, the call to $Gen_{\mathbf{S}}(n+1)$ will eventually terminate when it has found a previously ungenerated element. A minor difficulty is that we don't know how long this generator will take to output an element. This is not only because of the time the recognizer takes, but also because there may be large gaps between consecutive elements of **S**. On the other hand this generator does have the virtue that it produces the elements of **S** in numeric order from the smallest element up.

Ex 1.2. Show that an infinite set **S** is Recursive if and only if it has a strong generator $GEN_{\mathbf{S}}(n)$ which satisfies $GEN_{\mathbf{S}}(n) \;<\; GEN_{\mathbf{S}}(n+1)$ for all n.

1.3 Generators and Acceptors

We note that our loosest definition of generator exactly captures **RE**.

Theorem 1. *A set is in* **RE** *(has an* acceptor*)* **iff** *the set has a* **partial generator**.

If we have an acceptor for **S** define:

$GEN_{\mathbf{S}}(n) = n$ if $\mathrm{Acc}_{\mathbf{S}}(n) =$ **YES**.

If we have a generator for **S** define:

$\mathrm{Acc}_{\mathbf{S}}(x) =$ **YES** if there is some n so that $GEN_{\mathbf{S}}(n) = x$.

The difficulty is that GEN may fail to halt for various values of its input which are less than the n on which GEN produces x. We solve this problem by using the inverse of the pairing function. Let $n = PAIR_1^{-1}(K)$ and $I = PAIR_2^{-1}(K)$. Now, we put a WHILE loop with control variable K around GEN, so that on each iteration, GEN will be run for I steps on input n. Since if x is ever generated for some n, then there is some I so that $GEN(n)$ will output x after at most I steps, and so there is a K so that our loop will produce x, and then our procedure will accept x, and thus be an acceptor for **S**.

This theorem can be strengthened to replace **partial generator** with **generator** if we assume that the set is *non-null*. Further, if we are willing to assume that the set is *infinite*, we can replace **partial generator** with **strong generator**. We will only prove the "strong" result.

Theorem 2. *An infinite set is* **RE** *if and only if it has a* **strong generator**.

Proof. (IF) Assume that **S** has the strong generator $GEN_{\mathbf{S}}(n)$, the following program is $Acc_{\mathbf{S}}(x)$:

```
WHILE    I ≥ 0      DO
     IF GEN_S(I) = x THEN      RETURN( YES )
                     ELSE      I = I + 1
END
```

Note that if $x \notin \mathbf{S}$, this acceptor will run forever.

(ONLY IF) Assume $Acc_{\mathbf{S}}(x)$ is the acceptor for **S**, the following program is the strong generator $GEN_{\mathbf{S}}(n)$ for **S**:

```
START = n
WHILE    START ≥ n      DO
     I = PAIR_1^{-1}(START)
     J = PAIR_2^{-1}(START)
     IF Acc_S^{(I)}(J) = YES
        and J ∉ {GEN_S(0), ··· , GEN_S(n − 1)}
        THEN      RETURN( J )
        ELSE      START = START + 1
END
```

Of course, the set $\{GEN_{\mathbf{S}}(0), \cdots, GEN_{\mathbf{S}}(n-1)\}$ is empty for $n = 0$. The notation $Acc_{\mathbf{S}}^{(I)}(J)$ means run $Acc_{\mathbf{S}}$ for I steps on input J. Clearly each $J \in \mathbf{S}$ is produced for some n because J is accepted by $Acc_{\mathbf{S}}$ using some number of steps I. The assumption that **S** is infinite assures us that eventually $GEN_{\mathbf{S}}(n)$ will produce some element, and the inclusion test assures us that no element will be generated twice. □

Ex 1.3. Show that a non-null set **S** is **RE** if and only if it has a generator. (HINT: Assume that you "know" $x_0 \in \mathbf{S}$.)

1.4 Generators and RE

One of the reasons for defining generators was to give an alternate definition of **RE** as the sets that can be generated. Our familiar theorem that **Recursive** $=$ **RE** $\cap$ **coRE** now becomes:

Theorem 3. *If SET is an infinite set and its complement $\overline{SET}$ is also infinite, then SET has a recognizer if and only if both SET and $\overline{SET}$ have generators.*

The recognizer gives us an acceptor for SET and an acceptor for $\overline{SET}$, and by a previous construction, these acceptors give us generators for SET and $\overline{SET}$. On the other hand if we have the two generators, we can run them in "parallel" and say **YES** to x if SET's generator produces x, and say **NO** to x if $\overline{SET}$'s generator produces x.

Note that by Theorem 2 we may assume that the generators for SET and $\overline{SET}$ are **strong** generators.

1.5 How Fast Are Our Generators?

Paradoxically we can make our generators "faster" by slowing down the simulation of the acceptors. Similar to the above generator we give the following program for $GEN_{\mathbf{S}}(n)$:

$I = PAIR_1^{-1}(\mathrm{n})$
$J = PAIR_2^{-1}(\mathrm{n})$
IF $Acc_{\mathbf{S}}^{(\log I)}(J) =$ **YES**
THEN RETURN(J)
ELSE RETURN(x_0)

Notice that this generator has the default x_0 which we assume we know is in **S**. We have slowed the simulation by running the acceptor for $\log I$ steps rather than for I steps.

What's the running time of this program? We recall that the size of the input is $\log n$. For a *reasonable* pairing function, both I and J will have $O(\log n)$ bits and they can be computed in polynomial time in $\log n$. So, running the program will use $\log I = O(\log n)$ steps of the acceptor. Now for any *reasonable* model of computation, one step of $Acc_{\mathbf{S}}(J)$ should take at most polynomial time in the size of its input, i.e. $O((\log J)^K)$ for some K. Running the acceptor for $\log I$ steps should then take

$$\log I \cdot O((\log J)^K) \leq \log n \cdot O((\log n)^K) = O((\log n)^{K+1}).$$

All of the other instructions in the program, i.e. computing the inverse pairing functions, the IF test, and the RETURNS should all take polynomial time in $\log n$. In total, the program has $O(\,(\log n)^L\,)$ run time for some L, i.e. the program has polynomial run time. This gives us:

Theorem 4. *Every* **RE** *set has a* **weak** *polynomial time generator.*

This result is disappointing because we hoped that limiting the time for the generator would restrict the sets that could be generated.

Even if we require the generators to be strong generators, it is not clear which class of sets have polynomial time strong generators. For the Fibonacci numbers, we could turn the generator into a polynomial time recognizer. The Perfect Squares also have a polynomial time strong generator, but when we converted that generator into a recognizer, the recognizer took exponential time.

1.6 TOT is HARD

Here, we want to show that there are some sets which cannot have generators by giving a specific example.

Theorem 5. *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 is 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.