Implicit Computation: An Output-Polynomial Algorithm for Evaluating Straight-Line Programs

Troy A. Shahoumian
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-1999-25
February, 1999

E-mail: troy_shahoumian@hp.com

theory of computation, randomized algorithms, straight-line programs, output-polynomial algorithms

Inputless straight-line programs using addition, subtraction and multiplication are considered. An output-polynomial algorithm is given for computing the output of such a program. The program runs in time polynomial in the length of the program and the length of its output. As a consequence, even if intermediate results are exponentially longer than the program, the output can be computed efficiently when the output's size is small. This algorithm also applies to programs with integral inputs if the size of the inputs are considered to be part of the program's size.

Internal Accession Date Only

1 Introduction

This paper gives an output-polynomial algorithm for computing the output of an inputless straight-line program. In an inputless straight-line program all variables are defined prior to use. Being straight-line means that there are no loops or conditional statements. Such a program is a list of statements which are executed sequentially. The allowed operations are (1) loading an integer specified by the program into a variable, and (2) adding, subtracting or multiplying two previously defined variables and assigning the result to a third variable.

The length of the output can be exponentially longer than the original program. This can be achieved by repeatedly squaring a number. The length of a number doubles with each squaring. Therefore, no algorithm can evaluate a program's output in time which is polynomial in the program's length. The output may simply be too long. This paper gives an output-polynomial algorithm for evaluating a straight-line program. Such a program runs in time polynomial in the length of the straight-line program and its output. The length of intermediate results is not a factor in the running time. As a result, even if intermediate results are long, if the output is small it can be found efficiently using our algorithm.

This result came from considering the Program Sign Problem: given an inputless straight-line program, it is unknown whether the sign of the output can be determined in time polynomial in the program's length. The results of this paper imply that the sign can be determined when the output's length is polynomially bigger than the program's size. In such cases, the program's

output can be computed in time polynomial in the program's length and the output's sign can be read off.

2 The Algorithm

For purposes of the analysis, assume that the only constant loaded by any instruction is unity. If a program loads the constant C, this can be replaced by $O(\log C)$ instructions where the only constant loaded is unity. Using this convention, the number of instructions is a good measure for the size of a program. Otherwise, the number of instructions would not be a good measure of the program's size; the lengths of integers specified in the program would not be accounted for.

The algorithm is quite simple and is as follows:

Algorithm Compute Program's Output
Input: An inputless program P which has n instructions and only loads the constant unity.
Output: With probability $1/2$, it outputs the correct output of P.

$i \leftarrow 1$;
while true do
    Compute $c \leftarrow P \bmod 2^{2^i}$;
    Pick a prime p having $O(n \log n + i)$ bits;
    If $P \equiv c \bmod p$ then output c;
    Pick a prime p having $O(n \log n + i)$ bits;
    If $P \equiv 2^{2^i} - c \bmod p$ then output $-c$;
    $i \leftarrow i + 1$;
od

An error can occur when the p chosen is composite. An error can also occur when we choose a prime p for which $P \equiv c \bmod p$ or $P \equiv 2^{2^i} - c \bmod p$ but there is not actual equality. The following lemma bounds the probability of the latter occuring.

Lemma 2.1 *Let k be an integer with $0 < |k| < 2^{n+1}$. If p is a randomly chosen prime with $i+cn\log n$ bits, then the probability—taken over the choice of p—that $k \equiv 0 \bmod p$ is $O(1/c2^i)$.*

Proof: When p is chosen to have $cn\log n$ bits, the failure probability goes to $O(1/c)$ for large n because

$$\begin{aligned} \text{failure probability} &= \frac{\text{Number of primes dividing } k}{\pi(cn\log n)} \\ &\sim n \Big/ \frac{cn\log n}{\log(cn\log n)} \\ &= \frac{\log(cn\log n)}{c\log n}. \end{aligned}$$

This probability goes to $1/c$ for large n. For large n, $\pi(4n) > 2\pi(n)$ so when choosing the prime, increasing the number of bits by $2i$ will cut the failure probability by 2^i. □

The probability of failure in the ith iteration should be $1/2^{1+i}$. As mentioned before, there are two sources of possible error: choosing a composite and choosing a prime which gives equality modulo that prime when equality does not actually occur. In the ith iteration, the probability of either of these events should be 2^{-3-i}. In order for the second source of error to be less than 2^{-3-i}, Lemma 1 shows that a prime with $O(n\log n)$ bits suffices. Let $L(n,\, i)$ be the length of the primes chosen in the ith iteration; i.e., $L(n,\, i) = k \cdot (n\log n + i)$ for some constant k.

Let b be the number of bits required to represent the output of the program P and let $b' = 2^{\lceil \lg b \rceil}$ be the smallest power of two not less than b. In the last iteration of the algorithm, the primes chosen have $L(n, b')$ bits.

Let's analyze the running time of choosing the primes in the ith iteration with the desired probability of failure. About one in n n-bit numbers are prime. Specifically, among all integers up to 2^{n+1}, by the prime number theorem about one in $\log(n+1)$ are prime. There are polynomial-time algorithms for testing the primality of a number. For example, given a n-bit number, Rabin's algorithm [Rab80] takes $O(kn^2)$ time to achieve a probability of failure of less than $1/2^{2k}$. In order to choose a random prime, random integers are chosen and tested for being prime. In the ith iteration of the algorithm there will be $O(L(n, i))$ primality tests done in order to choose a $L(n, i)$-bit prime; we want each primality test to fail with probability at most $1/2^{3+i+\log(k_1 L(n,i))}$ for some constant k_1. This will give a probability of choosing a composite in the ith iteration to be 2^{-3-i}. In the ith iteration choosing a $L(n, i)$ bit prime takes $O((i + \log L(n, i))L(n, i)^2)$ time.

In the last iteration, computing c takes $O(b'n)$ time because c is being computed modulo $2^{b'}$. Each of the two tests—that check whether $P \equiv c \bmod p$ and $P \equiv 2^{2^i} - c \bmod p$—take $O(nL(n, b'))$ time. This is dominated by the $O((b' + \log L(n, b'))L(n, b')^2)$ time required to choose the primes. The running time of the algorithm is $O(b(b+\log L(n, b))L(n, b)^2)$. This is polynomial in the length of the program and the output.

3 Conclusion

Traditional methods of computing require enough memory to store all intermediate results. Our results show that this requirement is unnecessary. Our algorithm requires enough memory to store the result (is this necessary?) and time which is polynomial in the length of the program and its output. The length of intermediate results—even if they are exponentially larger than the amount of available RAM—is not a factor in the running time of the algorithm.

References

[Rab80] M.O. Rabin, "Probabilistic Algorithm for Testing Primality." *Journal of Number Theory*, 12 (1980) 128–138.