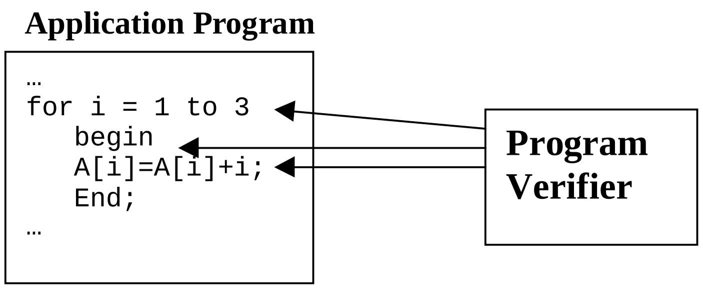
CS174 Lecture 12

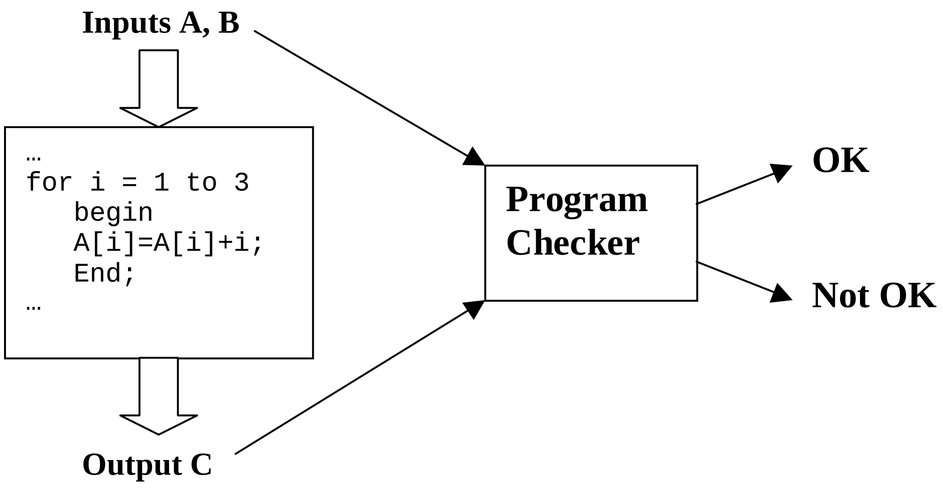
Program Checking

Many application programs (e.g. air traffic control, financial management) have become so complicated that it's very difficult to discover and correct errors and produce a correct (or sufficiently correct) program. *Program checking* is one technique for helping discover errors by continually checking the program's results. It is different from *program verification* where the idea is to check the program itself. A program verifier is shown below. It looks at actual code, and may or may not need to run the code on particular inputs.



```
…
for i = 1 to 3
   begin
   A[i]=A[i]+i;
   End;
…
```

By contrast, a program checker can only check the program on particular inputs. The checker looks at the input to the program, looks at its output, and tries to figure out if they are consistent.



```
…
for i = 1 to 3
   begin
   A[i]=A[i]+i;
   End;
…
```

A program checker will typically run every time the program is run. If it ever discovers an error, the programmer can use that input to figure out what went wrong. But unlike the program verifier, the checker wont point to a particular line(s) of code that is suspect. A checker is like an advanced form of the C++ `assert` statement.

The basic premise of program checking is that there is often a faster method for checking a calculation than for doing the calculation. (e.g. the method of “nines” for checking arithmetic). So the call to the checking program usually takes a vanishingly small amount of time compared to the calculation itself. Adding the checking improves correctness without increasing running time.

Checking Matrix Multiply

Matrix multiplication can be programmed very easily, so wouldn’t normally require checking. However, there are versions of matrix multiply which use advanced divide-and-conquer techniques to improve running time. E.g. there is an $O(n^{2.376})$-time algorithm that is extremely complicated. After writing such a program, you could run it on lots of test cases and compare it to a simple implementation, but you could never be sure that it wouldn’t fail on some strange case that you haven’t seen yet. Writing a checker would guard against that – or rather it would immediately notify the user that something was wrong and at least stop them from making use of incorrect results for some important task. Or you could decide to call a naïve but correct version of matrix multiply if your checker said the fast version had made a mistake. The user would always get the correct answer but might have to wait longer.

The matrix multiply checker program looks like this:

```
Input A, B, C
Choose r at random from $\{0,1\}^n$
Compute y = A(Br) and z = Cr
If (y = z)
   Output “Probably OK”
Else
   Output “not OK”
End
```

This routine requires only 3 matrix-vector multiplies to compute y and z, so its running time is $O(n^2)$. Note that there is an asymmetry in the output. When the checker says “not OK” it definitely knows that the program is incorrect. When it says “Probably OK”, it does not know that the program is correct, in fact it does not even know that the particular output C is really the product of A and B. Because of the random choice of r, it can only assert that C is the product of A and B with some probability.

Aside:

Program checking, because it relies on exact tests for equality, works well for certain types of data such as integer and string data. Floating point numbers unfortunately fail to satisfy most of the axioms that simple checkers rely on. It’s not impossible to write checkers for floating-point calculations, but they become so complicated that there is a real issue of making sure that the checker is itself correct.

Theorem

Let A,B and C be nxn matrices such that $AB \neq C$. Then for r chosen uniformly at random from $\{0,1\}^n$, then $Pr[Cr = ABr] \leq 1/2$.

Proof

Let D=AB-C. Pick any row of D that has non-zero entries (there must be one since AB is different from C). Let d be the n-vector whose entries come from this row of D. By assumption of the theorem, Cr=ABr, so (AB-C)r=Dr=0. Therefore dr =0. That means

$$\sum_{i=1}^{n} d_i r_i = 0$$

There must be at least two non-zero d_i terms in the sum for cancellation to happen. The r_i's are random {0,1}-valued numbers. Let d_k be the last non-zero d_i. Suppose all the r_i's are chosen before r_k. Then there is a 50-50 chance that r_k will be 0 or 1, and only one of those choices will cause the sum to be zero. So the probability of dr being zero is ≤ 0.5.

In fact the probability of ABr=Cr is much smaller than 0.5 most of the time. Its very unlikely for a collection of numbers like the d_i to sum to zero in their inner product with a random r. But if AB and C are only slightly different (e.g. differ by 1 in exactly two locations in the same row) then the probability that ABr=Cr will be close to 0.5.

Checking Polynomial Identities

Any program that involves only +,-,*,/ generates quantities which are polynomials or quotients of polynomials in the input arguments. Such a program can be checked using a generalization of the above technique.

Case 1: Products of polynomials

Suppose you implement a fast version of polynomial product. That is $P_1(x)*P_2(x)=P_3(x)$. You might use FFTs, or one of the recursive divide-and-conquer schemes (e.g. Karatsuba's method). Those methods are fairly complicated, so the issue of correctness is a serious one. The fastest of these methods has a running time of O(n log n) in the degree of the polynomials.

To check a polynomial multiply program, randomly select an r (integer) in the range {0,2n}. Then evaluate $P_1(r)$, $P_2(r)$ and $P_3(r)$. If $P_3(x)$ is correct, then $P_1(r)\ P_2(r) = P_3(r)$. If this identity doesn't hold, $P_3(x)$ is incorrect. Note that polynomial evaluation takes O(n) time if you do it in a clever way. Namely if $P_1(x)$ is

$$P_{1,0} + P_{1,1}x + P_{1,2}x^2 + \ldots. + P_{1,n}x^n$$

Then you can evaluate it at r in linear time as:

$$P_{1,0} + r*(P_{1,1} + r*(P_{1,2} + r*(P_{1,3} \ldots + (P_{1,n-1} + rP_{1,n})\ldots)))$$

That is, you work from the inside out, and do exactly n multiplies and n adds.

Let $Q(x) = P_1(x)*P_2(x)-P_3(x)$. Then if $P_3(x)$ is correct, $Q(x)$ is the zero polynomial. Otherwise $Q(x)$ is a polynomial with some degree d, and the difference between $P_1(r)$ $P_2(r)$ and $P_3(r)$ is equal to $Q(r)$. Now if $Q(x)$ is not the zero polynomial ($P_3(x)$ incorrect), then $Q(r)$ can be zero for at most d values of r (because a polynomial of degree d has at most d real roots). The probability that a random value for r makes $Q(r) = 0$ when $Q(x)$ is not zero is therefore:

$d/(2n+1) < ½$ because $d < n+1$

So each evaluation has probability at least ½ of catching an error in the polynomial $P_3(x)$. It takes O(n) time to check each answer (each $P_3(x)$), and since computing $P_3(x)$ takes at least O(n log n) time, this checker is still useful.

Checking General Identities

The verification methods we've seen so far use a trick that might be called black-box evaluation. A black box is a procedure that allows you to compute with an object (like a matrix or a polynomial) without having an explicit representation for that object. E.g. the checker for matrix multiply contained a black box to compute (AB)r, but since it computed that product as A(Br) there was never an explicit representation of the matrix product AB. Similarly, the polynomial checker computed $P_1*P_2(r)$ as $P_1(r)*P_2(r)$, and it never explicitly computed P_1*P_2.

For more general programs that compute polynomials, if there is a black box available, then the program can be checked. The checking again uses random elements. In the general case, the polynomial being computed will be multivariate. Let $Q(x_1,\ldots x_n)$ be the difference between the actual and computed polynomials as before. By making random evalutions for $x_1,\ldots,x_n$, we can check whether the program is correct. We use the following theorem:

Theorem (Schwartz-Zippel)

Let $Q(x_1,\ldots,x_n)$ be a multivariate polynomial of total degree d which is not identically zero. Let $r_1,\ldots,r_n$ be chosen uniformly at random from $\{0,\ldots,M-1\}$. Then

$$\Pr[Q(r_1,\ldots,r_n) = 0] \leq d/M$$

Note: the total degree of a multivariate polynomial is the maximum total degree of all of its terms. A multivariate polynomial will contain terms like:

$x_1^{d1}x_2^{d2}\ldots x_n^{dn}$

and the total degree of this term is $d_1+d_2+\ldots+d_n$.

Proof: By induction on the number of variables n. Assume the theorem holds for n-1. Write $Q(x_1,\ldots,x_n)$ as a polynomial in x_n with coefficients which are polynomials in $x_1,\ldots,x_{n-1}$:

$$Q(x_1,\ldots,x_n) = Q_m(x_1,\ldots,x_{n-1})x_n^m + Q_{m-1}(x_1,\ldots,x_{n-1})\, x_n^{m-1} + \ldots + Q_0(x_1,\ldots,x_{n-1})$$

The degree of this polynomial in x_n is $m \leq d$, and the total degree of $Q_m(x_1,\ldots,x_{n-1})$ is $\leq (d-m)$. There are two cases:

1. $Q_m(r_1,\ldots,r_{n-1}) = 0$, which by the inductive hypothesis happens with probability $\leq (d-m)/M$

2. $Q_m(r_1,\ldots,r_{n-1})$ is non-zero, and $Q(r_1,\ldots,r_{n-1},x_n)$ is has degree m in x_n. For a degree m polynomial, there are at most m roots, so at most m values for x_n can make Q vanish. The probability that r_n takes on one of these values is $\leq m/M$.

The total Pr[Case 1 or Case 2] $\leq (d-m)/M + m/M = d/M$, which proves the theorem for n. QED