Designing programs that check their work*

Manuel Blum
Comp. Sci. Division
U. of California
Berkeley, CA 94720

Sampath Kannan†
Comp. Sci. Division
U. of California
Berkeley, CA 94720

Subject Classification D.2.4 F.2.0 F.3.1 G.3

Abstract

A *program correctness checker* is an algorithm for checking the output of a computation. That is, given a program and an instance on which the program is run, the checker certifies whether the output of the program on that instance is correct. This paper defines the concept of a program checker. It designs program checkers for a few specific and carefully chosen problems in the class FP of functions computable in polynomial time. Problems in FP for which checkers are presented in this paper include Sorting, Matrix Rank and GCD. It also applies methods of modern cryptography, especially the idea of a probabilistic interactive proof, to the design of program checkers for group theoretic computations.

Two strucural theorems are proven here. One is a characterization of problems that can be checked. The other theorem establishes equivalence classes of problems such that whenever one problem in a class is checkable, all problems in the class are checkable.

*Supported by NSF Grant #CCR88-13632

†currently at Department of Computer Science, University of Arizona, Tucson, Arizona 85721

1 Introduction

In this paper we introduce the concept of a *program checker*. A program checker for a program P is itself a program C. For any instance I on which program P is run, C is run subsequently. C either certifies that the program P is correct on I or declares P to be buggy.

There have been other methods proposed for gaining confidence in the output of programs. For example, *program verification*[9] seeks to achieve this by *proving* that a program is correct. Program verification suffers from the problem that it is very hard to prove programs correct. It has also been argued that proofs of correctness of programs do not improve our confidence in their correctness because of the nature of these proofs[13]. For a recent discussion of the role of verification in software development see [3].

In *program testing*[12] we run the program on *test inputs* for which the output is known and see if the program output matches the expected output. Testing is a fairly *ad hoc* technique. There are no general methods for generating test data and no theorems are proven about the behavior of a program that passes the tests.

In addition there has been work in the the theoretical computer science community on the concept of *helping*[27, 35] which may be regarded as a deterministic version of checking.

Program checking is easier to do than verification; it yields mathematical proofs about program behavior unlike testing; it allows coin-tossing, greatly enhancing the power of the checker in comparison to the model of helping above.

The ideas in this paper arise from cryptography, probabilistic algorithms, and program testing. Particularly important for this work are the interactive proofs of Goldwasser, Micali and Rackoff[19] and subsequent related work. As will be seen, several of the correctness checkers constructed in

this paper use probabilistic interactive proofs as a first step in the design. Equally important for this work are the papers on randomized algorithms of Rabin[33] and Freivalds[16]. The latter, remarkably enough, includes excellent program checkers for integer, polynomial, and matrix multiplication. The works of Budd and Angluin[10] and Weyuker[38] are relevant in that they too seek to give program testing a rigorous mathematical basis.

The notion of program checking as used in this paper was first formally defined by Blum and Kannan[6]. This paper draws heavily from [6]. In [6] the concept of program checking was defined, checkers were exhibited for some group-theoretic problems and for selected problems in P, and the class of problems having polynomial-time checkers was characterized.

Since then several papers have shed light on this problem. Blum, Luby, and Rubinfeld[7] extend the notion of program checking one step further in several directions. They focus on a large collection of numerical problems that includes integer multiplication and modular multiplication. For these problems they show that it is not only possible to *detect* errors in programs, but also to *correct* errors in programs that are 'mostly correct.' They also provide efficient tests for determining whether a program is 'mostly correct.' In the process their results yield some of the few program testers with *provable* performance. If a program passes a self-test *a la* [7] on instances of some size n, then it will be possible to prove a theorem that says that with high probability, P is correct on 'most' instances of size n, where 'most' can be precisely quantified.

Another concept introduced in [7] is that of a *library of programs*. This allows a checker for one problem in the library to call programs for other problems in the library as long as all the programs in the library can be checked by these means. This extension allows for the design of efficient and simple checkers for problems which had hitherto had much more complex checkers.

Adleman, Huang, and Kompella[1] provide checkers for several number-theoretic problems in-

cluding integer greatest common divisor. In [6] it was conjectured that an efficient checker for g.c.d. would be hard to find. Lipton[29] considers programs that have been tested in some way to ensure that they are mostly correct and shows how one can correct the errors in the program by transforming a given instance to several random instances and computing the answer to the given instance from the answers to the random instances. Lipton[29], building on the work of Beaver and Feigenbaum[4], shows how polynomials in general and the permanent in particular are amenable to this technique. Rubinfeld[34] extends the notion of checking to parallel checking while Blum *et al.*[8] extend it to programs that store and retrieve data from unreliable memory. Kannan and Yao[24] have considered the problem of checking coin-tossing programs that produce specified output probability distributions.

There are several concepts in complexity theory that are intimately related to checking. Two such concepts are *coherence* and *random-self-reducibility*. These concepts have been considered extensively in the literature. Definitions of these concepts and their relation to program checking can be found for example in [5, 14].

The rest of this paper is organized as follows: A more formal description of the program checking model is given in section 2. In section 3 we illustrate the concept with the prototypical example of the graph isomorphism problem. In section 4 we derive structural theorems which allow us to derive checkers for one problem from checkers for others. In section 5 we present some program checkers for group-theoretic problems. This section demonstrates the close connections between the design of program checkers and the design of interactive proofs. In section 6 we present checkers for a number of common functions that can be computed in FP. The specific problems considered are Extended GCD, Sorting, and Matrix Rank. Finally in section 7 we characterize the class of problems that have polynomial-time checkers.

2 Program Checkers

Let π denote a (computational) decision or search problem. For x an input to π, let $\pi(x)$ denote the output of π. Let P be a deterministic program (supposedly) for π that halts on all instances of π. We say that such a program P has a *bug* if for some instance x of π, $P(x) \neq \pi(x)$.

Define an *efficient program checker* C_π for problem π as follows: $C_\pi^P(I;k)$ is any probabilistic (expected-poly-time) oracle Turing machine that satisfies the following conditions, for any program P (supposedly for π) that halts on all instances of π, for any instance I of π, and for any positive integer k (the so-called 'security parameter') presented in unary:

1. If P has no bugs, i.e., $P(x) = \pi(x)$ for all instances x of π, then with probability $\geq 1 - 1/2^k$, $C_\pi^P(I;k) = CORRECT$ (i.e., $P(I)$ is $CORRECT$).

2. If $P(I) \neq \pi(I)$, then with probability $\geq 1 - 1/2^k$, $C_\pi^P(I;k) = BUGGY$ (i.e., P is $BUGGY$).

This probability is computed over the sample space of all finite sequences of coin flips that C could have tossed.

Some remarks are in order:

i. The running time of C above *includes* whatever time it takes C to submit inputs to and receive outputs from P, but *excludes* the time it takes for P to do its computations.

ii. In the above definition, if P has bugs but $P(I) = \pi(I)$, ie. buggy program P gives the correct output on input I, then $C_\pi^P(I;k)$ may output $CORRECT$ or $BUGGY$.

It is assumed that any program P for problem π halts on all instances of π. This is done in order to help focus on the problem at hand. In general, however, programs do not always halt, and the definition of a 'bug' must be extended to cover programming errors that slow a program

down or cause it to diverge altogether. In this case, the definition of a program checker must also be extended to require the additional condition:

3. If $P(x)$ exceeds a precomputed bound $\Phi(x)$ on the running time, for $x = I$ or any other value of x submitted by the checker to the oracle, then the program checker is to sound a warning, namely $C_\pi^P(I;k) = \mathit{TIME}$.

In the remainder of this paper, it is assumed that any program P for a problem π halts on all instances of π; so condition 3 is everywhere suppressed.

It is possible to extend the notion of program checking to probabilistic algorithms in BPP. In order to do this, we simply run the program sufficiently often to make the probability of error of (a correct) program much smaller than $1/2^k$. Then we simply treat the program as though it were a deterministic program and check it accordingly. In the rest of the paper we only consider deterministic programs with the assurance that all of the results about checkers for deterministic programs can be extended to checkers for probabilistic programs.

In this approach to program correctness the question naturally arises: If one cannot be sure that a program is correct, how then can one be sure that its checker is correct? This is a very serious problem!

One solution is to *prove* the checker correct. Sometimes, this is easier than proving the original program correct, as in the case of the *Extended GCD* checker of section 6. Another possibility is to try to make the checker to some extent independent of the program it checks. To this end, we make the following definition: Say that a (probabilistic) program checker C has the *little oh* property with respect to program P if and only if the (expected) running time of C is little oh of the running time of P. We shall generally require that a checker have this little oh property with

respect to any program it checks.

The principal reason for this is to ensure that the checker is programmed *differently* from the program it checks. For instance, if there are two programs for a problem with the same running times this definition disallows the checker from running one as a 'check' for the other. This is what we desire in our checker. However, this definition does not necessarily constrain us to design *efficient* checkers, for although the running time of the checker is little oh of the program's running time, this does not account for the time spent in calls to the program. If the checker only made one call to the program, running the checker would not result in an increase in the asymptotic running time of the program. In general this is hard to achieve, but for a significant subclass of problems (such as the ones considered in [7]) one can design checkers that run in time that is no worse than a constant times the running time of the program being checked, taking into account the time spent running the program being checked.

3 An Example: Graph Isomorphism

We present an example of a good checker. Our checker is an adaptation of Goldreich, Micali, and Wigderson's interactive proof system for Graph Isomorphism (see [18]). The model in [18] relies on the existence of an all-powerful prover. The prover is replaced here by the program being checked. The power of the program turns out to be sufficient to simulate the prover for this application. The checker that results is a practical way to check computer programs for graph isomorphism. Graph isomorphism is a problem with a lot of heuristics that work on most instances. Appending a checker to a heuristic gives us confidence in the output of the (possibly unproven) heuristic.

The Graph Isomorphism decision problem is defined as follows:

Graph Isomorphism (GI):

Input: Two graphs G and H.

Output: *YES* if G is isomorphic to H; *NO* otherwise.

The checker $C^P_{GI}(G, H; k)$ checks program P on input graphs G and H.

Begin

Compute $P(G, H)$.

if $P(G, H) = YES$, **then**

Use P (as if it were bug-free) to search for an 'isomorphism' from G to H.

(This is done by a standard self-reduction as in Hoffmann[22, pages 24–27].)

Check whether the resulting correspondence is an isomorphism.

If not, **return** *BUGGY*; **if** yes, **return** *CORRECT*

if $P(G, H) = NO$, **then**

Do k times:

Toss a fair coin.

if coin = heads **then**

generate a random permutation G' of G.

Compute $P(G, G')$.

if $P(G, G') = NO$ **then return** *BUGGY*

if coin = tails **then**

generate a random permutation H' of H.

Compute $P(G, H')$.

if $P(G, H') = YES$ **then return** *BUGGY*.

End-do

Return *CORRECT.*

End

The above program checker correctly checks *any* computer program whatsoever that is purported to solve the graph isomorphism problem. Even the most bizarre program designed to fool the checker will be caught, when it is run on any input that causes it to output an incorrect answer. The following theorem proves this formally:

Theorem 3.1 *: If P is a correct program for graph isomorphism, then C^P_{GI} always outputs* correct. *If $P(G,H)$ is incorrect then $Prob(C^P_{GI}$ outputs* correct*$) \leq \frac{1}{2^k}$. Moreover, C^P_{GI} runs in polynomial time.*

Proof: Clearly C^P_{GI} runs in polynomial time in our way of counting the running time of the checker.

If P has no bugs and G is isomorphic to H, then $C^P_{GI}(G,H;k)$ constructs an isomorphism from G to H and (correctly) outputs $CORRECT$.

If P has no bugs and G is not isomorphic to H, then $C^P_{GI}(G,H;k)$ tosses coins. It discovers that $P(G,G') = YES$ for all G' and $P(G,H') = NO$ for all H', and so (correctly) outputs $CORRECT$.

If $P(G,H)$ is incorrect, there are two cases:

1. If $P(G,H) = YES$ but G is not isomorphic to H, then C^P_{GI} fails to construct an isomorphism (since none exists) and (correctly) outputs $BUGGY$.

2. If $P(G,H) = NO$ but G is isomorphic to H, then the only way that C will return $CORRECT$ is if $P(G, G' \text{or} H') = YES$ whenever the coin comes up heads and NO when it comes up tails. But G is isomorphic to H. Since G and H are permuted randomly to produce G' and H', G'

and H' have the same probability distributions. Therefore P correctly distinguishes G' from H' only by chance, i.e., for just 1 of the 2^k possible sequences of coin tosses.

∎

4 Beigel's Theorem

The following theorem is due to Richard Beigel:

Theorem 4.1 (Beigel) *Let π_1, π_2 be two polynomial-time equivalent decision problems. Then from any polynomial time checker for π_1 it is possible to construct a polynomial-time checker for π_2.*

Proof: For simplicity assume initially that π_1 and π_2 are decision problems, reducible to each other by Karp reductions. We have a checker C_{π_1} for π_1 and a program P_2 for π_2. We also have two way polynomial-time transformations, $f_{1,2}$ and $f_{2,1}$ going from π_1 to π_2 and from π_2 to π_1 respectively. The existence of $f_{1,2}$ gives us a program P_1 for π_1 defined in terms of $f_{1,2}$ and P_2. $P_1(x)$ is defined to be $P_2(f_{1,2}(x))$. In our way of counting the running time of the checker checking program P_2, a call to P_1 can be accomplished in polynomial time since $f_{1,2}$ is a polynomial-time function and a call to P_2 counts as 1 step.

To check P_2 on instance y we compute $P_2(y)$ and transform y to an instance z for π_1 using the function $f_{2,1}$. We then use the checker C_{π_1} to check the correctness of P_1 on z. Any call that the checker makes to P_1, including the call on the instance z, is transformed in polynomial time to a call to P_2 by the procedure described above. Being convinced about the correctness of P_1 on z convinces us of the correctness of P_2 on y. If P_2 is correct, then P_1, which is defined in terms of P_2, will be too. Thus the checker will find that P_1 is correct on z, convincing us that P_2 is correct on

y. If P_2 is wrong on y, there are two cases. If P_1 is correct on z, we will discover the contradiction immediately. If P_1 is wrong on z, the checker C_{π_1} is designed to catch precisely this situation and it will declare P_1 to be buggy. Thereby we will be convinced of the bugginess of P_2.

The checker for π_2 described above runs in polynomial time. Let n be the length of the instance of π_2 being checked. The running time of the checker for π_2 can be broken down into the following three components.

- The running time of the checker for π_1 on an instance whose length is polynomial in n.
- One application of the transformation $f_{2,1}$ on an input of length n.
- A polynomial number of applications of the transformation $f_{1,2}$ on inputs whose lengths are polynomial in n.

We show now that the theorem holds even when the problems are possibly search problems and the reductions between the problems are Cook reductions. In this case also we have a program P_1 for π_1 defined in terms of P_2. The proof of correctness of the checker essentially follows along the lines of the proof in the case of Karp reductions. There are just more details to check. The program P_1 will make polynomially many calls to P_2 on each input. The transformation $f_{1,2}$ is replaced by a program which takes an instance of π_1 and in polynomial time produces the set of instances of π_2 to be queried. So also the transformation $f_{2,1}$. ∎

One particular application of Beigel's theorem is to graph isomorphism. Since graph isomorphism is known to have a polynomial-time checker, all of the problems that are polynomial-time equivalent to graph isomorphism also have such checkers.

It is important to note that the statement of Beigel's theorem requires the equivalence of π_1 and π_2. The following example suggests that a reduction in one direction is not sufficient.

Observe that Group Isomorphism (GI) reduces to Extended Group Isomorphism (EGI) where

1. groups are given by multiplication tables, and
2. GI differs from EGI in that a YES answer in the former is an explicit isomorphism in the latter.

We know an efficient checker for EGI but not for GI.

4.1 Generalizing Beigel's Theorem

Let $\mathcal{F}$ be a complexity class and let π_1 and π_2 be problems which are reducible to each other in $\mathcal{F}$. Suppose we have a checker for π_1 in $\mathcal{F}$. Under what conditions does that give us a checker for π_2 in $\mathcal{F}$? We consider the situation in which $\mathcal{F}$ is a deterministic time complexity class. The situation is similar if we replace time complexity by circuit size or circuit depth.

Time Complexity Classes and NC

Let $f : \mathcal{N} \to \mathcal{N}$ be a time complexity function. Suppose having an algorithm whose running time is bounded by $f(n)$ on inputs of length n puts a problem in $\mathcal{F}$. Then we will call f a *time complexity function for* $\mathcal{F}$.

We will call a complexity class $\mathcal{F}$ *robust* if for any two time complexity functions f and g for $\mathcal{F}$, $f + g$, $f \cdot g$, and $f(g)$ are all time complexity functions for $\mathcal{F}$. In other words, $\mathcal{F}$ is a robust class if the sum, product and composition of any two time complexity functions for $\mathcal{F}$ is a time complexity function for $\mathcal{F}$. Examples of robust time complexity classes include P and $\log^{O(1)}(n)$.

Theorem 4.2 *Let $\mathcal{F}$ be a robust time complexity class, π_1 and π_2 two problems reducible to each other in $\mathcal{F}$ and C_{π_1} a checker for π_1 in $\mathcal{F}$. Then there is a checker, C_{π_2} for π_2 in $\mathcal{F}$.*

Proof (sketch): The checker C_{π_2} is constructed along the same lines as the checker constructed in Beigel's theorem. For the running time analysis note that the definition of robustness is precisely the one needed to guarantee that C_{π_2} lies in $\mathcal{F}$. This uses the fact that an algorithm with a running time of $f(n)$ can make at most $f(n)$ calls to an oracle (such as the program being checked) and can only produce outputs (transformed instances) whose lengths are bounded by $f(n)$ on an input of length n. Thus the running time of the checker for π_2 can be bounded by sums, products, and compositions of the running times of the two reductions and of the checker for π_1. This is true even in the case that the reductions between the problems are Cook reductions. ∎

Corollary 4.1 *If π_1 and π_2 are equivalent under NC-reductions and π_1 has an NC-checker, then so does π_2.*

Proof: Although NC is not a time complexity class, the proof follows from the robustness of the class NC. A 'complexity function' for NC can be thought of as an ordered pair of functions, $(size(n), depth(n))$. $size(n)$ is a function from the class P and $depth(n)$ is a function from $\log^{O(1)} n$. Sums, products, and compositions of complexity functions are computed component-wise on the ordered pair representing the function since the arguments in the theorem about the robustness of a time complexity class holds for depth and size as well. The robustness of the classes, P and $\log^{O(1)} n$ establishes the robustness of the complexity functions in NC. This in turn implies that the checker for π_2 is in NC. ∎

4.2 NC-checkers for Problems in P

Using the generalized version of Beigel's theorem, we can prove that all P-complete problems have checkers in NC.

Theorem 4.3 *All P-complete problems have checkers in NC.*

Proof: In the light of the generalized version of Beigel's theorem, it is sufficient to prove that *some* P-complete problem has a checker in NC. This is because all P-complete problems are NC reducible to each other. The particular P-complete problem for which we will provide an NC-checker is the Lexicographically First Maximal Independent Set (LFMIS) [11] problem. ∎

4.3 An NC-checker for LFMIS

Lexicographically First Maximal Independent Set (LFMIS)

Input: A Graph G with the vertices numbered from 1 to n and a vertex v of the graph.

Output: 'Yes' if the LFMIS contains the vertex v and 'No' if the LFMIS does not.

We will present the NC-checker informally as an algorithm for a PRAM. For details on the PRAM model see, for instance, [26].

Step 1: The i^{th} processor asks whether v_i is in the LFMIS. Thus the processors determine the LFMIS.

Step 2: For this step, associated with each vertex is a group of n processors. The i^{th} group of processors are associated with v_i. They assume that the answers obtained in step 1 on queries on $v_1, \ldots, v_{i-1}$ are correct. With this assumption they check to see if the answer to the query on v_i is correct. This can be done in $O(1)$ time, since v_i is in the LFMIS iff there is no edge from v_i to a smaller numbered vertex in the LFMIS.

Thus in the CRCW PRAM model the above checker runs in $O(1)$ time and uses $O(n^2)$ processors. Doing step 2 more carefully allows us to reduce the processor count to $O(n + m)$.

As already mentioned finding an NC checker for LFMIS gives us NC-checkers for all P-complete problems. It is interesting that we can prove that the 'most difficult' problems in P have checkers in NC, although we don't know whether all the decision problems in P have checkers in NC. While there are NC-checkers for all P-complete problems, a big open question is whether all NP-complete problems have checkers in P. In fact there is some negative evidence on this question[15].

5 Checkers for Group Theoretic Problems

Many group theoretic problems have checkers resembling that for graph isomorphism. Subsection 5.1 shows this for two fairly general classes of examples. 5.2 gives a general approach to checker construction that works particularly well for group theoretic problems.

Why all the work on group theoretic problems? Group theory is a rich source of problems with checkers. Very elementary properties of groups such as Lagrange's theorem can often be exploited in the design of checkers. The structure of groups often implies relationships among the correct answers to different instances. These relationships can be used to check the consistency of programs. Sometimes these consistency checks can be proven to be sufficient for ensuring the correctness of a program. For instance, the checker for graph isomorphism that was described in the introduction can be viewed as a group theoretic checker since the problem of graph isomorphism is polynomial-time equivalent to the problem of determining the automorphism group of a graph[31]. For graph isomorphism we essentially check the consistency of the program in the case that the program says that the input graphs G and H are not isomorphic. The structure of the problem implies that

1. G and a random permutation of G are isomorphic and

2. If G is not isomorphic to H, then G is not isomorphic to a random permutation H' of H.

Computation has been used extensively as a tool in group theory. In fact the classification of finite simple groups[21] has both motivated and been aided by computer calculations. The classification has shown that there are just 26 groups that do not belong to any infinite family of groups. These 26 groups are referred to as the sporadic groups. The existence of some of these sporadic groups was confirmed only by computer construction. For all of these reasons checking group theoretic problems is a very fruitful endeavour.

5.1 The Equivalence Search and Canonical Element Problems

The problems (and corresponding checkers) described in this subsection are all stated in terms of a set S of elements and a group G acting on S.

For a, b in S, define $a \equiv_G b$ if and only if $g(a) = b$ for some $g \in G$.

Let $ESP(S, G)$ denote the

Equivalence Search Problem

Input : a, b in S

Output : g such that $g(a) = b$ if $a \equiv_G b$;

NO otherwise.

Proposition 5.1 *Let $ESP(S, G)$ be the* Equivalence Search Problem *for given S and G. Suppose there exists an efficient probabilistic algorithm to find a random $g \in G$ according to the uniform distribution. Then there is an efficient program checker $C^P_{ESP(S,G)}$ for the problem $ESP(S, G)$.*

Examples of the Equivalence Search Problem include graph Isomorphism, quadratic residuosity, a generalization of discrete log and games such as Rubic's cube. Other examples arise in knot theory, block designs, codes, matrices over $GF(q)$, Latin Squares [28, page 32] and in applications of the Burnside and Polya theorems[32].

Related to the *Equivalence Search Problem* is the *Equivalence Decision Problem* defined by:

Equivalence Decision Problem (EDP)

Instance: $a, b \in S$

Question: Is $a \equiv_G b$?

It would be nontrivial to prove a similar proposition for *EDP* because *ESP* does not seem reducible to *EDP* as the following argument indicates:

Recall that for N a positive integer, Z_N^* denotes the group of positive integers less than N that are relatively prime to N under the group operation of multiplication mod N. For p a prime, let $S = Z_p^*$ and $G = Z_{p-1}^*$, where the action of $g \in G$ on $a \in S$ maps a to $a^g \bmod p$. Observe that $a \equiv_G b$ if and only if $b = a^g \bmod p$ for some g in Z_{p-1}^*.

Now suppose we are given an oracle A for factoring. To find g such that $b = a^g \bmod p$ is essentially to solve the discrete log problem which in cryptographic circles is believed to not be solvable in polynomial time, even given the oracle for factoring. On the other hand, the *EDP* *is* solvable in polynomial time given an oracle for factoring. The proof consists in showing that $b = a^g \bmod p$ for some g if and only if $\text{order}(b) | \text{order}(a)$. This is because $x^{\text{order}(a)} = 1 \bmod p$ has exactly $\text{order}(a)$ solutions, namely $a, a^2, \ldots, a^{\text{order}(a)} \equiv 1$. Finally, $\text{order}(a)$ and $\text{order}(b)$ can be determined from the factorization of $p - 1$.

Canonical Element Problem (CEP)

Input: $a \in S$

Output: (c, g) where c is the (unique) canonical element in the equivalence class of a, and $g \in G$ satisfies $g(a) = c$.

Proposition 5.2 *There is an efficient program checker for the canonical element problem, provided there is a probabilistic procedure to select a random $g \in G$ efficiently.*

Remark: If the CEP program should fail by having two or more canonical elements in some class, then we define the (true) canonical element of that class to be the unique element, if any, to which more than half the elements of the class are mapped by the program.

5.2 Group Intersection Problem

We use a two-step approach in designing a checker for group intersection. We first design an interactive proof system and then show that this interactive proof system can be converted into a checker. Babai and Moran[2] have independently (and earlier) provided an interactive proof system for group intersection.

We use the checker for group intersection and Beigel's trick to obtain checkers for several problems that are known to be polynomially equivalent to group intersection.

First, we briefly discuss the various representations of groups on the computer. Three common representations are used. In increasing order of difficulty of manipulation they are: The multiplication table representation, the permutation group representation, and the abstract group representation.

The multiplication table representation explicitly specifies the product of each pair of group elements. In the permutation representation the group is thought of as acting on a set. The group elements are permutations and the group operation is composition. Usually the group is specified by specifying a few (polynomially many in the size of the set) generating permutations. In the abstract group representation, the group is presented by generators which are related by the relations specified amongst them. Only relations implied by the specified relations hold between the generators. This completely specifies the group.

We now describe the checker for the group intersection problem which is the following:

Group Intersection Problem

Input: Two permutation groups G and H specified by generators.

Output: Generators for $G \cap H$.

Let n be the size of the set S on which G and H act. In general the specification of a generating set for G requires $\Omega(n)$ bits and can be done in $poly(n)$ bits. Hence we will take the input length to be n. No probabilistic polynomial-time algorithm is known for solving the group intersection problem. This is not surprising since graph isomorphism is polynomial-time reducible to group intersection. The following interactive proof protocol works for group intersection:

5.2.1 IP Protocol

1. The prover sends the verifier a set of permutations of $[1, 2 \ldots, n]$ which supposedly generate $G \cap H$.

2. The verifier checks that the elements sent by the prover actually lie in $G \cap H$. This involves testing membership in G and H which the verifier can do by the methods of [17]. As a

consequence the verifier is convinced that the elements sent by the prover either generate $G \cap H$ or a proper subgroup of it.

3. The verifier sends the prover an element $\pi \in GH$ which he obtains by selecting random elements $a \in G$ and $b \in H$ and multiplying them together.

4. The prover sends back a factorization of π as $a'b'$ with $a' \in G$ and $b' \in H$.

5. The verifier checks that $a^{-1}a'$ is an element of the group generated by the generators that the prover provided in step 1.

Theorem 5.1 *The above protocol with steps 3-5 repeated k times allows the prover only a $1/2^k$ probability of cheating the verifier.*

Proof: Denote the group generated by the generators that the prover sends in step 1 by M. It is clear after step 2 that $M \subseteq G \cap H$. Steps 3 and 4 are aimed at giving the verifier a random element of $G \cap H$. We have the following lemma to that effect.

Lemma 5.1 *With the notation as in the protocol, $a^{-1}a'$ is a random element of $G \cap H$.*

Proof: Suppose $\pi = ab$ with $a \in G$ and $b \in H$ and $x \in G \cap H$. Then $\pi = (ax)(x^{-1}b)$ where $ax \in G$ and $x^{-1}b \in H$. Thus from these two factorizations of π the element recovered by computing $a^{-1}a'$ is $x \in G \cap H$. Thus for each $x \in G \cap H$ there is a unique factorization of π which along with the factorization $\pi = ab$ yields x.

All that remains to be proved is that every pair of factorizations of π correspond to an element of $G \cap H$. Again suppose that $\pi = ab$ and $\pi = a'b'$ are two factorizations of π. Then $ab = a'b'$. Rearranging we have $a^{-1}a' = b(b')^{-1}$. On the left hand side of the last equation we have an element

of G and on the right hand side an element of H. Since they are equal, both elements must belong to $G \cap H$.

The randomness of the factorization ab of π implies the randomness of the element of $G \cap H$ obtained by this procedure since the prover does not know the factorization ab used by the verifier. ∎

The proof of lemma 5.1 essentially completes the proof of the theorem. We use Lagrange's theorem to note that if M is a proper subgroup of $G \cap H$ then a random element of $G \cap H$ belongs to M with probability at most a half. Performing k repetitions of steps 3-5 reduces the error probability to at most $(1/2)^k$. ∎

5.2.2 Converting the IP Protocol into a Checker

The verifier in the above protocol asks the prover to factor certain elements of GH. To convert this IP protocol into a checker one must show that a program for group intersection can be used to factor elements of GH. If the Factorization Search Problem (FSP) were shown equivalent to the group intersection problem one could use a program for group intersection to factor. FSP is the following problem:

Factorization Search Problem (FSP)

Input: Two permutation groups G and H and a permutation π.

Output: No, if π is not in GH. $a \in G, b \in H$ such that $ab = \pi$ otherwise.

The associated *Factorization Decision Problem (FDP)* is known to be equivalent to group intersection[22, pages 236-241]. The following Lemma shows the equivalence of FSP and FDP.

Lemma 5.2 *FDP is equivalent to FSP.*

Proof: It is obvious that FDP reduces to FSP. All that remains to be shown is that FSP reduces to FDP. The proof relies on the notion of 'strong generators' introduced by [17].

Assume that we have strong generators for G and H as defined in Furst, Hopcroft, and Luks[17]. This can be assumed without loss of generality because any set of generators can be converted to a set of strong generators in polynomial time.

Here is a brief description of the notion of strong generators, M_G, for the group G. M_G is an n x n matrix where n is the size of the permutation domain. The matrix has no entries below the diagonal. Above the diagonal, in position ij we have an entry if and only if there is a permutation in G that fixes (pointwise) the elements $1, 2, \ldots, i-1$ and moves i to j. In case such a permutation exists, the ij^{th} entry is any such permutation in G. It is convenient (and customary) to make the diagonal entries be the identity permutation.

Some properties of this representation are given here without proof. Every element of G can be expressed in a unique way as a product, $\pi_n \pi_{n-1} \ldots \pi_1$ where π_i is from row i of M_G. We are using the convention here that in a string of permutations the leftmost one acts first and the rightmost one last. As a consequence of the previous fact, $|G|$ is the product of the numbers of non-empty entries in each row of M_G. Another consequence is that a random element of G can be obtained by multiplying together random elements in each of the rows of M_G. Also, G_1, the subgroup of G that fixes the point 1, is generated by the entries in rows 2 through n of M_G. Finally, membership in G of a permutation σ can be tested as follows: If σ moves 1 to j, we look in position $1j$ for an entry. If none exists σ is not in G. Otherwise, if π_1 is the entry, $\sigma {\pi_1}^{-1}$ fixes the point 1 and we move on to the second row and check it for membership in G_1. Proceeding thus we will either find that σ is not in G or find an expression for σ as a product of entries in M_G.

Suppose now that π is in GH. We consider H_1, the subgroup of H consisting of all permutations that fix the point 1. Since π is in GH, $\pi = ab$ with a in G and b in H. Also b is equal to some product, $\sigma_n\sigma_{n-1}\ldots\sigma_1$ where σ_i is in the i^{th} row of M_H. Thus there is a permutation, σ_1, in the first row of M_H such that $ab{\sigma_1}^{-1}$ is in GH_1. We can use the oracle for FDP to find out which entry in the first row of M_H has the above property. If this entry is σ_1 we consider $\pi\sigma_1^{-1}$ and factor it in GH_1. A factorization in GH_1 will yield a factorization in GH of π. It can be seen that if this technique is applied recursively it yields a factorization of π in GH. This completes the reduction and shows that the IP protocol described can be converted into a checker. ∎

6 Problems in FP

In this section, some program checkers use their oracle just once (to determine $O = P(I)$) rather than several times. In such cases, instead of the program checker being denoted by $C_\pi^P(I;k)$, it will be denoted by $C_\pi(I,O;k)$. The latter notation has the advantage of clarifying what must be tested for. In cases where the checker is nonprobabilistic, it will be denoted by $C_\pi(I,O)$ instead of $C_\pi(I,O;k)$.

Many problems in FP have efficient program checkers, and it is a challenge to find them. In what follows, we give a fairly complete description of program checkers for just three problems in FP : *Extended GCD* (because it has one of the oldest nontrivial algorithms on the books), *Sorting* (because it is one of the most frequently solved problems), and *Matrix rank* (because it is most unusual in that it seems to require a multicall checker with *two-sided* error).

6.1 Extended GCD

The problem of integer GCD is, given two integers a and b find the gcd d of a and b. Adleman, Huang and Kompella[1] have recently given a probabilistic checker for the problem. An extension of the problem makes it easy to check. The idea of extending a problem (without incurring additional running time to solve the extended problem) is an important one in the area of program checking.

Extended GCD .

Input: Two integers a and b.

Output: An integer $d = gcd(a, b)$ and integers u, v such that $d = u \cdot a + b \cdot v$.

To check that d is the gcd, the checker has to perform only 5 arithmetic operations!

- Check that d divides a and b. The validity of this check is obvious from the definition of the gcd. At this point we are convinced that d is a divisor of a and b.
- Check that $d = u \cdot a + v \cdot b$. This is done with three arithmetic operations. To justify this check and show why these two checks should convince us that d is the gcd, we refer to the following (standard) lemma.

Lemma 6.1 *Let a and b be positive integers. Then the smallest positive integer that can be expressed as an integer combination of a and b is their gcd, d.*

6.2 Sorting

It is hardly necessary to mention that sorting is one of the most commonly solved problems in computer science. Because of this a large number of algorithms are available for sorting, some of

which are fairly complex to program. Thus it is necessary to check the output of these sorting programs.

Sorting is trivially checked in the comparison tree model. In this model, the inputs are the variables $x_1, x_2, \ldots, x_n$ while the output is given by an ordering of the input variables: For some permutation σ of $[1, \ldots, n]$, the output is $x_{\sigma(1)} \leq x_{\sigma(2)} \leq \cdots \leq x_{\sigma(n)}$. The checker for sorting has only to confirm that the output inequalities are all valid. This can be done using $n-1$ comparisons, in fact, using a linear number of operations in any reasonable model of computation. In general if we assume that the outputs point to the inputs that they came from, we can check sorting merely by checking that the outputs are in the right order. In the RAM model of computation it is again easy to check sorting in linear time. But the RAM does not reflect many sorting scenarios. We define the problem of sorting and provide a reasonable model of computation.

Sorting .

Input: An array of integers $X = [x_1, \ldots, x_n]$ representing a multiset.

Output: An array Y consisting of the elements of X listed in non-decreasing order.

Model of Computation: The computer has a fixed number of tapes, including one that contains X and another that contains Y. X and Y each have at most n elements and each element is in the range $[0,a]$. The random access memory has $O(\log n + \log a)$ words of memory and each of its words is capable of holding an integer in the range $[0,a]$; in particular, each word can hold any element of $X \cup Y$.

- Single precision operations: $+, -, \times, /, <, =$ each take one step. Here $/$ denotes integer divide.
- Multi-precision operations: $+, -, <, =$ take m steps on integers that are m words long. On

such integers, $\times, /$ take m^2 steps.

In addition the machine can do the usual operations. Each shift of the tape and each copy of a word on tape to the RAM or vice versa takes 1 step.

In the model of computation described above it is easy to check that the output list Y is in order in $O(n)$ steps. We need also to check that $X = Y$ as multisets. This can be done probabilistically in $O(n)$ steps, but the right method depends on the relative sizes of a and n. If $n > 2^a$ a simple bucket sort works. We need a buckets for numbers in the range $[0, a]$. Since $a < \log n$ the random access memory has space enough for a buckets. Thus we could run through the values in X and put each one in its appropriate bucket. We could then run through the elements in Y and take each one out of its appropriate bucket. If at any time the bucket we try to take a value out of turns out to be empty, the checker declares the program to be *buggy*.

The situation when $n < 2^a$ is more interesting. For this case we present the following two methods for checking multiset equality.

Method 1: This method (but not the specific and important choice of hash function) was first suggested by Wegman and Carter[39]: Compute $n = |X|$ and check that $|Y| = n$. If so, select a hash function $h : Z \to \{0, 1\}$ and compare $h(x_1) + \cdots + h(x_n)$ to $h(y_1) + \cdots + h(y_n)$. If h is random and $X \neq Y$ then with probability at least $1/2$ the above sums will differ. To see this remove from X and Y any largest sub-multiset of elements that is common to both. The resulting X and Y are still the same size and their intersection is empty. Compute $\sum_{i \neq 1} h(x_i)$ and $\sum_i h(y_i)$. If the two sums are equal, then setting $h(x_1)$ to 1 will distinguish X from Y. If the sums are different, setting $h(x_1)$ to 0 will distinguish the two. In either case h has a probability of $1/2$ of distinguishing between the two sets. Since a random function requires an enormous number of random bits we have to replace the random function h above by a suitably chosen hash function.

Choosing an easy to compute hash function is difficult. The Wegman-Carter hash function in particular requires a random access memory and hence it cannot be implemented in our model of computation. Here is a different hash function that does work. Recall that $n = |X| = |Y|$. Let $m = n+1$. Select a random prime p from the interval $[1, 3 \cdot a \cdot \log m]$. Set $h(x) = m^x \bmod p$. Observe that $X = Y$ if and only if $\sum m^{x_i} = \sum m^{y_i}$. Indeed if $X = Y$, then $\sum(m^{x_i}) \bmod p = \sum(m^{y_i}) \bmod p$ for all primes p. If $X \neq Y$ then $\sum m^{x_i} \neq \sum m^{y_i}$ and as pointed out by Karp and Rabin[25] $\sum m^{x_i} \bmod p \neq \sum m^{y_i} \bmod p$ for at least half of all primes in the interval $[1, 3 \cdot a \cdot \log m]$. The choice of the interval size arises from an estimate of how large $\sum m^{x_i}$ can get. Since the sum is over n terms and each term is bounded by m^a, the sum is no bigger than $n \cdot m^a$. Since $m = n + 1$, a bound for the sum is m^{a+1}. The interval has then to be chosen to be a suitable constant times $\log m^{a+1}$. Thus [25] shows that for primes randomly chosen in the interval $[1, 3 \cdot a \cdot \log m]$ the hash function has a probability of at least $1/2$ of catching an error.

Method 2: This idea was first suggested by Lipton[29] and more recently by Ravi Kannan[23]. Let $f(z) = (z - x_1)(z - x_2) \cdots (z - x_n)$ and $g(z) = (z - y_1)(z - y_2) \cdots (z - y_n)$. Then $X = Y$ as multisets iff $f = g$. Since f and g are polynomials of degree n, either $f(z) = g(z)$ for all z or $f(z) = g(z)$ for at most $n - 1$ values of z. A probabilistic algorithm can decide if $f = g$ by selecting k values at random from a set of $2n$ (or more) possibilities, say from [1,2n], then comparing $f(z)$ to $g(z)$ for these k values. The computations can be kept to a reasonable size by doing the arithmetic operations modulo randomly chosen small primes. In the computation of the product $f(z)$ each term is bounded in absolute value by $a + 2n$ and hence the product is bounded by $(a + 2n)^n$. Thus according to [25] the primes have to be chosen approximately in the range $[1, n \log(a + 2n)]$.

We will now compare the two methods and show that regardless of the relative values of a and n, one method will always run in time $o(n \log n)$.

Comparison of methods 1 and 2: Recall that each multiset has at most n integers, each in the range [0,a]. Also recall that if $n > 2^a$ bucket sort can be used to check the computation. Since word sizes in our model are $O(\log a)$, if $n \leq 2^a$ the primes in method 1 fit in a constant number of words. The number of words, w, required to hold a prime in method 2 is $O(\max(1, \frac{\log n}{\log a}))$.

The running time of method 1 is $O(n \log a)$. We need to perform $\log a$ multiplications to compute m^{x_i} for each i. These are all constant time operations since the prime moduli are just a constant number of words long. The running time of method 2 is a function of the number of words, w, and is equal to nw^2 since n multiplications are performed on numbers which are w words long. This is the overriding cost of method 2.

We now describe the transition from one method to another as a decreasing function of n. When $n > 2^a$ we use bucket sort. When n becomes less than 2^a and as long as $n \log a$ is $o(n \log n)$ we use method 1 which has running time $O(n \log a)$. For instance we could use method 1 as long as n is greater than $a^{\log \log a}$. At this threshold value of n, $\log n$ is $\log a \log \log \log a$ and hence $\log a$ is $o(\log n)$. When n dips below this threshold, the primes in method 2 fit in $\log \log a$ words and method 2 runs in time $O(n(\log \log a)^2)$ Notice that in the most typical case for sorting, $n < a$, and for this case method 2 runs in linear time. Thus all the algebraic finagling is mainly to prove the existence of a little oh checker for all relative values of n and a.

6.3 Checking Matrix Rank

In this subsection we describe a checker for matrix rank. Our checker for rank is mainly of theoretical interest. It satisfies the little oh property as required. However it makes $O(n^2)$ calls to the program being checked and hence would be highly inefficient to implement in practice. Blum, Luby, and Rubinfeld[7] have subsequently discovered a very practical checker for matrix rank. However, their

checker does not conform to the original definition of checking. Instead, they use the idea that if a program for matrix multiplication has been checked, then in checking rank, one can call the matrix multiplication program and count the call as one step.

We consider matrices, M, whose entries are drawn from some finite field, F. Let P be a program which takes such a matrix as input and outputs an integer, r, which is supposedly the rank of M. We will describe here a checker for P. The checker is given an integer k in unary. k is the desired confidence in the checker's output, i.e., the probability of the checker's being wrong should be at most $O(1/2^k)$.

We describe the checker in three parts. The first part of the checker produces an r x r submatrix of M which is supposedly of full rank. It does this by a process of self-reduction, using the program to obtain intermediate answers. Part 2 of the checker checks that the resulting r x r matrix is indeed of full rank. This incidentally proves that the rank of the original matrix M is at least r. Finally, we also need to ensure that the rank of M is no more than r and this is done by Part 3 of the checker.

6.3.1 Self-Reduction

Let M be an n x m matrix input to P and suppose P on M outputs r. Let $u_1, u_2, ..., u_m$ be the columns of M.

for $i := 1$ to m **do**

delete u_i from M and feed the resulting matrix to P

if P says (the rank is) $< r$ put u_i back

endfor

if the number of columns remaining $\neq r$ return

'BUGGY'

By self-reduction, we have obtained r column vectors which are supposedly linearly independent. Each of these r columns is an n-vector and by self-reduction on the rows of the n x r matrix that we have, we arrive at an r x r matrix which is supposedly of full rank. Of course, we do not want to take the program's word that this matrix is of full rank. We need to check that this matrix is actually of full rank. Thus, even if the program returned some wrong answers in the course of this self-reduction, we will detect this and declare that the program is bad. This is done in Part 2 of the checker.

6.3.2 Lower Bounding the Rank

If we have an r x r matrix of full rank, the columns of the matrix form a basis for F^r. In this case, every vector in F^r has a *unique* representation as a linear combination of the column vectors of the matrix. In this part of the checker, we exploit the uniqueness of the representation.

Let $v_1, ..., v_r$ be the columns of the r x r matrix which is supposedly of full rank. The idea is to create k linear combinations, $x_1, x_2, ..., x_k$ of the r columns of the matrix. Suppose, for example, that

$$x_1 = c_1 v_1 + c_2 v_2 + \ldots + c_r v_r.$$

We toss a fair coin. If it comes up heads we subtract $c_1 v_1$ from x_1. Otherwise, we choose a random $a \neq c_1$ in F and subtract $a v_1$ from x_1.

We expect that $x_1 - c_1 v_1$ cannot replace v_1 in the basis, but $x_1 - a v_1$ can if $a \neq c_1$. This is clearly true if the v_i form a basis. Suppose now that the v_i do not form a basis. Then, let v_j be the first of the v_i's that has a non-zero coefficient in a dependence relation among the v_i's. v_j could have any

coefficient at all in a linear combination to produce x_1 and this coefficient of v_j is not affected by the values of coefficients for v_1 through v_{j-1}. Thus the program has no way of distinguishing the situation when we subtract c_j times v_j, from the situation when we subtract some other multiple of v_j. Thus, for each linear combination x_i the program has only a probability of 1/2 of escaping undetected if it is wrong about its claim that $v_1, v_2, ..., v_r$ are independent. The above ideas yield the required algorithm which is described below:

Generate k random linear combinations of $v_1, ..., v_r$.

Let these k random combinations be $x_1, ..., x_k$.

for i :=1 **to** k **do**

for j :=1 **to** r **do**

begin

Toss a fair coin;

if Heads **then**

$y := x_i - c_j v_j$

else

$y := x_i - a v_j$ where a is random $\neq c_j$

Replace v_j by y in the original matrix

and ask the program for the rank of this new matrix.

if Heads and (rank $\neq r-1$) **return** 'Program is bad'.

if Tails and (rank$\neq r$) **return** 'Program is bad'.

endfor

endfor

It is clear that a program that wrongly claims that $v_1, v_2, ..., v_r$ are independent has at most a probability of $1/2^k$ of escaping detection.

6.3.3 Upper Bounding the Rank

We go back to the original matrix M with columns $u_1, u_2, ..., u_m$. By self-reduction we are left with r columns, say, $u_1, \ldots, u_r$ which are supposedly linearly independent in n-dimensional space. We randomly pick vectors $x_{r+1}, \ldots, x_n$ such that the vectors $u_1, \ldots, x_n$ form a basis for the n dimensional space. We use the program's help in deciding if the set of n columns we have, are of full rank. If the program says they are not, we redo the experiment of picking vectors, $x_{r+1}, \ldots, x_n$. We have the following lemma:

Lemma 6.2 *If $a_1, \ldots, a_r$ are independent, then with probability greater than a positive constant ($\frac{1}{2} \cdot \frac{3}{4} \cdot \frac{7}{8} \cdots = .28 \cdots$), the n vectors obtained by augmenting $a_1, \ldots, a_r$ with random vectors $b_{r+1}, \ldots, b_n$ form a basis for F^n.*

Proof: The worst-case occurs when F is $GF(2)$ and $r = 0$ i.e., when we are required to build up the random basis from scratch. In this case the number of good choices for the i^{th} vector (out of a total of 2^n choices) is $(2^n - 2^{i-1})$. This works out to a probability of $(1 - \frac{1}{2^{n+1-i}})$ for the i^{th} vector to be independent of the first $i - 1$. This yields the result in the lemma. ∎

It is clear from the above lemma that each random trial has a constant probability of succeeding, i.e., producing a basis. If we perform this experiment $O(k)$ times and the program always says that the set of vectors is dependent, we report that the program is buggy. We know that we will be

correct in doing so with overwhelming probability. There is however, a small chance ($< 1/2^k$), that the program is right but we were unlucky enough not to hit upon any basis.

Next, we need the following lemma.

Lemma 6.3 *If $u_{r+1}, \ldots, u_n$ are dependent on $u_1, \ldots, u_r$ then any linear combination of $u_{r+1}, \ldots, u_n$ is dependent on $u_1, \ldots, u_r$. If one of $u_{r+1}, \ldots, u_n$ is not dependent on $u_1, \ldots, u_r$ then a random linear combination of $u_{r+1}, \ldots, u_n$ is dependent on $u_1, \ldots, u_r$ with probability at most a half.*

Proof: The first statement of the lemma is obvious. For the second part, suppose that u_{r+j} is independent of $u_1, \ldots, u_r$. Then if some linear combination x is dependent, changing the coefficient of u_{r+j} to anything else besides the one in x will make the new vector independent. This counting establishes that there are at least as many independent combinations as dependent ones, equality occurring in the case of a vector space over GF(2). ∎

By lemma 6.3 it suffices to check that k random linear combinations,$y_1, \ldots, y_k$, of $u_{r+1}, \ldots, u_n$, are dependent on $u_1, \ldots, u_r$. This will ensure that with probability $\geq 1 - 1/2^k$ the program is correct.

Suppose one of $y_1, \ldots, y_k$, say y_1, is independent of $u_1, \ldots, u_r$. We will denote y_1 by y in what follows. Let the unique expression of y as a linear combination of $u_1, \ldots, u_r, x_{r+1}, \ldots, x_n$ be

$$y = \sum_{i=1}^{r} c_i \cdot u_i + \sum_{i=r+1}^{n} c_i \cdot x_i$$

Lemma 6.4 *In the above representation of y each c_i for $r+1 \leq i \leq n$ has a probability[1] $\geq \frac{1}{2}$ of being nonzero.*

[1] This probability is over the choice of the random extension of the basis, $x_{r+1}, \ldots, x_n$. Although the program has some influence over the distribution of these random extensions, the statement of the lemma still holds.

Proof: Let V_i be the vector space generated by the first i vectors in the basis. Let W be the complement of V_r. By taking appropriate components of vectors in W our problem can be restated as follows: Suppose we have a random basis $z_1, \ldots, z_l$ for W, an l-dimensional space, and a non-zero vector y in W. For each basis vector its coefficient in the unique representation of y as a linear combination of the basis vectors will be non-zero with probability $\geq \frac{1}{2}$.

We now prove the above statement. Having a fixed vector with respect to a random basis can be thought of as equivalent to having a random vector with respect to a random basis. For, let A be a random non-singular l x l matrix. Consider the 1-1 correspondence from the set of bases to the set of bases that takes the basis, $z_1, \ldots, z_l$ to the basis, $Az_1, \ldots, Az_l$. Let $y' = Ay$ be the image of y under the linear transformation A. By the non-singularity of A, y' is a random vector in W and the new basis is a random basis of W because of the 1-1 correspondence above.

Now given a basis, a random vector in W is generated by randomly picking coefficients for the basis vectors. Thus, for a random vector, the probability that any coefficient is zero is $\leq \frac{1}{2}$. This result can be translated back to the fixed vector y. ∎

As a result of lemmata 6.3 and 6.4 we note that if $u_{r+1}, \ldots, u_n$ are not all dependent on $u_1, \ldots, u_r$, then with very high probability, one of $y_1, \ldots, y_k$ can replace one of $x_{r+1}, \ldots, x_{r+k}$ in the basis, $u_1, \ldots, u_r, x_r + 1, \ldots, x_n$. This idea yields the following checker.

for $i := 1$ **to** k

for $j := k + 1$ **to** $k + r$

repeat k **times**

Toss a fair coin;

if Heads **then**

w := a random linear combination of the original basis

with a non-zero coefficient for x_j

else

w := a random linear combination of the

original basis, without x_j

and with y_i having a non-zero coefficient.

Replace x_j by w and feed the resulting matrix to P;

if Heads and rank $\neq r$ reject program;

if Tails and rank $\neq r-1$ reject program;

endrepeat

endfor

endfor

It is clear that if the program was wrong in its original claim that $u_{r+1}, \ldots, u_m$ were dependent it can escape detection with probability at most $1/2^k$. Thus the checker has an error probability of $O(1/2^k)$ in a number of places. The overall probability of error is bounded by the sum of these probabilities and is therefore $O(1/2^k)$.

6.3.4 Analysis of Running Time

The most expensive operation is the creation of random linear combinations of many vectors. Care has been taken here to keep the number of such operations down. Part 1 of the checker, the self-reduction, runs in $O(n)$ time. In part 2, generating k linear combinations takes time $O(kn^2)$. The

loop is repeated $O(nk)$ times and each run of the loop takes time $O(n)$. Thus the overall running time of part 2 is $O(kn^2)$. In part 3, generating the random basis takes $O(kn^2)$ time since we might generate kn different vectors before we finish. Creating k linear combinations again takes $O(kn^2)$ time. The bottleneck however is the loop which is repeated $O(k^3)$ times, each pass taking $O(n^2)$ time. Thus the overall complexity of the checker is $O(k^3n^2)$.

A point of discussion is the amount of time charged to each call of the program. The above analysis has been made with each call being charged 1 step. This can be justified at least in the theoretical sense as follows: We assume a model in which the checker has a query tape to write down instances on which the program is run. Each call to the program could justifiably be charged the amount of time it takes to modify the query tape in order to produce the new instance from the previous instance queried. It is then possible to use suitable data structures to implement such modifications in $O(1)$ steps in all of the above computation.

7 Checker Characterization Theorem

In this section we characterize the set of problems that can be checked in polynomial time. For the purposes of this section a checker running in polynomial time will be called efficient.

We take as our definition of IP (Interactive Proof-System) the definition appearing in Goldwasser, Micali, and Rackoff [19], except that we replace 'for all sufficiently large x' in that definition by 'for all x'. This modification of [19] conforms with the commonly accepted definition of IP as it appears, for example, in Goldwasser and Sipser [20], and Tompa and Woll [37].

Define *function-restricted* IP (CO-*function-restricted* IP) = the set of all decision problems, π, for which there is an interactive proof system for YES-instances (NO-instances) of π satisfying

the conditions that prover (= any honest prover) must compute the function π and $\overline{\text{prover}}$ (= any dishonest prover) must be a function from the set of instances of π to {YES, NO}. This restriction implies two things:

1. verifier may only ask questions that are instances of π, and

2. $\overline{\text{prover}}$ (and prover) must answer each of verifier's questions with an answer that is independent of $\overline{\text{prover}}$'s (prover's) previous history of questions and answers.

Theorem 7.1 *An efficient program checker C_π exists for decision problem $\pi \Leftrightarrow \pi$ lies in* function-restricted *IP* $\cap$ *CO*-function-restricted *IP.*

The proof of the above theorem is immediate from the definitions of efficient program checkers and the complexity class *function-restricted* IP.

Let *NP-search* denote the class of problems π such that $\pi(x) = NO$ if x is a NO-instance; *YES* together with a *proof* that x is a YES-instance, otherwise.

Corollary 7.1 *Let π be an NP-search problem. An efficient program checker C_π exists for $\pi \Leftrightarrow \pi$ is in function-restricted co-IP.*

The main purpose of the above corollary is to point out that if $NP \not\subseteq$ CO-*function-restricted* IP, as seems likely, then there can be no efficient program checker C_π (in the above sense) for *NP*-complete problems! Note that the results of Lund *et al.* [30] and Shamir[36] do not give *function-restricted* IP proofs for NP-complete languages.

8 Overview and Conclusions

The thrust of this paper is to show that in many cases, it is possible to check a program's output on a given input, thereby giving *quantitative mathematical evidence* that the program works correctly on that input. By allowing the possibility of an incorrect answer (just as one would if computations were done by hand), the program designer confronts the possibility of a bug and considers what to do if the answer is wrong. This gives an alternative to proving a program correct that may be achievable and sufficient for many situations.

One way to develop this theory would be to require that the program checker itself be proved correct. This paper, however, is about *pure* checking, meaning no proofs of correctness whatsoever. Instead, we require the checker C to be different from the program P that it checks in two ways: First, the input-output specifications for C are different from those for P (C gets P's output and it responds $CORRECT$ or $BUGGY$). Second, we demand that the running time of the checker be $o(S)$, where S is the running time of the program being checked. This prevents a programmer from undercutting this approach, which he could otherwise do by simply running his program a second time and calling that a check. Whatever else the programmer does, he *must* think more about his problem.

9 Acknowledgements

We are grateful to Ronitt Rubinfeld for many long conversations and marvelous ideas, including her extension of our checking ideas to parallel computation. She, Sandy Irani, and Raimund Seidel have designed interesting checkers for various problems in computational geometry such as convex hull. We wish to thank them as well as Russell Imapagliazzo, Shafi Goldwasser, and Len Adleman

for their ideas and enthusiastic support.

References

[1] L. Adleman, M. Huang, and K. Kompella. Efficient Checkers for Number-Theoretic Problems. Submitted to *Information and Computation*.

[2] L. Babai and S. Moran. Arthur-Merlin Games: A Randomized Proof System, and a Hierarchy of Complexity Classes. *J. Comput. System Sci.* **36** (1988), 254–276.

[3] J. Barwise. Mathematical Proofs of Computer System Correctness. *Notices of the AMS,* vol. 36, number 7 (1989).

[4] D. Beaver and J. Feigenbaum. Hiding Instances in Multioracle Queries. In *Proceedings of the 7th Symposium on Theoretical Aspects of Computer Science.* Lecture Notes in Computer Science, vol. 415, Springer, Berlin (1990), 37–48.

[5] R. Beigel and J. Feigenbaum. On Being Incoherent Without Being Very Hard. *Computational Complexity* **2** (1992), 1–17.

[6] M. Blum and S. Kannan. Designing Programs That Check Their Work. In *Proceedings of the 21st ACM Symposium on Theory of Computing* (Seattle, Wash. May 15–17). ACM, New York, (1989), 86–97.

[7] M. Blum, M. Luby, and R. Rubinfeld. Self-Testing and Self-Correcting Programs with Applications to Numerical Problems. In *Proceedings of the 22nd Symposium on Theory of Computing* (Baltimore, MD. May 14–16). ACM New York (1990), 73–83. Final version to appear in *J. Comput Syst. Sci.*

[8] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the Correctness of Memories. *Proceedings of the 32nd Symposium on Foundations of Computing* (San Juan, PR Oct. 1–4). IEEE Computer Society, Los Alamitos (1991), 90–99. Final version to appear in *Algorithmica.*

[9] R.S. Boyer and J.S. Moore. *The Correctness Problem in Computer Science.* Academic Press, London (1981).

[10] T.A. Budd and D. Angluin. Two notions of Correctness and Their Relation to Testing. *Acta Informatica*, **18** (1982) 31–45.

[11] S.A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, **64** (1985), 2–22.

[12] R.A. DeMillo, W.M. McCracken, R.J. Martin, and J.F. Passafiume. *Software Testing and Evaluation.* The Benjamin Cummings Publishing Company, Redwood City (1987).

[13] R.A. De Millo, R.J. Lipton, and A.J. Perlis. Social Processes and Proofs of Theorems and Programs. *Comm. ACM*, **22** No. 5 (1979).

[14] J. Feigenbaum. Locally Random Reductions in Interactive Complexity Theory. In *Advances in Computational Complexity, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 13, AMS, Providence (1993), 73–98.

[15] J. Feigenbaum and L. Fortnow. Random-Self-Reducibility of Complete Sets. *SIAM J. Comput.*, **22** (1993), 994–1005.

[16] R. Freivalds. Fast Probabilistic Algorithms. In *Springer Verlag Lecture Notes in CS #74, Mathematical Foundations of CS* (1979), 57–69.

[17] M. Furst, J.E. Hopcroft, E. Luks. Polynomial-Time Algorithms for Permutation Groups. In *Proceedings 21st IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos (1980), 36–41.

[18] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing But Their Validity or All Languages in NP Have Zero-Knowldege Proof Systems. *J. ACM*, **38** (1991), 691–729.

[19] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, **18** (1989), 186–208.

[20] S. Goldwasser and M. Sipser. Public Coins vs. Private Coins in Interactive Proof Systems. In *Advances in Computing Research — vol. 5: Randomness and Computation*, JAI Press, Greenwich (1989), 73–90.

[21] D. Gorenstein. *Finite Simple Groups — An Introduction to Their Classification*, Plenum Press, New York (1982).

[22] C.M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism,* Vol. 136 of the series, Lecture Notes in Computer Science, ed. G. Goos and J. Hartmanis, Springer-Verlag, Berlin (1982).

[23] R. Kannan. personal communication through S. Rudich.

[24] S. Kannan and A.C. Yao. Program Checkers for Probability Generation. *Proceedings International Colloquium on Automata, Languages and Programming*, (Madrid, Spain, Jul. 8–12), Springer-Verlag, Berlin (1991), 163–173.

[25] R.M. Karp and M.O. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, 31(2) (1987), 249–260.

[26] R.M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science*, vol. A: Algorithms and Complexity, Elsevier, Amsterdam (1990), 869–941.

[27] Ker-I Ko. On Helping by Robust Oracle Machines. *TCS*, 52 (1987), 15–36.

[28] J.S. Leon. Computing Automorphism Groups of Combinatorial Objects. In *Computational Group Theory* ed. M.D. Atkinson, Academic Press, London (1984), 321–335.

[29] R. Lipton. New Directions in Testing. In *Distributed Computing and Cryptography, DIMACS series in Discrete Mathematics and Theoretical Computer Science*, vol. 2, AMS, Providence (1991), 191–202.

[30] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *J. ACM*, 39 (1992), 859–868.

[31] R.A. Mathon. A Note on the Graph Isomorphism Counting Problem. *IPL* 8 (1979) 131–132.

[32] G. Polya and R.C. Read. *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds,* Springer-Verlag, Berlin (1987).

[33] M.O. Rabin. Probabilistic Algorithms. In *Algorithms and Complexity, Recent Results and New Directions*, ed. J.F. Traub, Academic Press (1976), 21–40.

[34] R. Rubinfeld. Designing Checkers for Programs that Run in Parallel. Tech. Report TR-090-040, International Computer Science Institute, Berkeley, August, 1990.

[35] U. Schöning. Robust Algorithms: A Different Approach to Oracles. *TCS*, 40 (1985), 57–66.

[36] A. Shamir. IP = PSPACE. *J. ACM*, 39 (1992) 869–877.

[37] M. Tompa and H. Woll. Random Self-Reducibility and Zero Knowledge Interactive Proofs of Possession of Information. In *Proceedings 28th IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, Los Alamitos, 1987, 472–482.

[38] W.J. Weyuker, *The Evaluation of Program-Based Software Test Data Adequacy Criteria*, Communications of the ACM, **31** 6, 668–675 (1988).

[39] M.N. Wegman and J.L. Carter, *New Hash Functions and Their Use in Authentication and Set Equality*, J. of Computer and System Science, **22** 3, 265–279 (1981).