Greatest Common Divisors of Polynomials Given by Straight-Line Programs

ERICH KALTOFEN

Rensselaer Polytechnic Institute, Troy, New York,
and
Mathematical Sciences Research Institute, Berkeley, California

Abstract. Algorithms on multivariate polynomials represented by straight-line programs are developed. First, it is shown that most algebraic algorithms can be probabilistically applied to data that are given by a straight-line computation. Testing such rational numeric data for zero, for instance, is facilitated by random evaluations modulo random prime numbers. Then, auxiliary algorithms that determine the coefficients of a multivariate polynomial in a single variable are constructed. The first main result is an algorithm that produces the greatest common divisor of the input polynomials, all in straight-line representation. The second result shows how to find a straight-line program for the reduced numerator and denominator from one for the corresponding rational function. Both the algorithm for that construction and the greatest common divisor algorithm are in random polynomial time for the usual coefficient fields and output a straight-line program, which with controllably high probability correctly determines the requested answer. The running times are polynomial functions in the binary input size, the input degrees as unary numbers, and the logarithm of the inverse of the failure probability. The algorithm for straight-line programs for the numerators and denominators of rational functions implies that every degree-bounded rational function can be computed fast in parallel, that is, in polynomial size and polylogarithmic depth.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*unbounded action devices*; I.1.1 [**Algebraic Manipulation**]: Expressions and Their Representation—*representations (general and polynomial)*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*algebraic algorithms*

General Terms: Algorithms, Design, Theory, Verification

Additional Key Words and Phrases: Greatest common divisor, intermediate expression swell, multivariate polynomial, parallelization of rational functions, program transformation, randomized algorithm, separation of numerator and denominator in rational functions, straight-line program

1. *Introduction*

This study is concerned with complexity questions about performing operations, such as greatest common divisor (GCD) computation and factorization, on multivariate polynomials. Several models for representing multivariate

This material is based upon work supported by the National Science Foundation under grant DCR 85-04391 and by an IBM faculty development award. This is a completely revised version of a paper that appared in the *Proceedings of the 17th Annual ACM Symposium on Theory of Computing.* ACM, New York, 1985, pp. 131–142. The main results in Section 8 first appeared in the *Proceedings of the 18th Annual ACM Symposium on Theory of Computing.* ACM, New York, 1986, pp. 330–337. Part of the work on Section 8 was done while the author was visiting Tektronix Computer Research Laboratory in Beaverton, Oregon.

Author's address: Deparment of Computer Science, Rensselaer Polytechnic Institute, Troy, N.Y. 12180-3590; kaltofen@cs.rpi.edu (arpanet)

polynomials have been suggested:

—the *dense representation*, which requires that all coefficients be written down;
—the *sparse representation*, which requires that all nonzero coefficients and the corresponding monomial exponent vectors be written down;
—*formulas*, denoted by expressions similar to those from higher programming languages;
—*straight-line programs*, such as the Gaussian elimination sequence on a determinant of polynomials.

We perceive these representations from a macroscopic point of view, which is that polynomial time or space differences within each class are not so important to us as the exponential differences between the classes. We remark, however, that the microscopic point of view is important to consider for pragmatic reasons (see, e.g., Stoutemyer's [41] comparison of different sparse representations). One easily realizes that the four models form a hierarchy; that is, within a polynomial extent of space we can represent more and more polynomials going from the dense to the straight-line representation. Therefore, the latter is (from a macroscopic point of view) the most powerful one. This is nicely illustrated by the famous example of symbolic determinants that have exponentially many terms when converted to "sparse" representation, but that can be represented by straight-line programs of length proportional to at most the cube of the dimension.

The question is, of course, whether polynomials given by straight-line programs, a notion that we shall make precise, can be manipulated at all. Arithmetic operations trivially become additional assignments and the first problem of interest is the GCD. Since even for univariate sparse polynomials this operation is NP-hard [35], a restriction necessarily has to be made. One natural additional parameter to bound polynomially, other than representation size, is the total degree of the input polynomial. Valiant [45] calls such families of polynomials of polynomially bounded degree and straight-line computation length *p-computable*. Several important transformations on p-computable polynomials are already known, for example, Strassen's elimination of divisions [43] or the parallelization technique by Valiant et al. [46]. Another such transformation by Baur and Strassen [3] allows the computation of all first partial derivatives with growth in length by only a constant factor and without even the need for a degree bound. There are also known negative results in [45] that show, for example, that general permanents can appear as coefficients of single monomials and as multiple partial derivatives (see also Section 5). It should be noticed, however, that all these algorithms are interpreted as program transformations and not as polynomial manipulation routines. Von zur Gathen [11] obtained a probabilistic algorithm that determines the factor degree pattern, that is, the total degrees and multiplicities of the factors, of polynomials given by straight-line programs. Schwartz's [37] evaluation technique at random points and modulo large random pseudoprimes (see also [18] and Section 3) play an important role in that and our new results. In the context of representing polynomials by straight-line programs, Heintz's result [15] also deserves to be mentioned.

The theory for polynomial manipulation on straight-line representation should deal with the computation of straight-line results whenever possible, for example, produce a straight-line program for the GCD of two determinants of polynomial matrices. In Valiant's language, it is the question of closure properties of p-computable families of polynomials. It is this theory we begin to develop here. In this paper we show how a straight-line program for the GCD of many polynomials can be constructed in random polynomial time from their straight-line representations, as well as a bound for their total degrees (Section 6). This probabilistic result is of the Monte-Carlo kind, which means that the algorithm always takes polynomial time, but may with controllably small probability return an incorrect answer. Our algorithm is polynomial time even for coefficient domains such as rational numbers. This is surprising because the coefficients of the input polynomials can be exponentially large. In general, the growth of the exponential size of the intermediately calculated rational numbers during most algorithms, such as Strassen's elimination of divisions, causes additional complications.

For various reasons, we chose to carefully develop the theory of straight-line program manipulation here (Sections 2–5) before discussing the concrete applications. For one reason, the running time of our algorithms must be estimated. One can consider polynomial-time complexity bounds as our *uniformity* requirement for the straight-line program transformations. No such requirement is needed or enforced in the lower-bound applications of the elimination of divisions transformation or the computation of derivatives. We use as our model of computation that of a *probabilistic algebraic random-access machine* (RAM), whose instruction set and binary complexity measures we define (Section 2). Since we strive to obtain random polynomial-time complexity, the execution of arithmetic operations will cost as many time units as are needed to perform these operations in binary. Under this "logarithmic cost criterion," the needed straight-line program transformations are established to be of polynomial-time complexity by the use of what we call the *simulation principle.* This principle shows that the usual RAM programs can be converted into RAMs of polynomially related binary asymptotic complexity that generate the straight-line programs corresponding to these computations (Section 4).

The GCD problem for dense multivariate polynomials was first made feasible by the work of Collins [8] and Brown [5]. Moses and Yun [34] showed how to apply the Hensel lemma to GCD computations. Zippel [47] invented an important technique to preserve sparsity of the multivariate GCD during Brown's interpolation scheme, though it should be noted that Zippel's approach is not random polynomial time. The reason is that the content and primitive part of the inputs can not be separated because some sparse polynomials have dense primitive parts; cf. [13] and Section 5. We also mention the heuristic GCD algorithm in [7], which may be practically a faster algorithm if the inputs have few variables.

Our algorithm for computing the GCD as a straight-line program requires several innovations. For one, we remove the need for the content computation by substituting linear forms of the main variable into minor variables. These substitutions lead with high probability to monic polynomials in the main variable and thus allow a single Euclidean sequence over the field of rational functions in the minor variables. We compute the coefficients in the main variable by determining these coefficients (now rational functions in the minor variables) for each assignment in the straight-line program. Finally, we encode a polynomial remainder sequence computation on the coefficient vectors by determining the degrees of the remainders through probabilistic evaluation of their coefficients. The GCD problem for many polynomials is reduced to that of two polynomials via a theorem stating that two random linear combinations of the set of input polynomials yield the same GCD with high probability.

We now turn to the computation of numerator and denominator of rational functions. Strassen [43] raised the question whether the reduced numerators, and therefore the denominators of rational functions, could be computed by straight-line programs of length polynomial in the length of straight-line programs for the functions themselves and the degrees of the relatively prime' numerator–denominator pairs. Here we answer this question affirmatively by showing that such straight-line programs can be also found in random polynomial time (Section 8). The construction is closely related to the straight-line GCD algorithm put together with computing Padé approximants via the extended Euclidean algorithm. Our solution requires an algorithm for finding a straight-line program for the Taylor series coefficients to a given order of the rational functions with respect to a single variable. In Section 7, we present a solution to this problem following the approach by Strassen [43], and it comes as no surprise that we obtain the result on eliminating divisions from straight-line programs for polynomials as a consequence (Theorem 7.1).

The resolution of the numerator and denominator complexity of rational functions has an important consequence in the theory of polylogarithmic parallel computations. First, we note that Hyafil [17] and Valiant et al. [46] established that families of p-computable polynomials can be evaluated in parallel in polynomial size and polylogarithmic depth. We now can apply this result to the straight-line programs for the numerators and denominators of rational functions and therefore can conclude that every family of rational functions of polynomial complexity and reduced numerator–denominator degrees can also be computed in parallel in polylogarithmic time with polynomially many processing elements (Corollary 8.3).

Notation. Let $\mathbf{Z}$ denote the integers, $\mathbf{Q}$ the rationals, and $\mathbf{F}_q$ the finite field with q elements. Let $\mathrm{QF}(D)$ denote the field of quotients of an integral domain D. Let $\mathrm{num}(a)$ denote the numerator and $\mathrm{den}(a)$ the denominator of $a \in \mathrm{QF}(D)$. The coefficient of the highest power of x_1 in $f \in (D[x_2, \ldots, x_n])[x_1]$ is referred to as the leading coefficient of f in x_1, $\mathrm{ldcf}_{x_1}(f)$.

Let $M(d)$ denote a function dominating the time for multiplying polynomials in $D[x]$ of maximum degree d. Note that $M(d)$ depends on the multiplication algorithm used, and the best known asymptotic result is $d \log d \log\log d$ [39].

The cardinality of a set R is denoted by $\mathrm{card}(R)$. All logarithms in this paper are to base 2 unless otherwise indicated.

2. *Straight-Line Programs and Algebraic RAMs*

We first present the precise definition of what we understand by (algebraic) straight-line programs.

Definition. Let D be an integral domain. The $P = (X, V, C, S)$ is an *algebraic straight-line program* over D if

(SLP1) $X = \{x_1, \ldots, x_n\} \subset D$, $S = \{s_1, \ldots, s_k\} \subset D$, $V = \{v_1, \ldots, v_l\}$, $V \cap D = \varnothing$. X denotes the set of *inputs*, V the set of *(program) variables*, S the set of scalars. If $S = \varnothing$, then P is *scalar-free*.

(SLP2) $C = (v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda)_{\lambda=1,\ldots,l}$ with $\circ_\lambda \in \{+, -, \times, \div\}$, $v'_\lambda, v''_\lambda \in S \cup X \cup \{v_1, \ldots, v_{\lambda-1}\}$ for all $\lambda = 1, \ldots, l$. C denotes the *computation sequence* and l the *length* of P, $l = \mathrm{len}(P)$. If all $\circ_\lambda \neq \div$, then P is *division free.*

(SLP3) For all $\lambda = 1, \ldots, l$, there exists $\mathrm{sem}(v_\lambda) \in D$, the *semantics* of v_λ, such that

$$\begin{array}{ll}
\mathrm{sem}(a) = a & \text{if } a \in S \cup X, \\
\mathrm{sem}(v_\lambda) = \mathrm{sem}(v'_\lambda) \pm \mathrm{sem}(v''_\lambda) & \text{if } \circ_\lambda = \pm, \\
\mathrm{sem}(v_\lambda) = \mathrm{sem}(v'_\lambda)\mathrm{sem}(v''_\lambda) & \text{if } \circ_\lambda = \times, \\
\mathrm{sem}(v''_\lambda) \neq 0 \text{ and } \mathrm{sem}(v_\lambda)\mathrm{sem}(v''_\lambda) = \mathrm{sem}(v''_\lambda) = \mathrm{sem}(v'_\lambda) & \text{if } \circ_\lambda = \div.
\end{array}$$

The *set of elements* computed by P is $\mathrm{sem}(P) = \bigcup_{\lambda=1}^{l} \{\mathrm{sem}(v_\lambda)\}$. □

We observe that the integrality of D guarantees the uniqueness of $\mathrm{sem}(v_\lambda)$. If D is a field, then the last case in axiom (SLP3) simplifies to $\mathrm{sem}(v''_\lambda) \neq 0$ if $\circ_\lambda = \div$, since then $\mathrm{sem}(v_\lambda)$ can be determined as $\mathrm{sem}(v'_\lambda)(\mathrm{sem}(v''_\lambda))^{-1}$. Our definition is more general than Strassen's [42], who always insists on the invertibility of $\mathrm{sem}(v''_\lambda)$ in case $\circ_\lambda = \div$. That our generalization is useful can be seen from the straight-line programs computing determinants and subresultants over D by exact division.

Our main application is for programs $P = (\{x_1, \ldots, x_n\}, V, C, S)$ over $D = F(x_1, \ldots, x_n)$, where $S \subset F$, F a field, and which determine certain polynomials $f \in F[x_1, \ldots, x_n]$ with $f \in \mathrm{sem}(P)$. In such a case we say f is *given by* the straight-line program P. Note that we use the notation $f \in \mathrm{sem}(P)$ with the implied understanding that we also know the $v_\lambda \in V$ with $f \in \mathrm{sem}(v_\lambda)$. However, sometimes our more general formulation is needed. One example would be to find the shortest straight-line program that computes the Newton polynomials $\sum_{\nu=1}^{n} x_\nu^i$, $i = 2, \ldots, n$, from the symmetric functions in the indeterminates. For lower bound considerations, one usually adds the condition in (SLP1) that $\{x_1, \ldots, x_n\}$ is algebraically independent over the field generated by S. For polynomials given by straight-line programs, this restriction is satisfied, but for ease in the formulation of later definitions and theorems, we do not adopt it in our main definition.

Algebraic computations over abstract domains D are usually formulated in terms of programs to be executed on an algebraic RAM over D. Let us describe this model of computation more precisely. An *algebraic RAM over D* has a CPU that is controlled by a finite sequence of labeled instructions and that has access to an infinite address and data memory (see Figure 1).

The split into two memories, one that facilitates pointer manipulation for array processing and maintains a stack for recursive procedures, and another in which the algebraic arithmetic is carried out, is also reflected in other models for algebraic computations, such as the parallel arithmetic networks in [12], or by the omnipresence of the built-in type Integer for indexing in the Scratchpad II language [20]. Each word in address memory can hold an integral address, and each word in data memory can store an element in D. The CPU also has access to an input and an output medium. The instructions in the CPU may have one or two operands that typically are integers. The operands refer to words in address or data memory, depending on whether the instruction is an address or a data instruction. Indirect addressing is indicated by a negative operand. For completeness, the microcode for a full instruction set is given in Figure 2.

The *arithmetic time* and *space complexity* of an algebraic RAM for a given input are defined as the number of instructions executed and the highest memory address referenced, respectively. It is not always realistic to charge one time unit for each arithmetic operation in D. We consider encoding data in binary and define as size(a), $a \in D$, where D is a concrete domain, such as $\mathbf{Z}$ or $\mathbf{F}_q$, the number of bits

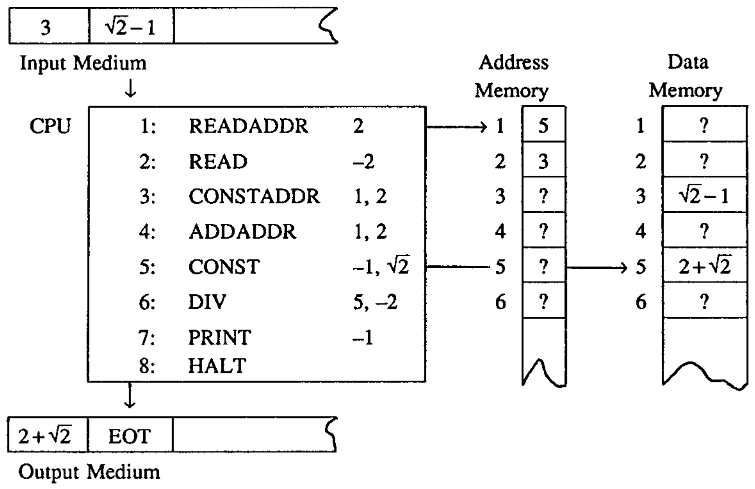

FIG. 1. Algebraic RAM over $\mathbf{Z}[\sqrt{2}]$.

Instruction		Description
ADD{ADDR}	i, j	$Op_i \leftarrow Op_i + Op_j$ (see below)
SUB{ADDR}	i, j	$Op_i \leftarrow Op_i - Op_j$.
MULT{ADDR}	i, j	$Op_i \leftarrow Op_i \times Op_j$.
DIVADDR	i, j	$Op_i \leftarrow \lfloor Op_i / Op_j \rfloor$.
DIV	i, j	$Op_i \leftarrow Op_i / Op_j$. The division over D must be exact, otherwise an interrupt occurs.
CONST{ADDR}	i, c	$Op_i \leftarrow c$.
MOVE{ADDR}	i, j	$Op_i \leftarrow Op_j$.
JMP	l	Execution continues at program label l.
JMPZ{ADDR}	i, l	If $Op_i = 0$, then execution continues at program label l.
JMPGZADDR	i, l	If $Op_i > 0$, then execution continues at program label l.
READ{ADDR}	i	The input medium is advanced and the next item is read into Op_i.
PRINT{ADDR}	i	The output medium is advanced and Op_i is written onto the medium.
HALT		An EOT marker is written onto the output tape and execution terminates.

$$Op_i = \begin{cases} \left.\begin{matrix}\mathrm{AM}[i] \\ \mathrm{DM}[i]\end{matrix}\right\} & \text{if } i > 0 \text{ and } \left\{\begin{matrix}\text{address} \\ \text{data}\end{matrix}\right\} \text{ instruction} \\ \left.\begin{matrix}\mathrm{AM}[\mathrm{AM}[-i]] \\ \mathrm{DM}[\mathrm{AM}[-i]]\end{matrix}\right\} & \text{if } i < 0 \text{ and } \left\{\begin{matrix}\text{address} \\ \text{data}\end{matrix}\right\} \text{ instruction} \end{cases}$$

AM = address memory, DM = data memory

AM$[-i]$ must be positive, otherwise an interrrupt occurs.

FIG. 2. Summary of algebraic RAM instructions.

needed to represent a. Then, the cost and space of an arithmetic instruction depends on the size of its operands. The *binary* time and space complexity of an algebraic RAM over D is derived by charging for each arithmetic step in D as many units as are needed to carry out the computation on a multitape Turing machine. Note that we generally assume that the domain arithmetic can be carried out in polynomial binary complexity with respect to the size of the operands. What that implies, in particular, is that elements in $\mathbf{F}_q$, say, always require $O(\log(q))$ representation size, whether or not the elements are residues of small integral value. For READ, PRINT, CONST, MOVE, or JMPZ instructions, we charge as many units as is the size of the transferred or tested element.

We also apply this "logarithmic cost criterion" to the address computations and assume that every address is represented as a binary integer. The binary cost for performing address arithmetic is again the Turing machine cost. For indirect addressing, we add the size of the final address to the binary time and space cost of the corresponding instruction. We note that, in most circumstances, the binary cost for performing address arithmetic is largely dominated by the binary cost of the algebraic operations and that, for all practical purposes, the largest storage location is of constant size. But our more precise measure has its advantages. First, all binary polynomial-time algorithms on algebraic RAMs are also polynomial-time algorithms in the Turing machine model. Second, the true binary complexity is measured if we can use the address memory for more than address computations, for example, for hashing with sophisticated signatures. Another such example is that of selecting random domain elements.

A *probabilistic* algebraic RAM is endowed with the additional instruction

$$\text{RANDOM}\{\text{ADDR}\}\, i, j,$$

with the following meaning. Into Op_i an element of D (or an address) is stored that was uniformly and randomly polled from a set R of elements (or integers) with card(R) equal to the address operand Op_j (see Figure 2 for the definition of Op). The selection of R is unknown, except that all its elements $a \in R$ have size(a) = $O(\log Op_j)$. This model of randomized algebraic computation overcomes the problem of how to actually generate a "random" rational number, say, and (as we show later) the failure probabilities can in our circumstances be fully analyzed.

Most of our algorithms read as input, produce as intermediate results, and print as output straight-line programs. In this paper, we do not describe a concrete data structure that can be used to represent straight-line programs on an algebraic RAM. It is fairly easy to conceive of suitable ones (e.g., labeled directed acyclic graphs (DAGs) could be used). A more intricate data structure was used for the first implementation of our algorithms and is described in [10].

At this point it is convenient to define the *element size* of a straight-line program

$$\text{el-size}(P) = \sum_{\substack{v_\lambda^* \in X \cup S,\\ * \in \{\prime, \prime\prime\}}} \text{size}(v_\lambda^*).$$

Notice that the actual size of P is in bits

$$O(\text{len}(P) \log \text{len}(P) + \text{el-size}(P)),$$

since it takes $\text{size}(v_\lambda) = O(\log(\lambda))$ bits to represent v_λ in address memory.

3. *Evaluation and Size Growth*

Classically, the inputs to a straight-line program are indeterminates that are evaluated at concrete values during execution of the program. Two problems arising with the evaluation process need to be discussed. The first is that evaluation may lead to a division by zero, which we must declare illegal. The second is that the binary complexity of evaluation can turn out to be exponential in the length of the program. In this section we address both problems. Let us formally define evaluation.

Definition. Let $P = (X, V, C, S)$ be a straight-line program of length l over D, $\bar{D}$ be another integral domain, and $\phi(a) = \bar{a}$ be a mapping from $X \cup S$ into $\bar{D}$.

We extend ϕ to $\bar{V} = \{\bar{v}_1, \ldots, \bar{v}_l\}$, $\bar{V} \cap \bar{D} = \emptyset$, by setting $\phi(v_\lambda) = \bar{v}_\lambda$, $1 \le \lambda \le l$, and define

$$\bar{X} = \{\phi(x) \mid x \in X\}, \qquad \bar{S} = \{\phi(s) \mid s \in S\}, \qquad \bar{C} = (\bar{v}_\lambda \leftarrow \phi(v'_\lambda) \circ_\lambda \phi(v''_\lambda))_{\lambda=1,\ldots,l}.$$

We call P *defined at* ϕ, if $\phi(P) = (\bar{X}, \bar{V}, \bar{C}, \bar{S})$ is a straight-line program over $\bar{D}$. □

It is clear that only condition (SLP3) of the straight-line program definition for $\phi(P)$ must be verified. If ϕ can be extended to a ring homomorphism from D into $\bar{D}$, it suffices to require $\phi(\mathrm{sem}(v''_\lambda)) \ne 0$ for $\circ_\lambda = \div$, $1 \le \lambda \le l$, because then exact division is guaranteed. But more general evaluations do occur, as in the following example:

Example. Let $D = \mathbf{Q}(x)$, $\bar{D} = \mathrm{GF}(2)$, $\phi(x) = \phi(\frac{1}{3}) = 1$, $\phi(2) = 0$, and $P = (\{x\}, \{v_1, v_2\}, (v_1 \leftarrow x + \frac{1}{3}, v_2 \leftarrow 2 \div v_1), \{2, \frac{1}{3}\})$. P is not defined at ϕ, since $\mathrm{sem}(v_1) \equiv 0 \bmod 2$. Note also that ϕ cannot be extended to a ring homomorphism from $\mathbf{Q}(x)$ into GF(2).

It is easy to see that, given the encoding of a straight-line program $P = (X, V, C, S)$ over D, we can compute $\mathrm{sem}(v_l)$ in $O(l)$ steps on an algebraic RAM over D. If we assume that $\bar{D}$ is a field or that ϕ is a ring homomorphism, we can also decide in $O(l)$ steps on an algebraic RAM over $\bar{D}$ whether the encoding of $\phi(P)$ represents a straight-line program. All we need to do is test whether $\mathrm{sem}(\phi(v''_\lambda)) \ne 0$, $\circ_\lambda = \div$, before performing the division. Controlling the binary complexity of evaluation is, however, a much more difficult matter because the straight-line programs may generate exponentially sized elements.

Example. Let $P = (\{x\}, V, C, \emptyset)$ over $\mathbf{Q}(x)$, where

$$\begin{aligned} C = (&v_1 \leftarrow x \times x,\ v_2 \leftarrow x \times v_1,\\ &v_3 \leftarrow v_1 \times v_1,\ v_4 \leftarrow v_2 \times v_3,\\ &v_5 \leftarrow v_3 \times v_3,\ v_6 \leftarrow v_4 \times v_5,\\ &\vdots\\ &v_{2l-1} \leftarrow v_{2l-3} \times v_{2l-3},\ v_{2l} \leftarrow v_{2l-2} \times v_{2l-1},\\ &v_{2l+1} \leftarrow v_{2l-1} \times v_{2l-1},\ v_{2l+2} \leftarrow v_{2l+1} \div v_{2l}). \end{aligned}$$

We remark that $\mathrm{sem}(v_{2\lambda-1}) = x^{2^\lambda}$, $\mathrm{sem}(v_{2\lambda}) = x^{2^{\lambda+1}-1}$, $1 \le \lambda \le l$, $\mathrm{sem}(v_{2l+1}) = x^{2^{l+1}}$, and $\mathrm{sem}(v_{2l+2}) = x$. The test of whether P is defined at $\phi(x) = 2$ would require on an arithmetic RAM over $\mathbf{Q}$ exponential binary running time. Notice also that the last element computed by P is again small in size.

In what follows, we combat the size blowup by a modular technique, an idea first suggested by Schwartz [37] and Ibarra and Moran [18]. A generalization of what follows to algebraic extensions of $\mathbf{Q}$ can be found in [11]. For completeness, we shall give the proof of the next lemma.

LEMMA 3.1. *Let $P = (\{x_1, \ldots, x_n\}, V, C, \{s_1, \ldots, s_m\})$ be a straight-line program of length l over $\mathbf{Q}(x_1, \ldots, x_n)$, $a_\nu \in \mathbf{Q}$, $\phi(x_\nu) = a_\nu$, $1 \le \nu \le n$, $b_\mu \in \mathbf{Q}$, $\phi(s_\mu) = b_\mu$, $1 \le \mu \le m$. Assume that P is defined at ϕ, and that $n + m \le 2l$, which is satisfied if all inputs and scalars actually occur in the computation sequence. Let $B_\phi \ge 2$ be an integer bound such that*

$$|num(a_\nu)|,\ |den(a_\nu)|,\ |num(b_\mu)|,\ |den(b_\mu)| \le B_\phi,$$
$$1 \le \nu \le n, \quad 1 \le \mu \le m.$$

Then there exists an integer $N_{\phi(P)} \leq B_\phi^{2^{l+3}}$ such that for all prime integers p that do not divide $N_{\phi(P)}$, the following is true:

(*i*) *den*(a_ν), *den*(b_μ) $\not\equiv 0 \bmod p$ *for all* $1 \leq \nu \leq n$, $1 \leq \mu \leq m$.
(*ii*) *If we define* $\psi: \{x_\nu\} \cup \{s_\mu\} \to \mathbf{F}_p$ *by* $\psi(x_\nu) \equiv a_\nu \bmod p$, $\psi(s_\mu) = b_\mu \bmod p$, $1 \leq \nu \leq n$, $1 \leq \mu \leq m$, *then P is defined at* ψ.

PROOF. Let $\phi(P) = (\{a_\nu\}, \{\bar{v}_\lambda\}, \bar{C}, \{b_\mu\})$. We must estimate

$$u_\lambda = \mathrm{num}(\mathrm{sem}(\bar{v}_\lambda)), \qquad t_\lambda = \mathrm{den}(\mathrm{sem}(\bar{v}_\lambda)), \qquad 1 \leq \lambda \leq l.$$

By induction, we can prove that

$$|u_\lambda|, |t_\lambda| \leq \frac{(2B_\phi)^{2^\lambda}}{2} \leq B_\phi^{2^{\lambda+1}}, \qquad 1 \leq \lambda \leq l. \tag{1}$$

Consider $\bar{v}_\lambda \leftarrow \phi(v'_\lambda) \circ_\lambda \phi(v''_\lambda)$. By induction hypothesis,

$$|u^*_\lambda|, |t^*_\lambda| \leq \frac{(2B_\phi)^{2^{\lambda-1}}}{2}, \qquad u^*_\lambda = \mathrm{num}(\phi(v^*_\lambda)), \quad t^*_\lambda = \mathrm{den}(\phi(v^*_\lambda)), \quad * = ', ''.$$

In case $\circ_\lambda = +$, we thus get

$$|u_\lambda| \leq |u'_\lambda t''_\lambda + u''_\lambda t'_\lambda| \leq |u'_\lambda||t''_\lambda| + |u''_\lambda||t'_\lambda| < 2\,\frac{(2B_\phi)^{2^{\lambda-1}+2^{\lambda-1}}}{4}.$$

The treatment of t_λ, and the cases $\circ_\lambda = -, \times$, and $\div$ are similar. Thus, (1) is established.

Now we observe that

$$M_\phi = \prod_{1\leq\nu\leq n} |\mathrm{den}(a_\nu)| \prod_{1\leq\mu\leq m} |\mathrm{den}(b_\mu)| \leq B_\phi^{2l}.$$

We set

$$N_{\phi(P)} = M_\phi \prod_{\lambda=1}^{l} |t_\lambda| < B_\phi^{2l} B_\phi^{2^{l+2}} < B_\phi^{2^{l+3}}.$$

Clearly, if a prime p does not divide $N_{\phi(P)}$, (i) and (ii) are satisfied. □

Although our bound for $N_{\phi(P)}$ in the above lemma is of exponential size, we can pick a suitable prime probabilistically quite efficiently. Let

$$k = 2^{l+5}\log(B_\phi) \geq 4\log(N_{\phi(P)})$$

and consider the first k primes $p_1, \ldots, p_k$. Since for each subset K of $\{1, \ldots, k\}$ of cardinality $\geq k/4$,

$$\prod_{\kappa\in K} p_\kappa > 2^{k/4} \geq 2^{\log(N_{\phi(P)})} = N_{\phi(P)},$$

we conclude that fewer than $k/4$ of the primes $p_1, \ldots, p_k$ can be divisors of $N_{\phi(P)}$. Now

$$p_k < k(\log_e k + \log_e \log_e k) < k \log k, \qquad k \geq 6$$

(cf. Rosser–Schoenfeld [36, sect. 3.13]). Therefore, if we randomly pick a prime p,

$$p < k\log k < C_{\phi(P)} = (l + 5 + \log\log(B_\phi))2^{l+5}\log(B_\phi), \tag{2}$$

with probability greater than $\frac{3}{4}$ this prime will certify that P is defined at ϕ. We have the following algorithm:

Algorithm *Zero-Division Test*

Input: A straight-line program $P = (\{x_1, \ldots, x_n\}, V, C, \{s_1, \ldots, s_m\})$ of length l over $\mathbf{Q}(x_1, \ldots, x_n)$, $a_\nu, b_\mu \in \mathbf{Q}$, $1 \le \nu \le n$, $1 \le \mu \le m$, and a failure probability $\epsilon \ll 1$.

Output: An integer p such that P is defined at $\psi(x_\nu) = a_\nu \bmod p$, $\psi(s_\mu) = b_\mu \bmod p$, or "failure". In case P is defined at ϕ, failure occurs with probability $<\epsilon$.

Step L (Loop on trials). Repeat steps P and E $\lceil \log 1/\epsilon \rceil$ times. Then return "failure".

Step P (Pick a prime). Let B_ϕ be as defined in Lemma 3.1 and set $C_{\phi(P)}$ according to (2).

```
for $i \leftarrow 1, \ldots, j = \lceil 21/10 \log C_{\phi(P)} \rceil$ do
    Select a random postive integer $p < C_{\phi(P)}$. Perform a probabilistic primality test
    on $p$, for example, Solovay and Strassen's [40], such that $p$ is either certified
    composite or is probably prime with chance $\ge 1 - 1/(8j)$. In the latter case goto
    step E.
```

At this point no prime was found, so go back to step L.

Step E (Evaluation). Evaluate $\psi(P)$ on an algebraic RAM over $\mathbf{F}_p$. If a division by zero or a zero divisor occurs, go back to step L. Otherwise return p. □

We note that the bound $C_{\phi(P)}$ is only of theoretical interest. In practice word-sized primes are already likely to certify that P is defined at ϕ. Clearly, the Zero-Division Test Algorithm runs in binary polynomial-time and requires polynomially many random-bit choices. We do not state explicit polynomial upper bounds for this or any of the subsequent algorithms, although the original version of this paper [21] contains several of them. Instead, we now refer to [10] for the actual performance of our algorithms, which would not be captured by those crude upper bounds. However, the theoretical failure probability of the Zero-Division Test Algorithm shall be analyzed in the following theorem.

THEOREM 3.1. *Algorithm Zero-Division Test requires $(l \log(B_\phi) \log(1/\epsilon))^{O(1)}$ binary steps on a probabilistic algebraic RAM over $\mathbf{Z}$. In case P is defined at ϕ, it returns "failure" with probability $<\epsilon$.*

PROOF. There are three circumstances under with steps P and E do not find the prime p even if P is defined at ϕ. First, a prime may never be picked in Step P. There are

$$\pi(C_{\phi(P)}) > \frac{7}{10} \frac{C_{\phi(P)}}{\log(C_{\phi(P)})}, \qquad C_{\phi(P)} \ge 17,$$

primes $< C_{\phi(P)}$; cf. [36, sect. 3.5]. Thus, the probability that we select a composite p in all iterations of step P is no more than

$$\left(1 - \frac{1}{10/7 \log C_{\phi(P)}}\right)^{3 \cdot (7/10) \log C_{\phi(P)}} < \frac{1}{e^3} < \frac{1}{8}.$$

Second, the chance that all composite p's selected in step P are recognized as such is certainly not less than $(1 - 1/(8j))^j > \frac{7}{8}$, because $(1 - \frac{1}{8})^{1/j} < 1 - 1/(8j)$. Thus, we pass a composite p on to step E with chance $\le\frac{1}{8}$. Third, by the previous discussion we have selected a prime with $p \mid N_{\phi(P)}$ with chance $\le\frac{1}{4}$. Therefore, the total probability of things going wrong is less than $\frac{1}{4} + \frac{1}{8} + \frac{1}{8} = \frac{1}{2}$. Now, steps P and E are repeated at least $\log 1/\epsilon$ times. Therefore, the probability of failing at all these trials is $<(\frac{1}{2})^{\log 1/\epsilon} = \epsilon$. □

We can use the Zero-Division Test Algorithm to determine whether an element in $\mathrm{sem}(\phi(P))$ is zero. Let us briefly write down the algorithm.

Algorithm *Zero Test*

Input: As in Algorithm Zero-Division Test. Furthermore, an index λ, $1 \le \lambda \le l$.

Output: Let $e_\lambda = \mathrm{sem}(\phi(v_\lambda))$. We return either "$e_\lambda$ is definitely not equal to zero" or "e_λ is probably 0". The latter happens if P is defined at ϕ and $e_\lambda \neq 0$ with probability $<\epsilon$.

Step Z (Zero Test): Run Algorithm Zero-Division Test on $P' = (\{x_1, \ldots, x_n\}, V \cup \{v_{l+1}\}, C$ concatenated with $(v_{l+1} \leftarrow 1 \div v_\lambda), S)$ and ϕ. If no failure occurs return "e_λ is definitely $\neq 0$"; otherwise, return "e_λ is probably 0". □

Another application of Algorithm Zero-Division Test is, in fact, to compute $e_\lambda = \mathrm{sem}(\phi(v_\lambda))$, $1 \le \lambda \le l$. Since e_λ can be of exponential size in l, polynomial-time complexity can only be expected if we know an a priori bound $B_\lambda \ge |\mathrm{num}(e_\lambda)|, |\mathrm{den}(e_\lambda)|$. We again explicitly present the algorithm.

Algorithm *Evaluation*

Input: As in Algorithm Zero-Division Test. Furthermore, an index λ, $1 \le \lambda \le l$, and a bound B_λ.

Output: Either "failure" (that with probability less than ϵ in case P is defined at ϕ) or $e_\lambda = \mathrm{sem}(\phi(v_\lambda))$, provided that

$$|\mathrm{num}(e_\lambda)|,\ |\mathrm{den}(e_\lambda)| \le B_\lambda.$$

Step T (Zero-Division Test). Call Algorithm Zero-Division Test. If it fails, return "failure".

Step M (Modular Image Evaluation). Let p be the integer returned by the call to the Zero-Division Test algorithm. Compute $\bar{e}_\lambda \equiv \mathrm{sem}(\phi(v_\lambda)) \bmod p^k$ where $p^k \ge 2B_\lambda^2$, $0 \le \bar{e}_\lambda < p^k$. This is possible because p must be relatively prime to all denominators computed.

Step C (Continued Fraction Recovery). Find the continued fraction approximation to $\bar{e}_\lambda/p^k$,

$$\frac{u_1}{t_1}, \ldots, \frac{u_i}{t_i}, \frac{u_{i+1}}{t_{i+1}}, \qquad t_i < B_\lambda, \quad t_{i+1} \ge B_\lambda.$$

Return $e_\lambda = (\bar{e}_\lambda t_i - p^k u_i)/t_i$. □

For $B = \max(B_\lambda, B_\phi)$, the algorithm takes $(l \log(B) \log(1/\epsilon))^{O(1)}$ binary steps. The correctness of Step C follows from the theory of continued fractions [14, chap. 10].

The idea of working modulo p^k is sometimes referred to as Hensel-code arithmetic [29]. If individual bounds for $|\operatorname{num}(e_\lambda)|$ and $|\operatorname{den}(e_\lambda)|$ are known, as they often are, then the approach is subject to improvement; cf. [26, theorem 4.1].

4. *Probabilistic Simulation*

We now turn to our uniformity considerations, that is, the complexity of performing the needed straight-line program manipulations on algebraic RAMs. We first demonstrate the issue on an example. Let

$$f(x) = y_n x^n + \cdots + y_0,$$
$$g(x) = x^m + z_{m-1}x^{m-1} + \cdots + z_0, \qquad n > m,$$

and let $D = E[y_n, \ldots, y_0, z_{m-1}, \ldots, z_0]$, where E is an abstract integral domain of characteristic $\neq 2$. We want to design an algebraic RAM over D that, upon input $n, m, y_0, \ldots, y_n, z_0, \ldots, z_{m-1}$ outputs (the encoding of) a straight-line program

$$P = (\{y_\nu \mid \nu = 0, \ldots, n\} \cup \{z_\mu \mid \mu = 0, \ldots, m-1\}, V, C, \{2\}),$$

such that $c_\kappa \in \operatorname{sem}(P)$, $0 \leq \kappa \leq n - m$, for the polynomial quotient of $f(x)$ and $g(x)$,

$$q(x) = c_{n-m}x^{n-m} + \cdots + c_0 \in D[x], \qquad \deg(f(x) - q(x)g(x)) < m,$$

and $\operatorname{len}(P) = O(n \log m \log(\log m))$. The existence of such a program follows from several sophisticated results on polynomial multiplication [39] and power series inversion [30]. Our question is: What is the binary complexity of the algorithm that generates such a straight-line program?

Fortunately, the answer is not difficult. An algebraic RAM over D that actually computes the c_κ by the asymptotically fast polynomial division algorithm performs only arithmetic on elements in D and tests only addresses. The problem is that such an algebraic RAM has high binary complexity owing to the fact that the calculated elements in D are dense multivariate polynomials. However, if we represent all calculated elements implicitly as the semantics of the variables of a certain straight-line program, this exponential growth does not occur. The algebraic RAM over D that generates the straight-line answer now "simulates" the arithmetic operations in D in the following way. Assume the elements

$$a = \operatorname{sem}(v_\kappa) \qquad \text{and} \qquad b = \operatorname{sem}(v_\lambda) \in D$$

need to be multiplied in the course of the polynomial division algorithm. At this point we already have a straight-line program

$$Q_l = (\{y_\nu \mid \nu = 0, \ldots, n\} \cup \{z_\mu \mid \mu = 0, \ldots, m-1\}, V_l, C_l, \{2\}),$$

such that $v_\kappa, v_\lambda \in V_l$. We now merely need to append the assignment $v_{l+1} \leftarrow v_\kappa \times v_\lambda$ to C_l and obtain a program Q_{l+1} with $ab \in \operatorname{sem}(Q_{l+1})$. The binary cost of such a simulated multiplication is $O(\log(l) + \operatorname{size}(v_\kappa) + \operatorname{size}(v_\lambda))$. It is this cheapness for arithmetic that makes the straight-line representation for multivariate polynomials so efficient. For later reference we formulate our observations as a theorem.

THEOREM 4.1 (SIMULATION PRINCIPLE). *Assume an algebraic RAM M over D on input $n \geq 1, x_1, \ldots, x_n \in D$ computes $y_1, \ldots, y_m \in D$ in $T(n)$ steps without testing an element in D for zero. Then we can construct an algebraic RAM M' over D that on the same input computes the encoding of a straight-line program P over D with*

$$\mathit{len}(P) \leq T(n) \qquad \mathit{and} \qquad \{y_1, \ldots, y_m\} \subset \mathit{sem}(P),$$

such that M' has binary complexity

$$O(T(n) \log T(n) + \sum_{\nu=1}^{n} size(x_\nu)).$$

PROOF. By simulating arithmetic instructions as above. The additional factor of log $T(n)$ in the binary complexity arises from the binary cost of each individual simulation step. The cost $\sum_{\nu=1}^{n}$ size(x_ν) enters because we must initialize certain program variables to x_ν. Other program variables will be initialized to constants from the program, but since the number of such constants is fixed, the binary cost of those initializations takes constant time. □

The polynomial division algorithm was special because no elements in D needed to be tested for zero. This is also true for polynomial or matrix multiplication, but for other important algebraic algorithms, such as computing the rank of a matrix polynomial, such tests cannot be entirely avoided. However, Algorithm Zero Test, together with choosing the evaluation points randomly, allows us to extend the simulation principle to certain algebraic RAM programs on data represented by straight-line programs, even when those RAMs also test domain elements for zero. We first justify the failure probabilities by the following two lemmas:

LEMMA 4.1 [37, lemma 1]. *Let $0 \neq f \in E[x_1, \ldots, x_n]$, E an integral domain, $R \subset E$. Then for randomly selected $a_1, \ldots, a_n \in R$, the probability*

$$Pr(f(a_1, \ldots, a_n) = 0) \leq \frac{deg(f)}{card(R)}.$$

(*We also refer to* [16] *for an interesting characterization of a suitable set of n-tuples that distinguishes all nonzero polynomials given by short straight-line programs from the zero polynomial.*)

LEMMA 4.2. *Let $P = (\{x_1, \ldots, x_n\}, V, C, S)$ be a straight-line program of length l over $F(x_1, \ldots, x_n)$, F a field, $S \subset F$. Furthermore, assume that $a_1, \ldots, a_n \in R \subset F$ were randomly selected. Then, the probability that P is defined at $\phi(x_\nu) = a_\nu$, $1 \leq \nu \leq n$, is not less than $1 - 2^{l+1}/card(R)$.*

PROOF. It follows by induction on λ that

$$\deg(\text{num}(\text{sem } v_\lambda)), \deg(\text{den}(\text{sem } v_\lambda)) \leq 2^\lambda, \qquad 1 \leq \lambda \leq l.$$

Thus

$$\deg\left(\prod_{\lambda=1}^{l} \text{den}(\text{sem}(v_\lambda))\right) \leq \sum_{\lambda=1}^{l} 2^\lambda < 2^{l+1},$$

and the lemma follows from the previous one. □

We now can demonstrate our probabilistic simulation principle on the example of computing a determinant with sparse polynomial entries in $F[x_1, \ldots, x_n]$. If we perform Gaussian elimination or the asymptotically faster algorithm by Bunch and Hopcroft [6], certain elements in $F[x_1, \ldots, x_n]$ need to be tested for nonzero before one can divide by them. At that point, these elements are computed by a straight-line program and we can probabilistically test them by picking a random evaluation and applying the Zero-Test Algorithm. The latter only needs to be called if size growth over $F = \mathbf{Q}$ is to be controlled. If we choose our evaluation points from a sufficiently large set, then, by Lemma 4.2 and the Zero-Test Algorithm, the chance that we miss a nonzero element can be made arbitrarily small. We point

```
1:  CONST  1, 7  Comment: Store 7 into data register 1
2:  JMPZ   1, 4  Comment: If data register 1 contains 0, goto label 4
3:  HALT   —  —
4:  JMP    4     Comment: Infinite loop
```

FIG. 3. RAM with unbounded computation tree.

out that it is here where we can make full use of our RANDOM instruction introduced in Section 2. If $F = \mathbf{F}_q$, we may have to evaluate over an algebraic extension $\mathbf{F}_{q^k}$ in order to poll from a large enough set. The produced straight-line program is always correct, provided we know in advance that the determinant is nonzero. Otherwise, we might with controllably small probability output a program computing 0, even if the determinant is not, instead of returning "failure".

The probabilistic computation of a straight-line program of an $m \times m$ determinant over $F[x_1, \ldots, x_n]$ takes binary polynomial-time in m, n, the coefficient size of the sparse polynomial entries, and $\log 1/\epsilon$, where the resulting program is guaranteed to be correct with probability greater than $1 - \epsilon$. This is true even if we miss a nonzero "pivot" element. The reason is that the Gaussian elimination or the Bunch and Hopcroft algorithms always terminate in $O(m^3)$ steps, whether or not the zero tests are decided correctly. General algebraic RAMs can be programmed in such a way that an impossible branch of the computation leads to an infinite loop. A section of a program with that property is shown in Figure 3.

In order to formulate the next theorem, we need, therefore, to introduce the *computation tree* complexity of an algebraic RAM, which is the maximum depth of any path in the computation, ignoring whether the decisions along the path can actually be taken. We then have the following theorem:

THEOREM 4.2 (PROBABILISTIC SIMULATION PRINCIPLE). *Assume that an algebraic RAM M over $D = F(x_1, \ldots, x_n)$ on input $n \geq 1$, $x_1, \ldots, x_n$ computes $y_1, \ldots, y_m \in D$ in $T(n)$ computation tree steps. Then we can construct a probabilistic algebraic RAM M' over D that computes the encoding of a straight-line program P over D with $len(P) \leq T(n)$ on the same input, such that with probability not less than $1 - \epsilon$, $\{y_1, \ldots, y_m\} \subset sem(P)$. Furthermore, M' requests random elements from a set $R \subset F$ with*

$$card(R) = \left\lceil \frac{T(n)2^{T(n)+2}}{\epsilon} \right\rceil,$$

and has arithmetic complexity $T(n)^{O(1)}$. For $F = \mathbf{Q}$ and $F = \mathbf{F}_q$, M' has binary complexity $(T(n)log(1/\epsilon))^{O(1)}$.

PROOF. All instructions of M except JMPZ instructions are treated as in Theorem 4.1. In order to decide which branch to select on simulation of a zero test, we randomly select elements in R and perform the Zero-Test Algorithm on the current straight-line program defining the element to be tested. The length of that intermediate program is no more than $T(n)$, and, by Lemma 4.2, an incorrect answer is returned with probability less than $2^{T(n)+2}/\text{card}(R)$, because the program in step Z of the Zero-Test Algorithm is one instruction longer. Clearly, at most $T(n)$ such tests arise, and we do not decide any of them wrongly—even using one and the same evaluation for sake of efficiency—with probability less than

$$\frac{T(n)2^{T(n)+2}}{\text{card}(R)} \leq \epsilon. \qquad \square$$

For special problems, such as the symbolic determinant computation, the randomizations introduced in the above theorem are not essential. If we remove the divisions from the generic determinant computation by Strassen's method (see Theorem 7.1), we can deterministically produce in polynomial time a straight-line program for a symbolic determinant. However, the length of this program is $O(M(m)m^3)$, where m is the dimension of the input matrix. It is an open question whether an improvement to $O(m^3)$ is possible. We remark also that Ibarra et al. [19] observed a similar trade-off for removing decisions in the matrix rank problem.

5. *Polynomial Coefficients*

We now describe an important utility algorithm for our theory. Assume that $f \in F[x_1, \ldots, x_n]$, F a field, is given by a straight-line program P over $F(x_1, \ldots, x_n)$ and that we know a bound d such that

$$f = \sum_{\delta=0}^{d} c_\delta(x_2, \ldots, x_n)x_1^\delta, \qquad c_\delta(x_2, \ldots, x_n) \in F[x_2, \ldots, x_n]. \tag{3}$$

We want to produce a straight-line program Q over $F(x_2, \ldots, x_n)$ such that $c_0, \ldots, c_d \in \operatorname{sem}(Q)$. The solution we present in this section is based on the idea of computing the c_δ by interpolating at different points. In Section 7, we give our original solution to this problem [21], which is based on the then needed Taylor Series algorithm, and which is not only more complicated but which also leads to an asymptotically longer result. Here now is the algorithm.

Algorithm *Polynomial Coefficients* 1

Input: $f \in F[x_1, \ldots, x_n]$ given by a straight-line program $P = (\{x_1, \ldots, x_n\}, V, C, S)$ over $F(x_1, \ldots, x_n)$ of length l, a failure probability $\epsilon \ll 1$, and a bound $d \geq \deg_{x_1}(f)$.

Output: Either "failure", this with probability less than ϵ, or a straight-line program $Q = (\{x_2, \ldots, x_n\}, V_Q, C_Q, S_Q)$ over $F(x_2, \ldots, x_n)$ such that

$$\{c_0, \ldots, c_d\} \subset \operatorname{sem}(Q) \qquad \text{and} \qquad \operatorname{len}(Q) = O(ld + M(d)\log d),$$

where c_δ is defined in (3).

Step E (Good Evaluation Points). From a set $R \subset F$ with

$$\operatorname{card}(R) > \frac{8\max((d+1)^2, 2^{l+1})}{\epsilon},$$

randomly select elements $a_1, \ldots, a_n$. If $F = \mathbf{F}_q$ and q is too small, we can work over an algebraic extension $\mathbf{F}_{q^j}$ with j sufficiently large.

Test whether P is defined at $\phi(x_\nu) = a_\nu$, $1 \leq \nu \leq n$. For $F = \mathbf{Q}$ we call Algorithm Zero-Division Test of Section 3, such that the probability of "failure" even if P were defined at ϕ is less than $\epsilon/4$. If P turns out to be (probably) undefined at ϕ, we return "failure". Otherwise, P is (definitely) defined at ϕ.

Step P (Interpolation Points). $B \leftarrow \{a_1\}$.

repeat the following at most $(d+1)^2$ times **until** $\operatorname{card}(B) = d + 1$.

From set R, select a random point b. If b was chosen in previous iterations or is equal to a_1, we continue with the next repetition. Otherwise, test whether P is defined at $\psi(x_1) = b$, $\psi(x_i) = a_i$, $2 \leq i \leq n$. If P is defined at ψ, adjoin the element b to B. For $F = \mathbf{Q}$, we make the probability that we do not recognize this fact properly by calling the Zero-Division Test Algorithm of Section 3 less than $\epsilon/(4d + 4)$.

If at this point $\mathrm{card}(B) < d + 1$, we return "failure".

Step I (Interpolation Construction). At this point we have $B = \{b_1, \ldots, b_{d+1}\}$, such that P is defined at all $\chi_i(x_1) = b_i$. We first build programs Q_i over $F(x_2, \ldots, x_n)$ such that

$$f(b_i, x_2, \ldots, x_n) \in \mathrm{sem}(Q_i), \qquad 1 \le i \le d + 1.$$

This is done by simply replacing each occurrence of x_1 on the right side of an assignment in the computation sequence of P by b_i. Then, we build a program Q_0 that from the symbolic values w_i of a d-degree polynomial evaluated at b_i finds the coefficients of that polynomial. This is the interpolation problem, which can be solved classically in $\mathrm{len}(Q_0) = O(d^2)$, or asymptotically faster in $\mathrm{len}(Q_0) = O(M(d)\log(d))$ [1]. Notice that the algebraic RAM performing interpolation does not require zero tests of field elements. Finally, we link the programs $Q_1, \ldots, Q_{d+1}, Q_0$ properly together making sure that there is no naming conflict and that the w_i are the corresponding variables in Q_i. □

The following theorem summarizes the complexity of our algorithm.

THEOREM 5.1. *Algorithm Polynomial Coefficients* 1 *does not fail with probability greater than* $1 - \epsilon$. *It requires polynomially many arithmetic steps in d and l on a probabilistic algebraic RAM over F. For $F = \mathbf{Q}$ and $F = \mathbf{F}_q$, its binary complexity is also polynomial in el-size(P) and $\log(1/\epsilon)$.*

PROOF. The algorithm can fail under four different circumstances. First, in Step E, P may be undefined at ϕ, that by Lemma 4.2 with probability less than $2^{l+1}/\mathrm{card}(R) < \epsilon/4$. Second, for $F = \mathbf{Q}$, we might fail to recognize that P is defined at ϕ, but we make this possibility happen with probability less than $\epsilon/4$. Third, the loop in step P may not generate $d + 1$ distinct b such that P is defined at the corresponding ψ. Since we try $(d + 1)^2$ points, we can estimate this particular failure possibility as follows. A newly selected b was not chosen earlier with probability greater than or equal to $1 - (d + 1)^2/\mathrm{card}(R) > 1 - \epsilon/8$. Then, again by Lemma 4.2, P is not defined at ψ for that individual point with probability less than $2^{l+1}/\mathrm{card}(R) < \epsilon/8$. Therefore, a suitable evaluation point can be found in a block of $d + 1$ points with probability greater than

$$1 - (\epsilon^*)^{d+1} > 1 - \frac{\epsilon^*}{d+1}, \qquad \epsilon^* = \frac{\epsilon}{4},$$

because $(1/\epsilon^*)^d > 2^d \ge d + 1$ for $\epsilon^* < \frac{1}{2}$. Now the probability that a good point occurs in all of the $d + 1$ blocks of points is greater than

$$\left(1 - \frac{\epsilon^*}{d+1}\right)^{d+1} > 1 - \epsilon^*,$$

and hence failure happens for the third case with probability less than $\epsilon/4$. Fourth and last, for $F = \mathbf{Q}$, we again may not recognize that P is defined at ψ, even if there were sufficiently many points. A good point is not missed with probability greater than $1 - \epsilon/(4d + 4)$ and hence the first $d + 1$ such points are recognized with probability

$$\left(1 - \frac{\epsilon}{4(d + 1)}\right)^{d+1} > 1 - \frac{\epsilon}{4}.$$

This concludes the argument for failure probability. The statements on the arithmetic and binary running times are a direct consequence of Theorems 3.1 and 4.1. □

The Polynomial Coefficients 1 Algorithm requires the knowledge of a bound $d \geq \deg_{x_1}(f)$. If no such bound is given, we can probabilistically guess the degree by running our algorithm for

$$d = 1, 2, 4, \ldots, 2^k, \ldots.$$

Let $f_k(x_1, \ldots, x_n)$ be the interpolation polynomial that is produced for the kth run. We then choose $a_1, \ldots, a_n \in R$ randomly and probabilistically test whether

$$f(a_1, \ldots, a_n) - f_k(a_1, \ldots, a_n) = 0.$$

If the Zero-Test Algorithm called with a failure chance $\epsilon/2$ returns "probably 0", then by Lemma 4.1 with probability greater than $1 - \epsilon$, $f_k = f$ and $2^k \geq \deg_{x_1}(f)$. Of course, by further testing $c_\delta(x_2, \ldots, x_n)$ for zero, $\delta = 2^k, 2^k - 1, \ldots$, we can get a probabilistic estimate for the actual degree, $\deg_{x_1}(f)$. This procedure has expected polynomial running time in $\deg_{x_1}(f)$ and can be made quite efficient by computing the $f_k(x_1, a_2, \ldots, a_n)$ incrementally without even constructing a straight-line program for any f_k [10]. The total degree of f can be similarly estimated using the translations that we introduce in Section 6, or by computing the degree of $f(y_1 z, \ldots, y_n z)$ in z. A more general degree test is discussed in Section 8 (cf. Corollary 8.1).

One may question whether it is possible to find a program of length polynomial in l only for a selected $c_\delta(x_2, \ldots, x_n)$, $1 \leq \delta \leq 2^l$. This is most likely not the case, as Valiant's example [45] exhibits. Consider

$$g(y_1, \ldots, y_n, z_{1,1}, \ldots, z_{n,n}) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} y_j z_{i,j} \right). \tag{4}$$

Then the coefficient of the monomial $y_1 \cdots y_n$ in g is the permanent of the matrix $[z_{i,j}]_{1 \leq i,j \leq n}$. Performing a Kronecker substitution of $x^{(n+1)^{i-1}}$ for y_i this permanent appears as the coefficient of $c_\delta(z_{1,1}, \ldots, z_{n,n})$ of x^δ for

$$\delta = 1 + (n + 1) + (n + 1)^2 + \cdots + (n + 1)^{n-1},$$

in

$$g(x, x^{n+1}, \ldots, x^{(n+1)^{n-1}}, z_{1,1}, \ldots, z_{n,n}).$$

Therefore, the degree-unrestricted coefficients problem is #P-hard [44].

The example above also shows that certain operations on straight-line programs most likely cannot be iterated without increasing the length of the output program exponentially. Take, for example, computing partial derivatives. Clearly, by our Polynomial Coefficients 1 Algorithm, we can find a program Q with

$$\frac{\partial^k f}{\partial x_1^k} \in \mathrm{sem}(Q) \qquad \text{and} \qquad \mathrm{len}(Q) = O(ld^2).$$

In order to obtain multiple partial derivatives in different variables, we could iterate this process on the distinct variables. However, every iteration increases the length of the previous program by at least a constant factor, and the final program turns out to be of exponential length in the number of different variables. This blow-up also appears to be inherent, because from (4) we get

$$\frac{\partial^n g}{\partial y_1 \cdots \partial y_n} = \operatorname{perm}([z_{i,j}]_{1\le i,j\le n}).$$

It came as a surprise to us that certain iterations causing a similar exponential growth, such as the variable-by-variable Hensel lifting [22], do not constitute inherent complexity and can be avoided [23].

6. *Polynomial Greatest Common Divisors*

We now come to the first application of our theory, that of computing polynomial GCDs. Our goal is to produce for r polynomials $f_\rho \in F[x_1, \ldots, x_n]$, $1 \le \rho \le r$, given by a straight-line program P, a straight-line program Q with $\mathrm{GCD}_{1\le\rho\le r}(f_\rho) \in \operatorname{sem}(Q)$. For simplicity, we are assuming that all f_ρ are computed by a single program P. Clearly, this can be enforced by merging any possibly different input programs. We also assume that we know an a priori bound $d \ge \deg(f_\rho)$, $1 \le \rho \le r$. Our algorithm is a probabilistic one, and the returned Q may not determine the correct GCD, that with probability less than ϵ. The difficulty is, of course, to accomplish the construction in binary polynomial time in

$$\operatorname{len}(P),\ \operatorname{el-size}(P),\ d,\ \log\left(\frac{1}{\epsilon}\right).$$

Note that the parameters n and r are dominated by $\operatorname{len}(P)$. We do not know how the approach of repeated GCD computations,

$$\mathrm{GCD}(f_1, f_2),\ \mathrm{GCD}(\mathrm{GCD}(f_1, f_2), f_3), \ldots,$$

or of extracting $\operatorname{cont}_{x_1}(f_\rho)$, $1 \le \rho \le r$ (cf. Brown [5]) can lead to a polynomial-time solution.

We first restrict ourselves to $r = 2$, that is, the GCD problem for two polynomials. We later show that the GCD problem for many polynomials can be probabilistically reduced to that for two. In order to avoid the content computation, we work with the translated polynomials

$$\tilde{f}_\rho = f_\rho(x_1, y_2 + b_2 x_1, \ldots, y_n + b_n x_1) \in F[x_1, y_2, \ldots, y_n], \qquad \rho = 1, 2,$$

where $b_\nu \in F$ are randomly selected elements, $2 \le \nu \le n$. Since the mapping $f \to \tilde{f}$ is a ring isomorphism from $F[x_1, \ldots, x_n]$ into $F[x_1, y_2, \ldots, y_n]$, we must have $\tilde{g} = \mathrm{GCD}(\tilde{f}_1, \tilde{f}_2)$, where $g = \mathrm{GCD}(f_1, f_2)$. The reason for performing this translation is that with high probability $\operatorname{ldcf}_{x_1}(\tilde{f}_\rho) \in F$, $\rho = 1$ or 2. The following easy lemma can be formulated.

LEMMA 6.1. *Let $f \in F[x_1, \ldots, x_n]$, $b_2, \ldots, b_n \in F$. Then there exists a non-zero polynomial $\tau(\beta_2, \ldots, \beta_n) \in F[\beta_2, \ldots, \beta_n]$, $\deg(\tau) \le \deg(f)$, such that $\tau(b_2, \ldots, b_n) \ne 0$ implies*

$$\operatorname{ldcf}_{x_1}(f(x_1, y_2 + b_2 x_1, \ldots, y_n + b_n x_1)) \in F \text{ over } F[x_1, y_2, \ldots, y_n].$$

PROOF. Let

$$0 \neq \tau = \operatorname{ldcf}_{x_1}(f(x_1, y_2 + \beta_2 x_1, \ldots, y_n + \beta_n x_1)) \in F[\beta_2, \ldots, \beta_n],$$

over $F[x_1, y_2, \ldots, y_n, \beta_2, \ldots, \beta_n]$ and apply Lemma 4.1 to τ. □

The trick is now to perform the Euclidean algorithm (i.e., compute a polynomial remainder sequence) on the translated polynomials over the coefficient field $F(y_2, \ldots, y_n)$ in the variable x_1. Let

$$\tilde{g} = \operatorname{GCD}(\tilde{f}_1, \tilde{f}_2) \text{ over } F(y_2, \ldots, y_n)[x_1], \operatorname{ldcf}_{x_1}(\tilde{g}) = 1.$$

The point is that, if Lemma 6.1 applies to $\tilde{f}_1$ or $\tilde{f}_2$, that is,

$$\operatorname{ldcf}_{x_1}(\tilde{f}_1) \quad \text{or} \quad \operatorname{ldcf}_{x_1}(\tilde{f}_2) \in F, \tag{5}$$

then $\tilde{g}$ will actually be the GCD of $\tilde{f}_1$ and $\tilde{f}_2$ over $F[y_2, \ldots, y_n, x_1]$. This is a consequence of Gauss's lemma [28, sect. 4.6.1, lemma G], which states that products of primitive polynomials must be primitive. The claim about $\tilde{g}$ can be shown from this, as follows: Assume that (5) is true for $\tilde{f}_1$, and let $\tilde{g}^* = \tilde{f}_1/\tilde{g}$. Furthermore, let c and $c^* \in F[y_2, \ldots, y_n]$ be the least common denominators of $\tilde{g}$ and $\tilde{g}^*$, respectively. Now

$$(c\tilde{g})(c^*\tilde{g}^*) = (cc^*)\tilde{f}_1,$$

where $c\tilde{g}$ and $c^*\tilde{g}^*$ are primitive in $F[x_1, y_2, \ldots, y_n]$ with respect to x_1. Therefore, $cc^* \in F$ and hence $\tilde{g} \in F[x_1, y_2, \ldots, y_n]$. Since $\tilde{g}$ is monic with respect to x_1, $\tilde{g}$ also divides $\tilde{f}_2$ over $F[x_1, y_2, \ldots, y_n]$, irrespective of whether $\operatorname{ldcf}_{x_1}(\tilde{f}_2) \in F$. Since $\tilde{g}$ is computed in a larger domain, our claim is immediate.

Our algorithm constructs the polynomial remainder sequence for $\tilde{f}_1$ and $\tilde{f}_2$. We shall work on the coefficient vectors with respect to x_1, which we can obtain initially in straight-line representation by the polynomial coefficients algorithm. During this process we must, however, compute the degrees of the remainders in x_1. We do this probabilistically by evaluating y_ν randomly at a_ν, as was done also for the probabilistic simulation principle. The algorithm now follows in detail.

Algorithm *Polynomial GCD*

Input: $f_0, f_1 \in F[x_1, \ldots, x_n]$ of degree $\leq d$ given by the straight-line program $P = (\{x_1, \ldots, x_n\}, V, C, S)$ of length l, and a failure allowance $\epsilon \ll 1$.

Output: Either "failure", with probability less than ϵ, or a straight-line program $Q_0 = (\{x_1, \ldots, x_n\}, V_0, C_0, S_0)$ over $F(x_1, \ldots, x_n)$ of length $O(ld + d^2)$ such that with probability greater than or equal to $1 - \epsilon$

$$\operatorname{GCD}(f_0, f_1) \in \operatorname{sem}(Q_0).$$

Step R (Random Points Selection). From a subset $R \subset F$ with

$$\operatorname{card}(R) > \frac{\max(2^{l+4}, 8d^3)}{\epsilon},$$

select randomly $a_1, \ldots, a_n, b_2, \ldots, b_n$. In case $F = \mathbf{F}_q$ where q is too small, we can work in $\mathbf{F}_{q^j}$ instead. Since the GCD can be computed by coefficient arithmetic alone, it remains invariant under field extensions.

Step T (Translation). Set $\tilde{C} = (u_2 \leftarrow x_1 \times b_2, \tilde{x}_2 \leftarrow y_2 + u_2, \ldots, u_n \leftarrow x_1 \times b_n, \tilde{x}_n \leftarrow y_n + u_n)$ concatenated with the modified computation sequence of P in which all occurrences of x_ν are replaced by $\tilde{x}_\nu$. Thus

$$\tilde{P} = \left(\{x_1, y_2, \ldots, y_n\}, V \cup \bigcup_{2\le\nu\le n} \{u_\nu, \tilde{x}_\nu\}, \tilde{C}, S \cup \{b_2, \ldots, b_n\}\right),$$

is a straight-line program over $F(x_1, y_2, \ldots, y_n)$ that computes

$$\tilde{f}_\rho = \sum_{\delta=0}^{d} c_{\rho,\delta}(y_2, \ldots, y_n)x_1^\delta \in F[x_1, y_2, \ldots, y_n], \qquad \rho = 0, 1.$$

Step C (Coefficient Determination). Test whether $\tilde{P}$ is defined at $\phi(x_1) = a_1$, $\phi(y_\nu) = a_\nu$, $2 \le \nu \le n$. If not, return "failure". For $F = \mathbf{Q}$, we call Algorithm Zero-Division Test with failure probability $\epsilon/3$.

Call Algorithm Polynomial Coefficients 1 with input program $\tilde{P}$, degree bound d, failure probability $\epsilon/3$, and the indices λ_ρ such that $\tilde{f}_\rho = \operatorname{sem}(v_{\lambda_\rho})$ for $\rho = 0$ and 1. We obtain

$$Q_1 = (\{y_2, \ldots, y_n\}, W_1, C_1, T_1),$$

such that all $c_{\rho,\delta} \in \operatorname{sem}(Q_1)$. Notice that, in the Polynomial Coefficients 1 algorithm, we only need to evaluate on one set of points, even though Q_1 encodes the calculation of two interpolation polynomials. This shortens $\operatorname{len}(Q_1)$ considerably. We also share $a_1, \ldots, a_n$ with that algorithm instead of selecting new points in step E there. This guarantees that Q_1 is defined at ϕ restricted to $y_2, \ldots, y_n$. We could have tested for this condition after constructing Q_1, but the error analysis would be a little more involved.

Step D (Degree Determination). In this step we probabilistically find $d_0 = \deg_{x_1}(\tilde{f}_0)$ and $d_1 = \deg_{x_1}(\tilde{f}_1)$.

for $\delta \leftarrow d, d-1, \ldots, 0$ **do**

Call Algorithm Zero Test with Q_1, $\phi(y_\nu) = a_\nu$, λ such that $c_{0,\delta} = \operatorname{sem}(w_{1,\lambda})$, $w_{1,\lambda} \in W_1$, and failure probability $\epsilon/(4d)$. If "definitely $\neq 0$" is returned, exit the loop with $d_0 = \delta$. Notice then that

$$\Pr(d_0 = \deg_{x_1}(\tilde{f}_0)) \ge 1 - \frac{\epsilon}{4d}.$$

Here we dropped through the loop, that is, with high probability $\tilde{f}_0 = 0$. By convention, we set $d_0 = -1$.

Similarly, compute d_1. Without loss of generality we now assume that $d_0 \ge d_1$.

Step E (Euclidean Loop). **for** $k \leftarrow 1, 2, \ldots,$ **do** Step R.

Step R (Polynomial Remaindering). At this point we have a straight-line program

$$Q_k = (\{y_2, \ldots, y_n\}, W_k, C_k, T_1),$$

such that for the ith polynomial remainder in the Euclidean remainder sequence of $\tilde{f}_0$ and $\tilde{f}_1$ over $F(y_2, \ldots, y_n)[x_1]$

$$\tilde{f}_i = \sum_{\delta=0}^{d_i} c_{i,\delta}x_1^\delta, \qquad c_{i,\delta} \in F[y_2, \ldots, y_n], \quad 1 \le i \le k,$$

with high probability

$$c_{j,\delta} \in \operatorname{sem}(Q_i) \qquad \text{for all} \quad 0 \le \delta \le d_j, \quad 1 \le j \le i.$$

Now we update Q_k to the straight-line program Q_{k+1}, which also simulates the polynomial division of $\tilde{f}_{k-1}$ and $\tilde{f}_k$ over $F(y_2, \ldots, y_n)[x_1]$. Provided Q_k was correct, the program exactly determines the next remainder $\tilde{f}_{k+1}$, whose degree $d_{k+1} < d_k$ we determine as shown in step D.

if $d_{k+1} = -1$ (i.e., $\tilde{f}_{k+1} = 0$), **then** proceed to step G.

Step G (GCD Generation). Q_k now determines, with high probability, the coefficients of a remainder $\tilde{f}_k = \sum_{\delta=0}^{d_k} c_{k,\delta} x_1^\delta$, which corresponds to the GCD of $\tilde{f}_0$ and $\tilde{f}_1$ over $F(y_2, \ldots, y_n)[x_1]$. Append assignments to C_k that compute

$$\frac{\tilde{f}_k}{\operatorname{ldcf}_{x_1}(\tilde{f}_k)} = \sum_{\delta=0}^{d_k} \frac{c_{k,\delta}}{c_{k,d_k}} x_1^\delta.$$

This makes the computed GCD monic in x_1 and by the discussion previous to the algorithm we have, with high probability, the GCD of $\tilde{f}_0$ and $\tilde{f}_1$ over $F[y_2, \ldots, y_n, x_1]$. Finally, put assignments computing the back-translation $y_2 \leftarrow x_2 - x_1 b_2, \ldots, y_n \leftarrow x_n - x_1 b_n$ in front of C_k and output $Q_0 = (\{x_1, \ldots, x_n\}, W_k', C_k', T_1)$, where W_k' and C_k' are the updated W_k and C_k. □

The following theorem summarizes the complexity of the Polynomial GCD Algorithm.

THEOREM 6.1. *Algorithm Polynomial GCD does not fail with probability greater than* $1 - \epsilon$. *In that case, its output correctly determines the GCD of its inputs with probability greater than* $1 - \epsilon$. *It requires polynomially many arithmetic steps in d and l on a probabilistic algebraic RAM over F. For* $F = \mathbf{Q}$ *and* $F = \mathbf{F}_q$, *its binary complexity is also polynomial in el-size(P) and* $\log(1/\epsilon)$.

PROOF. Polynomial running time follows from Theorems 3.1, 4.1, and 5.1. "Failure" can only be returned in step C. There are three possibilities that can cause such an event. First, the program $\tilde{P}$ may not be defined at ϕ. By Lemma 4.2, this happens with probability less than $2^{l+1}/\operatorname{card}(R) < 1/(3\epsilon)$. Second, for $F = \mathbf{Q}$, we might fail to recognize that $\tilde{P}$ is defined at ϕ, but we make this possibility happen with probability less than $\epsilon/3$. Third, the Polynomial Coefficients 1 Algorithm may fail, that with probabiltiy less than $\epsilon/3$.

We now establish the estimates for the probability that Q_k determines the GCD. Let τ_1 by the polynomial from Lemma 6.1 corresponding to f_1. The degree $\deg(\tau_1) \leq d$ and by Lemma 4.1

$$\Pr(\operatorname{ldcf}_{x_1}(\tilde{f}_1) \in F) \geq 1 - \frac{\epsilon}{8d^2} > 1 - \frac{\epsilon}{4}. \tag{6}$$

If this is the case, step G is justified. Now we consider under which circumstances we obtain the correct degrees d_k. In order to obtain a sharp estimate, we appeal to the theory of subresultants (cf. [28, sect. 4.6.1 and the references there]). A reader unfamiliar with that theory can refer back to the probabilistic simulation principle, but then card(R) would be much larger than what we can prove. By $\bar{c}_i \in F[y_2, \ldots, y_n]$ we denote the leading coefficient of the d_i-degree subresultant of $\tilde{f}_0$ and $\tilde{f}_1$ with respect to x_1, $0 \leq i \leq k$. Then $\deg(\bar{c}_i) \leq 2d^2$, and for each c_{i,d_i} there exist integers $e_{i,j}$ such that

$$c_{i,d_i} = \prod_{j=0}^{i} \bar{c}_j^{e_{i,j}}, \qquad 0 \leq i \leq k. \tag{7}$$

Furthermore, let $\sigma = \prod_{i=0}^{k} \tilde{c}_i \in F[y_2, \ldots, y_n]$. Since $\deg(\sigma) \leq 2d^3$,

$$\Pr(\sigma(a_2, \ldots, a_n) \neq 0 \mid a_2, \ldots, a_n \in R) \geq 1 - \frac{\epsilon}{4}. \tag{8}$$

Assume now that this is the case, which means by (7) that no leading coefficient of $\tilde{f}_i$ evaluates to zero. We test overall at most $2d$ coefficients of $\tilde{f}_0, \tilde{f}_1, \ldots, \tilde{f}_k$ for zero. For $F = \mathbf{Q}$, none of these tests misses a nonzero evaluation with probability greater than or equal to

$$\left(1 - \frac{\epsilon}{4d}\right)^{2d} > 1 - \frac{\epsilon}{2}. \tag{9}$$

Notice that all programs Q_i remain defined at $\phi(y_\nu) = a_\nu$, $2 \leq \nu \leq n$. Therefore, all events (6), (8), and (9) occur with probability greater than $1 - (\epsilon/4 + \epsilon/4 + \epsilon/2) > 1 - \epsilon$. In that case C_k is a straight-line program for the GCD. $\square$

We used the theory of subresultants only in our proof, but we could as well have used the more involved subresultant pseudodivisions in step R of our algorithm. Then the evaluations over $\mathbf{Q}$ would stay better bounded in size and we would be even less likely to miss a nonzero leading coefficient of a remainder. Instead of the classical Euclidean algorithm, we could also have used the asymptotically faster Knuth–Schönhage algorithm [33]. This would shorten the length of Q_0 asymptotically to $O(ld + M(d)\log(d))$ with a different bound for the cardinality of R; see Algorithm Rational Numerator and Denominator in Section 8 for more details.

We now consider the case of more than two input polynomials. For this case we use a probabilistic trick that reduces the problem of computing $g = \mathrm{GCD}_{1\leq i\leq r}(f_i)$, $f_i \in F[x_1, \ldots, x_n]$, to that of computing the GCD of two polynomials. All one has to do is take two random linear combinations $\sum_{i=1}^{r} a_i f_i$, $\sum_{i=1}^{r} b_i f_i$, $a_i, b_i \in R \subset F$, and with high probability their GCD coincides with g. The relevant theorem follows:

THEOREM 6.2. *Let $f_i \in F[x_1, \ldots, x_n]$, F a field, $\deg(f_i) \leq d$ for $1 \leq i \leq r$, $R \subset F$. Then for randomly chosen $a_i, b_i \in R$, $1 \leq i \leq r$,*

$$Pr\left(GCD_{1\leq i\leq r}(f_i) = GCD\left(\sum_{i=1}^{r} a_i f_i, \sum_{i=1}^{r} b_i f_i\right)\right) \geq 1 - \frac{2d}{card(R)}.$$

PROOF. We first show this theorem for $n = 1$. Let

$$\hat{f}_1 = \sum_{i=1}^{r} \alpha_i f_i, \hat{f}_2 = \sum_{i=1}^{r} \beta_i f_i \in E[x], \qquad E = F[\alpha_1, \ldots, \alpha_r, \beta_1, \ldots, \beta_r],$$

$g = \mathrm{GCD}_{1\leq i\leq r}(f_i)$. Clearly, $g \mid \hat{f}_1$, $g \mid \hat{f}_2$. The first claim is that $g = \hat{g}$ where $\hat{g} = \mathrm{GCD}(\hat{f}_1, \hat{f}_2)$. We observe that $\hat{g} \in F[x]$, since the sets of the other indeterminates in $\hat{f}_1$ and $\hat{f}_2$ are disjoint. Now write $\hat{f}_1 = \hat{g}\hat{f}_1^*$, where $\hat{f}_1^* \in E[x]$. If we evaluate this equation at $\alpha_i = 1$ and $\alpha_j = 0$, $j \neq i$, then we get $\hat{g} \mid f_i$, $1 \leq i \leq r$. Therefore, $\hat{g} \mid g$, which proves the claim. Now let $\sigma \in E$ be the leading coefficient of the subresultant of $\hat{f}_1$ and $\hat{f}_2$ with respect to x that corresponds to $\hat{g}$. If $\sigma(a_1, \ldots, a_r, b_1, \ldots, b_r) \neq 0$, then

$$\begin{aligned}&\mathrm{GCD}(\hat{f}_1(a_1, \ldots, a_r, b_1, \ldots, b_r, x), \hat{f}_2(a_1, \ldots, a_r, b_1, \ldots, b_r, x))\\ &\quad = \mathrm{GCD}(\hat{f}_1, \hat{f}_2)(a_1, \ldots, a_r, b_1, \ldots, b_r, x),\end{aligned}$$

which implies the asserted event. Since $\deg(\sigma) \leq 2d$, Lemma 4.1 establishes the stated probability.

We now reduce the multivariate to the univariate case by using the translation of Lemma 6.1 generically. Consider for $f \in F[x_1, \ldots, x_n]$

$$\bar{f} = f(x_1, y_2 + z_2x_1, \ldots, y_n + z_nx_1) \in F(z_2, \ldots, z_n)[x_1, y_2, \ldots, y_n].$$

Now $\bar{g} = \mathrm{GCD}_{1\le i\le r}(\bar{f}_i)$, where the latter can be computed over $F(z_2, \ldots, z_n, y_2, \ldots, y_n)[x_1]$ since

$$\mathrm{ldcf}_{x_1}(\bar{f}) \in F(z_2, \ldots, z_n).$$

From the univariate case, it follows then that

$$\Pr\left[\left(\bar{g} = \mathrm{GCD}\left(\sum_{i=1}^{r} a_i\bar{f}_i, \sum_{i=1}^{r} b_i\bar{f}_i\right) \,\middle|\, a_i, b_i \in R\right)\right] \ge 1 - \frac{2d}{\mathrm{card}(R)}.$$

However, the mapping Φ defined by $\Phi(x_1) = x_1$, $\Phi(y_i) = x_i - z_ix_1$ is a ring-isomorphism from $F(z_2, \ldots, z_n)[x_1, y_2, \ldots, y_n]$ into $F(z_1, \ldots, z_n)[x_1, \ldots, x_n]$. Applying this mapping to the above event, we therefore obtain the theorem. (See Note Added in Proof.) □

7. *Taylor Series Coefficients*

We now present a different approach to finding the coefficients of a polynomial. The idea is similar to Strassen's elimination of divisions [43] and has also been mentioned by Valiant [45, end of sect. 4]. Its essence is to compute the Taylor series coefficients over $F(x_2, \ldots, x_n)[[x_1]]$ to a given order for the functions computed in all program variables. For a particular variable, these coefficients are computed from the coefficients of previous variables by Taylor series arithmetic. As we note later, Strassen's results can be reduced to our algorithm by an appropriate substitution. We first formulate the general procedure under the assumption that the rational functions computed in the variables can be expanded into Taylor series at the point $x_1 = 0$. Then we apply this procedure to the coefficients problem, as well as to eliminating divisions.

Algorithm *Taylor Series Coefficients*

Input: $f \in F(x_1, \ldots, x_n)$ given by a straight-line program $P = (\{x_1, \ldots, x_n\}, V, C, S)$ of length l that is defined at $\phi(x_1) = 0$. From this it follows that f can be expanded as a Taylor series

$$f(x_1, \ldots, x_n) = \sum_{\delta=0}^{\infty} c_\delta(x_2, \ldots, x_n)x_1^\delta, \qquad c_\delta(x_2, \ldots, x_n) \in F(x_2, \ldots, x_n).$$

Furthermore, the desired order d is given.

Output: A straight-line program $Q = (\{x_2, \ldots, x_n\}, V_Q, C_Q, S)$ over $F(x_2, \ldots, x_n)$ such that

$$\{c_0, \ldots, c_d\} \subset \mathrm{sem}(Q) \quad \text{and} \quad \mathrm{len}(Q) = O(l\, M(d)).$$

Step L (Loop through instruction sequence). $C_Q \leftarrow \emptyset$.

for $\lambda \leftarrow 1, \ldots, l$ **do** Step T.

Finally, set $V_Q = \{w_{\lambda,\delta}\} \cup \{u \mid u \text{ is any of the intermediate variables}\}$. Return $Q = (\{x_2, \ldots, x_n\}, V_Q, C_Q, S)$.

Step T (Taylor series arithmetic). Let $v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda$ be the λth assignment in C. Instead of computing $\text{sem}(v_\lambda)$, we compute the first $d + 1$ coefficients of x_1 in its power series expansion over $F(x_2, \ldots, x_n)[[x_1]]$, which we store in $w_{\lambda,0}, \ldots, w_{\lambda,d}$.

Case $\circ_\lambda = \pm$. For $\delta \leftarrow 0, \ldots, d$, append $w_{\lambda,\delta} \leftarrow w'_{\lambda,\delta} + w''_{\lambda,\delta}$ to C_Q. If $v^*_\lambda = v_\mu$, then $w^*_{\lambda,\delta} = w_{\mu,\delta}$ for $* = {}', {}''$. If $v^*_\lambda \in X \cup S$, we use

$$w^*_{\lambda,\delta} = \begin{cases} v^*_\lambda & \text{if } \delta = 0 \\ 0 & \text{if } \delta > 0 \end{cases} \quad \text{and if} \quad v^*_\lambda \in (X \subset S)\setminus\{x_1\},$$
$$= \begin{cases} 1 & \text{if } \delta = 1 \\ 0 & \text{if } \delta \neq 0 \end{cases} \quad \text{and if} \quad v^*_\lambda = x_1.$$

Case $\circ_\lambda = \times$. Construct a straight-line program which, on input $w'_{\lambda,0}, \ldots, w'_{\lambda,d}, w''_{\lambda,0}, \ldots, w''_{\lambda,d}$, computes in the variable $w_{\lambda,\delta}$, $0 \le \delta \le d$, the convolution

$$\sum_{\substack{i+j=\delta \\ 0 \le i,j \le d}} w'_{\lambda,i} w''_{\lambda,j},$$

making sure that all temporary variables are new. Now append this straight-line program to C_Q. Notice that the increase in length depends on which multiplication algorithm is used.

Case $\circ_\lambda = \div$. We first append to C_Q assignments for $u_{\lambda,\delta}$, $0 \le \delta \le d$, such that

$$\frac{1}{\text{sem}(v''_\lambda)} = \frac{1}{\sum_{\delta \ge 0} \text{sem}(w''_{\lambda,\delta}) x_1^\delta}$$
$$= \sum_{\delta \ge 0} \text{sem}(u_{\lambda,\delta}) x_1^\delta, \qquad \text{sem}(u_{\lambda,\delta}) \in F(x_2, \ldots, x_n).$$

The fastest method uses Newton iteration. We shall briefly present the algorithm for an algebraic RAM over $F(x_2, \ldots, x_n)$.

```
$\alpha_0 \leftarrow 1/\text{sem}(w''_{\lambda,0})$.
for $i \leftarrow 1, \ldots, \lceil 1 + \log d \rceil$ do
    At this point $\alpha_{i-1}$ is the $(2^{i-1} - 1)$st order approximation of $1/\text{sem}(v''_\lambda)$.
    $\alpha_i \leftarrow 2\alpha_{i-1} - \alpha_{i-1}^2 (\sum_{\delta=0}^{d} \text{sem}(w''_{\lambda,\delta}) x_1^\delta) \bmod x_1^{2^i}$.
```

Two points should be made. First, $\text{sem}(w''_{\lambda,0}) \neq 0$ since P is defined at ϕ. Second, the total number of steps is $O(\sum_{i<\log(4d)} M(2^i))$, or again $O(M(d))$.

Once the $u_{\lambda,\delta}$ are introduced we proceed as in the previous case to obtain the convolutions $\sum_{i+j=\delta} w'_{\lambda,i} u_{\lambda,j}$. □

The binary complexity of this algorithm follows from Theorem 4.1, and is on an algebraic RAM over F of order $O(lM(d)\log(lM(d)) + \text{el-size}(P))$. We wish to point out that in case many divisions occur in the computation sequence of P we can reduce the number of power series divisions to just a single one by the following idea. Instead of computing the power series approximations to order d of all $\text{sem}(v_\lambda)$, we compute the approximations of $\text{num}(\text{sem}(v_\lambda))$ and $\text{den}(\text{sem}(v_\lambda))$ separately by polynomial multiplication. Thus, the only power series division necessary is that for $\text{num}(\text{sem}(v_l))/\text{den}(\text{sem}(v_l))$.

We now apply the Taylor Series Coefficients Algorithm to the coefficients problem. The trick is to translate $x_1 = y_1 + a_1$ for a randomly chosen a_1 such that P is defined at $\phi(y_1) = 0$, which is the same as $\phi(x_1) = a_1$. To unravel this translation, however, will require a bit of work.

Algorithm *Polynomial Coefficients* 2

Input: The same as in the Polynomial Coefficients 1 algorithm.

Output: Again the same as in Algorithm Polynomial Coefficients 1, except that $\text{len}(Q) = O(lM(d))$.

Step FT (Forward Translation). From a set $R \subset F$ with $\text{card}(R) > 2^{l+1}/\epsilon$ randomly select elements $a_1, \ldots, a_n$. If $F = \mathbf{F}_q$ and q is too small, we can work over an algebraic extension $\mathbf{F}_{q^j}$ with j sufficiently large.

Test whether P is defined at $\phi(x_\nu) = a_\nu$, $1 \le \nu \le n$. For $F = \mathbf{Q}$, we call Algorithm Zero-Division Test of Section 3 such that the probability of "failure" even if P were defined at ϕ is less than $\epsilon/2$. If P turns out to be (probably) undefined at ϕ, we return "failure". Otherwise, P is (definitely) defined at ϕ.

Set $\tilde{C} = (\tilde{x}_1 \leftarrow y_1 + a_1)$ concatenated with C in which all occurrences of x_1 are replaced by $\tilde{x}_1$. Now $\tilde{P} = (\tilde{X}, \tilde{V}, \tilde{C}, \tilde{S})$ with $\tilde{X} = \{y_1, x_2, \ldots, x_n\}$, $\tilde{V} = V \cup \{\tilde{x}_1\}$, and $\tilde{S} = S \cup \{a_1\}$ is a straight-line program that computes $f(y_1 + a_1, x_2, \ldots, x_n)$ over $F(y_1, x_2, \ldots, x_n)$, and which is defined at $\phi(y_1) = 0$.

Step C. Call Algorithm Taylor Series Coefficients with program $\tilde{P}$ and order d. A program $\tilde{Q}$ is returned that computes $\tilde{c}_\delta \in F[x_2, \ldots, x_n]$, $0 \le \delta \le d$, such that

$$\sum_{\delta=0}^{d} \tilde{c}_\delta y_1^\delta = f(y_1 + a_1, x_2, \ldots, x_n)$$

or

$$f(x_1, \ldots, x_n) = \sum_{\delta=0}^{d} \tilde{c}_\delta (x_1 - a_1)^\delta.$$

Step BT (Back-Transformation). We compute $c_\delta \in F[x_2, \ldots, x_n]$, $0 \le \delta \le d$, from a_1 and the $\tilde{c}_\delta$ such that

$$f(x_1, \ldots, x_n) = \sum_{\delta=0}^{d} c_\delta x_1^\delta,$$

by a fast "radix conversion" method [28, sect. 4.4, exercise 14], which we briefly present as an algebraic RAM algorithm.

Split $f = f_0 + (x_1 - a_1)^{\lceil d/2 \rceil} f_1$ with

$$f_0 = \sum_{\delta=0}^{\lceil d/2 \rceil - 1} \tilde{c}_\delta (x_1 - a_1)^\delta, \qquad f_1 = \sum_{\delta=0}^{\lfloor d/2 \rfloor} \tilde{c}_{\delta + \lceil d/2 \rceil} (x_1 - a_1)^\delta.$$

Convert f_0 and f_1 by recursive application of the algorithm.
Compute $f_2 = (x_1 - a_1)^{\lceil d/2 \rceil}$ by repeated squaring.
Compute $f_1 f_2 + f_0$ by polynomial multiplication and addition.

The complexity $T(d)$ of this algorithm satisfies $T(d) \le 2T(d/2) + \gamma M(d)$, γ a constant, and hence is $T(d) = O(\log(d) M(d))$. We note that for coefficient fields F of characteristic 0 this conversion can be accomplished in even $O(M(d))$ arithmetic steps [2]. The program Q to be returned is built from $\tilde{Q}$ by appending instructions for the conversion. Therefore, $\text{len}(Q) = O(lM(d) + \log(d)M(d))$, but since $\log(d) = O(l)$, the first term already dominates the asymptotic output length. □

It should be clear that Theorem 5.1 applies to Algorithm Polynomial Coefficients 2 as well. It is somewhat surprising that we are not able to remove the restriction in the Taylor Series Coefficients Algorithm that P be defined at $\phi(x_1) = 0$ by a translation similar to the above. In fact, we know of no strategy of polynomial length in l that would compute $\mathrm{ldcf}_{x_1}(f)$, $f \in F[x_1, \ldots, x_n]$, but that does not depend polynomially on $\deg_{x_1}(f)$. Note that $\mathrm{ldcf}_{x_1}(f)$ is the constant Taylor series coefficient of

$$x_1^{\deg_{x_1}(f)} f\left(\frac{1}{x_1}, x_2, \ldots, x_n\right).$$

Although Algorithm Polynomial Coefficients 2 produces a program of asymptotically longer length than Algorithm Polynomial Coefficients 1, there exist circumstances under which the second algorithm is better in practice. One such situation arises when there are many input and scalar operands in the straight-line assignments. In our implementation of the coefficients algorithm [10], we therefore first estimate the length of the output program for either method and then perform the one that led to the smaller estimate. Length estimates for both algorithms can be computed quickly and fairly accurately, that is, within 1 percent of the actual lengths.

We finally remark how Algorithm Polynomial Coefficients 2 can be used to remove divisions from a program P_1 of length l for a polynomial $f \in F[x_1, \ldots, x_n]$ of degree d. First, we need $a_1, \ldots, a_n \in F$ such that P_1 is defined at $\phi(x_i) = a_i$, $1 \leq i \leq n$, as they are found in Step FT of the above algorithm. Then, we apply the Taylor Series Coefficient Algorithm to a straight-line program P_2 for the polynomial

$$g(z, y_1, \ldots, y_n) = f(y_1 z + a_1, \ldots, y_n z + a_n),$$

with respect to the variable z and to the order d. The point is now that the only divisions necessary are those to compute $\alpha_0 \leftarrow 1/\mathrm{sem}(w''_{\chi,0})$ in the Newton reciprocal procedure. It is easy to see that the constant terms in the Taylor expansions at $z = 0$ for the rational functions computed by the program variables in P_2 are, in fact, elements in F; that is, α_0 can be encoded as a new scalar. Our division-free resulting program Q_1 now computes $c_\delta \in F[y_1, \ldots, y_n]$ such that

$$g(z, y_1, \ldots, y_n) = \sum_{\delta=0}^{d} c_\delta(y_1, \ldots, y_n) z^\delta.$$

Putting the proper translations for $y_i = x_i - a_i$, $1 \leq i \leq n$, in front of Q_1 we obtain the division-free program Q_2 for f as

$$f(x_1, \ldots, x_n) = \sum_{\delta=0}^{d} c_\delta(x_1 - a_1, \ldots, x_n - a_n).$$

The length of Q_2 is $O(lM(d))$. It should be noted that this particular transformation cannot be carried out in binary random polynomial time, since the new scalars α_0 might be of exponential size, but other formulations without that drawback are, of course, possible. We also note that the coefficient of z^δ in $f(x_1 z, \ldots, x_n z)$ is the homogeneous part of degree δ in $f(x_1, \ldots, x_n)$. Strassen [43] describes this method with the homogeneous parts taking the place of coefficients, but then the computation of the reciprocal by Newton iteration needs some extra thought. For the record, let us state the following theorem.

THEOREM 7.1. *Let $f \in F[x_1, \ldots, x_n]$ be given by a straight-line program P of length l over $F(x_1, \ldots, x_n)$, F a field with card$(F) \geq 2^{l+1}$. There exists a universal* *constant γ and a division-free straight-line program $Q = (\{x_1, \ldots, x_n\}, V_Q, C_Q, S_Q)$ such that*

$$f \in sem(Q), \qquad S_Q \subset F, \qquad and \qquad len(Q) \leq \gamma l M(deg(f)).$$

8. *Numerators and Denominators of Rational Functions*

In this section we describe an algorithm that transforms a straight-line computation for a rational function of known degree to one for its (reduced) numerator and denominator. A major application of this algorithm will be to parallelize computations for rational functions (cf. Corollary 8.3). But first we review some needed properties of Padé approximants. However, we do not prove any of these properties and instead refer to [4] for an in-depth discussion and the references to the literature. Let

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots \in F[[x]], \qquad c_0 \neq 0, \quad d, e \geq 0,$$

be given. Going back to Frobenius (1881), a rational function $p(x)/q(x)$ is called a (d, e)-Padé approximant to f if

$$\begin{aligned} \deg(p) \leq d, \qquad \deg(q) \leq e, \\ f(x)q(x) - p(x) \equiv 0 \bmod x^{d+e+1}. \end{aligned} \tag{10}$$

It turns out that, for any pair (d, e), there always exists a solution to (10), and, furthermore, that the ratio p/q is unique. This ratio forms the entry in row d and column e of an infinite matrix referred to as Padé table. Kronecker (1881) had already realized that the entries in the $d + e$ antidiagonal of the Padé table are closely related to the Euclidean remainder sequence of

$$f_{-1}(x) = x^{d+e+1}, \qquad f_0(x) = c_0 + c_1 x + \cdots + c_{d+e} x^{d+e}.$$

Consider the extended Euclidean scheme [28, sect. 4.6.1, exercise 3]

$$\begin{aligned} s_i(x) f_{-1}(x) + t_i(x) f_0(x) = f_i(x), \\ f_i(x) = f_{i-2}(x) \bmod f_{i-1}(x), \qquad i \geq 1. \end{aligned}$$

Then for the smallest index i with $\deg(f_i) \leq d$ we have $\deg(t_i) \leq e$, and f_i/t_i is the (d, e)-Padé approximant to f. Furthermore, $\mathrm{GCD}(f_i, t_i) = x^k$ for some $k \geq 0$. Thus, any algorithm for computing the extended Euclidean scheme results in one for the (d, e)-Padé approximant. Note that the assumption $c_0 \neq 0$ is unessential by changing the lower bound for d.

The classical Euclidean algorithm gives a $O((d + e)^2)$ method for computing the (d, e)-Padé approximant. The ingenious algorithm by Knuth [27], which was improved by Schönhage [38] and applied to polynomial GCDs by Moenck [33], allows us to compute the triple (f_i, s_i, t_i) with $\deg(f_i) \leq d$, $\deg(f_{i-1}) > d$, in $O(M(d + e)\log(d + e))$ operations in F.

We are now prepared to describe the algorithm. The key idea is that by substituting $x_\nu + b_\nu x_1$ for x_ν, $2 \leq \nu \leq n$, we can make the problem a univariate problem in x_1 over the field $F(x_2, \ldots, x_n)$, as was already done in the Polynomial GCD Algorithm. We then recover the fraction from its Taylor series approximation by computing the Padé approximant in $F(x_2, \ldots, x_n)[[x_1]]$. Since that approximant is unique, it must be the reduced numerator and denominator.

Algorithm *Rational Numerator and Denominator*

Input: A straight-line program P over $F(x_1, \dots, x_n)$ of length l that computes f/g, $f, g \in F[x_1, \dots, x_n]$ relatively prime, and $d \geq \deg(f)$, $e \geq \deg(g)$, and a failure allowance $\epsilon \ll 1$. We make the assumption that $d, e \leq 2^l$, since the latter is always a bound.

Output: Either "failure" (with probability $< \epsilon$), or a straight-line program Q over $F(x_1, \dots, x_n)$ of length $O(lM(d+e))$ such that Q computes f and g correctly with probability greater than $1 - \epsilon$.

Step FT (Forward Translation). From a set $R \subset F$ with

$$\operatorname{card}(R) > \frac{2^{(2\gamma_3+2)lM(d+e)}}{\epsilon},$$

select random elements $a_1, \dots, a_n, b_2, \dots, b_n$. In this case constant γ_3 depends on the polynomial multiplication algorithm used and can be computed once an algorithm is selected. If F is a finite field with too small a cardinality, we can work in an algebraic extension of F instead. Since the results can be computed by rational operations in F, they remain invariant with respect to field extensions.

Test whether P is defined at $\phi(x_\nu) = a_\nu$, $1 \leq \nu \leq n$. For $F = \mathbf{Q}$, we call Algorithm Zero-Division Test in Section 3, such that the probability of "failure" occurring even if P is defined at ϕ is less than ϵ. If in this test P turns out to be (probably) undefined at ϕ, we return "failure".

Now we translate the inputs of P as $x_1 \leftarrow y_1 + a_1$, $x_\nu \leftarrow y_\nu + b_\nu y_1$, $2 \leq \nu \leq n$. Let $\bar{P}$ be the straight-line program computing $\bar{f}/\bar{g}$ where

$$\bar{h}(y_1, \dots, y_n) = h(y_1 + a_1, y_2 + b_2 y_1, \dots, y_n + b_n y_1) \qquad \text{for} \quad h \in F[x_1, \dots, x_n].$$

Now $\bar{P}$ is defined at $\phi(y_1) = 0$. Also with high probability

$$\deg_{y_1}(\bar{f}) = \deg(f), \qquad \text{implying that} \quad \operatorname{ldcf}_{y_1}(\bar{f}) \in F. \tag{11}$$

Step S (Power Series Approximations). Compute a straight-line program Q_1 over $F(y_2, \dots, y_n)$ such that for the coefficients of the power series

$$\frac{\bar{f}}{\bar{g}} = c_0(y_2, \dots, y_n) + c_1(y_2, \dots, y_n)y_1 + \cdots + c_{d+e}(y_2, \dots, y_n)y_1^{d+e} + \cdots, \tag{12}$$

the c_i are computed by Q_1 for all $0 \leq i \leq d + e$. This can be done by directly applying the Taylor Series Coefficients Algorithm in Section 7. Notice that $\operatorname{len}(Q_1) \leq \gamma_1 lM(d + e)$, where γ_1 is a constant solely depending on the multiplication algorithm used.

Step P (Padé Approximation). Construct a straight-line program Q_2 over $F(y_2, \dots, y_n)$ that with high probability computes the (d, e)-Padé approximation p/q to (12), $p, q \in F(y_2, \dots, y_n)[y_1]$. From the preceding remarks, we know that this can be accomplished by an extended Euclidean algorithm. Essentially, we perform such an algorithm on the coefficient vectors $(c_i)_{0 \leq i \leq d+e}$ and that of y_1^{d+e+1}. In order to test elements in $F(y_2, \dots, y_n)$ for zero we evaluate the program computing these elements at $\psi(y_\nu) = a_\nu$, $2 \leq \nu \leq n$, as we do in Theorem 4.2 or step D in the Polynomial GCD algorithm. If we use the asymptotically faster

Knuth–Schönhage procedure (see also [9] for a full description of the algorithm), then

$$\mathrm{len}(Q_2) \le \gamma_1 l M(d+e) + \gamma_2 M(d+e)\log(d+e) \le \gamma_3 l M(d+e), \qquad (13)$$

where γ_2 and γ_3 are again constants solely depending on the polynomial multiplication procedure used. Notice that the produced straight-line program may be incorrect (that with small probability), since we may have incorrectly certified an element to be zero.

Once we have a straight-line program for polynomials f_i and $t_i \in F(y_2, \ldots, y_n)[y_1]$ in the extended Euclidean scheme, we must find $k \ge 0$ such that $\mathrm{GCD}(f_i, t_i) = y_1^k$ over $F(y_2, \ldots, y_n)[y_1]$. This we can again accomplish probabilistically by evaluating the coefficients in y_1 of f_i and t_i.

If we make $\mathrm{ldcf}_{y_1}(p) = 1$, then with high probability p is an associate of $\bar{f}$ in $F[y_1, \ldots, y_n]$. This is because of (11) and because Padé approximants are unique.

Step BT (Back-translate). The program Q is obtained by putting assignments for the back-translations

$$y_1 \leftarrow x_1 - a_1, \qquad y_\nu \leftarrow x_\nu - b_\nu(x_1 - a_1), \qquad 2 \le \nu \le n,$$

in front of Q_2. □

We now analyze the overall failure probability of the Rational Numerator and Denominator Algorithm. "Failure" is only returned if P is not defined or is not recognized to be defined at ϕ. However, several events must take place in order that the correct answer is returned. First, $\mathrm{ldcf}_{y_1}(\bar{f}) \in F$ that justifies the normalization of p in step P. By Lemma 6.1, this happens with probability greater than or equal to

$$1 - \frac{d}{\mathrm{card}(R)} > 1 - \frac{\epsilon}{4}.$$

Second, all zero-tests performed by evaluating at $\psi(y_\nu) = a_\nu$, $2 \le \nu \le n$, must give the correct answer. This is true if the Knuth–Schönhage algorithm performed over $F(y_2, \ldots, y_n)$ takes the same course as the algorithm performed over F on the elements obtained by evaluating at ψ. In other words, no nonzero element that is tested or by which is divided must evaluate to 0. Since the algorithm takes no more than

$$\gamma_2 M(d+e)\log(d+e)$$

steps, the degree of any unreduced numerator and denominator of these elements is, by (13), no more than

$$2^{\gamma_3 l M(d+e)}.$$

A (pessimistic) estimate for the number of elements to be tested and to be divided by, including determination of k, is

$$\gamma_3 l M(d+e) + (d+e) < (\gamma_3 + 1) l M(d+e).$$

Therefore, the probability that all tests lead to the same result at ψ and that all divisions are possible at ψ is no less than

$$1 - \frac{(\gamma_3 + 1) l M(d+e) 2^{\gamma_3 l M(d+e)}}{\mathrm{card}(R)} > 1 - \frac{\epsilon}{2}.$$

Hence, a correct program Q is output with probability greater than $1 - \frac{3}{4}\epsilon$.

In case $F = \mathbf{Q}$ one additional possibility of returning an incorrect result must be accounted for, namely, that the Zero Test Algorithm in Section 3 might not recognize a nonzero evaluation at ψ properly. However, the probability of such an event can be controlled—say we allow it to happen only with probability no greater than

$$\frac{\epsilon}{4(\gamma_3 + 1)M(d + e)\log(d + e)}.$$

Then all tests succeed with probability greater than $1 - \epsilon/4$, and a correct program is output with probability greater than $1 - \epsilon$. In summary, we have the following theorem:

THEOREM 8.1. *Algorithm Rational Numerator and Denominator does not fail and outputs a program Q that computes f and g with probability greater than $1 - 2\epsilon$. It requires polynomially many arithmetic steps as a function of len(P), d, and e. For $F = \mathbf{Q}$ this is also true for its binary complexity, which also depends on el-size(P). The algorithm needs polynomially many randomly selected field elements (bits for $F = \mathbf{Q}$).*

We now formulate three corollaries to the theorem. The first corollary deals with distinguishing straight-line programs that compute polynomials from those that do not. It is clear that, if we have the bounds d and e, we only need to probabilistically inspect the degree of g after we have a straight-line program for it. But what if we do not have a priori degree bounds? We then run our algorithm for

$$d = e = 2^k, \qquad k = 1, 2, 3, \ldots.$$

Let f_k and g_k be the numerator and denominator whose computation is produced. For randomly chosen $a_1, \ldots, a_n \in F$, we then probabilistically test whether

$$\frac{f}{g}(a_1, \ldots, a_n) = \frac{f_k(a_1, \ldots, a_n)}{g_k(a_1, \ldots, a_n)}.$$

If the test is positive, with high probability $f = f_k$ and $g = g_k$. We have the following corollary.

COROLLARY 8.1. *Let f/g be given by a straight-line program P over $F(x_1, \ldots, x_n)$. Then we can in random polynomial time in len(P) and deg(fg) determine deg(f) and deg(g) such that with probability greater than $1 - \epsilon$ no failure occurs and the answer is correct. In particular, we can decide whether $f/g \in F[x_1, \ldots, x_n]$.*

For simplicity we state the next corollaries over infinite fields, although this can be avoided, as mentioned in step D. The next one resolves Strassen's question on computing the numerator and denominator of a rational function without divisions. By

$$L_D(r_1, \ldots, r_m \mid s_1, \ldots, s_n), \qquad r_i, s_j \in D,$$

we denote the nonscalar or total complexity of computing r_i from s_j over D; see, for example, [43].

COROLLARY 8.2. *Let F be an infinite field. Then*

$$L_{F[x_1,\ldots,x_n]}(f, g \mid x_1, \ldots, x_n) = O\left(M(deg(fg))^2 L_{F(x_1,\ldots,x_n)}\left(\frac{f}{g} \mid x_1, \ldots, x_n\right)\right),$$

where $M(d)$ is the corresponding complexity of multiplying d-degree polynomials. In the nonscalar case $M(d) = O(d)$.

The third corollary concerns the parallelization of a straight-line computation for a rational function. From [46], we get

COROLLARY 8.3. *Let P be a straight-line program of length l over $F(x_1, \ldots, x_n)$, F infinite, that computes f/g where $deg(f)$, $deg(g) \leq d$. Then there exists a straight-line program Q of parallel depth $O(log(d)log(dl))$ and $size(ld)^{O(1)}$ that also computes f/g.*

There is an open problem related to this corollary. The question is whether there is a parallel algorithm that takes P as input and evaluates it at given points. For division-free input programs such an algorithm has been constructed [31]. For formulas as inputs, divisions do not cause additional difficulty [32]. However, the proof of the above corollary is tied to knowing the evaluation of the input program at a random point, and we do not know how the methods in [31] and [32] can be used to solve the problem.

Finally, we remark that, if instead of degree bounds the exact degree $d = \deg(f)$, $e = \deg(g)$ are input, the Numerator and Denominator Algorithm can be made "Las Vegas"; that is, if it does not fail the returned program Q will be guaranteed to be correct. An obvious condition to check would be whether $\deg_{y_1}(p) = d$ and $\deg_{y_1}(q) = e$, where p/q is the reduced Padé approximation to (12). However, this check is not sufficient, since, during the extended Euclidean algorithm, a leading coefficient of a remainder might have been dropped owing to incorrect zero testing, with the resulting incorrect quotient still satisfying the degree requirements. Instead, we compute p and q by setting up a linear system with undetermined coefficients for (10), that is,

$$(c_0 + \cdots + c_{d+e}y_1^{d+e})(1 + q_1 y_1 + \cdots + q_e y_1^e) - (p_0 + \cdots + p_d y_1^d)$$
$$\equiv 0 \bmod y_1^{d+e+1}.$$

If the a_ν are selected such that $c_0 \neq 0$ in (12), which can be verified by random evaluation, then the above system has a solution with

$$p_d(y_2, \ldots, y_n),\ q_e(y_2, \ldots, y_n) \neq 0$$

if and only if (11) is satisfied. In that case the linear system that arises is nonsingular, which can be verified, and a straight-line program for its solution can be deterministically constructed. It then remains to verify the just-mentioned nonzero conditions for p_d and q_e by random evaluation to make sure that (11) has been satisfied.

9. *Conclusion*

We have formulated our arithmetic complexities for arbitrary fields and our binary complexities for finite fields and the rational numbers. It is not difficult to extend the polynomial-time results to algebraic number fields. The main obstacle to binary polynomial-time complexity is the need for zero and zero-division testing. It should be clear that the corresponding algorithms generalize by working modulo a randomly chosen prime ideal. A more straightforward approach to evaluating straight-line programs over algebraic number fields can also be found in [11].

Straight-line results can be useful for further manipulation, but as the final result they are quite incomprehensible. Fortunately, there is always the possibility of probabilistically converting them to sparse representation. Zippel's algorithm [47] can be shown to accomplish this conversion in expected polynomial time in the input size, degree, and the number of nonzero monomials of the sparse answer [21, sect. 6]. In another formulation, given a bound t, one can probabilistically determine in polynomial time in t either the sparse representation of a polynomial with no more than t monomials given by a straight-line program, or, with controlably high probability, that a polynomial has more than t nonzero monomials [23]. Since sparse inputs can always be represented as straight-line programs of polynomially related size, by the conversion algorithm all our results apply to sparse polynomials as well. For example, we have a random polynomial-time algorithm for computing the sparse greatest common divisor of sparse polynomials.

This work began as the pilot for a series that consists of four papers. Our second paper [23] shows how to compute in random polynomial-time the full factorization of a polynomial, with input and outputs in straight-line representation. As mentioned before, that paper also contains a discussion on the sparse conversion question. We also refer to [24] for a detailed outline of the main results of the factoring paper. Our third and most recent article [25] discusses an approach to replacing the input degree bound d in the Polynomial GCD Algorithm, for instance, by a degree bound for the output polynomial. Also in that paper a completely different proof for Corollary 8.2, based on the factorization results, is given. Although it appears that our results are already of theoretical significance, we believe that the straight-line representation of multivariate polynomials is an important tool in computer algebra systems. Therefore, we have implemented our algorithms in LISP with an interface to MACSYMA. The details of this first implementation and our experience with test cases are reported in the fourth paper of this series [10].

Note Added in Proof. Theorem 6.2 remains valid if we replace $\sum_{i=1}^{r} b_i f_i$ by f_1, which is a slight improvement in the length of the generated straight-line program. In step P of Algorithm Rational Numerator and Denominator the computation of $\mathrm{GCD}(f_i, t_i) = y_1^k$ can be skipped, since it can be shown that under the given circumstances one always has $k = 0$. Finally, with B. Trager we have found a different solution to the numerator and denominator problem, such that the length of the produced program is $O(l\,d + M(d)\log(d))$.

ACKNOWLEDGMENTS. I thank my student collaborators Timothy Freeman, Brent Gregory, and Gregory Imirzian for their contributions to the entire project. This work was stimulated by the DMV seminar "The Complexity of Algebraic and Numerical Problems," given by Joachim von zur Gathen and Volker Strassen in September 1984. I thank both for inviting me to that workshop. A conversation with Allan Borodin led to the construction of the Rational Numerator and Denominator Algorithm. Wayne Eberly pointed out the Las Vegas result discussed in the last paragraph of Section 8. I also thank both of them for their observations.

REFERENCES

1. Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. Aho, A. V., Steiglitz, K., and Ullman, J. D. Evaluating polynomials at fixed sets of points. *SIAM J. Comput. 4* (1975), 533–539.
3. Baur, W., and Strassen, V. The complexity of partial derivatives. *Theoret. Comput. Sci. 22* (1983), 317–330.
4. Brent, R. P., Gustavson, F. G., and Yun, D. Y. Y. Fast solution of Toeplitz systems of equations and computation of Padé approximants. *J. Algorithms 1* (1980), 259–295.

5. Brown, W. S. On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM 18*, 4 (Oct. 1971), 478–504.
6. Bunch, J. R., and Hopcroft, J. E. Triangular factorization and inversion by fast matrix multiplication. *Math. Comput. 28* (1974), 231–236.
7. Char, B. W., Geddes, K. O, and Gonnet, G. H. GCDHEU: Heuristic polynomial GCD algorithm based on integer GCD computation. In *Proceedings of EUROSAM '84.* Vol. 174, Lecture Notes in Computer Science. Springer-Verlag, New York, 1984, pp. 285–296. To appear in *J. Symbolic Comp.*
8. Collins, G. E. Subresultants and reduced polynomial remainder sequences. *J. ACM 14* (1967), 128–142.
9. Czapor, S. R., and Geddes, K. O. A comparison of algorithms for the symbolic computation of Padê approximants. In *Proceedings of EUROSAM '84.* Vol. 174, Lecture Notes in Computer Science. Springer-Verlag, New York, 1984, pp. 248–259.
10. Freeman, T. S., Imirzian, G., and Kaltofen, E. A system for manipulating polynomials given by straight-line programs. In *Proceedings of the 1986 ACM Symposium on Symbolic Algebraic Computing.* ACM, New York, 1986, pp. 169–175.
11. Gathen, J. von zur. Irreducibility of multivariate polynomials. *J. Comput. Syst. Sci. 31* (1985), 225–264.
12. Gathen, J. von zur. Parallel arithmetic computation: A survey. In *Proceedings of MFCS '86.* Vol. 233, Lecture Notes in Computer Science, Springer-Verlag, New York, 1986, pp. 93–112.
13. Gathen, J. von zur, and Kaltofen, E. Factoring sparse multivariate polynomials. *J. Comput. Syst. Sci. 31* (1985), 265–287.
14. Hardy, G. H., and Wright, E. M. *An Introduction to the Theory of Numbers.* Oxford University Press, Oxford, England, 1979.
15. Heintz, J. A note on polynomials with symmetric Galois group which are easy to compute. *Theoret. Comput. Sci. 47* (1986), 99–105.
16. Heintz, J., and Schnorr, C. P. Testing polynomials which are easy to compute. In *Monographie de L'Enseignement Mathématique,* vol. 30. Imprimerie Kundig, Genève, Switzerland, 1982, pp. 237–254.
17. Hyafil, L. On the parallel evaluation of multivariate polynomials. *SIAM J. Comput. 8* (1979), 120–123.
18. Ibarra, O. H., and Moran, S. Probabilistic algorithms for deciding equivalence of straight-line programs. *J. ACM 30,* 1 (Jan. 1983), 217–228.
19. Ibarra, O. H., Moran, S., and Rosier, L. E. Probabilistic algorithms and straight-line programs for some rank decision problems. *Inf. Process. Lett. 12* (1981), 227–232.
20. Jenks, R. D. A primer: 11 keys to new SCRATCHPAD. In *Proceedings of EUROSAM '84.* Vol. 174, Lecture Notes in Computer Science, Springer-Verlag, New York, 1984, pp. 123–147.
21. Kaltofen, E. Computing with polynomials given by straight-line programs. I. Greatest common divisors. In *Proceedings of the 17th ACM Symposium on Theory of Computing* (Providence, R.I., May 6–8). ACM, New York, 1985, pp. 131–142.
22. Kaltofen, E. Computing with polynomials given by straight-line programs. II. Sparse factorization. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1985, pp. 451–458.
23. Kaltofen, E. Factorization of polynomials given by straight-line programs. In *Randomness in Computation: Advances in Computing Research,* S. Micali, Ed. JAI Press, Greenwich, Conn., Jan. 1987.
24. Kaltofen, E. Uniform closure properties of *p*-compatible functions. In *Proceedings of the 18th ACM Symposium on Theory of Computing* (Berkeley, Calif., May 28–30). ACM, New York, 1986, pp. 330–337.
25. Kaltofen, E. Single-factor Hensel lifting and its application to the straight-line complexity of certain polynomials. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (New York, N.Y., May 25–27). ACM, New York, pp. 443–452.
26. Kaltofen, E., and Rolletschek, H. Arithmetic in quadratic fields with unique factorization. In *Proceedings of EUROCAL '85,* Vol. 2. Vol. 204, Lecture Notes in Computer Science, Springer-Verlag, New York, 1985, pp. 279–288.
27. Knuth, D. E. The analysis of algorithms. *Actes du congrès international des Mathématiciens 3* (1970), 269–274.
28. Knuth, D. E. *The Art of Programming.* Vol. 2, *Semi-Numerical Algorithms,* 2nd ed. Addison-Wesley, Reading, Mass., 1981.
29. Krishnamurthy, E. V., Rao, T. M., and Subramanian, K. Finite segment *p*-adic number systems with applications to exact computation. *Proc. Indian Acad. Sci. 81,* sec. A, 2 (1975), 58–79.

30. KUNG, H. T. On computing reciprocals of power series. *Numer. Math. 22* (1974), 341–348.
31. MILLER, G. L., RAMACHANDRAN, V., AND KALTOFEN, E. Efficient parallel evaluation of straight-line code and arithmetic circuits. In *Proceedings of the Aegian Workshop on Computing '86.* Vol. 227, Lecture Notes on Computer Science. Springer-Verlag, New York, 1986, pp. 236–245.
32. MILLER, G. F., AND REIF, J. H. Parallel tree contraction and its application. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1985, pp. 478–489.
33. MOENCK, R. T. Fast computation of GCDs. In *Proceedings of the 5th ACM Symposium on Theory of Computing* (Austin, Tex., Apr. 30–May 2). ACM, New York, 1973, pp. 142–151.
34. MOSES, J. AND YUN, D. Y. Y. The EZ-GCD algorithm. In *Proceedings of the 1973 ACM National Conference.* ACM, New York, 1973, pp. 159–166.
35. PLAISTED, D. A. Sparse complex polynomials and polynomial reducibility. *J. Comput. Syst. Sci. 14* (1977), pp. 210–221.
36. ROSSER, J. B., AND SCHOENFELD, L. Approximate formulas of some functions of prime numbers. *Ill. J. Math. 6* (1962), 64–94.
37. SCHWARTZ, J. T. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM 27*, 4 (Oct. 1980), 701–717.
38. SCHÖNHAGE, A. Schnelle Kettenbruchentwicklungen. *Acta Inf. 1* (1971), 139–144 (in German).
39. SCHÖNHAGE, A. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf. 7* (1977), 395–398 (in German).
40. SOLOVAY, R. M., AND STRASSEN, V. A fast Monte-Carlo test for primality. *SIAM J. Comput. 6* (1977), 84–85. Correction: *7* (1978), 118.
41. STOUTEMEYER, D. R. Which polynomial representation is best? In *Proceedings of the 3rd MACSYMA Users' Conference*, General Electric, Schenectady, N.Y., 1984, pp. 221–243.
42. STRASSEN, V. Berechnung und Programm I. *Acta Inf. 1* (1972), 320–335 (in German).
43. STRASSEN, V. Vermeidung von Divisionen. *J. Reine u. Angew. Math. 264* (1973), 182–202 (in German).
44. VALIANT, L. The complexity of computing the permanent. *Theoret. Comput. Sci. 8* (1979), 189–201.
45. VALIANT, L. Reducibility by algebraic projections. *L'Enseignement Math. 28* (1982), 253–268.
46. VALIANT, L., SKYUM, S., BERKOWITZ, S., AND RACKOFF, C. Fast parallel computation of polynomials using few processors. *SIAM J. Comput. 12* (1983), 641–644.
47. ZIPPEL, R. E. Probabilistic algorithms for sparse polynomials. In *Proceedings of the EUROSAM '79.* Vol. 72, Lecture Notes on Computer Science, Springer-Verlag, New York, 1979, pp. 216–226.

RECEIVED JANUARY 1986; REVISED OCTOBER 1986, FEBRUARY 1987; ACCEPTED MAY 1987