Exact Geometric Computation: Theory and Applications

by

Chen Li

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2001

Approved: ____________________

Research Advisor: Professor Chee Yap

To my parents

Acknowledgment

First of all, I wish to express my profound gratitude to my advisor, Professor Chee Yap. Without his encouragement, support and patient guidance, most of this would not have been possible. He has always been available and enthusiastic to help when I need it during the last four years. His insightful advice laid the foundation of my work.

I am very grateful to Professor Marsha Berger and Professor Vijay Karamcheti for their continuous advice and support throughout my thesis research. I would also like to thank my thesis committee members Professor Michael Overton and Professor Edmond Schonberg for their careful reading and suggestions. In addition, I thank my research collaborators, Shankar Krishnan, Jose E. Moreira, Igor Pechtchanski and Daniela Tulone, for the inspiration and help they gave me.

My graduate school years cannot be a truly pleasant and memorable one without all my friends here. I salute them all, although I can only mention a few here. Among them are Ee-Chien Chang, Fangzhe Chang, Hseu-ming Chen, Xianghui Duan, Dongning Guo, Bin Li, Ninghui Li, Zhijun Liu, Madhu Nayakkankuppam, Xueying Qin, Xiaoping Tang, Zhe Yang, Ting-jen Yen and Dewang Zhu.

My biggest gratitude goes to my family. I thank my sisters Qing Li and Jie Li for their support and faith in me. This dissertation is dedicated to my parents, Shujiang Chen and Dr. Hongzhong Li, for their love and encouragement that I can always count on.

Abstract

This dissertation explores the theory and applications of Exact Geometric Computation (EGC), a general approach to robust geometric computing. The contributions of this thesis are organized into three parts.

A fundamental task in EGC is to support exact comparison of algebraic expressions. This leads to the problem of constructive root bounds for algebraic expressions. Such root bounds determine the worst-case complexity of exact comparisons. In the first part, we present a new constructive root bound which, compared to previous bounds, can give dramatically better performance in many common computations involving divisions and radical roots. We also improve the well-known degree-measure bound by exploiting the sharing of common sub-expressions.

In the second part, we discuss the design and implementation of the Core Library, a C++ library which embraces the EGC approach to robust numerical and geometric computation. Our design emphasizes ease of use and facilitates the rapid development of robust geometric applications. It allows non-specialist programmers to add robustness into new or existing applications with little extra effort. A number of efficiency and implementation issues are investigated.

Although focused on geometric computation, the EGC techniques and software we developed can be applied to other areas where it is critical to guarantee numerical precision. In the third part, we introduce a new randomized test for the vanishing of multivariate radical expressions. With this test, we develop a probabilistic approach to proving elementary geometry theorems about ruler-and-compass constructions. A probabilistic theorem prover based on this approach has been implemented using the Core Library. We present some empirical data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Numerical non-robustness problems associated with the approximate floating-point arithmetic are very common in practice when implementing geometric algorithms. Generally geometric computation is very vulnerable to numerical errors. There has been considerable recent interest in robust implementation of geometric algorithms. A general framework to achieve geometric robustness is Exact Geometric Computation (EGC). This thesis is aimed to make EGC more efficient and easily accessible to non-specialist programmers. We present some theoretical results on constructive root bounds for algebraic expressions. An EGC software library, the Core Library, has been developed. We apply the Core Library in proving elementary geometry theorems about ruler-and-compass constructions probabilistically based on a novel randomized zero test of multivariate radical expressions.

1.1 Geometric Computing and Exactness

Computational Geometry investigates algorithms for geometric problems. Geometric computing is different than numerical computation in that it involves both combinatorial and numerical information. The consistency between combinatorial and numerical data should be maintained.

Geometric algorithms are usually designed under a *Real RAM model of computation*, in which it is assumed that real numbers can be represented exactly and all arithmetic operations and comparisons within the real number field $\mathbb{R}$ can be performed *exactly* in unit time. Although this assumption is reasonable for the asymptotic analysis of the complexity of problems and has indeed enabled theoretical research to flourish, unfortunately in general it does not hold in practical situations where the approximate floating-point arithmetic is widely used as a standard substitute for the assumed (exact) real arithmetic. The Real RAM model understates a problem's complexity in more realistic computation models. Theoretically (say, in a Real RAM model) the consistency between numerical and combinatorial data can be guaranteed by the correctness of underlying algorithms. But when using floating-point numbers, it is common that numerical errors introduced can violate the consistency. As a result, the implementation of geometric algorithms is particularly challenging in practice. This is an important reason for the fact that the rate of technology transfer lags much behind the growth of theoretical results in computation geometry. Recently, there has been considerable interest in robust geometric computation [20, 54, 36, 7, 57, 49].

Floating-point number systems (such as the IEEE 754 standard [27]), although naively a reasonable approach to real number arithmetic, have serious shortcomings [20, 16]. Straight-forward implementation of geometric algorithms using floating-point numbers could easily introduce some undesirable numerical errors. These errors can accumulate and propagate throughout the whole application. It is often difficult to predict the occurrence and magnitude of these errors. Even worse, such numerical errors, however tiny, could trigger much more serious inconsistency problems between the numerical and combinatorial data. Sometimes even if the consistency is maintained, the combinatorial structures involved may still be incorrect (e.g., when they are only consistent with some numerically “perturbed” configurations). These problems could well confuse the subsequent steps in a program execution and make software non-robust. Without a general solution, it is tedious, if not impossible, to deal with them. This problem has received deserved attention in the Computational Geometry community since the 1980s (see [54, 36, 7]).

Yap [59] suggests that exactness in geometric computing does not necessarily mean that numerical values must be represented and computed exactly. Instead, “geometric exactness” means that the conditional tests, which determine the control flow of a program, must be handled in a mathematically correct way. Basically, this will guarantee the correctness of combinatorial structure involved in a computation. Nevertheless, this also presents a significant relaxation from the conventional concept of numerical exactness. First, it frees us from keeping and computing exact numerical values which sometimes is not feasible (e.g. when irrational numbers such as $\sqrt{2}$ are involved). It also suggests why the naive use of exact integer or rational arithmetic is subject to great performance penalty. The reason is that the full numerical accuracy is not always needed. Secondly, it implies the solution should be driven by the precision actually needed in critical conditional tests. This leads to the development of a number of techniques that can improve performance, such as precision-driven computation, lazy evaluation, adaptive computation and floating-point filters.

1.2 Previous Work

There is a significant amount of literature on the non-robustness problem in computational geometry. Various approaches have been proposed to attack this problem at the arithmetic, geometry or algorithm levels.

Exact Arithmetic Exact arithmetic supports exact computation with the four basic arithmetic operations $(+, -, \times, \div)$ over the rational field $\mathbb{Q}$. It represents big integer or rational numbers to arbitrary precision. Generally, the asymptotic complexity of each operation depends on the bit length of operands. For example, let us suppose both operands have n bits. The straight-forward "long multiplication" has a time complexity of $\mathcal{O}(n^2)$. Schönhage and Strassen discovered an $\mathcal{O}(n \lg n \lg \lg n)$ algorithm [50] based on FFT, which is the fastest multiplication algorithm so far under the Turing machine model. It has been shown that divisions can be performed within a speed comparable to that of multiplications, up to a constant factor [31]. Note that the bit length of operands could be increased drastically in cascaded computation (e.g., potentially doubled after each multiplication). Karasick, Lieber and Nackman [30] reported that the naive use of rational arithmetic in the divide-and-conquer algorithm for 2-D Delaunay triangulation costs a performance penalty of 10^4 over the floating-point implementation.

In evaluating integral polynomial expressions, *modular arithmetic* is an efficient alternative to exact integer arithmetic if the upper bound (or the possible range) of the result is known *a priori*. By carefully choosing the moduli, the multiplication can be done using $\mathcal{O}(n)$ primitive operations.

Generally, exact arithmetic discussed above requires software support. A number of big number packages have been developed, including the GNU's MP [24], the CLN package [22], the libI package [12], Arjen Lenstra's lip package [33] and LEDA Integer/Rational [25]. Although exact arithmetic eliminates the non-robustness by handling all numerical computations in a fully accurate way, the use of exact arithmetic is constrained by its huge cost, inability to handle irrational numbers, and most of all, the fact that it is not precision driven.

Tolerance based approaches Some approaches admit the existence of numerical errors and study the effects of such errors in predicate evaluation. ϵ-Tweaking is a simple and perhaps the most widely-used method to tackle numerical non-robustness problems, especially when the program in question is simple and the programmer has some knowledge about the input and intermediate values. Instead of comparing two expressions directly in a conditional test, the distance between them is compared against some small threshold value ϵ. The two expressions are considered equal if this distance is smaller than ϵ. Note that this effectively changes the geometry of objects (e.g., a line now becomes a pipe, etc.). Moreover, the relation defined by the rule

$$a \sim b \Leftrightarrow |a - b| < \epsilon$$

is not an equivalence relation because the transitivity does not hold (i.e., $a \sim b$ and $b \sim c$ do not imply $a \sim c$). It is non-trivial to guarantee the logical consistency among all the decisions made in a program execution using this strategy.

More sophisticated tolerance based approaches, such as the Epsilon Geometry introduced by Guibas et. al. [18], apply forward error analysis on selected geometric predicates to tell whether an answer is true, false or unknown (due to potential rounding errors in the predicate evaluation, etc.). Although no action is specified to take for the unknown case, such approaches nevertheless can serve as a filter for easy cases.

An arithmetic level approach to quantify possible numerical errors is *interval arithmetic* [43, 44, 1]. Given a real quantity x, it computes an interval $[x_l, x_u]$ such that $x_l \le x \le x_u$. A set of rules for interval operations is the following:

$$\begin{aligned}
a+b &\mapsto [a_l+b_l, a_u+b_u] \\
a-b &\mapsto [a_l-b_u, a_u-b_l] \\
a\times b &\mapsto [\min\{a_lb_l, a_lb_u, a_ub_l, a_ub_u\}, \max\{a_lb_l, a_lb_u, a_ub_l, a_ub_u\}] \\
a\div b &\mapsto [\min\{\frac{a_l}{b_l}, \frac{a_l}{b_u}, \frac{a_u}{b_l}, \frac{a_u}{b_u}\}, \max\{\frac{a_l}{b_l}, \frac{a_l}{b_u}, \frac{a_u}{b_l}, \frac{a_u}{b_u}\}], 0 \notin [b_l, b_u]
\end{aligned}$$

Taking only about twice as long as ordinary arithmetic, interval arithmetic provides truly reliable estimates for forward error analysis. The main drawback of interval arithmetic, especially when performed with fixed precision, is the rapid growth of interval size which can render a computation quickly ineffective.

Floating-point Filters It has been noted that the robustness of geometric computation usually depends on the correctness of critical conditional tests. Numerical errors from floating-point arithmetic can be tolerated as long as they do not compromise the outcome (e.g., true or false) of those critical tests. Considering the huge performance gap between exact computation and floating-point arithmetic (which has been supported by most current computer hardware), it is really important to activate exact computation only when it is *absolutely* necessary. The problem then is how to judge the reliability of a result from primitive floating-point operations.

An important concept here is filtering, which has been proved to be very effective in practice [15]. Basically, all the arithmetic computations are first performed using floating-point arithmetic, and then a floating-point filter can tell whether the results can be trusted or not in presence of potential rounding errors during the computation. For example, suppose a predicate is to determine the sign of an expression E. We first compute an approximate value $\tilde{E}$ using the floating-point arithmetic. At the same time, a filter computes an upper bound e on the accumulated numerical error through forward error analysis such that $|E - \tilde{E}| \leq e$. Therefore, if $|\tilde{E}| > e$, then E and $\tilde{E}$ must have the same sign. Otherwise, exact computation should be employed to determine the exact sign. The heuristic behind it is that in most cases, errors introduced by imprecise floating-point operations are not large enough to affect the signs. Thus, we should use the fast floating-point arithmetic whenever we can.

Shewchuk presents a fast adaptive-precision exact arithmetic method [52] based on floating-point computation. In his method, an arbitrary precision floating-point number is stored in the multiple-term format, as a sum of ordinary floating-point numbers. It first computes an approximation using the IEEE floating-point arithmetic. When needed, it can increase the precision of approximation by progressively producing chunks with an order of magnitude $\mathcal{O}(\epsilon^k)$ (for the k-th term) where ϵ is the machine epsilon defined in the IEEE standard [27]. However, breaking an expression evaluation into an adaptive form is not automated and could be a non-trivial task to non-specialists. His method is for integral expressions only which permit the addition, subtraction and multiplication operations. We are not aware of any extension of Shewchuk's paradigm to the division and square-root operations.

The error tracking in filters is usually based on some facts about the specification of the underlying floating-point arithmetic standard (such as IEEE 754). Depending on how an implementation divides the filtering task between the run time and the compilation time, there are three categories of filters: static [15], dynamic and semi-dynamic [3]. Most filtering techniques are at the arithmetic operation level. A recent paper by Pan and Yu [47] proposes a technique to certify the sign of determinants which suggests a promising direction in designing filters at the expression level so that the specific structure of certain expressions can be utilized.

Geometric Rounding and Perturbation Geometric rounding [17] converts algorithms and objects in the continuous domain to a uniform and discrete domain with finite resolution which simulates fixed precision representation and manipulation of real numbers. Although the conversion preserves certain critical topological constraints (e.g., intersection relationship between lines and orientation of an intersection point against a line, etc.), the topology of the scene may be changed (e.g. a line may become a polyline.). What constraints are critical to be kept is often problem-specific. Such techniques has been applied to solve the line arrangement problem robustly [17].

Perturbation approaches [19, 42] re-arrange the geometry of input objects to maintain some minimal distance between objects (i.e., in some sense, “well separated”) so that predicate evaluation can be correctly handled by comparing to some threshold value ϵ.

Generally geometric perturbation and rounding are algorithm-dependent and have only been applied to linear objects in low dimensions. For objects with high dimension, it is hard to confine the effect of a single positional perturbation within a local scope.

Design Robust Algorithms Fortune [14] classifies a geometry algorithm as *robust* if it can always produce a correct result under the Real RAM model, and under approximate arithmetic always produce an output which is consistent with some perturbation of the true input. If this perturbation is small, we call it *stable*.

It is helpful to consider robustness in designing algorithms. Some properties of algorithms (e.g. the degree of expression polynomial) affect the performance when translated and executed in a robust programming environment. Other important properties include numerical stability, degree of algebraic numbers involved and the redundancy among conditional tests. Liotta et al. [37] discussed a degree-driven algorithm design. Sugihara et al. [53] propose a complete separation of the combinatorial part and numerical part in an algorithm and emphasize the validity of combinatorial part of outputs.

1.3 Exact Geometric Computation (EGC)

Among the various approaches proposed to address the numerical non-robustness problem in computational geometry, Exact Geometric Computation (EGC) advocated by Yap [61, 60, 29] and others is very promising in that it can be applied directly to many geometric problems without requiring any special considerations and treatments specific to individual algorithms. EGC is a *general* framework to solve the numerical non-exactness problems in geometric computing. As we have discussed in Section 1.1, the non-exactness problem happens in predicate evaluation within a program. Most expressions in geometric computation are algebraic. Comparisons between algebraic expressions are common conditional tests in geometric programs. The EGC approach supports exact comparison of algebraic expressions.

Comparing two algebraic expressions can be reduced to determining the sign of algebraic algebraic numbers. In some simpler cases, it can be further reduced to testing whether an expression is zero or not. For example,

$$E_1 > E_2 \quad \Leftrightarrow \quad E_1 - E_2 > 0$$
$$E_1 = E_2 \quad \Leftrightarrow \quad E_1 - E_2 = 0.$$

Ideally we want the evaluation of such predicates to be carried out with complete accuracy in order to achieve geometric exactness. But the floating-point arithmetic used by most programmers is intrinsically approximate, and any pre-determined finite precision is not always sufficient for all applications. Instead of trying to compute the value exactly, usually resorting to big number packages, EGC focuses on *determining the sign of expressions correctly*. Determining the sign of a general real expression is hard (e.g., it is an open problem if the expression involves transcendental functions and/or π). However, the sign of an *algebraic expression* can be determined. We adopt a numerical approach based on algebraic root bounds. Basically we approximate the value of an expression numerically to sufficient precision until a positive or negative sign comes out or we know from root bounds that its value is really zero. Computation of root bounds usually depends on various algebraic attributes associated with that value (such as degree and length [62], etc.).

An EGC system presents users with a collection of number or expression types which substitute for the role of the primitive number types built in programming languages, such as `double` in C++. The operations over these types can *guarantee* any specified absolute or relative precision. Programmers can write robust codes by simply building their applications on these exact types as they do on primitive number types. Basically this set of exact types presents a *virtual* Real RAM machine for computation within the algebraic number field, and hence justifies the Real RAM assumption generally assumed in designing computational geometry algorithms (recall that this is exactly the reason for non-exactness problems). Unfortunately, the cost of arithmetic operations is no longer unit, neither is the cost for reading and writing a real number. The asymptotic time complexity depends on the bit complexity of operations. The numerical precision required in the worst cases is dominated by root bounds. Usually the actual running cost is much higher than that of the standard floating-point arithmetic. Hence, efficiency is an important problem here that needs to be studied.

Under EGC, it is usually assumed that the input is numerically accurate and valid (i.e., consistent with the topology as stated). Given such accurate and valid inputs, the EGC approach can *guarantee* the correctness of the combinatorial structure in the results. When necessary, this approach is also able to produce a result whose numerical part is consistent with the corresponding combinatorial structure and/or meets arbitrary relative or absolute precision requirements specified by users.

Since early 1990s, considerable research efforts have been made on this topic and some libraries based on this idea have been developed. Our Core Library [23, 29, 58, 35] provides a small numerical core consisting of a number of exact data types encapsulated by an easy-to-use interface which enables users to access different accuracy levels and to produce robust codes in a transparent manner. The Core Library project is based on a previous research effort, Real/Expr [61, 46], at NYU. A similar effort is LEDA [25], an ongoing project at the Max Planck Institute of Computer Science since 1988, which aims to provide an extensive range of robust data types and algorithms for combinatorial and geometric computing.

1.4 Our Contributions

The basic goal of my thesis research is to improve the efficiency of EGC and to make the EGC techniques easily accessible to all programmers.

The cost of the EGC approach is determined by two factors:

1. The root bound that decides the worst-case complexity of exact comparisons (e.g., when two expressions being compared are equal);

2. The algorithmic and system cost.

A basic problem in EGC is to find good root bounds. Although the root bound problem is an old problem in algebra which has been extensively studied with many classical results produced, most of these classical bounds assume that the (minimal) polynomial for an algebraic number is known. Unfortunately, computing minimal polynomials explicitly is very expensive in practice. In our work, we are especially interested in *constructive root bounds* which can be efficiently computed from the structure of an algebraic expression. We present a new constructive root bound [35] for general algebraic expressions by bounding the degree, the leading and tail coefficients of minimal polynomials, and some bounds on the conjugates as well. The basic tool we use is the resultant calculus in constructive polynomial algebra. We give a set of inductive rules to compute the root bound. We also improve the well-known degree-measure bound by exploiting the sharing of common radical sub-expressions. Among the existing constructive bounds that have been proposed, there is not a single one that is always better than others. We conduct some comparative study and experiments on different bounds and show that our new bound can give significant speedup over previous ones on some important classes of expressions.

From a user's point of view, an EGC system can be seen as a software data type library which forms a virtual Real RAM machine in which algebraic arithmetic operations and comparisons are performed exactly (though not necessarily of unit cost as in the original Real RAM model). In the implementation of an EGC system, there are many algorithmic and system problems that need to be addressed.

As a part of our research effort, we develop the Core Library [29, 58, 34, 23], an object-oriented C++ library for exact numeric and geometric computation. The Core Library provides a small, easy to use and efficient numerical core (in the form of a collection of C++ classes) that can meet arbitrary absolute or relative precision specified by users on numerical computation. In particular, it supports exact comparison of algebraic expressions. Our implementation embodies the precision-driven approach to EGC. Our design emphasizes the ease of use and compatibility. The concise API interface is compatible with that of primitive types. We give a transparent delivering mechanism to allow users to access different accuracy levels simultaneously according to their real needs. The library makes it very easy to write new robust codes or to inject robustness into existing applications. It facilitates rapid development of robust software by non-specialists since no special knowledge of non-robustness issues is required. In this thesis, we will use the Core Library as an example and testbed in illustrating various EGC techniques and applications we have developed.

We propose a probabilistic test of the vanishing of multivariate radical expressions [55] by extending Schwartz's well-known probabilistic test [51] on the vanishing of polynomials. The method chooses random instances from a finite test set with proper size, and tests the vanishing of an expression on these examples. Here we apply the Core Library in determining the exact sign in each instance test. Moreover, we apply this zero test in automated proving of elementary geometry theorems about ruler-and-compass constructions. A probabilistic prover based on the Core Library has been developed. We note that the zero test of radical expressions is an important problem by itself and has independent interest in other areas besides theorem proving.

We now summarize the contribution of this thesis:

- Present a new constructive root bound for algebraic expressions which can give significant improvement over existing bounds in many common computations involving division and root extraction operations. Improve the well-known degree-measure bound. Some experimental results are given.

- Develop the Core Library, a C++ library for robust numeric and geometric computation that embodies our precision-driven approach to EGC. Various design and implementation issues are investigated.

- Present a probabilistic zero test for multivariate radical expressions. In particular, following our previous work in [55], we give a new definition of *rational degrees* which not only simplifies the derivation of an upper bound on the cardinality of the finite test set, but also leads to a more efficient method to compute this bound. Moreover, we improve this bound for expressions with divisions. Based on this zero test of radical expressions, we apply the EGC techniques and the Core Library to prove elementary geometry theorems about ruler-and-compass constructions.

Acknowledgments The work on new root bounds is joint with Chee Yap [35]. The Core Library project is a joint research effort with Chee Yap. Some initial work on the Core Library is a result of our collaboration with Vijay Karamcheti and Igor Pechtchanski [29]. The library is built upon the Real/Expr package developed by Chee Yap,

Thomas Dubé and Kouji Ouchi [61, 46]. Our library uses the multi-precision representations of real numbers provided by Real/Expr. The underlying big integer and rational numbers are from LiDIA [26] and CLN [22]. The work on the zero testing of radical expressions and its application in automated theorem proving is a result of a joint research effort with Daniela Tulone and Chee Yap [55].

Chapter 2

Constructive Root Bounds

Computing effective root bounds for *constant* algebraic expressions is a critical problem in the Exact Geometric Computation approach to robust geometric computing. Classical root bounds are often non-constructive. Recently, various bounds [40, 41, 5, 61, 62, 2, 48] that can be computed inductively on the structure of expressions have been proposed. We call these bounds *constructive root bounds*. For the important class of *radical expressions*, Burnikel et al (BFMS) have provided a constructive root bound which, in the division-free case, is an improvement over previously known bounds and is essentially tight. But for general algebraic expressions, there is not any single bound that is always better than the others.

In this chapter, we present a new constructive root bound [35] that is applicable to a more general class of algebraic expressions. Our basic idea is to bound the leading and tail coefficients , and the conjugates of the algebraic expression with the help of resultant calculus. The new bound gives significantly better performance in many important computations involving divisions and root extractions. We describe the implementation of this bound in the context of the Core Library, and report some experimental results.

We also present an improvement [35] of the *degree-measure bound*, another constructive root bound proposed by Mignotte [40, 41] and Burnikel et al [2], by exploiting the sharing of common sub-expressions. Furthermore, we show that the degree-measure bound is not generally comparable to our new bound and other previous bounds and thus this improvement has independent interest.

In Section 2.1, we discuss the constructive root bound problem and its application in Exact Geometric Computation. In Section 2.2, we review some previous work. Section 2.3 formalizes the constructive root bound problem. We present our new constructive root bound in Section 2.4, and give an improved degree-measure bound in Section 2.5. A comparative study of various root bounds is given in Section 2.6. In Section 2.7, experimental results are reported. We summarize in Section 2.8. Most of these results have appeared in [35].

2.1 Root Bounds and Exact Geometric Computation

Exact Geometric Computation (EGC) [61] is a general approach to achieve robust geometric programs. This is the approach in, for instance, the LEDA [4, 25] and CGAL [21] libraries. A key goal of EGC is to eliminate numerical non-robustness in geometric predicate evaluations by supporting exact comparison of algebraic expressions. As we have noted before, this is equivalent to determining the sign of algebraic expressions correctly.

Exact sign determination A fundamental task in EGC is to determine the exact sign of a constant algebraic expression E. For example, the following expression arises in the implementation of Fortune's sweepline algorithm [13] for the Voronoi diagram of a planar point set:

$$E = \frac{a + \sqrt{b}}{d} - \frac{a' + \sqrt{b'}}{d'}, \tag{2.1}$$

where a, a', b, b', d, d' are integer constants. In order to determine the exact sign, we adopt a numerical approach based on algebraic root bounds.

Definition 2.1 (Root bound and root bit-bound). *We call a positive number b a root bound for an algebraic expression (or number) E if the following holds:*

$$\text{if } E \neq 0 \text{ then } |E| \geq b.$$

Moreover, we will call $(-\log_2 b)$ a root bit-bound for E.

There are some other variants on the definition of root bound. As our choice of terminology suggests, we are mainly interested in bounding roots away from 0. Typically, the sign determination task reduces to first finding some root bound b for E. With such a root bound b, we can then determine the sign of E as follows:

Step 1: Compute a numerical approximation $\widetilde{E}$ such that

$$|E - \widetilde{E}| < \frac{b}{2};$$

Step 2: Check whether $|\widetilde{E}| \geq \frac{b}{2}$ and get the exact sign of E using the following rule:

$$\text{sign}(E) = \begin{cases} \text{sign}(\widetilde{E}) & \text{if } |\widetilde{E}| \geq \frac{b}{2} \\ 0 & \text{otherwise.} \end{cases}$$

Precision-sensitive approach In practice [61], the precision required in approximation can be progressively increased until one of the following two events occurs:

Either (i) the approximation $\widetilde{E}$ satisfies

$$|\widetilde{E}| > |E - \widetilde{E}|, \tag{2.2}$$

or (ii) the approximation satisfies

$$|E - \widetilde{E}| < \frac{b}{2}. \tag{2.3}$$

Note that if $|E|$ is large, then condition (i) will usually be reached first, and the root bound does not play a role in the actual complexity of the sign determination process. However, if E is really zero (as happens in, say, degenerate cases or some theorem proving application [55]), then the root bound plays a critical role. In the worst case, it is the root bit-bound that determines the complexity of our sign determination algorithm.

Constructive root bound The problem of root bounds and, more generally, root location, is a very classical one with an extensive literature (e.g., [38] or [41, chap. 2]). Some classical bounds are highly non-constructive. But many known root bounds are given in terms of some simple function of P's coefficients and degree. For instance, Landau's bound says that any non-zero root α of $P(x)$ satisfies $|\alpha| \geq \|P(x)\|_2^{-1}$ where $P(x) = \sum_{i=0}^{n} a_i x^i$ and the length $\|P(x)\|_2 = \sqrt{\sum_{i=0}^{n} |a_i|^2}$. Unfortunately, in many applications, the coefficients of $P(x)$ are not explicitly given. For instance, in the LEDA and Core libraries, an algebraic number α is presented as a *radical expression* which is constructed from integers, and recursively built-up using the four arithmetic operations $(+, -, \times, \div)$ and radical extraction $\sqrt[k]{\cdot}$ (here $k \geq 2$ is the *index* of the radical extraction.). Thus, the notion of "constructive" depends on the presentation of α; we call such a presentation an *expression*. If E is a presentation of α, we will write $\mathrm{val}(E) = \alpha$. However, $\mathrm{val}(E)$ may be undefined for some E, e.g., when we divide by 0, or when we take the square root of a negative number. In the following, we will often write "E" in place of $\mathrm{val}(E)$, if there is no confusion. Furthermore, any assertion about $\mathrm{val}(E)$ is conditioned upon $\mathrm{val}(E)$ being well defined.

Definition 2.2 (Constructive root bound). *Given an algebraic expression E, if a bound for its value* $\mathrm{val}(E)$ *can be computed inductively from the structure description of E, we consider it a constructive root bound.*

The *constructive root bound problem* is this: given a set $\mathcal{E}$ of expressions (e.g., the radical expressions), derive a set of inductive rules for computing a root bound for each expression in $\mathcal{E}$. For example, a set of recursive rules for computing root bound for radical expressions are given in [62] based on Landau's bound (see Table 2.1 as well). It is important to realize that in our discussion, the term "expression" roughly corresponds to a directed acyclic graph (DAG) in which nodes are labeled by the appropriate constants and operations (as described in Section 2.3).

2.2 Previous Work

A number of constructive root bounds have been proposed. Here we briefly recall some of them.

Canny's bound. For a zero-dimensional system Σ of n polynomial equations with n unknowns, Canny [5] shows that if $(\alpha_1, \ldots, \alpha_n)$ is a solution, then $|\alpha_i| \geq (3dc)^{-nd^n}$ for each non-zero component α_i. Here c (resp., d) is an upper bound on the absolute value of coefficients (resp., the degree) of any polynomial in the system. An important proviso in Canny's bound is that the homogenized system $\widehat{\Sigma}$ has a non-vanishing U-resultant. Equivalently, $\widehat{\Sigma}$ has finitely many roots at infinity. Yap [62, p. 350] gives the treatment for the general case, based on the notion of "generalized U-resultant". Such multivariate root bounds are easily translated into a bound on expressions, as discussed in [2].

Table 2.1: Rules for degree-length and degree-height bounds

E	d	ℓ	h
rational $\frac{a}{b}$	1	$\sqrt{a^2+b^2}$	$\max\{\lvert a\rvert, \lvert b\rvert\}$
$E_1 \pm E_2$	$d_1 d_2$	$\ell_1^{d_2}\ell_2^{d_1}2^{d_1 d_2+\min\{d_1,d_2\}}$	$(h_1 2^{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}$
$E_1 \times E_2$	$d_1 d_2$	$\ell_1^{d_2}\ell_2^{d_1}$	$(h_1\sqrt{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}$
$E_1 \div E_2$	$d_1 d_2$	$\ell_1^{d_2}\ell_2^{d_1}$	$(h_1\sqrt{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}$
$\sqrt[k]{E_1}$	kd_1	ℓ_1	h_1

Degree-length and degree-height bounds. The degree-length bound [62] is a bound for general algebraic expressions, based on Landau's root bound. For an expression E, the algorithm computes the upper bounds on the degree d and on length ($\|\cdot\|_2$) ℓ of the minimal polynomial of E. If $E \neq 0$, then from Landau's bound we know $|E| \geq \frac{1}{\ell}$. The extended Hadamard bound on polynomial matrix is used to compute an upper bound of ℓ. A similar degree-height bound based on Cauchy's root bound is found in [61]. Here "length" and "height" refer to the 2-norm and ∞-norm of a polynomial, respectively. Both results are based on the resultant calculus. The bounds are maintained inductively on the structure of the expression DAG using the recursive rules found in Table 2.1.

Degree-measure bound. Given a polynomial $P(x) = a_m \prod_{i=1}^m (x - \alpha_i)$, with $a_m \neq 0$, the *measure* of P, $m(P)$, is defined as $|a_m| \cdot \prod_{i=1}^m \max\{1, |\alpha_i|\}$. Furthermore, the measure $m(\alpha)$ of an algebraic number α is defined as the measure of $\mathrm{Irr}(\alpha)$. It is known that if $\alpha \neq 0$, we have

$$\frac{1}{m(\alpha)} \leq |\alpha| \leq m(\alpha). \tag{2.4}$$

Let α and β be two nonzero algebraic numbers of degrees m and n respectively. The following relations on measures are given in [41],

$$
\begin{aligned}
m(\alpha \pm \beta) &\leq 2^{mn} m(\alpha)^n m(\beta)^m &&(2.5)\\
m(\alpha \times \beta) &\leq m(\alpha)^n m(\beta)^m &&(2.6)\\
m(\alpha \div \beta) &\leq m(\alpha)^n m(\beta)^m &&(2.7)\\
m(\alpha^{1/k}) &\leq m(\alpha) &&(2.8)\\
m(\alpha^k) &\leq m(\alpha)^k &&(2.9)
\end{aligned}
$$

Based on Mignotte's work, Burnikel et al [2] develop recursive rules to maintain the upper bounds for degrees and measures and call it the *degree-measure bound*. These rules are given in the last two columns of Table 2.7 where $M'(E)$ and $D'(E)$ are (respectively) upper bounds on $m(E)$ and $\deg(E)$. Similar rules are given in [41]. The degree-measure bound turns out to be always better than the degree-length bound.

BFMS bound. One of the best constructive root bounds for the class of radical expressions is from Burnikel et al [2] (hereafter called the "BFMS bound"). This is the bound that is used in the LEDA and CGAL libraries. The BFMS approach is based on a well-known transformation of an expression E to eliminate all but one division, producing two associated division-free expressions $U(E)$ and $L(E)$ such that $\mathrm{val}(E) = \mathrm{val}(U(E))/\mathrm{val}(L(E))$. E.g., if $E = \frac{a}{b} + \frac{c}{d}$, then $U(E) = ad + bc$ and $L(E) = bd$. Note that in this transformation, the number of square roots in $U(E)$ or $L(E)$ could be potentially doubled. Two parameters $u(E)$ and $l(E)$, the upper bounds on the absolute value of conjugates of $U(E)$ and $L(E)$, respectively, are maintained by the recursively rules in Table 2.2.

Table 2.2: BFMS rules

	E	$u(E)$	$l(E)$
1.	integer a	$\|a\|$	1
2.	$E_1 \pm E_2$	$u(E_1)l(E_2) + l(E_1)u(E_2)$	$l(E_1)l(E_2)$
3.	$E_1 \times E_2$	$u(E_1)u(E_2)$	$l(E_1)l(E_2)$
4.	$E_1 \div E_2$	$u(E_1)l(E_2)$	$l(E_1)u(E_2)$
5.	$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$	$\sqrt[k]{l(E_1)}$

Clearly, if E is division-free, then $L(E) = 1$ and $\mathrm{val}(E)$ is an algebraic integer (i.e., a root of some monic integer polynomial).

For an expression E having r radical nodes with indices $k_1, k_2, \ldots, k_r$, the BFMS bound is given by

$$\mathrm{val}(E) \neq 0 \Rightarrow (u(E)^{D(E)^2-1}l(E))^{-1} \leq |\mathrm{val}(E)| \leq u(E)l(E)^{D(E)^2-1}, \tag{2.10}$$

where $D(E) = \prod_{i=1}^{r} k_i$, and $u(E)$ and $l(E)$ are (respectively) upper bounds on the absolute values of algebraic conjugates of $\mathrm{val}(U(E))$ and $\mathrm{val}(L(E))$. For division-free expressions, the BFMS bound improves to

$$\mathrm{val}(E) \neq 0 \Rightarrow |\mathrm{val}(E)| \geq (u(E)^{D(E)-1})^{-1}. \tag{2.11}$$

The bound for division-free expressions was shown to be essentially sharp and better than previous bounds. But in presence of divisions, the BFMS bound is not necessarily an improvement of the degree-measure bound (see Section 2.6).

Note that the root bit-bound in 2.10 is quadratic in $D(E)$, [1] while in 2.11, it is linear in $D(E)$. Our experience is that this quadratic factor can be a serious efficiency

[1]Most recently, Mehlhorn et al [39] give a variation of the BFMS bound which depends on $D(E)$ linearly.

issue. Consider a simple example: $E = (\sqrt{x} + \sqrt{y}) - \sqrt{x + y + 2\sqrt{xy}}$ where x, y are L-bit integers (i.e., $|x|, |y| < 2^L$). Of course, this expression is identically 0 for any x, y. The BFMS bound yields a root bit-bound of $7.5L + \mathcal{O}(1)$ bits. But in case, x and y are viewed as rational numbers (with denominator 1), the bit-bound becomes $127.5L + \mathcal{O}(1)$. The example shows that introducing rational numbers at the leaves of expressions has a major impact on the BFMS bound. In practice, this is an important and common situation: for instance, it is usual to have floating point numbers as input constants in an expression. Since these are special cases of rational numbers, the BFMS bound becomes quite pessimistic.

Scheinerman bound. This adopts an interesting approach based on matrix eigenvalues [48]. Let $\Lambda(n, b)$ denote the set of eigenvalues of $n \times n$ matrices with integer entries with absolute value at most b. It is easy to see that $\Lambda(n, b)$ is a finite set of algebraic integers. Moreover, if $\alpha \in \Lambda(n, b)$ is non-zero then $|\alpha| \geq (nb)^{1-n}$. Scheinerman gives a constructive root bound for division-free radical expressions E by maintaining two parameters, $n(E)$ and $b(E)$, satisfying the property that the value of E is in $\Lambda(n(E), b(E))$. These recursive rules are given by Table 2.3.

Note that the rule for $\sqrt{cd}$ is rather special, but it can be extremely useful. In Rule 6, the polynomial $\overline{P}(x)$ is given by $\sum_{i=0}^{d} |a_i| x^i$ when $P(x) = \sum_{i=0}^{d} a_i x^i$. This rule is not explicitly stated in [48], but can be deduced from an example he gave. An example given in [48] is to test whether $\alpha = \sqrt{2} + \sqrt{5 - 2\sqrt{6}} - \sqrt{3}$ is zero. Scheinerman's bound requires calculating α to 39 digits while the BFMS bound and our new bound say 12 digits are enough.

Table 2.3: Scheinerman's rules

	E	$n(E)$	$b(E)$
1.	integer a	1	$\|a\|$
2.	$\sqrt{cd}$	2	$\max\{\|c\|, \|d\|\}$
3.	$E_1 \pm E_2$	$n_1 n_2$	$b_1 + b_2$
4.	$E_1 \times E_2$	$n_1 n_2$	$b_1 b_2$
5.	$\sqrt[k]{E_1}$	$k n_1$	b_1
6.	$P(E_1)$	n_1	$\overline{P}(n_1 b_1)$

2.3 The General Framework

We formalize the constructive root bound problem as follows. In our discussion hereafter, a "DAG" is an ordered, directed acyclic graph with a unique node that has out-degree 0, called the *root*. The DAG is ordered in the sense that the set of incoming edges to each node u is given a total ordering. Nodes with in-degree 0 are called *leaves*. Let Ω be a set of algebraic operations: each $\omega \in \Omega$ represents a partial function $f_\omega : \mathbb{C}^k \to \mathbb{C}$ where $\mathbb{C}$ are the complex numbers and $k = k(\omega)$ is called the *arity* of ω. If $k(\omega) = 0$ then ω may be identified with an element of $\mathbb{C}$ and is called a *constant*. An *expression* over Ω (or Ω-expression) is a DAG where each node u of in-degree k_u is labeled by an operation $\omega \in \Omega$ where $k(\omega)$ equals the in-degree of u. In particular the leaves are labeled by constants. In case[2] the DAG is a tree, then we call it a *tree expression*. Each node in an expression induces a natural subexpression. Let $\mathcal{E}(\Omega)$ denote the set of Ω-expressions. The following classes of expressions are the main ones in this paper:

[2] In some literature, our tree expressions are simply called "expressions" while our expressions are essentially "straightline programs" or "circuits".

- $\Omega_0 = \{\pm, \times\} \cup \mathbb{Z}$ (where $\mathbb{Z}$ are the integers). Thus Ω_0-expressions are integral polynomial expressions.
- $\Omega_1 = \Omega_0 \cup \{\div\}$. Thus Ω_1-expressions are rational expressions.
- $\Omega_2 = \Omega_1 \cup \{\sqrt[n]{\cdot} : n \geq 2\}$. Thus Ω_2-expressions are radical expressions.
- $\Omega_3 = \Omega_2 \cup \{\mathrm{Root}(P) : P \in \mathbb{Z}[x]\}$. Our main root bound applies to Ω_3-expressions. We assume the polynomial P is presented by its sequence of $n+1$ integer coefficients if $\deg(P) = n$.

We need to clarify the $\mathrm{Root}(P)$ operation in Ω_3 above. This is intended to be a constant referring to some root α of P. In practice, we will need some method for identifying the root α. For instance, if α is real (as we assume in our applications) and it is the kth largest real root of P, we could identify α as "$\mathrm{Root}(P, k)$". Instead of k, we could also use, say, an isolating interval for α. It turns out that our root bounds do not depend on the choice of the root of P, and hence, we normally write "$\mathrm{Root}(P)$" instead of "$\mathrm{Root}(P, k)$".

For any set $\mathcal{E}$ of expressions, there is a partial function $\mathrm{val} : \mathcal{E} \to \mathbb{C}$ that is naturally defined by applying the appropriate functions f_ω ($\omega \in \Omega$) at each node of an expression. Notice that $\mathrm{val}(E)$ is undefined if any of its nodes has an undefined value, for instance, $\mathrm{val}(\mathrm{Root}(P, k))$ is undefined if $P(x)$ has less than k real roots. Having undefined values is not a new phenomenon, since this already arises when we divide by zero or take the square-root of a negative number (when values are assumed to be real). All our statements about $\mathrm{val}(E)$ are also conditioned on $\mathrm{val}(E)$ being defined. The *constructive root bound problem* for a class $\mathcal{E}$ of expressions is that of providing a *bounding function*

$$B : \mathbb{R}^m \to \mathbb{R} \quad (\mathbb{R} = \text{reals})$$

and a set of "recursive rules" to compute for each $E \in \mathcal{E}$ a set of real parameters $\{a_i(E) : i = 1, \ldots, m\}$, plus possibly other non-numeric parameters, such that the following holds:

$$\mathrm{val}(E) \neq 0 \Rightarrow |\mathrm{val}(E)| \geq B(a_1(E), \ldots, a_m(E)).$$

The rules are "recursive" in the sense that the parameters for each node in the DAG can be effectively computed from the parameters of its predecessors. Non-numeric parameters are needed, for instance, in the BFMS bound, we need to compute the set of radical nodes in the DAG. In practice, the function B will be non-negative, with both B and the recursive rules relatively simple to compute. Another desirable property is that the bound $B(a_1(E), \ldots, a_m(E))$ should be as large as possible. Also, we call m the *order* of constructive root bound.

Example: In the original degree-measure bound, we compute two parameters, $a_1(E)$ and $a_2(E)$ where a_1 and a_2 are upper bounds on the degree and measure of E. Moreover, the bounding function $B : \mathbb{R}^m \to \mathbb{R}$ is given by $B(a, b) = 1/b$ (the first parameter is ignored by B). So the order of the degree-measure bound is 2.

Example: If there are two constructive root bounds using bounding function $B_1(x_1, \ldots, x_m)$ and $B_2(y_1, \ldots, y_n)$ then we can have a new composite constructive root bound using the bounding function

$$B(x_1, \ldots, x_m, y_1, \ldots, y_n) = \max\{B_1(x_1, \ldots, x_m), B_2(y_1, \ldots, y_n)\}$$

of order at most $m + n$. For instance, in the Core Library, we actually choose the maximum of the BFMS bound, the improved degree-measure bound, and our new bound.

2.4 A New Constructive Root Bound

In this section, we develop a constructive root bound for Ω_3-expressions. For any algebraic number α, we will exploit the following relation:

$$\alpha \neq 0 \Rightarrow |\alpha| \geq (\mu(\alpha)^{\deg(\alpha)-1}\mathrm{lead}(\alpha))^{-1}, \tag{2.12}$$

where $\mu(\alpha) = \max\{|\xi| \;:\; \xi$ is a conjugate of $\alpha\}$, $\deg(\alpha)$ is the degree of the minimal polynomial $\mathrm{Irr}(\alpha)$ of α and $\mathrm{lead}(\alpha)$ is the leading coefficient of $\mathrm{Irr}(\alpha)$.

In order to obtain the root bound for an expression E using the relation (2.12), we need three parameters: $\deg(E)$, $\mu(E)$ and $\mathrm{lead}(E)$. The definitions of these parameters involve the minimal polynomial of E, which is usually expensive to compute. Instead, we give recursive rules to maintain upper bounds

$$D(E), \quad \overline{\mu}(E), \quad \mathrm{lc}(E)$$

on the corresponding parameters.

Degree Bound. First we consider $D(E)$, the upper bound on the degree of E. Suppose that E has k radical nodes or root-of-polynomial nodes $\{r_1, r_2, \ldots, r_k\}$. Assume some topological sorting $r_1 \prec r_2 \prec \cdots \prec r_k$ of these nodes so that if r_i is a predecessor of r_j (i.e., r_i is referenced by the sub-expression r_j) then $i < j$. The value of E $\mathrm{val}(E)$ is an element in the finite algebraic extension field $\mathbb{Q}_k = \mathbb{Q}(r_1, r_2, \ldots, r_k)$ of $\mathbb{Q}$, which is obtained from a tower of extensions from $\mathbb{Q}$ as follows:

$$\mathbb{Q} \subseteq \mathbb{Q}(r_1) \subseteq \mathbb{Q}(r_1, r_2) \ldots \subseteq \mathbb{Q}(r_1, \ldots, r_k) = \mathbb{Q}_k \subset \bar{\mathbb{Z}}.$$

For simplicity, we denote $\mathbb{Q}_i = \mathbb{Q}(r_1, \ldots, r_i)$. We know that the dimension of $\mathbb{Q}_k$ over $\mathbb{Q}$, denoted as $[\mathbb{Q}_k : \mathbb{Q}]$, is as follows:

$$\begin{aligned}[\mathbb{Q}_k : \mathbb{Q}] &= [\mathbb{Q}_k : \mathbb{Q}_{k-1}][\mathbb{Q}_{k-1} : \mathbb{Q}_{k-2}] \cdots [\mathbb{Q}_1 : \mathbb{Q}_0] \\ &= \prod_{i=1}^{k} d_i\end{aligned}$$

where d_i is the degree of r_i over the extension field $\mathbb{Q}(r_1, \ldots, r_{i-1})$. Thus, the degree of E, an element of $\mathbb{Q}_k$, over $\mathbb{Q}$ is at most $\prod_{i=1}^{k} d_i$. Define $D(E) = \prod_{i=1}^{k} k_i$ where k_i is either the index of r_i if r_i is a radical node, or the degree of the polynomial if r_i is a polynomial-root node. Clearly, $D(E)$ is an upper bound on $\deg(E)$ since $d_i \leq k_i$ for all i. Note that the degree bound $d(E)$ used in the degree-measure approach (Table 2.7), which is actually also used in degree-length and degree-height approaches, is actually $\prod_{i=1}^{k} k_i^{c_i}$, where $c_i \geq 1$ is the number of distinct paths from the radical node r_i to the root. It can be easily verified that $D(E)$ is never worse than $d(E)$ and can be significantly better if there are lots of sharing on subexpressions.

Next, we investigate the methods to bound leading coefficients and conjugates. Given a non-zero polynomial $P(x)$, we denote its leading coefficient, its tail coefficient and its constant coefficient (respectively) by $\text{lead}(P)$, $\text{tail}(P)$, and $\text{const}(P)$. Note that the *tail coefficient* $\text{tail}(P)$ is defined to be the last non-zero coefficient of P. Hence $\text{tail}(P)$ and $\text{lead}(P)$ are non-zero by definition. Also, let $m(P)$ denote the *measure* of P. Given an algebraic number α, we define $\text{lead}(\alpha)$, $\text{tail}(\alpha)$, etc., to be $\text{lead}(\text{Irr}(\alpha))$, $\text{tail}(\text{Irr}(\alpha))$, etc.. Actually, $\text{tail}(\alpha)$ is the same as $\text{const}(\text{Irr}(\alpha))$ (when $\alpha = 0$ this is true by definition). In the following, we will show inductive rules to bound this parameters. Instead of computing $\text{Irr}(E)$ explicitly, we study the defining polynomials P_E of E constructed from the resultant calculus. As $\text{Irr}(E) \mid P_E$, it is clear that $|\text{lead}(P_E)|$, $|\text{tail}(P_E)|$ and $m(P_E)$ are upper bounds of $|\text{lead}(E)|$, $|\text{tail}(E)|$ and $m(E)$, respectively.

Bound on the Leading Coefficients and Table 2.4. We now consider $\mathrm{lc}(E)$, which is an upper bound on $|\mathrm{lead}(E)|$. The admission of divisions makes it necessary to bound tail coefficients as well. Moreover, we also need to bound the measure of E to help bound $|\mathrm{tail}(E)|$ (this is only used when E has the form $E = E_1 \pm E_2$). Let $\mathrm{tc}(E)$ and $M(E)$ denote upper bounds on $|\mathrm{tail}(E)|$ and $m(E)$, respectively. Table 2.4 gives the recursive rules to maintain $\mathrm{lc}(E)$, $\mathrm{tc}(E)$ and $M(E)$.

Table 2.4: Recursive rules for $\mathrm{lc}(E)$ (and associated $\mathrm{tc}(E)$ and $M(E)$)

	E	$\mathrm{lc}(E)$	$\mathrm{tc}(E)$	$M(E)$
1.	rational $\frac{a}{b}$	$\|b\|$	$\|a\|$	$\max\{\|a\|, \|b\|\}$
2.	$\mathrm{Root}(P)$	$\|\mathrm{lead}(P)\|$	$\|\mathrm{tail}(P)\|$	$\\|P\\|_2$
3.	$E_1 \pm E_2$	$\mathrm{lc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$M_1^{D_2}M_2^{D_1}2^{D(E)}$	$M_1^{D_2}M_2^{D_1}2^{D(E)}$
4.	$E_1 \times E_2$	$\mathrm{lc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$\mathrm{tc}_1^{D_2}\mathrm{tc}_2^{D_1}$	$M_1^{D_2}M_2^{D_1}$
5.	$E_1 \div E_2$	$\mathrm{lc}_1^{D_2}\mathrm{tc}_2^{D_1}$	$\mathrm{tc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$M_1^{D_2}M_2^{D_1}$
6.	$\sqrt[k]{E_1}$	lc_1	tc_1	M_1
7.	E_1^k	lc_1^k	tc_1^k	M_1^k

The upper bound $M(E)$ on the measure of E is shown[3] in the last column. It can be shown that, for any expression E, we have

$$\mathrm{tc}(E) \le M(E) \quad \text{and} \quad \mathrm{lc}(E) \le M(E). \tag{2.13}$$

Note that we introduce a special node for the power operation $E = E_1^k$. This is not just a shortcut for $(k-1)$ multiplications; it leads to much better bounds too. For

[3]This information is copied from column 2 in Table 2.7, and is discussed in conjunction with that table.

example, in computing $\mathrm{lc}(E)$ by naively expanding E into $(k-1)$ multiplications, we get $\mathrm{lc}(E) = \mathrm{lc}_1^{kD_1^{k-1}} \gg \mathrm{lc}_1^k$. Similar improvements can be shown for $\mathrm{tc}(E)$ and $m(E)$.

One subtlety arises for expressions of the form $E = E_1 \pm E_2$. In this case, resultant calculus gives us a polynomial $P_E(x)$ where $\mathrm{val}(E)$ vanishes. Although, we can also deduce a bound on $\mathrm{const}(P_E)$, unfortunately, this constant coefficient may vanish and hence tell us nothing about $\mathrm{tail}(E)$. Hence we need to resort to the measure $m(E)$ as a bound for $|\mathrm{tail}(E)|$.

Justification of Table 2.4. The basic techniques come from resultant calculus. Without resorting to computing the minimal polynomials, consider the defining polynomials constructed from resultant calculus instead and we can compute various bounds based on them. Let α and β be roots of the polynomials $A(x) = a_m \prod_{i=1}^m (x - \alpha_i)$ and $B(x) = b_n \prod_{j=1}^n (x - \beta_j)$. By the resultant calculus [62], we construct a polynomial P_E which vanishes at $E = \alpha \odot \beta$ $(\odot \in \{+, -, \times, /\})$ as follows:

$$P_{\alpha\pm\beta} = \mathrm{res}_y(A(x \mp y), B(y)) = c \cdot \prod_{i=1}^m \prod_{j=1}^n (x - (\alpha_i \pm \beta_j)) \tag{2.14}$$

$$P_{\alpha\times\beta} = \mathrm{res}_y(y^m A(x/y), B(y)) = c \cdot \prod_{i=1}^m \prod_{j=1}^n (x - (\alpha_i \beta_j)) \tag{2.15}$$

$$P_{\alpha/\beta} = \mathrm{res}_y(A(xy), B(y)) = b_0^m a^n \cdot \prod_{i=1}^m \prod_{j=1}^n (x - (\alpha_i / \beta_j)),\ b_0 \neq 0, \tag{2.16}$$

where the constant $c = (-1)^{mn} \cdot a_m^n b_n^m$ and $b_0 = B(0)$. The operator $\mathrm{res}_y(\cdot,\cdot)$ means taking the resultant of two polynomial arguments with y being the main variable to be eliminated. Since $\mathrm{Irr}(E) \mid P_E$, it is clear that

$$|\mathrm{lead}(E)| \leq |\mathrm{lead}(P_E)|, \quad |\mathrm{tail}(E)| \leq |\mathrm{tail}(P_E)|, \quad m(E) \leq m(P_E), \tag{2.17}$$

$$\overline{\mu}(P_E) \geq \overline{\mu}(E), \quad \underline{\nu}(P_E) \leq \underline{\nu}(E) \tag{2.18}$$

where $\underline{\nu}(E)$ is defined below. The leading and constant coefficients of P_E can be easily deduced from the above equations, and are summarized in Table 2.5 where the constant terms $a_0 = A(0)$ and $b_0 = B(0)$:

Table 2.5: The polynomial P_E and its leading and last coefficients

E	$P_E(x)$	$\mathrm{lead}(P_E(x))$	$\mathrm{const}(P_E(x))$
$\alpha \pm \beta$	$\mathrm{res}_y(A(x \mp y), B(y))$	$(-1)^{mn} a_m^n b_n^m$	$\mathrm{res}(A(x), B(x))$
$\alpha \times \beta$	$\mathrm{res}_y(y^m A(x/y), B(y))$	$(-1)^{mn} a_m^n b_n^m$	$a_0^n b_0^m$
$\alpha/\beta \ \ (b_0 \neq 0)$	$\mathrm{res}_y(A(xy), B(y))$	$a_m^n b_0^m$	$(-1)^{mn} a_0^n b_n^m$

The next lemma justifies the rules for $\mathrm{tc}(E)$ and $\mathrm{lc}(E)$:

Lemma 2.3. *Let E_1 and E_2 be algebraic expressions with degrees m and n, and the expression E be any of the expressions is constructed from E_1 and E_2 as in Table 2.3. Then the $|\mathrm{lead}(E)|$ and $|\mathrm{tail}(E)|$ (respectively) are bounded by* lc *and* tc *in the table, where* lc_i *and* tc_i *are upper bounds on $|\mathrm{lead}(E_i)|$ and $|\mathrm{tail}(E_i)|$, respectively.*

E	$\mathrm{lc}(E)$	$\mathrm{tc}(E)$
$E_1 \pm E_2$	$\mathrm{lc}_1^n \mathrm{lc}_2^m$	$m(E)$
$E_1 \times E_2$	$\mathrm{lc}_1^n \mathrm{lc}_2^m$	$\mathrm{tc}_1^n \mathrm{tc}_2^m$
$E_1 \div E_2$	$\mathrm{lc}_1^n \mathrm{tc}_2^m$	$\mathrm{tc}_1^n \mathrm{lc}_2^m$
$\sqrt[k]{E_1}$	lc_1	tc_1
E_1^k	lc_1^k	tc_1^k

Proof. The first three lines of Table 2.3 are essentially justified by Table 2.5 since the minimal polynomial $\mathrm{Irr}(E)$ divides $P_E(x)$ constructed from the resultant calculus.

Thus $\mathrm{lead}(E) \leq \mathrm{lead}(P_E)$ and $\mathrm{tail}(E) \leq \mathrm{tail}(P_E)$. The only exception is the rule for $\mathrm{tc}(E_1 \pm E_2)$ since the constant term of $P_{E_1 \pm E_2}$ may vanish and tell us nothing about the tail coefficient. Hence we use the measure of $E_1 \pm E_2$ instead, because for any algebraic number α, we have $\mathrm{tail}(E) \leq m(E)$.

It is easy to see the polynomial $A(x^k)$ vanishes at $E = \sqrt[k]{E_1}$ with the same leading and tail coefficients as those in A. Finally, for a power expression $E = E_1^k$, E is a root of the polynomial

$$R(x) = \mathrm{res}_y(A(y), x - y^k) = a^k \prod_{i=1}^{m}(x - \alpha_i^k)$$

where $A(x) = a_m \prod_{i=1}^{m}(x - \alpha_i)$ is the minimal polynomial of E. Hence $|\mathrm{lead}(E)| \leq |\mathrm{lead}(R)| = |a_m|^k \leq \mathrm{lc}_1^k$ and $|\mathrm{tail}(E)| \leq |\mathrm{tail}(R)| = |A(0)|^k \leq \mathrm{tc}_1^k$. **Q.E.D.**

Lemma 2.3 proves the validity of the bounding rules for one step. An argument by induction then can show that:

Corollary 2.4. *The* $\mathrm{lc}(E)$ *computed from the recursive rules in Table 2.4 bounds* $\mathrm{lead}(E)$.

Bound on Conjugates and Table 2.6. Now we consider $\overline{\mu}(E)$, which is an upper bound on the absolute value of all the conjugates of $\mathrm{val}(E)$. Because of the admission of divisions, we also have to maintain $\underline{\nu}(E)$, which is a lower bound on the absolute value of all the conjugates of $\mathrm{val}(E)$ whenever $\mathrm{val}(E) \neq 0$. The recursive rules to maintain these two bounds are given in Table 2.6. The most noteworthy entry in Table 2.6 is the bound for $\underline{\nu}(E)$ when $E = E_1 \pm E_2$. In this case, we cannot obtain a lower bound on $\nu(E)$ based on $\underline{\nu}(E_1)$ and $\underline{\nu}(E_2)$ due to potential cancellation. Instead, we can use either relation 2.12 or 2.4. As we will see in Section 2.6, neither bound is strictly better than the other. Hence, we take the maximum of the two bounds for $\underline{\nu}(E_1 \pm E_2)$. More generally, we could use 2.4 in all the entries of 2.6 if they give better bounds. We also note that our bounds on $\overline{\mu}(\mathrm{Root}(P))$ and $\underline{\nu}(\mathrm{Root}(P))$ are based on Cauchy's root bound [62, p. 148]. Recall that we assume that the polynomial $P = P(x) = \sum_{i=0}^{n} a_i x^i$ is explicitly given in terms of its coefficients. Hence a more precise bound can be used,

$$\frac{|a_0|}{|a_0| + \max\{|a_1|, \ldots, |a_n|\}} \leq \mathrm{Root}(P) \leq 1 + \frac{\max\{|a_0|, \ldots, |a_{n-1}|\}}{|a_n|}.$$

Of course, any of the classical root bounds can be used as convenient here.

Table 2.6: Recursive rules for bounds on conjugates

	E	$\overline{\mu}(E)$	$\underline{\nu}(E)$
1.	rational $\frac{a}{b}$	$\lvert\frac{a}{b}\rvert$	$\lvert\frac{a}{b}\rvert$
2.	$\mathrm{Root}(P)$	$1 + \lVert P\rVert_\infty$	$(1 + \lVert P\rVert_\infty)^{-1}$
3.	$E_1 \pm E_2$	$\overline{\mu}(E_1) + \overline{\mu}(E_2)$	$\max\{M(E)^{-1}, (\overline{\mu}(E)^{D(E)-1}\mathrm{lc}(E))^{-1}\}$
4.	$E_1 \times E_2$	$\overline{\mu}(E_1)\overline{\mu}(E_2)$	$\underline{\nu}(E_1)\underline{\nu}(E_2)$
5.	$E_1 \div E_2$	$\overline{\mu}(E_1)/\underline{\nu}(E_2)$	$\underline{\nu}(E_1)/\overline{\mu}(E_2)$
6.	$\sqrt[k]{E_1}$	$\sqrt[k]{\overline{\mu}(E_1)}$	$\sqrt[k]{\underline{\nu}(E_1)}$
7.	E_1^k	$\overline{\mu}(E_1)^k$	$\underline{\nu}(E_1)^k$

The following lemma justifies the rules in Table 2.6.

Lemma 2.5. *Let α and β be two non-zero algebraic numbers and γ be defined as below (in column 1). Then $\mu(\gamma) \leq \overline{\mu}(\gamma)$ and $\nu(\gamma) \geq \underline{\nu}(\gamma)$ where $\mu(\gamma)$ (or $\nu(\gamma)$) are the maximum (or minimum) absolute value of all the conjugates of γ ,and $\overline{\mu}(\gamma)$ and $\underline{\nu}(\gamma)$ are defined as follows:*

γ	$\overline{\mu}(\gamma)$	$\underline{\nu}(\gamma)$
$\alpha \pm \beta$	$\mu(\alpha) + \mu(\beta)$	$(\mu(\gamma)^{\deg(\gamma)-1}\mathrm{lc}(\gamma))^{-1}$
$\alpha \times \beta$	$\mu(\alpha)\mu(\beta)$	$\nu(\alpha)\nu(\beta)$
$\alpha \div \beta$	$\mu(\alpha)\nu(\beta)$	$\nu(\alpha)\mu(\beta)$
$\sqrt[k]{\alpha}$	$\sqrt[k]{\mu(\alpha)}$	$\sqrt[k]{\nu(\alpha)}$
α^k	$\mu(\alpha)^k$	$\nu(\alpha)^k$

Proof. Let $\{\alpha_1, \ldots, \alpha_m\}$ and $\{\beta_1, \ldots, \beta_n\}$ be the conjugates of α and β respectively. For the four basic arithmetic operations $\{+, -, \times, /\}$, let $R(x)$ be the polynomial $P_{\alpha\odot\beta}$ defined by resultant calculus (2.14) – (2.16). The roots of R have the format $\alpha_i \odot \beta_j$. We also know that for root extraction operations, we can choose $R(x)$ to be $\mathrm{Irr}(\alpha)(x^k)$ with the roots $\sqrt[k]{\alpha_i} \cdot \omega^j$ ($i \in [1, \deg(\alpha)], j \in [1, k]$ and ω is k-th unit root of unity). For power operations, we can choose $R(x) = \mathrm{res}_y(\mathrm{Irr}(\alpha)(y), x - y^k)$ which has the roots in the forms of α_i^k. In all these cases, by the inequalities in 2.18, it is sufficient to bound the roots for the polynomial R since $\mathrm{Irr}(\gamma)|R$. It can be easily verified that $\overline{\mu}(\gamma)$ and $\underline{\nu}(\gamma)$ are upper and lower bounds for the zeros of R, respectively. The exception is the bound $\underline{\nu}(\alpha \pm \beta)$, which is based on a direct argument about the minimal polynomial of $\alpha \pm \beta$. Therefore, $\overline{\mu}(\gamma)$ and $\underline{\nu}(\gamma)$ are the upper and lower bounds on the absolute value of the conjugates of γ. **Q.E.D.**

By induction on the structure of expressions, we have

Corollary 2.6. *Let E be an algebraic expression represented as a DAG. For all the conjugates ξ of E, $|\xi| \leq \overline{\mu}(E)$ where $\overline{\mu}(E)$ is the bound computed inductively using the rules in Table 2.6.*

Finally, we obtain the new root bound in the following theorem:

Theorem 2.7. *Given an Ω_3-expression E, if $E \neq 0$, then*

$$|E| \geq (\overline{\mu}(E)^{(D(E)-1)}\mathrm{lc}(E))^{-1}. \tag{2.19}$$

2.5 Improved Degree-Measure Bound

Recall that if an algebraic number $\alpha \neq 0$, we have

$$\frac{1}{m(\alpha)} \leq |\alpha| \leq m(\alpha),$$

where $m(\alpha)$ is the measure of α.

Let E be a Ω_3-expression. As explained in Section 2.2, $M'(E)$ and $D'(E)$ in Table 2.7 are the original degree-measure bound [40, 41, 2]. Recall the definition of $D(E)$ in Section 2.4 which gives an upper bound on $\deg(E)$. It is clear that $D(E)$ is never larger than $D'(E)$. In this section, we give an improved upper bound (denoted by $M(E)$ in Table 2.7) on measures by exploiting the sharing of common sub-expressions.

Table 2.7: The original and our improved degree-measure bounds

	E	$M(E)$ (new)	$M'(E)$ (old)	$D'(E)$ (old)
1.	rational $\frac{a}{b}$	$\max\{\lvert a\rvert, \lvert b\rvert\}$	$\max\{\lvert a\rvert, \lvert b\rvert\}$	1
2.	$\mathrm{Root}(P)$	$\lVert P\rVert_2$	–	–
3.	$E_1 \pm E_2$	$M_1^{D_2} M_2^{D_1} 2^{D(E)}$	${M_1'}^{D_2'} {M_2'}^{D_1'} 2^{D_1' D_2'}$	$D_1' D_2'$
4.	$E_1 \times E_2$	$M_1^{D_2} M_2^{D_1}$	${M_1'}^{D_2'} {M_2'}^{D_1'}$	$D_1' D_2'$
5.	$E_1 \div E_2$	$M_1^{D_2} M_2^{D_1}$	${M_1'}^{D_2'} {M_2'}^{D_1'}$	$D_1' D_2'$
6.	$\sqrt[k]{E_1}$	M_1	M_1'	kD_1'
7.	E_1^k	M_1^k	${M_1'}^k$	–

When $E = \mathrm{Root}(P)$ for some polynomial P, we use $\|P\|_2$ as the upper bound $M(E)$ because $\|P\|_2 \geq m(P)$. Besides the introduction of the new operations of $\mathrm{Root}(P)$ and power (E_1^k) in Table 2.7, we give a slightly improved rule for the measure of $E_1 \pm E_2$. Basically, we can replace the factor of $2^{D_1' D_2'}$ by 2^D. Here is the justification:

Lemma 2.8. *If α and β are algebraic numbers with degrees m, n, respectively, then the measure of $\alpha \pm \beta$ is bounded by $2^d m(\alpha)^n m(\beta)^m$ where $d = \deg(\alpha \pm \beta)$.*

Proof. Assume $\mathrm{Irr}(\alpha) = A(x) = a\prod_{i=1}^m (x - \alpha_i)$ and $\mathrm{Irr}(\beta) = B(x) = b\prod_{j=1}^n (x - \beta_j)$. From resultant calculus, we know that $\gamma = \alpha \pm \beta$ is a root of the polynomial

$$P(x) = \mathrm{res}_y(B(y), A(x \mp y)) = a^n b^m \cdot \prod_{i=1}^m \prod_{j=1}^n (x - (\alpha_i \pm \beta_j))$$

Let $\mathrm{Irr}(\gamma) = c\prod_{k=1}^d (x - \gamma_k)$. Since $\mathrm{Irr}(\gamma) \mid P(x)$ we can assume that $\gamma_k = \alpha_{f(k)} \pm \beta_{g(k)}$ for some function f and g. Moreover, it is clear that $|c| \,|\, |a|^n |b|^m$. Thus,

$$\begin{aligned}
m(\gamma) &= |c| \prod_{k=1}^d \max\{1, |\alpha_{f(k)} \pm \beta_{g(k)}|\} \\
&\leq |a|^n |b|^m \prod_{k=1}^d (2\max\{1, |\alpha_{f(k)}|\} \max\{1, |\beta_{g(k)}|\}) \\
&= 2^d (|a|^n |b|^m \prod_{k=1}^d (\max\{1, |\alpha_{f(k)}|\} \max\{1, |\beta_{g(k)}|\}) \\
&\leq 2^d (|a|^n |b|^m \prod_{i=1}^m \prod_{j=1}^n (\max\{1, |\alpha_i|\} \max\{1, |\beta_j|\}) \\
&= 2^d m(\alpha)^n m(\beta)^m
\end{aligned}$$

The first inequality utilizes the relation that given two numbers α and β, we have

$$\max\{1, |\alpha \pm \beta|\} \leq 2\max\{1, |\alpha|\} \max\{1, |\beta|\}.$$

The last inequality is true because for each α_i, it can only appears at most n times in the decomposition of conjugates of γ (since it only appears in n roots of the polynomial of

P_E and all the conjugates of γ is a root of P_E too.). Similarly, each β_j appears at most m times.

Q.E.D.

The improvement can be significant when there is sharing of subexpressions. For example, consider

$$E = ((\sqrt{x} + \sqrt{y}) - 2\sqrt{x + y + 2\sqrt{x}\sqrt{y}}) \cdot ((\sqrt{x} + \sqrt{y}) + 2\sqrt{x + y + 2\sqrt{x}\sqrt{y}})$$

where x and y are L-bit integers. The original degree-measure bound for E is $m(E) \leq 2^{3584L+7148}$. But when all the common subexpressions of E are merged, our new bound gives $2^{896L+1408}$.

2.6 Comparison of the Root Bounds

We compare the various root bounds discussed in this chapter. Because these bounds are rather different in recursive form, a direct comparison is sometimes not possible. Hence we compare their behavior on interesting classes $\mathcal{C}$ of expressions. For the ith constructive root bound (i=BFMS, degree-measure, etc) and expression $E \in \mathcal{C}$, let $B_i(E)$ denote the root bit-bound for E. Let $\mathcal{C}(L)$ denote the expressions in $\mathcal{C}$ whose input parameters are L-bit integers (or rational numbers, as the case may be). We want to study functions $\beta_i(L)$

$$\beta_i(L) = \max\{B_i(E) : E \in \mathcal{C}(L)\}.$$

We use the root bit-bounds in comparison because it directly determines the number of bits we have to compute in exact sign determination. Note that even if $\beta_i(L) \leq \beta_j(L)$ for all L, it does not necessarily mean that $B_i(E) \leq B_j(E)$ for all $E \in \mathcal{C}$. Sometimes such a stronger relation can be asserted.

1. By an examination of our tables, we can assert the following:

Lemma 2.9. *For any division-free radical expression E, our new bound is exactly the same as the BFMS bound.*

2. Consider the well-known problem of sum of square roots. Suppose $E = \sum_{i=1}^{n} c_i\sqrt{a_i}$ where $a_i > 0$ are L-bit integers and c_i are L'-bits. Then the degree-measure bound gives

$$|E| \geq M(E)^{-1} \geq 2^{-(L+2L'+2n-2)2^{n-1}}. \tag{2.20}$$

or, in terms of root bit bound,

$$-\log_2 |E| \leq (L/2 + L' + n - 1)2^n. \tag{2.21}$$

The BFMS bound is

$$-\log_2 |E| \leq (\log_2 n + (L/2) + L')(2^n - 1). \tag{2.22}$$

The Scheinerman bound gives

$$-\log_2 |E| \leq (\log_2 n + n + L + L')(2^n - 1). \tag{2.23}$$

In 2.23, L' can be improved to $\min\{L', \max\{0, 2L' - L\}\}$ if we exploit rule 2 in Table 2.3. Lemma 2.9 says that for such division-free radical expressions, our new bound is the same as BFMS. It is shown [2] that in division-free cases, the BFMS bound is never worse than the degree-measure bound. From 2.22 and 2.23, we can see that the BFMS bound is at least as tight as Scheinerman's bound. But the next lemma strengthens this conclusion in two ways: to a broader class of expressions, and to state the comparison for each individual expression in the class.

In [2], it's shown that for division-free radical expressions, the BFMS bound is better than the degree-measure bound and Canny's bound. Here we strengthen this conclusion in showing that for this class of expressions, the BFMS bound is never worse than the Scheinerman bound.

Lemma 2.10. *For every division-free constant radical expression E, the BFMS bound for E is at least as tight as the Scheinerman bound.*

3. Next consider the sum of square roots of rational numbers. Suppose $E = \sum_{i=1}^{n}(\pm\sqrt{a_i})$ where a_i are L-bit rational numbers. Then the degree-measure root bit-bound is $2^{n-1}(n(L+2)-2)$. The BFMS bound is $2^{2n-1}(2\log_2 n + nL) - \log_2 n$. Our bound gives $2^{n-1}(2\log_2 n + (n+1)L) - (\log_2 n + L/2)$. It may be verified that both the degree-measure bound and our bound are better than the BFMS bound. The difference between our root bound and degree-measure bound in terms of bit-bound is

$$\Delta = 2^n(\log_2 n + (L/2) - n + 1) - (\log_2 n + L/2).$$

Note that Δ can be positive or negative, depending on the relative sizes of n and L. Thus our new bound is incomparable with the degree-measure bound.

4. Consider the expression $E = \frac{a+b^{2^{-r}}}{c} - \frac{d+e^{2^{-r}}}{f}$ with $2r$ $(r \geq 1)$ square roots. This is a generalization of the Fortune's predicate in 2.1. Also, we now assume that a, b, c, d, e and f are all L-bit integers. The BFMS root bit-bound is $(5L+1) \cdot 2^{4r} - (L+2)$. The degree-measure root bit-bound is $(2L+3) \cdot 2^{2r} + L \cdot 2^{r+1}$. Our new bound gives $(2L+3) \cdot 2^{2r} - 3$ which is the best.

5. Consider the continued fraction expression:

$$E_n = a_n + \cfrac{1}{a_{n-1} + \frac{1}{a_{n-2}+\cdots}} = [a_n, a_{n-1}, a_{n-2}, \ldots],$$

where the a_i's are L-bit integers. Our new bound for E_n is $(n-1)L + (n-2)$. The degree-measure bound gives $(n+1)L - 1$. And the best bound for this example is given by the BFMS approach as $(n-1)L-n+3$. This is close to the best possible bound $(n-2)L-n$. Note that the degree of E_n is 1. Now we modify E_n to get an expression E_n' with degree up to 2^n by replacing the integers a_i with square roots $\sqrt{a_i}$,

$$E_n' = [\sqrt{a_n}, \sqrt{a_{n-1}}, \sqrt{a_{n-2}}, \ldots].$$

Then our new bound for E_n' is $2^{2n-1}((n-1)L+2n-3)$. The BFMS bound gives $c \cdot 2^{2n}nL$, for some constant $c \geq 1$. Here the degree-measure approach gives the best bound $2^{n-1}(nL+2n-2)$

6. Next, we compare our new bound with the BFMS bound for a restricted class of radical expressions.

Proposition 2.11. *Given a radical expression E with rational values as the leaves but no division operations in the internal nodes, we have*

$$\overline{\mu}(E)l(E) = u(E) \tag{2.24}$$

where $\overline{\mu}(E)$ is as defined in our new bound, and $u(E), l(E)$ are defined as in the BFMS bound. Furthermore, if there are no shared radical nodes in E, then

$$\mathrm{lc}(E) \leq l(E)^{D(E)}. \tag{2.25}$$

Proof. Proof by induction on the structure of the DAG, It is easy to verify the base case when E is a constant rational number. There are four kinds of internal nodes $\{+, -, \times, \sqrt[k]{\cdot}\}$.

First we prove the Equation 2.24 inductively with the following cases:

1. $E = E_1 \pm E_2$,

$$\overline{\mu}(E)l(E) = (\overline{\mu}(E_1) + \overline{\mu}(E_2))l(E_1)l(E_2) = u(E).$$

2. $E = E_1 \times E_2$,

$$\overline{\mu}(E)l(E) = (\overline{\mu}(E_1) \cdot \overline{\mu}(E_2))l(E_1)l(E_2) = u(E).$$

3. $E = \sqrt[k]{E_1}$.

$$\overline{\mu}(E)l(E) = \sqrt[k]{\overline{\mu}(E_1)}\sqrt[k]{l(E_1)} = u(E).$$

Secondly, we prove the inequality relation 2.25. If $E = E_1 \pm E_2$ or $E = E_1 \times E_2$,

$$\begin{aligned} \mathrm{lc}(E) &= \mathrm{lc}(E_1)^{D_2}\mathrm{lc}(E_2)^{D_1} \\ &\leq l(E_1)^{D_1 D_2} l(E_2)^{D_2 D_1} \\ &\leq l(E)^{D(E)}. \end{aligned}$$

Note that by definition $D(E) = D_1 D_2$ when there are no *shared* radical or polynomial root nodes in E. When $E = \sqrt[k]{E_1}$, we have

$$\mathrm{lc}(E) = \mathrm{lc}(E_1) \leq l(E_1)^{D_1} \leq l(E)^D.$$

Q.E.D.

Lemma 2.12. *Given a radical expression E with rational values at the leaves, if E has no divisions and shared radical nodes, our new root bound for E is never worse than the BFMS bound.*

Proof.

From Proposition 2.11, we have

$$\begin{aligned} (\overline{\mu}(E)^{D(E)-1}\mathrm{lc}E)^{-1} &= ((\frac{u(E)}{l(E)})^{D(E)-1} l(E)^{D(E)})^{-1} \\ &= (u(E)^{D(E)-1} l(E))^{-1} \\ &\geq (u(E)^{D(E)^2-1} l(E))^{-1}. \end{aligned}$$

Q.E.D.

It can be shown that our bound is strictly better than BFMS for expressions in this class whenever $u(E) > 1$ and $D(E) > 1$. If shared radical nodes are permitted, our bound can be proved to be better than the BFMS bound if $(D' - D)\log_2 l(E) < D(D-1)\log_2 u(E)$, where $D'(E)$ is a degree bound of E with the value $\prod_{i=1}^{k} k_i^{c_i}$, $c_i \geq 1$ is the number of different paths from the root to the radical node r_i. Note that the proof of Lemma 2.12 implies that the quadratic exponent $D(E)^2$ in the BFMS bound is unnecessary for the class of expressions discussed in that lemma. Just $D(E)$ is enough.

2.7 Experimental Results

The new constructive root bound has been implemented in our Core Library [23]. In implementation, we maintain an upper bound on the root *bit*-bound, instead of the root bounds themselves. The logarithms are represented and manipulated as objects of the class extLong, which is basically a wrapper around the standard long number type, together with facilities to handle overflows. Such a class may be used to support "level arithmetic" in which for any integer x, we maintain the ceiling of $\lg^{(i)} x$ where $\lg^{(i)}$ denotes i iterations of $\log_2(\cdot)$, i is the smallest natural number such that $\left\lceil \lg^{(i)} x \right\rceil$ fits in built-in integer types. Therefore, the integer x is represented by a pair $(\left\lceil \lg^{(i)} x \right\rceil, i)$.

Our experiments, based on Version 1.3 of the Core Library, will compare the performance of our new bound with the BFMS and degree-measure bounds. All the tests are performed on a Sun UltraSPARC with a 440 MHz CPU and 512MB main memory. All timings are in seconds.

1. Recall the critical test in Fortune's sweepline algorithm is to determine the sign

of the expression $E = \frac{a+\sqrt{b}}{d} - \frac{a'+\sqrt{b'}}{d'}$ in 2.1 where a's, b's and d's are $3L$-, $6L$- and $2L$-bit integers, respectively. The BFMS bound requires $(79L+30)$ bits and the degree-measure (D-M) bound needs $(64L+12)$ bits. Our root bit-bound improves the bound to $(19L+9)$ bits. We generate some random inputs with different L values which always make $E = 0$, and put the timings of the tests in Table 2.8. We also converted the

Table 2.8: Timings for Fortune's expression in 2.1

L	10	20	50	100	200
NEW	0.01	0.03	0.12	0.69	3.90
BFMS	0.03	0.24	1.63	11.69	79.43
D-M	0.03	0.22	1.62	10.99	84.54

Fortune's implementation of this algorithm to use the Core Library. We ran the program on two kinds of inputs: (1) First we test on a non-degenerate data set (100 random points provided in Fortune's code distribution). The time for our new bound is 3.62 seconds while the BFMS and D-M bounds take 3.75 and 3.64 seconds, respectively. This is not unexpected, since as explained in Section 2.1, our Core Library exploits the progressive evaluation technique, and the signs of Fortune's predicate on non-degenerate inputs can be determined using the inequality 2.2. Thus the inequality 2.3 based on root bounds has no effect on the complexity. (2) We used highly degenerate inputs comprising points on a (32×32) uniform grid with coordinates being L bits long. The timings are reported in Table 2.9.

2. The second test is to verify an expression which is identically zero. Let $x = \frac{a}{b}$ and $y = \frac{c}{d}$ $(a, b, c, d$ are L-bit integers), and $E = (\sqrt{x} + \sqrt{y}) - \sqrt{x + y + 2\sqrt{xy}}$. Our new bound requires computing $(40L + 38)$ bits, while the BFMS and the degree-measure (D-M) bounds require $(640L + 510)$ and $(80L + 56)$ bits, respectively. The

Table 2.9: Timings for Fortune's algorithm on degenerate inputs

L	10	20	30	50
NEW	35.2	41.7	47.5	112.3
BFMS	86.1	1014.1	1218.1	5892.2
D-M	418.5	1681.6	1874.4	$>$ 2 hrs

timings are in Table 2.10.

Table 2.10: Timings for Example 2

L	5	10	30	50
NEW	0.08	0.09	1.77	55.43
BFMS	88.16	91.03	3071.21	$>$ 2 hrs
D-M	1.71	1.79	89.72	531.25

In comparing the timings, it is the *relative* speedup that should be stressed. We expect similar relative improvements to show up if the comparisons were made in other systems such as LEDA.

2.8 Summary

We have described a new constructive root bound for a large class of algebraic expressions. In this new approach, we need to maintain, among other things, upper bounds on the leading as well as tail coefficients of the minimal polynomial of the algebraic number E.

Our work also addresses two issues raised by the BFMS bound. First, is the quadratic factor $D(E)^2$ in the root bit-bound of E essential for radical expressions? We show that for many expressions it is not: $D(E)$ is sufficient. The second issue is whether the

BFMS technique can be extended to more general algebraic expressions. For instance, suppose we introduce a new kind of leaves into our expressions denoted by $\mathrm{Root}(P(x))$ where $P(x)$ is an integer polynomial. The framework of BFMS cannot handle this extension since there is no analogue of the $E \mapsto (U(E), L(E))$ transformation. But our new approach can be applied to any algebraic expression.

For radical expressions *without* divisions, our new bound turns out to be exactly same as the BFMS bound and is never worse than previous constructive bounds. But for those *with* divisions, a comparative performance study of the BFMS bound, the degree-measure bound and our new bound shows that they are generally incomparable. So in practice, it may be worthwhile maintaining all of them simultaneously to choose the best.

We implemented the new bound in our Core Library and experiments show that it can achieve remarkable speedup over previous bounds in the presence of division. Although we have described our bounds for the class of Ω_3-expressions, it should be clear that our methods extend to more general expressions.

Chapter 3

The Core Library: Design and Implementation

The Core Library [29, 34, 58] provides a collection of C++ classes to support numerical computation of algebraic expressions to arbitrary relative or absolute precision. In particular, it provides a base for the Exact Geometric Computation (EGC) approach to robust geometric computing. Our implementation of the library embodies the precision-driven design.

In contrast to exact integer or rational arithmetic approaches based on big number packages, the Core Library supports a broader class of *radical expressions* which are constructed from the integers and closed under a finite number of additions, subtractions, multiplications, divisions, and root extractions. The library can determine the exact sign of such radical expressions and hence is able to perform exact comparisons.

A basic goal in the design of the Core Library is to make EGC techniques transparent and easily accessible to non-specialist programmers. Built upon the Real/Expr package of Yap, Dubé and Ouchi [61, 46], our library facilitates the rapid development of robust geometric applications.

The Core Library employs an object-oriented design. Written in C++, the library has a concise but complete interface which is fully compatible with that of built-in types such as double. As the name of this library suggests, the heart of our library is indeed a numerical core. Because of its unique numerical capabilities and the precision sensitive nature, the library has other applications beyond the EGC, where guaranteeing the numerical precision is critical.

The Core Library has been publicly released and can be freely downloaded from our project homepage at

`http://cs.nyu.edu/exact/core.`

The library has been developed and tested on the Sun UltraSPARC and Intel/Linux platforms, and ported to SGI Irix and MS Windows systems.

Overview of this chapter In this chapter, we discuss the design and implementation of the Core Library. In Section 3.1, we introduce the important features of this library. In Section 3.2, we discuss the concepts of precision and error in our library, and present a new method to propagate precision requirements in an expression DAG. The design and implementation of the library is detailed in Section 3.3, with special emphasis put on the top level of the library for exact geometric computation. And in Section 3.4, we give a preliminary study of some optimization techniques. We summarize in Section 3.5.

3.1 Introduction

The most interesting part of the Core Library is its notion of expressions, embodied in the class Expr. Instances of the class Expr can be thought of as algebraic expressions built up from instances of constant rational numbers via repeated application of the four basic arithmetic operations $+, -, \times, \div$ and the square root operation $\sqrt{\ }$. A simple example of such radical expressions is

$$E = 8721\sqrt{3} - 10681\sqrt{2} - \frac{1}{8721\sqrt{3} + 10681\sqrt{2}}, \qquad (3.1)$$

whose value happens to be identically zero. An expression is represented as a directed acyclic graph (DAG) internally. We assume the constants at leaves are error-free.

A distinctive feature in our library is that it can compute the numerical approximation of an expression to arbitrary *relative* or *absolute* precision. Suppose an expression E also has a *value* $\mathrm{val}(E)$ which is exact. Unfortunately, the value $\mathrm{val}(E)$ is in the mathematical realm (here we limit our discussion to the real number field $\mathbb{R}$) and not always directly accessible. Instead, we associate with E two other quantities: a *precision bound* θ_E and an *approximation* $\widetilde{E}$. The library guarantees that given a *precision bound* θ_E, it can compute an approximate value $\widetilde{E}$ such that the distance between $\widetilde{E}$ and $\mathrm{val}(E)$ is within the bound θ_E. For instance, if we were using absolute precision, we can specify θ_E by a non-negative number α_E and this would mean that the approximation value $\widetilde{E}$ computed by the system would satisfy $|\widetilde{E} - \mathrm{val}(E)| \leq \alpha_E$.

What is important is that θ_E can be freely set by the user, and the approximation $\widetilde{E}$ is automatically computed by the system to meet this precision. In particular, if we increase the precision θ_E, then the approximation $\widetilde{E}$ will be automatically updated.

Exact comparison The Core Library supports exact comparison among real algebraic expressions. While we can generate arbitrarily accurate approximations to an expression, this does not in itself allow us to do exact comparisons. Without root bounds, when we compare two numbers that happen to be equal, generating increasingly accurate approximations can only increase our confidence that they are equal, but cannot tell us whether they must be equal. For example, in Maple, when evaluating the expression (3.1) with a decimal precision 1000, it outputs $1.046985018 \times 10^{-995}$ and gives a negative answer on the query "$E = 0$?". Our Core Library can verify it positively within 0.005 seconds on a Sun UltraSPARC workstation. In order to separate an algebraic number away from zero, we need some elementary theory of algebraic root bounds [62, 35]. As we have discussed in Chapter 2, this is the basis of the Exact Geometric Computation (EGC) approach. We use a simple example to illustrate the importance of exact comparison in predicate evaluation within geometric programs:

Example 3.1. We construct a plane P in $\mathbb{E}^3$ with the equation $x + y + z = 1$ and intersect P with lines L_{ij} $(i, j = 1, 2, \ldots, 50)$ though the origin $(0, 0, 0)$ and the point $(i, j, 1)$. We then test if the intersection point $P_{ij} = L_{ij} \cap P$ lies on the plane P. When implemented using the Expr class in our library, the answer is positive in all the 2500 cases. But with the machine's built-in floating-point numbers, the answer is correct only in 1538 cases (62.5%).

Ease of use Our library is easy to use. It provides a number of "exact" data types whose syntax and semantics are compatible with those of primitive types, such as double in C++. It requires little extra effort for users to implement new robust geometric applications using our library. Typically, an existing geometric program can be made robust simply by inserting a preamble to redefine the number types used in the program to our "exact" data types. In our current distribution of the library, there are a number of demo programs which are simply taken from some standard implementations. For example, we converted O'Rourke's implementations [45] of Graham's algorithm for 2D convex hull, the incremental algorithms for 3D convex hull and 2D Delaunay triangulation. For these examples, the major modifications made are the conversion of the `printf` statements with double arguments to the stream I/O operations in C++, because numbers in our library are C++ objects and cannot use the built-in `printf` statements. Steven Fortune's C implementation of his sweepline algorithm [13] for the Voronoi diagram of a planar point set has also been converted. Generally, in converting unfamiliar existing programs, we find the most difficult part is to locate the critical data and predicates that need to be promoted for exact computation. Moreover, a number of language issues (such as the `printf` problem, etc.) should also be addressed. We refer interested readers to our tutorial on the library [34] for more details.

Precision-driven computation and active error tracking The evaluation of expressions is driven by the precisions explicitly specified or implicitly needed. The system first propagates the precision requirements down the DAG in a way that guarantees the approximation values computed from bottom-up in a later step must satisfy the precision bound θ_E at each node E.

Along with the approximation of an expression, we also compute an error bound $\mathrm{Err}_{\widetilde{E}}$ of the approximation such that it is guaranteed that $|\widetilde{E} - \mathrm{val}(E)| \leq \mathrm{Err}_{\widetilde{E}}$. This bound is obtained through forward error analysis on the approximation process, based on some standard interval arithmetic techniques. For instance, if $\widetilde{E} = \widetilde{E_1} + \widetilde{E_2}$ then the error bound in $\widetilde{E}$ is essentially [1] determined by the error bounds $\mathrm{Err}_{\widetilde{E_1}}$ and $\mathrm{Err}_{\widetilde{E_2}}$. In this sense, we say that error bounds are *á posteriori values* while precision bounds are *á priori values.*

Usually the error bound provides us with a more accurate estimate on the absolute

[1]In operations such as division or square roots, if the operands have no error, then we rely on some global parameter to bound the error in the result.

error of the current approximation than the precision bound does. Thus, before we re-evaluate an expression to higher precision, we always check the new precision requirement against the current error bound to see whether the existing approximation has already satisfied the new precision. If not, the re-evaluation will proceed. Otherwise, we just return the current approximate value.

Precision sensitive approach The Core Library follows the precision sensitive approach [61] to EGC. In exact sign determination, instead of computing to the precision inferred by root bounds directly, we increase the precision of approximation incrementally step by step, until the sign comes out for sure, or we hit the root bound.

Numerical I/O The expressions in our library support stream I/O in C++. Our system can read arbitrarily long numbers, in either positional or scientific notation, from input streams into expressions. Alternatively, there is an expression constructor from strings. In general, a decimal number such as 1.23 cannot be exactly represented in the binary format. Thus the system allows users to specify a precision for reading input numbers, or to choose to read them in exactly as rational numbers.

As for the output of expressions, the library does not output the structure of expressions. Instead, it prints out a numerical approximation of expressions. Note that this might result in the loss of the exact value of an expression. The numerical approximation can be printed in both positional and scientific notations. In both formats, users can specify the number of digits to be printed, provided there are that many *correct* digits in the approximation. In general, all the output digits are correct except that the last digit may be off by ± 1. Note that an output of 19.99 is considered correct for the value 20.00, according to this convention. Of course, the approximate value of an expression can be improved to as many correct significant digits as we want (but the user will have to force a re-evaluation before output).

COREX **extensions** The Core Library is an open system which incorporates a design idea that a light-weight general-purpose numerical core should be complemented by various extensions (called COREXs) that include a collection of robust implementations of frequently-used APIs. Various domain specific knowledge can be embedded into separate extension package. In our distribution, we include two simple COREX's, for linear algebra and for geometry, respectively.

Comparison with big number packages The Core Library is built upon the big number packages. Nevertheless, there are some fundamental differences between our library and big number packages. First, our library employs some elementary algebra knowledge about root bounds and hence is able to support exact computation of radical expressions. The traditional big number packages can only support rational computation. When root extractions are introduced, no single big number package alone can handle the exact comparison problem. Secondly, our library employs a novel precision-driven approach, while big number packages are usually precision insensitive and always compute to full accuracy up front regardless of whether this is really necessary.

3.2 Numerical Precision

It is important to understand the concepts of precision and error as used in our system. In this section, we give their definitions and discuss their roles and interaction in our system.

3.2.1 Definitions

We now use a notation from [61] to specify the nature of the precision bound θ_E.

Given a real number X, and "extended" integer numbers (i.e., $\mathbb{Z} \cup \pm\infty$) a and r, we say that a real number $\widetilde{X}$ is an *approximation* of X to the *(composite) precision* $[r, a]$, denoted

$$\widetilde{X} \simeq X\,[r, a]\,,$$

provided either

$$\left|\widetilde{X} - X\right| \leq 2^{-r}\,|X| \qquad \textit{or} \qquad \left|\widetilde{X} - X\right| \leq 2^{-a}.$$

Intuitively, r and a bound the number of "bits" of relative and absolute error (respectively) when $\widetilde{X}$ is used to approximate X. Note that we use the "or" semantics (either the absolute "or" relative error has the indicated bound). In the above notation, we view the combination "$X[r, a]$" as the given data (although the number X is really a black-box which might not be able to be explicitly represented) from which our system is able to generate an approximation $\widetilde{X}$. For any given data $X[r, a]$, we are either in the "absolute regime" (if $2^{-a} \geq 2^{-r}|X|$) or in the "relative regime" (if $2^{-a} \leq 2^{-r}|X|$).

To force a relative precision of r, we can specify $a = \infty$. Thus $X[r, \infty]$ denotes any $\widetilde{X}$ which satisfies $\left|\widetilde{X} - X\right| \leq 2^{-r}\,|X|$. Likewise, if $\widetilde{X} \simeq X[\infty, a]$ then $\widetilde{X}$ is an approximation of X to the absolute precision a, $|\widetilde{X} - X| \leq 2^{-a}$.

In implementation, r and a are extLong values. We use two default global variables to specify the global composite precision:

$$[\texttt{defRelPrec}, \texttt{defAbsPrec}].$$

It has the default value $[35, \infty]$. The user can change these values at run time.

Sometimes, we want to control this precision for individual variables. If `e` is an Expr object, the user can compute an approximation within the composite precision `[rel, abs]` by invoking the member function `e.approx(rel, abs)`. The function returns a Real instance that satisfies this requested precision. If `e.approx` is called without any arguments, it will use the global precision settings [`defRelPrec`, `defAbsPrec`].

3.2.2 Bounds on the Magnitude of Expressions

We need to know the magnitude of the value (i.e., $|E|$, or for simplicity $\lg(|E|)$) of an expression E in order to transform a relative error bound to an absolute one (e.g., in deciding which component in a composite precision bound is the weaker one that we shall follow). Such information is also needed in propagating precision bounds (see Section 3.2.3 below).

Intuitively, the logarithm form $\lg(|E|)$ tells us about the location of the first significant bit. Hereafter we simply call $\lfloor \lg(|E|) \rfloor$ the *Most Significant Bit* (or MSB) of an expression. By definition, the MSB of 0 is $-\infty$.

In general, $\lg(|E|)$ can be computed to arbitrary precision. But doing so at each node is quite expensive and is not always necessary. In practice, we compute an interval $[\mu_E^-, \mu_E^+]$ that contains $\lg(|E|)$. Table 3.1 gives the basic rules to compute such an interval inductively. Here, μ_E^- and μ_E^+ are lower and upper bounds on $\lg(|E|)$. Usually (e.g., in our Core Library), for simplicity and efficiency, the end-points of this interval are chosen to be integers. For example, in the first row, $\lg(\frac{a}{b})$ may be replaced by $\lceil \lg(\frac{a}{b}) \rceil$ and $\lfloor \lg(\frac{a}{b}) \rfloor$, respectively. We omit further implementation details here.

Note that because there is potential cancellation in additions or subtractions, we cannot derive the lower bounds for these two operations in terms of bounds about E_1

Table 3.1: Rules for upper and lower bounds on $\lg(|E|)$

E	μ_E^+	μ_E^-
rational $\frac{a}{b}$	$\lg(\frac{a}{b})$	$\lg(\frac{a}{b})$
$E_1 \pm E_2$	$\max\{\mu_{E_1}^+, \mu_{E_2}^+\} + 1$	$\lfloor \lg(E) \rfloor$
$E_1 \times E_2$	$\mu_{E_1}^+ + \mu_{E_2}^+$	$\mu_{E_1}^- + \mu_{E_2}^-$
$E_1 \div E_2$	$\mu_{E_1}^+ - \mu_{E_2}^-$	$\mu_{E_1}^- - \mu_{E_2}^+$
$\sqrt[k]{E_1}$	$\mu_{E_1}^+ / k$	$\mu_{E_2}^- / k$

and E_2 only. If this happens, we approximate E to the first significant bit (i.e., with the relative error 0.5) to get the the largest integral value not greater than $\lg(|E|)$. Here root bounds determine the worst case complexity in this computation. However, when there is no cancellation (e.g., two operands in an addition having the same sign), we can use a simpler rule for this lower bound: $\mu_{E_1 \pm E_2}^- = \max\{\mu_{E_1}^-, \mu_{E_2}^-\}$.

3.2.3 Propagation of Precision Requirements

Suppose E is an expression represented as a DAG. Given a composite precision requirement

$$\theta_E = [r_E, a_E]$$

on E, the system will propagate this precision requirement down the DAG.

First, it needs to determine which part (either r_E or a_E) in the composite precision is the dominant one in our convention, and to translate the effective component bound into a bound α_E on the maximum absolute error allowed. The following rule computes

α_E from θ_E and $\mathrm{val}(E)$:

$$\alpha_E = \begin{cases} 2^{-a_E} & \text{if } r_E = +\infty; \\ 2^{-r_E}|\mathrm{val}(E)| & \text{if } a_E = +\infty; \\ \max\{2^{-r_E}|\mathrm{val}(E)|, 2^{-a_E}\} & \text{otherwise.} \end{cases}$$

Note that actually the last rule alone suffices. But in practice, it is more efficient to detect the specific cases $r_E = +\infty$ or $a_E = +\infty$ first. Moreover, we shall note that the above transformation depends on the value of E. In Section 3.2.2, we have shown how to bound this value.

Next, we can propagate these (absolute) precision bounds in a top-down fashion using the rules presented in Table 3.2, where α_E means the absolute precision requirement imposed on the node E. For simplicity, we denote α_{E_i} as α_i. The notation $(E)_p$ means to evaluate the expression E within the maximum absolute error p. Note that for addition, subtraction and multiplication operations, the computation of $\widetilde{E}$ is performed exactly. But for division and square root operations, our rules allow an absolute error up to $\frac{\alpha_E}{2}$.

Justification of precision propagation rules in Table 3.2

Lemma 3.1. *Let E be a node in an expression DAG. Assume that the precision bounds are assigned to operand(s) according to the rules in Table 3.2, and the approximate value of each operand(s) satisfies these precision bounds. Then the approximation $\widetilde{E}$ at each node E (see the third column of that table) satisfies that*

$$|E - \widetilde{E}| \leq \alpha_E,$$

where α_E is the precision bound required on the node E.

Table 3.2: Syntax-guide propagation of precision bounds

E	Rules	$\widetilde{E}$
$E_1 \pm E_2$	$\alpha_1 = \alpha_2 = \frac{1}{2}\alpha_E$	$\widetilde{E} = (\widetilde{E_1} \pm \widetilde{E_2})_0$
$E_1 \times E_2$	If $\alpha_E \leq 8\|E\|$ then $\alpha_1 = \alpha_E/(4\|E_2\|)$, $\alpha_2 = \alpha_E/(4\|E_1\|)$ else $c = \left\lfloor \sqrt{\alpha_E/\|E\| + 1} \right\rfloor - 1$, and $\alpha_1 = c \cdot \|E_2\|$, $\alpha_2 = c \cdot \|E_1\|$	$\widetilde{E} = (\widetilde{E_1} \times \widetilde{E_2})_0$
$E_1 \div E_2$	$\alpha_1 = \alpha_E\|E_2\|/4$, $\alpha_2 = \frac{\alpha_E\|E_2\|^2}{4(\|E_1\|+\alpha_1)+\alpha_E\|E_2\|}$	$\widetilde{E} = (\widetilde{E_1} \oslash \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$\sqrt[k]{E_1}$	$\alpha_1 = \alpha_E\sqrt{E_1}/2$	$\widetilde{E} = (\widetilde{\sqrt{\widetilde{E_1}}})_{\frac{\alpha_E}{2}}$

Proof.

Proof by induction on the structure of the DAG representation of the expression E.

In the base case $E = \frac{a}{b}$, it is known that we can approximate a rational constant to arbitrary precision.

The intermediate node E can be any of a number of possible types as follows. By the induction hypothesis, we know that this lemma is true for every children of E.

1. $E = E_1 \pm E_2$. The approximate value $\widetilde{E} = \widetilde{E_1} \pm \widetilde{E_2}$. Note that here we assume that the additions and subtractions of two approximate operands are handled exactly. Thus,

$$\begin{aligned} |E - \widetilde{E}| &= |(E_1 \pm E_2) - (\widetilde{E_1} \pm \widetilde{E_2})| \\ &\leq |E_1 - \widetilde{E_1}| + |E_2 - \widetilde{E_2}| \\ &\leq \alpha_1 + \alpha_2 \end{aligned}$$

2. $E = E_1 \times E_2$. The multiplication in computing $\widetilde{E} = \widetilde{E_1} \times \widetilde{E_2}$ is exact. Thus, when $\alpha_E \le 8|E|$, we have

$$\begin{aligned}
|E - \widetilde{E}| &\le |E_1|\alpha_2 + |E_2|\alpha_1 + \alpha_1\alpha_2 \\
&\le \frac{\alpha_E}{4} + \frac{\alpha_E}{4} + \frac{\alpha_E^2}{16|E_1E_2|} \\
&\le \alpha_E
\end{aligned}$$

If $\alpha_E > 8|E|$,

$$\begin{aligned}
|E - \widetilde{E}| &\le |E_1|\alpha_2 + |E_2|\alpha_1 + \alpha_1\alpha_2 \\
&\le (2c + c^2)|E| \\
&\le \alpha_E/|E| * |E| = \alpha_E
\end{aligned}$$

3. $E = E_1 \div E_2$. The division in computing $\widetilde{E} = \widetilde{E_1} \oslash \widetilde{E_2}$ might not be exact and can has an error at most $\alpha_E/2$. Moreover, it is easy to verify that $\alpha_2 \le |E_2|$. Therefore,

$$\begin{aligned}
|E - \widetilde{E}| &\le |\frac{E_1}{E_2} - \frac{\widetilde{E_1}}{\widetilde{E_2}}| + |\frac{\widetilde{E_1}}{\widetilde{E_2}} - \widetilde{\frac{\widetilde{E_1}}{\widetilde{E_2}}}| \\
&\le \frac{|\widetilde{E_1}|\alpha_2 + |\widetilde{E_2}|\alpha_1}{|E_2\widetilde{E_2}|} + \frac{\alpha_E}{2} \\
&\le \frac{|\widetilde{E_1}|\alpha_2}{|E_2\widetilde{E_2}|} + \frac{3\alpha_E}{4} \\
&\le \frac{\alpha_2(|E_1| + \alpha_1)}{|E_2|(|E_2| - \alpha_2)} + \frac{3\alpha_E}{4} \\
&\le \frac{\alpha_E|E_2|^2}{4(|E_1| + \alpha_1) + \alpha_E E_2} \cdot \frac{(|E_1| + \alpha_1)}{|E_2|} \cdot \frac{1}{|E_2| - \frac{\alpha_E|E_2|^2}{4(|E_1|+\alpha_1)+\alpha_E E_2}} \\
&\quad + \frac{3\alpha_E}{4} \\
&\le \alpha_E
\end{aligned}$$

The implicit restriction that $\alpha_2 \leq |E_2|$ is necessary in divisions, because basically this means that the divisor $\widetilde{E_2}$ cannot be zero. Otherwise, since the error depends on $\frac{1}{|\widetilde{E_2}|}$, it might be potentially unbounded.

4. $E = \sqrt{E_1}$. Assume the square root operation in computing $\widetilde{E} = \widetilde{\sqrt{\widetilde{E_1}}}$ has a precision of $\theta_E/2$. Thus,

$$\begin{aligned} |E - \widetilde{E}| &\leq |E - \sqrt{\widetilde{E_1}}| + |\sqrt{\widetilde{E_1}} - \widetilde{\sqrt{\widetilde{E_1}}}| \\ &\leq \frac{|E_1 - \widetilde{E_1}|}{\sqrt{E_1} + \sqrt{\widetilde{E_1}}} + \frac{\theta_E}{2} \\ &\leq \frac{\theta_{E_1}}{\sqrt{E_1}} + \frac{\theta_E}{2} \\ &\leq \theta_E \end{aligned}$$

Q.E.D.

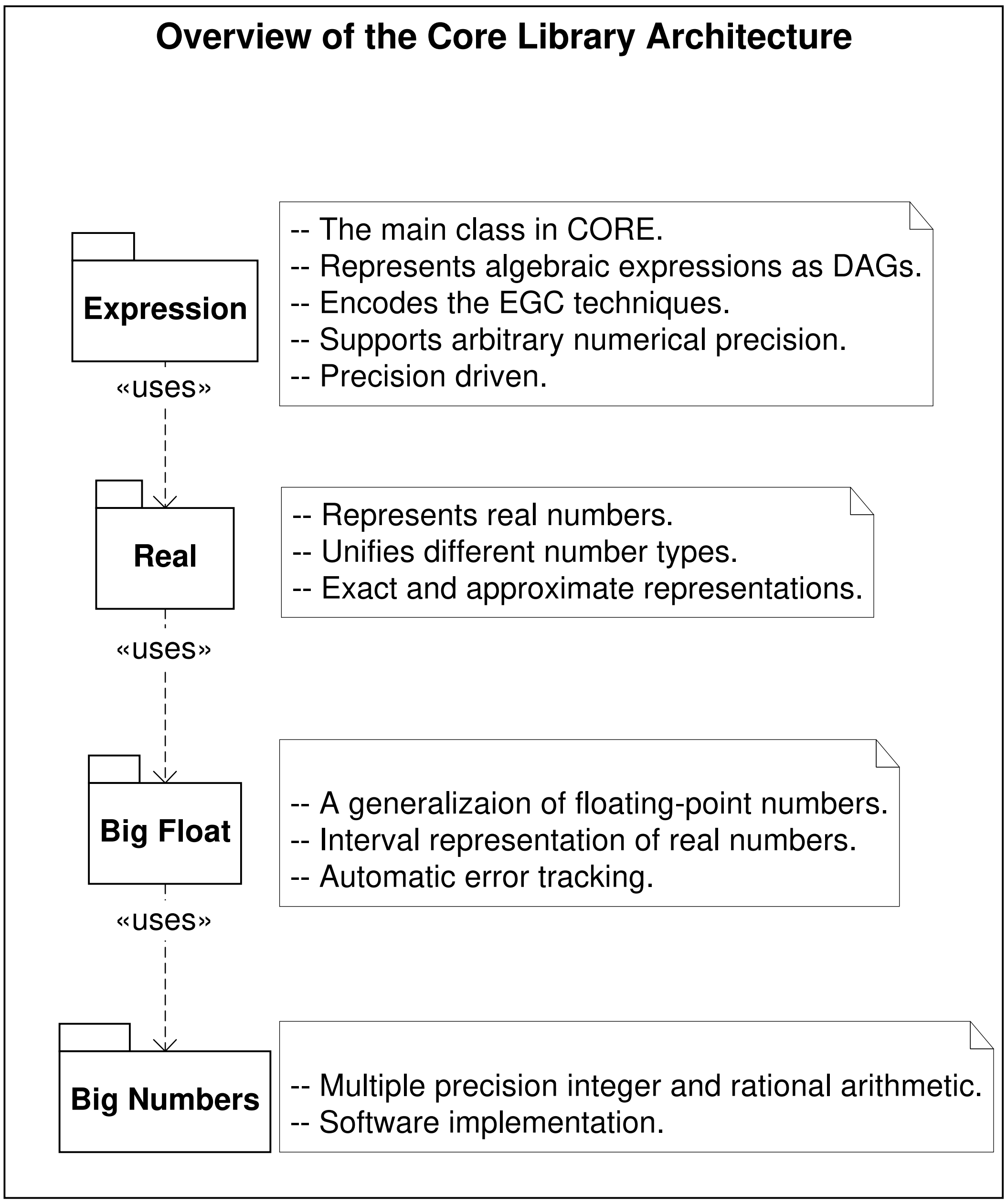
3.3 Design of the Library

The library features an object-oriented design and is implemented in C++. From a user's point of view, the library is a collection of "exact" numerical data types (in the form of C++ classes) which can be used in the same way as built-in primitive types such as int or double. Our library supports exact computation of algebraic expressions.

3.3.1 Overview of the Core Library Architecture

There are four main subsystems in the Core Library: expression (Expr), real numbers (Real), big floating-point numbers (BigFloat) and big integer/rational numbers. They are built up in a layered structure (see Figure 3.1). The Expr package at the top level provides the basic functionalities of exact geometric computation. Generally this consists the only interface that users need to program with. But experienced users can also access the unique numerical capabilities of underlying number classes directly.

Figure 3.1: Overview of the system architecture.


Overview of the Core Library Architecture
Expression
-- The main class in CORE.
-- Represents algebraic expressions as DAGs.
-- Encodes the EGC techniques.
-- Supports arbitrary numerical precision.
-- Precision driven.
«uses»
Real
-- Represents real numbers.
-- Unifies different number types.
-- Exact and approximate representations.
«uses»
Big Float
-- A generalizaion of floating-point numbers.
-- Interval representation of real numbers.
-- Automatic error tracking.
«uses»
Big Numbers
-- Multiple precision integer and rational arithmetic.
-- Software implementation.

Here is a brief summary of these sub-systems and their interactions:

Expr is the most important class of the library and provides the mechanism for exact geometric computation. It represents expressions that are constructed from rational constants by repeated application of the four basic arithmetic operations $\{+, -, *, /\}$ and square root. Internally, expressions are represented as directed acyclic graphs. The constants at leaves are instances of the Real class and are assumed error-free in our system. We can approximate an expression to arbitrary precision. The approximate value, along with its associated error, if any, is represented as a Real object.

Real is a "heterogeneous" number system that currently incorporates the following six subtypes: int, long, double, Integer, Rational, and BigFloat. The first three are standard machine primitive types with fixed precision while the latter three are multiple precision number types which are built upon some big number packages. Especially, BigFloat is an *interval* representation of real numbers. The role of Real is to integrate these different types and to provide a unified interface for real number arithmetic.

BigFloat is an arbitrary precision floating point number representation that we built on top of Integer. A BigFloat is represented by the triple $\langle m, \varepsilon, e\rangle$ where m is the mantissa of type Integer, ε is the error bound and e is the exponent. It represents the interval $[(m-\varepsilon)B^e, (m+\varepsilon)B^e]$ where $B = 2^{14}$. These intervals are automatically maintained when performing arithmetic with BigFloat's. The BigFloat is used by our library to represent approximate values and the associated numerical errors in computation.

BigNumbers This sub-system includes the implementation of multiple precision integer Integer and rational number Rational. The arithmetic among them is exactly handled, but not precision sensitive. There are a number of big number packages available. In Version 1.3 of the library, we incorporate the LiDIA's extendible interface, and use CLN's Integer and Rational as the kernel.

3.3.2 Expressions

This package captures a class of algebraic expressions in general geometric computation.

Definitions In our system, *expressions* refer to those which can be constructed from rational constants by a finite number of repeated application of the four basic arithmetic operations $\{+, -, *, /\}$ and square root. The value of an expression is a real algebraic number in $\bar{\mathbb{Z}}$. We assume the constants at leaves are error free. Given an arbitrary relative or absolute *precision bound*, the system can approximate the value of an expression within the specified precision bound.

An instance of the class Expr E is formally a triple

$$E = (T, P, A)$$

where T is an *expression tree*, P is a composite precision, and A is some real number or $\uparrow$ (undefined value). The internal nodes of T are labeled with one of those operators

$$+, -, \times, \div, \sqrt{\cdot}, \tag{3.2}$$

and the leaves of T are labeled by Real values or is $\uparrow$. $P = [r, a]$ is a pair of extLongs. If all the leaves of T are labeled by Real values and the operations at each decedent node is well defined, then there is a real number V that is the value of the expression T; otherwise, $V =\uparrow$. Finally, the value A satisfies the relation

$$A \simeq V[r, a].$$

This notation was explained in Section 3.2. This is interpreted to mean either $V = A =\uparrow$ or A approximates V to precision P. In the current implementation, leaves must hold exact values. Moreover, the value A is always a RealBigFloat, a subtype of Real and a wrapper around BigFloat.

Value semantics Expr, the main class in this package, provides basic functionalities of creating and manipulating expressions. The Expr adopts the standard *value semantics*. That is, after an assignment $s_1 = s_2$, the two Exprs s_1 and s_2 are fully distinct and subsequent changes to the one have no effect on the other. An alternative is "pointer semantics", which was used in the Real/Expr package. That would let changes to s_2 after the assignment $s_1 = s_2$ also affect the value of s_1.

Representation Expressions are represented internally as directed acyclic graphs which record the history of computation in constructing them in client programs.

For the value semantics to be affordable, an Expr is implemented as a handle to its representation and the representation is copied only when necessary (e.g. copy-on-write). This is realized by introducing a representation class ExprRep which forms the core of this package. As we will see, the expression DAG and most functionalities at each nodes are actually implemented in ExprRep first and then simply wrapped under the interface of Expr.

The nodes in expression trees are instances of the class ExprRep. More precisely, each instance of Expr has a member *rep* that points to an instance of ExprRep. Each instance of ExprRep is allocated on the heap and has a type, which is either one of the operations in 3.2 or type “constant”. Depending on its type, each instance of ExprRep has zero, one or two pointers to other ExprRep(s). For instance, a constant ExprRep, a $\sqrt{\cdot}$-ExprRep and a +-ExprRep has zero, one and two pointers, respectively. The collection of all ExprReps together with their pointers constitute a directed acyclic graph (DAG). Every node N of this DAG defines a sub-expression tree $E(N)$ in the natural way.

The separation of an expression object and its value representation allows multiple objects with the same value to share a single representation of that value. For example, considering an assignment statement (see Figure 3.3)

$$s_2 = s_1,$$

instead of giving s_2 a copy of s_1’s value, we have s_2 share s_1’s value. This not only saves space, but also leads to faster-running programs, because there’s no need to construct and destruct redundant copies of the same value. All we have to do is a little bookkeeping so we know who’s sharing what, and in return we save the cost of a call to new and the expense of copying anything. The fact that the objects s_1 and s_2 are sharing a data structure is transparent to clients. In fact, the only time the sharing of values makes a difference is when one or the other objects is modified; then it’s important that only one object is changed, not the others which are sharing the same value with it. In this case, if there is sharing, we usually create a separate copy of that value and then make changes on this new instance. The reference count will help us in deciding which case applies, and more important, can simplify the bookkeeping work surrounding heap objects.

Class Hierarchy The expression DAG is implemented through the abstract class ExprRep. Each instance of ExprRep represents a node in the DAG. We can categorize the node types into three classes based on the number of operands: constants, unary operations and binary operations. Three subtypes of ExprRep can be defined accordingly:

ConstRep represents constants in an expression. They always appear in the leaves. The value of an instance of ConstRep is stored in the data member $value$ of the type Real. We assume the value is an error-free rational number (e.g., fixed precision floating-point numbers are also rational) and hence can be approximated to arbitrary precision.

UnaryOpRep is an abstraction of unary operators including negation and square root. It holds a single pointer to its only child (operand). The two classes NegRep and SqrtRep implement this abstract interface.

BinOpRep is an abstraction of binary arithmetic operators, including addition, subtraction, multiplication and division. These four basic arithmetic operations are implemented in the four derived classes of BinOpRep: AddRep, SubRep, MultRep and division DivRep, respectively.

We show the class hierarchy diagram in Figure 3.2.

Construction and arithmetic of expressions An expression can be constructed starting from constants. The constructors in the Expr class can build an Expr object from numbers of primitive data type (such as int, double, etc.), big integer or rational numbers, error-free BigFloat and Real objects, and other Expr instances.

The basic arithmetic operators (minus, $+, -, \times, \div$) and the square root function have been overloaded. In this way, expressions are recursively constructed as the natural result of arithmetic operations.

Basically, in a user application, the class Expr can be programmed in the same way as it is a primitive data type. As an example, the code to verify the identity of the expression E in (3.1) is as follows:

```
#define Level 3
#include "CORE.h"

int main() {
    double x = 2;
    double y = 3;
    double sx = sqrt(x);
    double sy = sqrt(y);
    double e = 8721 * sy - 10681 * sx
            - 1 / (8721 * sy + 10681 * sx);
    cout << ((e == 0) ?  "yes (CORRECT)" :
            "no (INCORRECT)") << endl;
}
```

In the beginning of this program, we define a flag to specify the accuracy level in our library. In the level 3, all the doubles are redefined to be Expr objects that support exact computation.

We shall note that in our implementation of arithmetic operators, no numerical computation is actually performed. Instead, we employ the strategy of lazy expression evaluation. In the above example, no actual computation is done at the initial assignment of the object e. The real computation is postponed until the comparison embedded in the `cout` statement. In general, when an arithmetic operation such as $s_1 + s_2$ is executed, it creates a temporary Expr object for storing the result. This object contains only a pointer to a newly created data structure of the type AddRep, with its two children pointers directed to s_1 and s_2, which indicates that its value is the sum of these two children. In this strategy, we defer the real numerical evaluation of expressions until the result is really needed. Moreover, our evaluation is designed to be precision driven. That is, we only compute an expression to the precision that is required in a computation.

We compute the bounding interval for $\lg(|\mathrm{val}(E)|)$ at each node E at the construction time using the rules specified by Table 3.1 in Section 3.2.3. Note that because of possible cancellations, we might need to make a number of iterations before we can determine the lower bound of the interval at addition and subtraction nodes.

Evaluation of expressions The evaluation of an expression to specified precision generally includes two steps:

1. A top-down propagation of precision bounds in the expression DAG using the rules we present in Section 3.2.

2. Computation of the numerical approximation and error bound at each node in a bottom-up fashion. The error shall not exceed the precision bounds specified for that same node.

The approximate value is represented as a Real object. When computing approximate values, we also track the maximum possible error of an approximation. In Step 2, it is unnecessary to proceed below a node if the error of its current approximation already satisfies the precision bound required on it.

3.3.3 The Real Package

Built upon the Real/Expr library, the Real package integrates six data types:

$$\mathsf{int},\ \mathsf{long},\ \mathsf{double},\ \mathsf{Integer}\ ,\ \mathsf{Rational},\ \text{and}\ \mathsf{BigFloat}.$$

The class Real encapsulates implementation details and presents a uniform interface to these different representations of real numbers. There is a natural type coercion relation among these types as one would expect. It is as follows:

$$\mathsf{int} \prec \mathsf{long} \prec \mathsf{double} \prec \mathsf{BigFloat} \prec \mathsf{Rational}\,,$$
$$\mathsf{int} \prec \mathsf{long} \prec \mathsf{Integer} \prec \mathsf{Rational}\,.$$

The BigFloat in this coercion is assumed to be error-free. But in general, BigFloat is an interval representation of real numbers. Besides BigFloat, all the other data types are considered as “exact” in the sense that they unambiguously correspond to some unique real number. Clearly, any “exact” representation can be converted to a BigFloat with errors.

The Real class implements the usual value semantics. However, unlike the Expr class, it does not keep a tree structure. In arithmetic operations, the corresponding numerical computation is performed immediately and then the result is put in another Real object.

In the Core Library the class Real is used as the carrying vehicle of approximate values and error estimates, and as the tool to manipulate the approximate values in a unified and predictable manner.

3.3.4 The BigFloat Class

A BigFloat number x is given as a triple $\langle m, err, exp\rangle$ where m is the *mantissa*, the *error-bound* $err \in \{0, 1, \ldots, B-1\}$ and exp is the *exponent*. Here the *base* B is equal to 2^{14}. For efficiency reasons, we use the normalized notation.

The "number" x really represent the interval

$$[(m - err)\, B^{exp}, (m + err)\, B^{exp}] \tag{3.3}$$

We say that a real number X *belongs* to x if X is contained in this interval. We inherit the BigFloat class from the Real/Expr package [46]. In our implementation, m is Integer, err is `unsigned long`, and exp is `long` for efficiency. Version 1.3 uses the `LiDIA`/`CLN` as the big number kernel.

If $err = 0$ then we say the BigFloat x is *error-free*. When we perform the operations $+, -, *, /$ and $\sqrt{\ }$ on BigFloat numbers, the error-bound is automatically maintained subject in the following sense: *if X belongs to* BigFloat x *and Y belongs to* BigFloat y*, and we compute* BigFloat $z = x \circ y$ *(where $\circ \in \{+, -, \times, \div\}$) then $X \circ Y$ belongs to* z. A similar condition holds for the unary operations. In other words, the error-bound in the result z must be "large enough" to contain all the possible results.

There is leeway in the choice of the error-bound in z. Basically, our algorithm tries to minimize the error-bound in z subject to efficiency and algorithmic simplicity. This usually means that the error-bound in z is within a small constant factor of the optimum error-bound (see Koji's thesis [46] for more details). But this may be impossible if both

x and y are error-free: in this case, the optimum error-bound is 0 and yet the result z may not be representable exactly as a BigFloat. This is the case for the operations of $\div$ and $\sqrt{\cdot}$. In this case, our algorithm ensures that the error in z is within some default precision (the value of global variable `defAbsPrec`), or within some precision specified.

A practical consideration in the design of the class BigFloat is that we insist that the error-bound err is at most B. To achieve this, we may have to truncate the number of significant bits in the mantissa m and modify the exponent exp appropriately at the same time.

3.4 System Optimization

The Core Library effectively addresses the robustness concerns in geometric computing. However, this often comes at the cost of efficiency. In this section, we discuss some system and compilation techniques to improve the efficiency [29].

Because of the need to store dependencies between values; to maintain data structures that can store values, dependencies; and to overload operators like assignment, copying, and arithmetic, expression evaluation in Core Library requires a lot of software work and is considerably slower than that of those primitive data types, such as double, which are widely supported in current computer hardware.

Expression evaluation involves a recursive traversal of the expression DAG and iterated traversals may be necessary because of the precision-sensitive nature of our Core Library. Maintaining the explicit expression DAG guarantees robustness, but it reduces execution efficiency on current-day pipelined computer systems with deep memory hierarchies. Here we use the Gaussian elimination method for matrix determinants as an example. A frequently-used expression in this algorithm is

$$A(i,j) \ -= \ A(j,i) * A(i,k)/A(i,i).$$

We can exploit the fact that the three operations in this expression are fixed and known, and encode the type information into a new node type just for such composite mul-div-sub operations. In this way, we reduce the complexity of type inference and get rid of some runtime cost such as maintaining runtime type information, virtual function resolution, dynamic memory management to the compilation stage.

Moreover, some code specification techniques such as inlining and cloning are used to increase the effectiveness of traditional sequential optimizations. We also implement our own customized memory management routines which significantly reduces the cost of dynamic memory management based on the default generic allocator and delocator.

We conducted some pilot studies [29] which have shown these system techniques can speed up the overall performance by a factor of 2.

3.5 Summary

In this chapter, we present the Core Library, a library for exact numeric and geometry computation. The main features of this library include:

- Users can use our library in developing robust software which is free from intractable numerical errors in ordinary floating-point arithmetic.
- The library supports numerical computation of algebraic expressions to arbitrary relative or absolute precision. The running time is precision sensitive.
- In contrast to traditional exact integer or rational arithmetic approaches based on big number software packages, our library provides better, precision-sensitive

performance behavior. Moreover, our library support a broader class of algebraic expressions.

- It is very easy to use our library in development. Our design of the library features a nearly transparent integration with the conventional C++ programming style. In most cases, it simply amounts to substituting the imprecise primitive number types with the "exact" data types provided by the Core Library. No special knowledge regarding numerical analysis and non-robustness problems is needed. There is also no need to modify the underlying program logic.
- The library is of compact size and can be easily extended.

A challenge for future work is to further improve the efficiency of the Core Library. A number of topics to be explored include incremental computation, compilation and partial evaluation of expressions.

Figure 3.2: Class hierarchy in the Expr package.

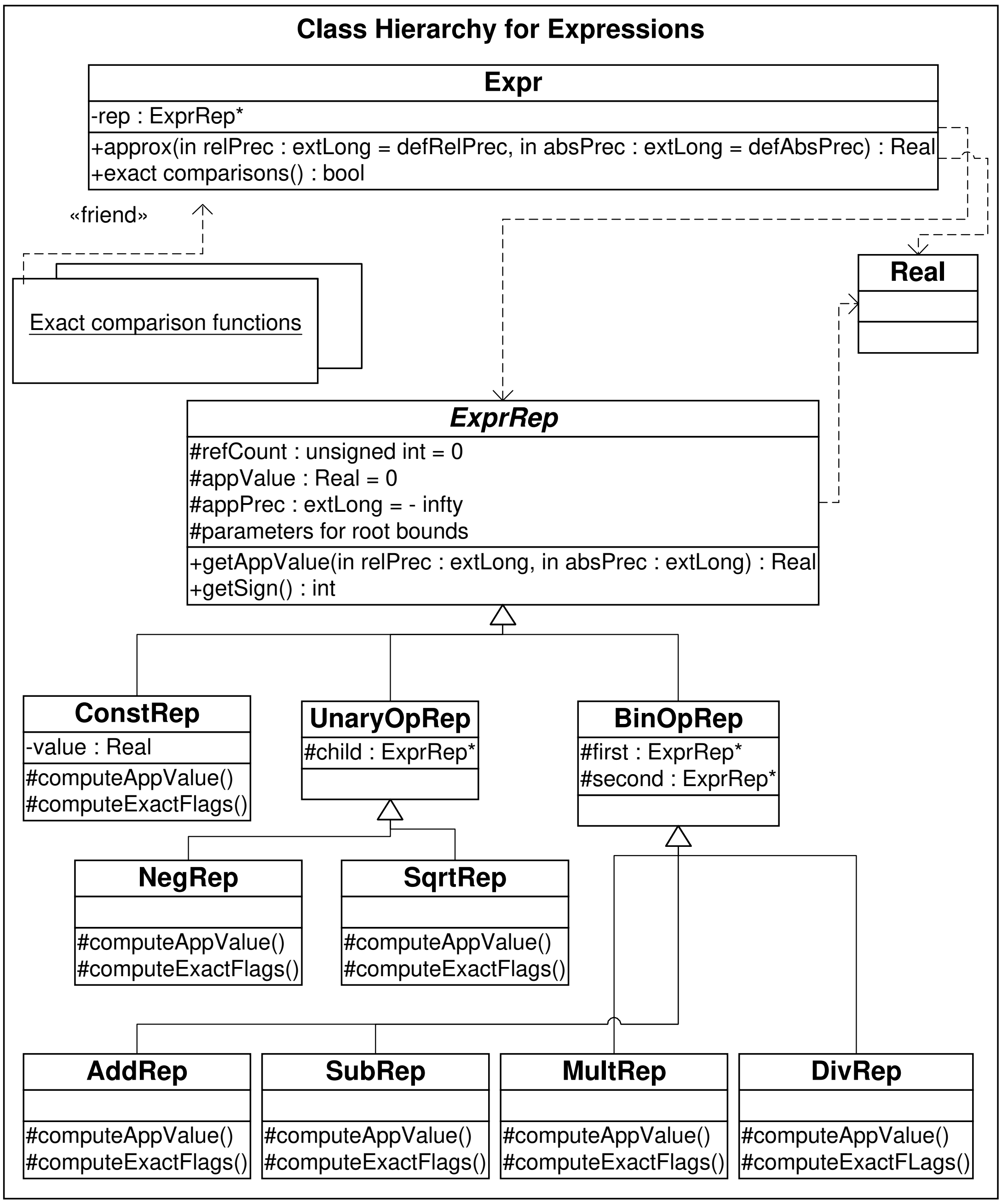
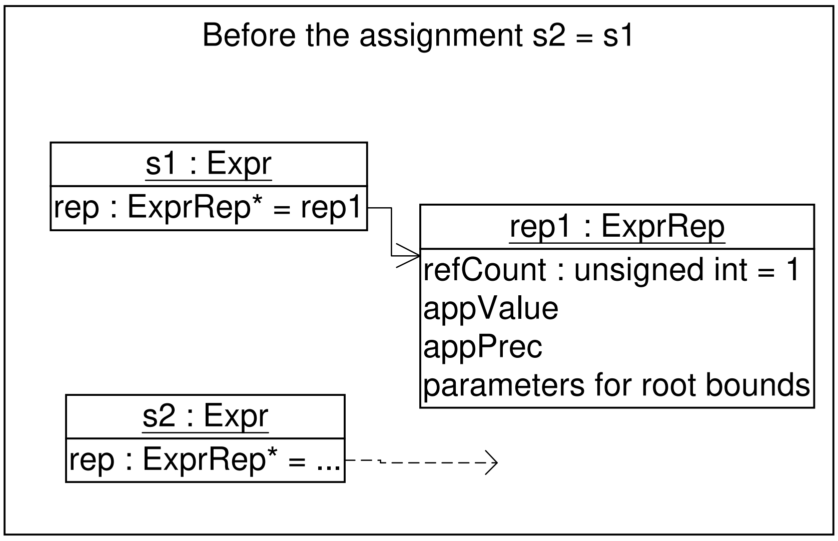
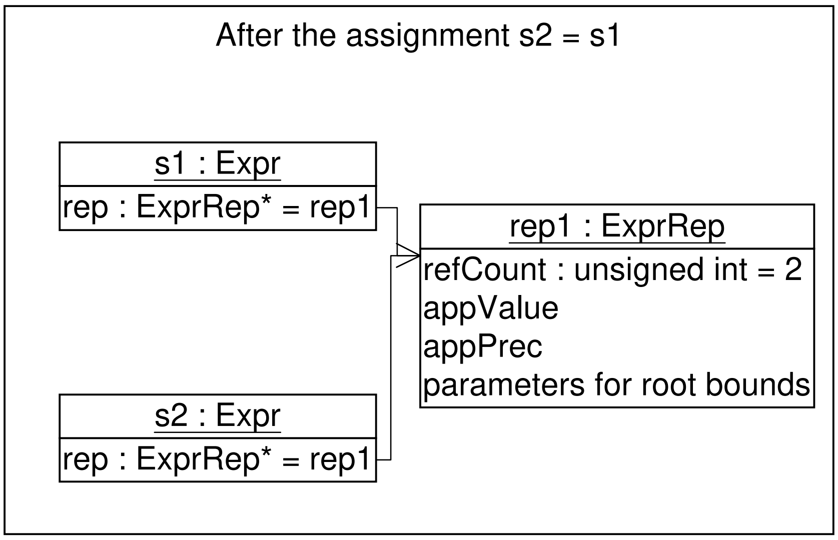


Figure 3.3: Reference counting.





Chapter 4

Randomized Zero Testing of Radical Expressions and Geometry Theorem Proving

Although focused on geometric computation, our CORE library can also be used in other areas where numerical computations require arbitrary absolute or relative precision. In this chapter, we discuss the application of our Core Library in testing the vanishing of multi-variate radical expressions, and in automated theorem proving [55].

We devise a novel probabilistic approach for the zero testing of multi-variate radical expressions with square roots (see Section 4.1). Our method is an extension of the well-known Schwartz's probabilistic test on the vanishing of polynomials. As in Schwartz's test, our method tests the vanishing of radical expressions on examples chosen from some finite test set with appropriate cardinality. Because of numerical errors, a challenge in practice is to guarantee the correctness of tests on these examples. Here we employ our Core Library, which is able to determine the sign of algebraic expressions exactly. As we have discussed, the cost of exact computation depends on the bit-length of inputs. Thus, for better efficiency, we want the size of test set to be as small as possible.

In [55], Yap proposed a novel method to bound this size based on the *preparation* of Straight Line Programs. The basic idea is to implicitly construct a polynomial which contains all the zeros of a radical expression and then bound the degree of this polynomial. After that, we can apply Schwartz's Lemma regarding the relation of error probability and the cardinality of the test set. In Section 4.1, we give a new simplified approach to derive this bound and this also leads more efficient computation of the bound.

Based on the zero test, we present a probabilistic approach in Section 4.2 to prove elementary geometry theorems about ruler-and-compass constructions by randomly chosen examples. A probabilistic theorem prover based on this approach was implemented by Tulone, using the Core Library. The prover can prove conjectures with an arbitrarily small *failure probability*, by testing the validity of a theorem on random examples. We report some experimental results at the end of this chapter. Most of the results of this chapter have been published in [55].

4.1 A Randomized Zero Test for Multivariate Radical Expressions with Square Roots

Suppose E is a multi-variate radical expression which is constructed from constants and m input variables $\mathbf{u} = \{u_1, u_2, \ldots, u_m\}$ through repeated application of four basic arithmetic operations $\{+, -, \times, \div\}$ and square root $\sqrt{\cdot}$. The expression E can be viewed as a DAG (directed acyclic graph). We allow the sharing of common subexpressions in the DAG. In this section, we present a probabilistic method to test the vanishing of E by extending the well-known Schwartz's test for polynomials. Below is a simple form of Schwartz's lemma:

Lemma 4.1 (Schwartz, 1980). *Suppose that $P(\mathbf{u})$ is a polynomial in the variables of $\mathbf{u} = \{u_1, u_2, \ldots, u_m\}$ with the degree $\deg(P)$ and that $P(\mathbf{u})$ is not identically zero. Let S be any finite set of elements in the domain or field F of the coefficients of P with the cardinality $|S|$. If each instance a_i $(i = 1, \ldots, m)$ is randomly chosen from S, then the probability that $P(a_1, \ldots, a_m) = 0$ is at most $\frac{\deg(P)}{|S|}$.*

Note that polynomials can be multivariate and in that case, the degree D is the total degree.

In order to apply the Schwartz Lemma in zero testing of radical expressions, our major task is that given a multivariate radical expression $E(\mathbf{u})$, find an appropriate upper bound of degree D such that there exists a non-zero polynomial P with the degree at most D and

$$\mathrm{zero}(E) \subseteq \mathrm{zero}(P).$$

The basic idea is to construct such a polynomial by keeping eliminating the outer-most square root operations in E, step by step. Note this construction is implicit and we do not compute this polynomial explicitly. Instead, we give a method to bound its degree, because it is the degree that determines the cardinality of test set.

4.1.1 Straight Line Program and Rational Degree

In the following analysis, we will use the Straight Line Program (SLP) to represent radical expressions, in order to take advantage of the sharing of common subexpressions to derive a tight degree bound.

Straight Line Program A SLP π is defined as a sequence of steps where each step is an assignment to a new programming variable. In our particular settings, the i-th step has one of the forms

$$z_i \leftarrow x_i \cdot y_i, \qquad (\cdot \in \{+, -, \times, \div\}), \text{or} \tag{4.1}$$

$$z_i \leftarrow \sqrt{x_i} \tag{4.2}$$

where z_i is the newly introduced programming variable at this step, x_i and y_i are either rational constants, *input variables* $u_i \in \mathbf{u} = \{u_1, u_2, \ldots, u_m\}$ for some m, or programming variables z_k ($1 \leq k \leq i$) introduced in previous steps. The variables x_i and y_i are said to be *used* in the i-th step. The last introduced variable is called the *main variable* of the SLP and is never used.

A SLP as defined above can be mapped to an expression DAG E in the natural way: the constants and input variables are leaves, and the programming variables introduced are the intermediate nodes labeled by arithmetic operators. For example, the variable z_i which is shown above is mapped to a node which has outgoing edges pointing to the operand node(s) (x_i and y_i in 4.1 and x_i only in 4.2) appearing in the right side of the assignment statement. If (u, v) is an edge, we say u is the predecessor of v and v is the successor of u. The only node with the in-degree 0 is called the *root* which is corresponding to the main variable. Those nodes having no out-edges are called *leaves* which are corresponding to the constants and input variables in a SLP. For a node p in the DAG, its induced DAG (i.e., the sub-DAG formed by all the nodes reachable from p and the edges induced by them) represents a sub-expression. Given a SLP π, let the set $Used$ be the set of all the variables used in π. Similarly we can define the induced SLP π_v of a programming variable v in a SLP π as the *subsequence* of π formed by all the steps contained in the least fixed point (LFP) S as follows:

$$S := \{s | LHS(s) = v, s \in \pi\} \cup \{s | LHS(p) \in Used(S), s \in \pi, p \in \pi\},$$

where the operator LHS takes the left-hand variable (the target) of a step. Sub-expressions can be shared. So there may be multiple paths between the root and a node below it. For a radical node r, we define its *radical depth* as the number of square root nodes in the path from r (inclusive) to the root. If there are more than one paths, by definition, we choose the one which has more square roots on it than any other such paths.

Let E be a radical expression in the variables $\{u_1, u_2, \ldots, u_m\}$. We denote $\mathbb{Q}_0$ as the extension $\mathbb{Q}(u_1, u_2, \ldots, u_m)$. Suppose there are r square roots in E, listed in the partial order of the dependency relation as follows:

$$\mathbf{RAD} = \{\sqrt{x_{l_1}}, \sqrt{x_{l_2}}, \ldots, \sqrt{x_{l_r}}\},$$

where x_{l_k} are intermediate subexpressions. Clearly there are at most k radical nodes in the induced DAG of x_{l_k}, and these radical nodes are elements in $\{\sqrt{x_{l_1}}, \sqrt{x_{l_2}}, \ldots, \sqrt{x_{l_k}}\}$. We define a tower of r extensions by adjunction of square roots starting from $\mathbb{Q}_0$ as follows:

$$\mathbb{Q}_0 \subseteq \mathbb{Q}_1 \subseteq \mathbb{Q}_2 \subseteq \cdots \subseteq \mathbb{Q}_r,$$

where $\mathbb{Q}_i = \mathbb{Q}_{i-1}(\sqrt{x_{l_i}})$, $x_{l_i} \in \mathbb{Q}_{i-1}$ and $i = 1, 2, \ldots, r$. For each variable intermediate z in the SLP $\pi(E)$ for E, we can define its value to be an appropriate element in certain extension $\mathbb{Q}_k$ over $\mathbb{Q}_0$ where k is the number of distinct square roots in the induced SLP of z. Note here we use the concepts variables and nodes interchangeably. This is justified by the fact that there is an injective mapping from a SLP to a DAG. Especially, the *value* of E is in the extension field $\mathbb{Q}_r$.

Table 4.1: Inductive definition of rational degrees

z	udeg(z)	ldeg(z)
constant	0	0
parameter	1	0
$x \times y$	udeg$(x) +$ udeg(y)	ldeg$(x) +$ ldeg(y)
$x \div y$	udeg$(x) +$ ldeg(y)	ldeg$(x) +$ udeg(y)
$x \pm y$	$\max\{\text{udeg}(x) + \text{ldeg}(y), \text{ldeg}(x) + \text{udeg}(y)\}$	ldeg$(x) +$ ldeg(y)
$\sqrt{x}$	$\frac{\text{udeg}(x)}{2}$	$\frac{\text{ldeg}(x)}{2}$

Rational degree Let x be a variable in a SLP π. We inductively define its "upper" and "lower" degrees, denoted as udeg(x) and ldeg(x) respectively, in the Table 4.1. We define the *rational degree* rdeg(x) of x as a pair of numbers and

$$\text{rdeg}(x) = \text{udeg}(x) : \text{ldeg}(x)\,.$$

The *rational degree* of the SLP π is defined to be udeg(z_π) : ldeg(z_π) where z_π is the main variable of π.

An important feature about rational degrees is that they are preserved in many equivalent transformations (with restrictions) of expression SLPs and give us a useful tool in bounding the final degree of the polynomial we implicitly constructed.

In [55] we gave an alternative definition rational degrees where all the values are natural numbers. The alternative definition is given in Table 4.2 (assume rdeg$_2(x) = a_x : b_x$ and rdeg$_2(y) = a_y : b_y$). This alternative definition of rational degrees need some auxiliary notions: for any node or variable x, let $RAD(x)$ denote the set of radical nodes in the sub-DAG of π rooted at x. Write $RAD(x, y)$ for $RAD(x) \setminus RAD(y)$ (set

Table 4.2: An alternative definition of rational degrees

z	$\text{udeg}_2(z)$	$\text{ldeg}_2(z)$
constant	0	0
parameter	1	0
$x \times y$	$a_x 2^{\rho(y,x)} + a_y 2^{\rho(x,y)}$	$b_x 2^{\rho(y,x)} + b_y 2^{\rho(x,y)}$
$x \div y$	$a_x 2^{\rho(y,x)} + b_y 2^{\rho(x,y)}$	$b_x 2^{\rho(y,x)} + a_y 2^{\rho(x,y)}$
$x \pm y$	$\max\{(a_x 2^{\rho(y,x)} + b_y 2^{\rho(x,y)}, b_x 2^{\rho(y,x)} + a_y 2^{\rho(x,y)})\}$	$b_x 2^{\rho(y,x)} + b_y 2^{\rho(x,y)}$
$\sqrt{x}$	a_x	b_x

difference). Also let $\rho(x) := |RAD(x)|$ and $\rho(x, y) := |RAD(x, y)|$.

The following lemma establishes the connection between the two definitions of rational degree.

Lemma 4.2. *For any variable z in a SLP, we have*

$$\text{udeg}_2(z) = 2^{\rho(z)}\,\text{udeg}(z), \qquad \text{ldeg}_2(z) = 2^{\rho(z)}\,\text{ldeg}(z),$$

where $\rho(z)$ is the number of square roots in the induced SLP of z in π.

Proof. We use induction on the structure of the sub-DAG rooted at z. If z is a constant, then $\text{udeg}_2(z) = \text{ldeg}_2(z) = 0$. If z is a parameter, then $\text{udeg}_2(z) = 1$ and $\text{ldeg}_2(z) = 0$. The lemma is clearly true in these base cases.

If $z = x \pm y$ then

$$\begin{aligned}
\text{udeg}_2(z) &= \max\{2^{\rho(y,x)}\,\text{udeg}_2(x) + 2^{\rho(x,y)}\,\text{ldeg}_2(y), \\
&\qquad 2^{\rho(y,x)}\,\text{ldeg}_2(x) + 2^{\rho(x,y)}\,\text{udeg}_2(y)\} \\
&= \max\{2^{\rho(z)-\rho(x)}\,\text{udeg}_2(x) + 2^{\rho(z)-\rho(y)}\,\text{ldeg}_2(y), \\
&\qquad 2^{\rho(z)-\rho(x)}\,\text{ldeg}_2(x) + 2^{\rho(z)-\rho(y)}\,\text{udeg}_2(y)\} \\
&= 2^{\rho(z)}\max\{\text{udeg}(x) + \text{ldeg}(y), \text{ldeg}(x) + \text{udeg}(y)\} \\
&= 2^{\rho(z)}\,\text{udeg}(z), \\
\text{ldeg}_2(z) &= 2^{\rho(y,x)}\,\text{ldeg}_2(x) + 2^{\rho(x,y)}\,\text{ldeg}_2(y) \\
&= 2^{\rho(z)-\rho(x)}\,\text{ldeg}_2(x) + 2^{\rho(z)-\rho(y)}\,\text{ldeg}_2(y) \\
&= 2^{\rho(z)}(\text{ldeg}(x) + \text{ldeg}(y)) \\
&= 2^{\rho(z)}\,\text{ldeg}(z)\,.
\end{aligned}$$

If $z = x \times y$ then

$$\begin{aligned}
\text{udeg}_2(z) &= 2^{\rho(y,x)}\,\text{udeg}_2(x) + 2^{\rho(x,y)}\,\text{udeg}_2(y) \\
&= 2^{(\rho(z)-\rho(x))}\,\text{udeg}_2(x) + 2^{(\rho(z)-\rho(y))}\,\text{udeg}_2(y) \\
&= 2^{\rho(z)}(\text{udeg}(x) + \text{udeg}(y)) \\
&= 2^{\rho(z)}\,\text{udeg}(z), \\
\text{ldeg}_2(z) &= 2^{\rho(y,x)}\,\text{ldeg}_2(x) + 2^{\rho(x,y)}\,\text{ldeg}_2(y) \\
&= 2^{(\rho(z)-\rho(x))}\,\text{ldeg}_2(x) + 2^{(\rho(z)-\rho(y))}\,\text{ldeg}_2(y) \\
&= 2^{\rho(z)}(\text{ldeg}(x) + \text{ldeg}(y)) \\
&= 2^{\rho(z)}\,\text{ldeg}(z)\,.
\end{aligned}$$

If $z = x/y$, then we have

$$\begin{aligned}
\text{udeg}_2(z) &= 2^{\rho(y,x)}\,\text{udeg}_2(x) + 2^{\rho(x,y)}\,\text{ldeg}_2(y) \\
&= 2^{(\rho(z)-\rho(x))}\,\text{udeg}_2(x) + 2^{(\rho(z)-\rho(y))}\,\text{ldeg}_2(y) \\
&= 2^{\rho(z)}(\text{udeg}(x) + \text{ldeg}(y)) \\
&= 2^{\rho(z)}\,\text{udeg}(z), \\
\text{ldeg}_2(z) &= 2^{\rho(y,x)}\,\text{ldeg}_2(x) + 2^{\rho(x,y)}\,\text{udeg}_2(y) \\
&= 2^{(\rho(z)-\rho(x))}\,\text{ldeg}_2(x) + 2^{(\rho(z)-\rho(y))}\,\text{udeg}_2(y) \\
&= 2^{\rho(z)}(\text{ldeg}(x) + \text{udeg}(y)) \\
&= 2^{\rho(z)}\,\text{ldeg}(z)\,.
\end{aligned}$$

If $z = \sqrt{x}$ then $\rho(z) = \rho(x) + 1$ since a new square root is introduced in this step. And we have

$$\begin{aligned}
\text{udeg}_2(z) &= \text{udeg}_2(x) \\
&= 2^{\rho(z)-1-\rho(x)}\,\text{udeg}_2(x) \\
&= 2^{\rho(z)}\,\text{udeg}(z), \\
\text{ldeg}_2(z) &= \text{ldeg}_2(x) \\
&= 2^{\rho(z)-1-\rho(x)}\,\text{ldeg}_2(x) \\
&= 2^{\rho(z)}\,\text{ldeg}(z)\,.
\end{aligned}$$

Q.E.D.

In the above lemma, we have shown that the alternative definition in Table 4.2 can be computed from the original definition by multiplying a factor $2^{\rho(z)}$. In practice, computing $RAD(x, y)$ could be an expensive process. Thus, it is more efficient to compute rational degrees defined by Table 4.1. In the following derivation, we adopt the first definition because it often gives more succinct proofs.

4.1.2 Equivalent Transformations

In Straight Line Program, two variables x and y are said to be *equivalent*, denoted as $x \equiv y$, if they have the same value. Transformations of a SLP which preserve its value are called *equivalent transformation*. Note that some equivalent transformations may change rational degrees we have defined above. For instance, by the distributive law, we have the equivalent transform:

$$(x + y)z \equiv xz + yz. \tag{4.3}$$

It can be easily verified that the rational degree of the LHS (left-hand side) is at most that of RHS, but can be less. However, the next lemma shows the rational degree in this transformation preserved if z is free of divisions:

Lemma 4.3. *The induced SLP of the variable z in π is division-free, then the equivalent transformation 4.3 (in both directions) preserves rational degrees. In particular,*

$$\text{rdeg}(z(x + y)) = \text{rdeg}(zx + zy)\,.$$

Proof. If z is division-free, ldeg$(z) = 0$. Then we have

$$\begin{aligned}
\text{udeg}(z(x+y)) &= \text{udeg}(z) + \text{udeg}(x+y) \\
&= \text{udeg}(z) + \max\{\text{udeg}(x) + \text{ldeg}(y), \\
&\qquad\qquad \text{ldeg}(x) + \text{udeg}(y)\} \\
&= \max\{\text{udeg}(z) + \text{udeg}(x) + \text{ldeg}(z) + \text{ldeg}(y), \\
&\qquad \text{ldeg}(z) + \text{ldeg}(x) + \text{udeg}(z) + \text{udeg}(y)\} \\
&= \text{udeg}(zx + zy)
\end{aligned}$$

$$\begin{aligned}
\text{ldeg}(z(x+y)) &= \text{ldeg}(x+y) \\
&= \text{ldeg}(x) + \text{ldeg}(y) \\
&= \text{ldeg}(zx + zy)
\end{aligned}$$

Q.E.D.

Next, we show that equivalent transformations based on the associative law and commutative law of multiplication and addition do not change rational degrees.

Lemma 4.4.

$$\text{rdeg}(x \pm y) = \text{rdeg}(y \pm x)$$

$$\text{rdeg}(x \times y) = \text{rdeg}(y \times x)$$

$$\text{rdeg}((x \pm y) \pm z) = \text{rdeg}(x \pm (y \pm z))$$

$$\text{rdeg}((xy)z) = \text{rdeg}(x(yz))$$

Proof. The equalities can be easily verified by the first definition of rational degrees. We omit the details of proof here. **Q.E.D.**

In the next two lemmas, we generalize the commutative and associative transformations.

Lemma 4.5.

$$\text{rdeg}(\prod_{i=1}^{k} x_i) = \sum_{i=1}^{k} \text{rdeg}(x_i)$$

Proof. This lemma can be proved by induction on the number of operands. We omit the details here. **Q.E.D.**

Lemma 4.6.

$$\text{udeg}(\sum_{i=1}^{k} x_i) = \max_{i=1}^{k}\{\text{udeg}(x_i) + \sum_{j=1, j\neq i}^{k} \text{ldeg}(x_j)\}$$
$$\text{ldeg}(\sum_{i=1}^{k} x_i) = \sum_{i=1}^{k} \text{ldeg}(x_i)$$

Proof. This lemma can be proved by induction on the number of operands. We omit the details here. **Q.E.D.**

The above two lemmas allow us to introduce two well-defined nodes: "sum" or $\sum$-node, and "product" or $\prod$-node. They generalize the ordinary binary addition and multiplication operations by taking an arbitrary number of arguments. A SLP with such extensions is called a *generalized SLP*. A path in a generalized SLP DAG is said to be *alternating* if along the path, no two consecutive nodes are $\sum$-nodes and no two consecutive nodes are $\prod$-nodes. The SLP is *alternating* if every path is alternating. From Lemmas 4.5 and 4.6, we know that any SLP can be made alternating without changing its rational degree.

4.1.3 Preparation

A SLP π is said to be *prepared* (or in prepared form) if either there are no radical operations in it, or the last three steps of it have the form

$$\begin{aligned}
&\cdots \\
&x \leftarrow \sqrt{w_C} \\
&y \leftarrow x \times w_B \\
&z \leftarrow y + w_A.
\end{aligned}$$

Here w_A, w_B, w_C are variables or constants. Thus z is the main variable with the value $A + B\sqrt{C}$, where A, B and C are the values of w_A, w_B, w_C, respectively. Clearly x is the last radical variable to be introduced and its radical depth is 1. We call the variable x the *main prepared variable* of the SLP π. Intuitively, the main prepared variable has been brought up as close to the root of the SLP as possible so that this square root can be removed by the transformation we will introduce below. Similarly, we can extend this concept to arbitrary variable in a SLP and say a variable v in a SLP π is prepared if the induced SLP of v in π is prepared.

Now let us investigate how a SLP can be transformed into an equivalent prepared form.

Let A_i, B_i be expressions for $i = 1, \ldots, n$ and $n > 0$. We can define an expression E_n inductively:

$$E_n = \begin{cases} A_0 \times B_0 & \text{when } n = 0; \\ (E_{n-1} + A_n) \times B_n & \text{otherwise.} \end{cases}$$

To highlight the dependency relation, we also write E_n as

$$E_n(A_0, B_0, A_1, B_1, \ldots, A_n, B_n).$$

Viewed as a tree, the path between A_0 and the root E_n is essentially an alternating path (see the left tree in Figure 4.1). The following lemma shows that through equivalent transformation, we can get a new expression E'_n in which the innermost expression A_0 of E_n becomes prepared.

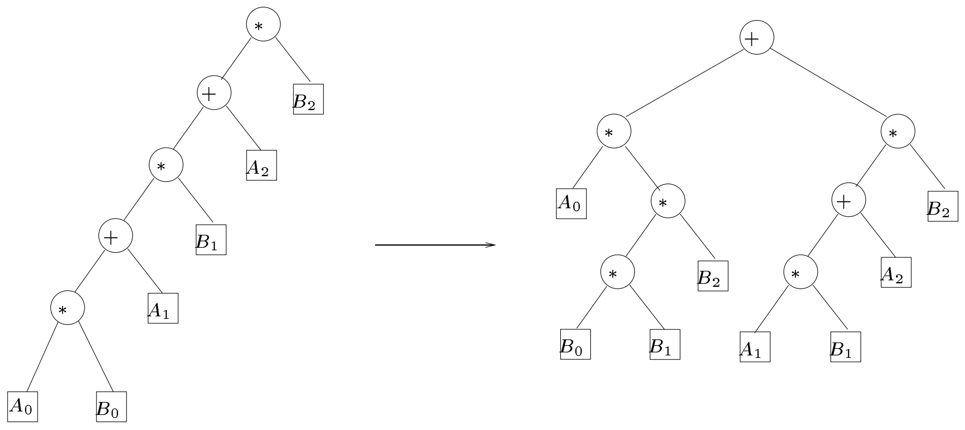


Figure 4.1: The transformation $E_2 \mapsto E'_2$ (from [55]).

Lemma 4.7. *For $n \geq 1$, the expression $E_n(A_0, B_0, \ldots, A_n, B_n)$ is equivalent to the expression*

$$E'_n := (A_0 \times B_{(n)}) + E_{n-1}(A_1, B_1, \ldots, A_n, B_n),$$

where $B_{(n)} := \prod_{j=0}^{n} B_j$. Moreover, if all the subexpressions B_is are division-free, then rdeg$(E_n) =$ rdeg(E'_n).

Proof.

Proof by induction,

When $n = 1$,

$$\begin{aligned} E_n &= (A_0 \times B_0) + A_1) \times B_1 \\ &= (A_0 \times B_0 \times B_1) + A_1 \times B_1 \end{aligned}$$

Assume that this lemma is held for $n \leq k$, then for $n = k+1$,

$$\begin{aligned} E_{k+1} &= (E_k + A_{k+1}) \times B_{k+1} \\ &= ((A_0 \times B_{(k)}) + E_{k-1}(A_1, B_1, \ldots, A_k, B_k) + A_{k+1}) \times B_{k+1} \\ &= (A_0 \times B_{(k+1)}) + E_k(A_1, B_1, \ldots, A_{k+1}, B_{k+1}). \end{aligned}$$

Thus we know the equivalence of this transformation is held for any $n \in \mathbb{N}$.

In both cases, we only apply the distributive and associative laws. Thus from Lemma 4.3 and Lemma 4.4, if all B_i's are division-free then $\text{rdeg}(E_n) = \text{rdeg}(E'_n)$.

Q.E.D.

Especially if A_0 is a radical node with a radical depth 1, the the above transformation effectively prepares the SLP with A_0 being the main prepared variable.

Note that the above lemma can be extended to generalized SLPs too, in which A_i's and B_i's need not be distinct, and the addition and multiplication operators can be replaced by multi-nary $\sum$- and $\prod$- operators.

Theorem 4.8. *Suppose π is a division-free SLP and u is a radical node in π with radical depth of 1. Then we can transform π into an equivalent SLP π' which is prepared and satisfies $\text{udeg}(\pi) = \text{udeg}(\pi')$. Moreover, the prepared variable u' of π' is the unique variable in π' with value $val_\pi(u)$.*

Proof. Without loss of generality, we can assume that π is a generalized, alternating SLP. Let us consider the corresponding DAG representation of π. Fix any path p from u to the root and we may assume that this alternative sum-product path has the same form as the path from A_0 to the root of E_n. We then apply Lemma 4.7 in which u now plays the role of the node A_0 in E_n. This collapses the path p to length 2. The resulting

DAG has the form $E' = u \times A + B$. We can repeat this process for the subexpressions A and/or B, if they contain references to the node u as well. There are two cases:

1. u is used in A, then A is transformed to $A' = u \times A_1 + B_1$ and $E' = u \times B_1 + (A_1 u^2 + B)$. Remember that u is a square root and thus the expression u^2 effectively eliminates the square root operation here;

2. u is used in B, then B is transformed to $B' = u \times A_2 + B_2$ and $E' = u \times (A + A_2) + B_2$.

In both cases, we can see that E' is still in a prepared form. We keep this process until there is no use of u except the one that is in the prepared position and has a unique path to the root with length 2. Since there must be a finite number of uses of u, this iterative process will eventually terminate. At that point, the resulting SLP π' has the desired form: π' is prepared and u is the main prepared variable. It is also clear that if there are other nodes with the same value as u, they can also be merged with u by the same process. Hence, u will be the unique node with value $val_\pi(u)$.

Note that we apply the commutative, associative and distributive laws in these transformations. The commutative and associative transformations do not change the rational degree. Since π is division free, Lemma 4.3 tells us that the distributive transformation preserves the rational degree too. Therefore, the preparation transformation does not change the rational degree of π.

Q.E.D.

4.1.4 Probabilistic Zero Test

Let π be a SLP whose value is $V = V(\mathbf{u}) \in \mathbb{Q}_r$. We define an associated real function

$$f_V : \mathbb{R}^m \to \mathbb{R}$$

where $f_V(a_1, \ldots, a_m)$ is the value that is obtained when we evaluate the function V at $(a_1, \ldots, a_m) \in \mathbb{R}^m$. We denote the the *zero set* of V by

$$\mathrm{zero}(V) := f_V^{-1}(0).$$

Theorem 4.9. *Suppose $V = V(\mathbf{u})$ is the non-zero value of a rooted division-free SLP π. Then there exists a non-zero polynomial $P(\mathbf{u})$ such that* $\mathrm{zero}(V) \subseteq \mathrm{zero}(P)$ *with* $\deg(P(\mathbf{u})) \leq 2^r\, \mathrm{udeg}(\pi)$*, where r is the number of distinct square roots in π.*

Proof.

Proof by induction on the number of square roots in π.

For $r = 0$, V is already a polynomial. By examining the definition of rational degrees (Table 4.1), it can be easily verified that the (total) degree of the multivariate polynomial V is rdeg(π).

Suppose this theorem is true for all $r < k$ ($k > 0$), next we will prove it is also true when $r = k$. Let $u = \sqrt{C}$ be a radical node in π with radical depth 1. We prepare the SLP π with u being the prepared variable and obtain an equivalent SLP π' which, by the definition of equivalent transformation, has the same zero set. And moreover, by Theorem 4.8, this transformation preserves the rational degree. Thus, we can write the value V in the form $V = A + B\sqrt{C}$, where A, B and C belong to the extension $\mathbb{Q}_{r-1}$.

If $B = 0$, then $V = A \in \mathbb{Q}_{r-1}$. If $A - B\sqrt{C} = 0$, then $V = 2 \cdot A \in \mathbb{Q}_{r-1}$. In both cases, $V \in \mathbb{Q}_{r-1}$ (i.e., having at most $(r-1)$ square roots). Thus the theorem is true by the induction assumption.

Now let us consider the more interesting case where $B \neq 0$ and $A - B\sqrt{C} \neq 0$. We transform the SLP π' to π^* whose value is

$$V' = (A + B\sqrt{C})(A - B\sqrt{C}) = A^2 - B^2C. \tag{4.4}$$

Note this is not necessarily an equivalent transformation, but we have $\mathrm{zero}(V) \subseteq \mathrm{zero}(V')$. By definition we know $V = A + B\sqrt{C} \neq 0$ and by assumption we have $A - B\sqrt{C} \neq 0$, then $V' \neq 0$. Clearly the SLP π^* has at most $(r-1)$ square roots since the square root $\sqrt{C}$ has been eliminated and no new square roots are introduced in the transformation. From the induction assumption, there must exist a non-zero polynomial P with degree $\deg(P) \leq 2^{r-1}\,\text{udeg}(\pi^*)$ such that $\mathrm{zero}(V) \subseteq \mathrm{zero}(V') \subseteq \mathrm{zero}(P)$. It remains to show that $\text{udeg}(\pi^*) \leq 2\,\text{udeg}(\pi)$:

$$\begin{aligned}
\text{udeg}(\pi^*) &= \max\{2\,\text{udeg}(A) + 2\,\text{ldeg}(B) + \text{ldeg}(C), \\
&\qquad 2\,\text{ldeg}(A) + 2\,\text{udeg}(B) + \text{udeg}(C)\} \\
&= 2\max\{\text{udeg}(A) + \text{ldeg}(B) + \frac{1}{2}\,\text{ldeg}(C), \\
&\qquad \text{ldeg}(A) + \text{udeg}(B) + \frac{1}{2}\,\text{udeg}(C)\} \\
&= 2\,\text{udeg}(\pi)
\end{aligned}$$

Therefore, the polynomial P is the polynomial we want to find and its degree is at most $2^r\,\text{udeg}(\pi)$.

Q.E.D.

With the degree bound given by Theorem 4.9, now we can extend the Schwartz lemma (see Lemma 4.1) to test the vanishing of radical expressions.

Theorem 4.10. *Let π be a non-zero, division-free SLP with r square roots. $V = V(\mathbf{u})$ is the value of π. If each instance a_i ($i = 1, \ldots, m$) is randomly chosen from a finite set $S \subset \mathbb{R}$, then the probability that $V(a_1, a_2, \ldots, a_i) = 0$ is at most $\frac{2^r\,\text{udeg}(\pi)}{|S|}$.*

Proof. By Theorem 4.9, there exists a non-zero polynomial $P(\mathbf{u})$ such that $\mathrm{zero}(V) \subseteq \mathrm{zero}(P)$ and the degree $\deg(P) \leq 2^r \operatorname{udeg}(\pi)$. Thus

$$\mathrm{Prob}\{V(a) = 0\} \leq \mathrm{Prob}\{P(a) = 0\} \leq 2^r \operatorname{udeg}(\pi) / |S|.$$

The last inequality relation is from a direct application of the Schwartz Lemma.

Q.E.D.

Presence of Divisions Theorem 4.10 presents a probabilistic approach to test the vanishing of division-free radical expressions. What if the SLP is not division-free? In this case, there is a well known transformation to move all divisions towards the root and merge them as we go. An instance of this transformation is

$$\frac{A}{B} + \frac{A'}{B'} \Rightarrow \frac{AB' + A'B}{BB'}.$$

We should note that during this transformation, the number of square roots might be doubled because if we move a division operator past a radical node, we split it into two radical nodes:

$$\sqrt{\frac{A}{B}} \Rightarrow \frac{\sqrt{A}}{\sqrt{B}}.$$

Hence we give two versions of this transformation in the following lemma: in version (i) we do not move any division node past a radical node, and in version (ii) we remove all but at most one division node.

Lemma 4.11. *Suppose a, b, c, d are variables in a SLP π and their induced SLPs have no divisions, then the rational degrees are preserved in the following transformations:*

$$\begin{aligned}
\text{rdeg}(\frac{a}{b} \times \frac{c}{d}) &= \text{rdeg}(\frac{ac}{bd}) \\
\text{rdeg}(\frac{a}{b} \div \frac{c}{d}) &= \text{rdeg}(\frac{ad}{bc}) \\
\text{rdeg}(\frac{a}{b} \pm \frac{c}{d}) &= \text{rdeg}(\frac{ad \pm bc}{bd})
\end{aligned}$$

Lemma 4.12. *Suppose x, y and z are variables in a SLP π and $z = \sqrt{x/y}$, we can extract the division operator through an equivalent transformation $z \mapsto z' = \sqrt{|x|}/\sqrt{|y|}$ without changing the rational degree, i.e.,* $\text{rdeg}(z) = \text{rdeg}(z')$.

Proof.

$$\begin{aligned}
\text{udeg}(z) &= \frac{1}{2}(\text{udeg}(x) + \text{ldeg}(y)) \\
&= \frac{1}{2}\,\text{udeg}(x) + \frac{1}{2}\,\text{ldeg}(y) \\
&= \text{udeg}(z')
\end{aligned}$$

$$\begin{aligned}
\text{ldeg}(z) &= \frac{1}{2}(\text{ldeg}(x) + \text{udeg}(y)) \\
&= \frac{\text{ldeg}(x)}{2} + \frac{\text{udeg}(y)}{2} \\
&= \text{ldeg}(z')
\end{aligned}$$

Q.E.D.

By Lemma 4.11 and Lemma 4.12, we know that we can transform a radical expression into one without divisions except in the root. This transformation does not change the rational degree of the original SLP. But note that the number of square roots can be potentially doubled as a result of this transformation.

Lemma 4.13 (Elimination of Divisions). *Let π be a rooted SLP with r square roots.*
(i) There is an equivalent SLP π' in which each division node is either the root of π or the child of a radical node. Moreover, $\mathrm{rdeg}(\pi') = \mathrm{rdeg}(\pi)$ *and π' has the same number of radical nodes as π.*
(ii) There is an equivalent SLP π'' with only one division node which is also the root. In this case $\mathrm{rdeg}(\pi'') = \mathrm{rdeg}(\pi)$. *But the number of square roots in π'' can be at most $2r$.*

In the next lemma, we prove the existence of a non-zero polynomial which vanishes as all the zeros of a given radical expression, and give a bound on its degree too.

Lemma 4.14. *Suppose a SLP π represents a radical expression with divisions, then there exists a non-zero polynomial $P(u)$ such that* $\mathrm{zero}(\pi) \subseteq \mathrm{zero}(P)$ *with* $\deg(P(u)) \leq 2^{2r}\,\mathrm{udeg}(\pi)$ *where r is the number of square roots in π.*

Proof.

First, we transform the SLP π to a SLP π' which does not have any division operators except in the root. Then we examine the numerator part of the π' which is division-free. All the zeros of π must be zeros of this numerator as well. There are at most $2r$ square roots in this part. From Theorem 4.10, there exists a polynomial P with $\deg(P(u)) \leq 2^{2r}\,\mathrm{udeg}(\pi)$. This P is the polynomial we want. It has all the zeros of π.

Q.E.D.

From the proof of Lemma 4.14, we can see that the degree of P has an extra factor of 2^r due to the splitting of square root nodes. Inspired by a recent work in [39], we can apply two other equivalent transformations on the square root nodes. They are

$$\sqrt{\frac{x}{y}} \quad \mapsto \quad \frac{\sqrt{xy}}{y}, \tag{4.5}$$

$$\sqrt{\frac{x}{y}} \quad \mapsto \quad \frac{x}{\sqrt{xy}}. \tag{4.6}$$

Unlike the case in Lemma 4.12, the rational degrees may be increased as a result of these transformations. If x and y are both division-free, the following lemma bounds the potential increase.

Lemma 4.15. *Let* $z = \sqrt{\frac{x}{y}}$ *be an expression SLP where* x *and* y *are division free. An equivalent transformation is defined as follows:*

$$z' = \begin{cases} \frac{\sqrt{xy}}{y} & \textit{if} \ \ \mathrm{udeg}(x) \geq \mathrm{udeg}(y) \\ \frac{x}{\sqrt{xy}} & \textit{if} \ \ \mathrm{udeg}(x) < \mathrm{udeg}(y)\,. \end{cases} \tag{4.7}$$

Then we have $\mathrm{udeg}(z') \leq 2\,\mathrm{udeg}(z)$ *and* $\mathrm{ldeg}(z') \leq 2\,\mathrm{ldeg}(z)$.

Proof.

Since x and y are division free, we know that $\mathrm{ldeg}(x) = \mathrm{ldeg}(y) = 0$.

When $\mathrm{udeg}(x) \geq \mathrm{udeg}(y)$,

$$\begin{aligned} \mathrm{udeg}(z') &= \frac{1}{2}(\mathrm{udeg}(x) + \mathrm{udeg}(y)) + \mathrm{ldeg}(y) \\ &\leq \mathrm{udeg}(x) + \mathrm{ldeg}(y) \\ &= 2\,\mathrm{udeg}(z) \end{aligned}$$

$$\begin{aligned} \mathrm{ldeg}(z') &= \frac{1}{2}(\mathrm{ldeg}(x) + \mathrm{ldeg}(y)) + \mathrm{udeg}(y) \\ &\leq 2\,\mathrm{ldeg}(z)\,. \end{aligned}$$

When udeg$(x) <$ udeg(y),

$$\begin{aligned}\text{udeg}(z') &= \text{udeg}(x) + \frac{1}{2}(\text{ldeg}(x) + \text{ldeg}(y)) \\ &\leq 2\,\text{udeg}(z)\end{aligned}$$

$$\begin{aligned}\text{ldeg}(z') &= \text{ldeg}(x) + \frac{1}{2}(\text{udeg}(x) + \text{udeg}(y)) \\ &< 2\,\text{ldeg}(z)\end{aligned}$$

Thus, we can see the the rational degrees can be at most doubled after the equivalent transformation (4.7).

Q.E.D.

By applying the equivalent transformations discussed in Lemma 4.11 and Lemma 4.15, we can transform an expression z to a division of two expressions in a bottom-up traversal of the DAG:

$$z \mapsto \frac{U(z)}{L(z)},$$

where the expressions $U(z)$ and $L(z)$ are division free. Moreover, the number of square roots does no increase in the transformation. We do not have to constructed the $U(z)$ and $L(z)$ explicitly at each node. Instead, we only need to compute the udeg$(U(z))$ and udeg$(L(z))$. The rules to compute these two degrees are listed in Table 4.3. For simplicity, we denote $u_z =$ udeg$(U(z))$ and $l_z =$ udeg$(L(z))$. Note that ldeg$(U(E)) =$ ldeg$(L(E)) = 0$ because these two expressions are division free. The next lemma gives us an alternative to Lemma 4.14.

Lemma 4.16. *Suppose a SLP π represents a non-zero radical expression E_π with divisions. Let u_π be the rational degree computed by Table 4.3. Then there exists a non-zero*

Table 4.3: Rules for rational degrees of transformed SLP

z	u_z	l_z
constant	0	0
parameter	1	0
$x \times y$	$u_x + u_y$	$l_x + l_y$
$x \div y$	$u_x + l_y$	$l_x + u_y$
$x \pm y$	$\max\{u_x + l_y, l_x + u_y\}$	$l_x + l_y$
$\sqrt{x}$	$\frac{1}{2}(u_x + l_x), \quad (u_x \geq l_x);$ $u_x, \quad (u_x < l_x).$	$l_x, \quad (u_x \geq l_x);$ $\frac{1}{2}(u_x + l_x), \quad (u_x < l_x).$

polynomial $P(u)$ *such that* $\mathrm{zero}(\pi) \subseteq \mathrm{zero}(P)$ *with* $\deg(P(u)) \leq 2^r u_\pi$ *where* r *is the number of square roots in* π.

Proof.

The equivalent transformations in Lemma 4.11 and Lemma 4.15 can be applied to transform E_π into a form of $\frac{U(E_\pi)}{L(E_\pi)}$, where $U(E_\pi)$ and $L(E_\pi)$ are division free. It can be easily verified that the $\mathrm{udeg}(U(E_\pi))$ and $\mathrm{udeg}(L(E_\pi))$ are computed by Table 4.3 and in particular $\mathrm{udeg}(U(E_\pi)) = u_\pi$. Clearly, all the zeros of π are the zeros of $U(E_\pi)$ as well. Note that in the equivalent transformation, we do not introduce any more square roots. Thus the expression $U(E_\pi)$ has at most r square roots. From Theorem 4.10, there exists a polynomial P with $\deg(P(u)) \leq 2^r u_\pi$ such that

$$\mathrm{zero}(P) \subseteq \mathrm{zero}(U(E_\pi)) \subseteq \mathrm{zero}(\pi).$$

Q.E.D.

Next, we will show that the degree bound given in Lemma 4.16 is never worse than that given by Lemma 4.14. In fact, in most cases, it is much sharper.

Lemma 4.17. *Suppose a SLP π represents a radical expression E_π with divisions, let u_π be the rational degree computed by Table 4.3, and* $\mathrm{udeg}(\pi)$ *be the rational degree of the original SLP π. Then*

$$2^r u_\pi \le 2^{2r}\,\mathrm{udeg}(\pi), 2^r l_\pi \le 2^{2r}\,\mathrm{ldeg}(\pi),$$

where r is the number of square roots in π.

Proof.

Proof by induction on the depth of the expression DAG corresponding to the SLP π. It is trivial to verify the this lemma is true for constant and parameter leaves.

Given an node z in the DAG, by the induction hypothesis, we know that the lemma is true for all its children.

If z is an addition node and $z = x+y$, then by definition $u_z = \max\{u_x+l_y, l_x+u_y\}$. Suppose there are r_x (and r_y) square roots in the subexpression x (and y, respectively). Clearly, $r_x, r_y \le r_z$ where r_z is the number of square roots in z. From the induction assumption,

$$2^{r_z} u_x \le 2^{r_z+r_x}\,\mathrm{udeg}(x) \le 2^{2r_z}\,\mathrm{udeg}(x)\,.$$

The same relation can be verified for l_x, u_y and l_y as well. Hence,

$$\begin{aligned} 2^{r_z} u_z &\le 2^{2r_z} \max\{\mathrm{udeg}(x)+\mathrm{ldeg}(y), \mathrm{ldeg}(x)+\mathrm{udeg}(y)\} \\ &= 2^{2r}\,\mathrm{udeg}(z) \end{aligned}$$

$$\begin{aligned} 2^{r_z} l_z &\le 2^{2r_z}(\mathrm{ldeg}(x)+\mathrm{ldeg}(y)) \\ &= 2^{2r_z}\,\mathrm{ldeg}(z) \end{aligned}$$

We omit the proofs for subtraction, multiplication and division operations, because they are similar to the proof for the addition case.

If $z = \sqrt{x}$ is a square root node, by Lemma 4.15, we know that

$$\begin{aligned} u_z &\leq u_x \\ &\leq 2^{r_x}\,\text{udeg}(x) \\ &= 2^{r_z}\,\text{udeg}(z) \end{aligned}$$

$$\begin{aligned} l_z &\leq l_x \\ &\leq 2^{r_x}\,\text{ldeg}(x) \\ &= 2^{r_z}\,\text{ldeg}(z) \end{aligned}$$

Therefore, from the induction argument, we have $2^r u_\pi \leq 2^{2r}\,\text{udeg}(\pi)$ and $2^r l_\pi \leq 2^{2r}\,\text{ldeg}(\pi)$.

Q.E.D.

Now we have obtained two degree bounds for radical expressions with division. By applying the Schwartz Lemma, we extend the probabilistic zero test to these class of expressions as follows:

Theorem 4.18. *Let π be a non-zero SLP with r square roots. Let $V = V(\mathbf{u})$ be the value of π. If each instance a_i ($i = 1, \ldots, m$) is randomly chosen from a finite set $S \subset \mathbb{R}$, then the probability that $V(a_1, a_2, \ldots, a_i) = 0$ is at most $\frac{2^r u_\pi}{|S|}$, where u_π is the degree inductively computed in Table 4.3.*

REMARK: The zero-test approach discussed above assumes the radical expressions concerned are well defined (i.e., no exceptions such as divisions by zeros, etc).

In practice, we choose the test set S to be a set of integers. Thus, to achieve the failure probability 2^{-c} for some natural number c, the maximum possible bit length of each parameter instance is $(r + \lg(u_\pi) + c)$, while in division-free cases, $(r + \lg(\text{udeg}(\pi)) + c)$ is enough. When exact arithmetic is deployed, the bit-length of inputs usually determines the running cost.

4.2 Probabilistic Proving of Elementary Geometry Theorems

4.2.1 Background

Proving geometry theorems mechanically has attracted a great deal of attention since the 1960s. Various methods following a number of directions have been pursued. The earlier work based on mathematical logic achieved only limited success in terms of effectiveness of provers and the range of theorems they can prove.

It has been well known that geometry theorem proving can be considered as an application of Taski's decision procedure for real closed fields. Collins [11] gave a more efficient decision procedure for Taski geometry. But algebraic methods based on these work were not so successful due to the high double-exponential complexity.

A major achievement to date in automated theorem proving is due to Wu Wen-tsün [56], who pioneered in proving geometry theorems using constructive methods in computer algebra. For geometry problems involving incidence, congruence and parallelism relations (but no ordering relation, e.g. a point is between the other two, etc.), Wu's method based on characteristic sets is practically efficient and has drastically enhanced the scope of what is computationally tractable in automated theorem proving. Chou's implementation of Wu's method and its variations was able to efficiently generate fully automatic proofs of hundreds of theorems of Euclidean geometry, including theorems that humans find difficult to prove such as Simson's theorem. Another important method is based on the Gröbner basis, developed by Kapur [28], Chou and Schelter [8], and Kutzler and Stifter [32]. Also, J. W. Hong introduced the "proof-by-example" technique based on a gap theorem.

Closer to the approach we will discuss here is the work by Carrá et al [6]. They applied the Schwartz's probabilistic test to check the vanishing of pseudo-remainders in Wu's method. The cardinality of the test set S depends on the degree estimate D of the final pseudo-remainder. To prove a generic truth (as we noted, most theorems in ordinary geometry are in this class), they show $D = 2^{O(C^3)}C^{O(C^2)}$ where C is the number of constructions. This bound on D is so high that they admit that the actual computational cost, when using exact arithmetic, could make this method practically infeasible.

In this section, we present a probabilistic approach to prove elementary geometry theorems by random examples.

4.2.2 Algebraic Formulation

The first step in proving geometry theorems algebraically is to transform the statement of theorem into the form of a polynomial system, under some proper coordinate system. Given a conjecture whose hypotheses can be expressed as a set of polynomial equations H and whose thesis can be represented as a polynomial equation T, the conjecture is true if the

$$\mathrm{Var}(H) \subseteq \mathrm{Var}(T),$$

where the operator Var means the algebraic variety defined by one or more polynomials.

Our work is focused on proving elementary geometry theorems with ruler-and-compass constructions only. After fixing some appropriate coordinate system, the algebraic analogue of a geometric construction of an object O amounts to introduce a pair of variables (x, y) and the corresponding polynomial equations $h_i(x, y, \ldots)$ which must be satisfied if (x, y) lies on O. Here "$\ldots$" refers to the variables that have already been introduced, if any. We classify all the variables into two categories: *independent variables* (or *parameters*), and *dependent variables*. Intuitively, independent variables refer to "free" parameters in a statement, while dependent variables are constrained by some polynomial conditions imposed by geometric constructions. Let us use the Simson's theorem as an example (from [10]) in illustrating this:

Example 4.1. (Simson's theorem) Let D be a point on the circumscribed circle O of triangle ABC. From D three perpendiculars are drawn to the three sides BC, CA and AB of $\triangle ABC$. Let E, F and G be the three feet respectively. Show that E, F and G are collinear (see Figure 4.2).

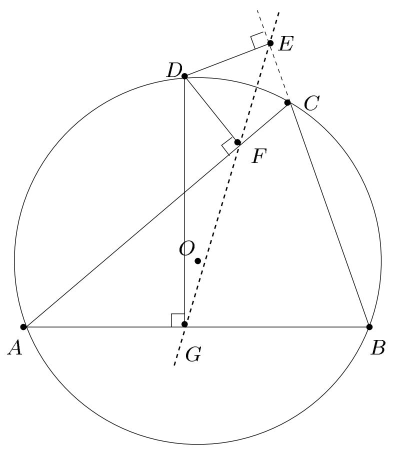


Figure 4.2: Simson's theorem.

Let $A = (0, 0)$, $B = (1, 0)$, $C = (u_1, u_2)$, $O = (x_1, x_2)$, $D = (x_3, u_3)$, $E = (x_4, x_5)$, $F = (x_6, x_7)$ and $G = (x3, 0)$. Here u_1, u_2 and u_3 are independent variables whose value can be freely assigned. For instance, the location of the three initial points A, B and C can be freely chosen (subject to the non-degenerate condition that they are not co-linear). Note that in this example, we put the point A at the origin and put B on the x-axis and is of a unit distance way from A. Such choices can simplify the algebraic formulation of this theorem and they are justified by the fact that a theorem in Euclidean space is true regardless of the translation, rotation and scaling of the underlying coordinate system. The variable $x_1, x_2, \ldots, x_7$'s are dependent variables. Their values must satisfy certain polynomial constraints. For example, the center of their circumscribed circle O is fully determined (note that both of its x and y coordinates are dependent variables which are constrained by the locations of A, B and C). The location of the point D has the freedom in one dimension and at the same time is constrained by the requirement that it must lie on the circle O.

The incremental constructions are translated into new variables and new constraint polynomial equations on these variables. Since we are only concerned about ruler-and-compass constructions, these constraint polynomials are at most quadratic. Table 4.4 gives the polynomial equations for the hypotheses in Simson's theorem.

The conclusion that E, F and G are collinear can be translated to the following polynomial equation:

$$g = x_5x_6 + (-x_4 + x_3)x_7 - x_3x_5 = 0.$$

Now we can formalize the problem of automated geometry theorem proving algebraically. For now, we only consider metric geometry only. Given a set H of ℓ polynomial equations (hypothesis) in $\mathbb{R}[u_1, \ldots, u_m, x_1, \ldots, x_n]$ where $\mathbf{u} = \{u_1, u_2, \ldots, u_m\}$

Table 4.4: Polynomial equations for the hypotheses in Simson's theorem

Equation	Geometry	Remark
$h_1 : 2x_1 - 1 = 0$	$[OA \equiv OB]$	Introduces x_1
$h_2 : 2u_1x_1 + 2u_2x_2 - u_2^2 - u_1^2 = 0$	$[OA \equiv OC]$	Introduces x_1, x_2
$h_3 : -x_3^2 + 2x_1x_3 + 2u_3x_2 - u_3^2 = 0$	$[OA \equiv OD]$	Introduces x_3
$h_4 : u_2x_4 + (-u_1 + 1)x_5 - u_2 = 0$	$[E \in BC]$	Introduces x_4, x_5
$h_5 : (u_1 - 1)x_4 + u_2x_5 + (-u_1 + 1)x_3 - u_2u_3 = 0$	$[DE \perp BC]$	Introduces x_4, x_5
$h_6 : u_2x_6 - u_1x_7 = 0$	$[F \in AC]$	Introduces x_6, x_7
$h_7 : u_1x_6 + u_2x_7 - u_1x_3 - u_2u_3 = 0$	$[DF \perp AC]$	Introduces x_6, x_7

are m independent variables (parameters) and $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ are n dependent variables:

$$\begin{aligned} h_1(u_1, \ldots, u_m, x_1, \ldots, x_n) &= 0 \\ h_2(u_1, \ldots, u_m, x_1, \ldots, x_n) &= 0 \\ &\vdots \\ h_n(u_1, \ldots, u_m, x_1, \ldots, x_n) &= 0 \end{aligned}$$

and a polynomial equation (thesis) $g(u_1, \ldots, u_m, x_1, \ldots, x_n)$ also in $\mathbb{R}[\mathbf{u}, \mathbf{x}]$, decide whether the following statement is true:

$$(\forall u_1, \ldots, u_m, x_1, \ldots, x_n)\ [(h_1 = 0) \wedge (h_2 = 0) \wedge \cdots \wedge (h_\ell = 0) \quad \Rightarrow \quad g = 0]. \quad (4.8)$$

Generic truth. The theorems in the form of (4.8) is called *universal truth.* They are valid in every point in the parameter space $\mathbb{R}^m$. But many theorems in elementary geometry have implicit *degenerate conditions.* A theorem is either not true or not valid at all in degenerate cases. For example, the non-degeneracy condition for the Simson's theorem is that the three initial points A, B and C are not co-linear. Wu proposed the notion of *generic truth*. A statement is *generically true* relative to a set of non-degenerate conditions $\{\Delta_1, \ldots, \Delta_k\}$ if

$$(\forall \mathbf{u}, \mathbf{x})[(\Delta_1, \Delta_2, \ldots, \Delta_k, (h_1 = 0) \wedge \cdots \wedge (h_\ell = 0)) \Rightarrow g = 0]. \tag{4.9}$$

In this definition, the theorem is considered trivially generically true in the degenerate cases.

The non-degeneracy conditions Δ_i's are predicates on the variables in the system. The theorems proved by Wu's method are about unordered geometry in an algebraically closed field (such as metric geometry in $\mathbb{C}$). For these theorems, the non-degeneracy condition usually has the form

$$\Delta : \delta(\mathbf{u}, \mathbf{x}) \neq 0 \tag{4.10}$$

where δ is a polynomial on the independent and dependent variables. We call this *non-degeneracy condition of the first type*. Non-degeneracy conditions for theorems in a real closed field (such as $\mathbb{R}$) often have the form of general inequalities:

$$\Delta : \delta(\mathbf{u}, \mathbf{x}) \diamond 0, \diamond \in \{<, \leq, >, \geq\}, \tag{4.11}$$

which are defined as the *second type non-degeneracy conditions*.

Ordered geometry Introducing order relation into the underlying geometry would make the automated theorem proving problem considerably more complicated. Basically this has to admit inequalities and inequations into the algebraic system. The associated field of every ordered geometry is an ordered field. In this part, we mainly focus on the ordinary Euclidean Geometry, the associated field of which is just the usual real field $\mathbb{R}$.

Admission of inequality relations It is well known [56] that the satisfiability problem regarding inequality relations among numbers in a real closed field (e.g., $\mathbb{R}$) can be transformed to the solvability problem of some equations in the same field. For example, suppose e is an element in $\mathbb{R}$,

$$\begin{aligned} e > 0 &\iff \exists z \in \mathbb{R}, ez^2 = 1 \\ e \geq 0 &\iff \exists z \in \mathbb{R}, z^2 - e = 0 \\ e < 0 &\iff \exists z \in \mathbb{R}, ez^2 = -1 \\ e \leq 0 &\iff \exists z \in \mathbb{R}, e + z^2 = 0. \end{aligned}$$

With these relations, the algebraic formulation discussed above can be extended to accommodate inequality relations which appear in the hypotheses and non-degenerate conditions. For these cases, we can transform the inequalities by applying the equivalent relations above. Suppose there are k inequalities, we need to introduce k new variables $\{z_1, z_2, \ldots, z_k\}$ and k new polynomials $f_1, f_2, \ldots, f_k$. Then, an equivalent form of the theorem statement is as follows:

$$(\forall \mathbf{u}, \mathbf{x})[(h_1 = 0) \wedge \cdots \wedge (h_\ell = 0) \wedge (\exists z_1, \ldots, z_k \ (f_1 = 0) \wedge \cdots \wedge (f_k = 0)) \Rightarrow g = 0]. \tag{4.12}$$

Recall that in the first-order predicate logic, we know that if x is not free in q then

$$\vdash ((\mathbf{Q}x \ p) \to q) \leftrightarrow (\mathbf{Q}^* x) \ (p \to q),$$

where $\mathbf{Q}$ is a universal or existential quantifier and $\mathbf{Q}^*$ is its dual. Because for all the new variables $z_i's$ under existential quantifiers do *not* appear in the right-hand side g (i.e., are not free in the conclusion), we know that the statement 4.12 is equivalent to the following:

$$(\forall \mathbf{u}, \mathbf{x}, \mathbf{z}) \ [((h_1 = 0) \wedge \cdots \wedge (h_\ell = 0) \wedge (f_1 = 0) \wedge \cdots \wedge (f_k = 0)) \ \Rightarrow \ g = 0]. \tag{4.13}$$

Thus the problem has been reduced to a polynomial inference problem again.

If the right-hand side is an inequality, that is more complicated. Although we can move the existential quantifier to the outside scope, unfortunately, it remains existential. The resulting problem is beyond the capacity of pure polynomial inference. Of course, such cases can still be solved by Taski's decision procedure based on algebraic cell decomposition.

4.2.3 Probabilistic Proving by Random Examples

We apply our randomized zero testing of radical expressions (see Theorems 4.10 and 4.18) in probabilistic proving of elementary geometry theorems with ruler-and-compass constructions. What we mean by "probabilistic theorem proving" is this: if a conjecture is true, our prover can always verify it positively; otherwise, the probability that the conjecture is false but we fail to detect it (the *failure probability*) can be made less than an arbitrarily small number ϵ.

Reduction to Radical Expressions For theorems with ruler-and-compass constructions only, each polynomial in H can be at most quadratic. Thus, we can solve each dependent variable explicitly as a radical expression step by step. Clearly, each dependent variable x can eventually be represented as a radical expression $x(u_1, \ldots, u_m)$ over the initial parameters $u_1, \ldots, u_m$. Let $G = G(u_1, \ldots, u_m)$ be the radical expression corresponding to the thesis g after substituting each appearance of x_i in g with the solution $x_i(u_1, \ldots, u_m)$. Thus, proving a theorem amounts to show that $G = 0$ for all valid parameter configurations. Note that when solving quadratic equations about a dependent variable x, we obtain two solutions, up to a sign flip of square roots contained in G. Suppose there are r square roots in G, then we should test simultaneous vanishing of all the 2^r cases, i.e.:

$$G_1 = G_2 = \cdots = G_{2^r} = 0,$$

where G_i is obtained from different sign combinations of the r square roots. If a single function G^* is desired, we can use $G^* = \sum_{i=1}^{2^r} G_i^2$. Sometimes, we can reduce the number of test cases by exploiting symmetry of the different configurations.

In our implementation, radical expressions are represented as DAGs. Let $\pi(\mathbf{u})$ be the natural SLP which computes $G^*(\mathbf{u})$. We compute udeg$(U(G^*))$ and udeg$(L(G^*))$ inductively on the structure of the DAG using the rules presented in Table 4.3.

By Theorem 4.18, we can then prove the vanishing of G^* probabilistically by testing its vanishing on some random examples in which the values of parameters $\{u_1, \ldots, u_m\}$ are chosen from a test set S with appropriate size. Let $D = 2^{2r}$ udeg(π) be the degree bound, the failure probability c of each single test is given by

$$c = D/|S|. \tag{4.14}$$

When the cardinality of the test set S is greater than 2^{2r} udeg(π), the failure rate $c < 1$. Moreover, if we run the prover with n independent examples on the same conjecture, the failure probability becomes c^n.

It is clear that the size of S depends on the rational degree and the number of square roots in G^*. When implemented using exact arithmetic, the bit-length of input parameters affects the complexity of verification of vanishing of an instantiated radical expression. In order to achieve certain failure probability ϵ, we can have a trade-off between the size of test set $|S|$ and the number of runs n as long as $c^n \leq \epsilon$. For simplicity, we assume $\epsilon = 2^{-p}$ and $|S| = 2^k D$ (i.e., the maximum bit-length of inputs are $(k + \lg D)$.) for some natural numbers p and k, and hence $c = 2^{-k}$. Thus the parameters

k and n must satisfy that

$$n \cdot k \geq p.$$

Let $M_{G^*}(L)$ be the asymptotic complexity for testing whether an instance of the radical expression G^* is zero or not using the exact sign determination techniques we discussed in Chapter 2. Here L is the bit length of input parameters in an instance. Then the total cost of proving a theorem with a failure probability ϵ is

$$n \cdot M_{G^*}(k + \lg D).$$

The function M_{G^*} depends on the structure of a radical expression and its root bound. An examination of the rules for various root bounds can reveal that usually the root bound is linear to the bit length L of inputs. In practice, fine tuning is necessary to find the optimal combination of the two parameters n and k.

An alternative to testing $G^*(\mathbf{u}) = 0$ is to view the problem as testing the simultaneous vanishing of a set of polynomial $\mathcal{G} := \{G_1(\mathbf{u}), \ldots, G_{2^r}(\mathbf{u})\}$. Although we must test 2^r radical expressions, for each test the complexity is reduced in two ways:

- The root bound (which determines the precision necessary to numerically determine the sign of radical expressions in the Core Library) is smaller.
- The size of the test set S is smaller.

Handling of non-degeneracy conditions In presence of non-degeneracy conditions, there are two natural models of what it means to have a failure probability ϵ: (A) The "strict model" says that our sample space is now restricted to $S^m \setminus \{\mathbf{a} : \mathbf{a} \text{ is degenerate.}\}$. (B) Alternatively, we can say that the sample space is still S^m but the theorem is trivially true at $S^m \cap \{\mathbf{a} : \mathbf{a} \text{ is degenerate.}\}$.

In the current implementation of our prover, we discard the degenerate sample configurations. Hence, the failure probability is relative to the model A. Suppose there are k type 1 non-degenerate conditions, which are at most quadratic. We can write the non-degeneracy conditions as $\delta^* := \delta_1\delta_2\cdots\delta_k \neq 0$. The degree of δ^* is thus at most $2k$ and forms a surface in the parameter domain $\mathbb{R}^m$. Moreover, δ^* can be seen as a radical expression in the parameter $\mathbf{u}$. Given a finite test set S, the possible zeros (degenerate configurations) in S^m is at most 2^{2r} udeg(δ^*) $|S|^{m-1}$. To guarantee certain failure probability, we can make the test set S large enough so that the probability that degenerate cases are chosen (i.e., 2^{2r} udeg(δ^*) $/|S|$) is arbitrarily small.

Next, we consider the complexity of this approach. In [55], the following lemma is shown:

Lemma 4.19. *Let the thesis polynomial be $g(\mathbf{u}, \mathbf{x})$ with t terms and $\deg(g) = d$, and $G(\mathbf{u})$ be any of the 2^r radical expressions derived from $g(\mathbf{u}, \mathbf{x})$ by eliminating dependent variables. Then* rdeg$(\pi(G)) \leq tdc^k$ *where k is the number of construction stages and c is some constant.*

Proof.

First let us prove the bound on the rational degrees of programming variables, by induction on the steps of construction.

There are a limited number of geometry statements of constructive type. Chou's book [10] gives a complete list of them. Since there are only limited number of possible constructions, The depth of these radical expressions must be bounded by some constants.

Let us consider the new variables (x_i, y_i) introduced in the i-th stage. The coordinates x_i and y_i can be represented as radical expressions built upon the programming variables previously introduced in the first $(i-1)$ stages. These variables appear in the leaves of these radical expressions. By the induction hypothesis, the rational degrees of all these variable are bounded by some number b_{i-1}. Through examining the definition of rational degrees, we can see that since the radical expressions are of constant depths, the ration degrees of x_i and y_i are bounded by cb_{i-1} for some constant c. This constant only depends on the structure of various radical expressions for the finite constructions. Thus, we have $b_k = \mathcal{O}(c^k)$.

Therefore, from Lemmas 4.5 and 4.6, we can see that for the final thesis radical expression $\pi(G)$, we have rdeg$(\pi(G)) \leq tdc^k$.

Q.E.D.

Note that in [55], the constant $c = 8$ was indicated.

Corollary 4.20. *Let the thesis polynomial be $g(\mathbf{u}, \mathbf{x})$ with t terms and $\deg(g) = d$, and $G(\mathbf{u})$ be any of the 2^r radical expressions derived from $g(\mathbf{u}, \mathbf{x})$ by eliminating dependent variables. Then $u_{\pi(G)} \leq 2^r tdc^k$ where k is the number of construction stages and c is some constant.*

Proof. This is a direct result of Lemma 4.17 and Lemma 4.19. **Q.E.D.**

The following theorem gives the complexity of our prover.

Theorem 4.21. *Suppose T is a conjecture about ruler & compass constructions with m independent variables, n dependent variables, r quadratic equations and type 1 non-degenerate conditions, and its thesis polynomial $g(\mathbf{u}, \mathbf{x})$ has t terms and the degree d. Then T can be verified with failure probability $\leq 2^{-k}$ in time polynomial in the parameters m, n, t, d, k and exponential on r. Specifically, the time complexity is $\mathcal{O}(pM(pL2^{2r}))$ where p is $\mathcal{O}(m+n+t+d)$, L is $\mathcal{O}(r+k+m+n+\lg t+\lg d)$ and*

$M(n)$ denotes the complexity of multiplying two n-bit integers.

Proof. The parameter p is a bound on the number of operations in the radical expression $G^*(\mathbf{a})$. Thus, the cost to construct the thesis expressions $G^*(\mathbf{a})$ is also $\mathcal{O}(m+n+t+d)$. Next, let us consider the complexity in verifying $G^*(\mathbf{a})$ for some sample configuration $\mathbf{a} = (a_1, a_2, \ldots a_m)$ randomly chosen from a finite test set S with a cardinality of $2^{2r+k}tdc^{(m+n)}$. By Theorem 4.18 and Corollary 4.20, we know that the failure probability of this test is at most 2^{-k}. Note that the number of construction stages are at most $(m+n)$. Moreover, this means the maximum bit length of input parameters $L = \lg(|S|)$ can be bounded by a polynomial on r, k, m, n, $\lg(t)$ and $\lg(d)$. By examining the entries in Tables 2.4 and 2.6, we can see that the logarithms of $\mathrm{lc}(G^*)$ and $\overline{\mu}(G^*)$ depend on L linearly. So the root bit-bound at the top is $\mathcal{O}(L2^{2r})$. At the root node, we need to compute an approximation up to the precision specified by the root bound. The computation in the intermediate nodes might require higher absolute precision (e.g., in a multiplication operation. See Section 3.2.3). But the increment on precision at each step depends linearly on the MSB of its operands. An upper bound on the value of each node is given by the Equation (2.10), and hence an upper bound on the MSBs of all the nodes is $\mathcal{O}(L2^{2r})$. Therefore, the number of bits that need to be computed at each nodes is bounded $\mathcal{O}(pL2^{2r})$. The arithmetic operation taken at each node costs $\mathcal{O}(M(pL2^{2r}))$ where $M(n) = \mathcal{O}(n \lg(n) \lg\lg(n))$ denotes the complexity to multiply two n-bit integers. There are at most p operations. Thus, the total cost in verifying a conjecture T with a failure probability no more than 2^{-k} is $\mathcal{O}(pM(pL2^{2r}))$ where p is $\mathcal{O}(m+n+t+d)$ and L is $\mathcal{O}(r+k+m+n+\lg t+\lg d)$.

Q.E.D.

Experiments We implement a probabilistic prover using the Core Library, which can test the vanishing of constant radical expressions correctly. The library is directly modified so that we can utilize the expression DAG structure in computing the exact rational degrees of the expressions (rather than use the estimates of the Lemma 4.19).

We prove some theorems from Chou [10]. The timings are for two values of k (this means the probability of error c is at most 2^{-k}). We also arbitrarily "perturb" the hypothesis of each theorem by randomly changing one coefficient of one of the input polynomials, and report their timings as well. These are all false theorems, naturally. Our tests were performed on a Sun UltraSPARC-IIi (440 MHz, 512 MB). The times are all in seconds, and represent the average of 6 runs each. The final column in the table gives the page number in Chou's book [10].

No.	Theorem	$k = 10$	$k = 20$	Perturbed	Page
1	Pappus	0.020	0.020	0.007	100
2	Pappus Point	0.152	0.147	0.025	100
3	Pappus-dual	0.017	0.023	0.008	111
4	Nehring	4.380	5.870	0.102	115
5	Chou-46	0.059	0.083	0.022	124
6	Ceva	0.027	0.033	0.010	264
7	Simson	70.318	39.000	0.017	240
8	Pascal	1715.8	2991.6	0.037	103

Let r be the number of square roots in the radical expression representing a theorem. If $r = 0$, we say the theorem is linear. A large part[1] of the 512 theorems in Chou's book are linear. Only the last two theorems (Simson and Pascal) in the above list are non-

[1]The theorems in Chou's book include an original list of 366 theorems from [9], of which 219 are reported to be linear [10, p. 12].

linear, with $r = 1$ and $r = 5$, respectively. In [55], we compared our timing with those based on Wu's method and Gröbner basis which shows that our approach is very effective on linear cases, but non-linear theorems still represent a challenge for our current system.

It is interesting to note that we have never observed a single wrong conclusion from our probabilistic tests – all the theorems are reported as true, and all perturbed theorems are reported as false. In some sense, that is not surprising because the probabilistic bounds based on Schwartz's lemma seem overly conservative in all real situations.

The running times for linear theorems are pretty consistent across different runs. However, for the non-linear theorems, the timing can show much more variation. This is not unexpected since the running time depends on the bit size of the random example. A more prominent behavior comes from the clustering of times around certain values. For instance, for Simson ($k = 20$), the times cluster around 10 seconds and around 70 seconds. This "multimodal" behavior of the timings are again seen in Pascal. This can be attributed to the random choice of signs for the radicals in nonlinear theorems. This may also account for the curious relative times for Simson $k = 10$ and $k = 20$.

Our method is extremely effective for discarding wrong or perturbed conjectures. In rejecting false conjectures, Wu's method would take the same procedure and time as for the true theorems. The ability to quickly reject false theorems is extremely useful in applications where the user has many conjectures to check but most of the conjectures are likely to be false.

4.3 Summary

In this chapter, we have developed a generalization of Schwartz's randomized zero test to check the vanishing of radical expressions. Our Core Library is exploited for exact sign determination. We shall note that the zero test of multivariate radical expressions is an important problem by itself and has independent interest beyond automated theorem proving. We expect this new method to have many applications as radical expressions are quite common.

We apply the zero test method in proving elementary geometry theorems about ruler-and-compass constructions. We develop an probabilistic approach and implement a prover using the Core Library. Some features of our prover include:

- It is probabilistic and allows trade-off between speed and failure probability.
- It rejects wrong conjectures very quickly.
- It exploits the special nature of ruler-and-compass constructions.
- It is very effective for linear theorems.
- We need to gain much more empirical data.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this dissertation, we investigate the theory and some applications of the Exact Geometric Computation (EGC) approach to robust geometric computation.

We study the problem of exact comparison of algebraic expressions. Our work show that numerical approximation together with constructive root bounds is a useful and efficient way other than the usual symbolic computation approaches in determining the exact sign of algebraic numbers. The worst-case complexity of our approach is determined by algebraic root bounds. In Chapter 2, we presented a new constructive root bound [35] for general algebraic expressions that can provide significant speed-up over previously known bounds in many common computations involving divisions and radical roots. This new bound can be computed inductively on the structure of an expression DAG using the rules in Tables 2.4 and 2.6. These rules are derived from resultant calculus. Moreover, we improved the well-known degree-measure bound [40, 41, 2] by exploiting the sharing of common sub-expressions.

As a part of our work, We developed the Core Library [23, 29, 58, 35], a compact, easy to use and efficient C++ library for exact numeric and geometric computation which incorporates the precision-driven EGC paradigm. It is very easy for users to use our library to implement new robust geometric applications or make existing programs robust. In Chapter 3, we discussed the design and implementation of the Core Library, with the focus on its top layer (i.e., the Expr package) which is the most interesting part in our library and the core for precision-driven exact computation. The current implementation supports radical expressions with square roots. The library can find applications in many areas where it is critical to guarantee the numerical precision.

In Chapter 4, we generalized Schwartz's well-known probabilistic verification of polynomial identities, and devised a probabilistic method to test the vanishing of multivariate radical expressions. We test the vanishing of a radical expression on sample instances which are randomly chosen from a finite test set. These sample tests are performed using the Core Library, and the size of the finite test set affects the running time. Following our previous work in [55], in this thesis we gave a simplified definition of rational degrees which can leads to more efficient computation of upper bounds on the cardinality of the finite test set. Moreover, we sharpened this upper bound for radical expressions with divisions, and hence improved the performance when exact computation is employed. Based on this randomized test for radical expressions, we proposed a probabilistic approach [55] to prove elementary geometry theorems with ruler-and-compass constructions only. Our approach can reject false conjectures very quickly. An automated theorem prover based on this method has been implemented using our Core Library. The experiments show that it is very effective for theorems with linear constructions.

5.2 Future Work

On root bounds and sign determination The performance of EGC is still constrained by conservative root bounds. More research is needed in sharpening our new root bound and in searching for other better root bounds.

Finding better ways of bounding the tail coefficient $\mathrm{tc}(E)$ for addition and subtraction nodes in our new root bound is a challenge for future work. The current bound based on polynomial measures is conservative for expressions with complex structure and large depth. Also, it may be possible to improve our current lower bound on conjugates $\underline{\nu}(E_1 \pm E_2)$.

Many existing bounds are obtained through bounding some properties regarding minimal polynomials of algebraic expressions. We have ongoing research on a novel idea to compute root bounds in extension fields. Instead of bounding minimal polynomials in $\mathbb{Z}[x]$, we bound representation polynomials of elements in some algebraic extensions $\mathbb{Q}(\gamma)$ for some algebraic number γ. As the computation goes on, the extension fields may be further extended (e.g., when new radical nodes are introduced.). Besides bounding the representation polynomials, we also need to maintain some bounds about the "primitive" element of each extension. Note that if there are no new radical or polynomial root nodes introduced at a step, then the result of that step is still in the same extension field as its operands'. An observation is that in many computations, the extension of fields does not happen very frequently. An advantage of this new approach is that if the current field is not extended as a consequence of an operation, then we can obtain the representation polynomial of the result simply by a direct polynomial operation (addition, subtraction or multiplication, etc.) on the representation polynomials of its operands in the same field. In this way, we can bound the desired properties (say, polynomial norms) of this resulting polynomial representation tightly. For example, given two algebraic numbers α and β with the degrees m and n respectively, the polynomial for $\alpha \odot \beta$ constructed through resultant calculus has a degree $m \cdot n$. But if they are in the same extension field $\mathbb{Q}(\gamma)$ and can be represented in polynomials of γ with the degrees m' and n', then the result can be represented as a polynomial of γ with a degree at most $m' + n'$.

Another important topic is to compare different root bound theoretically and experimentally. This is of special significance considering that many of the current root bounds are generally incomparable. We want to compare their behavior on more general and interesting classes of expressions. We also need to observe their performance in more experiments.

Many conditional tests in computational geometry programs are equality tests (e.g. detection of degeneracy). In these cases, we do not have to know the exact sign of the difference of two expressions being compared. What we need is just to determine whether it is zero or not. This seems to be an easier problem and perhaps could be better handled by other methods (e.g. symbolic or semi-numerical approaches) than the root bound based approaches commonly adopted in EGC. The reason for separating zero test with sign determination is that because conservative root bounds could force expensive numerical computation when the sign is really zero (recall that in such cases the absolute precision of approximation has to reach the root bound.). Besides addressing equality tests, zero test can also be used as a filter in general computation to avoid the expensive sign determination when an expression is exactly zero. Moreover, if we know an expression is non-zero, we can decide its sign through progressive approximation until a definite sign comes out (note that root bounds are not used here). More study on the algorithms and complexity of zero test is an interesting topic for future exploration.

On the development of Core Library We need to further improve the efficiency of our Core Library, at both the algorithmic level and the system level.

An interesting topic is partial evaluation of expressions. Although persistent structure reflects the usual arithmetic semantics supported by most programming languages, a dynamic expression DAG in which the leaves can be variables instead of constants could be useful in certain applications, such as interactive editing. Moreover, in many cases, a large part of one predicate expression may be relatively fixed during consecutive evaluations. So it could speed up the performance if the fixed part in an expression can be partially evaluated beforehand. For example, in the simple $\mathcal{O}(n^4)$ algorithm to compute Delaunay triangulation of a set of points in a plane, each in-circle test (computing the sign of a 4×4 determinant) would be applied to $(n-3)$ points. Therefore, if we partially evaluate the fixed part (in this case, that could be four 3×3 minors) first, then in every test thereafter, we can just plug in the new parameters and the evaluation would avoid repetitive computation of the invariant subexpressions every time.

We plan to extend our library to support more general algebraic expressions that contain the roots of polynomials with algebraic numbers as coefficients.

Another important future work is to apply the EGC techniques that we have developed to other areas of computation where guaranteeing the absolute or relative precision is critical. In particular, we want to deploy our Core Library to a wider range of applications, such as linear programming and optimization.

Bibliography

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computation*. Academic Press, New York, 1983.

[2] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica*, 27:87–99, 2000.

[3] Christoph Burnikel, Stefan Funke, and Michael Seel. Exact geometric predicates using cascaded computation. *Proceedings of the 14th Annual Symposium on Computational Geometry*, pages 175–183, 1998.

[4] Christoph Burnikel, Jochen Könnemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Proc. 11th ACM Symp. Computational Geom.*, 1995.

[5] John Francis Canny. *The complexity of robot motion planning*. ACM Doctoral Dissertation Award Series. The MIT Press, 1988. PhD thesis, M.I.T.

[6] G. Carrà-Ferro, G. Gallo, and R. Gennaro. Probabilistic verification of elementary geometry statements. In *Proceedings of the International Workshop on Automated Deduction in Geometry (ADG'96)*, volume 1360 of *LNAI*, pages 87–101, 1997.

[7] B. Chazelle et al. Application challenges to computational geometry. In *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 407–463. AMS, 1999. The Computational Geometry Impact Task Force Report (1996).

[8] S.-C. Chou and W. F. Schelter. Proving geometry theorems with rewrite rules. *J. of Automated Reasoning*, 2(3):253–273, 1986.

[9] Shang-Ching Chou. Proving geometry theorems using Wu's method: A collection of geometry theorems proved mechanically. Technical Report 50, Institute for Computing Science, University of Texas, Austin, July 1986.

[10] Shang-Ching Chou. *Mechanical Geometry Theorem Proving*. D. Reidel Publishing Company, 1988.

[11] G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, Lecture Notes in Computer Science, No. 33, pages 134–183. Springer-Verlag, Berlin, 1975.

[12] R. Dentzer. libI: Eine lange ganzzahlige Arithmetik, 1991.

[13] Steven J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[14] Steven J. Fortune. Stable maintenance of point-set triangulations in two dimensions. *IEEE Foundations of Computer Science*, 30:494–499, 1989.

[15] Steven J. Fortune and Christopher J. van Wyk. Static analysis yields efficient exact

integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[16] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[17] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. *IEEE Foundations of Computer Science*, 27:143–152, 1986.

[18] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. *ACM Symp. on Computational Geometry*, 5:208–217, 1989.

[19] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick. Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl.*, 9(2):50–59, 1989.

[20] Christoff M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3), March 1989.

[21] The CGAL Homepage. Computational Geometry Algorithms Library (CGAL) Project. URL `http://www.cs.uu.nl/CGAL/`.

[22] CLN – Class Library for Numbers Homepage. URL `http://clisp.cons.org/~haible/packages-cln.html`.

[23] The CORE Project Homepage. URL `http://www.cs.nyu.edu/exact/`.

[24] The GNU MP Homepage. URL `http://www.swox.com/gmp/`.

[25] The LEDA Homepage. URL `http://www.mpi-sb.mpg.de/LEDA/`.

[26] LiDIA Homepage, 1998. LiDIA: an efficient multiprecision number package for computational number theory. URL `http://www.informatik.th-darmstadt.de/TI/LiDIA/`.

[27] IEEE. IEEE standard 754-1985 for binary floating-point arithmetic, 1985. Reprinted in SIGPLAN 22(2) pp. 9-25.

[28] D. Kapur. Geometry theorem proving using Hilbert's Nullstellensatz. In *Proceedings of 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC 86)*, pages 202–208, 1986.

[29] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core Library for robust numeric and geometric computation. In *Proceedings of the Fifteenth ACM Symposium on Computational Geometry (SoCG 1999)*, pages 351–359, June 1999.

[30] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Trans. on Graphics*, 10:71–91, 1991.

[31] Donald Ervin Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 3 edition, 1997.

[32] B. Kutzler and S. Stifter. Automated geometry theorem proving using Buchberger's algorithm. In *Proceedings of 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC 86)*, pages 209–214, 1986.

[33] Arjen Lenstra. lip: long integer package, 1989.

[34] Chen Li and Chee Yap. *Tutorial for CORE Library: A Library for Robust Geometric Computation*. Courant Institute of Mathematical Sciences, 251 Mercer St., New York, NY 10012, USA, 1.3 edition, September 1999.

[35] Chen Li and Chee Yap. A new constructive root bound for algebraic expressions. In *Proceedings of the Twelfth ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 496–505, January 2001.

[36] Ming C. Lin and Dinesh Manocha, editors. *Proceedings of the First ACM Workshop on Applied Computational Geometry*, 1996.

[37] G. Liotta, F. Preparata, and R. Tamassia. An illustration of degree-driven algorithm design. *Proc. 13th Annual ACM Symp. on Computational Geometry*, pages 156–165, 1997.

[38] Morris Marden. *The geometry of the zeros of a polynomial in a complex variable*. American Mathematical Society, 1949.

[39] K. Mehlhorn and S. Schirra. A generalized and improved constructive separation bound for real algebraic expressions. Technical report, Max-Planck-Institut für Informatik, November 2000.

[40] M. Mignotte. Identification of algebraic numbers. *Journal of Algorithms*, 3(3), 1982.

[41] Maurice Mignotte and Doru Ştefănescu. *Polynomials: An Algorithmic Approach*. Springer, 1999.

[42] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988. An earlier version appeared in *Proceedings, Oxford Workshop on Geometric Reasoning*, (eds. Brady, Hopcroft, Mundy).

[43] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[44] R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979.

[45] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998. Source codes can be downloaded at ftp://cs.smith.edu/pub/compgeom.

[46] Kouji Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997.

[47] Victor Y. Pan and Yanqiang Yu. Certification of numerical computation of the sign of the determinant of a matrix. To appear in Algorithmica.

[48] Edward R. Scheinerman. When close enough is close enough. *American Mathematical Monthly*, 107:489–499, 2000.

[49] Stefan Schirra. Robustness and precision issues in geometric computation. Research Report MPI-I-98-1-004, Max-Planck-Institut für Informatik, Saarbrücken, Germany, January 1996.

[50] A. Schönhage and V. Strassen. Schnelle Multiplikation Grosser Zahlen. *Computing*, 7:281–292, 1971.

[51] J. T. Schwartz. Probabilistic verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, October 1980.

[52] Jonathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th ACM Symp. on Computational Geom.*, pages 141–150, May 1996.

[53] Kokichi Sugihara, Masao Iri, Hiroshi Inagaki, and Toshiyuki Imai. Topology-oriented implementation—an approach to robust geometric algorithms. *Algorithmica*, 27:5–20, 2000.

[54] Roberta Tamassia et al. Strategic directions in computational geometry. *ACM Computing Surveys*, 28(4), December 1996.

[55] Daniela Tulone, Chee Yap, and Chen Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. In *Proceedings of the Third International Workshop on Automated Deduction in Geometry (ADG 2000)*, pages 121–136, September 2000. Final version to appear in LNAI series.

[56] Wen-tsün Wu. *Mechanical Geometry Theorem Proving in Geometries*. Springer-Verlag, 1994. Originally published as Basic Principles of Mechanical Theorem Proving in Geometries (in Chinese) by Science Press, Beijing, 1984.

[57] C. K. Yap. Robust geometric computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, 1997.

[58] Chee Yap and Chen Li. Recent developments in Core Library. In *10th Annual Fall Workshop on Computational Geometry (WCG 2000)*, October 2000. Stony Brook, New York.

[59] Chee K. Yap. Towards exact geometric computation. In *Fifth Canadian Conference on Computational Geometry*, pages 405–419, Waterloo, Canada, August 5–9 1993. Invited Lecture.

[60] Chee K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7:3–23, 1997. Invited talk, Proceed. 5th Canadian Conference on Comp. Geometry, Waterloo, Aug 5–9, 1993.

[61] Chee K. Yap and Thomas Dubé. The exact computation paradigm. In D.-Z. Du

and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, 1995. 2nd edition.

[62] Chee Keng Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford Univ. Press, December 1999.