CHAPTER 1

ON GUARANTEED ACCURACY COMPUTATION

Chee K. Yap[a]
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York
NY 10012, USA
E-mail: yapcs.nyu.edu

The concept of *guaranteed accuracy computation* is a natural one: the user could specify any *á priori* relative or absolute precision bound on the numerical values which are to be computed in an algorithm. It is a generalization of *guaranteed sign computation*, a concept originally proposed to solve the ubiquitous problem of non-robustness in geometric algorithms. In this paper, we investigate some basic properties of such computational mode. We formulate a theory of real computation and approximation to capture guaranteed accuracy computation. We also introduce an algebraic and a numerical model of computation based on Schönhage's pointer machines.

1. Introduction

Numerical nonrobustness of computer programs is a ubiquitous phenomenon: it is experienced as program failures (crashes) that inevitably happen when the program is run on certain combinations of logically valid inputs. One approach to solving such problems is to compute "exactly", but only in the geometric sense[55]. This is called *Exact Geometric Computation* (EGC, for short). The basic idea of EGC is to ensure that each

[a]The work is supported by NSF/ITR Grant #CCR-0082056. This paper is an expansion of two keynote talks with the same title, at the National Computer Theory Conference of China, Changsha, China, Oct 13-18, 2002, and at the International Conference on Computational Science and its Applications (ICCSA 2003), Montreal, Canada, May 18-21, 2003.

computed real number $\widetilde{x}$ has the same sign as the exact value x for which $\widetilde{x}$ is an approximation. In particular, if $x = 0$ then $\widetilde{x} = 0$. We may call this *guaranteed sign computation*. Within the last 10 years, the EGC approach has emerged as the most successful approach to numerical nonrobustness. Unlike many approaches that require a case-by-case application of some general principle, the EGC solution to nonrobustness can be provided through the use of a general number library. Such a library provides *EGC numbers*, a designation that means the numbers support guaranteed sign computation. Two such number libraries are currently available: `LEDA Real`[12,35] and `Core Library`[30]. Using such libraries, programmers can routinely implement robust programs by using standard algorithms (not specially crafted "robust" algorithms). A large collection of such robust algorithms have been implemented in the major software libraries `CGAL`[20,27] and `LEDA`[35,14,28]. Many novel computing techniques to support EGC have been developed in the last decade, including new efficient guaranteed sign algorithms[10], floating point filters and its generalizations[9,23] and constructive zero bounds[13,44].

In this paper, we investigate a generalization of guaranteed sign: we want to *guarantee numerical accuracy*. This means that we want to be able to specify *á priori* any number of correct bits in each computed numerical quantity $\widetilde{x}$. This ability is desirable in various applications. One example comes from numerical statistical computations[34]. McCullough[17] describe the problem of evaluating the accuracy of statistical packages. One basic task here is to pre-compute model answers for standardized test suites. We need a certain guaranteed numerical precision in the model answers in order to evaluate the answers produced by commercial statistical packages. At the National Institute of Standards and Technology (NIST), such model answers must have 15 digits of accuracy, and these are generated by running the program at 500 bits of accuracy. It is by no means clear that 500 bits is sufficient; it is also possible 500 bits is sometimes more than is strictly necessary. What we would like is software that automatically computes to the accuracy that is sufficient to guaranteed the final 15 digits. Dhiflaoui et al[18] address the problem of guaranteeing the results from linear programming software. Frommer[22] and Tulone et al[50] provide examples of applications in proving mathematical conjectures such as the resolution of the Kepler Conjecture.

Guaranteed accuracy is closely related to several important topics in numerical computing. The first is *arbitrary precision computation*. This is often associated with the well-known concept of Big Numbers[56]. The main

guarantee of such number types is that the ring operations $(+, -, \times)$ are exact: these operations will not overflow or underflow, provided computer memory is available and provided the result is representable in the number type. In the presense of errors, we may add an additional capability: the iteration of a sub-computation at increasing precision. In the programming language `Numerical Turing`[26], this is encoded as a "precision block" (syntactically, it resembles the begin-end block of conventional programming languages). Perhaps guaranteed accuracy is most similar to *interval analysis* or more generally, *enclosure methods*[39,41,33]. As a computational mode, it is often known as[b] *certified accuracy computation*. One form of certified accuracy is *significance arithmetic*[36] where we automatically track the "significance" of the bits in numerical approximations. Such capabilities are found in, e.g., the `BigFloat` class in `Real/Expr`[56] and in the `PRECISE` package[32].

There are three common misconceptions about guaranteed accuracy computation. First is the distinction between guaranteed accuracy and certified accuracy. Consider the problem of computing the determinant of a numerical matrix M. A certified accuracy computation of $\det(M)$ might return with the answer: "the determinant is 12.34 ± 0.02". It certifies the bound $12.32 \leq \det(M) \leq 12.36$. More to the point, the error bound 0.02 is *á posteriori*, and deduced automatically by the computation. It depends not only on the determinant algorithm but also on some implicit accuracy for the basic arithmetic operations. In contrast, in a guaranteed computation of $\det(M)$, the input is a pair (M, θ) where θ is any desire error bound (say $\theta = 0.02$). The computation may return with "the determinant is $12.32 \pm \theta$". As before, this answer is certified. The difference is that the bound θ was given *á priori*.

The second misconception is to think of guaranteed accuracy as simply "iterated certified accuracy", that is, guaranteed accuracy can be achieved simply by repeating a computation with higher and higher certified accuracy. To see that this may not succeed, consider the special case of guaranteed sign where we want to discover the sign of a real quantity x. If x_i is an approximation for x in the i-th iteration, then clearly $x_i \to x$ as $i \to \infty$. Certified accuracy will further furnish us with a bound $\varepsilon_i > 0$ such that $|x - x_i| \leq \varepsilon_i$. In case $x = 0$, the pair (x_i, ε_i) is consistent with the con-

[b]Or *validated*, or *verified*, or *reliable accuracy*. It is sometimes known as "guaranteed accuracy", but this terminology is less established. In this paper, we reserve the guaranteed terminology for our special usage.

clusion that x has any sign $(0, \pm)$. Hence the sign determination algorithm cannot stop in the i-th iteration. Intuitively, *the gap between guaranteed accuracy and certified accuracy is analogous to the gap between total recursiveness and partial recursiveness* (see Section 3). This problem can be located in the so-called *Zero Problem*[45], which we shall treat in its several forms. Richardson[45], one of the pioneers in this area, puts it this way: "most people do not even see this as a problem at all".

Third, it is even less appreciated that there is a non-trivial gap between guaranteed accuracy of individual functions, and their composition. It is well-known that there are algorithms, even efficient ones, to compute most of the well-known mathematical functions (elementary functions, hypergeometric functions, etc) to any guaranteed accuracy. But it is not obvious how the guaranteed accuracy computation of two functions $f, g : \mathbb{R} \to \mathbb{R}$ implies that $f \circ g : \mathbb{R} \to \mathbb{R}$ can also be computed with guaranteed accuracy. This issue is captured in the problem of *expression evaluation*, another theme of this paper.

We hope to outline the basic features of a theory of guaranteed accuracy computation. But this presupposes a theory of real computation. A widely used approach here, following Weihrauch, is the *Type II Theory of Effectivity* (TTE)[51,31]. Weihrauch [Chapter 9][51] surveys several other approaches to computing with real numbers. Another rival approach is the Algebraic Theory of Blum, Shub and Smale (BSS, 1989)[6]. Neither approaches are suitable for us. For instance, approaches to real computing such as TTE concede the key property of guaranteed accuracy computation (namely, equality tests) from the start. Hence our approach cannot be equivalent to such approaches. On the other hand, the BSS Theory does not address issues of numerical approximation which is central in real world computation. Interestingly, one of the aims of the BSS theory [BCSS, Section 1.6][6] is to address complexity issues in numerical analysis.

The starting point of our approach to real computation is the following idea: all numerical inputs as well as intermediate results must be "representable". We axiomatically introduce a set $\mathbb{F} \subseteq \mathbb{R}$ of *representable reals*. For instance, $\mathbb{F}$ is countable but dense in $\mathbb{R}$, and is a ring extension of the integers $\mathbb{Z}$. The role of $\mathbb{F}$ mirrors that of floating point numbers in the world of numerical computing. We initially use the Turing model of computation to study guaranteed accuracy computation based on $\mathbb{F}$. Algebraic operators in our theory are replaced by approximate operators.

Next we introduce a model of numerical computation that lies between the Turing model and the algebraic models. To motivate this, note that the

numerical computation of a function f in the "real world" might be construed in two steps: (A) First find an algebraic algorithm A which computes f in an ideal error-free setting. This algorithm assumes some *basis set* Ω of algebraic operators (such as $\pm, \times$) as primitives. (B) Next, construct a numerical algorithm B that is modeled after A. But B takes into account numerical representation, and accuracy of implementing the primitives of Ω. Algorithm A might be regarded as a program in a suitable algebraic model (e.g., BSS Model, but we will propose another one). But algorithm B does not seem to have a natural theoretical model (the Turing model notwithstanding). We propose to fill this gap by introducing the *Numerical Pointer Model* based on Schönhage's elegant pointer machines[47]. We choose pointer machines to avoid artificial (Gödel) encoding of "semi-numerical" problems. Our main result here answers the following question: *when is a function F that is algebraically computable (over a basis Ω) also numerically approximable?* We give a sufficient condition on Ω.

Overview of Paper

Section 2 reviews the place of guaranteed precision computation in the landscape of numerical computing. A brief description of the `Core Library` implementation of guaranteed accuracy is also given. Section 3 proposes a new approach to computing with real numbers, and gives its main features using Turing computability. The key concept is relative approximability of functions. Section 4 examines in detail the approximability of standard algebraic operators such as $\pm, \times, \div, \sqrt{}$. Section 5 considers the relative approximability of a composition of such algebraic operators (the expression evaluation problem). The role of constructive zero bounds is emphasized. Section 6 describes the Algebraic Pointer Model based on Schönhage's pointer machines. This model is suitable for capturing the algebraic complexity of semi-numerical problems. Section 7 introduces the Numerical Pointer Model, and proves the basic transfer theorem relating algebraic computability with numerical computability. Section 8 closes with some open problems.

Preliminaries: Precision Bounds

We use $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$ for the sets of natural numbers, integers, rationals, reals and complex numbers. Let $x, \widetilde{x} \in \mathbb{C}$. If $\widetilde{x}$ is an approximation to x, the *error* in $\widetilde{x}$ is $|x - \widetilde{x}|$. There are two standard ways to quantify error, relative and absolute. Let $a, r \in \mathbb{R} \cup \{+\infty\}$. We say $\widetilde{x}$ has *a absolute*

bits of precision if $|\widetilde{x} - x| \leq 2^{-a}$ and write

$$\widetilde{x} \in x[a]. \tag{1}$$

Thus the expression "$x[a]$" denotes the interval $x \pm 2^{-a}$. If $a = \infty$ then $\widetilde{x} = x$. Similarly, we say $\widetilde{x}$ has r *relative bits* of precision if $|(\widetilde{x}-x)/x| \leq 2^{-r}$, and write

$$\widetilde{x} \in x\langle r\rangle. \tag{2}$$

We can combine them[56] and say $\widetilde{x}$ *approximates* x *to composite precision* $[a, r]$, written

$$\widetilde{x} \in x[a, r], \tag{3}$$

if $\widetilde{x} \in x[a]$ or $\widetilde{x} \in x\langle r\rangle$. Our terminology allows fractional number of bits. But when a, r are input arguments in a computation, we normally restrict a to $\mathbb{Z}_\infty := \mathbb{Z} \cup \{\infty\}$, and restrict r to $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$. To see why we may restrict r to non-negative values, note that when $r < 0$, then $0 \in x\langle r\rangle$ is always true. The composite precision $[a, r] = [\infty, \infty]$, amounts to asking for an exact answer which may not exist in our computational approach (Section 3).

Note on Terminology: in this paper, we do not distinguish between "accuracy" and "precision" (but see a distinction made by Higham[25]). Our definitions use "precision", leaving the term "accuracy" for informal usage. A related term is "error". But we regard precision and error as complementary views[c] of the same phenomenon.

2. Modes of Numerical Accuracy and the Core Library

In this section, we give an overview of how guaranteed accuracy computation fits into the world of numerical computation. Although the rest of this paper will focus on the theory of guaranteed accuracy, this section will overview a specific system, the `Core Library`[30]. Of course, big number package can also offer guaranteed accuracy as long as the computed numbers remain rational: but our main interest is in systems that admit irrational numbers (in particular square roots and more generally algebraic numbers). Currently, there is one[d] other general implementation of guaranteed accuracy, the `LEDA Real` number type[12,35].

[c] Just as "half-full" (precision or optimistic) and "half-empty" (error or pessimistic) both describe the state of a glass of milk.

[d] It should be noted that `LEDA Real` is part of a much more ambitious system called `LEDA` that provides efficient data structures and algorithms for many standard problems.

Numerical computing involves numbers. Depending on the nature of the problem, the number domain may be $\mathbb{N}$ as in number theory; finite fields as in algebraic coding; $\mathbb{R}$ or $\mathbb{C}$ as in most scientific and engineering computations. Computer algebra deals with more abstract algebraic structures but numerically, the underlying domain is often the algebraic numbers $\overline{\mathbb{Q}}$ (algebraic closure of $\mathbb{Q}$). Because of the diversity of these applications, the field of numerical computing has evolved several "modes" of numerical computation:

- The *symbolic mode* is best represented by computer algebra systems such as `Macsyma`, `Maple` or `Mathematica`. A number such as $\sqrt{2}$ is represented "symbolically" and exactly.
- The *FP mode* is by far the most important mode[e] in numerical computing today. Here numbers are represented by fixed precision numbers, typically machine numbers. In modern hardware, machine numbers have converged to the IEEE Standard[49]. This mode is very fast, and is the "gold standard" whereby other numerical modes are measured against. The main goal of numerical algorithm design in the FP mode is to achieve high numerical accuracy possible within the applicable constraints. The term "fixed precision" needs some clarification since it is clear that the precision θ can be introduced as a parameter in FP algorithms, where $0 \leq \theta < 1$. The algorithms will converge to the exact answer as $\theta \to 0$. E.g., see Chaitin-Chatelin and Frayssé[15], p. 9. Nevertheless, most FP algorithms are *precision oblivious* in the sense that their operations do not adapt to the θ parameter.
- The *arbitrary precision mode* is characterized by its use of multiprecision such as in Big Number Packages. Gowland and Lester[24], and Yap and Dubé[56] are two surveys. Well-known examples include the `MP` Multiprecision Package of Brent[8], and the `MPFun` Library of Bailey[2]. The `iRRAM` Package of Müller[40] has the interesting ability to compute limits of its functions. The ability to reiterate an arbitrary precision computation can be codified into suitable programming constructs, as in the `Numerical Turing` language[29]. Another variant exploits the fact that arbitrary precision arithmetic need not be viewed as monolithic operations, but can be performed incrementally. This gives rise to the *lazy evaluation mode*[37,3].

[e]FP stands for "floating point" or "fixed precision", both of which seems to be characteristic of this mode.

- Of growing importance are various *enclosure modes* such as represented by the use of interval arithmetic. Enclosure methods can be introduced in the FP mode or in the arbitrary precision mode.
- The *guaranteed precision mode* is increasingly used in computational geometry community, at least in the simplest form, of guaranteed sign mode. This mode is the norm in the libraries `LEDA` and `CGAL`. Conceptually, every computational problem is modified for this mode so that, in addition to the usual inputs, one is also given a precision bound. Thus, a function $f(x)$ is replaced by $f(x, \theta)$ where θ is some precision bound. In contrast to the oblivious algorithms in the FP mode, guaranteed precision algorithms will actively adjust its computation to according to θ.

The above modes can overlap, although each mode has typical areas of application and also its own "cultural" practices. Hence it is not easy to fully characterize these modes. But by focusing on their numerical accuracies, we can capture the main features of some of these modes under a common framework. Following Yap[52], we note three "prototype" numerical accuracies which we call *Level I*, *II* and *III accuracies*. These correspond roughly to the FP Mode, the Arbitrary Accuracy Mode and the Guaranteed Accuracy Mode, respectively. In this framework, there is the possibility to combine all three modes within a computation: this we call *Level IV*. Briefly:

Level I or FP Accuracy. For practical purposes, it is identified with the IEEE standard.

Level II or Arbitrary Accuracy. No overflow or underflow occurs in our number representation until some user-specified accuracy (say 2000 bits) is exceeded. Thus $\sqrt{2}$ will be initially be approximated to 2000 bits.

Level III or Guaranteed Accuracy. The computed value of variable is guaranteed to user-specified accuracy, in absolute or relative terms. To guarantee sign of x, we need to compute x to 1 relative bit of accuracy.

Level IV or Mixed Accuracy. Each numerical variable in a computation is given one of the previous three levels of accuracy. This gives the user better control of computational efficiency.

The Core Library

The integration of these levels of accuracy within a single programming framework is one of the main design goals[52] of the `Core Library`[30], a system implemented in `C++`. Providing all 4 accuracy levels in a single programming environment can be achieved by asking programmers to explicitly specify the accuracy level for each variable in their programs. This is essentially Level IV accuracy. But to make this framework widely usable, we want a little more. We would like to execute *any* program (either a standard `C++` program or one that is explicitly written using the `Core Library` constructs) at any Level Accuracy, just by a simple recompilation. For instance, taking an existing `C++` program, we would like to add a simple preamble in order to compile it into a Level X executable (X=I, II, III):

```
#define CORE_LEVEL N       /* N=1,2,3 or 4 */
#include "CORE.h"
// ... STANDARD C++ PROGRAM FOLLOWS ...
```

Thus a single program P can be executed in any of the four levels of accuracy. One then has the potential to trade-off the strengths and weaknesses of the different levels: clearly, Level I is faster than Level II, which is in turn faster than Level III. The robustness of the levels goes in the opposite direction: Level III is fully robust while Level I is the most error-prone.

How can this be achieved? We exploit the operator overloading capability of `C++`, of course. To see the `Core Library` solution, we first identify the *native number types* of each level: Level I inherits from `C++` the four machine number types: `int`, `long`, `float`, `double`. Level II has the usual number types found in Big Number packages: `BigInt`, `BigRat`, `BigFloat`. In `Core Library` we define a class `Real` that includes all the Level I and II number as subclasses. Level III has only one number type, called `Expr`. This number type is basically structured as a dag (directed acyclic graph) to maintain information about its defining expression. Both `Real` and `Expr` were originally introduced in the `Real/Expr` Package[56].

Let us define a *Level X program* (X=I, II, III) to be one that contains a Level X number type, but no number types at level greater than X. For instance, a *Level I Program* is synonymous with a "standard `C++` program". To allow such a Level I Program to access Level III accuracy, we introduce a *type promotion/demotion* mechanism. This mechanism is triggered by the "Compilation Level" (i.e., the `CORE_LEVEL` defined in the preamble above):

- At Level I Compilation, `BigInt` demotes to `long`, while `BigRat`, `BigFloat`, and `Expr` demotes to `double`.
- At Level II Compilation, `long` promotes to `BigInt`, `double` promotes to `BigFloat`, while `Expr` demotes to `BigFloat`.
- At Level III Compilation, `long`, `double`, `BigInt`, `BigRat` and `BigFloat` are all promoted to `Expr`.
- At Level IV Compilation, no promotion or demotion occurs.

Note that `int` and `float` remain at machine precision at all compilation levels. Hence every compilation level can access Level I variables in the form of `int` and `float`; these are useful for numerical quantities with low accuracy requirements (e.g., `int` variables for indexing arrays). Our approach has two major benefits. First, it reduces the effort necessary to convert existing libraries and application programs into fully robust programs. These programs are Level I programs, and we would like to make them fully robust just by recompiling them at Level III, say. Second, it does not automatically (a) force programmers to design new algorithms, nor (b) require the programmer to use new programming constructs.

Although our approach essentially allows the logic of a program to be left intact, some amount of adjustments may still be necessary for two reasons: (i) Issues of numerical input/output. This is inevitable because[53] numerical precision at different levels will lead to different input/output behavior. (ii) Efficiency issues. Level III computation can be extremely slow. We generally expect there to be opportunity for optimization. Innocuous decisions in Level I programs can be unnecessarily inefficient when run as a Level III program. A major challenge is to automatically detect such inefficiencies and to replace them by optimized constructs. Techniques similar to those in optimization compilers may ultimately be crucial[52]. For more details about such issues in Level III programming, see the Core Tutorial [53].

There is considerable research still to be done within the preceding framework. Nevertheless much has been achieved: first and foremost, non-robustness is no longer seen as the intractable problem of the early 1990's. After a decade of research, for a large class of computational problems, the speed of Level III programs can be brought down to within a factor of $3 - 10$ of a corresponding Level I computation on typical input data. This can be automatically achieved with general software tools, not hand-crafted code. Such results are deemed a suitable tradeoff between speed and robustness for many applications[56]. The critical technique here is the idea of *floating point filters*[9], originally pioneered by Fortune and van Wyk[21]. A

major direction in current efforts aim at extending the domain of successful practical applications to nonlinear domains[4].

3. Theory of Real Computation and Approximation

This section introduces a new approach to computing with real numbers. We will treat two aspects of guaranteed accuracy computation. This section, based on Turing computability, treats one aspect. Sections 6 and 7 will develop the algebraic and numerical models of computability.

In the approximation of real numbers, there are two basic decision[f] problems: deciding if a real number is zero and determining its sign. These issues are best approached from the view point of function evaluation. Let

$$f : \mathbb{R}^m \to \mathbb{R} \tag{4}$$

be a partial function and $\mathbf{x} \in \mathbb{R}^m$. If $f(\mathbf{x})$ is undefined, we write $f(\mathbf{x}) \uparrow$ and call $\mathbf{x}$ an *invalid input*. Otherwise, $f(\mathbf{x})$ is defined, written $f(\mathbf{x}) \downarrow$, then we say $\mathbf{x}$ is *valid*. We will associate five computational problems with f. But first we briefly review the computational model.

3.1. *Turing Computability*

The standard Turing model[43,54] of computation will be used in this section. We assume deterministic machines, and focus only on time complexity. In the Turing model, all objects must be represented as strings or words over some alphabet Σ. Let

$$g : \Sigma^* \to \Sigma^* \tag{5}$$

be a partial function. We recognize three notions of what it means to compute g:

- We say g is *conditionally computable* if there is a Turing machine M such that for all $w \in \Sigma^*$, if $g(w) \downarrow$ then M on input w will halt and output $g(w)$. We make no assumption about the behavior of M in case $g(w) \uparrow$. In particular, M may or may not halt.
- We say g is *unconditionally computable* if it is conditionally computable by a Turing machine M as before, but in case $g(w) \uparrow$, then

[f]In the theory of computation, decision problems have two possible outputs: 0/1 or yes/no or true/false. In geometric computation, it seems more natural to regard decision problems as any function with a finite range. E.g., most geometric predicates has three outputs: $-1/0/+1$ or in/on/out.

M must halt and enter a special state $q_\uparrow$. This state will never be entered on other inputs. Thus M can recognize invalid inputs and always halt.

- Finally, we say g is *partially computable* if it is conditionally computable by a Turing machine M as before, but in case $g(w) \uparrow$, then M must *loop* (i.e., not halt). Note that partial computability is the standard definition of what it means to compute a partial function in computability theory[46].

The three definitions coincide when g is a total function. We call g a *decision problem* if g is total and has finite range. In general, we have:

$$\begin{aligned} g \text{ is unconditionally computable} &\Longrightarrow g \text{ is partially computable} \\ &\Longrightarrow g \text{ is conditionally computable.} \end{aligned}$$

We will simply say "computability" for unconditional computability, as this will be the main concept we use. We can further introduce complexity considerations to the above, e.g., "polynomial-time unconditional computability" is just unconditional computability in which the Turing machine halts after a polynomial number of steps.

Representation and encodings. Let D be an algebraic domain with a partial function $\omega : D^m \to D$. In this paper, D will always be a subset of $\mathbb{C}$. We often call such a partial function an *operator*. To discuss computation over D we need to represent its elements as strings. A *representation* of D is any partial onto function $\rho : \Sigma^* \to D$. If $\rho(w) \uparrow$, then w is said to be *ill-formed*; otherwise it is *well-formed* and *represents* the element $\rho(w) \in D$. Since ρ is onto, every element in D has a representation. Relative to ρ, a partial function

$$f : (\Sigma^*)^m \to \Sigma^* \tag{6}$$

is an *implementation* of ω if (i) for all well-formed $w_1, \ldots, w_m$, if $\omega(\rho(w_1), \ldots, \rho(w_m)) \downarrow$ then

$$\rho(f(w_1, \ldots, w_m)) = \omega(\rho(w_1), \ldots, \rho(w_m)), \tag{7}$$

and (ii) if any w_i is ill-formed, or if $\omega(\rho(w_1), \ldots, \rho(w_m)) \uparrow$ then $f(w_1, \ldots, w_m) \uparrow$. Relative to ρ, we say ω is *(polynomial-time) computable* if it has an implementation (6) that is (polynomial-time) computable. Note that "polynomial-time" in computing $f(w_1, \ldots, w_m)$ is with respect to the *representation size*, $n = |w_1| + |w_2| + \cdots + |w_m|$.

Two basic decision problems arises with any representation ρ. The *parsing problem* wants to know from any given $w \in \Sigma^*$, whether $\rho(w)\downarrow$. The *isomorphism problem* wants to know, for any given $v, w \in \Sigma^*$, whether $\rho(v) = \rho(w)$. In our applications, both problems will be easily (polynomial-time) solvable.

3.2. *Representable Real Numbers*

Since $\mathbb{R}$ is uncountable, no representation of $\mathbb{R}$ is possible. In this paper, we propose to treat real computation through the following device: we postulate a set $\mathbb{F}$ called the *representable real numbers*. This set satisfies the following axioms.

- $(\mathbb{F}, +, -, \times, 0, 1)$ is a ring that extends the integer ring, $\mathbb{Z} \subseteq \mathbb{F}$.
- If $x \in \mathbb{F}$ then $x/2 \in \mathbb{F}$. Hence $\mathbb{F}$ is dense in the reals.
- $\mathbb{F}$ is countable and hence it has a representation, $\rho : \Sigma^* \to \mathbb{F}$ such that $\lg|\rho(w)| \leq |w|$ whenever w is well-formed. The parsing problem and the isomorphism problem for ρ are both polynomial-time decidable.
- Relative to ρ, there are polynomial-time implementations of the operations of $+, -, \times, div_2$ and the comparison of representable numbers. Here, $div_2(x)$ denote the function $x \mapsto x/2$.

This approach is natural and conforms fairly closely to numerical computation found in practice: typically, $\mathbb{F}$ is the set of floating point numbers in a fixed base $B \geq 2$. In practice, there may be limits on the precision in the floating point representation, but these will be removed for our purposes. The set of *base B floating point numbers* is given by $\{mB^e : m \in \mathbb{Z}, e \in \mathbb{Z}\}$. The standard representation of mB^e is given by a pair of binary integers (m, e). The *size* of mB^e is simply $1+\lceil\lg|m|\rceil+\lceil\lg|e|\rceil$. A possible alternative to floating point numbers is the choice $\mathbb{F} = \mathbb{Q}$, the rational numbers.

Once $\mathbb{F}$ and its representation $\rho : \mathbb{F} \to \Sigma^*$ are fixed, whenever we speak of "computing f", it is understood that our algorithms will be Turing machines that accepts an arbitrary string $w \in \Sigma^*$ as input. In particular, ill-formed inputs may be fed to our machine, but our axioms about ρ assures us that we can readily recognize these inputs (and enter the state $q_\uparrow$). But another situation arise: we are often interested in functions $f : R^m \to R$ where R is a proper subset of $\mathbb{F}$. In this case, the Turing machine for f must recognize inputs that do not represent elements of R. This is polynomial-time computable when $R = \mathbb{Z}$ or $R = \mathbb{N}$. This is shown in the following

fact:

Lemma 1: *The following total functions are polynomial-time computable:*
(i) $\lceil \lg |x| \rceil$ *and* $\lfloor \lg |x| \rfloor$
(ii) $\lceil x \rceil$ *and* $\lfloor x \rfloor$
(iii) The function $f : \mathbb{F} \to \{0, 1\}$ *where* $f(x) = 1$ *iff* $x \in \mathbb{Z}$.

Proof: Fix any $x \in \mathbb{F}$.
(i) We can determine in $O(\lg |x|)$ steps the smallest $k \in \mathbb{N}$ such that $2^k \geq |x|$. This k is $\lceil \lg |x| \rceil$. This is polynomial-time since every representation w of x satisfies $|w| \geq \lg |x|$ (see the axioms for $\mathbb{F}$). Similarly, we can compute $\lfloor \lg |x| \rfloor$ in polynomial time. Note that we can even do this in $O(\lg \lg |x|)$ time.
(ii) Using $\lceil \lg |x| \rceil$ from part (i), we can next compute the value $\lceil |x| \rceil$ in $O(\lg |x|)$ steps. The algorithm amounts to determining each bit in the binary representation of $\lceil |x| \rceil$. Similarly for $\lfloor |x| \rfloor$.
(iii) Using part (ii), we can compare x with $\lceil x \rceil$. Note that $x = \lceil x \rceil$ iff $x \in \mathbb{Z}$. □

It is interesting to see in this proof that (i) is the prerequisite to (ii). We will see this phenomenon again.

Decision problems associated with a function. Associated to the function (4), we have three natural decision problems:

- The *validity problem*, denoted $\texttt{VALID}(f)$, is to decide for any $x \in \mathbb{F}$, whether $f(x) \uparrow$. Recall that by our general conventions, a Turing machine for deciding validity actually accepts strings $w \in \Sigma^*$. If w is ill-formed, By assumption, we can detect whether w is ill-formed or not in polynomial-time. Assuming w is well-formed, our Turing machine must then decide whether $f(\rho(w)) \uparrow$. In the following discussion, we do not distinguish between an ill-formed w or a well-formed w such that $f(\rho(w)) \uparrow$. Both are simply considered invalid. Hence, the $\texttt{VALID}(f)$ problem has 2 possible outputs: invalid, valid.
- The *zero problem*, denoted $\texttt{ZERO}(f)$, is to decide for any $x \in \mathbb{F}$ whether $f(x) \downarrow$ and if so, whether $f(x) = 0$. This problem has 3 possible outputs: invalid, zero, non-zero.
- The *sign problem*, denoted $\texttt{SIGN}(f)$, is to determine for any $x \in \mathbb{F}$, whether $f(x)$ is valid and if so determine the sign of $f(x)$ (this is $0, \pm 1$). This problem has 4 possible outputs: invalid, zero, positive,

negative.

Although the sign problem is more important for practical applications, the zero problem is more general since it is meaningful in unordered domains such as $\mathbb{C}$. To investigate the decidability (i.e., computability) of these problems, the concept of reducibility is useful. We say f is *reducible* to g if there are total computable functions s, t such that for all $x \in \mathbb{F}$, $f(x) = s(g(t(x)))$. Also, f, g are *recursively equivalent* if f is reducible to g and vice-versa. It is immediate that if f is reducible to g and g is computable, then f is computable. So an uncomputable f is not reducible to a computable g.

Lemma 2: *For any f:*
(i) $\texttt{VALID}(f)$ *is reducible to* $\texttt{ZERO}(f)$*, but there is an* f_0 *such that* $\texttt{VALID}(f_0)$ *is decidable and* $\texttt{ZERO}(f_0)$ *is undecidable.*
(ii) $\texttt{ZERO}(f)$ *is reducible to* $\texttt{SIGN}(f)$*, but there is an* f_1 *such that* $\texttt{ZERO}(f_1)$ *is decidable and* $\texttt{SIGN}(f_1)$ *is undecidable.*

Proof: The reducibility of $\texttt{VALID}(f)$ to $\texttt{ZERO}(f)$, and $\texttt{ZERO}(f)$ to $\texttt{SIGN}(f)$ is immediate from the definition. To see f_0, we just define $f_0 : \mathbb{N} \to \{0, 1\}$ such that $f_0(i) = 1$ iff the ith Turing machine on i halts. The function f_1 is a simple variant, $f_1 : \mathbb{N} \to \{-1, 1\}$ such that $f_1(i) = 1$ iff the ith Turing machine on i halts. □

For many problems (in particular, the evaluation problems in Section 5), $\texttt{VALID}(f)$ and $\texttt{ZERO}(f)$ are basically the same problem. On the other hand, there is a potentially exponential gap between $\texttt{ZERO}(f)$ and $\texttt{SIGN}(f)$ in the well-known problem of *sum of square-roots*. More precisely, define the function S that, on any input sequence of integers $a_1, \ldots, a_n$, computes the sum

$$S(a_1, \ldots, a_n) = \sum_{i=1}^{n} \texttt{sign}(a_i)\sqrt{|a_i|}.$$

This is the famous sum of square-roots problem. An observation of Yap[5] is that $\texttt{ZERO}(S)$ is polynomial-time. On the other hand, the best current algorithms for $\texttt{SIGN}(S)$ require exponential time. Another important problem where there seems to be a complexity gap is $\texttt{ZERO}(det)$ and $\texttt{SIGN}(det)$ where *det* is the problem of computing the determinant of a square integer matrix. Since *det* can be solved in $O_L(n^3 M(n \lg n + L))$ time for n square matrices

with L-bit entries and $M(L)$ is the complexity of L-bit integer multiplication, this gap (if it exists) is at most a factor of n^2 (ignoring logarithmic terms).

Exact and approximate computability. The function f in (4) is said to be *exactly computable* if (i) $f(\mathbf{x}) \in \mathbb{F}$ for all valid $\mathbf{x} \in \mathbb{F}^m$, and (ii) f is unconditionally computable. For instance, if f is a ring operation $(+, -, \times)$ or div_2, then f is exactly computable, by our assumptions about $\mathbb{F}$. But when f does not satisfy (i), we next introduce weaker notions of computing f, based on approximation. Indeed, even when f satisfies (i), we may still want to compute it approximately.

A partial function

$$\widetilde{f} : \mathbb{F}^m \times \mathbb{Z} \to \mathbb{F} \tag{8}$$

is an *absolute approximation* function of f if for all $\mathbf{x} \in \mathbb{F}^m, a \in \mathbb{Z}$, we have $\widetilde{f}(\mathbf{x}, a) \in f(\mathbf{x})[a]$. By definition, this means $\widetilde{f}(\mathbf{x}, a)$ is undefined iff $f(\mathbf{x})$ is undefined. Similarly a partial function

$$\widetilde{f} : \mathbb{F}^m \times \mathbb{N} \to \mathbb{F} \tag{9}$$

is a *relative approximation* of f if for all $\mathbf{x} \in \mathbb{F}^m, r \in \mathbb{N}$, we have $\widetilde{f}(\mathbf{x}, r) \in f(\mathbf{x})\langle r \rangle$. Again, $\widetilde{f}(\mathbf{x}, r) \uparrow$ iff $\widetilde{f}(\mathbf{x}) \uparrow$.

NOTATION: we will add a "colon flourish" and write "$f(x : a)$" to denote an absolute approximation $\widetilde{f}(x, a)$. Similarly, we add a "semicolon flourish" and write "$f(x; r)$" to denote a relative approximation $\widetilde{f}(x, r)$.

The output of approximation functions (as in (8) and (9)) are restricted to $\mathbb{F}$. We say f is *absolutely approximable* if it has an approximation function (8) that is unconditionally computable. Similarly, f is *relatively approximable* if it has an approximation function (9) that is unconditionally computable. We also say f has *guaranteed accuracy* if it is relatively approximable. It follows from these definitions that if f is absolutely or relatively approximable, then $\texttt{VALID}(f)$ is decidable.

3.3. *Basic Relations*

The next theorem shows that guaranteed relative precision is a generalization of guaranteed sign computation.

Theorem 3: For all $x \in \mathbb{F}$, we have $\texttt{sign}(f(x)) = \texttt{sign}(f(x; 1))$.

Proof: We have

$$|f(x) - f(x; 1)| \leq |f(x)|/2. \tag{10}$$

If $f(x) = 0$ then $f(x;1)$ must also be 0. Conversely, if $f(x;1) = 0$ then $f(x)$ is also 0. Hence, assume $f(x)f(x;1) \neq 0$. If $f(x)f(x;1) > 0$, the result is also true. It remains to consider the case $f(x)f(x;1) < 0$. In this case, we have

$$|f(x) - f(x;1)| = |f(x)| + |f(x;1)| \geq |f(x)|. \tag{11}$$

But (10) and (11) implies $f(x) = 0$, contradicting $f(x)f(x;1) < 0$. □

Corollary 4: *The problem* $\texttt{SIGN}(f)$ *is reducible to the relative approximability of* f.

Theorem 5: The following are equivalent:
(i) The function f is relatively approximable.
(ii) The function f is absolutely approximable and $\texttt{ZERO}(f)$ is decidable.

Proof: In the first direction, assume f is relatively approximable. By the previous lemma, $ZERO(f)$ is decidable. So it is sufficient to show how to approximate f absolutely. First compute $x' = f(x;1)$ that approximates x to one relative bit. Thus $|x'| \geq |f(x)|/2$. Using Lemma 1, we compute $r = a + 1 + \lceil \lg |x'| \rceil$. Finally compute $x'' = f(x;r)$. We have $|x'' - f(x)| \leq 2^{-r}|f(x)|$, i.e.,

$$\lg |x'' - f(x)| \leq -r + \lg |f(x)| \leq -r + 1 + \lg |x'| = -a.$$

Hence we can output x'' as $f(x:a)$.

In the other direction, suppose f is absolutely approximable and $\texttt{ZERO}(f)$ is decidable. To compute $f(x;r)$, we first check if $f(x) = 0$, and if so, we output 0. Otherwise we perform the following code:

$a \leftarrow 1$;
While $|f(x:a)| < 2^{-a+1}$
Do $a \leftarrow a + 1$.

Since $f(x) \neq 0$, this while-loop will terminate. Upon loop termination, $|f(x:a)| \geq 2^{-a+1}$. Since $|f(x)| + 2^{-a} \geq |f(x:a)|$, we deduce that $|f(x)| \geq 2^{-a+1} - 2^{-a} = 2^{-a}$. If we choose $a' = r + a$, then $|f(x:a') - f(x)| \leq 2^{-a'} = 2^{-r-a} \leq 2^{-r}|f(x)|$. Thus $f(x:a')$ approximates $f(x)$ with r relative bits of precision. □

Theorem 5 suggests that absolute precision may be a weaker concept than relative precision. The next result confirms this.

Theorem 6: There is a function $f_K : \mathbb{F} \to \mathbb{F}$ that is absolutely approximable in polynomial time but f is not relatively approximable.

Proof: Let $M_0, M_1, M_2, \ldots$ be a standard enumeration of Turing machines restricted to binary input strings. By introducing a bijection between binary strings and $\mathbb{N}$ (e.g., the dyadic notation), we can view the input set for each M_i to be $\mathbb{N}$. Let $K : \mathbb{N} \to \{0, 1\}$ be the diagonal function where $K(i) = 0$ if M_i does not halt on input i and $K(i) = 1$ if otherwise. It is well-known that K is not computable. Consider the function $f_K : \mathbb{N} \to \mathbb{F}$ defined as follows:

$$f_K(i) = \begin{cases} 0 & \text{if } M_i \text{ on input } i \text{ does not halt,} \\ 2^{-k} & \text{if } M_i \text{ on input } i \text{ halts in exactly } k \text{ steps} \end{cases}$$

The theorem follows from two facts:
(a) f_K is not relatively approximable. For $i \in \mathbb{N}$, clearly $f_K(i; 1) = 0$ iff $K(i) = 0$. Hence if f_K were relatively approximable, then K would be computable, contradiction.
(b) f_K is absolutely approximable. It is sufficient to show how, given $(i, j) \in \mathbb{N} \times \mathbb{Z}$, we may compute an absolute approximation $f_K(i : j)$. If $j \leq 0$, we can just output 1. Hence assume $j > 0$. We first simulate M_i on i for j steps. If M_i halts in some $k \leq j$ steps, then we output 2^{-k}. Otherwise, we output 2^{-j}.

We show that this algorithm is correct. Consider two possibilities: (i) Suppose $f_K(i) = 0$. In case $j \leq 0$, then $|f_K(i) - f_K(i : j)| = 1 \leq 2^{-j}$. So assume $j > 0$. Since M_i on i does not halt, we will output 2^{-j} as the value of $f_K(i : j)$. This output is correct since $|f_K(i) - f_K(i : j)| = |f_K(i : j)| = 2^{-j}$. (ii) Suppose $f_K(i) \neq 0$. Assume M_i on i halts in $k \geq 0$ steps. In case $j \leq 0$, then $|f_K(i) - f_K(i : j)| = |1 - 2^{-k}| \leq 2^{-j}$. Otherwise, we will output 2^{-m} as the value of $f_K(i : j)$, where $m = \min\{j, k\}$. This output is correct since $|f_K(i : j) - f_K(i)| = |2^{-m} - 2^{-k}| < 2^{-m} \leq 2^{-j}$.

Finally, is this algorithm polynomial time? To simulate M_i on i for j steps takes $O(\log(i)j)$ time. Since the input size is $\Theta(\log(i) + \log(j))$, this is exponential time. To fix this, we can modify the function f_K so that instead of $f_K(i) = 2^{-k}$, we have $f_K(i) = 2^{\mathtt{msb}(k)}$. where $\mathtt{msb}(k) = \lfloor \lg |k| \rfloor$. But $\mathtt{msb}(k)$, and hence $2^{\mathtt{msb}(k)}$, can be computed in polynomial time, by Lemma 1. □

In the next section, we will address the problems of zero determination and sign determination, using more efficient and practical algorithms than those implied by the above generic proofs.

4. Guaranteed Accuracy for Basic Operators

The previous section gives an abstract treatment of the approximability of partial functions $f : \mathbb{R}^m \to \mathbb{R}$. This section examines in the main operators in practice: rational operators $(\pm, \times, \div)$, square root $(\sqrt{\cdot})$, exponential and logarithm operators $(\exp(\cdot), \ln(\cdot))$. We will assume the availability of algorithms that can implement these operations to any specified accuracy. Such algorithms may be found in Brent [8] (see Du et al [19] for hypergeometric functions). Our main concern is how to propagate precision bounds.

Such algorithm were first given in detail and analyzed by Ouchi[42] for the rational operators and square root. These were implemented in the `Real/Expr` package[56], and incorporated into the original `Core Library`[30]. The algorithms were based on propagating composite precision bounds. What is new in this section is to revisit these questions, but where we propagate either absolute or relative precision bounds, but not both. This is simpler and more intuitive.

In the following, whenever we guarantee "k relative bits", it is implicit that $k \geq 0$. But when we guarantee "k absolute bits", k may be negative.

The role of the most significant bit position. The proofs of Lemma 1 (and Theorem 5) indicate the usefulness of approximating $\lg|x|$. Another use is for transforming any precision bound, from an absolute bound to a relative one or vice-versa. To facilitate such transformations, we use the function, $\mu(x) := \lg|x|$. In implementations, we prefer to work with the related *msb function*, defined by $\mathtt{msb}(x) = \lfloor \lg|x| \rfloor$. By definition, $\mu(0) = \mathtt{msb}(0) = -\infty$. By Lemma 1, the function $\mathtt{msb}(x)$ is computable. Thus

$$2^{\mathtt{msb}(x)} \leq |x| < 2^{1+\mathtt{msb}(x)}.$$

If the binary notation for x is the $\cdots b_2 b_1 b_0 . b_{-1} b_{-2} \cdots$ $(b_i = 0, 1)$ then $\mathtt{msb}(x) = t$ iff $b_t = 1$ and for all $i > t$, $b_i = 0$. When x is a general expression, it may be difficult[g] to determine $\mathtt{msb}(x)$ exactly: let $\mu^+(x)$ and $\mu^-(x)$ denote any upper and lower bound on $\mu(x)$,

$$\mu^-(x) \leq \mu(x) \leq \mu^+(x).$$

Here, $\mu^+(x), \mu^-(x)$ are not functional notations, as the actual values of $\mu^+(x), \mu^-(x)$ will depend on the context. The choice $\mu^-(x) = -\infty$ and

[g]This remark does not contradict Lemma 1, which assumes x is explicitly given as an element of $\mathbb{F}$.

$\mu^+(x) = \infty$ are trivial bounds. Non-trivial bounds on $\mu(x)$ may not be hard to obtain; usually, $\mu^-(x)$ is harder than $\mu^+(x)$.

Lemma 7: *Let $x \in \mathbb{R}$ and $a, r \in \mathbb{R}$.*
(i) $x[a] \supseteq x\langle a + \mu^+(x)\rangle$.
(ii) $x\langle r\rangle \supseteq x[r - \mu^-(x)]$.
(iii) $x[a, r] \supseteq x\langle \min\{r, a + \mu^+(x)\}\rangle$.
(iv) $x[a, r] \supseteq x[\min\{a, r - \mu^-(x)\}]$.

This lemma is just another way of writing the following inequalities:
(i,iii) $2^{-a} \geq |x|2^{-a-\mu^+(x)}$,
(ii,iv) $|x|2^{-r} \geq 2^{-r+\mu^-(x)}$.
The four cases in this lemma should be viewed as rules for converting precision bounds. Thus, (i) says that if we want to guarantee a absolute bits in x, it is enough to guarantee $a + \mu^+(x)$ relative bits in x. Since $\mu^-(x)$ is generally harder to come by than $\mu^+(x)$, it is preferable to assume absolute bounds at the start to the propagation, and to convert such bounds into relative bounds as needed. In short, rules (i) and (iii) are generally preferable over the rules (ii) and (iv).

Guaranteeing 5 bits in multiplication. To understand the difference between relative and absolute precision, consider how to guarantee that a value x has 5 relative bits precision. Assume $x = y \cdot z$. Suppose we wish to compute $\widetilde{x} = x(1 + \rho_x)$ as an approximation to x. Moreover, we want to compute $\widetilde{x}$ as the product $\widetilde{y}\widetilde{z}$ where $\widetilde{y} = y(1 + \rho_y)$ and $\widetilde{z} = z(1 + \rho_z)$ are approximations to y, z. This gives

$$x(1 + \rho_x) = yz(1 + \rho_y)(1 + \rho_z) = yz(1 + \rho_y + \rho_z + \rho_y\rho_z).$$

Ignoring the second order term $\rho_y\rho_z$, we conclude that $\rho_x = \rho_y + \rho_z$. Thus, if $|\rho_y|$ and $|\rho_z|$ is at most 2^{-6}, then $|\rho_x| \leq 2^{-5}$. In other words, we only need to guarantee 6 relative bits in y and z, respectively. If we wish to take the second order effects into account, it is sufficient to guarantee an extra bit in either y or z.

Next, consider how to guarantee 5 absolute bits in $x = yz$. Now we need upper bounds on the sizes of y and z. Let us write $\widetilde{y} = y + \delta_y$, $\widetilde{z} = z + \delta_z$, and

$$x + \delta_x = (y + \delta_y)(z + \delta_z) \tag{12}$$
$$= yz + y\delta_z + z\delta_y + \delta_y\delta_z. \tag{13}$$

Ignoring the second order term again, we have $\delta_x = y\delta_z + z\delta_y$. Hence, if

$|\delta_y| \leq 2^{-6-\mu^+(z)}$ and $|\delta_z| \leq 2^{-6-\mu^+(y)}$, then we would have $|\delta_x| \leq 2^{-6} + 2^{-6} = 2^{-5}$. Thus it is sufficient to guarantee $6 + \mu^+(z)$ absolute bits for y, and $6 + \mu^+(y)$ absolute bits for z. We now account for the omission of the second order term: First, if $\mu(x) \geq -5$, then $|\delta_z \delta_y| \leq 2^{-12-\mu^+(x)} \leq 2^{-7}$ and so it is enough to guarantee an extra bit in either y or z. But what if $\mu(x) < -5$? Choose a_y and a_z such that $a_y + a_z = 7$ (for instance $a_y = 3, a_z = 4$). Then it suffices to to require $\max\{a_y, 7 + \mu^+(z)\}$ and $\max\{a_z, 6 + \mu^+(y)\}$ absolute bits from y and z (respectively). Then $|y\delta_z| + |z\delta_y| \leq 2^{-6} + 2^{-7}$, as before. Moreover, $|\delta_y \delta_z| \leq 2^{-a_y - a_z} \leq 2^{-7}$, and hence $|\delta_x| \leq 2^{-5}$, as desired.

We may represent the flow of information in the guaranteed precision multiplication operator as in figure 1. This is typical of the other operators as well. Basically, in computing an approximate value for x, we see a downward flow of precision bounds $[a, r]$, and an upward flow of approximation values $\widetilde{x}$. In general, we will need to iterate this downward-upward cycle of computation.

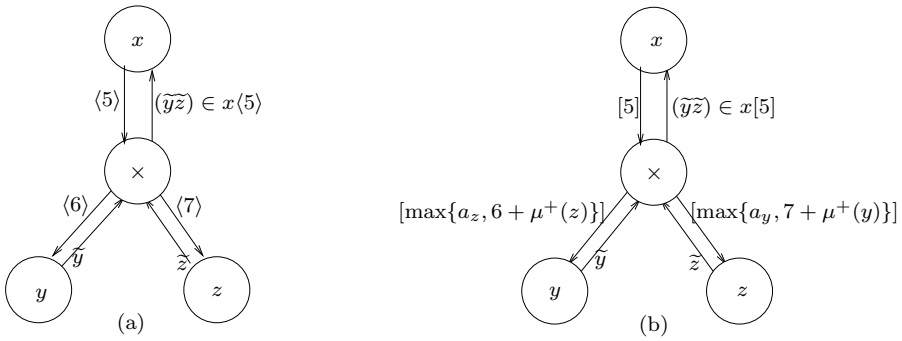

Fig. 1. Propagation Rules: (a) Relative Precision Multiplication (b) Absolute Precision Multiplication

The above analysis is completely general: to guarantee k bits in x, we just replace the constants "$5, 6, 7$" in the preceding arguments by "$k, k+1, k+2$" (respectively). Thus we have proved:

Lemma 8: *Let $x = yz$. We want to compute an approximate value $\widetilde{x}$ as $\widetilde{y}\widetilde{z}$ where $\widetilde{y}, \widetilde{z}$ are approximate values for y, z. Assume that we can multiply approximate values without error.*
(i) To guarantee k relative bits in x, it is sufficient to guarantee $k+1$ bits in y and $k+2$ bits in z.

(ii) Let $a_y + a_z = k + 2$. To guarantee k absolute bits in x, it is sufficient to guarantee $\max\{a_y, k + 1 + \mu^+(z)\}$ *absolute bits for y and* $\max\{a_z, k + 1 + \mu^+(y)\}$ *absolute bits for z.*

In the absence of other information, we propagate precision bounds to y and z symmetrically. When there is asymmetry in our treatment of y and z, it is clear that we reverse the roles of y and z. The optimal allocation of bits to y and z is an interesting problem that we will not treat in the present paper. This calls for a sensitivity analysis of the underlying expression and a reasonable cost model. In the example of multiplication, absolute precision is harder to guarantee than relative precision. We next see that the reverse is true for addition, but in a more profound way.

Can we guarantee relative precision in addition? It is trivial to guarantee absolute precision in addition. For instance, to guarantee 5 absolute bits of precision for $x = y + z$, it is enough to guarantee 6 absolute bits for y and for z. Then we have $|\delta x| \leq |\delta y| + |\delta z| \leq 2^{-6} + 2^{-6} = 2^{-5}$.

What about relative bits? If we guarantee 5 relative bits from y and z, then $x(1 + \rho_x) = y(1 + \rho_y) + z(1 + \rho_z)$ and so

$$|x\rho_x| = |y\rho_y + z\rho_z| \leq |y|2^{-5} + |z|2^{-5}.$$

In case $yz \geq 0$, then the last expression is equal to $|y + z|2^{-5} = |x|2^{-5}$, as desired. But if $yz < 0$ then we get nothing of the sort when catastrophic cancellation takes place. Indeed, when $y = -z$ then $x = 0$ and it is impossible to write a bound of the form $|x\rho_x| \leq |x|C$ for any finite value of C. This is the first indication that guaranteeing the relative precision of addition can be nontrivial.

But suppose we have some lower bound on $|x|$, say, in the form $\mu^-(x)$. Let $k' = 1+k-\mu^-(x)$. Then we can compute $\widetilde{y} = y(1+\rho_y)$ and $\widetilde{z} = z(1+\rho_z)$ such that $\lg|\rho_y| \leq -k'$ and $\lg|\rho_z| \leq -k'$. Then

$$|\widetilde{x} - x| = |y\rho_y + z\rho_z| \leq 2^{-k+\mu^-(x)} \leq |x|2^{-k},$$

thus ensuring k relative bits of precision. Note that in case $x = 0$, then $\mu^-(x) = -\infty$. In summary,

Lemma 9: *Let $x = y + z$.*
(i) To guarantee k absolute bits in x, it suffices to guaranteed $k + 1$ bits in y and z.
(ii) To guaranteed k-relative bits in x, it suffices to guarantee $k+1-\mu^-(x)$ relative bits in y and z.

Note that part (ii) cannot be used in a recursive evaluation method since what we need is an estimate of $\mu(x)$, not an estimate of $\mu(y)$ or $\mu(z)$. We shall see that $\mu^-(x)$ can be replaced by a weaker concept, by a *conditional lower bound* $\beta(x)$. Such a lower bound has the property that, in case $x \neq 0$, then $\beta(x) \leq \mu(x)$. We return to this issue in the next section.

Guaranteeing division. Next consider the problem of guaranteeing k relative bits in the division $x = y/z$, assuming $z \neq 0$. Let $\widetilde{y} = y(1+\rho_y)$ and $\widetilde{z} = z(1+\rho_z)$. A new phenomenon arises: the division, $\widetilde{y}/\widetilde{z}$, cannot be computed without error for number representations such as Big Floats. For addition and multiplication, we had assumed that $\widetilde{y}+\widetilde{z}$ and $\widetilde{y}\widetilde{z}$ can be computed exactly (this agrees with our axioms for the representable reals $\mathbb{F}$). For division, we need to specify a precision bound for this operation: assume a relative error of $\rho_\div$. Then we may write

$$\begin{aligned} x(1+\rho_x) &= \widetilde{x} \\ &= \frac{\widetilde{y}}{\widetilde{z}}(1+\rho_\div) \\ &= \frac{y(1+\rho_y)}{z(1-\rho_z)}(1+\rho_\div) \\ &= x(1+\rho_y+\rho_\div+\rho_y\rho_\div)\left(1+\rho_z+\rho_z^2\frac{1}{1-\rho_z}\right). \end{aligned}$$

To first order terms, $\rho_x = \rho_y + \rho_\div + \rho_z$. Therefore, if $|\rho_y|$, $|\rho_\div|$, $|\rho_z|$ were at most 2^{-k-2}, then we have $|\rho_x| \leq 2^{-k}$. The neglected nonlinear terms are in

$$D = \rho_z(\rho_y+\rho_\div) + \frac{\rho_z^2}{1-\rho_z} + \frac{\rho_y\rho_\div}{1-\rho_z} + (\rho_y+\rho_\div)\frac{\rho_z^2}{1-\rho_z}.$$

Assuming $\max\{|\rho_y|, |\rho_z|, |\rho_\div|\} \leq 2^{-k-2}$, we have

$$|D| \leq 2\cdot 2^{-2k-4} + 2\cdot 2^{-2k-4} + 2\cdot 2^{-2k-4} + 4\cdot 2^{2k-4} \leq 5\cdot 2^{-2k-3}.$$

The total absolute error is therefore at most $3\cdot 2^{-k-2} + 5\cdot 2^{-2k-3} \leq 2^{-k}$ (assuming $k \geq 2$). This proves:

Lemma 10: *Let $x = y/z$ and $z \neq 0$. We want to compute the approximate value $\widetilde{x}$ as $\widetilde{y}/\widetilde{z}$ where $\widetilde{y}, \widetilde{z}$ are approximate values for y, z Assume that we can divide approximate values with relative error $\rho_\div$. To guarantee $k \geq 2$ relative bits in x, it is sufficient to guarantee $k+2$ bits in both y and z, and to perform division with relative precision $k+2$.*

Next consider the propagation of absolute precision: instead of writing $\widetilde{y} = y + \delta_y$, we write $\widetilde{y} = y(1 + \delta'_y)$ where $\delta'_y = \delta_y/y$. Similarly, write $\widetilde{z} = z(1 - \delta'_z)$ where $\delta'_z = \delta_z/z$. Then

$$\begin{aligned} x + \delta_x &= \widetilde{x} \\ &= \frac{\widetilde{y}}{\widetilde{z}} + \delta_{\div} \\ &= \frac{y(1 + \delta'_y)}{z(1 - \delta'_z)} + \delta_{\div}. \end{aligned}$$

Assuming $|\delta'_z| \leq 1/2$, we see that $(1 - \delta'_z)^{-1} = 1 + C\delta'_z$ for some $|C| \leq 2$ or

$$\delta_x = x(\delta'_y + C\delta'_z + C\delta'_y\delta'_z) = \frac{\delta_y}{z} + \frac{C\delta_z y}{z^2} + \frac{C\delta_y\delta'_z}{z} = \frac{C'\delta_y}{z} + \frac{Cy\delta_z}{z^2}$$

for some $|C'| = 1 + |C\delta'_z| \leq 2$. Thus we have:

Lemma 11: *With the notations of Lemma 10, to guarantee k absolute bits from $x = y/z$, it is sufficient to guarantee $k+1$ absolute bits in the division operation, and to guarantee k_y and k_z absolute bits from y and z, where*

$$k_y \geq k + 2 - \mu^-(z), \qquad k_z \geq \max\{1 - \mu^-(z), k + 2 - 2\mu^-(z) + \mu^+(y)\}.$$

If $\mu(z) = -\infty$ (i.e., $z = 0$) the operation is invalid. But even when the operation is valid, we cannot propagate absolute precision bounds without knowing effective lower bounds on $\mu(z)$ or upper bounds on $\mu(y)$.

Guaranteeing square roots. Let $x = \sqrt{y}$, assuming $y > 0$. As in division, computing the square root of an approximate value is generally inexact, and we assume that the relative error is $\rho_{\surd}$. Hence, if $\widetilde{x} = x(1 + \rho_x)$ and $\widetilde{y} = y(1 + \rho_y)$, then we have

$$\begin{aligned} x(1 + \rho_x) &= \sqrt{\widetilde{y}}(1 + \rho_{\surd}) \\ &= \sqrt{y(1 + \rho_y)}(1 + \rho_{\surd}) \\ &= \sqrt{y}(1 + \rho_y)^{1/2}(1 + \rho_{\surd}). \\ (1 + \rho)^{1/2} &= 1 + \frac{\rho}{2} + \frac{(1/2)(-1/2)}{2!}\rho^2 + \frac{(1/2)(-1/2)(-3/2)}{3!}\rho^3 + \cdots \\ &= 1 + \sum_{k\geq 1}(-1)^{k+1}\frac{(2k-3)!!}{2^k k!}\rho^k \\ &= 1 + \frac{\rho}{2} - \sum_{k\geq 1}\rho^{2k}\frac{(4k-3)!!}{4^k(2k)!}\left(1 - \rho\frac{4k-1}{4k+2}\right). \end{aligned}$$

Here $n!!$ is the double factorial given by the recursive formula $n!! = (n-2)!! \cdot n$ when $n \geq 1$, with base case $(-1)!! = 0!! = 1$. To first order, we see that $x(1+\rho_x) = \sqrt{y}(1 + \frac{1}{2}\rho_y + \rho_{\surd})$ or $\rho_x = \frac{1}{2}\rho_y + \rho_{\surd}$. To bound the nonlinear terms, let us simply write ρ for ρ_y. Also let $(1+\rho)^{1/2} = 1 + \frac{1}{2}\rho + K\rho^2$ for some K. We will further assume $\max\{|\rho|, |\rho_{\surd}|\} \leq 1/2$. The following shows that $|K| < 5/24$:

$$\begin{aligned}\left|(1+\rho)^{1/2} - (1 + \frac{1}{2}\rho)\right| &< \sum_{k\geq 1} \rho^{2k} \frac{1}{8}\left(1 + \frac{1}{4}\right) \\ &= \frac{\rho^2}{1-\rho^2}\frac{5}{32} \\ &\leq \frac{5\rho^2}{24}.\end{aligned}$$

The neglected nonlinear terms in $x(1+\rho_x) = \sqrt{y}(1 + \frac{1}{2}\rho + K\rho^2)(1+\rho_{\surd})$ is bounded by

$$\left|K\rho^2 + \frac{1}{2}\rho\rho_{\surd} + K\rho^2\rho_{\surd}\right| < \rho\left(\frac{|K|}{2} + \frac{1}{4} + \frac{|K|}{4}\right) < \rho/6.$$

Summarizing,

$$|\rho_x| < \frac{1}{2}|\rho_y| + |\rho_{\surd}| + \frac{1}{6}|\rho_y|. \tag{14}$$

We can similarly guarantee absolute precision by propagating absolute precision bounds. Writing δ'_y for $\delta_y/\sqrt{y}$, we have:

$$\begin{aligned}x + \delta_x &= \sqrt{y + \delta_y} + \delta_{\surd} \\ &= \sqrt{y}\sqrt{1 + \delta'_y} + \delta_{\surd} \\ &= \sqrt{y}\left(1 + \frac{1}{2}\delta'_y + K\delta'^2_y\right) + \delta_{\surd}.\end{aligned}$$

To first order, we have $\delta_x = \sqrt{y}(\delta'_y/2) + \delta_{\surd} = \frac{1}{2}\delta_y + \delta_{\surd}$. The neglected term is $\sqrt{y}K\delta'^2_y = \delta_y(K\delta'_y)$. Assuming, $|\delta'_y| \leq 1/2$, we get the bound $|K\delta'_y| < 5|\delta'_y|/24 < 5/48$. Summarizing,

$$|\delta_x| \leq |\delta_y|\left(\frac{1}{2} + \frac{5}{48}\right) + |\delta_{\surd}|. \tag{15}$$

Lemma 12: *Let $x = \sqrt{y}$.*
(i) To guarantee $k \geq 0$ relative bits in $\widetilde{x}$, it suffices to ensure $k+1$ relative bits in $\widetilde{y}$ and $k+1$ relative bits in the approximate square root extraction.
(ii) To guarantee k absolute bits in $\widetilde{x}$, it suffices to ensure $k_y = \max\{k +$

$1, 1-\mu^-(y)/2\}$ *absolute bits in* $\widetilde{y}$ *and* $k+1$ *absolute bits in the approximate square root extraction.*

Proof:
(i) If $|\rho_y| \leq 2^{-k-1}$ and $|\rho_{\surd}| \leq 2^{-k-1}$ then we conclude from (14) that $|\rho_x| \leq 2^{-k}$. Note that requirement $\max\{|\rho_y|, |\rho_{\surd}|\} \leq 1/2$ is satisfied since $k \geq 0$.
(ii) If $|\delta_y| \leq 2^{-k_y} \leq 2^{-k-1}$ and $|\delta_{\surd}| \leq 2^{-k-1}$ then we conclude from (15) that $|\delta_x| \leq 2^{-k}$. But our derivation also require $|\delta'_y| \leq 1/2$ or $|\delta_y| \leq \sqrt{y}/2$. This follows if we ensure that $|\delta_y| \leq 2^{-k_y} \leq 2^{-1+(\mu^-(y))/2}$. □

Like propagating absolute precision for division, propagating relative precision for square roots requires a lower bound on y.

Exponential function. Let $x = \exp(y)$. Suppose we want to guarantee k absolute bits in the approximate value $\widetilde{x}$.

Lemma 13: *Let* $k_y \geq \max\{1, k+2+2^{\mu^+(y)+1}\}$. *If* $\widetilde{y}$ *has* k_y *absolute bits of precision and* $\widetilde{x} = \exp(\widetilde{y}, k+1)$ *(i.e.,* $\exp(\widetilde{y})$ *is computed to* $k+1$ *absolute bits) then* $\widetilde{x}$ *will have* k *absolute bits of precision.*

Proof: Now $\widetilde{y} = y + \delta_y$ where $|\delta_y| \leq 2^{-k_y}$. It is sufficient to show that $|x - \exp(\widetilde{y})| \leq 2^{-k-1}$. The lemma now follows from:

$$\begin{aligned}
|\exp(\widetilde{y}) - x| &= |\exp(y+\delta_y) - \exp(y)| \\
&= \exp(y)\,|\exp(\delta_y) - 1| \\
&< \exp(y)\,|2\delta_y|, \qquad (|\delta_y| \leq 1/2) \\
&\leq \exp(y) 2^{-k_y+1} \\
&\leq \exp(y) 2^{-k-1-2^{\mu^+(y)+1}} \\
&\leq 2^{-k-1}.
\end{aligned}$$

□

Next, suppose we want to compute $x = \exp(y)$ to $k \geq 0$ relative bits. Let $\widetilde{y} = y(1+\rho_y)$ where $|\rho_y| \leq 2^{-k_y}$ for some k_y, and assume that an approximate $\widetilde{x}$ is $\exp(\widetilde{y})$ computed to k_e relative bits.

Lemma 14: *If* $k_y \geq k + 2 + \mu^+(y)$ *and* $k_e \geq k+2$, *then* $\widetilde{x}$ *has at least* k *relative bits.*

Proof: We have

$$\begin{aligned}
\widetilde{x} &= \exp(\widetilde{y})(1+\rho_e), \qquad (\lg|\rho_e| \leq -k_e) \\
&= \exp(y)\exp(y\rho_y)(1+\rho_e), \qquad (\lg|\rho_y| \leq -k_y) \\
|x - \widetilde{x}| &= \exp(y) \cdot |1 - \exp(y\rho_y)(1+\rho_e)|
\end{aligned}$$

So we need

$$|1 - \exp(y\rho_y)(1 + \rho_e)| \leq 2^{-k}$$

or

$$1 - 2^{-k} \leq \exp(y\rho_y)(1 + \rho_e) \leq 1 + 2^{-k} \tag{16}$$

Since $|\rho_e| \leq 2^{-k-2}$, (16) will be satisfied if $\exp(y\rho_y) = (1 + \rho')$ with $|\rho'| \leq 2^{-k-1}$. Note that $|y\rho_y| < 1/2$ because $|y\rho_y| \leq |y|2^{-k_y} \leq 2^{-2-k}$. From the fact that $|\exp(z) - (1 + z)| \leq |z|$ for $|z| \leq 1/2$, we get $\exp(z) = 1 + \rho'$ with $|\rho'| \leq 2|z|$. Hence $\exp(y\rho_y) = 1 + \rho'$ with

$$|\rho'| \leq 2|y\rho_y| \leq 2|y|2^{-2-k-\mu^+(y)} \leq 2^{-k-1}$$

as desired. □

Logarithm function. Let $x = \ln(y)$. This is only defined when $y > 0$, which we will assume. First consider the problem of guarantee k absolute bits in $\widetilde{x}$ as an approximation of x.

Lemma 15: *Let $k_y \geq \max\{1 - \mu^-(y), k + 2 - \mu^-(y)\}$. If $\widetilde{y}$ has k_y absolute bits as an approximation of y, and $\widetilde{x} = \ln(\widetilde{y}, k+1)$ (i.e, $\widetilde{x}$ is $\ln(\widetilde{y})$ computed to $k + 1$ absolute bits), then $\widetilde{x}$ has k absolute bits of precision.*

Proof: It is enough to show that $|\ln(\widetilde{y}) - \ln(y)| \leq 2^{-k-1}$. Let $\widetilde{y} = y + \delta_y$, $\lg|\delta_y| \leq -k_y$. Since $k_y \geq 1 - \mu^-(y)$, we have $|\delta_y|/y \leq 1/2$. Then

$$\begin{aligned}
|\ln(y + \delta_y) - \ln(y)| &= |\ln(y(1 + \delta'_y)) - \ln(y)|, \ (\delta'_y = \delta_y/y) \\
&= |\ln(1 + \delta'_y)| \\
&\leq 2|\delta'_y|, \qquad\qquad (|\delta'_y| \leq 1/2) \\
&\leq 2^{-k_y+1}/y \\
&\leq 2^{-k-1+\mu^-(y)}/y \\
&\leq 2^{-k-1}.
\end{aligned}$$

□

Unfortunately, guaranteeing k *relative* bits using our usual framework of propagating relative precision bounds does not seem to work here. Intuitively, the reason is that $\ln(y)$ vanishes at $y = 1$.

Remarks.

1. The analysis shows that propagating absolute precision is easier ("more natural") for addition and logarithms. Similarly, relative precision is more natural for multiplication, division and square roots. Exponentiation seems not to have any preference.

2. More importantly, in case of $x = y \pm z$ and $x = \ln y$, we could not propagate relative precision from x to its children without knowledge of $\mu^-(x)$ (i.e., lower bounds size on $|x|$). Similarly, in case of $x = \sqrt{y}$, we could not propagate absolute precision without knowledge of $\mu^-(x)$. We consider these cases difficult, because lower bounds are not easy to compute in general.
3. In the above analysis, we either propagating absolute bits into absolute bits or propagating relative bits into relative bits. One could also propagate absolute bits into relative bits, or vice-versa. For instance, to compute $x = yz$ to k absolute bits, let k_y, k_z be the *relative* bits required for y or z. Then it is sufficient to choose

$$k_y = \max\{a + 1 + \mu^+(x), r_y\}, \qquad k_z = \max\{a + 2 + \mu^+(x), r_z\}$$

where $r_y + r_z = k + 2$.

5. Expression Evaluation and Constructive Zero Bounds

Until now, we examine the approximability of individual functions. We now examine the approximability of composition of functions. This turns out to be a key problem in guaranteed accuracy computation.

Suppose e is an algebraic expression involving the operators $\pm, \times, \div, \sqrt{}$ with constants $\mathbb{Z}$. We want to approximately compute the value of e (if e is valid). The considerations in the previous section show that, in the presence of $\pm$ operators, we could not guarantee relative precision in evaluating e. At least, it is not clear how to do this using only the elementary considerations of that section. Similarly, in the presence of division, we could not guarantee absolute precision. Some new idea is needed: this is the concept of constructive zero bounds to be introduced. The problem of approximate expression evaluation was first treated by Yap and Dubé[56].

Let Ω be any set of partial real functions. Each $\omega \in \Omega$ is called an *operator*. Let $\Omega^{(m)}$ denote all the operators of arity m in Ω. In particular, the operators in $\Omega^{(0)}$ are the constant operators, and these are identified with elements of $\mathbb{R}$. We call Ω a *computational basis* if $\Omega_0 \subseteq \Omega$ where $\Omega_0 = \{+, -, \times\} \cup \mathbb{Z}$. If, in addition, each operator in Ω is absolutely (resp., relatively) approximable, then we call Ω a *absolute basis* (resp., *relative basis*).

The evaluation problem. Let $\mathrm{Expr}(\Omega)$ be the set[h] of *expressions* over Ω. We view expressions in $\mathrm{Expr}(\Omega)$ as a rooted dags (directed acyclic graphs), where each node of out-degree m are labeled by operators in $\Omega^{(m)}$. So the leaves are labeled by the constant operators. The dag is ordered in the sense that the outgoing edges from each node has a total order (so that we can speak of the first outgoing edge, second outgoing edge, etc). There is a natural *evaluation function*, denoted Val_Ω (or simply Val),

$$\mathrm{Val}_\Omega : \mathrm{Expr}(\Omega) \to \mathbb{R}$$

which is also a partial function. The value $\mathrm{Val}(e)$ is defined recursively, by applying the operators at each node of e to their arguments. For instance, if the root of e has the operator $\div$ and its first child is e' and second child is e'' then $\mathrm{Val}(e) = \mathrm{Val}(e')/\mathrm{Val}(e'')$. We have the standard rule that $\omega(x_1, \ldots, x_m)$ is undefined if any x_i is undefined. We say e is *valid* if $\mathrm{Val}(e) \downarrow$, and *invalid* otherwise. Unlike some contexts (e.g., IEEE arithmetic), we do not distinguish among the invalid values. Thus $\pm 1/0 = \pm\infty$ as well as $0/0 = \texttt{NaN}$ are equally invalid.

The *evaluation problem for* Ω amounts to computing the function Val_Ω. In general, we want to approximately evaluate this function.

Approximate numerical and semi-numerical problems. The problem of approximating Val_Ω is slightly different from the kinds of functions discussed in Section 3. There, we defined approximability of "purely" numerical problems, of the form $f : \mathbb{R}^m \to \mathbb{R}$ where m is fixed. One immediate generalization we need is to allow m to vary and to become unbounded. For instance, if f is the problem of computing a determinant, then m range over the set $\{n^2 : n \in \mathbb{N}\}$. Hence a "purely numerical problem" is now a partial function of the form $f : \mathbb{R}^* \to \mathbb{R}^*$, where $\mathbb{R}^* = \cup_{m \geq 0} \mathbb{R}^m$.

But the problem of computing Val_Ω is not purely numerical because its domain is $\mathrm{Expr}(\Omega)$ and not $\mathbb{R}^*$. So the input to Val_Ω has combinatorial data (namely, a dag with internal operator labels) as well as numerical data (real numbers at leaves). Following Knuth, we call such problems *semi-numerical*. Traditionally, one can continue to pretend that a semi-numerical problem is purely numerical by encoding its domain in $\mathbb{R}^*$. This is plausible for simple problems, but we will be granted that trying to encode $\mathrm{Expr}(\Omega)$

[h]For emphasis, we may call these *constant expressions* to contrast them to the more general notion of expression which allow free variables. E.g., $x^2 + 3y - 1$ where x, y are free variables.

in $\mathbb{R}^*$ is not a satisfactory solution. In general, the output is also semi-numerical (e.g., in convex hulls). Yap[55] argued that the semi-numerical data arising in geometry can generally be modeled as digraphs whose nodes and edges are labeled with tuples of numbers. The digraphs comprise the *combinatorial data* and the numerical labels comprise the *numerical data*. In the following, we will assume that all semi-numerical data are of this sort. We now extend our definition of approximability of numerical problems to approximability of semi-numerical problems as follows:

- As usual, the input is augmented with a composite precision bound $[a, r]$.
- The combinatorial data remains exact, for the input as well as output.
- The numerical data in the input and output are restricted to $\mathbb{F}$.
- The numerical output satisfies the given precision bound $[a, r]$.

This definition of *approximate semi-numerical problem* is consistent with the Exact Geometric Computation paradigm, which stipulates that ouput combinatorial data must be exact[55].

The question of approximating Val_Ω (relative or absolute) amounts to this: does the approximability of individual operators in Ω translate into the approximability of expressions over Ω? The significance of this will become clear in Section 8. As noted in the introduction, it may not be obvious that there is an issue here. Consider the composition of two functions, $f(g(x))$. The input x in our framework is restricted to representable reals, but the input to f is now $g(x)$ and this may not be representable. Hence, the approximability of f and g may not necessarily imply the approximability of $f(g(x))$.

Before we go further into the approximability question, let us consider the associated decision problems $\texttt{VALID}(\mathrm{Val}_\Omega)$, $\texttt{ZERO}(\mathrm{Val}_\Omega)$ and $\texttt{SIGN}(\mathrm{Val}_\Omega)$, which may simply be denoted by

$$\texttt{VALID}(\Omega), \qquad \texttt{ZERO}(\Omega), \qquad \texttt{SIGN}(\Omega).$$

They are the "fundamental problems" of guaranteed accuracy computation over Ω. In Section 3, we show that $\texttt{VALID}(f), \texttt{ZERO}(f), \texttt{SIGN}(f)$ may not be recursively equivalent. But when $f = \mathrm{Val}_\Omega$, these problems are often recursively equivalent:

Lemma 16: *Let Ω be a basis.*
(i) If $\div \in \Omega$ then $\texttt{VALID}(\Omega)$ and $\texttt{ZERO}(\Omega)$ are recursively equivalent.
(ii) If $\sqrt{\cdot} \in \Omega$ then $\texttt{VALID}(\Omega)$ and $\texttt{SIGN}(\Omega)$ are recursively equivalent.

Proof: (i) It suffices to reduce $\texttt{ZERO}(\Omega)$ to $\texttt{VALID}(\Omega)$: given an expression e, $\text{Val}(e) = 0$ iff e is valid and $1/e$ is invalid.
(ii) Similarly, to reduce $\texttt{SIGN}(\Omega)$ to $\texttt{VALID}(\Omega)$, note that $\text{Val}(e) \geq 0$ iff e and $\sqrt{e}$ are both valid. □

A hierarchy of computational bases. We first describe a hierarchy of bases that are important in practice.

- $\Omega_0 = \{+, -, \times\} \cup \mathbb{Z}$. The expressions over Ω_0 is the set of constant integral polynomials. Expressions such as determinants are found here. By definition, Ω_0 is the smallest basis. A useful extension of Ω_0 is $\Omega_0^+ = \Omega_0 \cup \mathbb{Q}$, (see Pion and Yap[44]).
- $\Omega_1 = \Omega_0 \cup \{\div\}$. The expressions over Ω_1 is the set of constant[i] rational functions.
- $\Omega_2 = \Omega_1 \cup \{\sqrt{\cdot}\}$. The expressions over Ω_2 are called *constructible expressions*, as they evaluate to the so-call constructible reals. The majority of problems in computational geometry are computable over this basis. We may extend Ω_2 to Ω_2^+ if we add $\sqrt[k]{\ }$ for each $k > 2$. This basis defines the *radical expressions*.
- $\Omega_3 = \Omega_2 \cup \{\text{RootOf}(P, i) : P \in \mathbb{Z}[X], i \in \mathbb{Z}\}$. If $i > 0$, $\text{RootOf}(P, i)$ denotes the ith largest real root of $P(X)$. E.g., $i = 1$ refers to the largest real root. If $i < 0$, we refer to the $|i|$th smallest real root of $P(X)$. If $i = 0$, we refer to the smallest positive root of $P(X)$, and we may also write $\text{RootOf}(P)$ instead of $\text{RootOf}(P, 0)$. Note that $\text{RootOf}(P, i)$ is considered a constant (0-ary) operator. We could also allow the coefficients of $P(X)$ to be expressions, so that $\text{RootOf}(P, i)$ is a $(d+1)$-ary operator that takes $d+1$ expressions as the coefficients of $P(X)$; this more general operator is denoted $\diamond(E_0, E_1, \ldots, E_d, i)$ in Burnikel et al[13]. Let Ω_3^+ be the extension of Ω_3 when we allow the $\diamond$-operators (diamond-operators).
- $\Omega_4 = \Omega_3 \cup \{\exp(\cdot), \ln(\cdot)\}$. This gives us the class of constant elementary expressions[16].
- $\Omega_5 = \Omega_3 \cup \mathcal{H}$ where $\mathcal{H}$ is the set of real hypergeometric functions. The hypergeometric parameters in ${}_pF_q(\mathbf{a}, \mathbf{b}; x) \in \mathcal{H}$ are assumed

[i]It is paradoxical to call a constant expression a "rational function" or an "integral polynomial". To justify such a view is justified, think of a constant expression as a functional expression *together with* input constants. Our approximation algorithms take this viewpoint, and evaluate constant expressions as functional expressions with perturbed input numbers.

to be in $\mathbb{F}$. Now, $\text{Expr}(\Omega_5)$ contains the trigonometric and inverse trigonometric functions.

Lemma 17: *Assume $\mathbb{F}$ is the set of floating point numbers over some base B, with the standard representation.*
(i) The basis Ω_i $(i = 0, \ldots, 4)$ is a relative basis.
(ii) The basis Ω_5 is an absolute basis.

Proof: (i) It is sufficient to show that Ω_4 is a relative basis. It is well-known that each operator $\omega \in \Omega_4$ is absolutely approximable. If $\mathbf{x} \in \mathbb{F}^m$ and ω has arity m, we can also determine if $\omega(\mathbf{x})$ is defined or not, and whether $\omega(\mathbf{x}) = 0$. It follows that ω is relatively approximable.
(ii) To evaluate ${}_pF_q(\mathbf{a}, \mathbf{b}; x)$ with absolute error bound of ε, it is sufficient to determine an $n = n(\mathbf{a}, \mathbf{b}, x)$ such that, if we ignore terms beyond the nth term, the absolute value of the sum of the neglected terms is at most $\varepsilon/2$. This was shown in Du et al[19]. Then it is sufficient to evaluate the sum of the first n terms with error $\varepsilon/2$, which we can easily do. □

REMARKS: Neumaier[41] (p. 7) describes a slightly different class of "elementary operators" that are important in interval analysis. The $\text{RootOf}(P, i)$ operator can be replaced by $\text{RootOf}(P, I)$ where I is an isolating interval whose endpoints can be specified by other expressions. If i is out of bounds, or if I is not isolating interval, then $\text{RootOf}(P, i)$ and $\text{RootOf}(P, I)$ are invalid. We could generalize much of this discussion by viewing the operators of Ω to be be partial functions over $\mathbb{C}$, or some even more general algebraic structure. When viewed as complex operators, the trigonometric functions already appear in $\text{Expr}(\Omega_4)$. In the presence of trigonometric functions, it is natural to admit π as a constant operator of Ω.
Computable zero bounds. According to Theorem 5, we could achieve relative approximability by absolute approximability plus a decision procedure for zero. For example, for the class Ω_2, one could use a direct method for deciding zero (indeed, the sign) of expressions, by repeated squaring. In practice, such an approach is not used. The most effective method for this seems to be the use of constructive zero bounds. Mignotte[38] was the first to use this, for testing the equality of two algebraic numbers. In the context of EGC, it was first introduced in the `Real/Expr` package[56]. We

call[j] a function

$$B : \mathrm{Expr}(\Omega) \to \mathbb{R}_{>0}.$$

a *zero bound function* for Ω if for all $e \in \mathrm{Expr}(\Omega)$, whenever e is valid and $\mathrm{Val}(e) \neq 0$ then

$$|\mathrm{Val}(e)| \geq B(e).$$

Such bounds are always "conditional bounds" since it is a bound only when e is valid and non-zero. A simple example of zero bound function is $B(e) = |\mathrm{Val}(e)|$ (when e is invalid, $B(e)$ can be arbitrary). This is not a useful choice for B since its main purpose is to help us approximate the value $\mathrm{Val}(e)$. What we need are "easily" computable zero bound functions. If B is a zero bound function, the function $\beta : \mathrm{Expr}(\Omega) \to \mathbb{R}$ where $\beta(e) := -\lg B(e)$ is called a *zero bit-bound function* for Ω. We use B or β interchangeably.

Several such constructive zero bounds are known[44]. These zero bounds are not easy to compare because they depend on different parameters. One of the most effective bounds currently available is the so-called BFMSS Bound[13].

The result of Section 3 shows that Val_Ω can relatively approximated by combining an absolute approximation algorithm, with a decision procedure for $\mathtt{ZERO}(\Omega)$. We now give an alternative and more practical approach based on zero bounds.

In general, we are interested in subsets $E \subseteq \mathrm{Expr}(\Omega)$. Given $e \in E$ and $a \in \mathbb{Z}$, consider three related problems:

- $\mathrm{Val}(e : a)$ computes an absolute approximation to $\mathrm{Val}(e)$ with a absolute bits.
- $\mu^+(e)$: to compute an upper bound on $\lg(|\mathrm{Val}(e)|)$.
- $\mathtt{sign}(e)$: to determine the sign of $\mathrm{Val}(e)$,

These problems are intertwined: from $\mathrm{Val}(e : a)$, we can obtain $\mu^+(e)$ and sometimes deduce $\mathtt{sign}(e)$. But to compute $\mathrm{Val}(e : a)$, we may need first determine $\mathtt{sign}(e')$ or $\mu^+(e')$ where e' is a child of e. If e is invalid, then all three values $\mathrm{Val}(e, a), \mu^+(e), \mathtt{sign}(e)$ are undefined.

Let $\mathrm{Val}_E : \mathrm{Expr}(\Omega) \to \mathbb{R}$ be the problem of evaluating expressions $e \in E$, with $\mathrm{Val}_E(e) \uparrow$ when $e \notin E$. We need some restrictions on E. In general, for sets $X \subseteq Y$, we call X *a decidable subset of* Y if there is a Turing machine which, given $y \in Y$, will return 1 or 0, depending on whether $y \in X$ or not.

[j]These have also been called "root bounds".

A set $E \subseteq \mathrm{Expr}(\Omega)$ is said to be *admissible* if (1) $\mathrm{Expr}(\Omega_0) \subseteq E$, (2) E is decidable subset of $\mathrm{Expr}(\Omega)$, and (3) E is closed under subexpressions, i.e., if $e \in E$ and e' is a subexpression of e then $e' \in E$.

Theorem 18: Let $E \subseteq \mathrm{Expr}(\Omega_4)$ be admissible. If $\beta : E \to \mathbb{F}_{\geq 0}$ is a computable zero bound function then $\mathrm{Val}(e : a), \mu^+(e)$ and $\mathtt{sign}(e)$ are computable for $e \in E$.

Proof: Let $\beta : E \to \mathbb{F}_{\geq 0}$ be a computable zero bit-bound function. The following proof gives a single algorithm to compute all three functions simultaneously. Given an expressin e, we consider the "type" of e:

$e \in \Omega^{(0)}$:
(1) $b \leftarrow \max\{a, \beta(e) + 2\}$. and $v \leftarrow \mathrm{Val}(e : b)$. By assumption, we can compute such a v. Note that the $\mathrm{RootOf}(P, i)$ operator falls under this case.
(2) $\mu^+(e) \leftarrow \lceil \lg |v| + 1 \rceil$.
(3) If $v \leq 2^{-\beta(e)-1}$, return(ZERO); else $\mathtt{sign}(e) \leftarrow \mathtt{sign}(v)$.

$e = e_1 \pm e_2$:
(1) $\mu^+(e) \leftarrow 1 + \max\{\mu^+(e_1), \mu^+(e_2)\}$.
(2) $v \leftarrow \mathrm{Val}(e_1 : b) \pm \mathrm{Val}(e_2 : b)$. where $b \leftarrow \max\{a + 1, \beta(e) + 2\}$.
(3) If $v \leq 2^{-\beta(e)-1}$, return(ZERO); else $\mathtt{sign}(e) \leftarrow \mathtt{sign}(v)$.

$e = e_1 e_2$:
(1) $\mathtt{sign}(e) \leftarrow \mathtt{sign}(e_1)\mathtt{sign}(e_2)$. If $\mathtt{sign}(e) = 0$, return(ZERO).
(2) $\mu^+(e) \leftarrow \mu^+(e_1) + \mu^+(e_2)$.
(3) $v_i \leftarrow \mathrm{Val}(e_i : a + 1 + \mu^+(e_{3-i})$ and $v \leftarrow v_1 v_2$. [cf. Lemma 8]

$e = e_1/e_2$:
(1) If $\mathtt{sign}(e_2) = 0$, return(INVALID). If $\mathtt{sign}(e_1) = 0$, return(ZERO); else $\mathtt{sign}(e) \leftarrow \mathtt{sign}(e_1)\mathtt{sign}(e_2)$.
(2) $\mu^+(e) \leftarrow \mu^+(e_1) - \beta(e_2)$.
(3) $v_1 \leftarrow \mathrm{Val}(e_1 : a+2-\beta(e_2)$ and $v_2 \leftarrow \mathrm{Val}(e_2 : \max\{1-\beta(e_2), k+4-2\beta(e_2) + \mu + (e_1)\})$. Finally, $v \leftarrow v_1/v_2[a+1]$ (approximate to $a+1$ absolute bits). [cf. Lemma 11]

$e = \sqrt{e_1}$:
(1) If $\mathtt{sign}(e_1) < 1$ then return(INVALID). If $\mathtt{sign}(e_1) = 0$, return(ZERO); else $\mathtt{sign}(e) \leftarrow 1$.
(2) $\mu^+(e) \leftarrow \mu^+(e_1)/2$.
(3) $v_1 \leftarrow \mathrm{Val}(e_1 : \max\{a + 1, 1 - \beta(e_1)/2\})$ and compute v as an $a+1$ absolute bit approximation to $\sqrt{v_1}$. [cf. Lemma 12]

$e = \exp(e_1)$:

(1) $\texttt{sign}(e) \leftarrow 1$.
(2) $\mu^+(e) \leftarrow 4^{\mu^+(e_1)}$.
(3) $v_1 \leftarrow \mathrm{Val}(e_1 : a + 2 + 2^{\mu^+(y)+1})$ and $v \leftarrow \exp(v_1 : a + 1)$. [cf. Lemma 13]

$e = \ln(e_1)$:
(1) If $\texttt{sign}(e_1) \leq 0$ then return(INVALID).
(2) $\mu^+(e) \leftarrow \lceil \lg(\mu^+(e_1)) \rceil$.
(3) $b_0 \leftarrow \beta(e_1 - 1)$, and $v_1 \leftarrow \mathrm{Val}(e_1 : b_0 + 1)$. Note that $e_1 - 1$ is a new expression whose conditional zero bound is needed.
(4) If $|v_1 - 1| < 2^b$ then return(ZERO); else $\texttt{sign}(e) \leftarrow \texttt{sign}(v_1 - 1)$.
(5) $b_1 \leftarrow \max\{1 + \beta(e_1), a + 2 + \beta(e_1)\}$ and $v \leftarrow \mathrm{Val}(e_1 : b_1)$. [cf. Lemma 15]

Normally, the values returned are $v = \mathrm{Val}(e : a)$, $\mu^+(e)$ and $\texttt{sign}(e)$. But there are two special return statements: INVALID and ZERO, in which cases these values are determined.

The justification of the various cases comes from the propagation bounds we derived in the previous section. We just cover the details of the last case, for logarithms. After checking validity of the expression (Step 1), we can bound $\mu^+(e)$ as in step 2. Determining the sign of $\mathrm{Val}(e)$ is trickier, since it amounts to comparing $\mathrm{Val}(e_1)$ to 1. Hence we need to determine a zero bound $b_0 = \beta(e_1 - 1)$ for a new expression "$e_1 - 1$". With this in hand, we evaluate $\mathrm{Val}(e_1)$ to $b_0 + 1$ absolute bits. This approximation can then tell us whether $\mathrm{Val}(e_1)$ is equal to, less than, or greater than 1 (Step 4). This is the information needed to determine $\texttt{sign}(e)$. Finally in Step 5, we approximate $\mathrm{Val}(e)$ to a absolute bits, following Lemma 15. We could have combined Steps 3 and 5 for efficiency. □

The algorithm in the proof aims at simplicity. In practice, it would be more efficient to separate the algorithm into three mutually recursive algorithms. Furthermore, the zero bound β should not be used directly, but to control an adaptive algorithm.

Corollary 19: *Let $E \subseteq \mathrm{Expr}(\Omega_4)$ be admissible. Then E has a computable zero bound function iff Val_E is relatively approximable.*

Proof: ($\Rightarrow$) If $\beta : E \to \mathbb{R}_{\geq 0}$ is a computable zero bit-bound function, then by the preceding theorem, Val_E is absolutely approximable and $\texttt{SIGN}(E)$ are computable. By Theorem 5, Val_E is relatively approximable.

($\Leftarrow$) If Val_E is relatively approximable, then a zero bit-bound for $e \in E$

can be computed as

$$\beta(e) \leftarrow 1 - \lg |\mathrm{Val}(e; 1)|.$$

If $\mathrm{Val}(e) = 0$, we may set $\beta(e) = 0$ (or any other value we like). □

Algebraic expressions and beyond. The strongest positive result about the guaranteed accuracy evaluation of expressions from our hierarchy is the following:

Theorem 20: The function $\mathrm{Val}_{\Omega_3^+}$ is relatively approximable.

One way to show this result is to invoke a decision procedure for Tarski's language of real closed fields. A weaker version of this theorem says that Val_{Ω_3} is relatively approximable: this follows from Corollary 19, and the fact[13] that $\mathrm{Expr}(\Omega_3)$ has a computable zero bound function.

It is a major open problem whether $\mathtt{ZERO}(\Omega_4)$ is decidable. This question is closely related to undecidable questions (by introducing variables into these expressions). Put another way, it is unknown whether we can compute with guaranteed precision over the basis Ω_4. The main result in this direction is from Richardson[45]. It seems to imply the following claim: $\mathtt{ZERO}(\Omega_4)$ *is decidable if Schanuel's conjecture is true.*

Here, *Schanuel's conjecture* says if $x_1, \ldots, x_n \in \mathbb{C}$ are linearly independent over $\mathbb{Q}$ then the transcendence degree of $x_1, \ldots, x_n, e^{x_1}, \ldots, e^{x_n}$ is at least n. This assertion is highly non-trivial because it implies many known but deep results in transcendental number theory. Richardson's result does not directly this claim. The reason we do not have an immediate result is because Richardson has a different framework than us. First, he treats the more general complex case. But he uses a concept of "expressions", which is captured as follows. Let $\Omega_4^- := \Omega_4 \setminus \{\div, \ln(\cdot)\}$. His expressions are systems of equations (involving free variables) over the operators of Ω_4^-, together with some additional side restrictions in order to ensure that such a system determines a unique number. The advantage of Ω_4^- is that one can compute absolute approximations for its expressions *without zero bounds*. Richardson's algorithm for deciding zero uses two non-trivial algorithms, lattice reduction and Wu's algorithm.

6. The Algebraic Computational Model

Standard complexity theory, based on the Turing model, requires all inputs to be encoded as strings. This is unsuitable for some problems in

algebraic computing. An example is the Mandelbrot set comprising those $z \in \mathbb{C}$ such that the infinite sequence $T(0), T^2(0), T^3(0), \ldots$ is bounded where $T(w) = w^2 + z$. Is this set computable? This question is not meaningful in the standard theory (see discussion in [BCSS, Section 1.2.1][6]). The most direct way to attack this problem is to consider algebraic models of computation[7,11]. In the algebraic model, we postulate an algebraic set D together with a set Ω of operators on D. For our purposes, we take $D = \mathbb{R}$.

The simplest algebraic model is the *straightline program*[7]. By allowing decision nodes, we get *algebraic decision trees*. Such models are finite or non-uniform. The uniform version of such models was first studied by Blum, Shub and Smale[6]. The Mandelbrot decision problem above turns out to be undecidable. The BSS Model achieves uniformity by introducing a bi-infinite array, indexed by the integers, $i \in \mathbb{Z}$. Each machine instruction transforms the contents of the cell at position 0. To bring other cells into the 0 position, we use the left- and right-shift operators. Let $f : \mathbb{R}^* \to \mathbb{R}$ be a *numerical problem*; an input $w = (w_1, \ldots, w_n) \in \mathbb{R}^*$ is placed into the array so that w_i is in position i $(i = 1, \ldots, n)$. To indicate the number n of arguments, we may place the number n in position 0. Finally, the output can be placed in position 0.

This model is awkward for modeling semi-numerical problems. Our evaluation problem Val_Ω is such an example. The BSS Model would require encoding the input expressions as a linear sequence of array values. To overcome this, we introduce an algebraic model which supports semi-numerical objects more naturally. We based it on the elegant Storage Modification Machines, or *Pointer Machines*, of Schönhage [47]. Similar models were earlier proposed by Kolmogorov and Uspenskiĭ, and by Barzdin and Kalnin'sh.

Pointer structures. Like Turing's model, pointer machines use finite state control to determine the step-by-step execution of instructions. What is interesting is that pointer machines manipulated data structures with changeable neighborhoods, unlike the fixed neighborhoods of Turing machine tapes. Let Δ an arbitrary finite set of symbols; each $a \in \Delta$ is called a *color*. Consider the class of finite, directed graphs with out-degree $|\Delta|$ but arbitrary in-degree. Let G be a member of this class. The edges of G are called *pointers*, and each edge is labeled ("colored") by some $a \in \Delta$. The outgoing edges from a node have distinct colors. Thus, for each color a and each node u, there is a unique *a-pointer* coming out of u. One of the nodes is designated the *origin*. Call G a *Δ-structure* or *pointer structure*. Each word $w \in \Delta^*$ is said to *access* the unique node obtained by following the

sequence of pointers labeled by colors in w, starting from the origin. Let this node be denoted $[w]_G$ (or simply $[w]$ when G is understood). The *empty word* ϵ accesses the origin, denoted $[\epsilon]$. In general, there will be inaccessible nodes. For any node $u \in G$, let $G|u$ denote *u-accessible structure*, namely, the Δ-structure with origin u and comprising all nodes accessible from u. If $w \in \Delta^*$ then we write $G|w$ instead of $G|[w]$.

Let $\mathcal{G}_\Delta$ denote the class of all Δ-structures, and $\mathcal{G}$ denote the union of $\mathcal{G}_\Delta$ over all Δ. Notice that if $\Delta \subseteq \Delta'$ then there is a natural embedding of $\mathcal{G}_\Delta$ in $\mathcal{G}_{\Delta'}$. For simplicity, we shall just treat $\mathcal{G}_\Delta$ as a subset of $\mathcal{G}_{\Delta'}$.

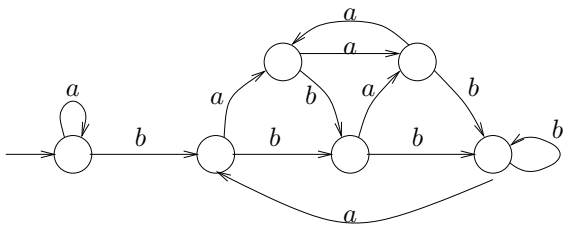

Fig. 2. Pointer machine Δ-structure ($\Delta = \{a, b\}$).

As directed labeled graph, each Δ-structure has a standard graphical representation. This is illustrated in Figure 2. The origin (node 1) is indicated by an unlabeled arrow from nowhere. Node 4 can be accessed by $w = aabb$ as well as $w' = bab$. So $4 = [w] = [w']$. We use two conventions to reduce clutter: (1) If a pointer is a self-loop (i.e., its target and source are the same, they are omitted in the diagram. For instance, the self-loop at node 1 can be omitted. Node 6 has a self-loop with color b that has already been omitted. (2) If two or more pointers share the same source and target, then we only draw one arrow and label them with a list of colors for this arrow. Thus, the two pointers out of node 5 have already been collapsed into one using this convention.

We define a *pointer machine* (for any color set Δ) as a finite sequence of instructions of the following four types:

Type	Name	Instruction	Meaning
(i)	Node Assignment	$w \leftarrow w'$	$[w]_{G'} = [w']_G$
(ii)	Node Creation	$w \leftarrow$ **new**	$[w]_{G'}$ is new
(iii)	Node Comparison	**if** $w \equiv w'$ **goto** L	$G' = G$
(iv)	Halt and Output	**HALT**(w)	Output $G|w$

In this table, $w, w' \in \Delta^*$ and L is a natural number viewed as the label of instructions. The instructions of the pointer machines are implicitly labeled by the numbers $1, 2, 3, \ldots$ in sequential order. Normally, instruction $i+1$ is executed after instruction i unless we branch to an instruction after a Type (iii) instruction. Let us explain the last column of this table (the meaning of the instructions). Let G be the Δ-structure before executing an instruction; it is transformed by the instruction to G'.

(i) If w' accesses the node v in G then after executing this assignment, both w and w' access v in G'. In symbols, $[w]_{G'} = [w']_G$ $(= [w']_{G'})$. This is achieved by modifying a single pointer in G. If $w = u.a$ where $u \in \Delta^*$ and $a \in \Delta$, then this instruction makes the a-pointer issuing from $[u]$ to next point to $[w']_G$. There is a special case, when $w = \epsilon$. In this case, no pointer is modified, but the new origin is $[w']_G$.

(ii) We add a "brand new" node v to G to form G', and w now accesses v. Furthermore, each pointer from v points back to itself. As in (i), the transformation $G \to G'$ is achieved by modifying a single pointer in G.

(iii) If $[w']_G = [w]_G$ then we branch to the Lth statement; otherwise we execute the next instruction in the normal fashion. The Δ-graph is unchanged, $G = G'$.

(iv) The machine halts and outputs the Δ-structure $G|w$. We also allow a variant of halt with *no* output (i.e., w is unspecified). This analogous to a Turing machine halting in state $q_\uparrow$.

Computation and I/O conventions. Each pointer machine M computes a partial function

$$f_M : \mathcal{G}_\Delta \to \mathcal{G}_\Delta$$

for some color set Δ: on input $G \in \mathcal{G}_\Delta$, the machine will transform G according to the instruction it is executing. At each step, it is executing some instruction (numbered) L. At the next step, it normally executes instruction $L+1$ unless a type (iii) instruction succeeds in transferring it to some other instruction L'. The machine *halts* iff it executes a type (iv) instruction. When it halts, it either produces an output $f(G) \in \mathcal{G}_\Delta$, or has no output (equivalent to entering state $q_\uparrow$). It may not halt for one of two reasons: it executes infinitely many instructions of non-type (iv), or it tries to execute a non-existent instruction. If it does not halt or halts with no output, then $f(G)$ is undefined. It is then clear what it means for M to *unconditionally (resp., partially, conditionally) compute* a function

$f : \mathcal{G}_\Delta \to \mathcal{G}_\Delta$.

An arbitrary Turing machine M can be simulated by a pointer machine P: Suppose M has k work tapes and the set of tape symbols is Σ. Then we let $\Delta = \Sigma \cup \{S, L, R, C_1, \ldots, C_k\}$ where C_i will indicate the current position of the ith tape head. The colors L, R is used to from a tape cell u to its left (L) or its right (R) neighbor. The cell u is said to store the symbol $\sigma \in \Sigma$ if $[u.\sigma] \neq [u]$ (we must make sure that there is exactly one such σ). The states of M will be directly remembered in the states of P (identified with the instruction numbers of P). Each step of M will only require $O(1)$ steps of P. We leave the detailed simulation to the reader. When we use a Turing machine to compute a function f, we have some input/output convention. This convention is easily transformed into our I/O convention for pointer machines. In particular, if M enters the special state $q_\uparrow$, we can also ensure that P enters a corresponding special state (still denoted $q_\uparrow$). If the output size is k, our pointer machine will take $O(k)$ steps to produce an output. This extra time does not change the overall time complexity. The following lemma record these observations:

Theorem 21: A partial function $f : \Sigma^k \to \Sigma$ is unconditionally (partially, conditionally) computed by a Turing machine in time $T(n)$ iff it is unconditionally (partially, conditionally) computed by a Pointer machine in time $O(T(n))$.

In other words, the concept of computability is invariant whether we use Turing machines or Pointer machines (again confirming Church's thesis).

Algebraic pointer machines. We now augment the Pointer machines to support algebraic computation. Let R be any ring and be Ω be a set of operators (i.e., partial functions of various arity) over R. Such machines compute over the set of *algebraic pointer structures*: these are just pointer structures in which each node u can hold an arbitrary value of R or may be undefined. Let $\mathcal{G}_\Delta(R)$ denote the set of pointer structures with color set Δ and values taken from R. For $G \in \mathcal{G}_\Delta(R)$ and $w \in \Delta^*$, write $\mathrm{Val}_G(w)$ for the value stored at $[w]_G$. Let

$$\mathcal{G}(R) = \bigcup_\Delta \mathcal{G}_\Delta(R)$$

where Δ range over all color sets.

We add two new types of instructions:

Type	Name	Instruction
(v)	Value Comparison	**if** $(w \circ w')$ **goto** L where $\circ \in \{=, <, \leq\}$
(vi)	Value Assignment	$w := f(w_1, \ldots, w_m)$ where $f \in \Omega$ and $w, w_i \in \Delta^*$

Let us discuss the meaning of the new instruction types. A type (v) instruction causes a branch to instruction L if the predicate $\mathrm{Val}_G(w) \circ \mathrm{Val}_G(w')$ is true, but does not change the pointer structure: $G = G'$. The comparison $\circ$ would be restricted to "=" when R is not ordered. A type (vi) instruction changes G to G' so that $\mathrm{Val}_{G'}(w) = f(\mathrm{Val}_G(w_1), \ldots, \mathrm{Val}_G(w_m))$. The values of other nodes are unchanged. The pointers in G and G' are unchanged.

The treatment of undefined values in type (vi) instructions is standard – they are propagated by assignment. But in the case of type (v) instructions, there is no standard treatment. We adopt the following convention: viewing the undefined value $\uparrow$ as a special symbol, we assume the undefined value is equal only to another undefined value but to no other values. This implies we can test for the undefined value. Also the predicate "$\uparrow \leq x$" holds iff x is undefined, and the predicate "$\uparrow < x$" never hold.

Observe[k] that types (i) and (vi) are analogous: we use $w \leftarrow \ldots$ to denote pointer assignment, while $w := \ldots$ denotes assigning $f(\mathrm{Val}(w_1), \ldots, \mathrm{Val}(w_m))$ to $\mathrm{Val}(w)$. Similarly, (iii) and (v) are analogous: $w \equiv w'$ compares the nodes $[w]$ and $[w']$, while $w \circ w'$ compares their *values*, $\mathrm{Val}(w)$ and $\mathrm{Val}(w')$.

An *algebraic pointer machine over basis* Ω (or simply, *algebraic* Ω*-machine*) is a finite sequence of instructions of types (i)-(vi). Computation by algebraic pointer machines is follows exactly the same conventions as given by the regular pointer machines. So an algebraic machine M computes a partial function

$$f_M : \mathcal{G}_\Delta(R) \to \mathcal{G}_\Delta(R). \tag{17}$$

Given another partial function

$$F : \mathcal{G}_\Delta(R) \to \mathcal{G}_\Delta(R), \tag{18}$$

[k]Here is a mnemonic device to differentiate "$\leftarrow$" from ":=", and "$\equiv$" from "=". The arrow in "$\leftarrow$" suggests a pointer link, and hence refers to pointer assignment; in contrast, the symbol := recalls the "=" in comparing algebraic values. Similarly, the symbol $\equiv$ suggests symbolic identity (as in polynomial identity), and hence refers to equality of nodes; in contrast, the symbol = suggests equality of values in the mathematical domain.

we say M to *unconditionally computes* F if (i) M halts on all inputs and (ii) F and f_M are identical as partial functions. We then say that F is *algebraically computable* or Ω-*computable*. Note that in "algebraically computability", there is always a computational basis Ω which may be implicit.

REMARK: In any programming model, we expect the identity assignment. In our notation, this amounts to the instruction "$w := v$" where $w, v \in \Delta^*$. This amounts to assuming that the basis Ω has the identity function. This assumption is harmless: since R is a ring and Ω contains Ω_0, the identity assignment may be simulated by two instructions "$w := v + v_0$; $w := w - v_0$" where $[v_0]$ is any node with a defined value.

Real pointer machines. Let us now specialize R to the reals $\mathbb{R}$. Then algebraic pointer machines will be called *real pointer machines* and these, operates on *real pointer structures*, $\mathcal{G}(\mathbb{R})$. Such machines compute partial functions of the form

$$F : \mathcal{G}(\mathbb{R}) \to \mathcal{G}(\mathbb{R}).$$

Other semi-numerical structures can easily be embedded in $\mathcal{G}(\mathbb{R})$ as in the next two examples:

EXAMPLE 1. Assume some fixed encoding of $\mathbb{R}^*$ in $\mathcal{G}_{\Delta_0}(\mathbb{R})$ where Δ_0 is a suitable color set. Then we may speak of a purely numerical problem $F : \mathbb{R}^* \to \mathbb{R}^*$ as being Ω-computed by real pointer machines using any color set $\Delta \supseteq \Delta_0$ (recall our subset embedding convention, $\mathcal{G}_{\Delta_0}(\mathbb{R}) \subseteq \mathcal{G}_{\Delta}(\mathbb{R})$).

EXAMPLE 2. Consider the problem of evaluation of expressions over Ω. We assume that $\Omega^{(0)} \subseteq \mathbb{R}$ and $\Omega \setminus \Omega^{(0)}$ is a finite set. Let Δ contains a color $op(\omega)$ for each $\omega \in \Omega \setminus \Omega^{(0)}$, and the integers $1, \ldots, m^*$ where m^* is the maximum arity in Ω. We describe the encoding $e \in \mathrm{Expr}(\Omega)$ as a Δ-structures $G(e)$. If we restrict $G(e)$ to the pointers colored by numbers $(1, \ldots, m^*)$ and which are not self-loops, then G is isomorphic to the DAG of e. The i-pointer ($i = 1, \ldots, m^*$) leads to the i-th argument of a node. If node u in e is an operator $\omega \in \Omega^{(m)}$ $m \geq 1$, then the $op(\omega)$-pointer of u points to the origin; all other operator pointers are self-loops. Finally, if u is a leaf, then all i-pointers are self-loops and $\mathrm{Val}_G(u)$ stores a valie $\Omega^{(0)}$. Given such $G(e)$, an algebraic Ω-machine computes $\mathrm{Val}_\Omega(e)$ in the obvious way: it amounts to a bottom up evaluation of the nodes of the DAG. Finally we return the value at the root of the DAG.

REMARKS: In terms of computability, the Algebraic Pointer Model is equivalent to the BSS model. The Algebraic Pointer Model is clearly elegant basis for algebraic computation involving combinatorial structures.

But the fundamental reason for preferring the Algebraic Pointer Model is a complexity-theoretic one: the BSS model can distort the complexity of problems with low complexity. This has two causes: first, the BSS Model does not encode combinatorial structures easily (it requires the analogue of Gödel numberings in recursive function theory). Second, BSS machines are too slow in accessing new array elements with its shift operator. One possible solution is to augment the BSS model by introducing special "index variables" which are restricted to values in $\mathbb{Z}$ and can be added and subtracted (or even multiplied). Index variables are to be used as arguments to the shift operators. In pointer machines, no such facility is needed: the standard technique of "pointer doubling" can achieve the same effect of rapid access. Like the Turing model, point machines are capable of many interesting variations. It is easy to expand the repertoire of instructions in non-essential ways (e.g., allowing the ability to branch on a general Boolean combination of equality tests). We may assume these without much warning.

7. Numerical Model of Computation

The algebraic model is natural and useful for investigating many algebraic complexity questions. But it is far removed from the real world "computation modes" described in Section 2. For instance, it does not address two known criticisms ([Weihrauch, Chapter 9][51]) of non-effectiveness in real algebraic models: (I) Arbitrarily numbers as objects that are directly manipulated. Such numbers might be uncomputable reals. In the real world, we need to represent numerical quantities with non-trivial description sizes. (II) The operators in Ω as perfect oracles. Since the operators can be applied to values with non-trivial complexity, even "simple" operators such as $+$ are highly non-trivial.

This section introduces a numerical model of computation which lies intermediate[l] between the algebraic model (which is too abstract) and the Turing model (which is too concrete). Our model restricts numerical inputs to some representable set $\mathbb{F} \subseteq \mathbb{R}$. Second, we consider "approximate operators" that accepts an auxiliary "precision" parameter $p \geq 0$. These steps remove the above objections (I) and (II).

[1]We are aware that the BSS model *formally* incorporates the Turing model as a special case, when $R = \mathbb{Z}_2$. But it is clear that the development of the BSS theory is novel only when R is an infinite ring like $R = \mathbb{R}$. It seems more useful for our purposes to view these as two distinct theories.

Numerical pointer machines. These are essentially a special kind of real pointer machines. Let Ω be a basis of real operators. We need three changes: First, the value set R is now the set $\mathbb{F}$ of representable reals introduced in Section 3. The new computational structures $\mathcal{G}(\mathbb{F})$ are called *numerical pointer structures*. Second, each $f \in \Omega^{(m)}$ is replaced by a relative approximation function $f(x_1, \ldots, x_m; p)$. Third, the instructions of type (vi) are replaced by the following type (vii) instructions:

Type	Name	Instruction
(vii)	Approximate Assignment	$w := f(w_1, \ldots, w_m; v)$ where $f \in \Omega$

Here, $w_1, \ldots, w_m, v \in \Delta^*$. The semantics is evident: $\mathrm{Val}_{G'}(w)$ will be assigned a relative approximate value $f(\mathrm{Val}_G(w_1), \ldots, \mathrm{Val}_G(w_m); \mathrm{Val}(v))$. In practice $\mathrm{Val}(v)$ will be non-negative and even integers, but there is no harm allowing it to be unrestricted for this definition.

This modification has one interesting consequence: even constants $\omega \in \Omega^{(0)}$, can become non-trivial functions that takes a precision parameter. For instance, if $\pi \in \Omega^{(0)}$ then the numerical model must provide an operator $\pi(r)$ to produce arbitrarily precise approximations to π.

Approximating semi-numerical functions. A sequence of instructions of types (i)-(v) and type (vii) will be called a *numerical Ω-machine* (or *numerical pointer machine*). Let N be such a machine. Clearly, N computes a partial function similar to (17), but with $R = \mathbb{F}$. But we want to view N as *approximating* some semi-numerical function. We proceed as follows: fix some standard embedding of $\mathcal{G}_\Delta(\mathbb{F}) \times \mathbb{F}$ into $\mathcal{G}_\Delta(\mathbb{F})$. Then we can re-interpret N as computing the following partial function

$$f_N : \mathcal{G}_\Delta(\mathbb{F}) \times \mathbb{F} \to \mathcal{G}_\Delta(\mathbb{F}), \tag{19}$$

with an extra precision parameter.

We have already clarified what it means to approximate semi-numerical data (Section 5). Applied to $G \in \mathcal{G}(\mathbb{R})$, let

$$G\langle p \rangle$$

denote the set of $G' \in \mathcal{G}(\mathbb{R})$ that approximates G with relative precision p: this means the underlying pointer structures of G and G' agree, but at each node $u \in G$, we have $|\mathrm{Val}_{G'}(u) - \mathrm{Val}_G(u)| \leq 2^{-p}|\mathrm{Val}_G(u)|$. Similarly, $G[p]$ denotes the approximations to G to absolute precision p.

If

$$F : \mathcal{G}_\Delta(\mathbb{R}) \to \mathcal{G}_\Delta(\mathbb{R})$$

is any partial function, we shall say the machine N *relatively approximates* F if for all $(G, p) \in \mathcal{G}_\Delta(\mathbb{F}) \times \mathbb{F}$, if $F(G)$ is defined then

$$f_N(G, p) \in F(G)\langle p \rangle,$$

and if $F(G)$ is undefined, then N halts with no output. We say the function F is *numerically approximable* if F is relatively approximated by some numerical pointer machine. Note that for a function F to be "numerically approximable", there is an implicit basis, Ω. So we say F is Ω-*approximable* to make this basis explicit.

It is not hard to see that the results of Section 5 about "relative approximability" can now be restated as results about "numerical approximability". For instance, Theorem 18 and its corollary translates into the following result:

Theorem 22: Let $E \subseteq \mathrm{Expr}(\Omega_4)$ be admissible. Then E has a computable zero bound iff Val_E is numerically Ω_4-approximable.

Main result. We give a sufficient condition for when algebraic computability implies numerical approximability. More precisely, we want conditions on Ω such that an (algebraic) Ω-computable function is (numerically) Ω-approximable. For this we need to make the assumption that

$$\Omega \setminus \mathbb{F} \tag{20}$$

is a finite set. This is because each operator in this set requires an approximation operator, and our model allows only a finite number of them.

Theorem 23: Let the function $F : \mathcal{G}(\mathbb{R}) \to \mathcal{G}(\mathbb{R})$ be Ω-computable. If Val_Ω is Ω-approximable then F is Ω-approximable.

Proof: Let A be an algebraic pointer machine that computes F. We must describe a numerical pointer machine N to numerically approximate F. Assume the color set of A is Δ; the color set of N will be some superset Δ' of Δ. The valid inputs[m] for N has the form pair $(G_0, p) \in \mathcal{G}_\Delta(\mathbb{F}) \times \mathbb{F}$. Our goal is to simulate the computation of A on the input G_0, and ultimately produce an output in $F(G_0)\langle p \rangle$.

[m]In particular, if $G_0 \in \mathcal{G}_{\Delta'}(\mathbb{F}) \setminus \mathcal{G}_\Delta(\mathbb{F})$ then N can halt with no output. Recall our convention that $\mathcal{G}_\Delta(\mathbb{F}) \subseteq \mathcal{G}_{\Delta'}(\mathbb{F})$.

The machine N simulates A step-by-step. Suppose at some step, the algebraic Δ-structure of A is G. Then for machine N, we maintain a corresponding numerical Δ'-structure G'. Basically G' is G with extra embellishments. In particular, for each node $u \in G$ the the corresponding node in G' (still denoted u) has an associated expression that can be accessed as $u.\texttt{Expr}$. Here $\texttt{Expr} \in \Delta'$ is a special color for accessing expressions associated with nodes.

We encode expressions over Ω as in EXAMPLE 2 (Section 6). For each $f \in \Omega \setminus \mathbb{F}$, we have the color $op(f) \in \Delta'$ to represent this operator in expressions. Consider the various types of instructions:

- For instructions of types (i)-(iii), N will execute exactly the same instructions as A. These instructions manipulate purely combinatorial data.
- For type (iv) instruction, we halt with output. N must go over the output Δ'-structure, and for each node u, to evaluate the expression $u.\texttt{Expr}$ to precision required by the input specification. By assumption, this is possible.
- Consider a type (vi) instruction of the form "$w := f(w_1, w_2)$". We assume f is binary here, but it clearly generalizes to any m-ary f. We execute the following sequence of instructions:

$$\begin{array}{l} w.\texttt{Expr} := \textbf{new}; \\ w.\texttt{Expr}.op(f) := \epsilon; \\ w.\texttt{Expr}.1 := w_1.\texttt{Expr}; \\ w.\texttt{Expr}.2 := w_2.\texttt{Expr}; \end{array}$$

  Thus, we simply construct the corresponding expression for the desired value.
- Consider a type (v) instruction of the form "**if** $(w \circ w')$ **goto** L". Although N has type (v) instructions like A, their semantics are not the same. In A, when we ask for the comparison $w \circ w'$ where $w, w' \in \Delta^*$, we are comparing the values $\mathrm{Val}_G(w), \mathrm{Val}_G(w') \in \mathbb{R}$. In N, we can only approximate these values. We execute the following sequence of instructions to construct a temporary expression corresponding to $[w].\texttt{Expr} - [w'].\texttt{Expr}$:

```
tmp.Expr :=new;
tmp.Expr.op(−) := ϵ;
tmp.Expr.1 := w.Expr;
tmp.Expr.2 := w′.Expr;
goto L_tmp;
```

where $\texttt{tmp} \in \Delta'$ is just another color and L_{tmp} is the beginning of instructions to evaluate the temporary expression just constructed. We invoke the relative approximability of Val_Ω to achieve this, and this implies we can get the correct sign and hence jump to the correct "next instruction" of A. For simplicity, we assume that N has a special location L_{tmp} for each branch instruction of A. Then this segment of code knows the correct next instruction. Clearly, more general programming techniques can reduce this code bloat in N. □

If we wish to compute F according to the principles of EGC (see introduction), then we can relax the conditions of this theorem: in that case, the Ω-approximability of Val_Ω can replaced by the Ω-computability of $\texttt{SIGN}(\Omega)$.

8. Conclusion

This paper outlines a theory of real approximation and introduces a model of numerical computation. Together, they capture the main features of "guaranteed precision mode of computation" which is being developed in the software libraries `LEDA Real` and `Core Library`. The practical deployment of this computational mode will open up many new applications, from the verification of conjectures to the advancement of reliable computing. We pose several open problems in this context.

- Guaranteed precision is a very strong requirement, not known to be possible outside of the algebraic realm. The main open question revolves around the decidability of the fundamental problems $\texttt{ZERO}(\Omega)$ where Ω is a basis containing non-algebraic operators.
- We focused on the computability of approximation, to outline the main features of this theory. Clearly, the complexity theoretic aspects ought to be developed. Another extension is to develop nondeterminism and give yet another form of NP-completeness (this is expected to be different from the known theories).

- The expression evaluation problem is central. There are several open problems here: generalize the Ω-results of this paper by requiring only general conditions on Ω (e.g., conditions on the derivatives). Even simpler: when is $f(g(x))$ approximable? Another problem is to give provably optimal algorithms for approximating Val_Ω or for $\mathtt{SIGN}(\Omega)$. We want here some "precision sensitive"[48,1]. concept of optimality. This is unclear even for $\Omega = \Omega_0$.
- In constructive zero bounds, an open question is whether there is a zero bound for $\mathrm{Expr}(\Omega_2)$ whose zero bit-bound is linear in the degree. There remains the practical need for stronger and more adaptive zero bounds. For instance, approximating expressions over Ω_3^+ is currently impractical with known zero bounds.
- Section 2 suggests a programming environment (or language) where different numerical accuracy requirements can co-exist and interplay. This presents many practical as well as theoretical challenges. Programming environments of the future ought to support such paradigms. For instance, as Moore's law predicts an inexorable increase of machine speed. Such an environment can exploit this, to achieve a trade-off between speed and accuracy (or "robustness").
- Section 7 gives us a condition when an abstract algebraic algorithm A can be implemented as a numerical algorithm B. Such A-to-B type results can provide some theoretical foundation for numerical analysis (as sought by the BCSS Theory[6]). Clearly, there are other A-to-B type results.

Acknowledgments

I am grateful for the support of Kurt Mehlhorn and the facilities of the Max-Planck Institute of Computer Science in Saarbrücken where this paper was completed. Thanks is due to Susanne Schmitt for careful reading of this manuscript and comments.

References

1. T. Asano, D. Kirkpatrick, and C. Yap. Pseudo approximation algorithms, with applications to optimal motion planning. In *ACM Symp. on Computational Geometry*, volume 18, pages 170–178. ACM Press, 2002. Barcelona, Spain. To appear, Special Conference Issue of J.Discrete & Comp. Geom.
2. D. H. Bailey. Multiprecision translation and execution of Fortran programs. *ACM Trans. on Math. Software*, 19(3):288–319, 1993.
3. M. Benouamer, D. Michelucci, and B. Péroche. Boundary evaluation using a

lazy rational arithmetic. In *Proceedings of the 2nd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 115–126, Montréal, Canada, 1993. ACM Press.

4. E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *Proc. 10th European Symp. on Algorithms (ESA'02)*, pages 174–186. Springer, 2002. Lecture Notes in CS, No. 2461.
5. J. Blömer. *Simplifying Expressions Involving Radicals*. PhD thesis, Free University Berlin, Department of Mathematics, October, 1992.
6. L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1997.
7. A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems*. American Elsevier Publishing Company, Inc., New York, 1975.
8. R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. on Math. Software*, 4:57–70, 1978.
9. H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1-2):25–47, 2001.
10. H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. *Algorithmica*, 27:21–56, 2000.
11. P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity theory*. Series of Comprehensive Studies in Mathematics, Vol.315. Springer, Berlin, 1997.
12. C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.
13. C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *Lecture Notes in Computer Science*, pages 254–265. Springer, 2001. to appear, Algorithmica.
14. C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th ACM Symp. Comp. Geom.*, pages C18–C19, 1995.
15. F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.
16. T. Y. Chow. What is a closed-form number? *Amer. Math. Monthly*, 106(5):440–448, 1999.
17. B. M. Cullough. Assessing the reliability of statistical software: Part II. *The American Statistician*, 53:149–159, 1999.
18. M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schmer, R. Schulte, , and D. Weber. Certifying and repairing solutions to large lps, how good are lp-solvers? In *Proc. SODA 2003 (to appear)*, 2003.
19. Z. Du, M. Eleftheriou, J. Moreira, and C. Yap. Hypergeometric functions in exact geometric computation. In V.Brattka, M.Schoeder, and K.Weihrauch, editors, *Proc. 5th Workshop on Computability and Complexity in Analysis*, pages 55–66, 2002. Malaga, Spain, July 12-13,

2002. In Electronic Notes in Theoretical Computer Science, 66:1 (2002), http://www.elsevier.nl/locate/entcs/volume66.html. Also available as "Computability and Complexity in Analysis", Informatik Berichte No.294-6/2002, Fern University, Hagen, Germany.

20. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr. The CGAL kernel: a basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 191–202, Berlin, 1996. Springer. Lecture Notes in Computer Science No. 1148; Proc. 1st ACM Workshop on Applied Computational Geometry (WACG), Federated Computing Research Conference 1996, Philadelphia, USA.
21. S. J. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.
22. A. Frommer. Proving conjectures by use of interval arithmetic. In U. Kulisch, R. Lohner, and A. Facius, editors, *Perspectives on Enclosure Methods*. Springer-Verlag, Vienna, 2001.
23. S. Funke, K. Mehlhorn, and S. Näher. Structural filtering: A paradigm for efficient and exact geometric programs. In *Proc. 11th Canadian Conference on Computational Geometry*, 1999.
24. P. Gowland and D. Lester. A survey of exact arithmetic implementations. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.
25. N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.
26. Holt, Matthews, Rosselet, and Cordy. *The Turing Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
27. CGAL Homepage, 1998. Computational Geometry Algorithms Library (CGAL) Project. A 7-institution European Community effort. See URL http://www.cgal.org/.
28. LEDA Homepage, 1998. Library of Efficient Data Structures and Algorithms (LEDA) Project. From the Max Planck Institute of Computer Science. See URL http://www.mpi-sb.mpg.de/LEDA/.
29. T. Hull, M. Cohen, J. Sawchuk, and D. Wortman. Exception handling in scientific computing. *ACM Trans. on Math. Software*, 14(3):201–217, 1988.
30. V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric libraries. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.
31. K.-I. Ko. *Complexity Theory of Real Functions*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1991.
32. S. Krishnan, M. Foskey, T. Culver, J. Keyser, and D. Manocha. PRECISE: Efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation. *ACM Symp. on Computational Geometry*, 17:274–283, 2001.

33. U. Kulisch, R. Lohner, and A. Facius, editors. *Perspectives on Enclosure Methods*. Springer-Verlag, Vienna, 2001.
34. K. Lange. *Numerical Analysis for Statisticians*. Springer, New York, 1999.
35. K. Mehlhorn and S. Schirra. Exact computation with `leda_real` – theory and geometric application. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, volume 379, pages 163–172, Vienna, 2001. Springer-Verlag.
36. N. Metropolis. Methods of significance arithmetic. In D. A. H. Jacobs, editor, *The State of the Art in Numerical Analysis*, pages 179–192. Academic Press, London, 1977.
37. D. Michelucci and J.-M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9):961–975, 1997.
38. M. Mignotte. Identification of algebraic numbers. *J. of Algorithms*, 3:197–204, 1982.
39. R. E. Moore. *Interval Analysis*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J., 1966.
40. N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.
41. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, Cambridge, 1990.
42. K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997. Download from `http://cs.nyu.edu/exact/doc/`.
43. C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.
44. S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. In *Proc. 19th ACM Symp. on Comp. Geom.*, pages 256–263. ACM Press, 2003. San Diego, California.
45. D. Richardson. How to recognize zero. *J. of Symbolic Computation*, 24:627–645, 1997.
46. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
47. A. Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980.
48. J. Sellen, J. Choi, and C. Yap. Precision-sensitive Euclidean shortest path in 3-Space. *SIAM J. Computing*, 29(5):1577–1595, 2000. Also: 11th ACM Symp. on Comp. Geom., (1995)350–359.
49. The Institute of Electrical and Electronic Engineers, Inc. IEEE Standard 754-1985 for binary floating-point arithmetic, 1985. ANSI/IEEE Std 754-1985. Reprinted in SIGPLAN 22(2) pp. 9-25.
50. D. Tulone, C. Yap, and C. Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. In J. Richter-Gebert and D. Wang, editors, *Proc. 3rd Int'l. Workshop on Automated Deduction in Geometry (ADG'00)*, number 2061 in Lecture Notes in Artificial Intelligence,

pages 58–82. Springer, 2001. Zurich, Switzerland.
51. K. Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.
52. C. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Geometric Computing*, 1998. Invited Talk. Brown University, Oct 11–12, 1998.
Abstracts, `http://www.cs.brown.edu/cgc/cgc98/home.html`.
53. C. Yap, C. Li, and S. Pion. Core Library Tutorial: a library for robust geometric computation, 1999. Released with the Core Library software package, 1999–2003. Download: `http://cs.nyu.edu/exact/core/`.
54. C. K. Yap. Introduction to the theory of complexity classes, 1987. Book Manuscript. Preliminary version,
URL `ftp://cs.nyu.edu/pub/local/yap/complexity-bk`.
55. C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41. CRC Press LLC, Boca Raton, FL, 2nd edition (revised, expanded) edition, 2003, to appear.
56. C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, Singapore, 1995. 2nd edition.